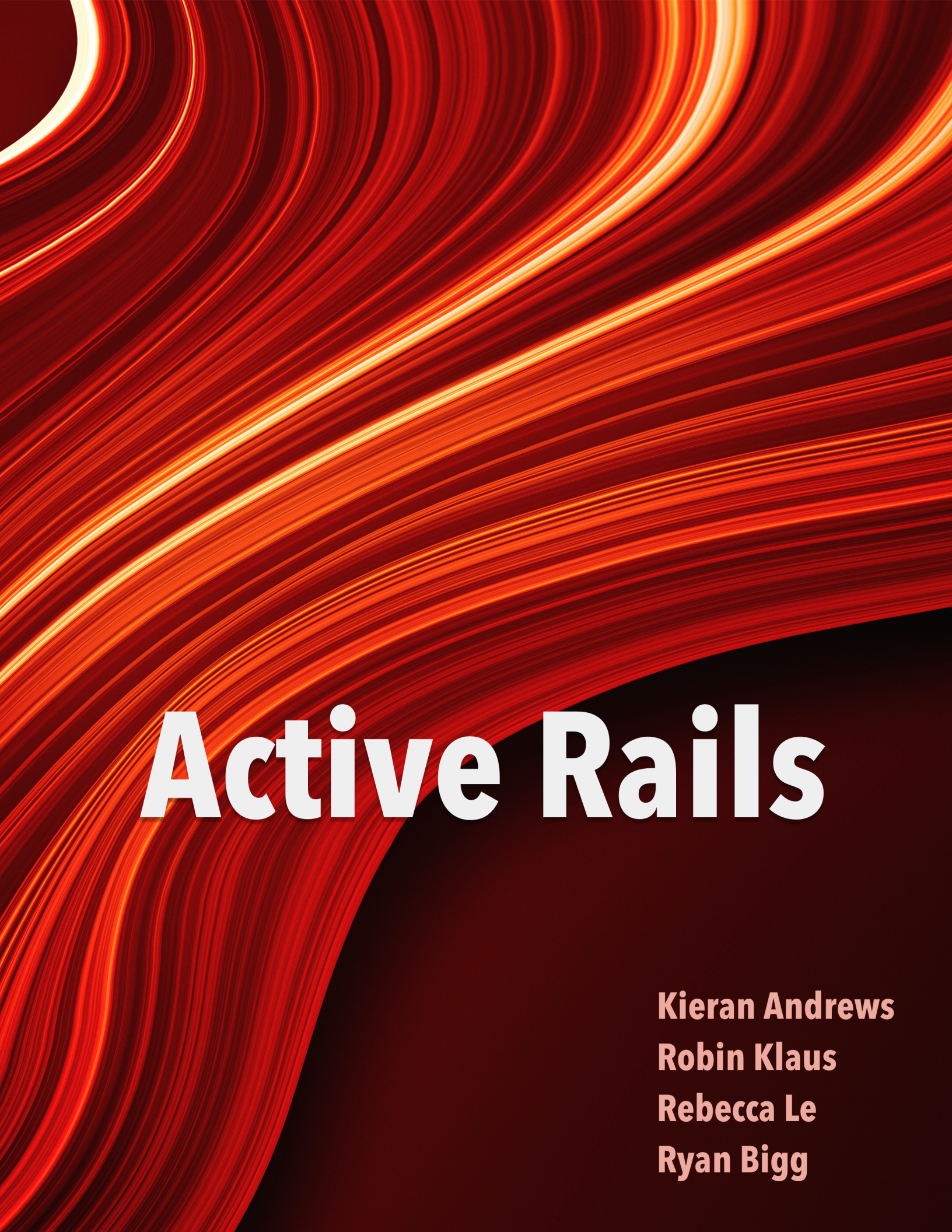

Active Rails
Kieran Andrews
Robin Klaus
Rebecca Le
Ryan Bigg

Active Rails

Table of Contents

Active Rails

This book is for sale at http://leanpub.com/rails.

This version was generated on 2021-01-24 from f0de2c495.

© 2011 - 2020 Ryan Bigg, Rebecca Skinner, Kieran Andrews, Robin Dunbar

Preface

This book is currently a work-in-progress beta book. We practically guarantee there will be misteaks.

If you see any mistakes in this book, please file an issue on our public GitHub issues: https://github.com/rubysherpas/active_rails_examples/issues/new. Please include this information:

- "This version was generated on 2021-01-24 from f0de2c495."
- If you're reading online: the URL of the page where you saw this mistake.
- If you're reading the PDF: The page number (in the bottom left or right)
- Some text of where you think the mistake is.

If you're unsure of if it's a mistake or not, then it's better to tell us anyway. This early in the book there has been one mistake already. **There will be more**. Beware.

If you find something else you want to tell us about, then use those issues too. Even if you want us to explain things better!

With your help, we can make this book as perfect as it can be.

Author Introductions

Ryan Bigg

I came to be an author on this book back in April 2010 and then spent about a year and a half writing it from scratch and working full time. The first edition (called Rails 3 in Action), focussing on Rails 3, was published in September of 2011. The second edition (called Rails 4 in Action), focussing on Rails 4.2 started in 2012 (or was it 2013?) and was published in August 2015.

Now nearly 5 years later, here is the third edition! It focusses on Rails 6. I started working on this edition in November 2019, and it's now April 2020 as I write this, and the book is almost done. Again.

I suppose I should answer why this one is called Active Rails and not Rails 6 in Action. We

didn't want to continue using the "in Action" name, as it sounded too much like "inaction". Active Rails is about vibrancy and energy, and is a nice pun on the naming conventions of the parts of Rails (Active Record, Active Support, etc.) to boot.

We even have a great cover to match, made from an idea Kieran had. No more weird soldier dudes on the cover that have nothing to do with Rails at all. I am glad that I won't get weird-soldier-dude questions any more.

We're now able to publish this book ourselves, free from the shackles of a publisher who… well, I won't say much more about that. Let's just leave it at "Ryan is over the moon happy", yeah?

This means that we are now able to publish this book on our own, through Leanpub, without the… let's call it "interference"… of a "traditional" publisher. This will (perhaps paradoxically) lead to a higher-quality book and it will mark the first edition that comes to you in beautiful color. Even the code is syntax highlighted.

It'll make reading a nicer experience overall.

Why even sign up to write a third edition when the first two took so long to write? For two very simple reasons: I keep hearing from people within the Ruby community who've read either (or both!) of the earlier versions who have read the book and who've become Rails developers since. Or even those who were Rails developers and found something valuable within the pages of the first two editions. Personally, I've wanted to do a third, self-published edition for a long time. Perhaps for 5 years! And so with those two things in mind, it just makes sense to do a third edition.

If you're one of those early-edition readers who's still buying, reading and recommending this book to others: thank you times infinity. I appreciate your support.

Rails continues to evolve, and this book represents this. Rails always evolves, and for the better. Between Rails 4 and Rails 6, I can think of a few cool features that've come through: Webpacker (for 1st-class JavaScript support), Action Cable, Action Text, Action Mailbox, Active Storage, and that's just off the top of my head. This book covers those features, and has been refreshed to keep it up-to-date with this latest evolution of Rails applications.

The earlier editions of this book have been used by many people to jumpstart their careers in Rails and you could be next with this third edition. Skimming through these pages won't get you there, but reading it thoroughly and applying the lessons in it might just get you there.

Good luck.

Ryan Bigg

Acknowledgements

TODO: These are usually written once the book is complete!

About this book

This book will teach you everything you need to know about the Ruby on Rails web framework.

Ruby on Rails is a leading web application framework built on top of the fantastic Ruby programming language. Both the language and the framework place an extreme emphasis on having a principle of least surprise and getting out of the way of the developers using it.

Ruby on Rails has been growing at a rapid pace, with large internet companies such as GitHub and Twitter using it for their core functionality. With the latest release of Rails, version 6.0, comes a set of changes that improve the already brilliant framework that has been constructed over the past sixteen years. The fantastic community around the framework has also been growing at a similar pace.

This book is designed to take you through developing a full-featured Rails application from step one, showing you exactly how professionals in the real world are developing applications right now.

Who should read this book

This book is primarily for those who are looking to work with the Ruby on Rails framework and who have some prior experience with Ruby, although that is not entirely necessary. The chapters get more and more advanced as you go along and provide a smooth learning curve in order to teach you how Rails applications are built.

If you're looking for a book that teaches you the same practices that are used in the real world, then this is the book you are looking for.

What's new in the third edition

Wow, sixteen years of Rails. That's a long time in software!

A third edition for a Rails book is not just a matter of fixing up typos, images and other things. It almost requires an entire rewrite of the whole thing. We've split up some chapters, removed others, and touched up the rest. Everything has been pored over and vetted by authors and volunteer reviewers, yet again.

We have spent hundreds of hours around updating this book, and all just for you. We hope you like it.

Roadmap

TODO: Update once ToC is finalized

Chapter 1 introduces the Ruby on Rails framework and begins to show how you can develop the beginnings of an application.

Chapter 2 shows off test-driven development and behaviour-driven development, which are two core concepts that'll be used throughout the remainder of this book and that can be applied instantly to any Ruby and Rails code you may write in the future. By testing the code you write, you can be assured that it's always working in the expected way.

Chapters 3 and 4 discuss the application you develop in this book - a project-management app of sorts - and delve into the core concepts of a Rails application. They also look at developing the first core features of your application.

Chapter 5 begins an introduction to nested resources, building on top of the features developed on the previous two chapters.

Chapter 6 shows you how to style that application using the wonderful Bootstrap design framework. With not-so-much-effort, we can get a decent looking design for our application very quickly.

Chapter 7 introduces authentication, and uses the Devise gem to implement features such as requiring users to sign in to the application before they can perform certain tasks.

Chapter 8 builds on the work in Chapter 7 by adding authorization to the application. There will be new areas of the application that are accessible only to a certain kind of user.

In Chapter 9 you learn about file uploading using the Active Storage gem. In this chapter you also learn about testing parts of your application that use JavaScript.

Chapter 10 builds not one but two new features for the application, adding the ability to comment on a ticket as well as track the ticket's lifecycle through varying states.

Chapter 11 begins our foray into dealing with email in a Rails application. You'll see how Rails makes it easy to send email using a part of its framework called ActionMailer.

In Chapter 12, you add a feature that lets users assign tags to tickets so they can be easily grouped. You also add a feature to allow users to search for tickets matching a certain state, tag or both.

Chapter 13 involves deploying the application to Heroku:https://heroku.com, a well-established hosting provider that offers a free service. This chapter also introduces Continuous Integration through GitHub Actions, which will run the tests for the application and deploy the application to Heroku if all the tests are passing.

Chapter 1. Ruby on Rails, the framework

Welcome aboard! It's great to have you with us on this journey through the world of Ruby on Rails. Ruby on Rails is known as a powerful web framework that helps developers rapidly build modern web applications. In particular, it provides lots of niceties to help you in your quest to develop a full-featured real-world application and be happy doing it. Happy developers are great developers!

If you're wondering who uses Rails, there are plenty of companies that do: Twitter, Netflix, and Urban Dictionary, just to name a few. This book will teach you how to build a very small and simple application in this first chapter, right after we go through a brief description of what Ruby on Rails actually is. Within the first couple of chapters, you'll have some solid foundations of an application, and you'll build on those throughout the rest of the book.

1.1. Ruby on Rails overview

Ruby on Rails is a framework built on the Ruby language—hence the name Ruby on Rails. The Ruby language was created back in 1993 by Yukihiro "Matz" Matsumoto of Japan, and was released to the general public in 1995. Since then, it has earned both a reputation and an enthusiastic following for its clean design, elegant syntax, and wide selection of tools available in the standard library and via a package-management system called RubyGems. It also has a worldwide community and many active contributors constantly improving the language and the ecosystem around it. We're not going to go into great depth about the Ruby language in this book though, because we'd rather talk about Ruby on Rails[1].

The foundation for Ruby on Rails was created during 2004 when David Heinemeier Hansson was developing an application called Basecamp. For his next project, the foundational code used for Basecamp was abstracted out into what we know as Ruby on Rails today. This project was released under a software license called the MIT License[2], and this permits anyone anywhere to use Ruby on Rails.

Since then, Ruby on Rails has quickly progressed to become one of the leading web development frameworks. This is in no small part due to the large community surrounding it, improving everything from documentation, to bug fixes, all the way up to adding new features to the framework.

This book is written for version 6.0 of the framework, which is the latest version of Rails. If

you've used Rails 4 or Rails 5, you'll find that much feels the same, yet Rails has learned some new tricks, as well.[3]

1.1.1. Benefits

Ruby on Rails allows for rapid development of applications by using a concept known as convention over configuration. A new Ruby on Rails application is created by running the application generator, which creates a standard directory structure and the files that act as a base for every Ruby on Rails application. These files and directories provide categorization for pieces of your code, such as the `app/models` directory for containing files that interact with the database and the `app/assets` directory for assets such as stylesheets, JavaScript files, and images. Because all this is already there, you won't be spending your time configuring the way your application is laid out. It's done for you.

One great example of rapid development of a Rails application is the 20-minute Rails tour video recorded by David Heinemeier Hansson.[4] This screencast takes you from having nothing at all to having a basic blogging and commenting system, in less than half an hour!

Once learned, Ruby on Rails affords you a level of productivity unheard of in other web frameworks, because every Ruby on Rails application starts out the same way - with exactly the same layout. The similarity between the applications is so close that the paradigm shift between different Rails applications isn't tremendous. If and when you jump between Rails applications, you don't have to relearn how it all connects together - it's mostly the same.

The Rails ecosystem may seem daunting at first, but Rails conventions allow even new things to seem familiar very quickly, smoothing the learning curve substantially.

Ruby Gems

The core features of Rails are split up into many different libraries, such as Active Record, Active Storage, Active Job, Active Support, Action Mailer, and Action Pack. These are called Ruby Gems, or gems for short.[5] These gems provide a wide range of methods and classes that help you develop your applications. They eliminate the need for you to perform boring, repetitive tasks—such as coding how your application hooks into your database—and let you get right down to writing valuable code for whatever it is you want to do.

Ever wished for a built-in way of automatically making sure your code works? The Ruby gem ecosystem has a gem called RSpec. When you use RSpec, you'll write automated test code for your application, as you'll see many, many times throughout this book. This test code will

make sure your code works continuously. Testing your code saves your bacon in the long term, and that's a fantastic thing.

In addition to testing frameworks, the Ruby community has produced many high-quality gems for use in your day-to-day development with Ruby on Rails. Usually, if you can think of it, there's a gem out there that will help you do it.

Noticing a common pattern yet? Probably. As you can see, Ruby on Rails (and the great community surrounding it) provides code that performs the trivial application tasks for you, from setting up the foundations of your application to handling the delivery of email. The time you save with all these libraries is immense! And because the code is open source, you don't have to go to a specific vendor to get support. Anyone who knows Ruby will help you if you're stuck. Just ask! We're a friendly bunch.

MVC

One of the other benefits of Rails is its adherence to a pattern called MVC, or Model-View-Controller.

The Model-View-Controller (MVC) pattern isn't unique to Ruby on Rails, but provides much of the core foundation for a Ruby on Rails application. This pattern is designed to keep the different parts of the application separate while providing a way for data to flow between them.

In applications that don't use MVC, the directory structure and how the different parts connect to each other is commonly left up to the original developer. Generally, this is a bad idea because different people have different opinions about where things should go. In Rails, a specific directory structure encourages developers to conform to the same layout, putting all the major parts of the application inside an `app` directory.

This `app` directory has three main subdirectories: `models`, `controllers`, and `views`.

- Models contain the domain logic of your application. This logic dictates how the records in your database are retrieved, validated, or manipulated. In Rails applications, models define the code that interacts with the database's tables to retrieve and set information in them. Domain logic also means things such as validations or particular actions to perform on the data.
- Controllers interact with the models to gather information to send to the view. They're the layer between the user and the database. They call methods on the model classes,

which can return single objects representing rows in the database or collections (arrays) of these objects. Controllers then make these objects available to the view through instance variables. Controllers are also used for permission checking, such as ensuring that only users who have special permission to perform certain actions can perform those actions, and users without that permission can't.

- Views display the information gathered by the controller, by referencing the instance variables set there, in a developer-friendly manner. In Ruby on Rails, this display is done by default with a templating language known as Embedded Ruby (ERB). ERB allows you to embed Ruby (hence the name) into any kind of file you wish. This template is then preprocessed on the server side into the output that's shown to the user.

1.1.2. Rails in the wild

One of the best-known sites that runs Ruby on Rails is GitHub, a hosting service for Git repositories. The site was launched in February 2008, and is now the leading Git web-hosting site. GitHub's massive growth was in part due to the Ruby on Rails community quickly adopting it as their de facto repository hosting site. Now GitHub is home to over 100 million repositories[6] for just about every programming language on the planet. It's not exclusive to programming languages, either; if it can go in a Git repository, it can go on GitHub. As a matter of fact, this book and its source code are kept on GitHub!

Other companies you might've heard of, like Airbnb, Stripe, Square and Twitter all use Rails too. It's used by some pretty big players!

But you don't have to build huge applications with Rails. There's a Rails application that was built for the specific purpose of allowing people to review the previous edition of this book, and it's just over 2,000 lines of code. This application allowed reviewers during the writing of the book to view the book's chapters and leave notes on each element, leading overall to a better book.

Now that you know what other people have accomplished with Ruby on Rails, let's dive into creating your own application.

1.2. Rails from first principles

This application that we will create now will give you a general overview of the main components of a Rails application. The components that we will cover will be the routes, controllers, models, and views.

This will be a simple application that you can use to track your purchases. We'll end up with a page that looks like this:

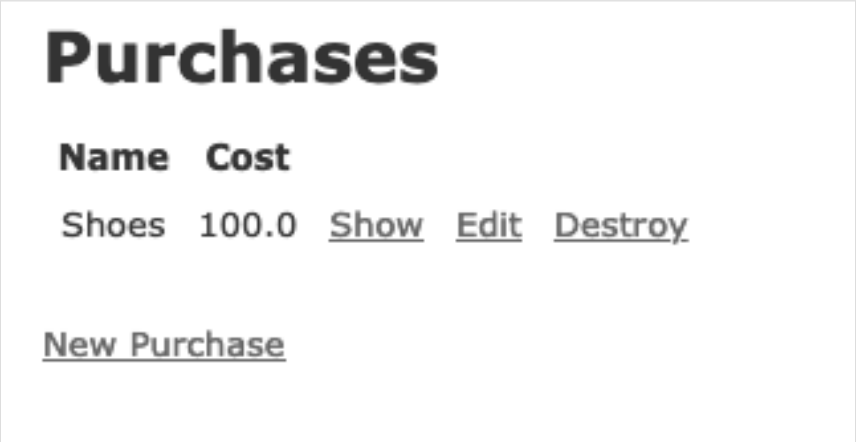


Figure 1. Purchases list

This application will allow us to list, create, update, and delete from a list of purchases.

This application is relatively straightforward and so it's perfect to give us our first taste of Rails.

1.2.1. Installing Rails

To get started, you must have these three things installed:

- Ruby
- RubyGems
- Rails

If you're on a UNIX-based system (Linux or Mac), we recommend that you use `ruby-install` [7] to install Ruby and RubyGems. For Windows, we recommend the RubyInstaller application[8]. There's a complete installation guide for Ruby and Rails, on Mac OS X, Linux, and Windows, in Appendix A of this book.

Before proceeding, let's check to be sure you have everything. Type these commands, and check out the responses:

```
$ ruby -v
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-darwin19]
$ gem -v
3.1.2
$ rails -v
Rails {rails_version}
```

If you see something that looks close to this, you're good to go! You might see `[x86_64-linux]` instead of `[x86_64-darwin19]`, or a slightly different patch (`p`) number, but that's okay. These particular values are the ones we're using right now and we've tested everything in the book against them; as long as you have Ruby 2.7, Rails 6.0, and RubyGems 3.0, everything should be fine.

Make sure you're setup!

If you don't get these answers, or you get some sort of error message, please be sure to complete this setup before you try to move on; you can't just ignore errors with this process! Certain gems (and Rails itself) only support particular versions of Ruby, so if you don't get this right, things won't work.

1.2.2. Generating an application

Remember before how we said that every Rails application starts out with the same layout? Well, we're about to see that for ourselves!

With Rails installed, to generate an application, you run the `rails` command and pass it the `new` argument and the name of the application you want to generate: `things_i_bought`. When you run this command, it creates a new directory called "things_i_bought", which is where all your application's code will go.

Don't use reserved words for application naming

You can call your application almost anything you wish, but it can't be given a name that is a reserved word in Ruby or Rails. For example, you wouldn't call your application `rails`, because the application class would be called `Rails`, and that would clash with the `Rails` constant within the framework. Names like `test` are also forbidden.

When you use an invalid application name, you'll see an error like one of these:

```
$ rails new rails
Invalid application name rails, constant Rails is already in use.
Please choose another application name.
$ rails new test
Invalid application name test. Please give a name which does not match one
of
the reserved rails words.
```

The application you're going to generate will be able to record purchases you've made. You can generate it using this command:

```
$ rails new things_i_bought
```

The output from this command may seem a bit overwhelming at first, but rest assured, it's for your own good. All the directories and files generated provide the building blocks for your application, and you'll get to know each of them as we progress. For now, let's get rolling and learn by doing, which is the best way.

1.2.3. Starting the application

To get the server running, you must first change into the newly created application's directory, and then start the application server:

```
$ cd things_i_bought
$ rails server
```

`rails server` (or `rails s`, for short) starts a web server on your local address on port 3000,

using a Ruby web server called Puma. It will say 'starting in development on http://localhost:3000', which indicates that the server will be available on port 3000 on the loopback network interface of this machine. To connect to this server, go to http://localhost:3000 in your favorite browser. You'll see the "Yay! You're on Rails" page:

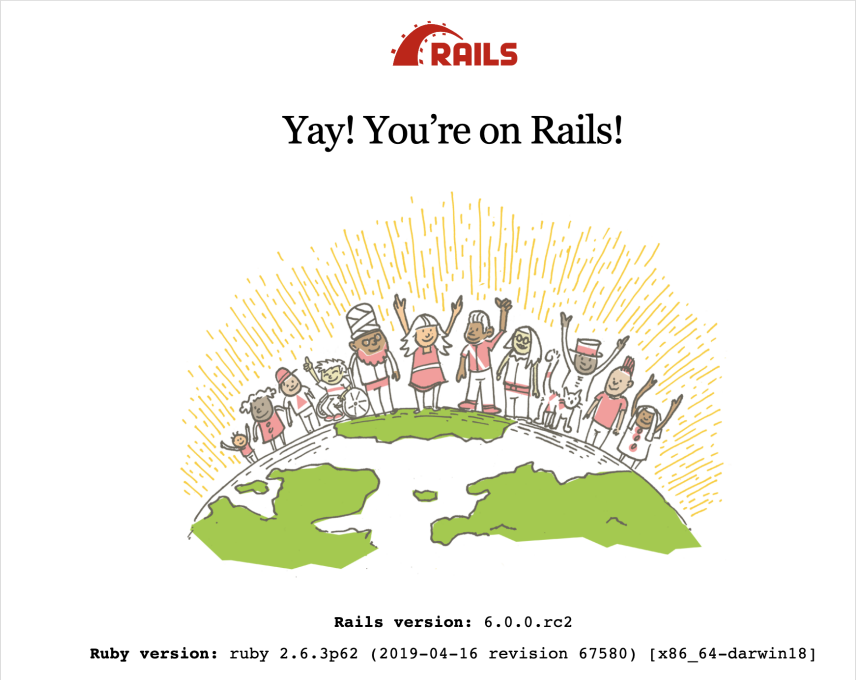


Figure 2. Yay, you're on Rails!

At the bottom of this page, you'll see two things: the version of Rails, and the version of Ruby that you're using. These should match the earlier version numbers that we mentioned earlier. That is, the Rails version should be {rails_version} and the Ruby version should start with "Ruby 2.7.1".

1.2.4. Getting started with Rails

The rest of this chapter is going to be a walk through for the different pieces of a Rails application. We're going to use a utility called the scaffold generator that will generate a lot of our code for us. After we've done that, we'll walk through all the individual pieces that this generator provides.

This should give you a fairly good overview of the parts that a Rails application is comprised of.

1.2.5. Scaffolding

To get started with this Rails application, you can generate a scaffold. Scaffolds in Rails provide a lot of basic functionality and are generally used as temporary structures to get started, rather than for full-scale development. In this case, it's a really good way to get started with exploring what Rails is capable of straight away.

Let's generate a scaffold by running this command:

```
$ rails generate scaffold purchase name:string cost:decimal
```

This generator will generate quite a lot:

```
invoke  active_record
create    db/migrate/20200227205209_create_purchases.rb
create    app/models/purchase.rb
invoke    test_unit
create      test/models/purchase_test.rb
create      test/fixtures/purchases.yml
invoke  resource_route
 route    resources :purchases
invoke  scaffold_controller
create    app/controllers/purchases_controller.rb
invoke    erb
create      app/views/purchases
create      app/views/purchases/index.html.erb
create      app/views/purchases/edit.html.erb
create      app/views/purchases/show.html.erb
create      app/views/purchases/new.html.erb
create      app/views/purchases/_form.html.erb
invoke    test_unit
create      test/controllers/purchases_controller_test.rb
create      test/system/purchases_test.rb
invoke    helper
create      app/helpers/purchases_helper.rb
invoke      test_unit
invoke    jbuilder
create      app/views/purchases/index.json.jbuilder
create      app/views/purchases/show.json.jbuilder
create      app/views/purchases/_purchase.json.jbuilder
invoke  assets
invoke    scss
create      app/assets/stylesheets/purchases.scss
invoke  scss
create    app/assets/stylesheets/scaffolds.scss
```

When you used the `rails` command earlier, it generated an entire Rails application. You can use this command inside an application to generate a specific part of the application by passing the `generate` argument to the `rails` command, followed by what it is you want to generate. You can also use `rails g` as a shortcut for `rails generate`.

The `scaffold` command generates everything that we will need for managing purchases within our application. We'll walk through the individual pieces shortly, but first let's talk about this command that was just used.

Everything after the name of the scaffold defines the fields for the database table, and the attributes for the objects of this scaffold. Here you tell Rails that the table for your `purchase` scaffold will contain `name` and `cost` fields, which are a string and a decimal, respectively.[9] To

create this table, the scaffold generator generates what's known as a migration. Let's look at what migrations are.

1.2.6. Migrations

When we're working within a Rails application, our data has to come from somewhere, and that somewhere is usually a database. In order to use this scaffold of ours properly, we're going to need some data. And that data is going to need to come from a database table. But that table doesn't exist yet! To create a database table to store our data in, we use what's known as a migration.

Migrations provide a way to implement incremental changes to the database's structure. Each migration is timestamped right down to the second, which provides you (and anybody else developing the application with you) an accurate timeline of your database's evolution. Let's open the only file in `db/migrate` now and see what it does. Its contents are shown in the following listing.

Listing 1. db/migrate/[date]_create_purchases.rb

```
class CreatePurchases < ActiveRecord::Migration[6.1]
  def change
    create_table :purchases do |t|
      t.string :name
      t.decimal :cost

      t.timestamps
    end
  end
end
```

Migrations are Ruby classes that inherit from `ActiveRecord::Migration`. Inside the class, one method is defined: the `change` method.

Inside the `change` method, you use database-agnostic commands to create a table. When we execute the code in this migration, it will create a table called `purchases` with a `name` column that's a string, a `cost` column that's a decimal, and two timestamp fields. These timestamp fields are called `created_at` and `updated_at` and are automatically set to the current time when a record is created or updated, respectively. This feature is built into a part of Rails called Active Record. If there are fields present with these names (or `created_on` and `updated_on`), they will be automatically updated when necessary.

Creating a purchases table

To run the migration and create the `purchases` table we can type this command into the console:

```
$ rails db:migrate
```

Because this is your first time running migrations in your Rails application, and because you're using a SQLite3 database—the default database for Rails applications—Rails first creates the database in a new file at `db/development.sqlite3` and then creates the `purchases` table inside it. When you run `rails db:migrate`, it doesn't just run the `change` method from the latest migration but runs any migration that hasn't yet been run, allowing you to run multiple migrations sequentially.

Your application is, by default, already set up to talk to this new database, so you don't need to change anything. If you ever wanted to roll back this migration, you'd use `rails db:rollback`, which rolls back the latest migration by running the `down` method of the migration (or reverses the steps taken in the `change` method, if possible.)[10]

Rails keeps track of the last migration that was run by storing it using this line in the `db/schema.rb` file:

Listing 2. db/schema.rb

```
ActiveRecord::Schema.define(version: [timestamp]) do
```

There is also a database table called `schema_migrations` that stores a record with the same version number.

This version should match the prefix of the migration you just created[11], and Rails uses this value to know what migration it's up to. The remaining content of this file shows the combined state of all the migrations to this point. This file can be used to restore the last known state of your database if you run the `rails db:schema:load` command.

With your database set up with a `purchases` table in it, let's look at how you can add data to this table by using the scaffold that you've just created.

1.2.7. Viewing purchases

Ensure that your Rails server is still running, or start a new one by running `rails s` or `rails server` again. Start your browser now, and go to http://localhost:3000/purchases. You'll see the scaffolded screen for purchases:

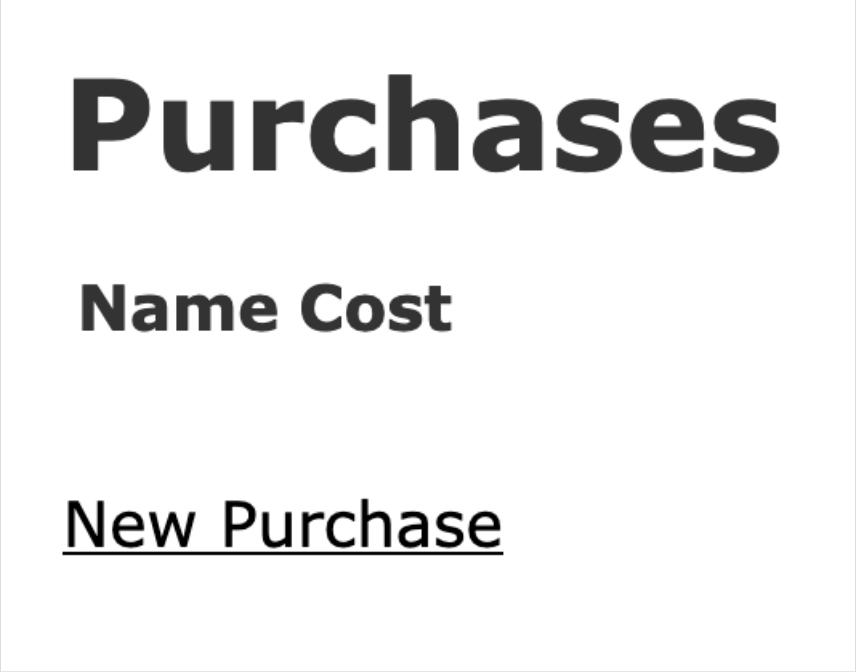


Figure 3. Purchases

Let's take a look at how this page came to be. How is it that we're able to go to http://localhost:3000/purchases and see something already? The answer to that lies in a file called `config/routes.rb`. The scaffold generator has added a line there:

Listing 3. config/routes.rb

```
Rails.application.routes.draw do
  resources :purchases
  # For details on the DSL available within this file, see
https://guides.rubyonrails.org/routing.html
end
```

This `resources :purchases` line is very simple, but it does quite a lot. This line generates routes for purchases actions that can be taken within this application. To see what those routes are, we can run this command:

```
$ rails routes
```

Here's the output from that command in a tabular form:

Prefix	Verb	URI Pattern	Controller#Action
purchases	GET	/purchases	purchases#index
	POST	/purchases	purchases#create
new_purchase	GET	/purchases/new	purchases#new
edit_purchase	GET	/purchases/:id/edit	purchases#edit
purchase	GET	/purchases/:id	purchases#show
	PATCH	/purchases/:id	purchases#update
	PUT	/purchases/:id	purchases#update
	DELETE	/purchases/:id	purchases#destroy

This one line in the `config/routes.rb` file has provided us with eight different routes. The one that we care about for the page we're looking at right now is the top route:

Prefix	Verb	URI Pattern	Controller#Action
purchases	GET	/purchases	purchases#index

When we make a request to `http://localhost:3000/purchases` in our browser, this will be a `GET` request. The route that matches this request in our Rails application is this top route, and it says that any `GET` request to `/purchases` will be routed to `purchases#index`.

This `purchases#index` syntax is a combination of two things, and it indicates the name of a controller and a name of an action.

A controller in a Rails application is a class that handles incoming requests and outgoing responses. In our application, the controller that serves this `GET /purchases` route is called `PurchasesController`, and it lives at `app/controllers/purchases_controller.rb`.

The action for this particular route is defined as a method inside this controller. When you hear "action" in a Rails application, you can think of it as being synonymous with "method defined inside a controller". Let's look at the method for the `index` action now:

Listing 4. app/controllers/purchases_controller.rb

```
# GET /purchases
# GET /purchases.json
def index
  @purchases = Purchase.all
end
```

This `index` method calls out to a class called `Purchase`. This is a class that was generated when we ran our scaffold generator, and it's defined in `app/models/purchase.rb`:

Listing 5. app/models/purchase.rb

```
class Purchase < ApplicationRecord
end
```

This class inherits from another class called `ApplicationRecord`, which was generated when we created our application:

Listing 6. app/models/application_record.rb

```
class ApplicationRecord < ActiveRecord::Base
  self.abstract_class = true
end
```

This class inherits from `ActiveRecord::Base`, and it's that class that defines the `all` method we're using in the `index` action of the `PurchasesController`.

This `Purchase.all` call will run a query on our database to find all purchases that currently exist. There aren't any right now, which is why we're seeing a nearly blank page!

When the controller calls `Purchase.all`, it assigns the result of that call to `@purchases`, an instance variable.

The next thing that is used is called a view. A view is a template that is used to show information collected from our controllers. By default, actions within controllers will always render views that match the action's name. The view that will be used to show the information collected by this `index` action is a file located at `app/views/purchases/index.html.erb`:

Listing 7. app/views/purchases/index.html.erb

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

<h1>Purchases</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Cost</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @purchases.each do |purchase| %>
      <tr>
        <td><%= purchase.name %></td>
        <td><%= purchase.cost %></td>
        <td><%= link_to 'Show', purchase %></td>
        <td><%= link_to 'Edit', edit_purchase_path(purchase) %></td>
        <td><%= link_to 'Destroy', purchase, method: :delete, data: { confirm: 'Are you
sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Purchase', new_purchase_path %>
```

This view is written mostly in HTML, but you'll notice a few extra things about it too. There's two special kinds of tags here, called ERB tags. There's `<% %>` and `<%= %>`. This kind of view is often referred to as an HTML+ERB view, as it is using another templating language called ERB along with HTML.

Inside `<%= %>` ERB tags, we evaluate Ruby code. This code runs, and whatever the Ruby code

outputs will be displayed on the page. When we use an `<%` tag, the Ruby code is only evaluated, and the return value of that code is not put onto the page.

This view will be processed by Rails when it serves the request for `GET /purchases`, the Ruby code will be executed and the result will be pure HTML. Here is what the view is currently outputting:

```
<h1>Purchases</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Cost</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
  </tbody>
</table>

<br>

<a href="/purchases/new">New Purchase</a>
```

So we have now seen how this page is built by Rails:

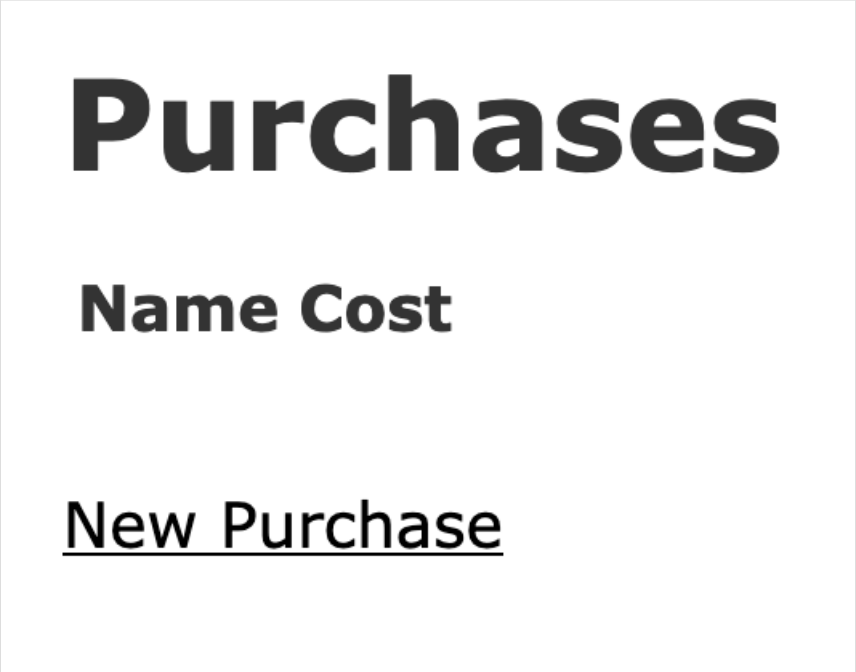
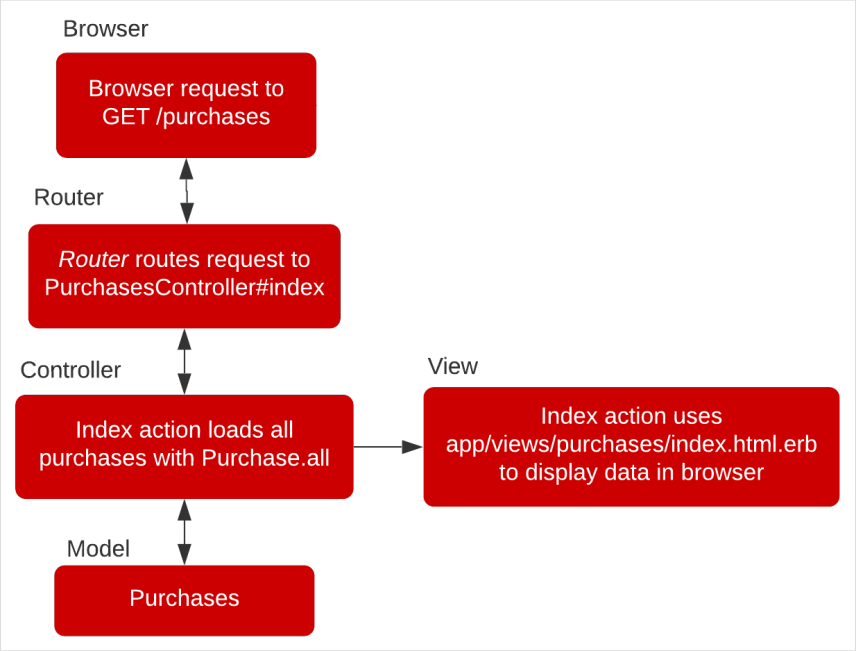


Figure 4. Purchases

The way that this content gets on the page is:

1. The browser makes a request to `GET /purchases`
2. The router routes the request to the `index` action inside `PurchasesController`
3. The `index` action attempts to load all of the purchases by calling `Purchase.all`, where `Purchase` is the model.
4. The `index` action then automatically uses the view located at `app/views/purchases/index.html.erb` to display a HTML representation of the data we've collected.

Here's a flowchart to help you understand what's happening:



We've now seen our first example of the big five main components of a Rails application:

- Routes — tell requests where to go when our application receives a request from the browser
- Actions — serve requests by collecting data from a model
- Controllers — a class that groups together related actions
- Models — a class that is used to interact with our application's database
- Views — displays the data that is gathered by the controller

We'll see these five things feature prominently in this book, as they really are the five pillars that Rails is built upon.

Three of these, the models, views and controllers are critical parts of the Model-View-Controller pattern we mentioned earlier in this chapter. This pattern is commonly used by applications (not just Rails ones!) in the programming world, as it is a nice way of organizing and separating different responsibilities within an application.

1.2.8. Creating a purchase

Now that we have seen how to list purchases, let's look at another feature of our application that scaffolding has given us automatically: the ability to create a new purchase.

Let's add a new purchase by going to http://localhost:3000/purchases and then clicking "New Purchase".

On this form, you will see two inputs for the fields you generated when you ran the scaffold generator:

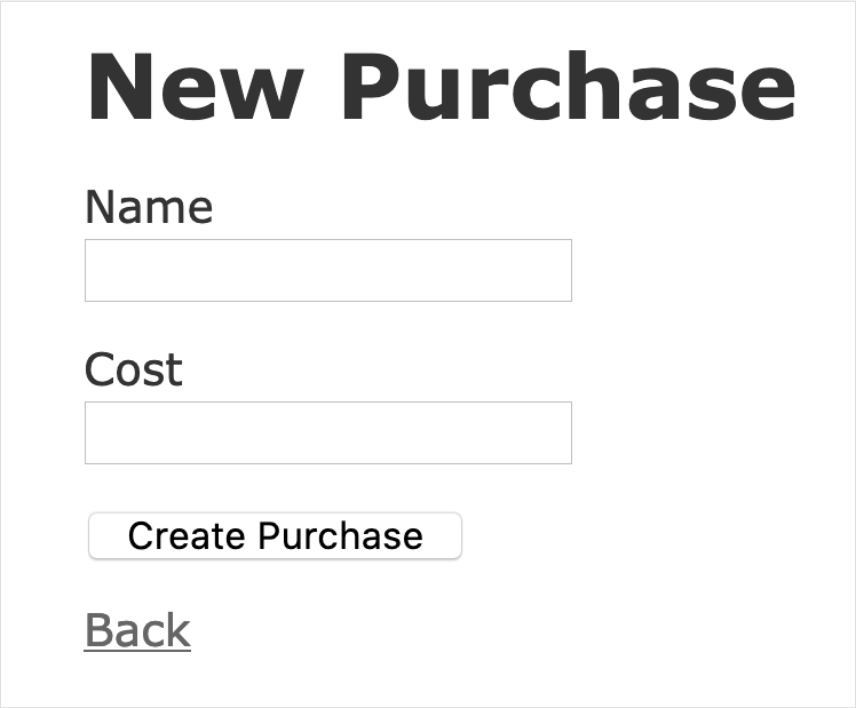


Figure 5. A new purchase

You'll notice in the URL bar in your browser that the route for this request is localhost:3000/purchases/new.

If we look at the `rails routes` output again, the route that matches this path is this one:

Prefix	Verb	URI Pattern	Controller#Action
new_purchase	GET	/purchases/new	purchases#new

This route sends requests for `GET /purchases/new` to the `new` action within the `PurchasesController`. Let's look at this action now.

Listing 8. app/controllers/purchases_controller.rb

```
# GET /purchases/new
def new
  @purchase = Purchase.new
end
```

This action initializes a new instance of the `Purchase` model, just so we can use it in the view. What you see on the page comes from the view located at `app/views/purchases/new.html.erb`, and it looks like the following listing.

Listing 9. app/views/purchases/new.html.erb

```
<h1>New Purchase</h1>

<%= render 'form' %>

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

This is an ERB file just like `index.html.erb`, . The two ERB `<%=` tags here will evaluate some Ruby and output values. Let's first talk about the `render` method.

The `render` method, when passed a string as in this example, tells Rails to use a partial. A partial is a separate template file that you can include into other templates to repeat similar code. Let's go through this file now.

The first half of the form partial

This particular partial is at `app/views/purchases/_form.html.erb`, and the first half of it looks like the following listing.

Listing 10. The first half of app/views/purchases/_form.html.erb

```
<%= form_with(model: purchase, local: true) do |form| %>
  <% if @purchase.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@purchase.errors.count, "error") %> prohibited
        this purchase from being saved:</h2>

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

This half is responsible for defining the form by using the `form_with` helper. The `form_with` method is passed the `model` option with the `purchase` local variable and from that it generates a form. The `local` option tells the form we want to use a "local" request to when submitting the form, as opposed to a "remote" request, which would submit the form in the background.

```
This `purchase` variable comes from the `new` action of `PurchasesController`, which we saw
before, but let's look at it again:
```

Listing 11. The `new` action of `PurchasesController`

```
def new
  @purchase = Purchase.new
end
```

The first line in this action sets up a new `@purchase` variable by calling the `new` method on the `Purchase` model. This initializes a new instance of the `Purchase` class, but doesn't create a new record in the database. The `@purchase` variable is then automatically passed through to the `new.html.erb` view by Rails.

When it gets there, the `@purchase` variable is then passed to the form partial with this line:

Listing 12. app/views/purchases/new.html.erb

```
<%= render 'form', purchase: @purchase %>
```

This then makes the `@purchase` instance variable available as that `purchase` local variable in `app/views/purchases/_form.html.erb`.

So far, all this functionality is provided by Rails. You've coded nothing yourself. With the `scaffold` generator, you get an awful lot for free.

Going back to the `app/views/purchases/_form.html.erb` partial, the block for the `form_with` is defined between its `do` and the `<% end %>` at the end of the file. Inside this block, you check the `@purchase` object for any errors by using the `@purchase.errors.any?` method. These errors will come from the model if the object didn't pass the validation requirements set in the model. If any errors exist, they're displayed by the content inside this `if` statement. Validation is a concept that we will be covering shortly.

The second half of the form partial

The second half of this partial looks like the following listing.

Listing 13. The second half of app/views/purchases/_form.html.erb

```
...
  <div class="field">
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div class="field">
    <%= form.label :cost %>
    <%= form.text_field :cost %>
  </div>

  <div class="actions">
    <%= form.submit %>
  </div>
```

Here, the `form` object from the `form_with` block is used to define labels and fields for your form. At the end of this partial, the `submit` method provides a dynamic submit button; the text on this button when it is displayed in the browser says "Create Purchase", even though we didn't put that in this partial.

Let's fill in this form now and click the submit button. You will see this:

Figure 6. Your first purchase

What you see is the result of your submission: a successful creation of a `Purchase`. Let's see how it got there.

When the `new.html.erb` view is processed by Rails, it will output this HTML code:

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

When this form is submitted, it will make a request to `POST /purchases`. The route that matches this request is this one:

Prefix	Verb	URI Pattern	Controller#Action
	POST	/purchases	purchases#create

This submit button posts the data from the form to the `create` action within `PurchasesController`, which looks like this.

Listing 14. The `create` action of PurchasesController

```
def create
  @purchase = Purchase.new(purchase_params)

  respond_to do |format|
    if @purchase.save
      format.html { redirect_to @purchase, notice: 'Purchase was successfully created.' }
      format.json { render :show, status: :created, location: @purchase }
    else
      format.html { render :new }
      format.json { render json: @purchase.errors, status: :unprocessable_entity }
    end
  end
end
```

Here, you use the same `Purchase.new` method you first saw used in the `new` action. But this time you pass it an argument of `purchase_params`, which is actually another method in this controller.

```
def purchase_params
  params.require(:purchase).permit(:name, :cost)
end
```

That method calls `params` (short for parameters), which is a method that returns the data sent from your form in a `Hash`-like object. We'll talk more about why you need this little dance later (in Chapter 3); this is a feature called strong parameters. When you pass this `params` hash into `new`, Rails sets the attributes [12] to the values from the form.

Inside `respond_to` is an `if` statement that calls `@purchase.save`. This method validates the record; and if it's valid, the method saves the record to the database and returns `true`.

If the return value is `true`, the action responds by redirecting to the new `@purchase` object using the `redirect_to` method, which takes either a path or an object that it turns into a path (as seen in this example). The `redirect_to` method inspects the `@purchase` object, and determines that the path required is `purchase_path` because it's an instance of the `Purchase` model. This path takes you to the `show` action for this controller. The `:notice` option passed to `redirect_to` sets up a flash message. A flash message is a message that can be displayed on the next request. This is the green text at the top of the above image.

You've seen what happens when the purchase is valid, but what happens when it's invalid?

Well, it uses the `render` method to show the `new` template again. We should note here that this doesn't call the `new` action again[13], it only renders the template.

You can make the creation of the `@purchase` object fail by adding a validation. Let's do that now.

Validations

You can add validations to your model to ensure that the data conforms to certain rules, or that data for a certain field must be present, or that a number you enter must be greater than a certain other number. You're going to write your first code for this application and implement both of these things now.

Open your `Purchase` model, and change the entire file to what's shown here:

Listing 15. app/models/purchase.rb

```
class Purchase < ActiveRecord::Base
  validates :name, presence: true
  validates :cost, numericality: { greater_than: 0 }
end
```

You use the `validates` method to define a validation that does what it says on the box: validates that the field is present. The other validation option, `:numericality`, validates that the `cost` attribute is a number and then, with the `:greater_than` option, validates that it's greater than 0.

Let's test these validations by going back to http://localhost:3000/purchases, clicking "New Purchase", and clicking "Create Purchase". You should see the errors shown here:

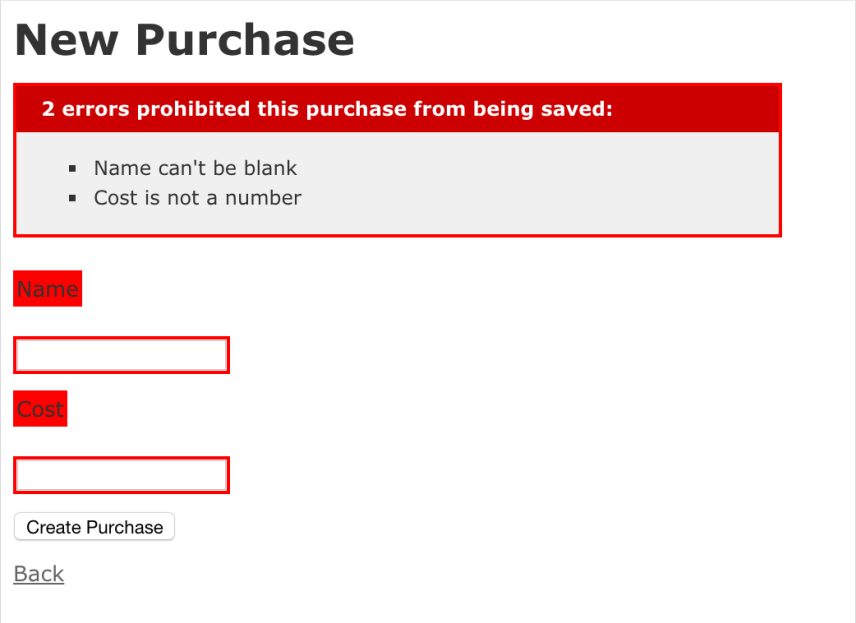


Figure 7. Errors on purchase

Great! Here, you're told that name can't be blank and that the value you entered for cost isn't a number. Let's see what happens if you enter "foo" for the Name field and "-100" for the Cost field, and click "Create Purchase". You should get a different error for the Cost field now:

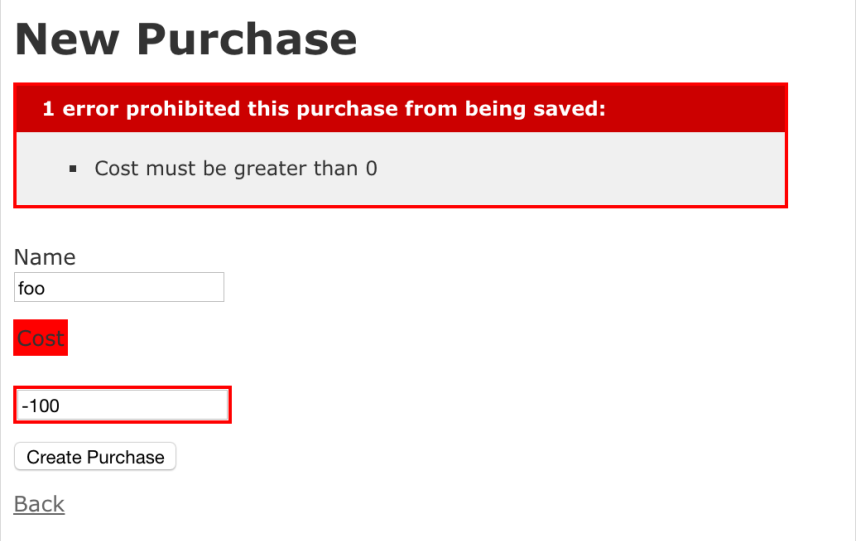


Figure 8. Cost must be greater than 0.

Good to see! Both of your validations are working. When you change Cost to `100` and click "Create Purchase", the value should be considered valid by the validations and take you to the `show` action. Let's look at what this particular action does now.

1.2.9. Showing off

After creating a purchase, we arrive at a page that displays the information of the purchase we have just created:

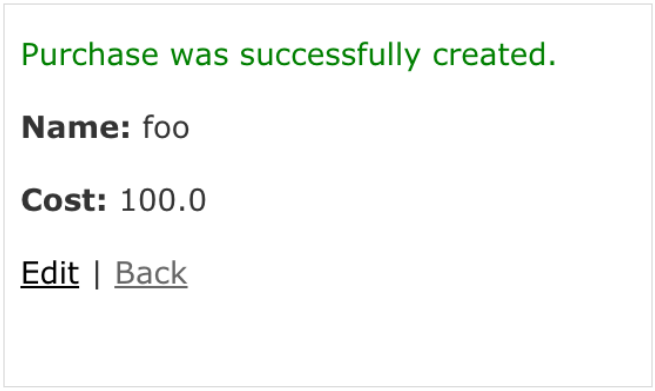


Figure 9. A single purchase

The URL for this page is http://localhost:3000/purchases/2. The number at the end of the URL is the unique numerical ID for this purchase. The route that matches this path is this one:

Prefix	Verb	URI Pattern	Controller#Action
purchase	GET	/purchases/:id	purchases#show

This path goes to the `show` action in `PurchasesController`. Let's look at that action now:

Listing 16. app/controllers/purchases_controller.rb

```
def show
end
```

This action is blank, but Rails will still render the related view. Let's look at that view for the `show` action:

Listing 17. app/views/purchases/show.html.erb

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

<p>
  <strong>Name:</strong>
  <%= @purchase.name %>
</p>

<p>
  <strong>Cost:</strong>
  <%= @purchase.cost %>
</p>

<%= link_to 'Edit', edit_purchase_path(@purchase) %> |
<%= link_to 'Back', purchases_path %>
```

On the first line is the `notice` method, which displays the `notice` set on the `redirect_to` from the `create` action. After that, field values are displayed in `p` tags by calling them as methods on your `@purchase` object. You might expect the `@purchase` object to be defined in the `show` action of `PurchasesController`:

Listing 18. The `show` action of `PurchasesController`

```
def show
end
```

But it is not! Instead, this `@purchase` object is defined by a `before_action` in this controller, at the very top of the class:

Listing 19. The `set_purchase` `before_action` in `PurchasesController`

```
class PurchasesController < ApplicationController
  before_action :set_purchase, only: [:show, :edit, :update, :destroy]

  ...

  # Use callbacks to share common setup or constraints between actions.
  def set_purchase
    @purchase = Purchase.find(params[:id])
  end

  ...
end
```

A `before_action` method will be executed before every action specified in the `only` option: hence the name `before_action`. The `find` method of the `Purchase` class is used to find the record with the ID of `params[:id]` and instantiate a new `Purchase` object from it (`params[:id]` being the number on the end of the URL). We can know that the parameter will be called `:id` if we look at the URL in our routes again:

Prefix	Verb	URI Pattern	Controller#Action
purchase	GET	/purchases/:id	purchases#show

Where we have `:id` in the routes, whatever value is there in the path will be made available as `params[:id]` in a controller.

Going back to the view (`app/views/purchases/show.html.erb`), at the end of this file is `link_to`, which generates a link using the first argument as the text value, and the second argument as the `href` for that URL. The second argument for `link_to` is a method: `edit_purchase_path`. This `edit_purchase_path` method is provided by the `resources :purchases` line in `config/routes.rb`, and we would call it a routing helper method.

Let's look at our routing table one more time. Particularly, let's look at the line that shows us the edit route:

Prefix	Verb	URI Pattern	Controller#Action
edit_purchase	GET	/purchases/:id/edit	purchases#edit

The "Prefix" column in this table shows us the routing helper prefix. This "Prefix" column indicates the prefix that we will have for routing helper methods. Rails will generate one for us called `edit_purchase_path` and one called `edit_purchase_url`. The `_path` methods provided by routes will return short paths like `/purchases/2/edit`, whereas the `_url` methods will return the full URL: `http://localhost:3000/purchases/2/edit`. We only need the path here, so that's what scaffold using: `edit_purchase_path`.

1.2.10. Updating purchases

Let's change the cost of the foo purchase now. Perhaps it only cost 10. To change it, go back to

http://localhost:3000/purchases and click the Edit link next to the foo record. This will go to the route provided by `edit_purchase_path` route, which will be `/purchases/2/edit`. In our browser, we'll see the route being http://localhost:3000/purchases/2/edit.

On this page, you should see a page that looks similar to the "New Purchase" page:

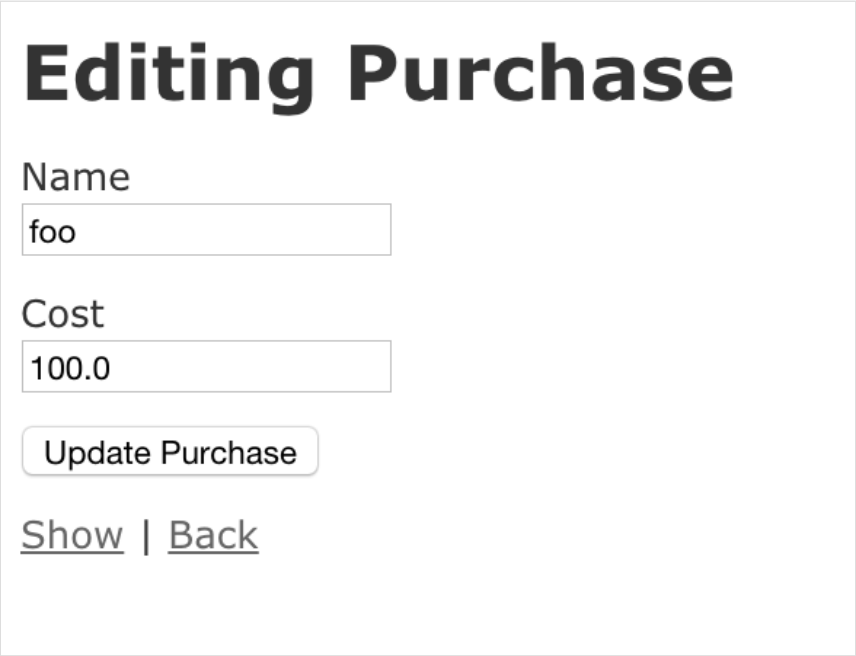


Figure 10. Editing a purchase

This page looks similar because it reuses the `app/views/purchases/_form.html.erb` partial that was also used in the template for the `new` action. Such is the power of partials in Rails: you can use the same code for two different requests to your application. The template for the `edit` action this action is shown in the following listing.

Listing 20. app/views/purchases/edit.html.erb

```
<h1>Editing Purchase</h1>

<%= render 'form' %>

<%= link_to 'Show', @purchase %> |
<%= link_to 'Back', purchases_path %>
```

For this action, you're working with a pre-existing object rather than a new object, which you used in the `new` action. This preexisting object is found by the `edit` action in `PurchasesController`, as shown here.

Listing 21. The `edit` action of `PurchasesController`

```
def edit
end
```

Oops: it's not here! The code to find the `@purchase` object is identical to what you saw earlier in the `show` action: it's set in `before_action` which runs before the `show`, `edit`, `update` and `destroy` actions.

```
def set_purchase
  @purchase = Purchase.find(params[:id])
end
```

Back in the view for a moment, at the bottom of it you can see two uses of `link_to`. The first creates a "Show" link, linking to the `@purchase` object, which is set up in the `edit` action of your controller. Clicking this link would take you to `purchase_path(@purchase)` or `/purchases/:id`. Rails will figure out where the link needs to go according to the class of the object given. Using this syntax, it will attempt to call the `purchase_path` method because the object has a class of `Purchase`, and it will pass the object along to that call, generating the URL. [14]

The second use of `link_to` in this view generates a "Back" link, which uses the routing helper `purchases_path`. It can't use an object here because it doesn't make sense to; calling `purchases_path` is the easy way to go back to the `index` action.

Let's try filling in this form—for example, by changing the cost from 100 to 10 and clicking "Update Purchase". You now see the `show` page but with a different message:

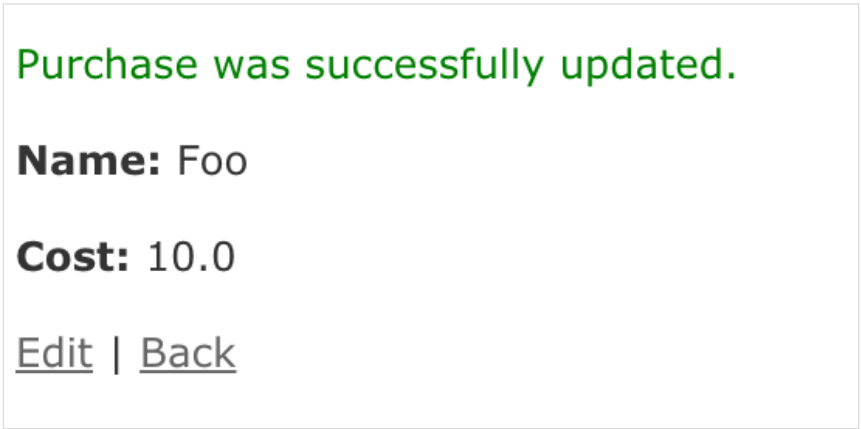


Figure 11. Viewing an updated purchase

Clicking "Update Purchase" brought you back to the `show` page. How did that happen? Click the back button on your browser, and view the source of this page, specifically the `form` tag and the tags directly underneath, shown in the following listing.

Listing 22. Rendered HTML for app/views/purchases/edit.html.erb

```
...
<form accept-charset="UTF-8" action="/purchases/2" method="post">
  <input name="_method" type="hidden" value="patch" />
...
```

The `action` of this `form` points at `/purchases/2`, which is the route to the `show` action in `PurchasesController`. You should also note two other things. The `method` attribute of this form is a `post`, but there's also the `input` tag underneath.

The `input` tag passes through the `_method` parameter with the value set to `patch`. Rails catches this parameter and turns the request from a `POST` into a `PATCH`.[15] This is the second (of three) ways `/purchases/:id` responds according to the method. The relevant line from `rails routes` is this line:

Prefix	Verb	URI Pattern	Controller#Action
	PATCH	/purchases/:id	purchases#update

By making a `PATCH /purchases/:id` request, the request is routed to the `update` action in `PurchasesController`.

Let's look at this next.

Listing 23. The `update` action of `PurchasesController`

```
def update
  respond_to do |format|
    if @purchase.update(purchase_params)
      format.html { redirect_to @purchase, notice: 'Purchase was
        successfully updated.' }
      format.json { render :show, status: :ok, location: @purchase }
    else
      format.html { render :edit }
      format.json { render json: @purchase.errors, status:
        :unprocessable_entity }
    end
  end
end
```

Just as in the `show` and `edit` actions, the `@purchase` object is first fetched by the call to `before_action :set_purchase`.

The parameters from the form are sent through in the same fashion as they were in the `create` action, coming through as `purchase_params`. Rather than instantiating a new object by using the `new` class method, you use `update` on the existing `@purchase` object. This does what it says: updates the attributes. What it doesn't say, though, is that it validates the attributes and, if the attributes are valid, saves the record and returns `true`. If they aren't valid, it returns `false`.

When `update` returns `true`, you're redirected back to the `show` action for this particular purchase by using `redirect_to`. If the `update` call returns `false`, you're shown the `edit` action's template again, just as back in the `create` action where you were shown the `new` template again. This works in the same fashion and displays errors if you enter something wrong. Let's try editing a purchase, setting Name to blank, and then clicking "Update Purchase". It should error exactly like the `create` method did:

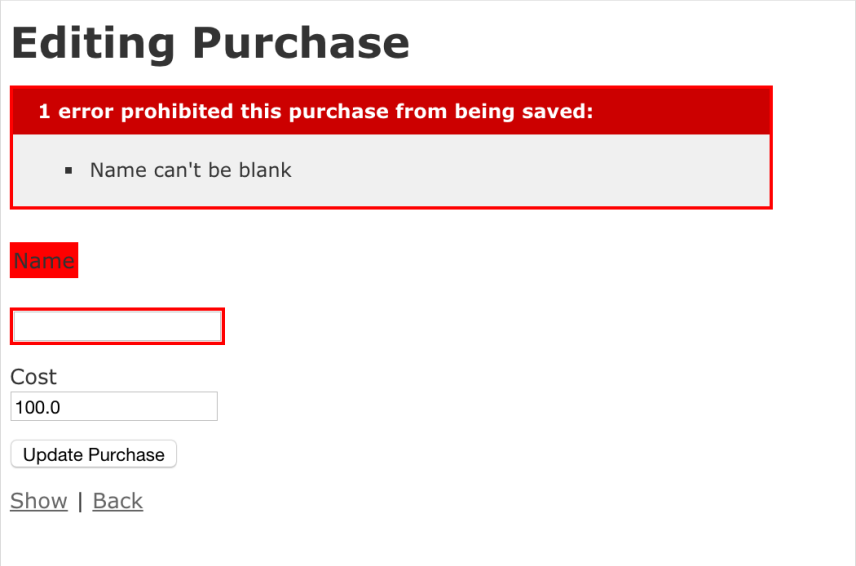


Figure 12. Update fails!

As you can see by this example, the validations you defined in your `Purchase` model take effect automatically for both the creation and updating of records.

What would happen if, rather than updating a purchase, you wanted to delete it? That's built in to the scaffold, too!

1.2.11. Deleting

In Rails, delete is given a much more forceful name: destroy. This is another sensible name, because to destroy a record is to 'put an end to the existence of'.[16] Once this record's gone, it's gone, baby, gone.

You can destroy a record by going to http://localhost:3000/purchases, clicking the "Destroy" link shown below: and then clicking "OK" in the confirmation box that pops up.

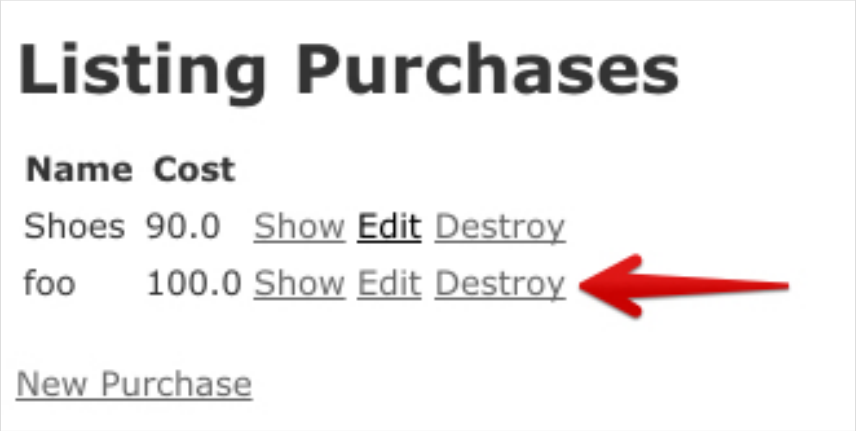


Figure 13. Destroy!

When that record's destroyed, you're taken back to the Listing Purchases page. You'll see that the record no longer exists. You should now have only one record:

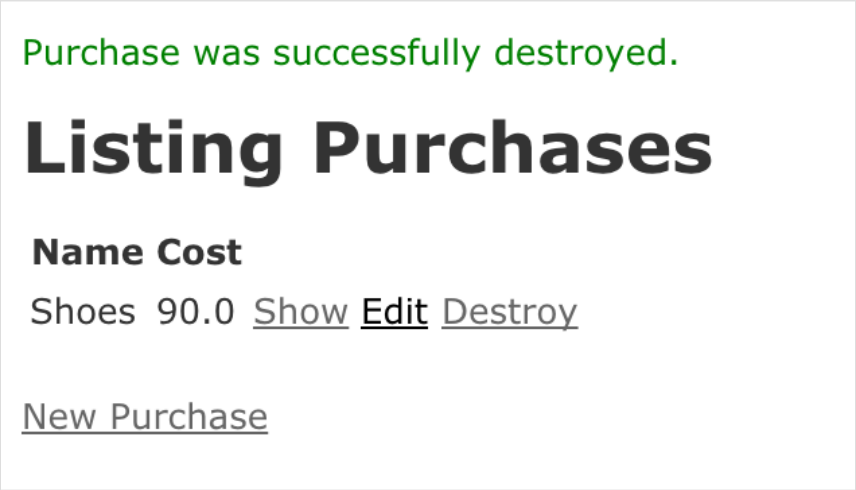


Figure 14. Last record standing

How does all this work? Let's look at the `index` template in the following listing to understand, specifically the part that's used to list the purchases.

Listing 24. app/views/purchases/index.html.erb

```
<% @purchases.each do |purchase| %>
  <tr>
    <td><%= purchase.name %></td>
    <td><%= purchase.cost %></td>
    <td><%= link_to 'Show', purchase %></td>
    <td><%= link_to 'Edit', edit_purchase_path(purchase) %></td>
    <td><%= link_to 'Destroy', purchase, method: :delete, data:
      { confirm: 'Are you sure?' } %></td>
  </tr>
<% end %>
```

In this template, `@purchases` is a collection of all the objects from the `Purchase` model, and `each` is used to iterate over each, setting `purchase` as the variable used in this block.

The methods `name` and `cost` are the same methods used in `app/views/purchases/show.html.erb` to display the values for the fields. After these, you see the three uses of `link_to`.

The first `link_to` passes in the `purchase` object, which links to the `show` action of `PurchasesController` by using a route such as `/purchases/:id`, where `:id` is the ID for this `purchase` object.

The second `link_to` links to the `edit` action using `edit_purchase_path` and passes the `purchase` object as the argument to this method. This routing helper determines that the path is `/purchases/:id/edit`.

The third `link_to` links seemingly to the `purchase` object exactly like the first, but it doesn't go to the `show` action. The `:method` option on the end of this route specifies the method `:delete`, which is the third and final way the `/purchases/:id` route can be used. If you specify `:delete` as the method of this `link_to`, Rails interprets this request as a `DELETE` request and takes you to the `destroy` action in the `PurchasesController`.

Prefix	Verb	URI Pattern	Controller#Action
	DELETE	/purchases/:id	purchases#destroy

A `DELETE /purchases/:id` route goes to the `destroy` action of `PurchasesController`. Let's look at this action now:

Listing 25. The `destroy` action of `PurchasesController`

```
def destroy
  @purchase.destroy
  respond_to do |format|
    format.html { redirect_to purchases_url, notice: 'Purchase was
      successfully destroyed.' }
    format.json { head :no_content }
  end
end
```

This action destroys the record loaded by `before_action :set_purchase` by calling `destroy` on it, which permanently deletes the record. Then it uses `redirect_to` to take you to `purchases_url`, which is the route helper defined to take you to http://localhost:3000/purchases. Note that this action uses the `purchases_url` method rather than `purchases_path`, which generate a full URL back to the purchases listing.

That wraps up our application run-through!

1.3. Takeaways

1.3.1. Rails is a conventions-based framework

Rails follows strict conventions, such as MVC (Model View Controller). With these conventions, every Rails application is built in much the same way, which allows developers to be able to work on any Rails application with ease.

This also means that we're not going to have to make decisions about the architecture of our Rails application. Rails has made those decisions for us already, and so we can get started on building our application right away.

1.3.2. The five main pillars of a Rails application

The five main pillars of a Rails application are:

- Routes
- Controllers
- Actions

- Models
- Views

Routes tell our application what controller and what action should serve our request. Actions that work on the same resource (a purchase in our application) are grouped within a single controller. To put it simply: routes are a way to match a request's path to a controller and an action within the application.

Controller actions use a model to interact with the database. This can be anything from creating new purchases, to finding all purchases, to updating purchases, to deleting purchases. Any actions that we take on our purchase data goes through our model first.

Finally, the views are used to combine HTML and Ruby to produce dynamic HTML content which is then served to our browser. Without these, we wouldn't be able to see the data that the controller was fetching in the first place!

1.3.3. The scaffold generator gives us a lot

The scaffold generator provides us with a total of seven different actions we can do on a particular resource:

- `index` - Show a list of purchases
- `show` - View a single purchase
- `new` - Display a form for creating a purchase
- `create` - Create a purchase and store it in the database
- `edit` - Display a form for editing a purchase
- `update` - Applies an update to a purchase
- `destroy` - Deletes a purchase

The scaffold generator gave us these actions, and their relevant templates. It's a great example of how to quickly build something with Rails.

1.4. Summary

In this chapter, you learned what Rails is and how to get an application started with it: the absolute bare, bare, bare essentials of a Rails application. But look how fast you got going! It

took only a few simple commands and an entire two lines of your own code to create the bones of a Rails application. From this basic skeleton, you can keep adding bits and pieces to develop your application, and all the while you get things for free from Rails. You don't have to code the logic of what happens when Rails receives a request or specify what query to execute on your database to insert a record—Rails does it for you.

You also saw that some big-name players—such as Twitter and GitHub—use Ruby on Rails. A wide range of companies have built successful websites on the Rails framework, and a lot more will do so in the future. Rails also has been around for a decade and a half, and shows no signs of slowing down any time soon.

Still wondering if Ruby on Rails is right for you? Ask around. You'll hear a lot of people singing its praises. The Ruby on Rails community is passionate not only about Rails but also about community building. Events, conferences, user group meetings, and even camps are held around the world for Rails. Attend these, and discuss Ruby on Rails with the people who know about it. If you can't attend these events, the Ruby on Rails forum at https://discuss.rubyonrails.org/, not to mention Stack Overflow and a multitude of other areas on the internet where you can discuss with experienced people what they think of Rails. Don't let this book be the only source for your knowledge. There's a whole world out there, and no book could cover it all!

The best way to answer the question "What is Rails?" is to experience it for yourself. This book and your own exploration can eventually make you a Ruby on Rails expert.

When you added validations to your application earlier, you manually tested that they were working. This may seem like a good idea for now, but when the application grows beyond a couple of pages, it becomes cumbersome to manually test it. Wouldn't it be nice to have some automated way of testing your applications? Something to ensure that all the individual parts always work? Something to provide the peace of mind that you crave when you develop anything? You want to be sure that your application is continuously working with the most minimal effort possible, right?

Well, Ruby on Rails does that too. Several testing frameworks are available for Ruby and Ruby on Rails, and in chapter 2 we look at one of the major ones: RSpec.

[1] For a full treatment of the Ruby language, we highly recommend The Well-Grounded Rubyist, by David A. Black: https://www.manning.com/books/the-well-grounded-rubyist-third-edition
[2] http://en.wikipedia.org/wiki/MIT_License
[3] The upgrade guides and release notes provide a great overview on the new features, bugfixes, and other changes in each major and minor versions of Rails, and can be found under "Release Notes" in the Rails Guides. http://guides.rubyonrails.org
[4] 20-minute blog screencast: https://www.youtube.com/watch?v=OaDhY_y8WTo
[5] These gems share the same version number as Rails, which means that when you're using Rails 6.0, you're also using version 6.0 of the sub-gems. This is helpful to know when you upgrade Rails, because the version number of the installed gems should be the same as the version number of Rails.
[6] Read more GitHub stats on their "State of the Octoverse" report: https://octoverse.github.com/
[7] http://github.com/postmodern/ruby-install
[8] http://rubyinstaller.org
[9] Alternatively, you can store the amount in cents as an integer and then do the conversion back to a full dollar amount. In this example we are using decimal because it's easier to not have to define the conversion at this point. It is worth noting that you shouldn't use a float to store monetary amounts, because it can lead to incorrect rounding errors.
[10] If you want to roll back more than one migration, use the `rails db:rollback STEP=3` command, which rolls back the three most recent migrations.
[11] Where `[timestamp]` in this example is an actual timestamp formatted like `YYYYmmddHHMMSS`: Four digits for the year, two digits for the month, two digits for the day, two digits for the hour, two digits for the minutes and two digits for the second.
[12] The Rails word for fields.
[13] To do that, you call `redirect_to new_purchase_path`, but that wouldn't persist the state of the `@purchase` object to this new request without some seriously bad hackery. By re-rendering the template, you can display information about the object if the object is invalid.
[14] This syntax is exceptionally handy if you have an object and aren't sure of its type but still want to generate a link for it. For example, if you had a different kind of object called `Order` and it was used instead, it would use `order_path` rather than `purchase_path`. A longer explanation of how this works in Rails can be found here: https://ryanbigg.com/2018/12/polymorphic-routes
[15] The `PATCH` HTTP method is implemented by Rails by affixing a `_method` parameter on the form with the value of `PATCH`, because the HTML specification does not allow the `PATCH` method for form elements. It only allows `GET` and `POST`. See here: http://www.w3.org/TR/html401/interact/forms.html#adef-method
[16] As described by the Mac OS X Dictionary application.

Chapter 2. Writing automated tests

In the last chapter, we saw how we could build a very simple application to track a list of purchases. When we used this application we manually tested that it was working. This may seem like a good idea for now, but when we build bigger applications that have a lot of features, it will take a very long time to test it ourselves in this way. So how do we make our Rails applications maintainable and testable, even as they grow larger?

The answer is that you write automated tests for the application as you develop it, and you write these all the time. By writing automated tests for your application, you can quickly ensure that your application is working as intended. If you didn't write tests, your alternative would be to check the entire application manually every time you make a change, which is time consuming and error prone.

Automated testing saves you a ton of time in the long run, and leads to fewer bugs. Humans make mistakes; programs (if coded correctly) don't. We're going to be doing it correctly from step one.[17]

In the Ruby world, a huge emphasis is placed on testing, specifically on test-driven development (TDD) and behavior-driven development (BDD). This chapter covers using RSpec and Capybara, two testing gems that we can use to test our application is behaving correctly.

By learning good testing techniques now, at this early point of your Rails journey, you will have a solid way to make sure nothing is broken when you start to write your first real Rails application. If you didn't write tests, there would be no automatic way of telling what could go wrong in your code. Manually testing a large application takes a very, very long time!

A cryptic yet true answer to the question "Why should I test?" is, "Because you're human." Humans—the large majority of this book's audience—make mistakes. It's one of our favorite ways to learn. And of course: "to err is human". Having a tool to inform us when we make one is helpful, isn't it? Automated testing provides a quick safety net to inform developers when they make mistakes. By they, of course, we mean you. We want you to make as few mistakes as possible.

There are many different types of tests that we can write for our application. Some of the tests could be as small as checking that a method returns a particular value when given certain arguments. These are called unit tests; they test that a single unit of our application is

working.

Others can be as large as making sure that when a user performs a certain set of tasks within our application that the application works correctly. These ones are called feature tests; they test that a combination of things in our application work together in unison.

In this chapter, we're going to look at that latter kind of test, a feature test. We'll see plenty of examples of unit tests and other kinds of tests throughout this book, but for this chapter we'll be focussing on just the one: the feature tests.

A feature test typically checks that an entire feature for a Rails application works. For example, one of the features of the `things_i_bought` application is creating purchases. When we write a feature test for this part of our application, we would want to make sure that at least the user could create a purchase and then immediately view that purchase's details in our application.

By writing this feature test, we will have an ongoing automated way of ensuring that our code is working as we intended it.

We'll see a few examples of how to write these feature tests in this topic. Later on in this book, we'll be writing a lot more in a process called Behaviour Driven Development, or BDD. We'll talk more about BDD in the next chapter. For now, let's just familiarize ourselves with how to write feature tests.

2.1. Installing and setting up RSpec

To write this feature test, we're going to add a new gem to our application called `rspec-rails`. The `rspec-rails` gem provides us with a testing framework that is widely popular in the Ruby community. [18] In this chapter, we'll add the `rspec-rails` gem to our application and use it to write a few tests.

Before we can start writing a test for this application, we are going to need to setup RSpec.

Firstly, we'll need to add `rspec-rails` to our `Gemfile`, inside the same group as another gem called `byebug`:

Listing 26. Gemfile

```
group :development, :test do
  gem 'rspec-rails', '~> 4.0.0'
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
end
```

Next, we'll need to run `bundle install` inside the application's directory to install this gem.

The final step for setting up this gem is to run the `rails g` command again:

```
$ rails g rspec:install
```

This command will generate three files and one directory:

```
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

The `.rspec` file contains options for running RSpec. So far, it's fairly sparse:

```
--require spec_helper
```

The lines in this file are automatic command line arguments passed to RSpec. To find out what other command line arguments you could pass, run `rspec --help`.

This one line directs RSpec to require the `spec_helper.rb` file (located under the `spec` directory) every time RSpec runs.

The `spec/spec_helper.rb` includes basic configuration for RSpec, and `spec/rails_helper.rb` requires the `spec/spec_helper.rb` file, but will also load our Rails application at the same time. We don't need to look at these particular files right now.

Now that we have setup RSpec, let's write our first feature test.

2.2. Writing our first feature test

We're going to write a feature that will test the process of creating posts in our journal. To do this, we can create a new file under `spec/features` called `spec/features/creating_purchases_spec.rb`:

Listing 27. spec/features/creating_purchases_spec.rb

```
require "rails_helper"

RSpec.feature "Creating Purchases" do
  scenario "creating a purchase successfully" do
    visit "/purchases"
    click_link "New Purchase"
    fill_in "Name", with: "Shoes"
    fill_in "Cost", with: 100
    click_button "Create Purchase"

    expect(page).to have_content("Purchase was successfully created.")
  end
end
```

This test uses RSpec's generated `spec/rails_helper.rb` to load the configuration required for this test, as well as `spec/spec_helper.rb`. When we require `rails_helper`, this loads our Rails application and makes it available to our tests automatically.

The `RSpec.feature` block wraps all of the tests for creating a purchase within our application. We use `RSpec.feature` to group together specific scenarios that we want to test for a particular feature.

Each `scenario` block inside of this main `feature` block describes scenarios for creating a purchase. So far, we have just the one, but we could have more. For instance, later on we'll add another scenario that checks what happens if a purchase's `name` is left blank.

Inside the `scenario` block, we use methods like `visit` and `click_link`. These are methods from a gem called `capybara` that comes pre-installed with every Rails application. You'll see this gem listed under the `test` group in your application's `Gemfile`:

Listing 28. Gemfile

```
group :test do
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '>= 2.15'
  gem 'selenium-webdriver'
  # Easy installation and use of web drivers to run system tests with browsers
  gem 'webdrivers'
end
```

Capybara is a testing tool that allows us to simulate the steps of a user of our application. We can tell it visit pages, fill in fields, click buttons (and links) and assert that pages have certain content. There's a lot more that it can do too, which we'll see throughout this book. In fact, we're going to be using it in every chapter!

Back to the test itself. This test follows the same flow that we have been doing when we go to create a purchase. It visits the `/purchases` path for our application by using the `visit` method., just like we would when using a real browser.

The test then fills in the "Name" and "Cost" fields by using the `fill_in` method. Then, it clicks the "Create Purchase" button with `click_button`. These methods all come from the Capybara gem.

Lastly, we use the `have_content` matcher from Capybara to test for there being a message indicating that the purchase was created successfully. This assertion would only work if our purchase creation succeeded. If the purchase creation failed, then we should not see this message and our test would fail.

Notice here that the test reads a lot like plain English. This is an intentional design decision by the creators of Capybara. It's one of the nice things about Ruby: we can write code that reads almost like English. Other people who read this code, even those who don't know Ruby, could read it and understand themselves how to test this feature out themselves if they wished.

We can run this test by running this command:

```
$ bundle exec rspec
```

When we run this command, we'll see that the test passes already:

```
1 example, 0 failures
```

This test is passing because our creating purchase feature is still working! We haven't done anything to break this test.

The great thing about having a test like this in our application is that if we do break something, then the test will tell us!

2.3. Writing a second scenario

Let's look at an example by writing another test. This time, we'll check for what happens if we do not enter a cost for a purchase.

If you try to do this through the application right now, here's what you'll see:

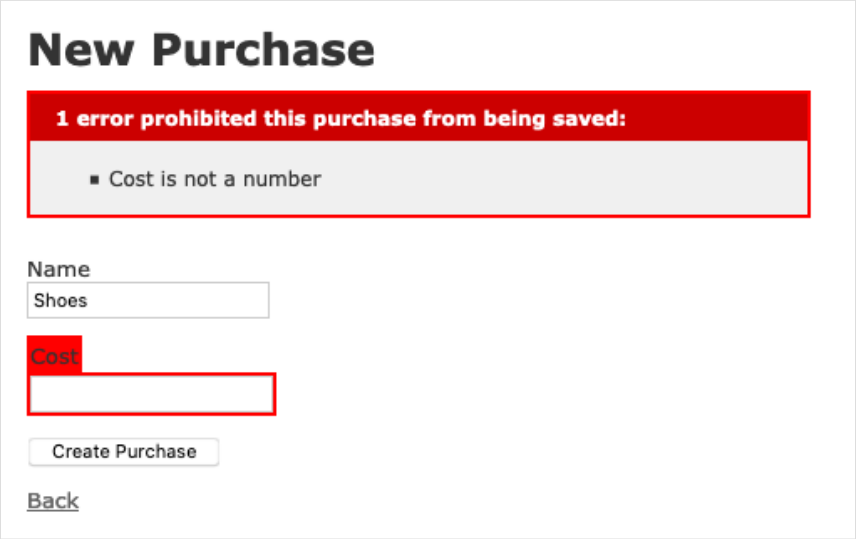


Figure 15. Validation error: Cost is not a number

With this newest test, we want to ensure that when we enter a name but don't put in a value for cost, this error appears. We'll write a new `scenario` block inside the `feature` block to test what happens here:

Listing 29. spec/features/creating_purchases_spec.rb

```
scenario "creating a purchase without a cost" do
  visit "/purchases"
  click_link "New Purchase"
  fill_in "Name", with: "Shoes"
  click_button "Create Purchase"

  expect(page).to have_content("1 error prohibited this purchase from being saved:")
  expect(page).to have_content("Cost is not a number")
end
```

This test expects that when we fill in the form incorrectly, that the validation message appears. Let's run our tests one more time:

```
$ bundle exec rspec
```

When we run this command, we'll see that both of our tests are currently passing:

```
2 examples, 0 failures
```

This is a good thing, as it means our application is still working as we want it to. Now, what happens if, let's say, in a few months time you come back to this application and accidentally (or on purpose!) delete the cost validation line from the `Purchase` model.

Listing 30. app/models/purchase.rb

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

Well, your tests will catch this mistake. If you delete this line and run the tests again, this is what you'll see:

```
Failures:

  1) Creating Purchases creating a purchase without a cost
     Failure/Error: expect(page).to have_content("1 error prohibited this purchase from being
saved:")
       expected to find text "1 error prohibited this purchase from being saved:"
         in "Purchase was successfully created.\nName: Shoes\nCost:\nEdit | Back"
     # ./spec/features/creating_purchases_spec.rb:20:in `block (2 levels) in <top
(required)>'
```

While our first test is passing, this second is one is now failing. This test shows that there's been a change in the application and that the change disagrees with the test's view of How Things Are Meant To Be. This test is giving us a warning that hey, perhaps we were too quick to make that change. This is the saving grace of tests: they prevent you from making easy mistakes like this one.

Let's go ahead and undo that change now:

Listing 31. app/models/purchase.rb

```
class Purchase < ActiveRecord::Base
  validates :name, presence: true
  validates :cost, numericality: { greater_than: 0 }
end
```

Now if we run our tests again, we'll see that they're back to passing:

```
2 examples, 0 failures
```

Great!

2.4. Takeaway

Testing is good for your application's health and your sanity.

Write tests.

2.5. Summary

Writing tests for your application's code provides a quick-and-easy way to have an automated process of checking your application is behaving correctly. If a mistake is made, the tests will (most likely) alert you to that change and you can decide from there if you want to want to change the code, or the test.

You'll apply what you learned in this chapter to building a Rails application from scratch in upcoming chapters. From the next chapter, you will use RSpec and Capybara to write tests that describe the behavior of your application, before you write code. Then you'll implement the behavior of the application to make these tests pass, and you'll know you're doing it right when the tests are all green.

This process is called Behaviour Driven Development, and it's how we're going to be working for the remainder of this book.

[17] Unlike certain other books.
[18] An informal Twitter poll put RSpec at a 3-to-1 preference for Rails developers: https://mobile.twitter.com/ryanbigg/status/1230960263764967424

Chapter 3. Developing a real Rails application

This chapter gets you started on building a Ruby on Rails application from scratch using the techniques covered in the previous chapter, plus a couple of new ones. With the techniques you learned in chapter 2, you can write features describing the behavior of the specific actions in your application and then implement the code you need to get the feature passing.

For the remainder of the book, this application is the main focus. We guide you through it in an Agile-like fashion. Agile focuses largely on iterative development: developing one feature at a time from start to finish, and then refining the feature until it's viewed as complete before moving on to the next one.[19]

For this example application, your imaginary client, who has limitless time and budget (unlike clients in the real world), wants you to develop a ticket-tracking application to track the company's numerous projects. You'll develop this application using the methodologies outlined in chapter 2, with a small difference: you'll be writing tests first, then code.

You'll work iteratively, delivering small working pieces of the software to the client and then gathering the client's feedback to improve the application as necessary. If no improvement is needed, you can move on to the next prioritized chunk of work.

The first couple of features you develop for this application will lay down the foundation for the application, enabling people to create projects and tickets. Later, in chapters 7 and 8, you'll implement authentication and authorization so that people can sign in to the application. Other chapters cover things like adding comments to tickets, notifying users by email and file uploading in chapter 9.

All the way through this application's development process we will be writing tests. This provides the client with a stable application; and when (not if) a bug crops up, you have a nice test base you can use to determine what's broken. Then you can fix the bug so it doesn't happen again, a process called regression testing.

Overall, this development process is called behavior-driven development. We will start by writing tests that describe the behavior that we want the application to have, and then we will set about writing code to make those tests pass.

As you work with your client to build the features of the application using this behavior-driven development technique, the client may ask why all this prework is necessary. This can be a tricky question to answer. Explain that writing the tests before the code and then

implementing the code to make the tests pass creates a safety net to ensure that the code is always working. (Note that tests will make your code more maintainable, but they won't make your code bug-proof.)

The tests also give you a clearer picture of what your client really wants. Having it all written down in code gives you a solid reference to point to if clients say they suggested something different. Story-driven development is BDD with emphasis on things a user can do with the system.

By using story-driven development, you know what clients want, clients know you know what they want, you have something you can run automated tests with to ensure that all the pieces are working, and, finally, if something does break, you have the test suite in place to catch it. It's a win-win-win situation.

Some of the concepts covered in this chapter were explained in chapter 1. But rather than using scaffolding, as you did previously, you'll write this application from the ground up using the BDD process and other generators provided by Rails. The `scaffold` generator is great for prototyping, but it's less than ideal for delivering simple, well-tested code that works precisely the way you want it to work. The code provided by the scaffold generator often may differ from the code you want. In this case, you can turn to Rails for lightweight alternatives to the scaffold code options, and you'll likely end up with cleaner, better code.

First, you need to set up your application!

3.1. First steps

Chapter 1 explained how to quickly start a Rails application. This chapter explains a couple of additional processes that improve the flow of your application development. One process uses BDD to create the features of the application; the other process uses version control. Both will make your life easier.

3.1.1. The application story

Your client may have a good idea of the application they want you to develop. How can you transform the idea in your client's brain into beautifully formed code? First, you sit down with your client and talk through the parts of the application. In the programming business, we call these parts user stories, and you'll use RSpec and Capybara to develop them.

Start with the most basic story, and ask your client how they want it to behave. Then sketch

out a basic flow of how the feature would work by building an acceptance test using RSpec and Capybara. If this feature was a login form, the test for it would look something like this:

```
RSpec.feature "Users can log in to the site" do
  scenario "as a user with a valid account" do
    visit "/login"
    fill_in "Email", with: "user@ticketee.com"
    fill_in "Password", with: "password"
    click_button "Login"
    expect(page).to have_content("You have been successfully logged in.")
  end
end
```

The form of this test is simple enough that even people who don't understand Ruby should be able to understand the flow of it. With the function and form laid out, you have a pretty good idea of what the client wants.

Laying the foundations

To start building the application you'll be developing throughout this book, run the good-old `rails` command, preferably outside the directory of the previous application. Call this app ticketee, the Australian slang for a person who validates tickets on trains in an attempt to catch fare evaders. It also has to do with this project being a ticket-tracking application, and a Rails application, at that.[20] To generate this application, run this command:

```
$ rails new ticketee
```

Help!

If you want to see what else you can do with this `new` command (hint: there's a lot!), you can use the `--help` option:

```
$ rails new --help
```

The `--help` option shows you the options you can pass to the `new` command to modify the output of your application.

Presto, it's done! From this bare-bones application, you'll build an application that does the following:

- Tracks tickets (of course) and groups them into projects
- Provides a way to restrict users to certain projects
- Allows users to upload files to tickets
- Lets users tag tickets so they're easy to find
- Provides an API on which users can base development of their own applications

TODO: Update this list when we've finalised all chapters

You can't do all this with a command as simple as `rails new [application_name]`, but you can do it step by step and test it along the way so you develop a stable and worthwhile application.

Throughout the development of the application, we advise you to use a version-control system. The next section covers that topic using Git. You're welcome to use a different version-control system, but this book uses Git exclusively.

3.2. Version control

It is wise during development to use version-control software to provide checkpoints in your code. When the code is working, you can make a commit; and if anything goes wrong later in development, you can revert back to that known-working commit. Additionally, you can create branches for experimental features and work on those independent of the main codebase, without damaging working code.

This book doesn't go into detail on how to use a version-control system, but it does recommend using Git. Git is a distributed version-control system that is easy to use and extremely powerful. If you wish to learn about Git, we recommend reading Pro Git, a free online book by Scott Chacon (Apress, 2014, http://git-scm.com/book/en/v2).

Git is used by most developers in the Rails community and by tools such as Bundler, discussed shortly. Learning Git along with Rails is advantageous when you come across a gem or plug-in that you have to install using Git. Because most of the Rails community uses Git, you can find a lot of information about how to use it with Rails (even in this book!) should you ever get stuck.

If you don't have Git already installed, GitHub's help site offers installation guides for these platforms:

- Mac — https://help.github.com/articles/set-up-git#platform-mac
- Linux — https://help.github.com/articles/set-up-git#platform-linux
- Windows — https://help.github.com/articles/set-up-git#platform-windows

The precompiled installer should work well for Macs, and the package-distributed versions (via `apt`, `yum`, `emerge`, and so on) work well for Linux machines. For Windows, the GitHub for Windows program does just fine.

3.2.1. Getting started with GitHub

For an online place to put your Git repository, we recommend GitHub (http://github.com), which offers free accounts.[21] If you set up an account now, you can upload your code to GitHub as you progress, ensuring that you won't lose it if anything were to happen to your computer. To get started with GitHub, you first need to generate a secure shell (SSH) key, which is used to authenticate you with GitHub when you do a `git push` to GitHub's servers. You can find a guide for this process at https://help.github.com/articles/generating-ssh-keys.

When you've setup your account, it is now time to create a new repository. Click the "New Repository" button on the dashboard to begin creating a new repository. Enter the Project Name as `ticketee`, and click the "Create Repository" button to create the repository on GitHub.

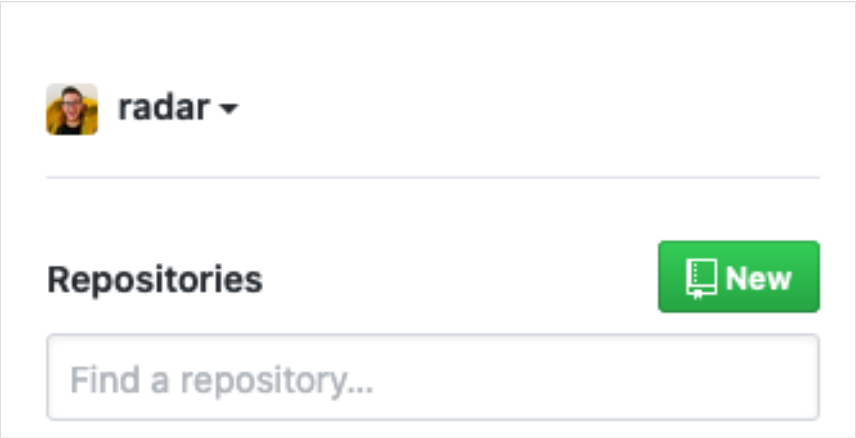


Figure 16. Creating a new repository

Now you're on your project's page. It has some basic instructions on how to set up your code in your new repository, but first you need to configure Git on your own machine. Git needs to know a bit about you for identification purposes - so you can properly be credited (or blamed) for any code that you write.

3.2.2. Configuring your Git client

Run the commands below in your terminal or command prompt to tell Git about yourself, replacing `"Your Name"` with your real name and `you@example.com` with your email address. The email address you provide should be the same as the one you used to sign up to GitHub, so that when you push your code to GitHub, it will also be linked to your account.

Listing 32. Configuring your identity in Git

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

You already have a `ticketee` directory after you generated your Rails app, and you're probably already in it.

If not, you should be. To make this directory a Git repository, run this easy command:

```
$ git init
```

If the git repository already exists, it will show "Reinitialized existing Git repository" which is safe to do as it will not overwrite anything.

Your `ticketee` directory now contains a `.git` directory, which is your Git repository. It's all kept in one neat little package.

To add all the files for your application to this repository's staging area, run

```
$ git add .
```

The staging area for the repository is the location where all the changes for the next commit are kept. A commit can be considered a checkpoint for your code. If you make a change, you must stage that change before you can create a commit for it. To create a commit with a message, run

```
$ git commit -m "Generate the Rails 6 application"
```

This command generates quite a bit of output, but the most important lines are the first two:

```
[master 38e8472] Generate the Rails 6 application
 91 files changed, 9116 insertions(+)
```

`d825bbc` is the short commit ID, a unique identifier for the commit, so it changes with each commit you make. The number of files and insertions may also be different. In Git, commits are tracked against branches, and the default branch for a Git repository is the `master` branch, which you just committed to.

The second line lists the number of files changed, insertions (new lines added) and deletions. If you modify a line, it's counted as both an insertion and a deletion, because, according to Git, you've removed the line and replaced it with the modified version.

To view a list of commits for the current branch, type `git log`. You should see output similar to the following listing.

Listing 33. Viewing the commit log

```
commit d825bbc23854cc256d5829a06516ceb19d148131
Author: Your Name <you@example.com>
Date:   [date stamp]

    Generate the Rails 6 application
```

The hash after the word `commit` is the long commit ID; it's the longer version of the previously sighted short commit ID. A commit can be referenced by either the long or the short commit ID in Git, providing no two commits begin with the same short ID.[22] With that commit in your repository, you have something to push to GitHub, which you can do by running the following, making sure to substitute your own GitHub username in:

```
$ git remote add origin git@github.com:[your username]/ticketee.git
$ git push origin master -u
```

The first command tells Git that you have a remote server called origin for this repository. To access it, you use the `git@github.com:[your username]/ticketee.git` path, which connects to the repository you created on GitHub, using SSH. The next command pushes the named branch to that remote server, and the `-u` option tells Git to always pull from this remote server for this branch unless told differently. The output from this command is similar to the following.

Listing 34. `git push` **output**

```
Counting objects: 73, done.
Compressing objects: 100% (58/58), done.
Writing objects: 100% (73/73), 86.50 KiB, done.
Total 73 (delta 2), reused 0 (delta 0)
To git@github.com:rubysherpas/active_rails_examples.git
* [new branch]  master -> master
Branch master set up to track remote branch master from origin.
```

The second-to-last line in this output indicates that your push to GitHub succeeded, because it shows that a new branch called `master` was created on GitHub. Note that as we go through the book, we'll also `git push` just like you. You can compare your code to ours by checking out our repository on GitHub: https://github.com/rubysherpas/active_rails_examples.

To roll back the code to a given point in time, check out `git log`. What you'll see will be different to what we show below, but it will look similar:

```
commit d1e9b6f398748d3ca8583727c1f86496465ba298
Author: [name] <[email redacted]>
Date:   [timestamp]

    Protect state_id from users who do not have permission
      to change it

commit ceb67d45cfcddbb8439da7b126802e6a48b1b9ea
Author: [name] <[email redacted]>
Date:   [timestamp]

    Only admins and managers can change states of a ticket

commit ef5ec0f15e7add662852d6634de50648373f6116
Author: [name] <[email redacted]>
Date:   [timestamp]

    Auto-assign the default state to newly-created tickets
```

Each of these lines represents a commit, and the commits line up with when we tell you to commit in the book. You can also check out the commit list on GitHub, if you find that easier: https://github.com/rubysherpas/active_rails_examples/commits.

Once you've found the commit you want to go back to, make note of the long commit ID associated with it. Use this value with `git checkout` to roll the code back in time:

```
$ git checkout 23729a
```

You only need to know enough of the hash for it to be unique: six characters is usually enough. When you're done poking around, go forward in time to the most recent commit with `git checkout` again:

```
$ git checkout master
```

This is a tiny, tiny taste of the power of Git. Time travel at will! You just have to learn the commands.

Next, you must set up your application to use RSpec.

3.3. Application configuration

Even though Rails passionately promotes the convention over configuration line, some parts of the application still will need configuration. It's impossible to avoid all configuration. The main parts are gem dependency configuration, database settings, and styling. Let's look at these parts now.

3.3.1. The Gemfile and generators

The `Gemfile` is used for tracking which gems are used in your application. "Gem" is the Ruby word for a library of code, all packaged up to be included into your app - Rails is a gem, and it in turn depends on many other gems. Bundler is a gem, and Bundler is also responsible for everything to do with this `Gemfile`. It's Bundler's job to ensure that all the gems listed inside the `Gemfile` are installed when your application is initialized. Let's look at the following listing to see how it looks inside.

Listing 35. Default Gemfile in a new Rails app

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

ruby '2.7.1'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 6.1.0'
# Use sqlite3 as the database for Active Record
```

```
gem 'sqlite3', '~> 1.4'
# Use Puma as the app server
gem 'puma', '~> 5.0'
# Use SCSS for stylesheets
gem 'sass-rails', '>= 6'
# Transpile app-like JavaScript. Read more: https://github.com/rails/webpacker
gem 'webpacker', '~> 5.0'
# Turbolinks makes navigating your web application faster. Read more:
https://github.com/turbolinks/turbolinks
gem 'turbolinks', '~> 5'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.7'
# Use Redis adapter to run Action Cable in production
# gem 'redis', '~> 4.0'
# Use Active Model has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Active Storage variant
# gem 'image_processing', '~> 1.2'

# Reduces boot times through caching; required in config/boot.rb
gem 'bootsnap', '>= 1.4.4', require: false

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  # Access an interactive console on exception pages or by calling 'console' anywhere in the
code.
  gem 'web-console', '>= 4.1.0'
  # Display performance information such as SQL time and flame graphs for each request in
your browser.
  # Can be configured to work on production as well see:
https://github.com/MiniProfiler/rack-mini-profiler/blob/master/README.md
  gem 'rack-mini-profiler', '~> 2.0'
  gem 'listen', '~> 3.3'
  # Spring speeds up development by keeping your application running in the background. Read
more: https://github.com/rails/spring
  gem 'spring'
end

group :test do
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '>= 3.26'
  gem 'selenium-webdriver'
  # Easy installation and use of web drivers to run system tests with browsers
  gem 'webdrivers'
end
```

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

In this file, Rails sets a source to be https://rubygems.org (the canonical repository for Ruby gems). All gems you specify for your application are gathered from the source. Next, it tells Bundler it requires version {rails_version} of the `rails` gem. Bundler inspects the dependencies of the requested gem, as well as all gem dependencies of those dependencies (and so on), and then does what it needs to do to make them all available to your application.

This file also requires the `sqlite3` gem, which is used for interacting with SQLite3 databases, the default when working with Rails. If you were to use another database system, you would need to take out this line and replace it with the relevant gem, such as `mysql2` for MySQL or `pg` for PostgreSQL.

Groups in the `Gemfile` are used to define gems that should be loaded in specific scenarios. When using Bundler with Rails, you can specify a gem group for each Rails environment, and by doing so, you specify which gems should be required by that environment. A default Rails application has three standard environments: development, test, and production.

Rails application environments

The development environment is used for your local application, such as when you're playing with it in the browser on your local machine. In development mode, page and class caching are turned off, so requests may take a little longer than they do in production mode. (Don't worry—this is only the case for larger applications.) Things like more detailed error messages are also turned on, for easier debugging.

The test environment is used when you run the automated test suite for the application. This environment is kept separate from the development environment so your tests start with a clean database to ensure predictability, and so you can include extra gems specifically to aid in testing.

The production environment is used when you finally deploy your application out into the world for others to use. This mode is designed for speed, and any changes you make to your application's classes aren't effective until the server is restarted.

This automatic requiring of gems in the Rails environment groups is done by this line in `config/application.rb`:

```
Bundler.require(*Rails.groups)
```

The `Rails.groups` line provides two groups for Bundler to require: `default` and `development`. The latter will change depending on the environment that you're running. This code will tell Bundler to load only the gems in the "default" group (which is all gems not in any specific group), as well as any gems in a group that has the same name as the environment.

Starting with Behavior Driven Development

Chapter 2 focused on Behavior Driven Development (BDD), and, as was more than hinted at, you'll be using it to develop this application. To get started, alter the `Gemfile` to ensure that you have the correct gem for RSpec for your application. To add the `rspec-rails` gem, we'll add this line to the `:development, :test` group in our `Gemfile`:

```
group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a debugger console
  gem 'byebug', platforms: [:mri, :mingw, :x64_mingw]

  gem 'rspec-rails', '~> 4.0.0'
end
```

This group in our `Gemfile` lists all the gems that will be loaded in the `development` and `test` environments of our application. These gems will not be available in a `production` environment. We're adding `rspec-rails` to this group because we're going to need a generator from it to be available in development. Additionally, when you run a generator for a controller or model, it'll use RSpec, rather than the default Test::Unit, to generate the tests for that class.

You've specified a version number with `~> 4.0.0` [23] which tells RubyGems you want `rspec-rails` version 4.0.0 or higher, but less than `rspec-rails` 4.1.0. This means when RSpec releases 4.0.1 and you go to install your gems, RubyGems will install the latest version it can find, rather than only 4.0.1.

The other gem that we'll be using here is the `capybara` gem, but that is already in our `Gemfile`:

Listing 36. Gemfile

```
group :test do
  # Adds support for Capybara system testing and selenium driver
  gem 'capybara', '>= 3.2.6'
  gem 'selenium-webdriver'
  # Easy installation and use of web drivers to run system tests with browsers
  gem 'webdrivers'
end
```

As we saw in the last chapter, Capybara can simulate actions on our application, allowing us to test the application automatically.

Capybara also supports real browser testing. If you tell RSpec that your test is a JavaScript test, it will open a new Firefox window and run the test there - you'll actually be able to see your tests as they occur, and your application will behave exactly the same as it does when you view it yourself. You'll use this extensively when we start writing JavaScript in chapter 9.

To install these gems to your system, run this command:

```
bundle install
```

With the necessary gems for the application installed, you should next run the `rspec:install` generator, a generator provided by RSpec to set your Rails application up for testing.

```
$ rails g rspec:install
```

Remember, `rails g` is a shortcut for running `rails generate`!

You can also remove the default generated `test` directory in the root folder of your application - you won't be using it. You'll write tests under the `spec` directory instead.

With this generated code in place, you should make a commit so you have another base to roll back to if anything goes wrong:

```
$ git add .
$ git commit -m "Set up gem dependencies and run RSpec generator"
$ git push
```

3.3.2. Database configuration

By default, Rails uses a database system called SQLite3, which stores each environment's database in separate files in the `db` directory. SQLite3 is the default database system because it's the easiest to set up. Out of the box, Rails also supports the MySQL and PostgreSQL databases, with gems available that can provide functionality for connecting to other database systems such as Oracle.

If you want to change which database your application connects to, you can open `config/database.yml` (development configuration shown in the following listing) and alter the settings to the new database system.

Listing 37. config/database.yml, SQLite3 example

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

For example, if you want to use PostgreSQL, you change the settings to read as in the following listing. It's common convention, but not mandatory, to call the environment's database `[app_name]_[environment]`.

Listing 38. config/database.yml, PostgreSQL example

```
development:
  adapter: postgresql
  database: ticketee_development
  username: root
  password: t0ps3cr3t
```

You're welcome to change the database if you wish. Rails will go about its business. However, it's good practice to develop and deploy on the same database system to avoid strange behavior between two different systems. Systems such as PostgreSQL perform faster than SQLite, so switching to it may increase your application's performance. Be mindful, however, that switching database systems doesn't automatically move your existing data over for you.

It's generally wise to use different names for the different database environments: if you use the same database in development and test modes, the database would be emptied of all data

when the tests were run, eliminating anything you might have set up in development mode. You should never work on the live production database directly unless you're absolutely sure of what you're doing, and even then extreme care should be taken.

Finally, if you're using MySQL, it's wise to set the encoding to `utf-8` for the database, using this setup in the `config/database.yml` file.

Listing 39. config/database.yml, MySQL example

```
development:
  adapter: mysql2
  database: ticketee_development
  username: root
  password: t0ps3cr3t
  encoding: utf8
```

This way, the database is set up automatically to work with UTF-8, eliminating any potential encoding issues that may be encountered otherwise.

That's database configuration in a nutshell. For this book and for our Ticketee application, we'll be using the default of SQLite3, but it's good to know about the alternatives and how to configure them.

3.4. Beginning your first feature

You now have version control for your application, and you're safely storing the code for it on GitHub. It's now time to write your first Capybara-based test, which isn't nearly as daunting as it sounds. We'll explore things such as models and RESTful routing while you do it. It'll be simple, promise!

3.4.1. Creating projects

The CRUD (create, read, update, delete) acronym is something you will see all the time in the Rails world. It represents the creation, reading, updating, and deleting of something, but it doesn't say what that something is.

In the Rails world, CRUD is usually referred to when talking about resources. Resources are the representation of the information throughout your application - the "things" that your application is designed to manage. The following section goes through the beginnings of generating a CRUD interface for a project resource, by applying the BDD practices you

learned in chapter 2 to the application you just bootstrapped. What comes next is a sampler of how to apply these practices when developing a Rails application. Throughout the remainder of the book, you'll continue to apply these practices to ensure that you have a stable and maintainable application. Let's get into it!

The first story for your application is the creation (the C in CRUD). You'll create a resource representing projects in your application by first writing a test for the process by which a user will create projects, then creating a controller and model, and then creating a route. Then you'll add a validation to ensure that no project can be created without a name. When you're done with this feature, you'll have a form that looks like this:

Figure 17. Form to create projects

First, create a new directory at `spec/features` - all of the specs covering our features will go there. Then, in a file called `spec/features/creating_projects_spec.rb`, you'll put the test

that will make sure this feature works correctly when it's fully implemented. This code is shown in the following listing.

Listing 40. spec/features/creating_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new projects" do
  scenario "with valid attributes" do
    visit "/"

    click_link "New Project"

    fill_in "Name", with: "Visual Studio Code"
    fill_in "Description", with: "Code Editing. Redefined"
    click_button "Create Project"

    expect(page).to have_content "Project has been created."
  end
end
```

To run this test, run this command from inside the `ticketee` directory:

```
bundle exec rspec
```

This command will run all of your specs and display your application's first test's first failure:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   ActionController::RoutingError:
     No route matches [GET] "/"
```

It falls on the application's router to figure out where the request should go. Typically, the request would be routed to an action in a controller, but at the moment there's no routes at all for the application. With no routes, the Rails router can't find the route for "/" and so gives you the error shown.

You have to tell Rails what to do with a request for `/`. You can do this easily in `config/routes.rb`. At the moment, this file has the following content:

Listing 41. config/routes.rb

```
Rails.application.routes.draw do
  # For details on the DSL available within this file, see
https://guides.rubyonrails.org/routing.html
end
```

To define a root route, you use the `root` method like this in the block for the `draw` method:

```
Rails.application.routes.draw do
  root "projects#index"
end
```

This defines a route for requests to `/` (the root route) to point at the `index` action of the `ProjectsController`. This means that when anyone visits the "root" path of our application (for example: http://localhost:3000), they will see this page.

This controller doesn't exist yet, and so the test should probably complain about that if you got the route right. Run `bundle exec rspec` to find out:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   ActionController::RoutingError:
     uninitialized constant ProjectsController
```

This error is happening because the route is pointing at a controller that doesn't exist. When the request is made, the router attempts to load the controller, and because it can't find it, you'll get this error. To define this `ProjectsController` constant, you must generate a controller. The controller is the first port of call for your routes (as you can see now!) and is responsible for querying the model for information in an action and then doing something with that information (such as rendering a template). (Lots of new terms are explained later. Patience, grasshopper.) To generate this controller, run this command:

```
$ rails g controller projects
```

You may be wondering why we're using a pluralized name for the controller. Well, the controller is going to be dealing with a plural number of projects during its lifetime, and so it only makes sense to name it like this. The models are singular because their name refers to

their type. Another way to put it: you're a Human, not a Humans. But a controller that dealt with multiple humans would be called HumansController.

The controller generator produces output similar to the output produced when you ran `rails new` earlier, but this time it creates files just for the controller we've asked Rails to generate. The most important of these is the controller itself, which is housed in `app/controllers/projects_controller.rb` and defines the `ProjectsController` constant that your test needs. This controller is where all the actions will live, just like `app/controllers/purchases_controller.rb` back in chapter 1. Here's what this command outputs:

```
create  app/controllers/projects_controller.rb
invoke  erb
create    app/views/projects
invoke  rspec
create    spec/requests/projects_request_spec.rb
invoke  helper
create    app/helpers/projects_helper.rb
invoke    rspec
create      spec/helpers/projects_helper_spec.rb
invoke  assets
invoke    scss
create      app/assets/stylesheets/projects.scss
```

Before we dive into that, a couple of notes about the output.

- `app/views/projects` contains the views relating to your actions (more on this shortly).
- `invoke helper` shows that the `helper` generator was called here, generating a file at `app/helpers/projects_helper.rb`. This file defines a `ProjectsHelper` module. Helpers generally contain custom methods to use in your view that help with the rendering of content, and they come as blank slates when they're first created.
- `invoke erb` signifies that the Embedded Ruby (ERB) generator was invoked. Actions to be generated for this controller have corresponding ERB views located in `app/views/projects`. For instance, the `index` action's default view will be located at `app/views/projects/index.html.erb` when we create it later on.
- `invoke rspec` shows that the RSpec generator was also invoked during the generation. This means RSpec has generated a new file at `spec/helpers/projects_helper.rb`, which you can use to test your helper—but not right now.[24]

- Finally, the stylesheet for this controller is generated, `app/assets/stylesheets/projects.scss`. This file will contain any CSS related to the controller's views, written using SCSS (http://sass-lang.com). These files will be automatically converted into CSS so that they can be read and used by the browser.

You've just run the generator to generate a new `ProjectsController` class and all its goodies. This should fix the "uninitialized constant" error message. If you run `bundle exec rspec` again, it declares that the `index` action is missing:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"

   AbstractController::ActionNotFound:
     The action 'index' could not be found for ProjectsController
```

Defining a controller action

To define the `index` action in your controller, you must define a method in the `ProjectsController` class, just as you did when you generated your first application, as shown in the following listing.

Listing 42. app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end
end
```

If you run `bundle exec rspec` again, this time Rails complain of a missing template `projects/index`:

```
1) Users can create new projects with valid attributes
  Failure/Error: visit "/"

  ActionController::MissingExactTemplate:
    ProjectsController#index is missing a template for request formats: text/html
```

This error says that we're missing a template for the request format of `text/html`. This means that we will need to create a view.

To generate this view, create the `app/views/projects/index.html.erb` file and leave it blank for now. This file is called `index.html.erb` so that we have the correct format (HTML), and this file will be using ERB to evaluate some Ruby to generate some of that HTML, hence the `.erb` extension.

Let's run this test one more time:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_link "New Project"
   Capybara::ElementNotFound:
     Unable to find link "New Project"
```

You've defined a home page for your application by defining a root route, generating a controller, putting an action in it, and creating a view for that action. Now Capybara is successfully navigating to it, and rendering it. That's the first step in the first test passing for your first application, and it's a great first step!

The second line in your spec is now failing, and it's up to you to fix it. You need a link on the root page of your application that reads "New Project". That link should go in the view of the controller that's serving the root route request: `app/views/projects/index.html.erb`. Create a new file at `app/views/projects/index.html.erb` and open it for editing. Put the "New Project" link in by using the `link_to` method:

Listing 43. app/views/projects/index.html.erb

```
<%= link_to "New Project", new_project_path %>
```

This single line reintroduces two old concepts and a new one: ERB output tags, the `link_to` method (both of which you saw in chapter 1), and the mysterious `new_project_path` method.

As a refresher, in ERB, when you use `<%=` (known as an ERB output tag), you're telling ERB that whatever the output of this Ruby is, put it on the page. If you only want to evaluate Ruby, you use an ERB evaluation tag `<%`, which doesn't output content to the page but only evaluates it. Both of these tags end in `%>`.

The `link_to` method in Rails generates an `<a>` tag with the text of the first argument and the `href` of the second argument. This method can also be used in block format if you have a lot of text you want to link to:

```
<%= link_to new_project_path do %>
  bunch
  of
  text
<% end %>
```

Where `new_project_path` comes from deserves its own section. It's the very next one.

3.4.2. RESTful routing

The `new_project_path` method is as yet undefined. If you ran the test again, it would complain of an "undefined local variable or method 'new_project_path'". You can define this method by defining a route to what's known as a resource in Rails. Resources are collections of objects that all belong in a common location, such as projects, users, or tickets. You can add the projects resource in `config/routes.rb` by using the `resources` method, putting it directly under the `root` method in this file.

Listing 44. `resources :projects` **line in config/routes.rb**

```
Rails.application.routes.draw do
  root "projects#index"

  resources :projects
end
```

This is called a resource route, and it defines the routes to the seven RESTful actions in your `ProjectsController`.

We saw this method used back in Chapter 1, except then it generated routes to the actions in `PurchasesController`.

When something is said to be RESTful, it means it conforms to Rails' interpretation of the Representational State Transfer (REST) architectural style.[25]

With Rails, this means the related controller has seven potential actions:

- `index`
- `show`
- `new`

- `create`
- `edit`
- `update`
- `destroy`

These seven actions match to just four request paths:

- `/projects`
- `/projects/new`
- `/projects/:id`
- `/projects/:id/edit`

How can four be equal to seven? It can't! Not in this world, anyway. Rails will determine what action to route to on the basis of the HTTP method of the requests to these paths. We can see this if we run this command:

```
rails routes -c projects
```

This command will show us the routes that are defined for the `ProjectsController`:

Table 1. RESTful routing matchup

Prefix	Verb	URI Pattern	Controller#Action
projects	GET	/projects	projects#index
	POST	/projects	projects#create
new_project	GET	/projects/new	projects#new
edit_project	GET	/projects/:id/edit	projects#edit
project	GET	/projects/:id	projects#show

Prefix	Verb	URI Pattern	Controller#Action
	PATCH	/projects/:id	projects#update
	PUT	/projects/:id	projects#update
	DELETE	/projects/:id	projects#destroy

The routes listed in the table are provided when you use `resources :projects`. This is yet another great example of how Rails takes care of the configuration so you can take care of the coding.

The words in the leftmost column of this output are the beginnings of the method names you can use in your controllers or views to access them. If you want just the path to a route, such as /projects, then use `projects_path`. If you want the full URL, such as http://yoursite.com/projects, use `projects_url`. It's best to use these helpers rather than hard-coding the URLs; doing so makes your application consistent across the board. For example, to generate the route to a single project, you would use either `project_path` or `project_url`:

```
project_path(@project)
```

This method takes one argument, shown in the URI pattern with the `:id` notation, and generates the path according to this object. If the `id` attribute for `@project` was `1`, then the path this method would generate is `/projects/1`.

Running `bundle exec rspec` now produces a complaint about a missing `new` action:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_link "New Project"
   AbstractController::ActionNotFound:
     The action 'new' could not be found for ProjectsController
```

As shown in the following listing, define the `new` action in your controller by defining a `new` method directly underneath the `index` method.

Listing 45. app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end

  def new
  end
end
```

Running `bundle exec rspec` now results in a complaint about a missing `new` template, just as it did with the `index` action:

```
1) Users can create new projects with valid attributes
  Failure/Error: click_link "New Project"

  ActionController::MissingExactTemplate:
    ProjectsController#new is missing a template for request formats: text/html
```

You can create the file at `app/views/projects/new.html.erb` to make this test go one step further, although this is a temporary solution. You'll come back to this file later to add content to it. When you run the spec again, the line that should be failing is the one regarding filling in the "Name" field. Find out if this is the case by running `bundle exec rspec`:

```
1) Users can create new projects with valid attributes
   Failure/Error: fill_in "Name", with: "Visual Studio Code"
   Capybara::ElementNotFound:
     Unable to find field "Name"
```

Now Capybara is complaining about a missing "Name" field on the page it's currently on, the `new` page. You must add this field so that Capybara can fill it in. Before you do that, however, fill out the `new` action in the `ProjectsController` like the following.

```
def new
  @project = Project.new
end
```

When we fill out the view with the fields we need to create a new project, we'll need something to base the fields on - an instance of the class we want to create. This `Project` constant will be a class, located at `app/models/project.rb`, thereby making it a model.

Of models and migrations

A model is used to perform queries on a database, to fetch or store information. Because models by default inherit from Active Record, you don't have to set up anything extra. Run the following command to generate your first model:

```
$ rails g model project name description
```

This syntax is similar to the controller generator's syntax except that you specified you want a model, not a controller. When the generator runs, it generates not only the model file but also a migration containing the code to create the table for the model, containing the specified fields. You can specify as many fields as you like after the model's name. They default to `string` type, so you didn't need to specify them. If you wanted to be explicit, you could use a colon followed by the field type, like this:

```
$ rails g model project name:string description:string
```

A model provides a place for any business logic that your application does. One common bit of logic is the way your application interacts with a database. A model is also the place where you define validations (seen later in this chapter), associations (discussed in chapter 5) and scopes (easy-to-use filters for database calls, discussed in chapter 7), among other things. To perform any interaction with data in your database, you go through a model.[26]

Migrations are effectively version control for the database. They're defined as Ruby classes, which allows them to apply to multiple database schemas without having to be altered. All migrations have a change method in them when they're first defined. For example, the code shown in the following listing comes from the migration that was just generated.

Listing 46. db/migrate/[date]_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration[6.1]
  def change
    create_table :projects do |t|
      t.string :name
      t.string :description

      t.timestamps
    end
  end
end
```

When you run the migration forward (using `bundle exec rails db:migrate`), it creates the table in the database. When you roll the migration back (with `rails db:rollback`), it deletes (or drops) the table from the database. If you need to do something different on the up and down parts, you can use those methods instead:

Listing 47. Explicitly using `up` and `down` methods to define a migration

```
class CreateProjects < ActiveRecord::Migration[6.1]
  def up
    create_table :projects do |t|
      t.string :name
      t.string :description

      t.timestamps
    end
  end

  def down
    drop_table :projects
  end
end
```

Here, the `up` method would be called if you ran the migration forward, and the `down` method would be run if you ran it backward.

This syntax is especially helpful if the migration does something that has a reverse function that isn't clear, such as removing a column[27]:

```
class CreateProjects < ActiveRecord::Migration[6.1]
  def up
    remove_column :projects, :name
  end

  def down
    add_column :projects, :name, :string
  end
end
```

This is because Active Record won't know what type of field to re-add this column as, so you must tell it what to do in the case of this migration being rolled back.

In our projects migration, the first line of the `change` method tells Active Record that you want to create a table called `projects`. You call this method using the block format, which returns an object that defines the table. To add fields to this table, you call methods on the block's object (called `t` in this example and in all model migrations), the name of which usually reflects the type of column it is; the first argument is the name of that field. The `timestamps` method is special: it creates two fields, `created_at` and `updated_at`, which are by default set to the current time in co-ordinated universal time (UTC)[28] by Rails when a record is created and updated, respectively.

A migration doesn't automatically run when you create it—you must run it yourself using this command:

```
$ rails db:migrate
```

These commands migrate the database up to the latest migration, which for now is the only migration. If you create a whole slew of migrations at once, then invoking `rails db:migrate` will migrate them in the order in which they were created. This is the purpose of the timestamp in the migration filename - to keep the migrations in chronological order.

With this model created and its related migration run, your test doesn't get any further but you can start building out the form to create a new project.

Form building

To add this field to the `new` action's view, you can put it in a form, but not just any form: a `form_with`, as in the following listing.

Listing 48. app/views/projects/new.html.erb

```
<h1>New Project</h1>
<%= form_with(model: @project, local: true) do |form| %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.label :description %>
    <%= form.text_field :description %>
  </div>

  <%= form.submit %>
<% end %>
```

So many new things!

Starting at the top, the `form_with` method is Rails' way of building forms for Active Record objects. You pass it the `@project` object you defined in your controller as the argument for the `model` option and with this, the helper does much more than simply place a form tag on the page. `form_with` inspects the `@project` object and creates a form builder specifically for that object. The two main things it inspects are whether it's a new record and what the class name is.

Determining what `action` attribute the form has (the URL the form submits its data to) depends on whether the object is a new record or not. A record is classified as new when it hasn't been saved to the database. This check is performed internally by Rails using the `persisted?` method, which returns `true` if the record is stored in the database or `false` if it's not.

The class of the object also plays a pivotal role in where the form is sent - Rails inspects this class and, from it, determines what the route should be. Because `@project` is new and is an object of class `Project`, Rails determines that the submit URL will be the result of `projects_path`, which will mean the route is `/projects` and the method for the form is `POST`. Therefore, a request is sent to the `create` action in `ProjectsController`.

If we were to look at the form's generated HTML, we would see this:

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

The `action` and `method` in combination will make this form submit to `POST /projects`, sending the form's data to the `create` action within `ProjectsController`.

After that part of `form_with` is complete, you use the block syntax to receive a `form` variable, which is a `FormBuilder` object. You can use this object to define your form's fields. The first element you define is a `label`. `label` tags directly relate to the input fields on the page and serve two purposes. First, they give users a larger area to click, rather than just the field, radio button, or check box. The second purpose is so you can reference the label's text in the test, and Capybara will know what field to fill in.

Alternative label naming

By default, the label's text value will be the 'humanized' value of the field name, eg. `:name` becomes "Name". If you want to customize the text, you can pass the `label` method a second argument:

```
<%= form.label :name, "Your name" %>
```

After the label, you put the `text_field`, which renders an `<input>` tag corresponding to the label and the field. The output tag looks like this:

```
<input type="text" name="project[name]" id="project_name" />
```

Then you use the `submit` method to provide users with a submit button for your form. Because you call this method on the `form` object, Rails checks whether the record is new and sets the text to read "Create Project" if the record is new or "Update Project" if it isn't. You'll see this in use a little later when you build the `edit` action. For now, focus on the `new` action!

Now, running `bundle exec rspec` once more, you can see that your spec is one step closer to finishing—the field fill-in steps have passed:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   AbstractController::ActionNotFound:
     The action 'create' could not be found for ProjectsController
```

Capybara finds the label containing the "Name" text you ask for in your scenario, and fills out the corresponding field with the value we specify. Capybara has a number of ways to locate a

field, such as by the name of the corresponding label, the `id` attribute of the field, or the `name` attribute. The last two look like this:

```
fill_in "project_name", with: "Visual Studio Code"
# or
fill_in "project[name]", with: "Visual Studio Code"
```

Should we use the ID or the label?

Some argue that using the field's ID or name is a better way because these attributes don't change as often as labels may. But your tests should aim to be as human-readable as possible - when you write them, you don't want to be thinking of field IDs, you're describing the behavior at a higher level than that.

To keep things simple, you should continue using the label name.

Capybara does the same thing for the "Description" field, and then will click the button we told it to click. The spec is now complaining about a missing action called `create`. Let's fix that.

Creating the create action

To define this action, you define the `create` method underneath the `new` method in the `ProjectsController`, as in the following listing.

Listing 49. The `create` action of `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    # nothing, yet
  end
end
```

The `Project.new` method takes one argument, which is a list of attributes that will be assigned to this new `Project` object. For now, we're just calling that list `project_params`.

After you build your new `@project` instance, you call `@project.save` to save it to the `projects` table in your database. Before that happens, though, Rails will run all the data validations on the model, ensuring that it's valid. At the moment, you have no validations on the model, so it will save just fine.

The `flash` method in your `create` action is a way of passing messages to the next request, and it takes the form of a hash. These messages are stored in the session and are cleared at the completion of the next request. Here you set the `:notice` key of the flash hash to be "Project has been created" to inform the user what has happened. This message is displayed later, as is required by the final step in your feature.

The `redirect_to` method can take several different arguments - an object, or the name of a route. If an object is given, Rails inspects it to determine what route it should go to: in this case, `project_path(@project)` because the object has now been saved to the database. This method generates the path of something such as /projects/:id, where :id is the record's `id` attribute assigned by your database system. The `redirect_to` method tells the browser to begin making a new request to that path and sends back an empty response body; the HTTP status code will be a "302 Redirect", and point to the currently non-existent `show` action.

Combining redirect_to and flash

You can combine `flash` and `redirect_to` by passing the `flash` as an option to the `redirect_to`. If you want to pass a success message, you use the `notice` flash key; otherwise you use the `alert` key. By using either of these two keys, you can use this syntax:

```
redirect_to @project, notice: "Project has been created."
# or
redirect_to @project, alert: "Project has not been created."
```

If you don't wish to use either `notice` or `alert`, you must specify `flash` as a hash:

```
redirect_to @project, flash: { success: "Project has been created."}
```

If you run `bundle exec rspec` now, you'll get an error about an undefined local variable or method "project_params".

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   NameError:
     undefined local variable or method `project_params' for
     #<ProjectsController:0x007fe704e31848>
```

Where does the data we want to make a new project from, come from? They come from the `params` provided to the controller, available to all Rails controller actions.

The `params` method returns the parameters passed to the action, such as those from the form or query parameters from a URL, as a `HashWithIndifferentAccess` object. These are different from normal `Hash` objects, because you can reference a `String` key by using a matching `Symbol` and vice versa. In this case, the `params` hash looks like this:

```
{
  "authenticity_token" => "WRHnKqU...",
  "project" => {
    "name" => "Visual Studio Code",
    "description" => "Code Editing. Redefined."
  },
  "commit" => "Create Project",
  "controller" => "projects",
  "action" => "create"
}
```

You can easily see what params your controller is receiving by looking at the server logs in your terminal console. If you run your `rails server`, visit http://localhost:3000/projects/new and submit the data that your test is trying to submit, you'll see the following in the terminal:

```
Started POST "/projects" for ::1 at [timestamp]
Processing by ProjectsController#create as HTML
  Parameters: {"authenticity_token"=>"WRHnKqU...",
  "project"=>{"name"=>"Visual Studio Code", "description"=>"A text editor
  for everyone"}, "commit"=>"Create Project"}
```

And the parameters are listed right there.

The authenticity_token` parameter

There's a "special" parameter in the `params` hash: authenticity_token`.

The `authenticity_token` parameter is used by Rails to validate the request is authentic. Rails generates this in `<meta>` tag on the page (using `<%= csrf_meta_tags %>` in `app/views/layouts/application.html.erb`) and also stores it in the user's session. Upon the submission of the form, it compares the value in the form with the one in the session and if they match, then the request is deemed authentic. Using `authenticity_token` mitigates CSRF attacks, and so is recommended best-practice.

All the hashes nested inside the `params` hash are also `HashWithIndifferentAccess` hashes. If you want to get the name key from the `project` hash here, you can use either `{ :name "Visual Studio Code" }[:name]`, as in a normal `Hash` object, or `{ :name "Visual Studio Code" }['name']`; you may use either the `String` or the `Symbol` version—it doesn't matter.

The first key in the `params` hash, `commit`, comes from the submit button of the form, which has the value "Create Project". This is accessible as `params[:commit]`. The second key, `action`, is one of two parameters always available; the other is `controller`. These represent exactly what their names imply: the controller and action of the request, accessible as `params[:controller]` and `params[:action]`, respectively. The final key, `project`, is, as mentioned before, a `HashWithIndifferentAccess`. It contains the fields from your form and is accessible via `params[:project]`. To access the `name` key in the `params[:project]` object, use `params[:project][:name]`, which calls the `[]` method on `params` to get the value of the `:project` key and then, on the resulting hash, calls `[]` again, this time with the `:name` key to get the name of the project passed in.

`params[:project]` has all the data we need to pass to `Project.new`, but we can't just pass it directly in. If you try to substitute `project_params` with `params[:project]` in your controller, and then run `rspec` again, you'll get the following error:

```
Failure/Error: click_button "Create Project"
ActiveModel::ForbiddenAttributesError:
  ActiveModel::ForbiddenAttributesError
```

Strong parameters

Oooh, forbidden attributes. Sounds scary. This is important: it's one form of security help that Rails gives you via a feature called strong parameters. You don't want to accept just any submitted parameters: you want to accept the ones that you want and expect, and no more. That way, someone can't mess around with your application by doing things like tampering with the form and adding new fields, before submitting it.

Change the `ProjectsController` code to add a new definition for the `project_params` method.

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    # nothing, yet
  end
end

private

def project_params
  params.require(:project).permit(:name, :description)
end
```

You now call the `require` method on your `params`, and you require that the `:project` key exists. You also allow it to have `:name` and `:description` entries - any other fields submitted will be discarded. Finally, you wrap up that logic into a method so you can use it in other actions, and you make it private so you don't expose it as some kind of weird action! We'll use this method in one other action in this controller later on, the `update` action.

With that done, run `bundle exec rspec` again, and you'll get a new error:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   AbstractController::ActionNotFound:
     The action 'show' could not be found for ProjectsController
```

The test has made it through the `create` action, followed the redirect we issued, and now it's stuck on the next request - the page we redirected to, the `show` action.

The `show` action is responsible for displaying a single record's information. Retrieving a record to display is done by default using the record's ID. You know the URL for this page will be something like `/projects/1`, but how do you get the `1` from that URL? Well, when you use resource routing, as you have done already, the `1` part of this URL is available as `params[:id]`, just as `params[:controller]` and `params[:action]` are also automatically made available by Rails. You can then use this `params[:id]` parameter in your `show` action to find a specific `Project` object. In this case, the `show` action should be showing the newly created project.

Put the code from the following listing into `app/controllers/projects_controller.rb` to set up the `show` action. Make sure it comes above the `private` declaration, or you won't be able to use it as an action!

Listing 50. The `show` action of `ProjectsController`

```
def show
  @project = Project.find(params[:id])
end
```

You pass the `params[:id]` object to `Project.find`. This gives you a single `Project` object that relates to a record in the database, which has its `id` field set to whatever `params[:id]` is. If Active Record can't find a record matching that ID, it raises an `ActiveRecord::RecordNotFound` exception.

When you rerun `bundle exec rspec spec/features/creating_projects_spec.rb`, you'll get an error telling you that the `show` action's template is missing:

```
1) Users can create new projects with valid attributes
  Failure/Error: click_button "Create Project"

  ActionController::MissingExactTemplate:
    ProjectsController#show is missing a template for request formats: text/html
```

You can create the file `app/views/projects/show.html.erb` with the following content for now to display the project's name and description:

Listing 51. app/views/projects/show.html.erb

```
<h1><%= @project.name %></h1>

<p><%= @project.description %></p>
```

It's a pretty plain page for a project, but it will serve our purpose. When you run the test again with `bundle exec rspec spec/features/creating_projects_spec.rb`, you see this message:

```
1) Users can create new projects with valid attributes
   Failure/Error: expect(page).to have_content "Project has been
   created."
     expected to find text "Project has been created." in "Sublime
     Text 3 Code Editing. Redefined"
   # ./spec/features/creating_projects_spec.rb:13:in ...
```

This error message shows that the "Project has been created." text isn't being displayed on the page. Therefore, you must put it somewhere, but where?

The application layout

The best location is in the application layout, located at `app/views/layouts/application.html.erb`. This file provides the layout for all templates in your application, so it's a great spot to output a flash message - no matter what controller we set it in, it will be rendered on the page.

The application layout is quite the interesting file:

Listing 52. app/views/layouts/application.html.erb

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

    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

The first line sets up the doctype to be HTML for the layout, and three new methods are used: `stylesheet_link_tag`, `javascript_pack_tag`, and `csrf_meta_tags`.

The `stylesheet_link_tag` and `javascript_pack_tag` methods include CSS and JavaScript assets for our application.

The `csrf_meta_tags` is for protecting your forms from cross-site request forgery (CSRF[29]) attacks. These types of attacks were mentioned a short while ago when we looked at the parameters for the `create` action; when we were talking about the `authenticity_token` parameter. The `csrf_meta_tags` helper creates two `meta` tags, one called `csrf-param` and the other `csrf-token`. This unique token works by setting a specific key on forms that is then sent back to the server. The server checks this key, and if the key is valid, the form is deemed valid. If the key is invalid, an `ActionController::InvalidAuthenticityToken` exception occurs and the user's session is reset as a precaution.

Later in `app/views/layouts/application.html.erb` is the single line:

```
<%= yield %>
```

This line indicates to the layout where the current action's template is to be rendered. Create a new line just before `<%= yield %>`, and place the following code there:

```
<% flash.each do |key, message| %>
  <div><%= message %></div>
<% end %>
```

This code renders all the `flash` messages that are defined, regardless of their name and the controller they come from. These lines will display the `flash[:notice]` that you set up in the `create` action of the `ProjectsController`. Run `bundle exec rspec` again, and see that the test is now fully passing:

```
3 examples, 0 failures, 2 pending
```

Why do you have two pending tests? If you examine the output more closely, you'll see this:

```
.**

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) ProjectsHelper add some examples to (or delete)
     .../ticketee/spec/helpers/projects_helper_spec.rb
     # Not yet implemented
     # ./spec/helpers/projects_helper_spec.rb:14

  2) Project add some examples to (or delete)
     .../ticketee/spec/models/project_spec.rb
     # Not yet implemented
     # ./spec/models/project_spec.rb:4

Finished in 0.07268 seconds (files took 1.26 seconds to load)
3 examples, 0 failures, 2 pending
```

The key part is that "or delete". Let's delete those two files, because you're not using them yet:

```
$ rm spec/models/project_spec.rb
$ rm spec/helpers/projects_helper_spec.rb
```

Afterward, run `bundle exec rspec` one more time:

```
.

Finished in 0.07521 seconds (files took 1.25 seconds to load)
1 example, 0 failures
```

Yippee! You have just written your first BDD test for this application! That's all there is to it. If this process feels slow, that's how it's supposed to feel when you're new to anything. Remember when you were learning to drive a car? You didn't drive like Michael Schumacher as soon as you got behind the wheel. You learned by doing it slowly and methodically. As you progressed, you were able to do it more quickly, as you can all things with practice.

3.4.3. Committing changes

Now you're at a point where all (just the one for now) your specs are running. Points like this are great times to make a commit:

```
$ git add .
$ git commit -m "'Create a new project' feature complete."
```

You should commit often, because commits provide checkpoints you can revert back to if anything goes wrong. If you're going down a path where things aren't working, and you want to get back to the last commit, you can safely store all your changes by running:

```
$ git stash
```

Then you can re-apply those changes if you wish with:

```
$ git stash pop
```

If you want to completely forget about those changes and remove them, use:

```
$ git checkout
```

Use `git checkout .` carefully!

This command doesn't prompt you to ask whether you're sure you want to take this action. You should be incredibly sure that you want to destroy your changes. If you're not sure and want to keep your changes while reverting back to the previous revision, it's best to use the `git stash` command. This command stashes your unstaged changes to allow you to work on a clean directory, and lets you restore the changes using `git stash pop`.

With the changes committed to your local repository, you can push them off to the GitHub servers. If for some reason the code on your local machine goes missing, you have GitHub as a backup. Run this command to push the code up to GitHub's servers:

```
$ git push
```

Commit early. Commit often.

3.4.4. Setting a page title

Before you completely finish working with this story, there is one more thing to point out: the templates (such as `show.html.erb`) are rendered before the layout. You can use this to your benefit by setting an instance variable such as `@title` in the `show` action's template; then you can reference it in your application's layout to show a title for your page at the top of the tab or window.

To test that the page title is correctly implemented, add a little bit extra to your scenario for it. At the bottom of the test in `spec/features/creating_projects_spec.rb`, add these four lines:

Listing 53. spec/features/creating_projects_spec.rb

```
project = Project.find_by!(name: "Visual Studio Code")
expect(page.current_url).to eq project_url(project)

title = "Visual Studio Code - Projects - Ticketee"
expect(page).to have_title title
```

The first line here uses the `find_by!` method to find a project by its name. This finds the project that has just been created by the code directly above it. If the project cannot be found,

an exception will be raised: an `ActiveRecord::RecordNotFound` exception. If you see one of these, make sure both places where you're using your project's name are the same name.

The second line ensures that you're on what should be the `show` action in the `ProjectsController`. The third and fourth lines finds the `title` element on the page by using Capybara's `find` method and checks using `have_title` that this element contains the page title of "Visual Studio Code - Projects - Ticketee". If you run `bundle exec rspec spec/features/creating_projects_spec.rb` now, you'll see this error:

```
1) Users can create new projects with valid attributes
   Failure/Error: expect(page).to have_title title
     expected "Ticketee" to include "Visual Studio Code - Projects -
     Ticketee"
```

This error is happening because the `title` element doesn't contain all the right parts, but this is fixable! Write this code into the top of `app/views/projects/show.html.erb`:

```
<% @title = "Visual Studio Code - Projects - Ticketee" %>
```

This sets up a `@title` instance variable in the template. Because the template is rendered before the layout, you're able to then use this variable in the layout. But if a page doesn't have a `@title` variable set, there should be a default title of "Ticketee". To do this, enter the following code in `app/views/layouts/application.html.erb` where the `title` tag currently is:

```
<title><%= @title || "Ticketee" %></title>
```

In Ruby, instance variables that aren't set return `nil` as their value. If you try to access an instance variable that returns a `nil` value, you can use `||` to return a different value, as in this example.

With this in place, the test should pass when you run `bundle exec rspec`:

```
1 example, 0 failures
```

Now that this test passes, you can change your code and have a solid base to ensure that whatever you change works as you expect. To demonstrate this point, you'll change the code

in `show` to use a helper instead of setting a variable.

Helpers are methods you can define in the files in `app/helpers`, and they're made available in your views. Helpers are for extracting the logic from the views; views should just be about displaying information. Every controller that comes from the controller generator has a corresponding helper, and another helper module exists for the entire application: the `ApplicationHelper` module, that lives at `app/helpers/application_helper.rb`. Open `app/helpers/application_helper.rb`, and insert the code from the following listing.

Listing 54. app/helpers/application_helper.rb

```
module ApplicationHelper
  def title(*parts)
    unless parts.empty?
      content_for :title do
        (parts << "Ticketee").join(" - ")
      end
    end
  end
end
```

When you specify an argument in a method beginning with the splat operator (`*`), any arguments passed from this point will be available in the method as an array. Here that array can be referenced as `parts`. Inside the method, you check to see if `parts` is `empty?` by using keyword that's the opposite of `if`: `unless`. If no arguments are passed to the `title` method, `parts` will be empty and therefore `empty?` will return `true`.

If parts are specified for the `title` method, then you use the `content_for` method to define a named block of content, giving it the name `"title"`. Inside this content block, you join the parts together using a hyphen (-), meaning this helper will output something like `"Visual Studio Code - Projects - Ticketee"`.

So this helper method will build up a text string that we can use as the title of any page, including the default value of "Ticketee", and all we need to do it call it from the view with the right arguments - an array of the parts that will make up the title of the page. Neat.

Now you can replace the title line in `app/views/projects/show.html.erb` with this:

```
<% title @project.name, "Projects" %>
```

Let's replace the `title` tag line in `app/views/layouts/application.html.erb` with this code:

```
<title>
  <% if content_for?(:title) %>
    <%= yield :title %>
  <% else %>
    Ticketee
  <% end %>
</title>
```

This code uses a new method called `content_for?`, which checks that the specified content block is defined. It is defined only if `content_for(:title)` is called somewhere, such as the template. If it is, you use `yield` and pass it the name of the content block, which causes the content for that block to be rendered. If it isn't, then you output the word "Ticketee", and that becomes the title.

When you run this test again with `bundle exec rspec spec/features/creating_projects_spec.rb`, it will still pass:

```
1 example, 0 failures
```

That's a lot neater, isn't it? Let's create a commit for that functionality and push your changes:

```
$ git add .
$ git commit -m "Add title functionality for project show page"
$ git push
```

Next up, we look at how to stop users from entering invalid data into your forms.

3.4.5. Validations

The next problem to solve is preventing users from leaving a required field blank. A project with no name isn't useful to anybody. Thankfully, Active Record provides validations for this issue. Validations are run just before an object is saved to the database, and if the validations fail, then the object isn't saved. Ideally, in this situation, you want to tell the user what went wrong so they can fix it and attempt to create the project again.

We saw validations back in Chapter 1 too: we validated that a purchase had to have a name, as well as a cost that was above 0. In this chapter, we'll just validate that a project's name is

present.

With this in mind, let's add another test for ensuring that this happens to `spec/features/creating_projects_spec.rb` using the code from the next listing.

Listing 55. spec/features/creating_projects_spec.rb

```
scenario "when providing invalid attributes" do
  visit "/"

  click_link "New Project"
  click_button "Create Project"

  expect(page).to have_content "Project has not been created."
  expect(page).to have_content "Name can't be blank"
end
```

The first two lines are identical to the ones you placed in the other scenario. You should eliminate this duplication by making your code DRY (Don't Repeat Yourself!). This is another term you'll hear a lot in the Ruby world.[30] It's easy to extract common code from where it's being duplicated, and into a method or a module you can use instead of the duplication. One line of code is 100 times better than 100 lines of duplicated code.

To DRY up your code, before the first scenario, you can define a `before` block. For RSpec, `before` blocks are run before every test in the file. Change `spec/features/creating_projects_spec.rb` to look like this.

Listing 56. spec/features/creating_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new projects" do
  before do
    visit "/"

    click_link "New Project"
  end

  scenario "with valid attributes" do
    fill_in "Name", with: "Visual Studio Code"
    fill_in "Description", with: "Code Editing. Redefined"
    click_button "Create Project"

    expect(page).to have_content "Project has been created."

    project = Project.find_by(name: "Visual Studio Code")
    expect(page.current_url).to eq project_url(project)

    title = "Visual Studio Code - Projects - Ticketee"
    expect(page).to have_title title
  end

  scenario "when providing invalid attributes" do
    click_button "Create Project"

    expect(page).to have_content "Project has not been created."
    expect(page).to have_content "Name can't be blank"
  end
end
```

There! That looks a lot better! Now when you run `bundle exec rspec`, it will fail because it can't see the error message that it's expecting to see on the page:

```
1) Users can create new projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Project has not been
   created."
     expected to find text "Project has not been created." in "Project
     has been created."
```

Adding validations

To get this test to do what you want it to do, you'll need to add a validation. Validations are

defined on the model and are run before the data is saved to the database. To define a validation to ensure that the `name` attribute is provided when a project is created, open the `app/models/project.rb` file and make it look like the following listing.

Listing 57. app/models/project.rb

```
class Project < ApplicationRecord
  validates :name, presence: true
end
```

The `validates` method's usage is exactly how you used it for the first time in chapter 1. It tells the model that you want to validate the `name` field, and that you want to validate its presence. There are other kinds of validations as well; for example, the `:uniqueness` key, when passed `true` as the value, validates the uniqueness of this field as well, ensuring that only one record in the table has that specific value.[31]

With the `presence` validation in place, you can experiment with the validation by using the Rails console, which allows you to have all the classes and the environment from your application loaded in a sandbox environment. You can launch the console with this command:

```
$ rails console
```

or with its shorter alternative:

```
$ rails c
```

If you're familiar with Ruby, you may realize that this is effectively IRB with some Rails sugar on top. For those of you new to both, IRB stands for **I**nteractive **R**u**b**y, and it provides an environment for you to experiment with Ruby without having to create new files. The console prompt looks like this:

```
Loading development environment (Rails {rails_version})
irb(main):001:0>
```

At this prompt[32], you can enter any valid Ruby, and it'll be evaluated. But for now, the purpose of opening this console was to test the newly appointed validation. To do this, try to create a new project record by calling the `create` method. The `create` method is similar to the `new` method, but it attempts to create an object and then a database record for it rather than

just the object. You use it identically to the `new` method:

```
irb(main):001:0> Project.create
=> #<Project id: nil, name: nil, description: nil, created_at: nil,
     updated_at: nil>
```

Here you get a new `Project` object with the `name` and `description` attributes set to `nil`, as you should expect because you didn't specify it. The `id` attribute is `nil` too, which indicates that this object isn't persisted (saved) in the database.

If you comment out or remove the validation from in the `Project` class and type `reload!` in your console, the changes you just made to the model are reloaded. When the validation is removed, you have a slightly different outcome when you call `Project.create`:

```
irb(main):001:0> Project.create
=> #<Project id: 1, name: nil, description: nil,
     created_at: [timestamp], updated_at: [timestamp]>
```

Here, the `name` field is still expectedly `nil`, but the other three attributes have values. Why? When you call `create` on the `Project` model, Rails builds a new `Project` object with any attributes you pass it[33] and checks to see if that object is valid. If it is, Rails sets the `created_at` and `updated_at` attributes to the current time and then saves the object to the database. After it's saved, the `id` is returned from the database and set on your object. This object is valid, according to Rails, because you removed the validation, and therefore Rails goes through the entire process of saving.

The `create` method has a bigger, meaner brother called `create!` (pronounced create **BANG!**). Re-add or uncomment the validation from the model, and type `reload!` in the console, and you'll see what this mean variant does with this line:

```
irb(main):001:0> Project.create!
ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

The `create!` method, instead of nonchalantly handing back a `Project` object regardless of any validations, raises an `ActiveRecord::RecordInvalid` exception if any of the validations fail; it shows the exception followed by a large stacktrace, which you can safely ignore for now. You're notified which validation failed. To stop it from failing, you must pass in a `name` attribute, and `create` will happily return a saved `Project` object:

```
irb(main):002:0> Project.create!(name: "Visual Studio Code")
=> #<Project id: 2, name: "Visual Studio Code", description: nil,
     created_at: [timestamp], updated_at: [timestamp]>
```

That's how to use `create` to test your validations in the console. We've created some bad data in our database during our experimentation, we should clean that up before we continue.

```
irb(main):003:0> Project.delete_all
=> 2
```

Back in your `ProjectsController`, we're using the method shown in the following listing instead.

Listing 58. Part of the `create` action of `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    ...
```

If the validations pass, `save` here will return `true`. You can use this to your advantage to show the user an error message when this returns `false` by using it in an `if` statement. Make the `create` action in the `ProjectsController` look like the following listing.

Listing 59. The new `create` action from `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    flash.now[:alert] = "Project has not been created."
    render "new"
  end
end
```

`flash` **vs.** `flash.now`

The above controller action uses two different methods to access the array of flash messages for your page - `flash` and `flash.now`. What's the difference?

`flash` is the standard way of setting flash messages, and will store the message to display on the very next page load. We do this immediately before issuing redirects - in this case we are redirecting immediately to the `show` page in the `ProjectsController`, and that page is the next page load, meaning the flash message displays on the `show` view.

`flash.now` is an alternate way of setting flash messages, and will store the message to display on the current page load. In this case, we're not redirecting anywhere, we're simply rendering a view out from the same action, so we need to use `flash.now` to make sure the user sees the error message when we render the `new` view.

There's also a third method - `flash.keep` - but this is used very rarely. If you want to keep an existing flash message around for another request, you can call `flash.keep` in your controller, and the flash message will hang around for a little while longer.

If you were to use `flash` instead of `flash.now` in this case, the user would actually see the message twice - once on the current page and once on the next page!

Now, if the `@project` object has a `name` attribute—meaning it's valid—`save` returns `true` and executes everything between `if` and `else`. If it isn't valid, then everything between `else` and the following `end` is executed. In the `else`, you specify a different key for the flash message because you'll want to style alert messages differently from notices later in the application's lifecycle. When good things happen, the messages for them will be colored with a green background. When bad things happen, red.

When you run `bundle exec rspec spec/features/creating_projects_spec.rb` here, the line in the spec that checks for the "Project has not been created." message now doesn't fail; so, it goes to the next line, which checks for the "Name can't be blank" message. You haven't done anything to make this message appear on the page right now, which is why the test is failing again:

```
1) Users can create new projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Name can't be blank"
     expected to find text "Name can't be blank" in "Project has not
     been created. New Project Name Description"
```

The validation errors for the project aren't being displayed on this page, which is causing the test to fail. To display validation errors in the view, you need to code something up yourself.

When an object fails validation, Rails will populate the `errors` of the object with any validation errors. You can test this back in your Rails console:

```
irb(main):001:0> project = Project.create
=> #<Project id: nil, name: nil, description: nil, created_at: nil,
     updated_at: nil>
irb(main):002:0> project.errors
=> #<ActiveModel::Errors:0x007fd5938197f8 @base=#<Project id: nil,
     name: nil, description: nil, created_at: nil, updated_at: nil>,
     @messages={:name=>["can't be blank"]}>
```

`ActiveModel::Errors` provides some nice helper methods for working with the validation errors, that we can use in our views to display the errors back to the user. Directly under this `form_with` line, on a new line, insert the following into `app/views/projects/new.html.erb` to display the error messages in the form.

Listing 60. app/views/projects/new.html.erb

```
<%= form_with(model: @project, local: true) do |form| %>
  <% if @project.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(@project.errors.count, "error") %>
      prohibited this project from being saved:</h2>

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

  ...
<% end %>
```

Error messages for the object represented by your form, the `@project` object, will now be displayed by `each`. When you run `bundle exec rspec`, you get this output:

```
2 examples, 0 failures
```

Commit and push, and then you're done with this story!

```
$ git add .
$ git commit -m "Add validation to ensure names are specified when
  creating projects"
$ git push
```

3.5. Takeaways

3.5.1. Behavior Driven Development

Behavior Driver Development is the process of writing tests asserting that our application will have certain behaviour, before we write code. These tests provide us with an automated way of testing our application, providing constant assurances that our application is working as it should.

Some people might argue that this is extra work; that it isn't contributing anything valuable. We disagree. Having a quick, automatic way to ensure that the application is working as intended throughout the application's life is extremely valuable. It prevents mistakes that may change the behaviour of the application from being introduced.

An application with tests, is more maintainable than an application without.

3.5.2. Version Control

In this chapter, we have used Git throughout to store our code's changes.

This allows us to have checkpoints where our code is at known-working states where all our tests are passing. These are valuable, because it helps break our work into small atomic chunks.

Further to this, if we happen to develop down the wrong path long enough and discover that later on, we can run `git checkout .` to reset our code back to the latest known-working state

3.5.3. You can build Rails apps without scaffold

In Chapter 1, we used the scaffold generator to quickly build a part of our application. However, this is not how Rails developers do it in the real world.

In the real world, Rails developers write tests, and then incrementally build up their features as we've seen in this chapter. As you practice this technique, you will get faster at it. For example, you'll know that if you have a route that you'll need a controller and an action to go along with that. You might just go ahead and build those, rather than waiting for your tests to tell you that those are needed.

While the scaffold generator gives you a lot, we can take the training wheels off now and develop our application without relying on Rails to do it for us.

3.6. Summary

We first covered how to version-control an application, which is a critical part of the application development cycle. Without proper version control, you're liable to lose valuable work or be unable to roll back to a known working stage. We used Git and GitHub as examples, but you may use an alternative—such as SVN or Mercurial—if you prefer. This book covers only Git, because covering everything would result in a multivolume series, which is difficult to transport.

Next we covered the basic setup of a Rails application, which started with the `rails new` command that initializes an application. Then we segued into setting up the `Gemfile` to require certain gems for certain environments, such as RSpec in the test environment. You learned about the beautiful Bundler gem in the process, and then you ran the installers for these gems so your application was fully configured to use them. For instance, after running `rails g rspec:install`, your application was set up to use RSpec and so will generate RSpec specs rather than the default Test::Unit tests for your models and controllers.

Finally, you wrote the first story for your application, which involved generating a controller and a model as well as an introduction to RESTful routing and validations. With this feature of your application covered by RSpec, you can be notified if it's broken by running `bundle exec rspec`, a command that runs all the tests of the application and lets you know if everything is working or if anything is broken. If something is broken, the spec will fail, and then it's up to you to fix it. Without this automated testing, you would have to do it all manually, and that isn't any fun.

Now that you've got a first feature under your belt, let's get into writing the next one!

[19] You can find more information about Agile on Wikipedia: http://en.wikipedia.org/wiki/Agile_software_development
[20] Hey, at least we thought it was funny!
[21] BitBucket (http://bitbucket.org) is a popular alternative, and it also allows you to have free private repositories.
[22] The chances of this happening are 1 in 268,435,456.
[23] The `~>` operator is called the approximate version constraint or by its slang term, the twiddle-wakka.
[24] By generating RSpec tests rather than Test::Unit tests, a longstanding issue in Rails has been fixed. In previous versions of Rails, even if you specified the RSpec gem, all the default generators still generated Test::Unit tests. With Rails, the testing framework you use is just one of a large number of configurable things in your application.
[25] See http://en.wikipedia.org/wiki/Representational_state_transfer.
[26] Although it's possible to perform database operations without a model in Rails, 99% of the time you'll want to use a model.
[27] Rails actually does know how to reverse the removal of a column, if you provide an extra field type argument to `remove_column`, eg. `remove_column :projects, :name, :string`. We'll leave this here for demonstration purposes though.
[28] Yes, Co-ordinated Universal Time has an initialism of "UTC". This is what happens when you name things by committee. http://en.wikipedia.org/wiki/Coordinated_Universal_Time#Etymology
[29] http://en.wikipedia.org/wiki/CSRF
[30] Some people like to use 'DRY' like an adjective, and also refer to code that isn't DRY as WET (which doesn't actually stand for anything). We think those people are a bit weird.
[31] There are potential gotchas with the ActiveRecord uniqueness validation, that may allow duplicate data to be saved to the database. We're intentionally ignoring them for now, but we'll be covering these, and how to resolve the issues they raise, later on.
[32] Although you may see something similar to `ruby-2.7.1:001 >` too, which is fine.
[33] The first argument for this method is the attributes. If no argument is passed, then all attributes default to their default values.

Chapter 4. Oh, CRUD!

In chapter 3, you began writing stories for a CRUD (create, read, update, delete) interface for your `Project` resource. Here you continue in that vein, beginning with writing a story for the r part of CRUD: reading. We often refer to reading as viewing in this and future chapters—we mean the same thing, but sometimes viewing is a better word.

For the remainder of the chapter, you'll round out the CRUD interface for projects, providing your users with ways to edit, update, and delete projects too. Best of all, you'll do this using behavior-driven development (BDD) the whole way through, continuing your use of the RSpec and Capybara gems that we saw in use in the last chapter. This chapter's length is testament to exactly how quickly you can get some CRUD actions up and running on a resource with Ruby on Rails.

Also in this chapter, you'll see a way to create test data extremely easily for your tests, using a gem called `factory_bot`, as well as a way to make standard controllers a lot neater.

4.1. Viewing projects

The `show` action generated for the story in chapter 3 was only half of this part of CRUD. The other part is the `index` action, which is responsible for showing a list of all of the projects. From this list, you can navigate to the `show` action for a particular project. The next story is about adding functionality to allow you to do that.

Create a new file in the features directory called `spec/features/viewing_projects_spec.rb`, shown in the following listing.

Listing 61. spec/features/viewing_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can view projects" do
  scenario "with the project details" do
    project = FactoryBot.create(:project, name: "Visual Studio Code")

    visit "/"
    click_link "Visual Studio Code"
    expect(page.current_url).to eq project_url(project)
  end
end
```

To run this single test, you can use `bundle exec rspec spec/features/viewing_projects_spec.rb`. When you do this, you'll see the following failure:

```
1) Users can view projects with the project details
   Failure/Error: project = FactoryBot.create(:project,
   name: "Visual Studio Code")
   NameError:
     uninitialized constant FactoryBot
```

The `FactoryBot` constant is defined by another gem: the `factory_bot` gem.

4.1.1. Introducing Factory Bot

The `factory_bot` gem, created by thoughtbot (http://thoughtbot.com), provides an easy way to use factories to create new objects for your tests. Factories define a bunch of default values for an object, allowing you to easily craft example objects you can use in your tests.

Before you can use this gem, you need to add it to the `:test` group in your `Gemfile`. Now the entire group looks like this:

```
group :test do
  gem "capybara", ">= 3.2.6"
  gem "factory_bot_rails", "~> 6.1"
end
```

To install the gem, run `bundle`. With the `factory_bot_rails` gem installed, the `FactoryBot` constant is now defined. Run `bundle exec rspec spec/features/viewing_projects_spec.rb` again, and you'll see a new error:

```
1) Users can view projects with the project details
   Failure/Error: project = FactoryBot.create(:project,
   name: "Visual Studio Code")
   ArgumentError:
     Factory not registered: project
```

When using Factory Bot, you must create a factory for each model you wish to use the gem with. If a factory isn't registered with Factory Bot, you'll get the previous error. To register/create a factory, create a new directory in the `spec` directory called `factories`, and

then in that directory create a new file called `projects.rb`. Fill that file with the content from the following listing.

Listing 62. spec/factories/projects.rb

```
FactoryBot.define do
  factory :project do
    name { "Example project" }
  end
end
```

When you define the factory in this file, you give it a `name` attribute, so that every new project generated by the factory via `FactoryBot.create :project` will have the name "Example project". The `name: "Visual Studio Code"` part of this method call in `spec/features/viewing_projects_spec.rb` changes the name for that instance to the one passed in. You use factories here because you don't need to be concerned about any other attribute on the `Project` object. If you weren't using factories, you'd just create the project the way you would anywhere else, like in the console:

```
Project.create(name: "Visual Studio Code")
```

Although this code is about the same length as its `FactoryBot.create` variant, it isn't future proof. If you were to add another field to the `projects` table and add a validation (say, a presence one) for that field, you'd have to change all occurrences of the `create` method in your tests to contain this new field. Over time, as your `Project` model gets more and more attributes, the actual information that you care about—in this case, the name—would get lost in amongst all of the unrelated data. When you use a factory, you can change it in one place: where the factory is defined. If you cared about what that field was set to, you could modify it by passing it as one of the key-value pairs in the `Factory` call.

That's a lot of theory—how about some practice? Let's see what happens when you run `bundle exec rspec spec/features/viewing_projects_spec.rb` again:

```
1) Users can view projects with the project details
   Failure/Error: click_link "Visual Studio Code"
   Capybara::ElementNotFound:
     Unable to find link "Visual Studio Code"
```

A link appears to be missing. You'll add that right now.

4.1.2. Adding a link to a project

Capybara is expecting a link on the page with the words "Visual Studio Code", but can't find it. The page in question is the homepage, which is the `index` action from your `ProjectsController`. Capybara can't find it because you haven't yet put it there, which is what you'll do now. Open `app/views/projects/index.html.erb` and change the contents to the following:

Listing 63. app/views/projects/index.html.erb

```
<h1>Projects</h1>

<%= link_to "New Project", new_project_path %>

<div class="projects">
  <% @projects.each do |project| %>
    <h2><%= link_to project.name, project %></h2>
    <p><%= project.description %></p>
  <% end %>
</div>
```

We've added a heading, and some details on each of the projects. If you run the spec again, you get this error, which isn't helpful at first glance:

```
1) Users can view projects with the project details
   Failure/Error: visit "/"
   ActionView::Template::Error:
     undefined method `each' for nil:NilClass
   # ./app/views/projects/index.html.erb:6:in ...
```

This error points at line 6 of your `app/views/projects/index.html.erb` file, which reads `<% @projects.each do |project| %>`. From this you can determine that the error must have something to do with the `@projects` variable. This variable hasn't yet been defined, and as mentioned in Chapter 3, instance variables in Ruby return `nil` rather than raise an exception. So because `@projects` is `nil`, and there's no `each` method on `nil`, you get this error `undefined method 'each' for nil:NilClass`. Watch out for this in Ruby—as you can see here, it can sting you hard.

We need to define this variable in the `index` action of our controller. Open `ProjectsController` at `app/controllers/projects_controller.rb`, and change the `index` method definition to look like this.

Listing 64. `index` **action of** `ProjectsController`

```
def index
  @projects = Project.all
end
```

By calling `all` on the `Project` model, you retrieve all the records from the database as `Project` objects, and they're available as an enumerable `Array`-like object. Now that you've put all the pieces in place, you can run the feature with `bundle exec rspec spec/features/viewing_projects_spec.rb`, and it should pass:

```
1 example, 0 failures
```

The spec now passes. Is everything else still working, though? You can check by running `bundle exec rspec`. Rather than just running the one test, this code runs all the tests in the spec directory. When you run the code, you should see this:

```
3 examples, 0 failures
```

All the specs are passing, meaning all the functionality you've written so far is working as it should. Commit your changes and push them to GitHub, using these commands:

```
$ git add .
$ git commit -m "Add the ability to view a list of all projects"
$ git push
```

The reading part of this CRUD resource is done! You've got the `index` and `show` actions for the `ProjectsController` behaving as they should. Now you can move on to updating.

4.2. Editing projects

With the first two parts of CRUD (creating and reading) done, you're ready for the third part: updating. Updating is similar to creating and reading in that it has two actions for each part (creation has `new` and `create`; reading has `index` and `show`). The two actions for updating are `edit` and `update`. Let's begin by writing a feature and creating the `edit` action.

4.2.1. The edit action

As with the form used for creating new projects, you want a form that allows users to edit the information of a project that already exists. You first put an "Edit Project" link on the `show` page, that takes users to the `edit` action where they can edit the project. Write the code from the following listing into `spec/features/editing_projects_spec.rb`.

Listing 65. spec/features/editing_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can edit existing projects" do
  scenario "with valid attributes" do
    FactoryBot.create(:project, name: "Visual Studio Code")

    visit "/"
    click_link "Visual Studio Code"
    click_link "Edit Project"
    fill_in "Name", with: "VS Code"
    click_button "Update Project"

    expect(page).to have_content "Project has been updated."
    expect(page).to have_content "VS Code"
  end
end
```

If you remember, `FactoryBot#create` builds you an entire object and lets you tweak the defaults. In this case, you're changing the name.

Also, it's common for tests to take this overall form: arrange, act, assert. (This is also referred to as 'given', 'when', and 'then', to describe the actions that take place in each section.) That's why the whitespace is there: it clearly splits the test. Your tests won't always look like this, but it's good form.

After writing this story, again use the `rspec` command to run just this one feature: `bundle exec rspec spec/features/editing_projects_spec.rb`. The first couple of lines for this scenario pass because of the work you've already done, but it fails on the line that attempts to find the "Edit Project" link:

```
1) Users can edit existing projects with valid attributes
   Failure/Error: click_link "Edit Project"
   Capybara::ElementNotFound:
     Unable to find link "Edit Project"
```

To add this link, open `app/views/projects/show.html.erb` and add the link under the heading for the project name:

```
<h1><%= @project.name %></h1>

<%= link_to "Edit Project", edit_project_path(@project) %>
```

The `edit_project_path` method generates a link pointing towards the `edit` action of the `ProjectsController`. This method is provided to you because of the `resources :projects` line in `config/routes.rb`.

If you run `bundle exec rspec spec/features/editing_projects_spec.rb` again, it now complains about the missing `edit` action:

```
1) Users can edit existing projects with valid attributes
   Failure/Error: click_link "Edit Project"
   AbstractController::ActionNotFound:
     The action 'edit' could not be found for ProjectsController
```

Define this action in your `ProjectsController`, under the `show` action (but above the `private` line), as in the following listing.

Listing 66. app/controllers/projects_controller.rb

```
def edit
  @project = Project.find(params[:id])
end
```

As you can see, this action works in a fashion identical to the `show` action, where the ID for the resource is automatically passed as `params[:id]`. Let's work on DRYing[34] this up once you're done with this controller. When you run the spec again, you're told that the `edit` view is missing:

```
1) Users can edit existing projects with valid attributes
  Failure/Error: click_link "Edit Project"

  ActionController::MissingExactTemplate:
    ProjectsController#edit is missing a template for request formats: text/html
```

It looks like you need to create this template. The `edit` action's form is similar to the form in the `new` action. If only there were a way to extract out just the form into its own template. Well, in Rails, there is! You can extract out the form from `app/views/projects/new.html.erb` into what's called a partial. We saw these briefly in Chapter 1.

A partial is a template that contains some code that can be shared between other templates. To extract the form from the `new` template into a new partial, take this code out of `app/views/projects/new.html.erb`:

Listing 67. app/views/projects/new.html.erb

```
<%= form_with(model: @project, local: true) do |form| %>
  <% if @project.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@project.errors.count, "error") %>
    prohibited this project from being saved:</h2>

    <ul>
      <% @project.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
  <% end %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </duv>

  <div>
    <%= form.label :description %>
    <%= form.text_field :description %>
  </div>

  <%= form.submit %>
<% end %>
```

This will leave the `new` template looking pretty bare, with just the heading. Then create a new

file called `app/views/projects/_form.html.erb` and put into it the code you just extracted from the `new` template. While moving it, you should also change all instances of `@project` to be `project` instead.

Listing 68. app/views/projects/_form.html.erb

```
<%= form_with(model: project, local: true) do |form| %>
  <% if project.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(project.errors.count, "error") %>
    prohibited this project from being saved:</h2>

    <ul>
      <% project.errors.full_messages.each do |msg| %>
        <li><%= msg %></li>
      <% end %>
    </ul>
  </div>
  <% end %>
  <div>
    <%= form.label :name %>
    <%= form.text_field :name %>
  </div>

  <div>
    <%= form.label :description %>
    <%= form.text_area :description %>
  </div>

  <%= form.submit %>
<% end %>
```

The filenames of partials must always start with an underscore, which is why we've written `_form` instead of `form`.

Why have we done this variable renaming? Because to be reusable, partial views shouldn't rely on instance variables - they should be totally self-sufficient. When we render the partial from our main view, we can pass in the data that the partial needs to render - in this case, we'll pass in the `@project` variable from the `new` template, which means it will be accessible as a local variable from within the partial.

To render the partial and pass in the `@project` instance, modify your `new` template in `app/views/projects/new.html.erb` and add this line where the form previously was:

```
<%= render "form", project: @project %>
```

This will leave your `new` template very slim indeed:

Listing 69. The complete `new` template

```
<h1>New Project</h1>
<%= render "form", project: @project %>
```

This is an example of a refactoring - changing the internals of code without affecting functionality. You can confirm that this refactoring has not affected the ability to create projects, by running the test for it - `bundle exec rspec spec/features/creating_projects_spec.rb`. It will still pass, so you can be very confident that everything still works.

Now you need to create the `edit` action's template. Create a new file at `app/views/projects/edit.html.erb` with this content.

Listing 70. app/views/projects/edit.html.erb

```
<h1>Edit Project</h1>
<%= render "form", project: @project %>
```

When you pass a string to the `render` method, Rails looks up a partial in the same directory as the current template matching the string and renders that instead. Using the partial, the next line passes without any further intervention from you when you run `bundle exec rspec spec/features/editing_projects_spec.rb`.

```
1) Users can edit existing projects with valid attributes
   Failure/Error: click_button "Update Project"
   AbstractController::ActionNotFound:
     The action 'update' could not be found for ProjectsController
```

The test has filled in the "Name" field successfully; but it fails when the "Update Project" button is clicked, because it can't find the `update` action in the `ProjectsController`. To make this work, you'll need to create that `update` action.

4.2.2. The update action

As the following listing shows, you can now define `update` under the `edit` action in your controller. Make sure to do this above the `private` keyword too!

Listing 71. app/controllers/projects_controller.rb

```
def update
  @project = Project.find(params[:id])
  @project.update(project_params)

  flash[:notice] = "Project has been updated."
  redirect_to @project
end
```

Notice the new method on `@project` here, `update`. It takes a hash of attributes identical to the ones passed to `new` or `create`, updates those specified attributes on the object, and then saves them to the database if they're valid. This method, like `save`, returns `true` if the update is valid or `false` if it isn't.

Now that you've implemented the `update` action, let's see how the test is going by running `bundle exec rspec spec/features/editing_projects_spec.rb`:

```
1 example, 0 failures
```

That was easy! But what happens if somebody fills in the "Name" field with a blank value? The user should receive an error, just as in the `create` action, due to the validation in the `Project` model. You should write a test to verify this behaviour.

Move the first four steps from the first scenario in `spec/features/editing_projects_spec.rb` into a `before` block, because when a user is editing a project, the first four steps will always be the same: a project needs to exist, and then a user goes to the homepage, finds a project, and clicks "Edit Project". Change `spec/features/editing_projects_spec.rb` so it looks like this.

Listing 72. spec/features/editing_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can edit existing projects" do
  before do
    FactoryBot.create(:project, name: "Visual Studio Code")

    visit "/"
    click_link "Visual Studio Code"
    click_link "Edit Project"
  end

  scenario "with valid attributes" do
    fill_in "Name", with: "Visual Studio Code Nightly"
    click_button "Update Project"

    expect(page).to have_content "Project has been updated."
    expect(page).to have_content "Visual Studio Code Nightly"
  end
end
```

A `before` block can help set up state for multiple tests: the block runs before each test executes. Sometimes, setting up is more than just creating objects; interacting with an application is totally legitimate as part of setup.

Defining behaviour for when an update fails

Now you can add a new scenario, shown in the following listing, to test that the user is shown an error message for when the validations fail during the `update` action. Add this new scenario directly under the one currently in this file.

Listing 73. spec/features/editing_projects_spec.rb

```
scenario "when providing invalid attributes" do
  fill_in "Name", with: ""
  click_button "Update Project"

  expect(page).to have_content "Project has not been updated."
end
```

When you run `bundle exec rspec spec/features/editing_projects_spec.rb`, filling in "Name" works, but when the form is submitted, the test doesn't see the "Project has not been updated." message:

```
1) Users can edit existing projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Project has not been
   updated."
     expected to find text "Project has not been updated." in "Project
     has been updated. Visual Studio Code Edit Project"
```

The test can't find the message on the page because you haven't written any code to test for what to do if the project being updated is now invalid. In your controller, use the code in the following listing for the `update` action so that it shows the error message if the `update` method returns `false`.

Listing 74. The `update` action of `ProjectsController`

```
def update
  @project = Project.find(params[:id])

  if @project.update(project_params)
    flash[:notice] = "Project has been updated."
    redirect_to @project
  else
    flash.now[:alert] = "Project has not been updated."
    render "edit"
  end
end
```

And now you can see that the feature passes when you rerun `bundle exec rspec spec/features/editing_projects_spec.rb`:

```
2 examples, 0 failures
```

Again, you should ensure that everything else is still working by running `bundle exec rspec`. You should see this summary:

```
5 examples, 0 failures
```

Looks like a great spot to make a commit and push:

```
$ git add .
$ git commit -m "Projects can now be updated"
$ git push
```

The third part of CRUD, updating, is done now. The fourth and final part is deleting.

4.3. Deleting projects

We've reached the final stage of CRUD: deletion. This involves implementing the final action of your controller: the `destroy` action, which allows you to delete projects.

Of course, you'll need a feature to get going: a "Delete Project" link on the `show` page that, when clicked, prompts the user for confirmation that they really want to delete the project.[35] Put the feature at `spec/features/deleting_projects_spec.rb` using the code in the following listing.

Listing 75. spec/features/deleting_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can delete projects" do
  scenario "successfully" do
    FactoryBot.create(:project, name: "Visual Studio Code")

    visit "/"
    click_link "Visual Studio Code"
    click_link "Delete Project"

    expect(page).to have_content "Project has been deleted."
    expect(page.current_url).to eq projects_url
    expect(page).to have_no_content "Visual Studio Code"
  end
end
```

When you run this test using `bundle exec rspec spec/features/deleting_projects_spec.rb`, the first couple of lines pass because they're just creating a project using Factory Bot, visiting the homepage, and then clicking the link to go to the project page. The fourth line in this scenario fails, however, with this message:

```
1) Users can delete projects successfully
   Failure/Error: click_link "Delete Project"
   Capybara::ElementNotFound:
     Unable to find link "Delete Project"
```

To get this to work, you need to add a "Delete Project" link to the `show` action's template, `app/views/projects/show.html.erb`. Put it on the line after the "Edit Project" link, using this code:

Listing 76. app/views/projects/show.html.erb

```
<%= link_to "Delete Project",
    project_path(@project),
    method: :delete,
    data: { confirm: "Are you sure you want to delete this project?" }
%>
```

Here you pass two new options to the `link_to` method: `:method` and `:data`. The `:method` option tells Rails what HTTP method this link should be using, and here's where you specify the `:delete` method. In the previous chapter, the four HTTP methods were mentioned; the final one is `DELETE`. When you developed your first application, chapter 1 explained why you use the `DELETE` method, but let's review the reasons.

If all actions are available by `GET` requests, then anybody can create a URL that directly corresponds to the `destroy` action of your controller. If they send you the link, and you click it, then it's bye-bye precious data. By using `DELETE`, you protect an important route for your controller by ensuring that you have to follow the correct link from your site, to make the proper request to delete this resource.

The `:data` option containing a `:confirm` key brings up a prompt, using JavaScript, that asks users if they're sure that's what they want to do. If you launch a browser and follow the steps in the feature to get to this "Delete Project" link, and then you click the link, you see the confirmation prompt. This prompt is exceptionally helpful for preventing accidental deletions.

Because Capybara doesn't support JavaScript by default, the prompt is ignored, so you don't have to tell Capybara to click "OK" in response to the prompt—there is no prompt, because Rails has a built-in fallback for users without JavaScript enabled.

When you run the spec again with `bundle exec rspec spec/features/deleting_projects_spec.rb`, it complains of a missing `destroy` action:

```
1) Users can delete projects successfully
   Failure/Error: click_link "Delete Project"
   AbstractController::ActionNotFound:
     The action 'destroy' could not be found for ProjectsController
```

This is the final action you need to implement in your controller; it goes under the `update` action. The action is shown in the following listing.

Listing 77. The `destroy` action from `ProjectsController`

```
def destroy
  @project = Project.find(params[:id])
  @project.destroy

  flash[:notice] = "Project has been deleted."
  redirect_to projects_path
end
```

Here you call the `destroy` method on the `@project` object you get back from your `find` call. No validations are run, so no conditional setup is needed. Once you call `destroy` on that object, the relevant database record is gone for good; but the Ruby object representation of this record still exists until the end of the request. After the record has been deleted from the database, you set the `flash[:notice]` to indicate to the user that their action was successful, and redirect back to the projects `index` page by using the `projects_path` routing helper in combination with `redirect_to`.

With this last action in place, your newest feature should pass when you run `bundle exec rspec spec/features/deleting_projects_spec.rb`:

```
1 example, 0 failures
```

Let's see if everything else is still passing, with `bundle exec rspec`:

```
6 examples, 0 failures
```

Great! Let's commit that:

```
$ git add .
$ git commit -m "Projects can now be deleted"
$ git push
```

Done! Now you have full support for CRUD operations in your `ProjectsController`. Let's refine this controller into simpler code before we move on.

4.4. What happens when things can't be found

People sometimes poke around an application looking for things that are no longer there, or they muck about with the URL. As an example, launch your application's server by using `rails server`, and try to navigate to http://localhost:3000/projects/not-here. You'll see the exception shown here:

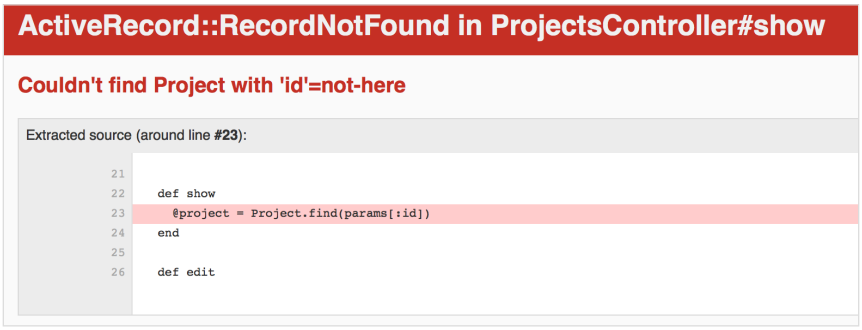


Figure 18. `ActiveRecord::RecordNotFound` **exception**

This is Rails' way of displaying exceptions in development mode. Under this error, more information is displayed, such as the backtrace of the error. Rails will only do this in the development environment because of the `consider_all_requests_local` configuration setting in `config/environments/development.rb`. This file contains all the custom settings for your development environment, and the `consider_all_requests_local` setting is `true` by default. This means Rails will show the complete exception information when it runs in the `development` environment. When this application is running under the production environment, users would instead see this:

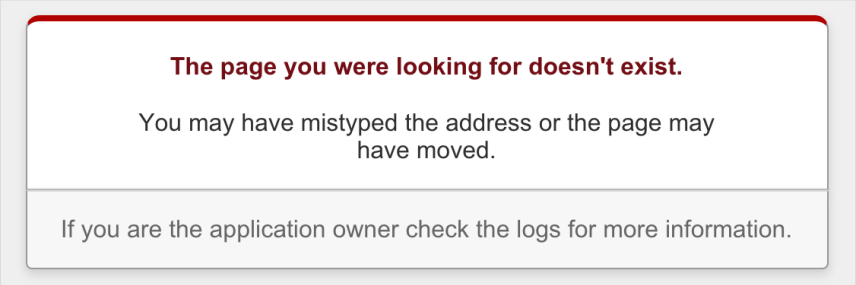
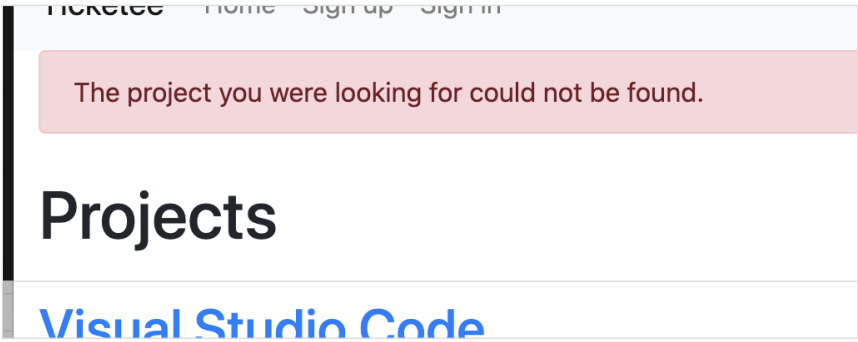


Figure 19. This is not the page you are looking for

This is not ideal! What we would rather see is a message that the project cannot be found.



4.4.1. Handling the ActiveRecord::RecordNotFound exception

To handle the `ActiveRecord::RecordNotFound` exception, you can rescue it and, rather than letting Rails render a 404 page, redirect the user to the `index` action with an error message. To test that users are shown an error message rather than a "Page does not exist" error, you'll write an RSpec request test instead of a feature test, because viewing projects that aren't there is something a user can do, but not something that should happen over the course of a normal browsing session. Plus, it's easier.

The file for this request test, `spec/requests/projects_request_spec.rb`, was automatically generated when you ran the controller generator, because you have the `rspec-rails` gem in your Gemfile.[36] Open this request spec file:

Listing 78. spec/requests/projects_request_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :request do

end
```

In this request spec, you want to test that you're redirected to the Projects page if you attempt to access a resource that no longer exists. You also want to ensure that a `flash[:alert]` is set.

To do all this, put this `it` block inside the `describe` block:

Listing 79. spec/requests/projects_request_spec.rb

```
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  it "handles a missing project correctly" do
    get project_path("not-here")

    expect(response).to redirect_to(projects_path)

    message = "The project you were looking for could not be found."
    expect(flash[:alert]).to eq message
  end
end
```

The first line in this RSpec test—more commonly called an example—tells RSpec to make a `GET` request to the route `/projects/not-here`. This route will then be served by the `ProjectsController's `show` action.

In the next line, you tell RSpec that you expect the response to take you back to the `projects_path` through a `redirect_to` call. If it doesn't, the test fails, and nothing more in this test is executed: RSpec stops in its tracks.

The final line tells RSpec that you expect `flash[:alert]` to contain a useful message explaining the redirection to the `index` action.

To run this spec, use the `bundle exec rspec spec/requests/projects_request_spec.rb` command. When this runs, you'll see this error:

```
1) Projects handles a missing project correctly
   Failure/Error: @project = Project.find(params[:id])
     ActiveRecord::RecordNotFound:
       Couldn't find Project with 'id'=not-here
```

This is the same failure you saw when you tried running the application using the development environment with `rails server`. Now that you have a failing test, you can fix it.

Open the `app/controllers/projects_controller.rb` file, and put the code from the following listing under the `private` line in the controller.

Listing 80. `set_project` **method in** `ProjectsController`

```
def set_project
  @project = Project.find(params[:id])
rescue ActiveRecord::RecordNotFound
  flash[:alert] = "The project you were looking for could not be found."
  redirect_to projects_path
end
```

Because it's under the `private` line, the controller doesn't respond to this method as an action. To call this method before every action that looks up a project, use the `before_action` method. Place these lines directly under the `class ProjectsController` definition:

```
before_action :set_project, only: %i(show edit update destroy)
```

What does all this mean? Methods referenced by `before_action` are run before all the actions in your controller, unless you specify either the `:except` or `:only` option. Here you have the `:only` option, defining actions for which you want `before_action` to run. The `:except` option is the opposite of the `:only` option, specifying the actions for which you don't want `before_action` to run. `before_action` calls the `set_project` method before the specified actions, setting up the `@project` variable for you. This means you can remove the following line from your `show`, `edit`, `update`, and `destroy` actions:

```
@project = Project.find(params[:id])
```

By doing this, you make the `show` and `edit` actions empty. If you remove these actions from your controller and run `bundle exec rspec` again, all the scenarios will still pass.

```
7 examples, 0 failures
```

Controller actions don't need to exist in the controllers if there are templates corresponding to those actions, which you have for these actions. For readability's sake though, it's best to leave these in the controller so anyone who reads the code knows that the controller can respond to these actions, so put the empty `show` and `edit` actions back in. Future you will thank you.

Back to the spec now. If you run `bundle exec rspec spec/controllers/projects_controller_spec.rb` once more, the test now passes:

```
1 example, 0 failures
```

Let's check to see if everything else is still working by running `bundle exec rspec`. You should see this:

```
7 examples, 0 failures
```

Red-green-refactor! With that out of the way, let's commit and push:

```
$ git add .
$ git commit -m "Redirect the users back to the projects page if they
  try going to a project that does not exist"
$ git push
```

This completes the basic CRUD implementation for your projects resource. Now you can create, read, update, and delete projects to your heart's content.

4.5. Summary

This chapter covered developing the first part of your application using test-first practices with RSpec and Capybara, building it one step at a time. Now you have an application that is truly maintainable. If you want to know if these specs are working later in the project, you can run `bundle exec rspec`; if something is broken that you've written a test for, you'll know about it. Doesn't that beat manual testing? Just think of all the time you'll save in the long run.

You learned firsthand how rapidly you can develop the CRUD interface for a resource in Rails. There are even faster ways to do it (such as by using scaffolding, discussed in Chapter 1); but to absorb how this process works, it's best to go through it yourself, step by step, as you did in these last two chapters.

So far, you've been developing your application using test-first techniques, and as your application grows, it will become more evident how useful these techniques are. The main thing they'll provide is assurance that what you've coded so far still works exactly as it did when you first wrote it. Without these tests, you may accidentally break functionality and not

know about it until a user—or worse, a client—reports it. It's best that you spend some time implementing tests for this functionality now so you don't spend even more time later apologizing for whatever's broken and fixing it.

With the basic projects functionality done, you're ready for the next step. Because you're building a ticket-tracking application, it makes sense to implement functionality that lets you track tickets, right? That's precisely what you'll do in the next chapter. We'll also cover nested routing and association methods for models. Let's go!

[34] As a reminder: DRY = Don't Repeat Yourself!
[35] Although the test won't check for this prompt, due to the difficulty in testing JS confirmation boxes in tests.
[36] The `rspec-rails` gem automatically generates the file using a Railtie, the code for which can be found at https://git.io/Jfmxo.

Chapter 5. Nested resources

With the project resource CRUD done in chapter 4, the next step is to set up the ability to create tickets within the scope of a given project. This chapter explores how to set up a nested resource in Rails, by defining routes for `Ticket` resources and creating a CRUD interface for them, all scoped under the `Project` resource that you just created. In this chapter, you'll see how easy it is to retrieve all ticket records for a specific project and perform CRUD operations on them, mainly with the powerful associations interface that Rails provides through its Active Record component.

5.1. Creating tickets

To add the functionality to create tickets under projects, you'll apply Behaviour Driven Development here again, writing RSpec specs with Capybara. Nesting one resource under another involves additional routing, working with associations in Active Record, and using more calls to `before_action`. Let's get into this.

To create tickets for your application, you need an idea of what you're going to implement. You want to create tickets only for particular projects, so you need a "New Ticket" link on a project's `show` page. The link must lead to a form where a name and a description for your ticket can be entered, and the form needs a button that submits it to a `create` action in your controller. You also want to ensure that the data entered is valid, as you did with the `Project` model. This new form will look like:

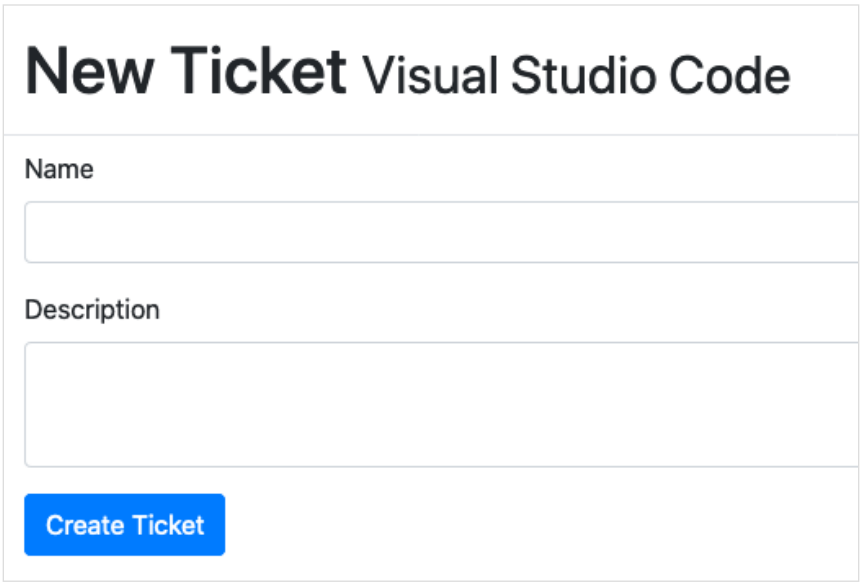


Figure 20. Form for creating new tickets

Start by using the code from the following listing in a new file.

Listing 81. spec/features/creating_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new tickets" do
  before do
    project = FactoryBot.create(:project, name: "Internet Explorer")

    visit project_path(project)
    click_link "New Ticket"
  end

  scenario "with valid attributes" do
    fill_in "Name", with: "Non-standards compliance"
    fill_in "Description", with: "My pages are ugly!"
    click_button "Create Ticket"

    expect(page).to have_content "Ticket has been created."
  end

  scenario "when providing invalid attributes" do
    click_button "Create Ticket"

    expect(page).to have_content "Ticket has not been created."
    expect(page).to have_content "Name can't be blank"
    expect(page).to have_content "Description can't be blank"
  end
end
```

You've seen the `before` method before, in section 4.2 when we were setting up the project data we needed for our "Editing Projects" spec to run. Here we're doing a similar thing - setting up the project that our tickets will be attached to. Your ticket objects need a parent project object to belong to (in our system a ticket can't exist outside of a project), so it makes sense to build one before every test.

In the above spec, we want to make sure to test the basic functionality of creating a ticket. It's pretty straightforward: start on the project page, click the "New Ticket" link, fill in the attributes, click the button, and make sure it works!

You should also test the failure case. Because you need to have a name and description, a failing case is easy: we click the "Create Ticket" button prematurely, before filling out all of the required information.

Now run this new feature using the `bundle exec rspec`

`spec/features/creating_tickets_spec.rb` command, both of your tests will fail due to your `before` block:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   Capybara::ElementNotFound:
     Unable to find link "New Ticket"

# and the second error is identical
```

You need to add this "New Ticket" link to the bottom of the `app/views/projects/show.html.erb` template, so that this line in the test will work. We'll copy the format we did for the projects `index` view, and build a `header` with the action link in it.

Listing 82. app/views/projects/show.html.erb

```
<header>
  <h2>Tickets</h2>

  <ul class="actions">
    <li><%= link_to "New Ticket", new_project_ticket_path(@project) %></li>
  </ul>
</header>
```

This helper is called a nested routing helper, and it's like the standard routing helper. The similarities and differences between the two are explained in the next section.

5.1.1. Nested routing helpers

When defining the "New Ticket" link, you used a nested routing helper—`new_project_ticket_path`—rather than a standard routing helper such as `new_ticket_path`, because you want to create a new ticket for a given project. Both helpers work in a similar fashion, except the nested routing helper always takes at least one argument: the `Project` object that the ticket belongs to. This is the object you're nested inside. The route to any ticket URL is always scoped by `/projects/:id` in your application. This helper and its brethren are defined by changing this line in `config/routes.rb`:

```
resources :projects
```

to these lines:

```
resources :projects do
  resources :tickets
end
```

This code tells the routing for Rails that you have a `tickets` resource nested inside the `projects` resource. Effectively, any time you access a ticket resource, you access it within the scope of a project. Just as the `resources :projects` method gave you helpers to use in controllers and views, this nested one gives you the helpers shown in this table:

Table 2. Nested RESTful routing matchup

Route	Helper
/projects/:project_id/tickets	`project_tickets_path`
/projects/:project_id/tickets/new	`new_project_ticket_path`
/projects/:project_id/tickets/:id/edit	`edit_project_ticket_path`
/projects/:project_id/tickets/:id	`project_ticket_path`

The routes belonging to a specific `Ticket` instance will now take two parameters - the project that the ticket belongs to, and the ticket itself - to generate URLs like http://localhost:3000/projects/1/tickets/2/edit.

As before, you can use the `*_url` or `*_path` alternatives to these helpers, such as `project_tickets_url`, to get the full URL if you so desire.

In the table's left column are the routes that can be accessed, and in the right are the routing helper methods you can use to access them. Let's use them by first creating your `TicketsController`.

5.1.2. Creating a tickets controller

Because you defined this route in your `routes.rb` file, Capybara can now click the link in your feature and proceed before complaining about the missing `TicketsController`. If you re-run

your spec with `bundle exec rspec spec/features/creating_tickets_spec.rb`, it spits out an error followed by a stack trace:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   ActionController::RoutingError:
     uninitialized constant TicketsController

# and the second error is identical
```

Some guides may have you generate the model before you generate the controller, but the order in which you create them isn't important. When writing tests, you follow the bouncing ball; and if the test tells you it can't find a controller, then the next thing you do is generate the controller it's looking for. Later, when you inevitably receive an error that it can't find the `Ticket` model, as you did for the `Project` model, you generate that, too. This is often referred to as top-down design [37].

To generate this controller and fix the `uninitialized constant` error, use this command:

```
$ rails g controller tickets
```

You may be able to pre-empt what's going to happen next if you run the test: it'll complain of a missing `new` action that it's trying to get to by clicking the "New Ticket" link. Let's just re-run the test to make sure:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   AbstractController::ActionNotFound:
     The action 'new' could not be found for TicketsController
```

So our next step is to define the `new` action. Open `app/controllers/tickets_controller.rb`, and add the `new` action inside the `TicketsController` definition, as shown in the following listing.

Listing 83. The `new` action from `TicketsController`

```
def new
  @ticket = @project.tickets.build
end
```

There's a lot of magic in this one line. We're referring to `@project` but we haven't defined it, we're referring to a `tickets` method on a `Project` instance but we haven't defined one, and we're calling a method called `build` on whatever `tickets` returns. Whew! One step at a time.

5.1.3. Demystifying the new action

We'll start with the `@project` instance variable. As we declared in our routes, our `tickets` resource is nested under a `projects` resource, giving us URLs like those shown in the earlier table, which is reproduced below:

Table 3. Nested RESTful routing matchup

Route	Helper
/projects/:project_id/tickets	`project_tickets_path`
/projects/:project_id/tickets/new	`new_project_ticket_path`
/projects/:project_id/tickets/:id/edit	`edit_project_ticket_path`
/projects/:project_id/tickets/:id	`project_ticket_path`

The placeholders in the URLs (`:project_id` and `:id`) are what we get as part of our `params` when we request these URLs. When we request http://localhost:3000/projects/1/tickets/2, our placeholders have the values of `1` and `2`, so `params` will include the following:

```
{ project_id: 1, id: 2 }
```

We can use the provided `:project_id` value to load up the right `Project` instance in a `before_action`, like we did for certain actions in our `ProjectsController`. Unlike the `ProjectsController` though, this `before_action` will be run before every action, because the project will always be present; and it will use `params[:project_id]`, instead of `params[:id]`

Add the following line under the `class` definition in `app/controllers/tickets_controller.rb`:

```
before_action :set_project
```

And now under the `new` action, you can define this `set_project` method that will use the `params[:project_id]` variable to load the `@project` variable.

```
private

def set_project
  @project = Project.find(params[:project_id])
end
```

Now our `@project` variable is defined. What about a `tickets` method? Is that the next thing we need to define? Re-run the test with `bundle exec rspec spec/features/creating_tickets_spec.rb` to see:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   NoMethodError:
     undefined method `tickets' for #<Project:0x007f26fa162628>
```

It is the next thing we need to define. We'll define `tickets` to be an association on our `Project` model - a link between the two models, so we can call `@project.tickets` and get an array of all of the `Ticket` instances that are part of the `@project`. Seems magical. Let's look at how it works.

5.1.4. Defining a has_many association

The `tickets` method on `Project` objects is defined by calling an association method in the `Project` class called `has_many`, add this as follows in `app/models/project.rb`:

```
class Project < ActiveRecord::Base
  has_many :tickets

  ...
```

As mentioned before, this defines the `tickets` method you need as well, as the association. With the `has_many` method called in the `Project` model, you can now get to all the tickets for any given project by calling the `tickets` method on any `Project` object.

By defining a `has_many` association in the model, it also gives you a whole slew of other useful methods[38], such as the `build` method, which you're also currently calling in the `new` action of `TicketsController`. The `build` method is equivalent to `new` for the `Ticket` class (which you create in a moment) but associates the new object instantly with the `@project` object by setting a foreign key called `project_id` automatically.

Upon rerunning `bundle exec rspec spec/features/creating_tickets_spec.rb`, you'll get this:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   NameError:
     uninitialized constant Project::Ticket
```

You can determine from this output that the method is looking for the `Ticket` class, but why? The `tickets` method on `Project` objects is defined by the `has_many` call in the `Project` model. This method assumes that when you want to get the tickets, you actually want instances of the `Ticket` model. This model is currently missing; hence, the error. You can add this model now with the following command:

```
$ rails g model ticket name:string description:text project:references
```

The `project:references` part defines an `integer` column for the `tickets` table called `project_id`. It also defines an index on this column so that lookups for the tickets for a specific project will be faster. The new migration for this model looks like this:

Listing 84. db/migrate/[timestamp]_create_tickets.rb

```
class CreateTickets < ActiveRecord::Migration[6.1]
  def change
    create_table :tickets do |t|
      t.string :name
      t.text :description
      t.references :project, null: false, foreign_key: true

      t.timestamps
    end
  end
end
```

The `project_id` column represents the project to which this ticket links and is called a foreign key. The purpose of this field is to store the primary key of the project the ticket relates to. By creating a ticket on the project with the `id` field of 1, the `project_id` field in the `tickets` table will also be set to 1.

The `foreign_key: true` part of the command enforces database-level foreign key restrictions for those platforms that support it, such as PostgreSQL. You can read more about the specifics of Rails foreign key support at: http://guides.rubyonrails.org/4_2_release_notes.html#foreign-key-support. The SQLite driver we're using doesn't support foreign keys like this, so we don't get any benefit from specifying them, but nor does it do any harm. We'll also be looking at using PostgreSQL when we cover deployment in chapter 13.

Run the migration with `rails db:migrate`. The `db:migrate` task runs the migrations and then dumps the structure of the database to a file called `db/schema.rb`. This structure allows you to restore your database using the `rails db:schema:load` task if you wish, which is better than running all the migrations on a large project again![39]

Now when you run `bundle exec rspec spec/features/creating_tickets_spec.rb`, you're told the `new` template is missing:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_link "New Ticket"
   TicketsController#new is missing a template for request formats: text/html
```

You must create this file in order to continue.

5.1.5. Creating tickets in a project

Create the file at `app/views/tickets/new.html.erb`, and put the following in it:

Listing 85. app/views/tickets/new.html.erb

```
<header>
  <h1>
    New Ticket
    <small><%= @project.name %></small>
  </h1>
</header>

<%= render "form", project: @project, ticket: @ticket %>
```

Like we did with projects, this template will render a `form` partial (so we can reuse it for the `edit` page when we get to it). The partial also goes in the `app/views/tickets` folder. Create a new file called `_form.html.erb`, using this code:

Listing 86. app/views/tickets/_form.html.erb

```
<%= form_with(model: [project, ticket], local: true) do |form| %>
  <% if ticket.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(ticket.errors.count, "error") %>
      prohibited this project from being saved:</h2>

      <ul>
        <% ticket.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <p>
    <%= form.label :name %><br>
    <%= form.text_field :name %>
  </p>
  <p>
    <%= form.label :description %><br>
    <%= form.text_area :description %>
  </p>
  <%= form.submit %>
<% end %>
```

Note that `form_with` is passed an array of objects rather than

```
<%= form_with(model: @ticket) do |f| %>
```

This code indicates to `form_with` that you want the form to post to a nested route. For the `new` action, this generates a route like /projects/1/tickets and for the `edit` action, it generates a route like /projects/1/tickets/2. This type of routing is known as polymorphic routing.[40]

When you run `bundle exec rspec spec/features/creating_tickets_spec.rb` again, you're told the `create` action is missing:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_button "Create Ticket"
   AbstractController::ActionNotFound:
     The action 'create' could not be found for TicketsController
```

To define this action, put it directly under the `new` action in `TicketsController` but before the `private` method. Also add the appropriate strong parameters helper method right below `private`, as shown in the following listing.

Listing 87. The `create` action from `TicketsController`

```
def create
  @ticket = @project.tickets.build(ticket_params)

  if @ticket.save
    flash[:notice] = "Ticket has been created."
    redirect_to [@project, @ticket]
  else
    flash.now[:alert] = "Ticket has not been created."
    render "new"
  end
end

private

def ticket_params
  params.require(:ticket).permit(:name, :description)
end
```

In this action, you use `redirect_to` and specify an `Array`—the same array you used in `form_with` earlier—containing a `Project` object and a `Ticket` object. Rails inspects any array passed to helpers, such as `redirect_to` and `link_to`, and determines what you mean from the values. For this particular case, Rails figures out that you want this helper:

```
project_ticket_path(@project, @ticket)
```

Rails determines this helper because, at this stage, `@project` and `@ticket` are both objects that exist in the database, and you can therefore route to them. The route generated would be /projects/1/tickets/2 or something similar. Back in the `form_with`, `@ticket` was new, so the route happened to be /projects/1/tickets. You could have been explicit and specifically used `project_ticket_path` in the action, but using an array is less repetitive.

When you run `bundle exec rspec spec/features/creating_tickets_spec.rb`, both scenarios continue report the same error:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_button "Create Ticket"
   AbstractController::ActionNotFound:
     The action 'show' could not be found for TicketsController
```

Therefore, you must create a `show` action for the `TicketsController`. But when you do so, you'll need to find tickets only for the given project.

5.1.6. Finding tickets scoped by project

Currently, both of your scenarios are failing due to a missing action. The next logical step is to define the `show` action for your controller, which will look up a given ticket by ID. But, being quick to learn and spot trends, you can anticipate that you'll also need to find a ticket by ID for the `edit`, `update`, and `destroy` actions, and pre-empt similar errors when it comes to building those actions. You can make this a `before_action`, as you did in the `ProjectsController` with the `set_project` method. You define this finder under the `set_project` method in the `TicketsController`:

```
def set_ticket
  @ticket = @project.tickets.find(params[:id])
end
```

`find` is yet another association method provided by Rails when you declared that your `Project` model `has_many :tickets`. This code attempts to find tickets only within the collection of tickets owned by the specified project. Put your new `before_action` at the top of your class, under the action to find the project:

```
before_action :set_project
before_action :set_ticket, only: %i(show edit update destroy)
```

The sequence here is important because you want to find the `@project` before you go looking for tickets for it. Then you can create the action that your test is asking for, below the `create` method (but above `private`) in your `TicketsController`:

Listing 88. show **action in app/controllers/tickets_controller.rb**

```
def show
end
```

Again, it doesn't need to have anything in it - we've already loaded all of the content the action needs, in our `before_action` calls. But it's good to know it's there. Then create the view template for this action at `app/views/tickets/show.html.erb`, using this code:

Listing 89. app/views/tickets/show.html.erb

```
<div class="ticket">
  <header>
    <h1><%= @project.name %></h1>
  </header>

  <header>
    <h2><%= @ticket.name %></h2>
  </header>

  <%= simple_format(@ticket.description) %>
</div>
```

The new method, `simple_format`, converts the line breaks[41] entered into the description field into HTML break tags (`<br>`) so that the description renders exactly how the user intends it to.

Based solely on the changes you've made so far, your first scenario should be passing. Let's see with a quick run of `bundle exec rspec spec/features/creating_tickets_spec.rb`:

```
1) Users can create new tickets when providing invalid attributes
   Failure/Error: expect(page).to have_content "Ticket has not been
   created."
     expected to find text "Ticket has not been created." in "Ticket has been
created.\nInternet Explorer"

...
2 examples, 1 failure
```

This means you've got the first scenario under control, and users of your application can create tickets within a project. Next, you need to add validations to the `Ticket` model to get the second scenario to pass.

5.1.7. Ticket validations

The second scenario fails because the `@ticket` that it saves isn't valid, at least according to your tests in their current state:

```
expected to find text "Ticket has not been created." in "Ticketee ...
```

You need to ensure that when somebody enters a ticket into the application, the `name` and `description` attributes are filled in. To do this, define the following validations in the `Ticket` model.

Listing 90. app/models/ticket.rb

```
validates :name, presence: true
validates :description, presence: true
```

Validating two fields using one line

You could also validate the presence of both of these fields using a single line:

```
validates :name, :description, presence: true
```

However, it is easier to see the associations for a given field if they are all in one place. If you were to add, for example, an extra length validation to the description field, it might look like this:

```
validates :name, :description, presence: true
validates :description, length: { maximum: 1000 }
```

And it would not be immediately obvious that both validations apply to one field (the `description` field). As more and more fields get added (you might validate the presence of over a dozen fields!), the problem would get worse and worse as the details get spread further and further apart.

So it's our preference to have validations for different fields on individual lines. You don't have to use two lines to do it; we can still be friends.

Now, when you run `bundle exec spec/features/creating_tickets_spec.rb`, the entire

feature passes:

```
2 examples, 0 failures
```

Before we wrap up here, let's add one more scenario to ensure that what is entered into the "Description" field is longer than 10 characters. You want the descriptions to be useful! Add this scenario to the `spec/features/creating_tickets_spec.rb` file:

```
scenario "with an invalid description" do
  fill_in "Name", with: "Non-standards compliance"
  fill_in "Description", with: "It sucks"
  click_button "Create Ticket"

  expect(page).to have_content "Ticket has not been created."
  expect(page).to have_content "Description is too short"
end
```

To implement the code needed to make this scenario pass, add another option to the end of the validation for the `description` in your `Ticket` model, like this:

```
validates :description, presence: true, length: { minimum: 10 }
```

By default, this will generate a message identical to the one we've used in our test. You can verify this with the console - if you run `rails console` and try to create a new `Ticket` object by using `create!`, you can get the full text for your error:

```
irb(main):001:0> Ticket.create!
ActiveRecord::RecordInvalid: ... Description is too short
(minimum is 10 characters)
```

If you're getting that error message on the console, that means it will appear like that in the app too. Find out by running `bundle exec rspec spec/features/creating_tickets_spec.rb` again:

```
3 examples, 0 failures
```

That one's passing now. Excellent! You should ensure that the rest of the project still works by running `bundle exec rspec` again. You'll see this output:

```
12 examples, 0 failures, 2 pending
```

There are two pending specs here: one located in `spec/helpers/tickets_helper_spec.rb` and the other in `spec/models/ticket_spec.rb`. These were automatically generated when you ran the commands to generate your `TicketsController` and `Ticket` model - you don't need them right now, so you can just delete these two files. When you've done that, rerunning `bundle exec rspec` outputs a lovely green result:

```
10 examples, 0 failures
```

Great! Everything's still working. Commit and push the changes!

```
$ git add .
$ git commit -m "Implement creating tickets for a project"
$ git push
```

This section covered how to create tickets and link them to a specific project through the foreign key called `project_id` on records in the `tickets` table. The next section shows how easily you can list tickets for individual projects.

5.2. Viewing tickets

Now that you have the ability to create tickets, you'll use the `show` action of the `TicketsController` to view them individually. When displaying a list of projects, you use the `index` action of the `ProjectsController`. For tickets, however, we'll list them them as part of showing the details of a project, on the `show` action of the `ProjectsController`. This page currently isn't being used for anything else in particular, but also it just makes sense to see the project's tickets when you view the project. To test it, put a new feature at `spec/features/viewing_tickets_spec.rb` using the code from the following listing.

Listing 91. spec/features/viewing_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can view tickets" do
  before do
    vscode = FactoryBot.create(:project, name: "Visual Studio Code")
    FactoryBot.create(:ticket, project: vscode,
      name: "Make it shiny!",
      description: "Gradients! Starbursts! Oh my!")

    ie = FactoryBot.create(:project, name: "Internet Explorer")
    FactoryBot.create(:ticket, project: ie,
      name: "Standards compliance", description: "Isn't a joke.")

    visit "/"
  end

  scenario "for a given project" do
    click_link "Visual Studio Code"

    expect(page).to have_content "Make it shiny!"
    expect(page).to_not have_content "Standards compliance"

    click_link "Make it shiny!"
    within(".ticket h2") do
      expect(page).to have_content "Make it shiny!"
    end

    expect(page).to have_content "Gradients! Starbursts! Oh my!"
  end
end
```

Quite the long feature! It covers a couple of things - both viewing the list of tickets for a project, and then viewing the details for a specific ticket.[42] We'll go through it piece by piece in a moment. First, let's examine the `within` usage in the scenario. Rather than checking the entire page for content, this step checks the specific element using Cascading Style Sheets (CSS) selectors. The `.ticket h2` selector finds all `h2` elements within a div with the `class` attribute set to `ticket`, and then we make sure the content is visible within one of those elements.[43] This content should appear in the specified tag only when you're on the ticket page, so this is a great way to make sure you're on the right page and that the page is displaying relevant information.

When you run this spec with `bundle exec rspec spec/features/viewing_tickets_spec.rb`, you'll see that it can't find the ticket factory:

```
Failure/Error:
  FactoryBot.create(:ticket, project: vscode,
    name: "Make it shiny!",
    description: "Gradients! Starbursts! Oh my!")

KeyError:
  Factory not registered: "ticket"
```

Just as before, when the project factory wasn't registered, you need to create the ticket factory now. It should create an example ticket with a valid name and description. To do this, create a new file called `spec/factories/tickets.rb` with the following content.

Listing 92. spec/factories/tickets.rb

```
FactoryBot.define do
  factory :ticket do
    name { "Example ticket" }
    description { "An example ticket, nothing more" }
  end
end
```

With the ticket factory defined, the `before` block of this spec should now run all the way through when you run `bundle exec rspec spec/features/viewing_tickets_spec.rb`. You'll see this error:

```
1) Users can view tickets for a given project
    Failure/Error: expect(page).to have_content "Make it shiny!"
      expected to find text "Make it shiny!" in "Visual Studio Code\nEdit Project Delete
Project\nTickets\nNew Ticket"
```

The spec is attempting to see the ticket's name on the page. But it can't see it at the moment, because you're not displaying a list of tickets on the project `show` template yet.

5.2.1. Listing tickets

To display a ticket on the `show` template, you can iterate through the project's tickets by using the `tickets` method on a `Project` object, made available by the `has_many :tickets` call in your model. Put this code at the bottom of `app/views/projects/show.html.erb`.

Listing 93. app/views/projects/show.html.erb

```
<ul class="tickets">
  <% @project.tickets.each do |ticket| %>
    <li>
      #<%= ticket.id %> -
      <%= link_to ticket.name, [@project, ticket] %>
    </li>
  <% end %>
</ul>
```

Be careful when using `link_to`.

If you use a `@ticket` variable in place of the `ticket` variable as the second argument to `link_to`, it will be `nil`. You haven't initialized the `@ticket` variable, and uninitialized instance variables are `nil` by default. If `@ticket` rather than the correct `ticket` is passed in, the URL generated will be a projects URL, such as /projects/1, rather than the correct /projects/1/tickets/2.

Here you iterate over the items in `@project.tickets` using the `each` method, which does the iterating for you, assigning each item to a `ticket` variable used in the block. The code in this block runs for every ticket. When you run `bundle exec rspec spec/features/viewing_tickets_spec.rb`, it passes because the app now has the means to go to a specific ticket from the project's page:

```
1 example, 0 failures
```

Time to make sure everything else is still working by running `bundle exec rspec`. You should see all green:

```
11 examples, 0 failures
```

Fantastic! Push!

```
$ git add .
$ git commit -m "Implement tickets display"
$ git push
```

You can see tickets for a particular project, but what happens when a project is deleted? The tickets for that project aren't magically deleted. To implement this behavior, you can pass some options to the `has_many` association, which will delete the tickets when a project is deleted.

5.2.2. Culling tickets

When a project is deleted, its tickets become useless: they're inaccessible because of how you defined their routes. Therefore, when you delete a project, you should also delete the tickets for that project. You can do that by using the `:dependent` option on the `has_many` association for tickets defined in your `Project` model.

This option has five choices that all act slightly differently. The first one is the `:destroy` value:

```
has_many :tickets, dependent: :destroy
```

If you put this in your `Project` model, any time you call `destroy` on a `Project` object, Rails will iterate through the tickets for this project, and call `destroy` on each of them in turn (as well as any other destroy-related callbacks on the project itself). In turn, each ticket object will have any destroy-related callbacks called on it, and if it has any `has_many` associations with the `dependent: :destroy` option set, then those objects will be destroyed, and so on. The problem is that if you have a large number of tickets, `destroy` is called on each one, which will be slow.

The solution is the second value for this option:

```
has_many :tickets, dependent: :delete_all
```

This deletes all the tickets using a SQL delete, like this:

```
DELETE FROM tickets WHERE project_id = :project_id
```

This operation is quick and is exceptionally useful if you have a large number of tickets that don't have callbacks or that have callbacks you don't necessarily care about when deleting a project. If you do have callbacks on `Ticket` for a destroy operation, then you should use the first option, `dependent: :destroy`.

Thirdly, if you want to disassociate tickets from a project and unset the `project_id` field, you can use this option:

```
has_many :tickets, dependent: :nullify
```

When a project is deleted with this type of `:dependent` option defined, it will execute an SQL query such as this:

```
UPDATE tickets SET project_id = NULL WHERE project_id = :project_id
```

Rather than deleting the tickets, this option keeps them around; but their `project_id` fields are unset, leaving them orphaned, which isn't suitable for this system.

Using this option would be helpful, for example, if you were building a task- tracking application and instead of projects and tickets, you had users and tasks. If you deleted a user, you might want to unassign rather than delete the tasks associated with that user, in which case you'd use the `dependent: :nullify` option instead.

Finally, you have two options that work similarly - `:restrict_with_error` and `:restrict_with_exception`. These options will both prevent records from being deleted if the association isn't empty - for example, in our projects and tickets scenario we wouldn't be able to delete projects if they had any tickets in them.

If we were using `:restrict_with_error` then calling `@project.destroy` on a project with tickets would add a validation error to the `@project` instance, as well as returning false. Using `:restrict_with_exception` in this case would raise an exception that our application would have to manually catch and handle, or else the user would receive a HTTP response of 500 - Internal Server Error. An example of where this could be useful is in a billing scenario: it wouldn't be good for business if users were able to cancel/delete their own accounts in your system, if they had associated bills that still required payment.

In our projects and tickets scenario, though, you would use `dependent: :destroy` if you have callbacks to run on tickets when they're destroyed or `dependent: :delete_all` if you have no callbacks on tickets. To ensure that all tickets are deleted on a project when the project is deleted, change the `has_many` association in your `Project` model to this:

Listing 94. app/models/project.rb

```
has_many :tickets, dependent: :delete_all
```

With this new `:dependent` option in the `Project` model, all tickets for the project will be deleted when the project is deleted.

You aren't writing any tests for this behavior, because it's simple and you'd basically be testing that you changed one tiny option. This is more of an internal implementation detail than it is customer-facing, and you're writing feature tests right now, not model tests. Let's check that you didn't break existing tests by running `bundle exec rspec`:

```
11 examples, 0 failures
```

Good! Let's commit:

```
$ git add .
$ git commit -m "Cull tickets when project gets destroyed"
$ git push
```

Next, let's look at how to edit the tickets in your application.

5.3. Editing tickets

You want users to be able to edit tickets, the updating part of this CRUD interface for tickets. This section covers creating the `edit` and `update` actions for the `TicketsController`. This functionality follows a thread similar to the projects edit feature, where you follow an "Edit" link in the `show` template, change a field, and then click an update button and expect to see two things: a message indicating that the ticket was updated successfully, and the modified data for that ticket.

As always, we'll start with the test that covers the functionality we wish we had. Then we'll write the code that will make the test pass.

5.3.1. The "Editing tickets" spec

Just as you made a spec for creating a ticket, you need one for editing and updating existing

tickets. Specs testing update functionality are always a little more complex than specs for testing create functionality, because you need to have an existing object that's built properly before the test, and then you can change it during the test.

With that in mind, you can write this feature using the code in the following listing. Put the code in a file at `spec/features/editing_tickets_spec.rb`.

Listing 95. spec/features/editing_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can edit existing tickets" do
  let(:project) { FactoryBot.create(:project) }
  let(:ticket)  { FactoryBot.create(:ticket, project: project) }

  before do
    visit project_ticket_path(project, ticket)
  end

  scenario "with valid attributes" do
    click_link "Edit Ticket"
    fill_in "Name", with: "Make it really shiny!"
    click_button "Update Ticket"

    expect(page).to have_content "Ticket has been updated."

    within(".ticket h2") do
      expect(page).to have_content "Make it really shiny!"
      expect(page).not_to have_content ticket.name
    end
  end

  scenario "with invalid attributes" do
    click_link "Edit Ticket"
    fill_in "Name", with: ""
    click_button "Update Ticket"

    expect(page).to have_content "Ticket has not been updated."
  end
end
```

At the top of this feature, you use a new RSpec method called `let`. In fact, you use it twice. It defines a new method with the same name as the symbol passed in, and that new method then evaluates (and caches) the content of the block whenever that method is called. It's also lazy-loaded - the block won't get evaluated until the first time you call the method denoted by

the symbol, eg. `project` or `ticket` in this case.

It also has a bigger brother, called `let!` (with a bang!) `let!` isn't lazy-loaded - when you define a method with `let!` it will be evaluated immediately, before your tests start running.

For a concrete example, if we had a test that looked like the following:

Listing 96. Testing `let` and `let!`

```
RSpec.describe "A sample test" do
  let!(:project) { FactoryBot.create(:project) }
  let(:ticket)   { FactoryBot.create(:ticket) }

  it "lazily loads `let` methods" do
    puts Project.count
    puts Ticket.count

    puts ticket.name
    puts Ticket.count
  end
end
```

If you were to run it, what do you think it might output? If you guessed the following:

- `Project.count` → 1 (as `project` is already evaluated)
- `Ticket.count` → 0 (`ticket` has not been evaluated yet)
- `ticket.name` → "Example ticket" (from our factory)
- `Ticket.count` → 1 (`ticket` has now been evaluated and exists in the database)

You get a gold star!

In our case, it makes no difference if we use `let` or `let!`. The first thing we're doing in the `before` block is instantiating both `project` and `ticket` by visiting the ticket's show page. If, however, we were visiting the homepage and then navigating to the ticket's page, it wouldn't work - the ticket would never be created.

After we visit the ticket's `show` page, then we click the "Edit" link, make some changes, and verify that those changes get persisted. We're also testing the failure case - what happens if we can't update a ticket for some reason. It looks pretty similar to the update case, but rather than try to factor out all the commonalities, you repeat yourself. Some duplication in tests is OK; if it makes the test easier to follow, it's worth a little repetition.

When you run this feature using `bundle exec rspec spec/features/editing_tickets_spec.rb`, the first three lines in the `before` run fine, but the fourth fails:

```
1) Users can edit existing tickets with valid attributes
   Failure/Error: click_link "Edit Ticket"
   Capybara::ElementNotFound:
     Unable to find link "Edit Ticket"
```

To fix this, add the "Edit Ticket" link to the `show` template of the `TicketsController`, because that's the page you've visited in the feature. It sounds like an action link for the ticket, so we can add a list of action links into the `header` that specifies the ticket's name.

Listing 97. app/views/tickets/show.html.erb

```
<header>
  <h2><%= @ticket.name %></h2>

  <ul class="actions">
    <li><%= link_to "Edit Ticket", [:edit, @project, @ticket] %></li>
  </ul>
</header>
```

Here is yet another use of the `Array` argument passed to the `link_to` method, but rather than passing just Active Record objects, you pass a `Symbol` first. Rails, yet again, works out from the `Array` what route you wish to follow. Rails interprets this array to mean the `edit_project_ticket_path` method, which is called like this:

```
edit_project_ticket_path(@project, @ticket)
```

Now that you have an "Edit Ticket" link, you need to add the `edit` action to the `TicketsController`, because that will be the next thing to error when you run `bundle exec rspec spec/features/editing_tickets_spec.rb`:

```
1) Users can edit existing tickets with valid attributes
   Failure/Error: click_link "Edit Ticket"
   AbstractController::ActionNotFound:
     The action 'edit' could not be found for TicketsController
  ...
2 examples, 2 failures
```

5.3.2. Adding the edit action

The next logical step is to define the `edit` action in your `TicketsController`. Like the `edit` action in `ProjectsController`, it doesn't technically need to exist because it will be empty - all it needs to do is load the `@project` and `@ticket` variables, which are already done via `set_project` and `set_ticket`. But it's good practice to define it, so add it in under the `show` action in `TicketsController`, but before the `private` call.

Listing 98. app/controllers/tickets_controller.rb

```
def edit
end
```

The next logical step is to create the view for this action. Put it at `app/views/tickets/edit.html.erb`, and fill it with this content:

Listing 99. app/views/tickets/edit.html.erb

```
<header>
  <h1>
    Edit Ticket
    <small><%= @project.name %></small>
  </h1>
</header>

<%= render "form", project: @project, ticket: @ticket %>
```

Here you reuse the `form` partial you created for the `new` action, which is handy. The `form_with` knows which action to go to. If you run the feature spec again with `bundle exec rspec spec/features/editing_tickets_spec.rb`, you're told the `update` action is missing:

```
1) Users can edit existing tickets with valid attributes
   Failure/Error: click_button "Update Ticket"
   AbstractController::ActionNotFound:
     The action 'update' could not be found for TicketsController
```

5.3.3. Adding the update action

You should now define the `update` action in your `TicketsController`, as shown in the following listing.

Listing 100. The `update` action of `TicketsController`

```
def update
  if @ticket.update(ticket_params)
    flash[:notice] = "Ticket has been updated."
    redirect_to [@project, @ticket]
  else
    flash.now[:alert] = "Ticket has not been updated."
    render "edit"
  end
end
```

Remember that in this action you don't have to find the `@ticket` or `@project` objects, because a `before_action` does it for the `show`, `edit`, `update`, and `destroy` actions. With this single action implemented, both scenarios in the "Editing Tickets" feature will now pass when you run `bundle exec rspec spec/features/editing_tickets_spec.rb`:

```
2 examples, 0 failures
```

Check to see if everything works with a quick run of `bundle exec rspec`:

```
13 examples, 0 failures
```

Great! Let's commit and push that:

```
$ git add .
$ git commit -m "Tickets can now be edited"
$ git push
```

In this section, you implemented `edit` and `update` for the `TicketsController` by using the scoped finders and some familiar methods, such as `update`. You've got one more part to go: deletion.

5.4. Deleting tickets

We now reach the final story for this nested resource, deleting tickets. As with some of the other actions in this chapter, this story doesn't differ from what you used in the `ProjectsController`, except you'll change the name project to ticket for your variables and

`flash[:notice]`. It's good to have the reinforcement of the techniques previously used: practice makes perfect.

Use the code from the next listing to write a new feature in `spec/features/deleting_tickets_spec.rb`.

Listing 101. spec/features/deleting_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can delete tickets" do
  let(:project) { FactoryBot.create(:project) }
  let(:ticket)  { FactoryBot.create(:ticket, project: project) }

  before do
    visit project_ticket_path(project, ticket)
  end

  scenario "successfully" do
    click_link "Delete Ticket"

    expect(page).to have_content "Ticket has been deleted."
    expect(page.current_url).to eq project_url(project)
    expect(page).not_to have_content(ticket.name)
  end
end
```

When you run this spec using `bundle exec rspec spec/features/deleting_tickets_spec.rb`, it will fail because you don't yet have a "Delete Ticket" link on the `show` template for tickets:

```
1) Users can delete tickets successfully
   Failure/Error: click_link "Delete Ticket"
   Capybara::ElementNotFound:
     Unable to find link "Delete Ticket"
```

You can add the "Delete Ticket" link to the list of actions on `app/views/tickets/show.html.erb`, right after the "Edit Ticket" link.

```
<li><%= link_to "Delete Ticket", [@project, @ticket], method: :delete,
  data: { confirm: "Are you sure you want to delete this ticket?"} %></li>
```

The `method: :delete` is specified again, turning the request into one headed for the `destroy`

action in the controller. Without this `:method` option, you'd be off to the `show` action because the `link_to` method defaults to the `GET` method. Upon running `bundle exec rspec spec/features/deleting_tickets_spec.rb` again, you're told a `destroy` action is missing:

```
1) Users can delete tickets successfully
   Failure/Error: click_link "Delete Ticket"
   AbstractController::ActionNotFound:
     The action 'destroy' could not be found for TicketsController
```

The next step must be to define this action, right? Open `app/controllers/tickets_controller.rb`, and define it directly under the `update` action.

Listing 102. The `destroy` action from `TicketsController`

```
def destroy
  @ticket.destroy
  flash[:notice] = "Ticket has been deleted."

  redirect_to @project
end
```

After you delete the ticket, you redirect the user back to the `show` page for the project the ticket belonged to. With that done, your feature should now pass when you run `bundle exec rspec spec/features/deleting_tickets_spec.rb` again:

```
1 example, 0 failures
```

Yet again, check to see that everything is still going as well as it should by using `bundle exec rspec`. You haven't changed much, so it's likely that things are still working. You should see this output:

```
14 examples, 0 failures
```

Commit and push!

```
$ git add .
$ git commit -m "Implement deleting tickets feature"
$ git push
```

git You've now completely created another CRUD interface, this time for the `tickets` resource, which is only accessible within the scope of a project. This means you must request it using a URL such as `/projects/1/tickets/2` rather than `/tickets/2`.

5.5. Summary

In this chapter, you generated another controller, the `TicketsController`, which allows you to create records for your `Ticket` model that will end up in your `tickets` table. The difference between this controller and the `ProjectsController` is that the `TicketsController` is accessible only within the scope of an existing project, because you used nested routing.

In this controller, you scoped the finds for the `Ticket` model by using the `tickets` association method, provided by the association helper method `has_many` call in your `Project` model. `has_many` also provides the `build` method, which you used to begin creating new `Ticket` records that are scoped to a project.

In the next chapter, you'll learn how to let users sign up and sign in to your application. You'll also implement a basic authorization for actions such as creating a project.

[37] http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design
[38] For a complete list of what you get with a simple call to `has_many` - http://guides.rubyonrails.org/association_basics.html#has-many-association-reference
[39] Large projects can have hundreds of migrations, which may not run due to changes in the system over time. It's best to use `rails db:schema:load`. Keep in mind that this will destroy all data in your database.
[40] A great description of which can be found at http://ryanbigg.com/2012/03/polymorphic-routes.
[41] Line breaks are represented as `\n` and `\r\n` in strings in Ruby rather than as visible line breaks.
[42] Purists would probably split this out into two separate features, but the second feature would depend on the first - if you can't see a list of tickets (feature 1), it would be impossible to click the link to see a ticket's details (feature 2). So we've put them as part of one feature.
[43] We'll revisit this in chapter 10 - hardcoding CSS selectors in a test isn't a great idea, because we're testing what the user can see, and they don't care about selectors and tags, they just care about content.

Chapter 6. Styling the application

The application is currently looking a bit plain:

Figure 21. A plain form

We can change this by using Bootstrap[44], which is a front-end CSS and JavaScript framework that comes with a collection of styles that we can apply to our application. When we use these styles, our new project form will turn into this:

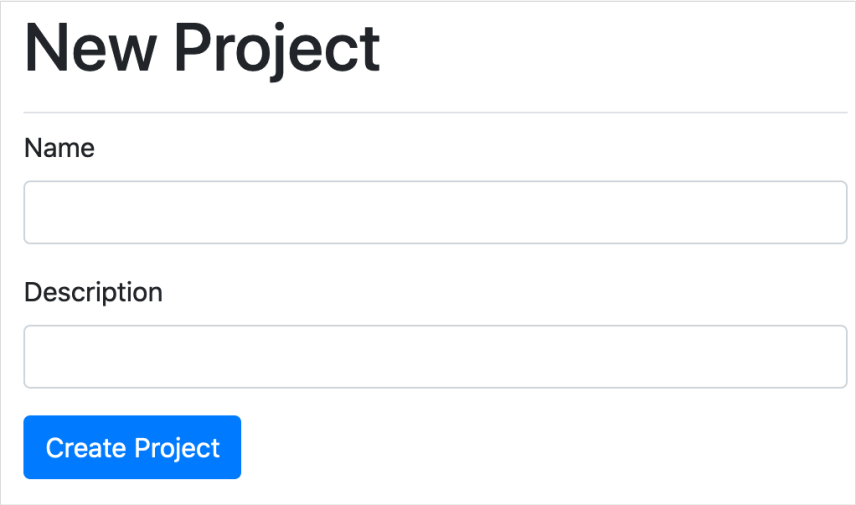


Figure 22. A neater form

You'll notice the asterisk next to the "Name" field here. This is because the "Name" field is a required field, and that's because we have a validation on our `Project` model for the presence of this field. We're going to be using a gem called `bootstrap_form` to style our form in this way.

The `bootstrap_form` gem not only makes our form neater, but also makes the code for

generating a form much simpler. Whereas we have this now:

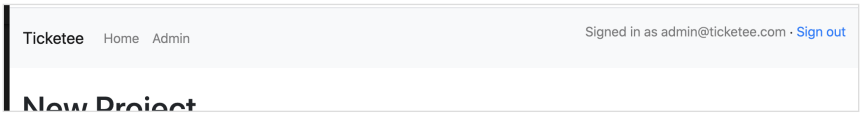
Listing 103. app/views/projects/_form.html.erb

```
<div>
  <%= form.label :name %>
  <%= form.text_field :name %>
</div>

<div>
  <%= form.label :description %>
  <%= form.text_field :description %>
</div>
```

The `bootstrap_form` gem allows us to write this instead:

```
<%= f.text_field :name %>
<%= f.text_field :description %>
```

The Bootstrap framework also lends itself to more than just forms. We'll be using it in this section to style the flash messages from our application, as well as adding a navigation bar to the top of the screen.



Let's get started!

6.1. Installing Bootstrap

The first thing that we need to do is to install the `bootstrap` gem, which comes with Sass versions of the Bootstrap assets. Sass bills itself as "CSS with Superpowers", and we'll be using it to write our CSS for our application.

This `bootstrap` gem is the recommended way to install Bootstrap in to a Rails application. Let's add this gem to our application now by running this command:

```
bundle add bootstrap -v {bootstrap_version}
```

To add Bootstrap's styles to your application, you'll need to make your main application

stylesheet a Sass file, by renaming `app/assets/stylesheets/application.css` to `application.css.scss`, adding the `.scss` extension to the end of the filename. This extension tells the asset pipeline that this file should be processed using the Sass pre-processor, before it's presented as a CSS file. This pre-processing will convert any Sass-specific code to CSS code, so that browsers can understand it.

Let's replace the entire contents of that `application.css.scss` file with this:

Listing 104. app/assets/stylesheets/application.css.scss, with Bootstrap imported

```
@import "bootstrap";
```

This line imports all of Bootstrap's CSS components.[45] With these assets now imported, we can restart our server and go to http://localhost:3000 to see some changes immediately:

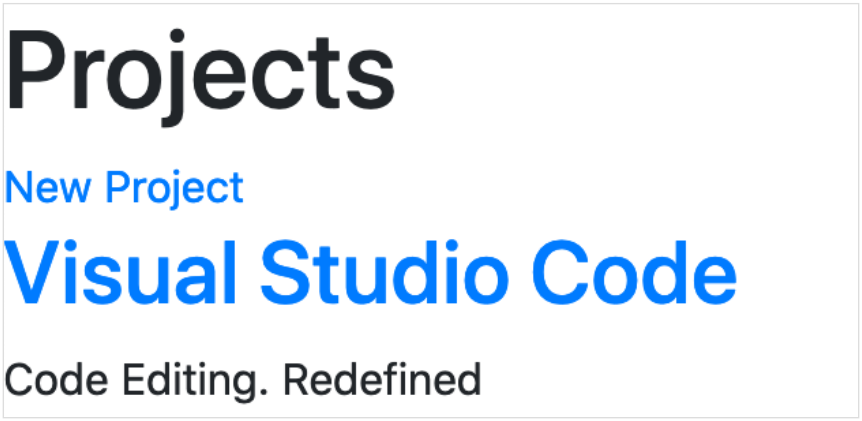


Figure 23. Projects page, Bootstrap-ified

The styling of our home page has changed to reflect Bootstrap's default styling! That was easy. It's all hard up against the left side of the page though. We can fix this by wrapping all of the content in a container in `app/views/layouts/application.html.erb`:

Listing 105. app/views/layouts/application.html.erb

```
<body>
  <div class="container-fluid">
    <% flash.each do |key, message| %>
      <div><%= message %></div>
    <% end %>

    <%= yield %>
  </div>
</body>
```

This element will shift the content away from the left-hand side of the page by using Bootstrap's `container-fluid` class[46]:

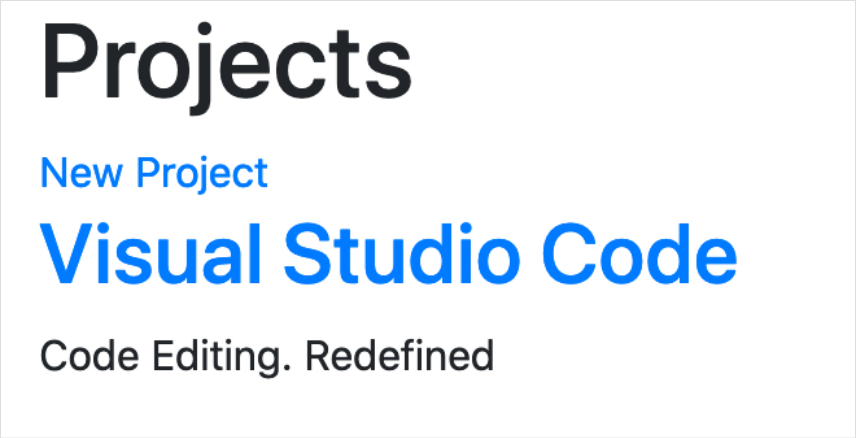


Figure 24. Now with padding!

6.2. Improving the page's header

Next up, let's make the header section of our page look a little bit nicer, starting with the main heading. Bootstrap provides utility classes[47] that we can use to give a bit of style, so let's wrap these around the `h1` in `app/views/projects/index.html.erb`.

Listing 106. app/views/projects/index.html.erb

```
<div class="pb-2 mt-4 mb-2 border-bottom">
  <h1>Projects</h1>
</div>
```

It adds some nice spacing around the important header, and an underline. So far so good. Now we can look at the actions we can take on this page - the most obvious action is to create a

new project, so we'll make our "New Project" link stand out.

Ultimately, we want to make the "New Project" be part of the page header, and we'll put it on the right hand side to give the page a bit of balance.

In `app/views/projects/index.html.erb`, change the code from this:

```
<div class="pb-2 mt-4 mb-2 border-bottom">
  <h1>Projects</h1>
</div>

<%= link_to "New Project", new_project_path %>
```

To this:

```
<div class="pb-2 mt-4 mb-2 border-bottom">
  <h1>Projects</h1>

  <ul class="actions">
    <li><%= link_to "New Project", new_project_path,
      class: "btn btn-success" %></li>
  </ul>
</div>
```

The relevant page actions is now part of the header, in a list (as some pages will have multiple actions to take, such as edit or delete). And adding the two `btn btn-success` classes to any HTML element will change its appearance into a button, like this:

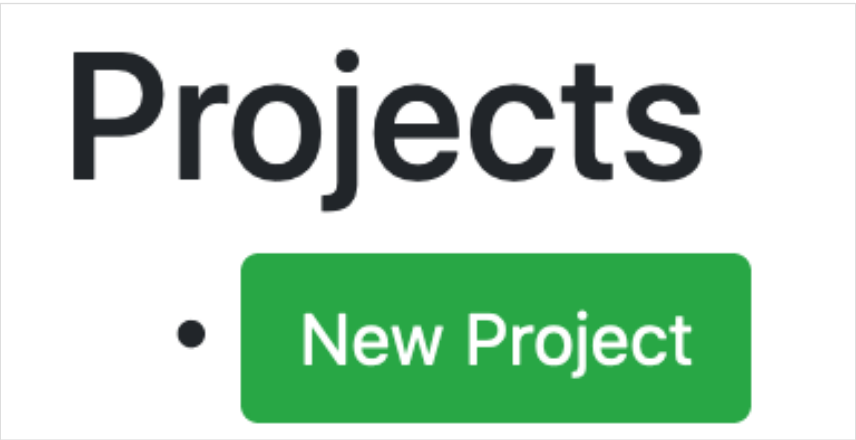


Figure 25. The new "New Project" button

The `btn-success` turns it into a specific type of button.[48] We can add a little bit more flair to

this button by adding an icon from the Font Awesome project[49], turning it into this:

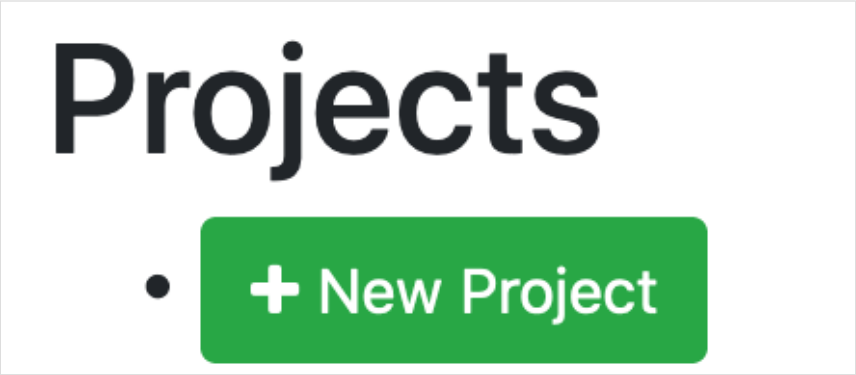


Figure 26. The new new "New Project" button

To get to that newest, best looking version of the button, let's add the `font-awesome-rails` gem to our application with this command:

```
bundle add font-awesome-rails -v 4.7.0.6
```

Just like the `bootstrap` gem, the `font-awesome-rails` gem also comes with some assets. The assets from the `font-awesome-rails` gem give us a whole range of icons, shown on the Font Awesome icons page: http://fortawesome.github.io/Font-Awesome/icons/. To use these icons, we must first run `bundle` and restart the Rails server. Then we'll need to add another `@import` line to `app/assets/stylesheets/application.css.scss`, after the bootstrap line that you have already:

Listing 107. Adding `font-awesome` to your `application.css.scss`

```
@import "bootstrap";
@import "font-awesome";
```

To add the icon to the button, you can use the `fa_icon` helper—this comes from the `font-awesome-rails` gem—as part of the link in `app/views/projects/index.html.erb`, like this:

```
<li><%= link_to fa_icon("plus") + " New Project", new_project_path,
  class: "btn btn-success" %></li>
```

It will now look like this:

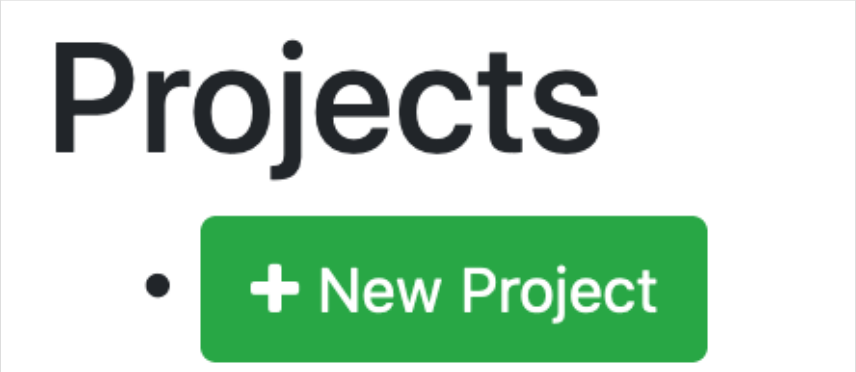


Figure 27. The new new "New Project" button

6.2.1. Positioning actions with custom CSS

This header looks a bit weird. What we'd like it to be is this:

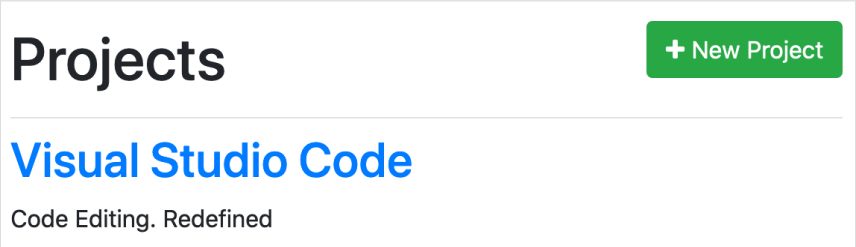


Figure 28. Our dream page header

The header over there on the left, and the actions over there on the right.

To do this, we can add a couple of lines of CSS to `app/assets/stylesheets/application.css.scss`:

```
.page-header {
  @extend .pb-2, .mt-4, .mb-2, .border-bottom, .row;

  h1,
  h2,
  h3,
  h4,
  h5,
  h6 {
    @extend .col-sm-12, .col-md-6;
  }

  ul.actions {
    @extend .col-sm-12, .col-md-6;
    @extend .list-unstyled, .list-inline;

    li {
      @extend .list-inline-item;
    }

    text-align: right;
  }
}
```

This does a couple of things:

- Provides us with a single class that we can use to apply all the page header utility styles in `.page-header`. We will change our view shortly to use this class.
- Uses Bootstrap's `.row` and `.col-*` classes to specify the widths of the headers and actions. This will ensure that they display neatly on all screen sizes. When we use `col-sm-12`, it says that on small screens that element should take up 12 columns (the entire screen). With `col-md-6`, this means on medium screens (and above) those same elements will only take up 6 columns, half the screen.
- Removes the dot to the left of the button with `.list-unstyled` on `ul.actions`.
- Makes all potential actions inline with `.list-inline`. The default behaviour is to display each of them on their own line.
- Moves all the content in `ul.actions` to the right with `text-align: right`, pushing all actions to the right-hand-side of the screen.

When we make these changes to `application.scss`, we'll need to also change the class used in `app/views/projects/index.html.erb`:

Listing 108. app/views/projects/index.html.erb

```
<div class="pb-2 mt-4 mb-2 border-bottom">
```

This should now be changed to:

```
<div class="page-header">
```

We've configured our CSS to apply the same utility classes for us (by using `@extend`), so we do not have to do that in this tag any longer.

With these changes in place, when we refresh the page we'll see our new header:

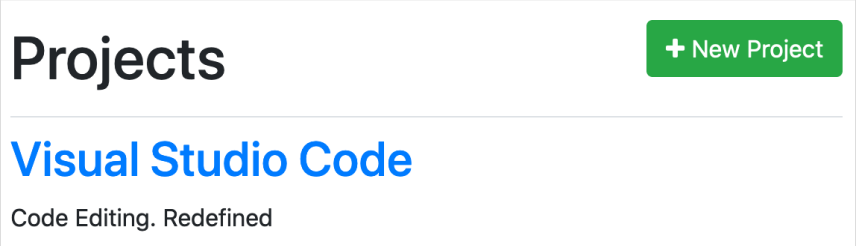


Figure 29. The final page header on the Projects index view

6.2.2. Applying the style elsewhere

You can apply similar styling to the views for the `new`, `edit` and `show` actions. The `new` and `edit` views have no action menus, so their page headers can simply be tweaked to add the `.page-header` wrapper element:

Listing 109. The new header on app/views/projects/new.html.erb

```
<div class="page-header">
  <h1>New Project</h1>
</div>
```

Listing 110. The new header on app/views/projects/edit.html.erb

```
<div class="page-header">
  <h1>Edit Project</h1>
</div>
```

6.2.3. Improving the show view

Improving the `ProjectsController#show` view is a little more work, as it has links of different types. One link edits; the other deletes. The header section of the page currently looks like this:

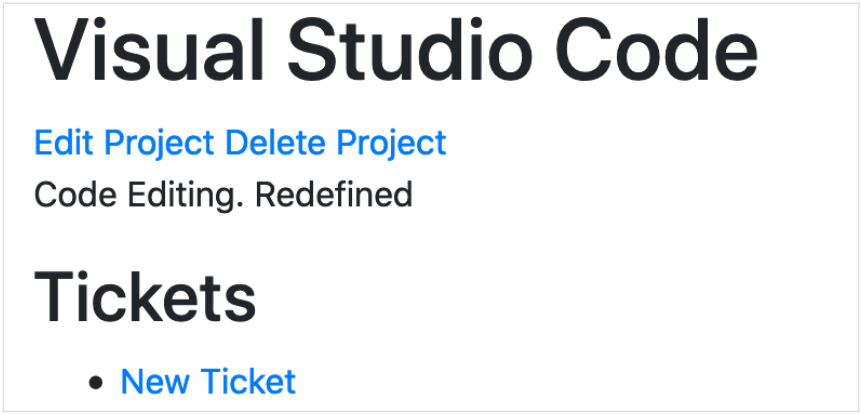


Figure 30. The unstyled page header on app/views/projects/show.html.erb

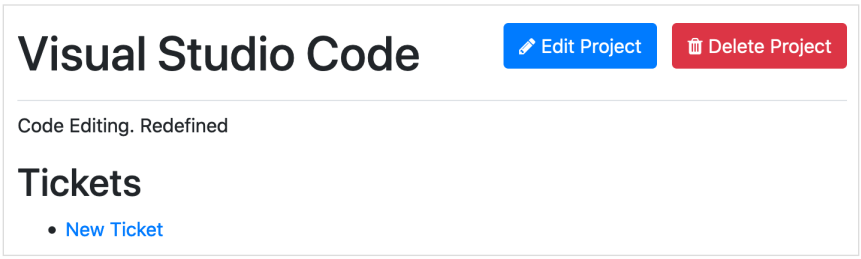
Let's change that header to look like this:

Listing 111. app/views/projects/show.html.erb

```
<div class="page-header">
  <h1><%= @project.name %></h1>

  <ul class="actions">
    <li><%= link_to fa_icon("pencil") + " Edit Project",
      edit_project_path(@project), class: "btn btn-primary" %></li>
    <li><%= link_to fa_icon("trash") + " Delete Project",
      project_path(@project),
      method: :delete,
      data: { confirm: "Are you sure you want to delete this project?" },
      class: "btn btn-danger" %></li>
  </ul>
</div>
```

Like before, there's a `<div class="page-header">` around the header, and our links are now in a list of action links. We've chosen different button styles for this page - `btn-primary` for editing, and `btn-danger` for deleting. `btn-danger` links are bright red, indicating a dangerous action. There's also some appropriate icons for the links.

These links will now be styled nicely:

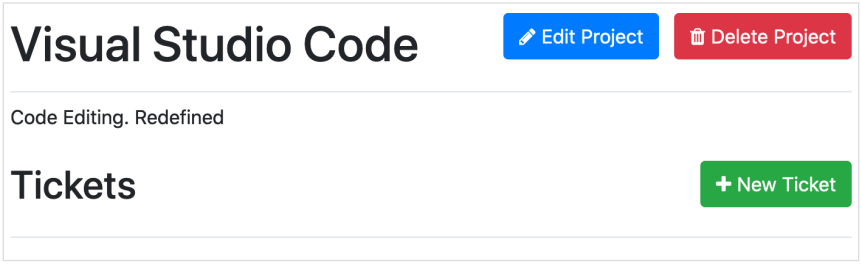


Further down on the page, we have the tickets area. Let's change this header now too:

Listing 112. app/views/projects/show.html.erb

```
<header class="page-header">
  <h2>Tickets</h2>

  <ul class="actions">
    <li>
      <%= link_to fa_icon("plus") + " New Ticket",
        new_project_ticket_path(@project),
        class: "btn btn-success" %>
    </li>
  </ul>
</header>
```

This will make the whole page look great:



Now that's all looking a lot better, but we can do a lot better in our code!

6.3. Semantic styling

As it stands at the moment, whenever we have a "New", "Edit" or "Delete" link in our application, we're going to have to add all this markup around it with the `fa_icon` and the `class` attribute. Rather than repeating all that markup, we can use semantic styling. All the "New" links are going to look the same way, all the "Edit" links are going to look the same way and all the "Delete" links are going to look the same way, so why not style them all in an easier fashion? Plus, if we decide later down the track that all "New" links should now be styled

differently, we'll only have to update the code in one place - in the stylesheet. Very DRY of us.

6.3.1. Styling buttons

To start fixing this, we can go back to `app/views/projects/index.html.erb` and change the "New Project" link to this:

Listing 113. Less presentational markup in the HTML

```
<%= link_to "New Project", new_project_path, class: "new" %>
```

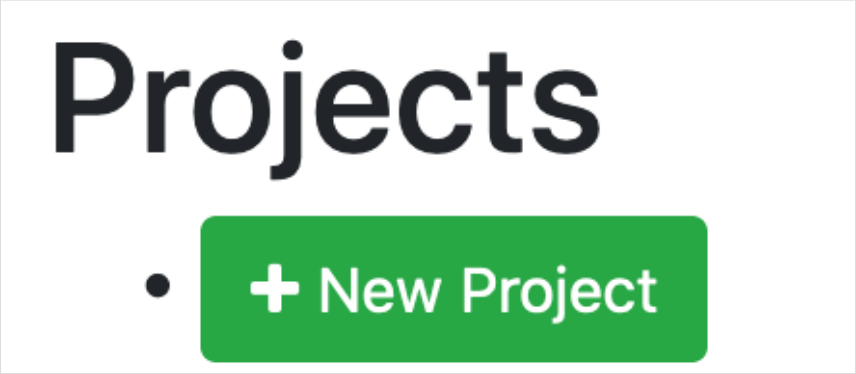
The `new` class will contain **all** the stylings for any "New" link in our application, including the icon. To make this work, we'll need to write some Sass in `app/assets/stylesheets/application.css.scss` which adds the same stylings as we had before:

Listing 114. And more presentational styling in the CSS, where it belongs

```
a.new {
  @extend .btn, .btn-success;

  &:before {
    font-family: "FontAwesome";
    @extend .fa-plus;
    padding-right: 0.5em;
  }
}
```

This new code first adds the `btn` and `btn-success` classes to this element using the `@extend` directive from Sass. This directive allows us to extend any element's styling with any other element's styles. Next, we use the `&:before` rule which allows us to place content **before** the content within an element. In this case, we're setting the `font-family` to "FontAwesome" so that it uses the icon font. Then we're using `@extend` again to add the same Plus icon that we added earlier using `fa_icon`. The final line, `padding-right`, adds padding to the right-hand-side of this element so that the icon and the "New Project" text have space between them.

If we look at our "New Project" link again, it will still have the same styles:



All of this has been accomplished with less styling in the view, and more in the CSS file where it belongs. We can use these same techniques for the "Edit" and "Delete" links inside of `app/views/projects/show.html.erb` converting them to just this:

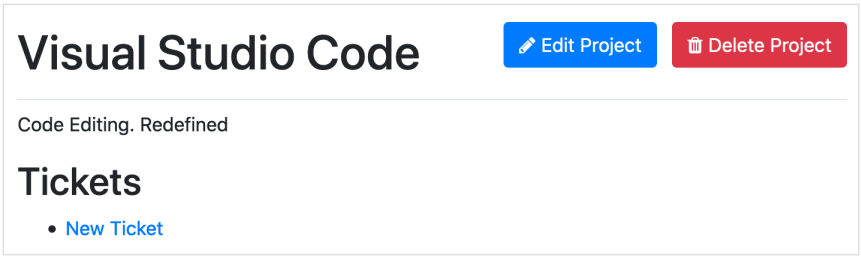
Listing 115. app/views/projects/show.html.erb

```
<ul class="actions">
  <li><%= link_to "Edit Project", edit_project_path(@project),
    class: "edit" %></li>

  <li><%= link_to "Delete Project", project_path(@project),
     method: :delete,
     data: { confirm: "Are you sure you want to delete this project?" },
     class: "delete" %></li>
</ul>
```

Next, we can add styles for the `edit` and `delete` classes to `app/assets/stylesheets/application.css.scss`, in much the same way as we added styles for the `new` class:

```
a.edit {
  @extend .btn, .btn-primary;

  &:before {
    font-family: "FontAwesome";
    @extend .fa-pencil;
    padding-right: 0.5em;
  }
}

a.delete {
  @extend .btn, .btn-danger;

  &:before {
    font-family: "FontAwesome";
    @extend .fa-trash;
    padding-right: 0.5em;
  }
}
```

If we go to our project's page once again, we'll see the "Edit Project" and "Delete Project" links have stayed the same:



We're not done yet though, there's a bit of duplication happening in this file, which we can clean up to just this:

Listing 116. Removing duplication from the `new`, `edit` and `delete` styles

```
a.new,
a.edit,
a.delete {
  @extend .btn;

  &:before {
    font-family: "FontAwesome";
    padding-right: 0.5em;
  }
}

a.new {
  @extend .btn-success;

  &:before {
    @extend .fa-plus;
  }
}

a.edit {
  @extend .btn-primary;

  &:before {
    @extend .fa-pencil;
  }
}

a.delete {
  @extend .btn-danger;

  &:before {
    @extend .fa-trash;
  }
}
```

The links with `new`, `edit` and `delete` classes will now be styled identically as buttons that use the "FontAwesome" font. From there, each different class has its button type and icon specified in different rules.

6.3.2. Styling headers

Where else can we apply this semantic styling? Well, we could replace this `<div class="page-header">` tag with something more meaningful, like a `<header>` tag. After all, if

we want to use more than one of the page headers on a single page, it's not really a page header is it! Let's make that change on all of the changes we've made it on, so far:

Listing 117. app/views/projects/index.html.erb

```
<header>
  <h1>Projects</h1>
  ...
</header>
```

Listing 118. app/views/projects/new.html.erb

```
<header>
  <h1>New Project</h1>
</header>
```

Listing 119. app/views/projects/edit.html.erb

```
<header>
  <h1>Edit Project</h1>
</header>
```

Listing 120. app/views/projects/show.html.erb

```
<header>
  <h1><%= @project.name %></h1>
  ...
</header>

...

<header>
  <h2>Tickets</h2>
</header>
```

Now we can style our new `header` tags, by changing our `.page-header` CSS selector to `header`:

Listing 121. app/assets/stylesheets/application.css.scss

```
header {
  @extend .pb-2, .mt-4, .mb-2, .border-bottom, .row;

  h1,
  h2,
  h3,
  h4,
  h5,
  h6 {
    @extend .col-sm-12, .col-md-6;
  }

  ul.actions {
    @extend .col-sm-12, .col-md-6;
    @extend .list-unstyled, .list-inline;

    li {
      @extend .list-inline-item;
    }

    text-align: right;
  }
}
```

If we refresh all our pages, they should look exactly the same as they did before - we haven't changed any styles with our `header` tag, we've just made them more semantic. Using generic `<div>` elements to represent headers is bad practice—we should use `<header>` elements because they more clearly explain what the element does.

6.3.3. Styling flash messages

The next thing that we can style is our flash messages that appear. If you create another project in the Ticketee application, you can see how plain they are:

[created flash notice] | created_flash_notice.png

To make these stand out more, we're going to apply Bootstrap's alert styling to it. Let's open `app/views/layouts/application.html.erb` and change this code:

Listing 122. Unstyled flash message output

```
<% flash.each do |key, message| %>
  <div><%= message %></div>
<% end %>
```

To this:

Listing 123. Styled flash message output

```
<% flash.each do |key, message| %>
  <div class="alert alert-<%= key %>">
    <%= message %>
  </div>
<% end %>
```

The `alert` class for each piece of the `flash` will make this object stand out more, and the `alert-<%= key %>` will use another class called `alert-notice` or `alert-alert` to color the flash message box a different color. If we look at Bootstrap's documentation for their alerts (http://getbootstrap.com/components/#alerts), we can see that `alert-notice` and `alert-alert` are not available:

Copy

```
<div class="alert alert-primary" role="alert">
  A simple primary alert—check it out!
</div>
<div class="alert alert-secondary" role="alert">
  A simple secondary alert—check it out!
</div>
<div class="alert alert-success" role="alert">
  A simple success alert—check it out!
</div>
<div class="alert alert-danger" role="alert">
  A simple danger alert—check it out!
</div>
<div class="alert alert-warning" role="alert">
  A simple warning alert—check it out!
</div>
<div class="alert alert-info" role="alert">
  A simple info alert—check it out!
</div>
<div class="alert alert-light" role="alert">
  A simple light alert—check it out!
</div>
<div class="alert alert-dark" role="alert">
  A simple dark alert—check it out!
</div>
```

In that example, we can see two that look like the kind of thing we want: `alert-success` and `alert-danger`. We can make our `alert-notice` and `alert-alert` classes use the stylings of

these two other classes with this code added to `app/assets/stylesheets/application.css.scss`:

```
.alert-notice {
  @extend .alert-success;
}

.alert-alert {
  @extend .alert-danger;
}
```

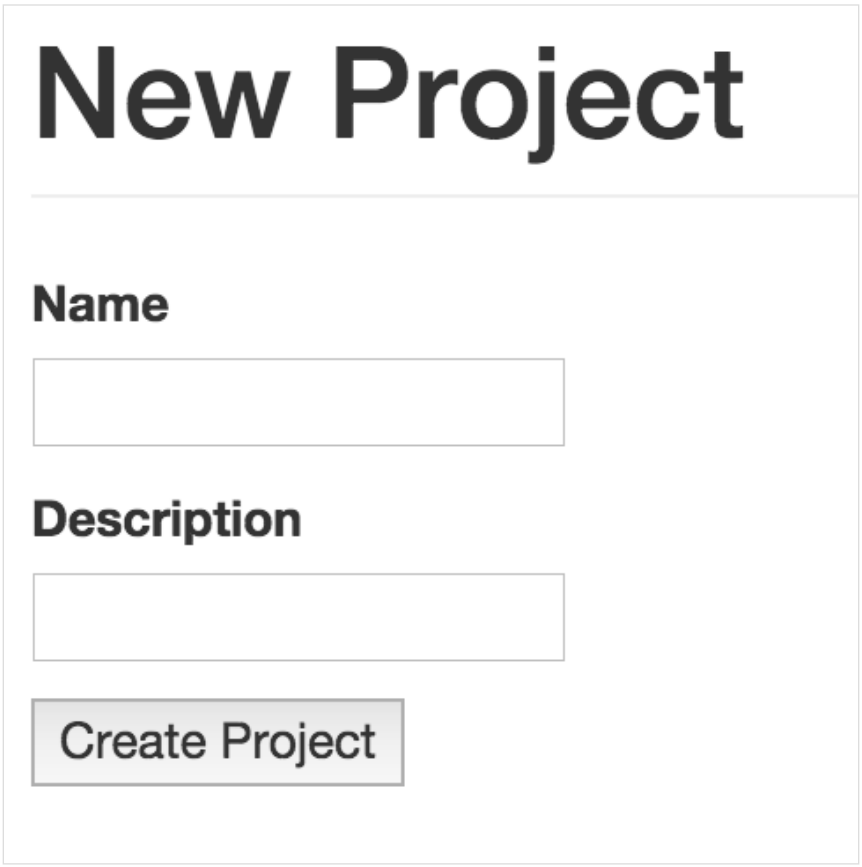
When an alert with the `class` attribute of `alert-notice` is displayed, it will be styled as if it had a `class` attribute of `alert-success`. This is thanks to the `@extend` directive in Sass, which we used earlier with the "New", "Edit" and "Delete" links.

When we create a project again, we should see a much nicer styled flash messages:

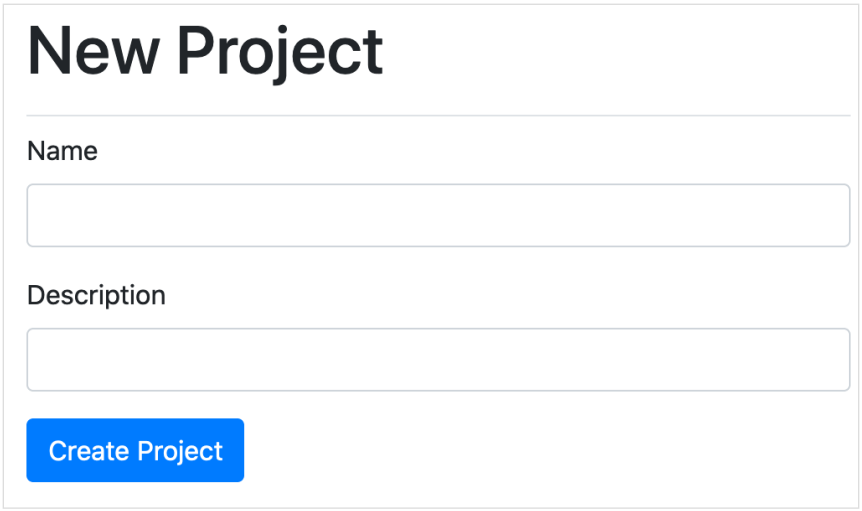
That's much nicer! Let's see what it looks like when we force a validation error by not entering a name for a new project:

That's looking good too! That's all for our flash stylings. We're making some really good progress.

The next thing we'll do is re-style our projects form to take it from this:



To this:



6.4. Using Bootstrap Form

To change the form to its new styling, we're going to enlist the help of another gem called `bootstrap_form`, which provides not only a shorter syntax for rendering forms, but also has

Bootstrap integration. Let's add this gem now:

```
bundle add bootstrap_form -v 4.5.0
```

Next, you'll need add the import line for this gem to `application.scss`:

Listing 124. app/assets/stylesheets/application.scss

```
@import "rails_bootstrap_forms";
```

Next, we can change our forms to use this new gem. We'll do both the projects and tickets form.

6.4.1. Updating the styling of the project form

Over in `app/views/projects/_form.html.erb`, let's change this code:

```
<%= form_with(model: project, local: true) do |form| %>
```

To this:

```
<%= bootstrap_form_with(model: project, local: true) do |form| %>
```

This will tell the view to use the form builder from Bootstrap Form, rather than the one that's built into Rails. Next, we can simplify the code used for rendering the `name` and `description` fields for our project form, turning it from this:

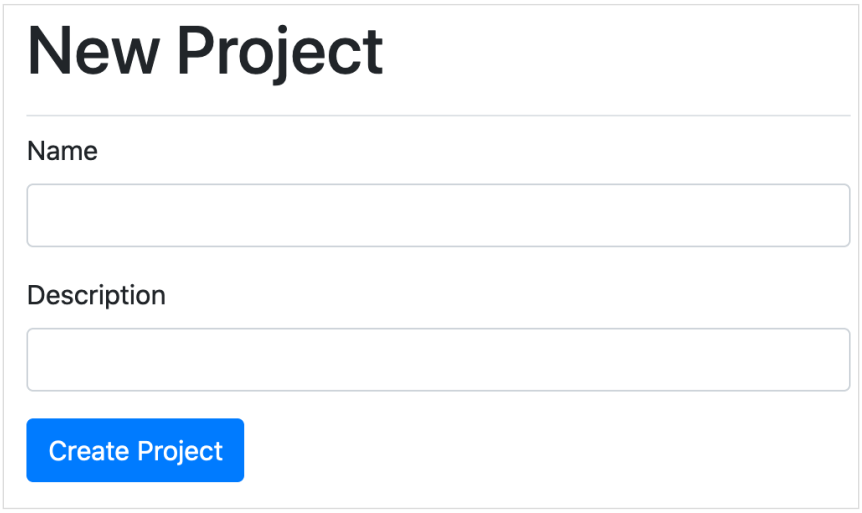
```
<div>
  <%= form.label :name %>
  <%= form.text_field :name %>
</div>

<div>
  <%= form.label :description %>
  <%= form.text_field :description %>
</div>

<%= form.submit %>
```

To this:

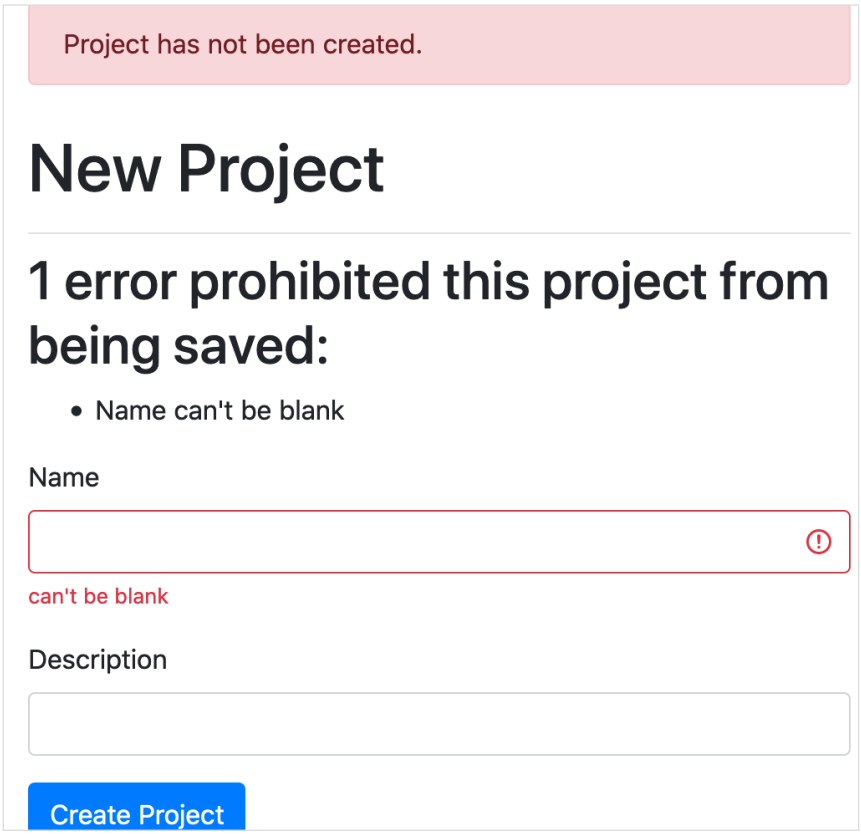
```
<%= form.text_field :name %>
<%= form.text_field :description %>

<%= form.primary %>
```

We don't need to use the surrounding `<div>` tags any more or to provide the labels for the fields. Bootstrap Form takes care of all of that for us with the `input` helper. It also generates a button for us using its `primary` helper which operates similarly to the old `f.submit` method, but styles it using Bootstrap's default styles—adding a `btn` and `btn-primary` class.

If you refresh the form page now, the form will now look like this:



It's looking good. But if you submit the form with some validation errors, you'll get a nasty shock:



We now have two sets of error messages, one styled, and one not! We only put one set of errors on the page - the top "1 error prohibited this message from being saved" set. The second set of errors, next to the fields they relate to, are provided by the `bootstrap_form` gem. They're easier for users to understand - if you had a long form and you wanted to know which fields you have errors for, it might require scrolling up and down. So we can just use the errors from `bootstrap_form`, and delete the error messages we added.

Open up `app/views/projects/_form.html.erb`, and delete the whole errors block so that it just looks like this:

Listing 125. The new _form.html.erb partial, without duplicate error messages

```
<%= bootstrap_form_with(model: project, local: true) do |form| %>
  <%= form.text_field :name %>
  <%= form.text_field :description %>

  <%= form.primary %>
<% end %>
```

The code is much simpler than the original form partial we started off with, and the unsightly

double errors are now gone!

This has an unfortunate side effect though, as you'll see if you run the spec for creating projects, with `bundle exec rspec spec/features/creating_projects_spec.rb`:

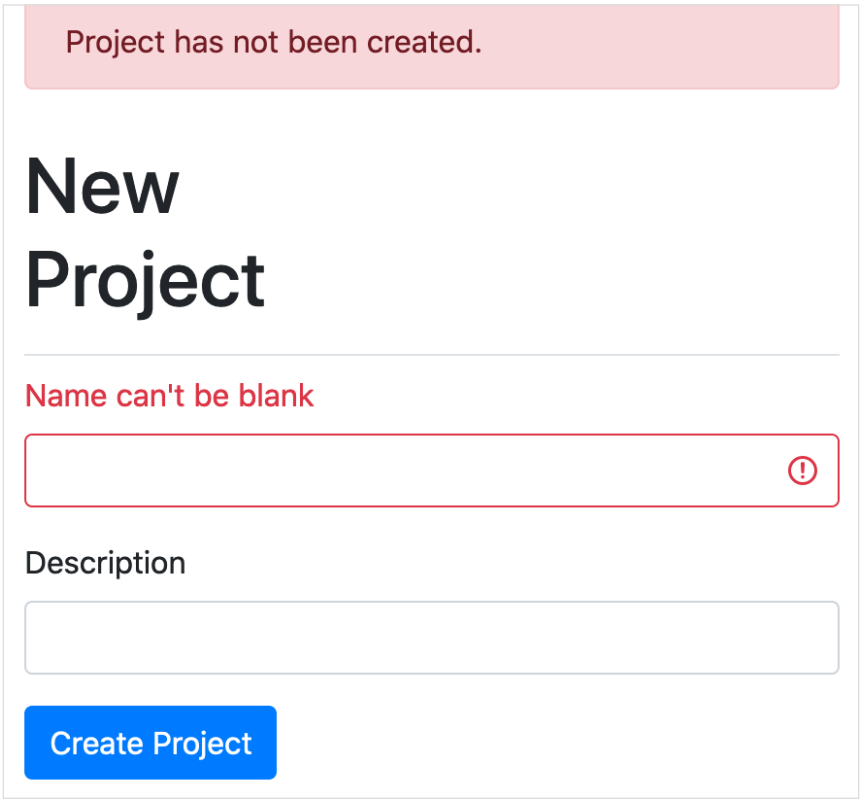
```
1) Users can create new projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Name can't be blank"
     expected to find text "Name can't be blank" in "Project has not
     been created. New Project * Namecan't be blank Description"
```

Oops. We were checking for the presence of an error message that we just deleted, and by default `bootstrap_form` displays shortened error messages, such as "can't be blank". Luckily, it's easy to configure Bootstrap Form to use full error messages that include the name of the field.

We can do this by changing our form to this:

```
<%= bootstrap_form_with(model: project, local: true, label_errors: true) do |form| %>
  <%= form.text_field :name %>
  <%= form.text_field :description %>

  <%= form.primary %>
<% end %>
```

This will make the error now appear in the label instead of underneath the input:



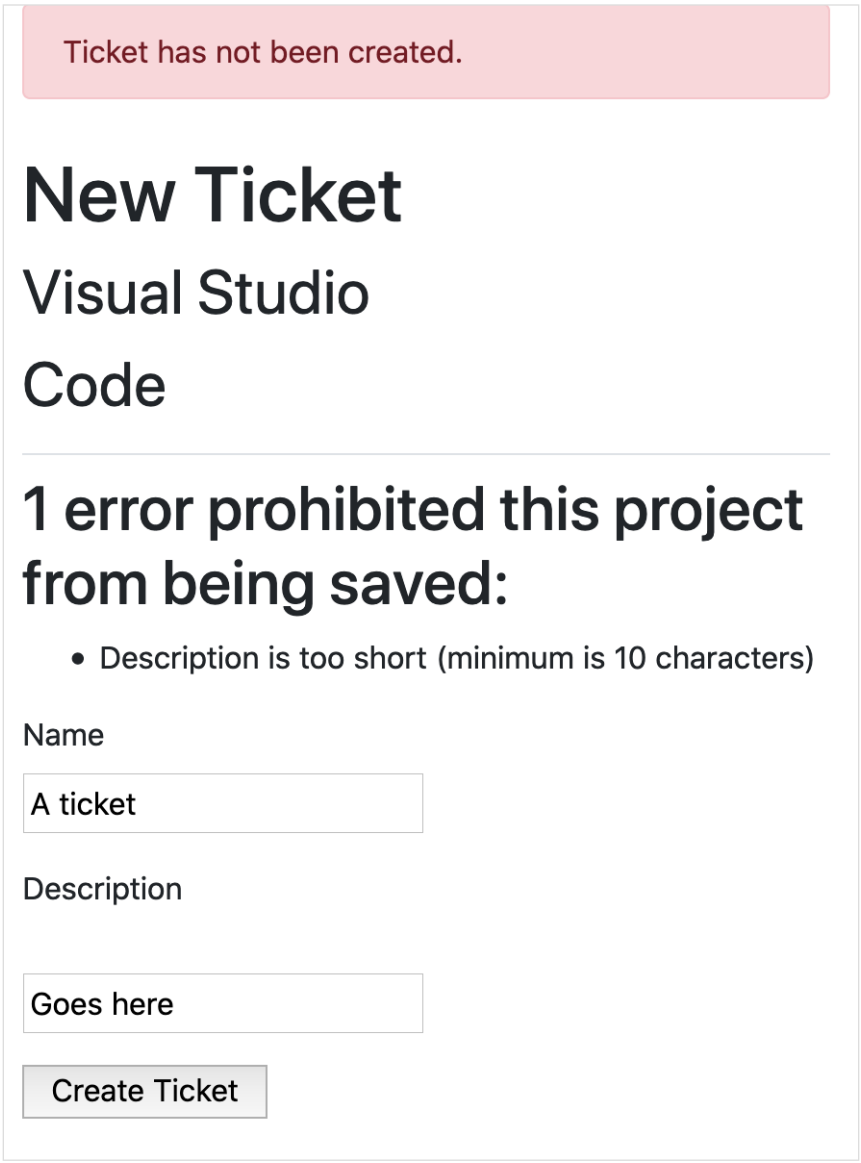
This will also fix the broken spec - you can verify this by running `bundle exec rspec spec/features/creating_projects_spec.rb`:

```
2 examples, 0 failures
```

Great! Now all of the project pages in our application are looking good.

6.4.2. Updating the styling of the ticket form

Our ticket form is currently not styled the same way as our project form:



To fix this, we'll change the ticket form partial from this:

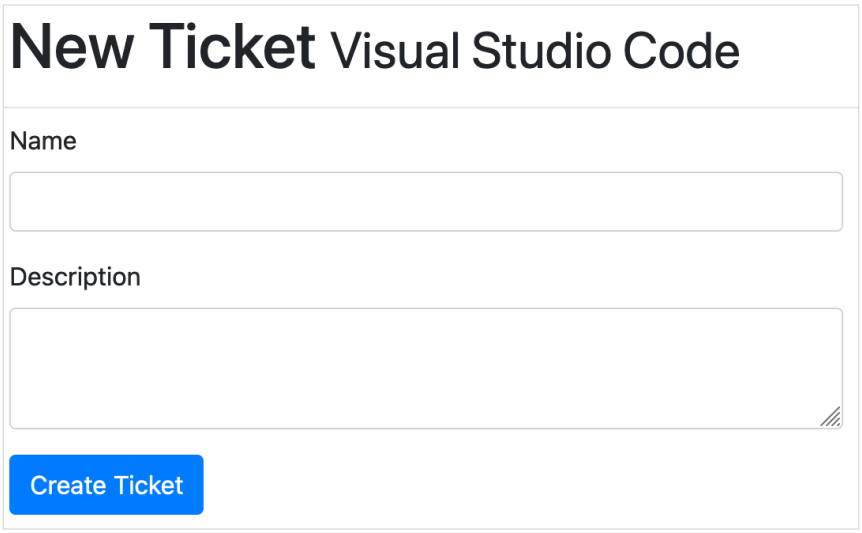
Listing 126. app/views/tickets/_form.html.erb

```
<%= form_with(model: [project, ticket], local: true) do |form| %>
  <% if ticket.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(ticket.errors.count, "error") %>
      prohibited this project from being saved:</h2>

      <ul>
        <% ticket.errors.full_messages.each do |msg| %>
          <li><%= msg %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
  <p>
    <%= form.label :name %><br>
    <%= form.text_field :name %>
  </p>
  <p>
    <%= form.label :description %><br>
    <%= form.text_area :description %>
  </p>
  <%= form.submit %>
<% end %>
```

To this:

```
<%= bootstrap_form_with(model: [project, ticket], local: true, label_errors: true) do |form|
%>
  <%= form.text_field :name %>
  <%= form.text_area :description %>

  <%= form.primary %>
<% end %>
```

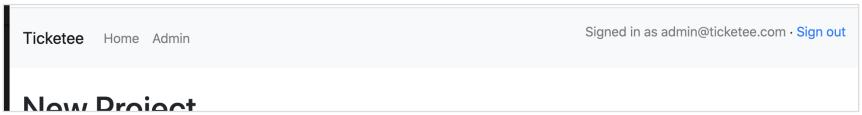
This form will now look like this:



Great!

6.5. Adding a navigation bar

We'll do one more thing and then wrap up here: add a navigation bar to the top of our application's layout. It will look like this:



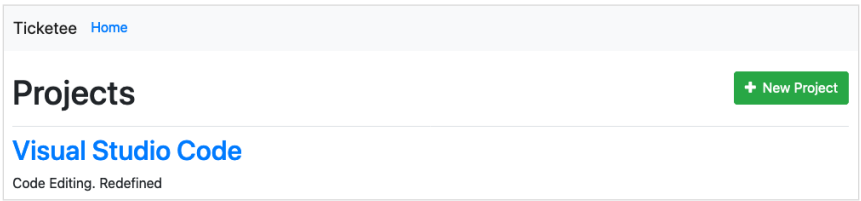
This is by far the easiest part of the Bootstrap work that we've done so far; all we need to do is to add this content above the flash messages (above the `<div class="container-fluid">` in our application's layout):

Listing 127. app/views/layouts/application.html.erb

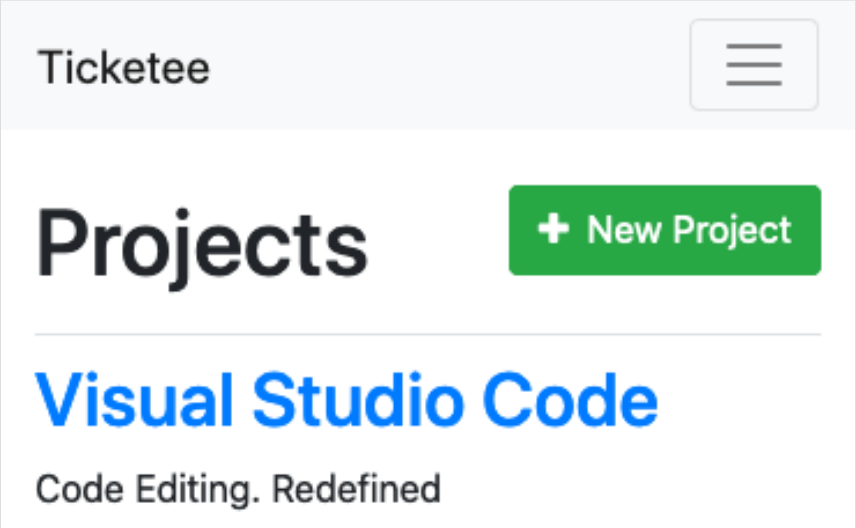
```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <%= link_to "Ticketee", root_path, class: "navbar-brand" %>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#collapse"
aria-controls="collapse" aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" id="collapse">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item <%= "active" if current_page?("/") %>">
        <%= link_to "Home", root_path, class: "nav-link" %>
      </li>
    </ul>
  </div>
</nav>
```

The navbar contains two different parts: a header and the navigation itself. The header contains a link which shows the "Ticketee" text, and when that link is clicked it will take the user back to the homepage. In the navigation, we add a "Home" link, just in case people don't realise that clicking "Ticketee" will take them back to the homepage. If the user is currently at the `/` path, then an extra class called `active` is added to the `li` for that link, turning it a different colour.

If you go to the application's homepage, you'll now see the navbar:



We've also elected to use the responsive navbar, meaning that it will display nicely on both large and small screens. You can test this out by resizing your browser, wider and smaller. When the window gets below 990px wide, the navbar will automatically switch to its "small" display:



It looks good! But the button at the top-right, which is supposed to cause the menu to expand, doesn't work yet - it uses JavaScript to show and hide the menu contents, and we haven't included Bootstrap's JavaScript into our application yet. If you click that button, nothing will happen at the moment.

Like we included Bootstrap's CSS into our `application.css.scss` file, we also need to include its JavaScript. Open up `app/javascript/packs/application.js`, and check out what it looks like. We haven't modified it yet, so at the bottom it will have four important lines:

Listing 128. The important parts of the default `application.js` file

```
require("@rails/ujs").start();
require("turbolinks").start();
require("@rails/activestorage").start();
require("channels");
```

These lines require all the main JavaScript features that Rails uses. We won't concern ourselves with these yet. We'll look into some of them later on.

All that we need to do here is to add Bootstrap's JavaScript:

```
require("bootstrap");
```

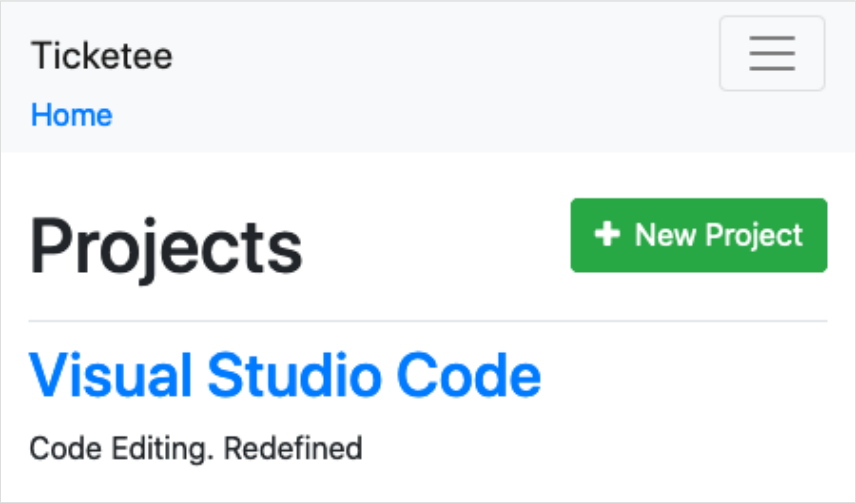
Next up, we will need to add Bootstrap's JavaScript dependencies, which we do by running this command in our terminal:

```
yarn add bootstrap popper.js jquery
```

The `popper.js` and `jquery` JavaScript packages are other packages that Bootstrap relies on for its functionality. When these packages are installed, we will need to add `require` lines for these to `application.js` too:

```
import 'jquery'
import 'popper.js'
import 'bootstrap'
```

Now when you refresh your browser, and press the little three-bar icon[50] the menu should expand and contract, like in the following screenshot:



While the navbar is a little bare at the moment, we will be filling it out in future chapters.

6.6. More responsive styling

We've fixed the top navigation bar to be fully responsive, but it would be great if our whole app looked great on-the-go, from a mobile phone, tablet, or any other device. Let's look at implementing that now.

The first thing we need to do is include a special `meta` tag in our document, to ensure that responsive styles get picked up by mobile devices. Without this tag, you'd see the same layout as you do on a desktop, just very very small to fit on the smaller screens.

Add the following line of text just inside the `<head>` section of the page, in

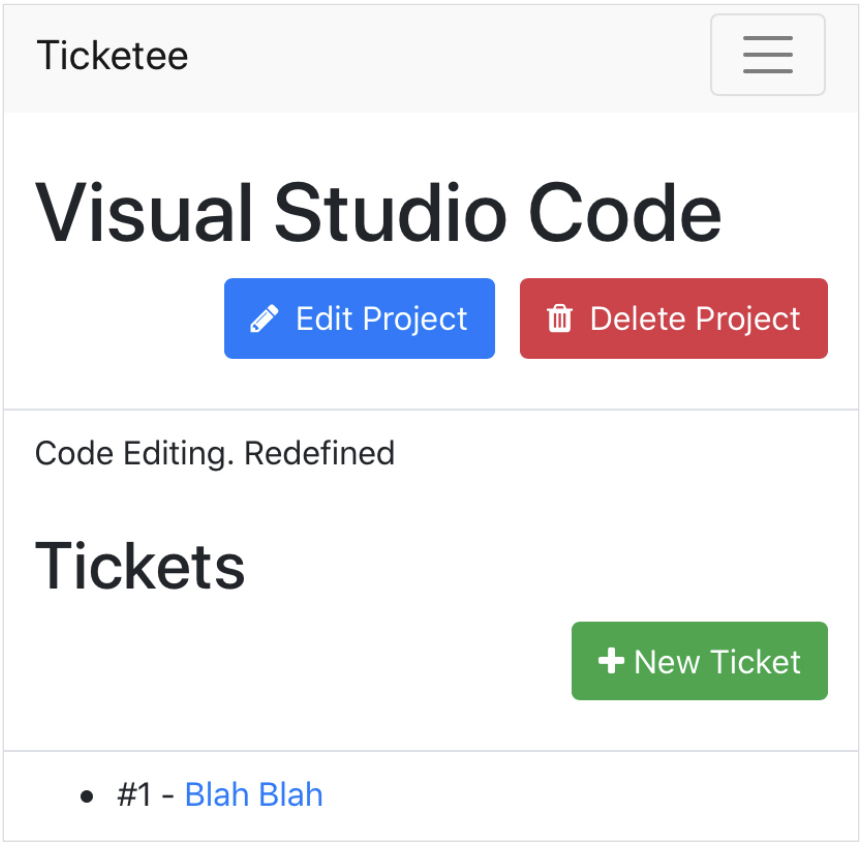
app/views/layouts/application.html.erb:

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Some background information on this `meta` tag can be found on the Mozilla Developer Network[51]. For now, we don't have to know exactly how it works, but now we know that it does work - our mobile browser won't artificially pretend it has more pixels than it has, to render larger layouts in smaller spaces.

Now our pages will use properly responsive styles on mobile. The way this works is with a CSS feature called media queries. Bootstrap uses media queries internally for lots of things, like the navbar styles we looked at earlier. If the screen is larger, the links in the bar appear in a line; if its smaller, they appear hidden, but then on their own line when revealed. We can do a similar sort of thing, using Bootstrap's own defined styles.

Here's what our project page will look like on an iPhone:



Whew, that was a lot of styling work! That completes all of our Bootstrap additions for the time being. Throughout the remainder of the book, we'll be using features of Bootstrap as

they’re needed to improve the design of our application, but we’ve got the basic foundation in place.

Let’s commit and push our recent changes now:

```
$ git add .
$ git commit -m "Added Bootstrap for styling"
$ git push
```

[44] You could use other things such as Tailwind, http://tailwindcss.com, or any one of the other CSS frameworks out there. We're recommending Bootstrap out of preference and ease-of-use.
[45] This includes **a lot** of CSS and probably includes pieces that you won't ever use. If you're concerned about how much this is including, you can pick and choose `@import` lines from here (https://git.io/JLhpz) to only include the parts you want.
[46] Read about Bootstrap container styles here: https://getbootstrap.com/docs/4.5/layout/overview/
[47] Read about Bootstrap utility classes here: https://getbootstrap.com/docs/4.4/utilities/borders/
[48] The other button types can be found here: https://getbootstrap.com/docs/4.4/components/buttons/
[49] Font Awesome's site can be found here: http://fortawesome.github.io/Font-Awesome
[50] Or "hamburger" icon, as we often see it called.
[51] https://developer.mozilla.org/en/docs/Mozilla/Mobile/Viewport_meta_tag

Chapter 7. Authentication

You've created two resources for your Ticketee application: projects and tickets. And in the last chapter you added some CSS stylings to make things look good. Now you'll add authentication to let users sign in to your application.

With this feature, you can track which tickets were created by which users. A little later, you'll use these user records to allow and deny access to certain parts of the application. The general idea behind having users for this application is that some users are in charge of creating projects (project owners) and others use whatever the projects provide. If they find something wrong with it or wish to suggest an improvement, filing a ticket is a great way to inform the project owner about their request. To round out the chapter, we'll link tickets to the users who created them. This way, anyone viewing a ticket can know exactly who created it, rather than it just be yet another ticket in the application.

In this chapter, you'll add authentication to your application using a gem called Devise[52]. Devise has been proven time and time again to be a capable gem for authentication, and so that is what we'll be using here. Most of the functionality for this chapter will come from within Devise itself.

7.1. Using Devise

Devise is a gem which provides the authentication features that nearly every Rails application needs, such as user registration, sign in, password reset emails and confirmation emails. We're going to cover the first two of those in this chapter.

When a user signs up with Devise, their credentials are stored securely in a database using industry-standard cryptography. If we were to build authentication ourselves, then the cryptography methods that we choose may not be as strong. Devise saves us from having to worry about these things.

Let's install the Devise gem now by adding it as a dependency of our application:

```
bundle add devise -v '~> 4.7.3'
```

Next, we'll need to run the generator which will install Devise:

```
$ rails g devise:install
```

This generator will create an initializer at `config/initializers/devise.rb`, which contains the configuration for Devise. The files in `config/initializers` are run during the process of booting a Rails application and are used to setup anything that is necessary for the application to run. In this case, Devise's configuration sets the scene for Devise use later on in our application.

The `devise:install` generator makes our application ready for Devise by setting up some default configuration, but it's the `devise` generator which does the hard work of adding in the major pieces. Let's run this command now:

```
$ rails g devise user
```

This generator generates a `User` model, which will be used to keep track of users within our application. Along with this comes a migration to generate a `users` table, so let's run this command to apply this new migration to our database:

```
$ rails db:migrate
```

This model contains configuration specific to Devise:

Listing 129. app/models/user.rb

```
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable, :trackable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
end
```

The `devise` method in this model sets up the model to use the specified Devise modules. By default the modules and features that Devise provides with this default config are:

- `database_authenticatable` - Allows the user to sign in to your app using credentials stored in the database, such as an email address and password.
- `registerable` - Users can register or sign up to our application.

- `recoverable` - If a user forgets their password, they can reset it via email.
- `rememberable` - A user's session in our application will be remembered. This means they won't have to sign in every time they restart their browser.
- `validatable` - Validates the user's email address and password length. By default, passwords are expected to be between 8-128 characters. This setting can be changed in `config/initializers/devise.rb` by altering the `config.password_length` value.

As you can see just from this list, Devise offers us quite a lot! It even has more, as listed in the comment above the `devise` method:

- `confirmable` - Requires a user to confirm their email address by clicking a link in a confirmation email, before they can sign in.
- `lockable` - Provides extra security by automatically locking accounts after a given number of failed sign in attempts.
- `timeoutable` - Provides extra security by automatically logging out users who haven't been active in a given amount of time.
- `omniauthable` - Adds support for Omniauth[53], which will allow user to authenticate with your app via an external service, such as Facebook or Twitter.
- `trackable` - Tracks information such as last sign in time and IP for each user.

We won't be using any of these advanced modules, but it's good to know that they're there, and they're built in and well supported.

Beware race conditions with a uniqueness validator

One of the validations that Devise's `validatable` module adds is a uniqueness rule for email addresses, so two people can't sign up with the same email address, or an existing user can't change their email address to be the same as another user. The code for that validation looks like this:

```
validates :email, uniqueness: { allow_blank: true, if: :email_changed? }
```

This uniqueness validator works by checking to see whether any records matching the validation criteria exist in the database already. In our case, the validator checks if there are any `User` records with the same email address as this user. If no such records exist, then the validation passes.

A problem arises if two connections to the database both make this check at almost exactly the same time. Both connections will claim that no such records exist, and therefore each will pass validation and allow the record to be saved, resulting in non-unique records.

A way to prevent this is to use a database uniqueness index so that the database, not Rails, does the uniqueness validation. For information how to do this, consult your database's manual.

Although this problem doesn't happen all the time, it can happen, especially on more larger and more popular sites, so it's something to watch out for.

The `devise` generator also adds a line to `config/routes.rb`:

```
devise_for :users
```

One little line like this generates a bunch of routes for our application, which we can see when we run `rails routes -c devise`:

Prefix	Verb	URI Pattern	Controller#Action
new_user_session	GET	/users/sign_in	devise/sessions#new

Prefix	Verb	URI Pattern	Controller#Action
user_session	POST	/users/sign_in	devise/sessions#create
destroy_user_session	DELETE	/users/sign_out	devise/sessions#destroy
new_user_password	GET	/users/password/new	devise/passwords#new
edit_user_password	GET	/users/password/edit	devise/passwords#edit
user_password	PATCH	/users/password	devise/passwords#update
	POST	/users/password	devise/passwords#create
cancel_user_registration	GET	/users/cancel	devise/registrations#cancel
new_user_registration	GET	/users/sign_up	devise/registrations#new
edit_user_registration	GET	/users/edit	devise/registrations#edit
user_registration	PATCH	/users	devise/registrations#update
	DELETE	/users	devise/registrations#destroy

These routes are all for controllers within Devise. Devise is not only a gem, but also a Rails engine[54]. This means that it contains its own set of controllers and views which exist outside of the application. This keeps our application's code separate from Devise, giving us less code

to manage overall.

With Devise installed and configured, let's go about adding the ability for users to sign up in our application with Devise.

7.2. Adding sign up

Users will be able to sign up in our application by clicking a link in Ticketee's navigation bar called "Sign up". When they click that link, they'll see this page:

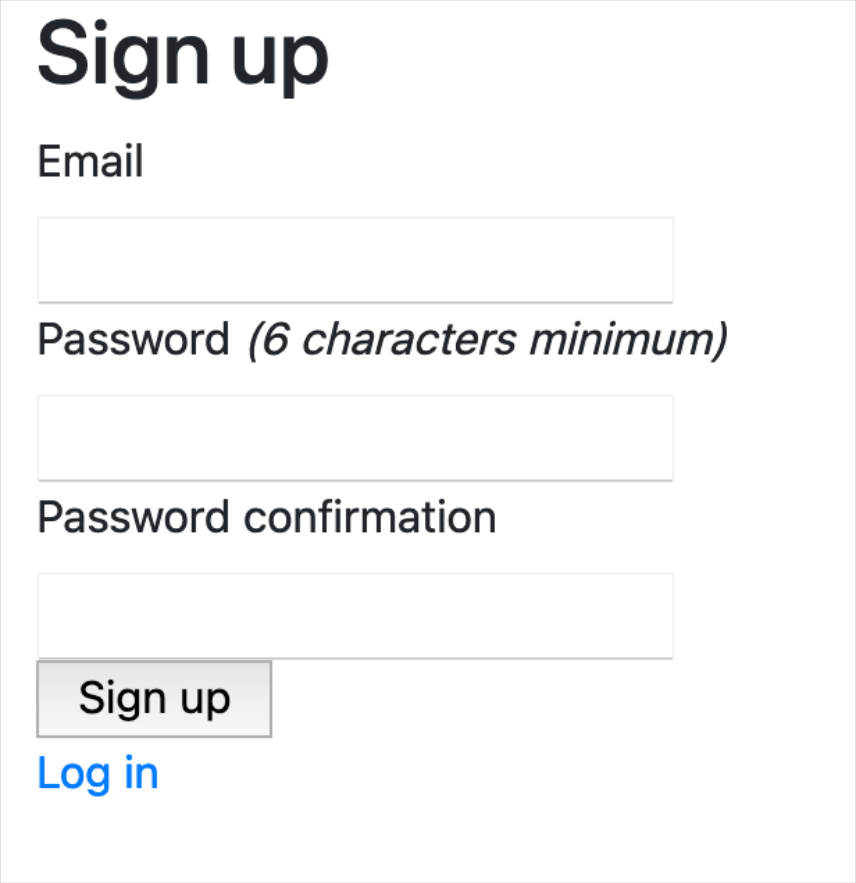


Figure 31. The sign up form

You can see this page by going to http://localhost:3000/users/sign_up now, if you wish.

From here, they'll be able to enter their email address and password, and sign up to our application. From then on, they can come back to our application and sign in and use the application to their heart's content.

To make sure that this feature works, we're going to write a test for it in

spec/features/signing_up_spec.rb, using the code from the following listing:

Listing 130. spec/features/signing_up_spec.rb

```
require "rails_helper"

RSpec.feature "Users can sign up" do
  scenario "when providing valid details" do
    visit "/"
    click_link "Sign up"
    fill_in "Email", with: "test@example.com"
    fill_in "Password", with: "password"
    fill_in "Password confirmation", with: "password"
    click_button "Sign up"
    expect(page).to have_content("You have signed up successfully.")
  end
end
```

While this might seem silly - after all, Devise provides all of this functionality to us, and Devise already has its own tests - this can prevent very silly mistakes, such as changing the view and accidentally introducing a bug that prevents people from signing up.[55] Besides, the test is very straightforward - it just walks through the process that we just described. It navigates to the homepage, clicks a "Sign up" link and then proceeds to sign up.

Now when we run our test with `bundle exec rspec spec/features/signing_up_spec.rb` we'll see this error:

```
1) Users can sign up when providing valid details
   Failure/Error: click_link "Sign up"
   Capybara::ElementNotFound:
     Unable to find link "Sign up"
```

This one is easy enough to fix. We're just missing a link to "Sign up" in our application. We'll add this to the navigation bar in app/views/layouts/application.html.erb, underneath the "Home" link we already have there.

```
<li class="<%= "active" if current_page?("/users/sign_up") %>">
  <%= link_to "Sign up", new_user_registration_path %>
</li>
```

The new_user_registration_path helper is provided by Devise, and you can see it and its brethren by running `rails routes -c devise`, as we showed above. We're not just pulling

these out of the air here!

With that link in place, our test should run a little further:

```
1 example, 0 failures
```

Oh that's surprising! It's all passed. The only thing that we needed to do was to add the "Sign up" link. Devise provides us with the rest.

Let's run all of our tests now to ensure that we haven't broken anything. Run `bundle exec rspec` to see this:

```
16 examples, 0 failures, 1 pending
```

We have one pending spec at `spec/models/user_spec.rb`, which came from the `devise` generator. Let's remove this and re-run `bundle exec rspec`:

```
15 examples, 0 failures
```

That's better! With that all done, let's make a commit:

```
$ git add .
$ git commit -m "Added Devise + sign up feature"
$ git push
```

With users able to sign up to our application, the next thing that we can add is the ability for them to sign in.

7.3. Adding sign in and sign out

Devise allowed us to easily add a sign up feature to our application. Now let's see about adding a way for users to sign in and out of our application. To start with again, we'll add a new feature spec at `spec/features/signing_in_spec.rb` using the code from the following listing.

Listing 131. spec/features/signing_in_spec.rb

```
require "rails_helper"

RSpec.feature "Users can sign in" do
  let!(:user) { FactoryBot.create(:user) }

  scenario "with valid credentials" do
    visit "/"
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: "password"
    click_button "Log in"

    expect(page).to have_content "Signed in successfully."
    expect(page).to have_content "Signed in as #{user.email}"
  end
end
```

If this test looks very similar to our "Sign up" feature, that's because it is! The two flows are very similar. In this test, the difference is that we're creating a user using a `FactoryBot` factory, and then signing in as that user.

When we run the test with `bundle exec rspec spec/features/signing_in_spec.rb` we'll see that the user factory is missing:

```
1) Users can sign in with valid credentials
   Failure/Error: let!(:user) { FactoryBot.create(:user) }
   KeyError:
     Factory not registered: "user"
```

Let's create this new factory file at `spec/factories/users.rb`:

Listing 132. spec/factories/users.rb

```
FactoryBot.define do
  factory :user do
    sequence(:email) { |n| "test#{n}@example.com" }
    password { "password" }
  end
end
```

This factory can be used to create new users in our tests. The `sequence` method will generate

sequential email addresses for our users, such as "test1@example.com" and "test2@example.com". We're doing this so that each user has a unique email address, and that will keep Devise's unique email validation happy.

When we run our test again, we'll see that it can't find the "Sign in" link:

```
1) Users can sign in with valid credentials
   Failure/Error: click_link "Sign in"
   Capybara::ElementNotFound:
     Unable to find link "Sign in"
```

Let's add this underneath the "Sign up" link in `app/views/layouts/application.html.erb`:

Listing 133. app/views/layouts/application.html.erb

```
<li class="<%= "active" if current_page?("/users/sign_in") %>">
  <%= link_to "Sign in", new_user_session_path %>
</li>
```

The `new_user_session_path` is another routing helper provided by Devise, this time to a `SessionsController`. When we run our test again, it will go all the way up to the last step:

```
1) Users can sign in with valid credentials
   Failure/Error: expect(page).to have_content "Signed in as
   #{user.email}"
     expected to find text "Signed in as test1@example.com" in "Ticketee
     Toggle navigation Home Sign up Sign in Signed in successfully..."
```

This final line of the feature is checking that a message on the page indicates to the user which email address they've used to sign in. This can be useful in situations where a computer may be shared. We're going to put this line in `app/views/layouts/application.html.erb`, but we don't want it to show all the time. Conversely, it's not useful for the sign in or sign up links to appear when the user has already signed in. Therefore we'll hide those links when the user is signed in, and replace them with this "Signed in as..." message.

Let's do this by changing this content in `app/views/layouts/application.html.erb`:

Listing 134. Showing the "Sign up" and "Sign in" links to all users

```
<li class="<%= "active" if current_page?("/users/sign_up") %>">
  <%= link_to "Sign up", new_user_registration_path, class: "nav-link" %>
</li>
<li class="<%= "active" if current_page?("/users/sign_in") %>">
  <%= link_to "Sign in", new_user_session_path, class: "nav-link" %>
</li>
```

To this:

Listing 135. Showing the "Sign up" and "Sign in" links to only non-signed-in users

```
<% unless user_signed_in? %>
  <li class="<%= "active" if current_page?("/users/sign_up") %>">
    <%= link_to "Sign up", new_user_registration_path, class: "nav-link" %>
  </li>
  <li class="<%= "active" if current_page?("/users/sign_in") %>">
    <%= link_to "Sign in", new_user_session_path, class: "nav-link" %>
  </li>
<% end %>
```

Then immediately after the `ul.nav.navbar-nav` tag that those `li` tags are contained within, add this code:

Listing 136. Showing the currently-signed-in user's email address

```
<% if user_signed_in? %>
  <div class="navbar-right">
    <p class="navbar-text">
      Signed in as <%= current_user.email %>
    </p>
  </div>
<% end %>
```

This new code uses two new methods: `user_signed_in?` and `current_user`. Both of these methods are provided to us by Devise and both methods do exactly as they say. The `user_signed_in?` method returns `true` if the user is signed in, otherwise it returns false. The `current_user` method will return either a `User` instance which represents the current user, or `nil` if the user isn't signed in. With these two methods we've hidden the "Sign up" and "Sign in" links and we've shown the "Signed in as..." message on the right hand side of the navbar, when there is a user signed in.

The navbar will now look like this when a user is signed in:

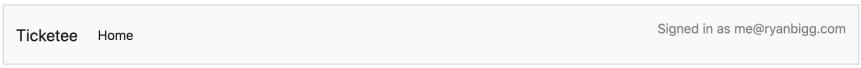


Figure 32. "Signed in as" message in the navbar

Let's run our test again with `bundle exec rspec spec/features/signing_in_spec.rb`. This time it will pass:

```
1 example, 0 failures
```

Excellent! This is the sign in part of this section done, now let's add the sign out part of this task.

7.3.1. Adding sign out

Before we write any code to build this sign out functionality, let's write a test for this using the code from the following listing:

Listing 137. spec/features/signing_out_spec.rb

```
require "rails_helper"

RSpec.feature "Signed-in users can sign out" do
  let!(:user) { FactoryBot.create(:user) }

  before do
    login_as(user)
  end

  scenario do
    visit "/"
    click_link "Sign out"
    expect(page).to have_content "Signed out successfully."
  end
end
```

This test is a fairly simple one that re-uses a bit of code from our sign in feature, but with a subtle twist. There is now a `login_as` call in `before` block. This `login_as` method doesn't come from Devise, but rather a gem Devise uses called Warden. Warden provides the user session management, whereas Devise provides the pretty face for it all. The `login_as` method will log in a user without us having to walk through the whole sign in process ourselves.

When we're done with this feature, we'll have a sign out link in our application that looks like this:

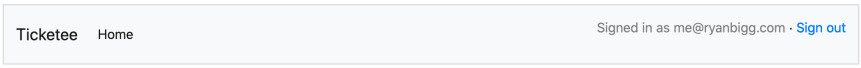


Figure 33. The sign out link

The `login_as` method isn't defined for our tests yet though, as we'll see if we try to run this test with `rspec spec/features/signing_out_spec.rb`:

```
1) Signed-in users can sign out
   Failure/Error: login_as(user)
   NoMethodError:
     undefined method `login_as' for #<RSpec::ExampleGroups::SignedIn...
```

This method isn't included automatically by Warden, so we will need to include the module that defines it manually. We can do this in `spec/rails_helper.rb`, the file that defines all of the configuration for our Rails tests, by putting this code at the bottom of the `RSpec.configure` block:

Listing 138. spec/rails_helper.rb, Configuring Warden for usage in feature specs

```
RSpec.configure do |config|
  ...
  config.include Warden::Test::Helpers, type: :feature
  config.after(type: :feature) { Warden.test_reset! }
end
```

The `include` method will include the specified module into our tests, and the `type` option passed to it will make it so that this module is only included into tests which reside in `spec/features`. We also need to tell Warden to reset itself after each test, which is done with the second line.

With those lines now in place, when we run our test it will complain that it can't find the "Sign out" link:

```
1) Signed-in users can sign out
   Failure/Error: click_link "Sign out"
   Capybara::ElementNotFound:
     Unable to find link "Sign out"
```

Let's add this link next to the "Signed in as ..." message in `app/views/layouts/application.html.erb`:

Listing 139. Adding the "Sign out" link after the "Signed in as..." text

```
<% if user_signed_in? %>
  <div class='navbar-text'>
    Signed in as <%= current_user.email %> &middot;
  </div>
  <%= link_to "Sign out", destroy_user_session_path, method: :delete, class: "btn btn-
outline-danger ml-1" %>
<% end %>
```

This will now make our navbar look like this:

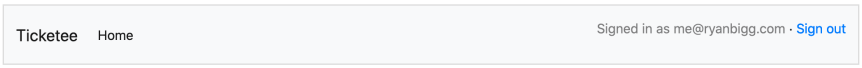


Figure 34. Signout link

When we run our test again with `bundle exec rspec spec/features/signing_out_spec.rb`, we can see that Devise has—for the third time in a row—taken care of the hard work. All we needed to provide was the link and now our test passes:

```
1 example, 0 failures
```

Great to see. Let's run all of our tests now with `bundle exec rspec`, and see if they're all working:

```
17 examples, 0 failures
```

Yes, good! They are all indeed working. Let's commit this:

```
$ git add .
$ git commit -m "Add sign in and sign out"
$ git push
```

With this section we've implemented sign in and sign out for our application to complement the sign up feature that we added just before. You now have a taste of what Devise can do for you. Go and play around with in the application now. Trying signing up, signing in and signing out.

You may notice during these experiments that the sign up and sign in forms aren't styled as neatly as our project and ticket forms.

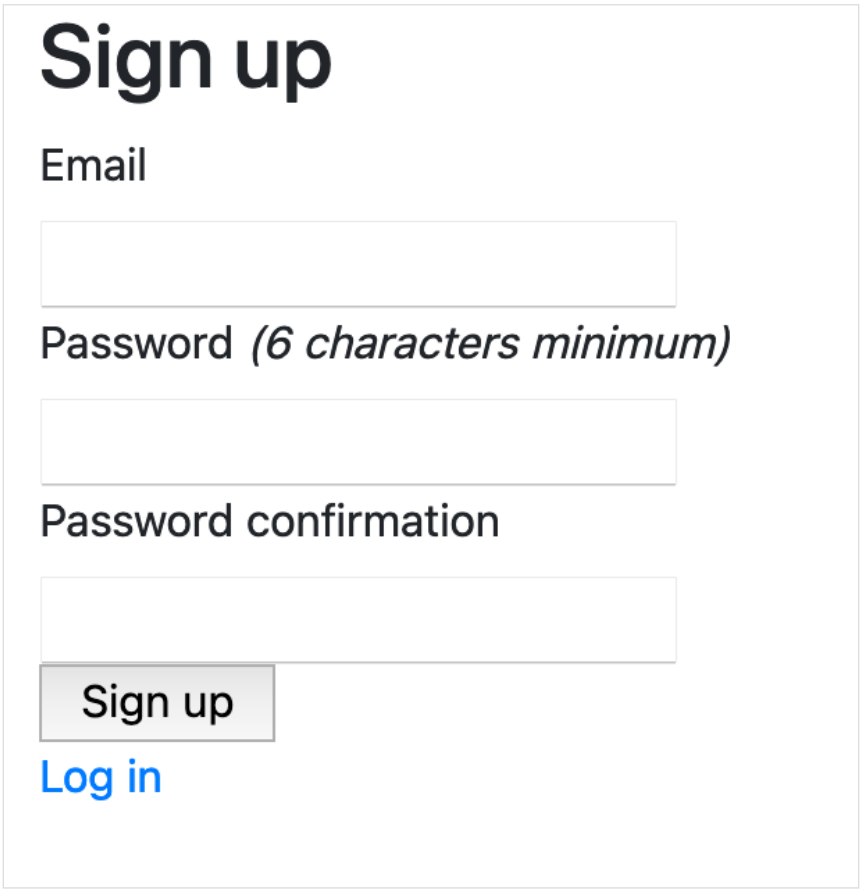


Figure 35. The unstyled sign up form

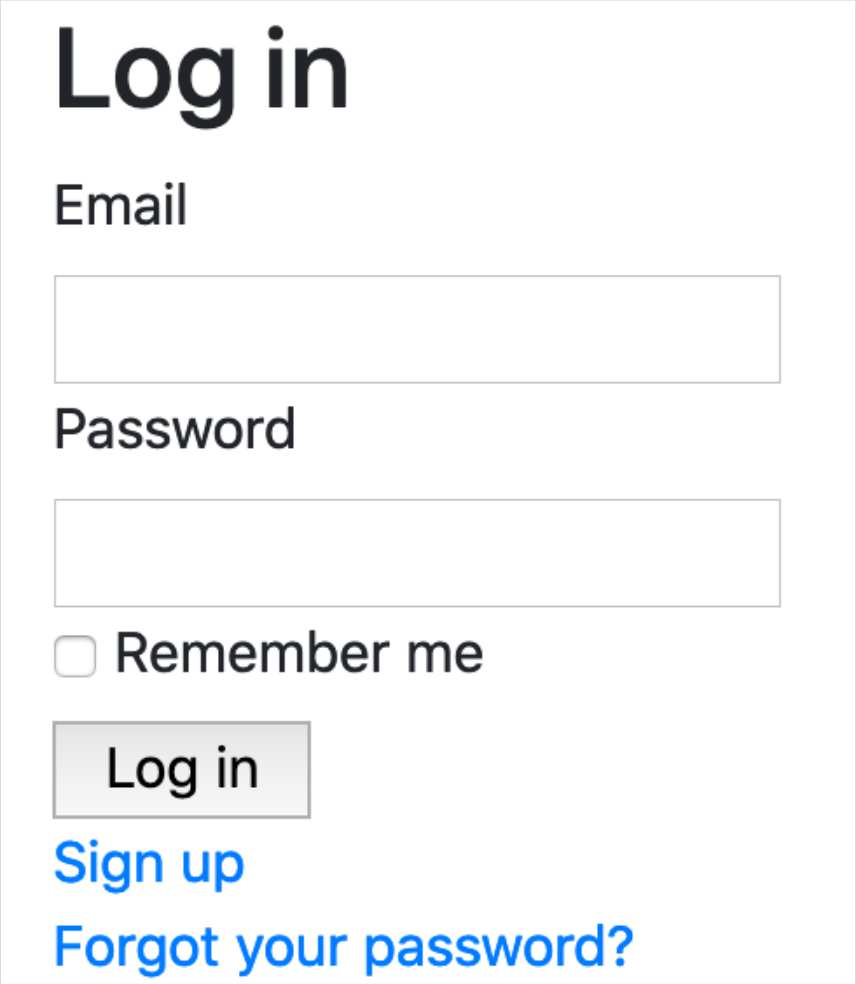


Figure 36. The unstyled sign in form

This is because Devise provides basic views that don't use Bootstrap styling. Our next task will be to fix up these views.

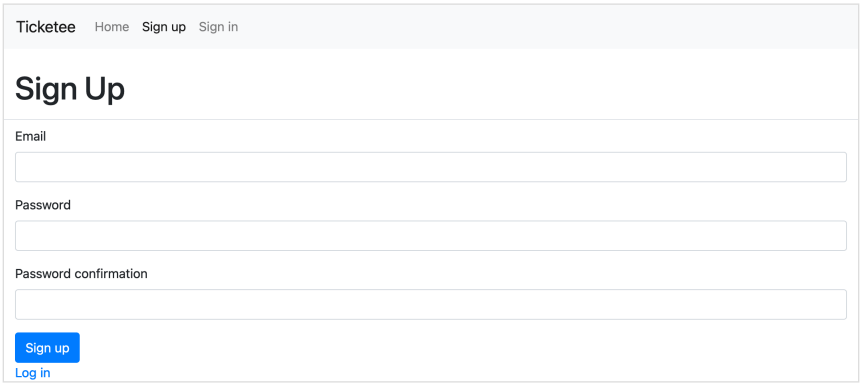
7.3.2. Styling the Devise views

Devise is an engine, and this means that its controllers and views live inside the gem, rather than inside our application. In order to customize these views, we will need to copy them into our application. Thankfully Devise provides a method for us to do this by way of its `devise:views` generator. Let's run this in our application now:

```
$ rails g devise:views
```

This command copies over all of Devise's views to our application's `app/views` directory,

inside another directory called `app/views/devise`. If you navigate to the sign up page now, you'll be pleasantly surprised to see that this page has already been styled using Bootstrap:



There's only a couple of small changes we'll make:

- Adding the `header` wrapper tag to make the heading consistent,
- Changing the form to use the Bootstrap form helpers,
- Removing the block error messages, and
- Making the "Sign up" button blue which matches the styling for the rest of our application.

The view for this page is located at `app/views/devise/registrations/new.html.erb` so open it up and have a look. We can replace the old heading in this view:

Listing 140. The old Devise-generated heading

```
<h2>Sign up</h2>
```

With a new one that matches the style of our application:

Listing 141. The new Bootstrap-styled heading

```
<header>
  <h1>Sign Up</h1>
</header>
```

We'll change the `form_for`:

```
<%= form_for(resource, ...
```

To a `bootstrap_form_for`:

```
<%= bootstrap_form_for(resource, as: resource_name, url: registration_path(resource_name)) do
|f| %>
  <%= f.email_field :email, autofocus: true, autocomplete: "email" %>
  <%= f.password_field :password %>
  <%= f.password_field :password_confirmation %>

  <%= f.primary "Sign up" %>
<% end %>
```

These actions will turn the page into this:

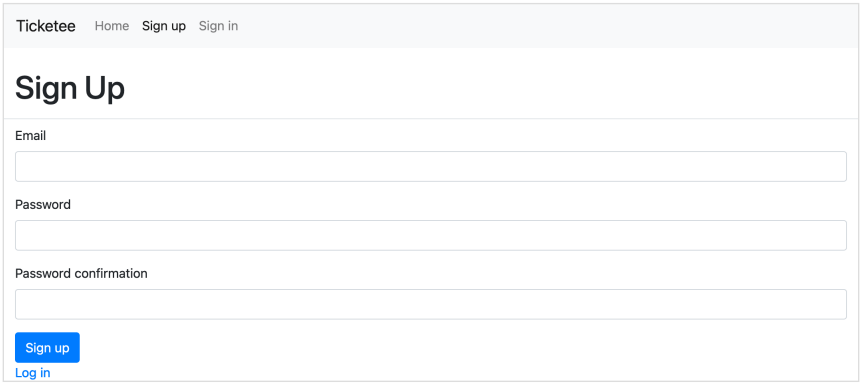


Figure 37. The styled sign up form

We'll also need to make this same changes to the view for the sign in page, which is in `app/views/devise/sessions/new.html.erb`. While you're here it would be nice to change the words "Log in" to "Sign in" as well, to keep it consistent with the link we put in the top navigation.

Listing 142. The new heading on the sign in page

```
<header>
  <h1>Sign In</h1>
</header>
```

And we'll change the form itself:

Listing 143. The new sign in form

```
<%= bootstrap_form_for(resource, as: resource_name, url: session_path(resource_name)) do |f|
%>
  <%= f.email_field :email, autofocus: true, autocomplete: "email" %>
  <%= f.password_field :password, autocomplete: "current-password" %>

  <% if devise_mapping.rememberable? %>
    <%= f.check_box :remember_me %>
  <% end %>

  <%= f.primary "Sign in" %>
<% end %>
```

And that's it. We'll run our tests with `bundle exec rspec` to make sure we haven't broken anything...

```
Failures:

  1) Users can sign in with valid credentials
     Failure/Error: click_button "Log in"
     Capybara::ElementNotFound:
       Unable to find button "Log in"

...
17 examples, 1 failure
```

We've broken something! Our sign in test is looking for a button named "Log in", and we just renamed it to be "Sign in". It's a quick fix, we just need to update the test to use the right value for the button. Inside `spec/features/signing_in_spec.rb`, update the step that clicks the sign in button, to update the text it looks for:

Listing 144. Part of spec/features/signing_in_spec.rb

```
...
click_button "Sign in"
...
```

We've renamed the button because the action users are taking is "signing in", as they click the "Sign in" link in the navbar to do so. Therefore we want the button's name to match.

If we re-run our specs with `bundle exec rspec` after making this change, you'll see that everything is now passing again.

```
17 examples, 0 failures
```

Now the design for our sign up and sign in forms is more consistent with the rest of our application.

Our sign in page will now look like this:

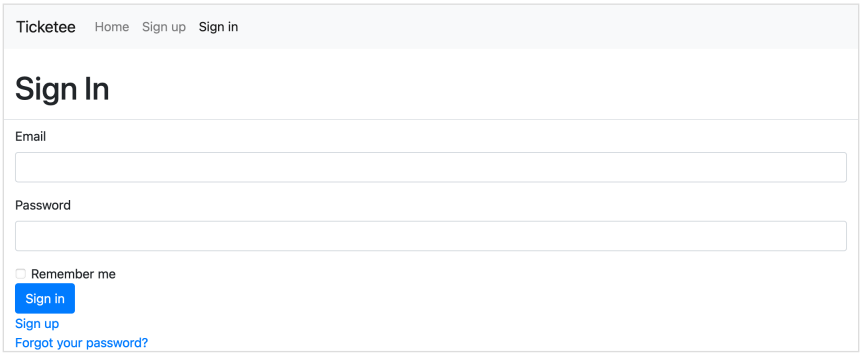


Figure 38. The styled sign in form

Let's make a commit for this change:

```
$ git add .
$ git commit -m "Styled sign up and sign in forms"
$ git push
```

You can go through and style the other views that Devise provides similarly if you feel like. This book doesn't go through that whole process in order to keep this section short.

Now that we have users in our application, let's put them to use.

7.4. Linking tickets to users

Currently when a user creates a ticket in the application, there is no way to tell after the fact which user created that ticket. We are going to fix up this little problem with our application as the last part of this chapter.

When we're done, a ticket will clearly indicate who created it:

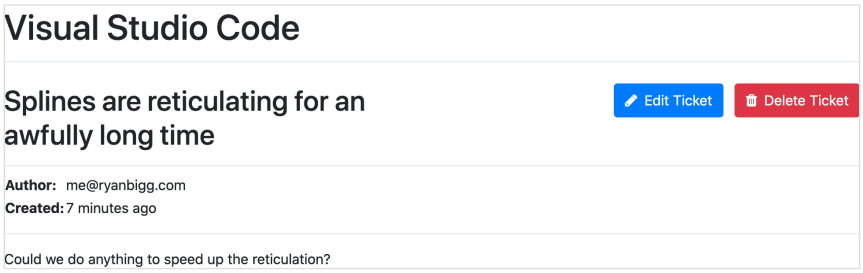


Figure 39. Ticket authorship in full view

Rather than creating a new feature, we'll be adding to a previous feature: `spec/features/creating_tickets_spec.rb`. In the very first scenario for that feature, we'll add few lines to assert that we can see that the current user is the author of the ticket:

Listing 145. Verifying that we show which user created a ticket

```
scenario "with valid attributes" do
  fill_in "Name", with: "Non-standards compliance"
  fill_in "Description", with: "My pages are ugly!"
  click_button "Create Ticket"

  expect(page).to have_content "Ticket has been created."
  within(".ticket") do
    expect(page).to have_content "Author: #{user.email}"
  end
end
```

The `user` variable that we use in this new code doesn't exist yet. Let's add a `let` at the top of this feature. We'll also need to sign in as this user using the `login_as` helper from Warden, which we can do by adding this code to the top of the `before` block in this feature:

Listing 146. Defining the user that will author the new ticket, and signing them in

```
RSpec.feature "Users can create new tickets" do
  let(:user) { FactoryBot.create(:user) }

  before do
    login_as(user)
    ...
```

Now our new code is set up and ready to go. Let's give it a whirl by running `bundle exec rspec spec/features/creating_tickets_spec.rb`. The first thing we'll notice is that the content we expect to be present on the page, is not:

```
1) Users can create new tickets with valid attributes
   Failure/Error: expect(page).to have_content "Author: #{user.email}"
     expected to find text "Author: test1@example.com" in "Internet
     Explorer Non-standards compliance Edit Ticket Delete Ticket My..."
```

To fix this error, let's add this element to the code within `app/views/tickets/show.html.erb`, underneath the `header` with the ticket title and the ticket's edit and delete actions.

Listing 147. Displaying attributes in app/views/tickets/show.html.erb

```
<table class='attributes'>
  <tr>
    <th>Author:</th>
    <td><%= @ticket.author.email %></td>
  </tr>
  <tr>
    <th>Created:</th>
    <td><%= time_ago_in_words(@ticket.created_at) %> ago</td>
  </tr>
</table>

<hr>
```

We can sense that our `Ticket` object will have many attributes added to it over the course of the book, so we're leaving room for expansion by creating a table that we can add extra rows to.

With this code we're calling an `author` method on our `@ticket` object. The `author` method will return the user who created the ticket, and `email` will show us the email address for that user.

(We're also using a view helper called `time_ago_in_words` [56], that will present the timestamp that the ticket was created at, in a nice readable format, such as "about 3 minutes" or "about 2 hours". Just a little nicety.)

The `author` association isn't defined on our `Ticket` model yet, but we can add that with a single line of code underneath the `belongs_to :project` line in `app/models/ticket.rb`:

```
class Ticket < ActiveRecord::Base
  belongs_to :project
  belongs_to :author, class_name: "User"
  ...
```

Here we're defining a new association called `author` on our `Ticket` instances. By default, the association name of `:author` assumes that our class is named `Author` - since we don't have a model called `Author` and the model is instead called `User`, we need to tell the association that. We do that with the `class_name` option.

With a new `belongs_to` association comes the necessity to add a new field to our `tickets` table to track the id of the authors of our tickets. Let's do that now by running this command:

```
$ rails g migration add_author_to_tickets author:references
```

This migration will add the `author_id` to our `tickets` table by using this code:

Listing 148. db/migrate/[timestamp]_add_author_to_tickets.rb

```
class AddAuthorToTickets < ActiveRecord::Migration[6.1]
  def change
    add_reference :tickets, :author, null: false, foreign_key: true
  end
end
```

There's just one small change we have to make to this code before we can run it. Rails assumes that we want to add a foreign key constraint to our association, which we do - but we don't want it how it will behave out of the box.

We need to remove the `foreign_key: true` part of the `add_reference` line, and add a foreign key constraint separately, so our migration should look like this.

We also need to remove the `null: false` part of the `add_reference` line as this will mean our existing items need to have an author_id set and they do not. We would need to create a data migration to add this data to our previously created tickets and assign them to a user.

Listing 149. After fixing the foreign key that Rails generated

```
class AddAuthorToTickets < ActiveRecord::Migration[6.1]
  def change
    add_reference :tickets, :author, null: false, foreign_key: { to_table: :users }
  end
end
```

Why do we need to do this? Because Rails' automatic inference will try to apply a foreign key

on our `tickets` table, pointing to an `authors` table - and we don't have an `authors` table. Our author will be a `User`, living in the `users` table, so we need to specifically tell Rails that the foreign key should point to the `users` table instead (but still use the `author_id` field to do so.)

If we left the line about foreign keys as-is, without changing it, it would still work in this scenario, as long as we're using SQLite. Rails doesn't support foreign keys natively in SQLite, only in PostgreSQL and MySQL, so this would work just fine, it just wouldn't do anything.

We'd run into big problems down the track when it comes to using alternative database systems, like we do in chapter 13 when we look at using PostgreSQL on Heroku. So it's best to fix it up now, to prevent problems later on.

We can now run the migration, to add the `author_id` field to our `tickets` table.

```
$ rails db:migrate
```

Let's see if our feature has gotten any further by running it again, with `bundle exec rspec spec/features/creating_tickets_spec.rb`.

```
1) Users can create new tickets with valid attributes
     Failure/Error: expect(page).to have_content "Ticket has been created."
       expected to find text "Ticket has been created." in "Signed in as
test1@example.com\nSign out\nTicket has not been created.\nNew Ticket Internet Explorer\n1
error prohibited this project from being saved:\nAuthor must exist\nName\nDescription"
     # ./spec/features/creating_tickets_spec.rb:19:in...
```

This is error happening because because we're not actually linking users to the tickets that they create yet. In order to remedy this, we'll need to do that actual linking, and the best place for that linking is in the controller action where tickets are created: the `create` action of `TicketsController`.

After we build a ticket in this action, let's also set the ticket's author. We can do this adding the author assignment to the `create` action for `TicketsController`:

Listing 150. Setting the current user as the ticket's author

```
def create
  @ticket = @project.tickets.build(ticket_params)
  @ticket.author = current_user

  if @ticket.save
    ...
```

By associating an author to the `@ticket` object here directly before the save, we're linking the `current_user` to the ticket. Once the ticket has been saved, that `Ticket` instance and that `User` instance will be tied together forever in a ticket-author relationship.

A naive, but unfortunately common, way to associate tickets to users would be to create a hidden field for `author_id` in the "New Ticket" form, and populate it with the current user's ID. This is a big security hole - a malicious user can simply edit the HTML and change the user ID to be something else, thus creating tickets on another user's behalf.

Manually setting the author in the controller is much safer - there's no way for the user to fake this data if they're logged in. They can only ever assign the tickets to themselves.

Let's see if our test gets any further by running it again with `bundle exec rspec spec/features/creating_tickets_spec.rb`. It should now pass:

```
3 examples, 0 failures
```

Great! We are now showing a ticket's author on the ticket page itself.

Let's run all our tests with `bundle exec rspec` and confirm that we haven't broken anything.

```
17 examples, 4 failures

Failed examples:

rspec ./spec/features/deleting_tickets_spec.rb:11
rspec ./spec/features/editing_tickets_spec.rb:12
rspec ./spec/features/editing_tickets_spec.rb:24
rspec ./spec/features/viewing_tickets_spec.rb:17
```

Oops, it appears we've broken some of our features! Fortunately they all fail for the same reason:

```
1) Users can delete tickets successfully
     Failure/Error: let(:ticket)  { FactoryBot.create(:ticket, project: project) }

     ActiveRecord::RecordInvalid:
       Validation failed: Author must exist
     # ./spec/features/deleting_tickets_spec.rb:5...
```

They're all failing because the tickets created by FactoryBot in the features don't link to an author. When the ticket is created, rails raises a validation error that it can't find the author and so it raises this error.

Let's see about fixing them up, one at a time.

7.4.1. Fixing the failing features

These features are all failing for the same reason: when we create a ticket using our `ticket` factory, an `author` is not set. We can fix this by adding an `association` method call to our factory in `spec/factories/tickets.rb`:

Listing 151. spec/factories/tickets.rb

```
FactoryBot.define do
  factory :ticket do
    name { "Example ticket" }
    description { "An example ticket, nothing more" }
    association :author, factory: :user
  end
end
```

This line will automatically use the `user` factory to create an author for the ticket. If we wanted to override the setting of our factory for the `author` attribute, we could set it manually, the same as any other attribute:

```
a_different_user = FactoryBot.create(:user, email: "different@example.com")
FactoryBot.create(:ticket, author: a_different_user)
```

This change will be enough to get our tests passing, which we'll see by running `bundle exec`

`rspec`:

```
17 examples, 0 failures
```

Excellent. Let's go ahead and make a commit for all this now:

```
$ git add .
$ git commit -m "Link tickets and users upon ticket creation"
$ git push
```

That wraps up the last section of this chapter.

7.5. Summary

This chapter covered how to set up authentication so that users can sign up and sign in to your application to accomplish certain tasks. You learnt about a very popular gem used to handle authentication named Devise, and you also verified the functionality it provides by writing Capybara features to go with it.

Then came linking tickets to users, so you can track which user created which ticket. You did this by using the setter method provided by the `belongs_to` method's presence on the `Ticket` class.

We encourage you to start up the application with `rails server`, visit http://localhost:3000, and play around, to get an idea of how it's looking right now. The application is taking shape and currently offers a lot of functionality for the not-much work we've put in so far.

In the next chapter, we'll look at restricting certain actions to only users who are signed in or who have a special attribute set on them.

[52] https://github.com/heartcombo/devise
[53] https://github.com/omniauth/omniauth
[54] For more information about engines, read the official Engines Guide: http://guides.rubyonrails.org/engines.html
[55] You may scoff, but we've seen this happen. In front of paying clients. They were not amused.
[56] http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html#method-i-time_ago_in_words

Chapter 8. Basic access control

As your application now stands, anybody, whether they're signed in or not, can create new projects. You'll restrict access to certain actions in the `ProjectsController`, allowing only a certain subset of users—users with one particular attribute set in one particular way—to access the actions.

You'll track which users are administrators by putting a boolean field called `admin` in the `users` table. This is the most basic form of user authorization, which isn't to be confused with authentication, which you implemented in chapter 6. Authentication is the process users go through to confirm their identity, whereas authorization is the process used by the system to determine which users should have access to certain things.

(Or more simply, authentication is "who are you?" and authorization is "now that I know who you are, what are you allowed to do?")

You'll see how you can organize code into namespaces so that you can easily restrict access to all subcontrollers to only admin users. If you didn't do this, you would need to restrict access on a per-controller basis, which is prone to errors - it's easy to miss one, and accidentally leave a part of your app wide open for the world to use and abuse.

Some people may suggest using gems such as `rails_admin` or `activeadmin` for this type of feature. While these gems can provide you with an easier way of creating admin interfaces, they do obfuscate the underlying code that is required to get this type of feature to work, and can often be very hard to customize. It's for these reasons we recommend staying away from these gems, and learning to build your own admin interface.

8.1. Turning users into admins

To start the process of restricting the creation of projects to admins, you'll add an `admin` attribute to `User` objects. Only users who have this `admin` attribute set to `true` will be able to create projects. We'll start enforcing this first via our existing tests - we have a feature spec for creating projects, we'll make sure the user creating the projects is an admin.

Alter the existing `before` in `spec/features/creating_projects_spec.rb`, and insert a line to log in as an admin user at the beginning of the `before` block:

Listing 152. Logging in as an admin before creating a project

```
before do
  login_as(FactoryBot.create(:user, :admin))
  ...
```

This line uses the `user` factory we defined in Chapter 7, and adds what's known as a FactoryBot trait [57]. Traits can describe a certain type of a model, or group together similar and related attributes under a meaningful name. In our example, any user with this admin trait will be able to perform special actions within our application. To add this trait to our user factory, all we need to do is call the `trait` method inside the factory, like in this example:

Listing 153. Defining an `admin` trait

```
FactoryBot.define do
  factory :user do
    sequence(:email) { |n| "test#{n}@example.com" }
    password { "password" }

    trait :admin do
      admin { true }
    end
  end
end
```

A trait will take the original factory's attributes and modify them. This new addition to our code means that any user created with `FactoryBot.create(:user, :admin)` will have their `admin` attribute set to `true`.

When you run `bundle exec rspec spec/features/creating_projects_spec.rb` you'll see that no `admin=` method is defined for a `User` object:

```
1) Users can create new projects with valid attributes
   Failure/Error: login_as(FactoryBot.create(:user, :admin))
   NoMethodError:
     undefined method `admin=' for #<User:...>
```

Therefore the next logical step is to define a field in the database so the attribute-setter method is available.

8.1.1. Adding the admin field to the users table

You can generate a migration to add the `admin` field by running this command:

```
$ rails g migration add_admin_to_users admin:boolean
```

Rails does a pretty good job of inferring what you want the migration to do, just from the name you specified. From `add_admin_to_users` it presumes you want to add a field named `admin` to the table called `users`, which is exactly what you want. The extra `admin:boolean` tells Rails that the `admin` field should be a boolean field.

However, you'll want to modify this migration so that when users are created, the `admin` field is set to `false` rather than defaulting to `nil`. Even though `nil` is "falsey"[58] in Ruby, it's clearer to make it explicitly `false`. `nil` means users have no admin information, but they do: they're not an admin. Better to be explicit about things and to use `false` here.

Changing the default value makes even more sense if you consider it from an database SQL perspective. We traditionally treat `nil` in Ruby (meaning "no value") as equivalent to `null` in SQL, but `null` in SQL means "an unknown value". In this case, `null` in the database isn't an appropriate default value because we will always know whether or not the user is an admin.

To change the default, open the freshly-generated migration (which will be in `db/migrate/<timestamp>_add_admin_to_users.rb`) and change this line:

```
add_column :users, :admin, :boolean
```

To this:

Listing 154. Now with added `default` option

```
add_column :users, :admin, :boolean, default: false
```

When you pass in the `:default` option, the `admin` field defaults to `false`, ensuring that users aren't accidentally created as admins.

You can roll back changes if you wish

If you jumped the gun and ran `rails db:migrate` before modifying the migration, this field will default to `null`, which is no good. It may seem like you're screwed at this point, but you're not. Run `rails db:rollback` to undo this latest migration, so that you can modify it and get back on track. Once the modification is done correctly, don't forget to run `bundle exec rake db:migrate` again!

Run `rails db:migrate` so that the migration adds the `admin` field to the `users` table in both the development and test databases. When you run `bundle exec rspec spec/features/creating_projects_spec.rb`, it will run fully:

```
2 examples, 0 failures
```

Great! With the new `admin` trait on the `user` factory defined, you can use it to test restricting the acts of creating, updating, and destroying projects to only those users who are admins.

8.1.2. Creating the first admin user

Now that we have the ability to distinguish normal users from admin users, it would be great if we actually had an admin user to use our development app with. We've got one in our tests, but not one in the development environment.

Data like this—that you really need to have created in your database before the application can be used—is called seed data. Rails has a defined place to put your seed data, in `db/seeds.rb`.

Open up `db/seeds.rb` - it's empty at the moment, but contains instructions for how it can be used. You can put commands to create the relevant data in the file, and then run it with `rails db:seed`.

We'll put the code needed to create a new admin user at the bottom of this file:

Listing 155. Seeding an admin user

```
unless User.exists?(email: "admin@ticketee.com")
  User.create!(email: "admin@ticketee.com", password: "password", admin: true)
end
```

While we're here, we'll also add some other sample data, to play with when we use the application.

Listing 156. Seeding a non-admin user, and some sample projects

```
unless User.exists?(email: "viewer@ticketee.com")
  User.create!(email: "viewer@ticketee.com", password: "password")
end

["Visual Studio Code", "Internet Explorer"].each do |name|
  unless Project.exists?(name: name)
    Project.create!(name: name, description: "A sample project about #{name}")
  end
end
```

And then you can run this command to load the seeds into your application:

```
$ rails db:seed
```

Now you have an admin user that can sign in to Ticketee, and can see and test all of the admin functionality we will be building from here on out.

Before you do that, let's commit everything:

```
$ git add .
$ git commit -m "Added admin flag to User model, and seeded the first
  admin user"
$ git push
```

8.2. Controller namespacing

It's all well and good having an `admin` flag on a user record, but at the moment it doesn't actually do anything. We'll fix that now.

We already have some functionality that we only want admins to be able to access—the ability to create and delete projects—and later on we'll be building more. An easy way to restrict access to all of this functionality at once is to move it into its own controller namespace.

You've seen an example of namespaces already for controllers, though you might not realize

it - your `ApplicationController` (in `app/controllers/application_controller.rb`) that was created when you generated the initial Rails app, inherits from `ActionController::Base`. In this case, `ActionController` is the namespace, and `Base` is name of the class inside the namespace.

We can define our own namespace, that we'll imaginatively call `Admin`. Inside this namespace, we can define all of the controllers we like, that will all inherit from a base controller also inside the namespace. And inside that base controller, we can implement a `before_action` that will run before every action and check if the user is an admin - if they're not, we can simply turn them away.

8.2.1. Generating a namespaced controller

The first thing you'll need is the base controller in the new namespace, that all of the other admin controllers will inherit from. We can generate it by running this command:

```
$ rails g controller admin/application index
```

When you use the `/` separator between parts of the controller, Rails knows that it means you want a namespace. In this case, it will generate a namespaced controller called `Admin::ApplicationController` at `app/controllers/admin/application_controller.rb`. The views for this controller are at `app/views/admin/application`, and the spec is at `spec/controllers/admin/application_controller_spec.rb`. By passing in the word `index` at the end, this controller will contain an `index` action, and there will also be a view at `app/views/admin/application/index.html.erb` for this action, as well as a route defined in `config/routes.rb`, like this:

```
namespace :admin do
  get 'application/index'
end
```

Why have we picked the name `Admin::ApplicationController` over something that might be more explicit, like `Admin::BaseController`? The main controller of our app, in the root namespace, is called `ApplicationController`, so it makes sense that any other main controllers in namespaces should also be called `ApplicationController`.

What URLs and named routes does this generate? You can run `bundle exec rake routes` and see (controller column omitted for brevity):

```
                Prefix  Verb  URI Pattern
admin_application_index  GET   /admin/application/index(.:format)
```

The `namespace` block in our routing has directly translated to a folder name in the generated URL structure, which is nice. The `application/index` part is ugly though - it would make more sense for our action to be the root route of the namespace, like we defined `projects#index` to be the root route of the root namespace.

Replace the entire `namespace` block in `config/routes.rb` with the following:

Listing 157. Defining a root route for the `admin` namespace

```
namespace :admin do
  root "application#index"
end
```

And now `bundle exec rake routes` gives you some nicer URLs:

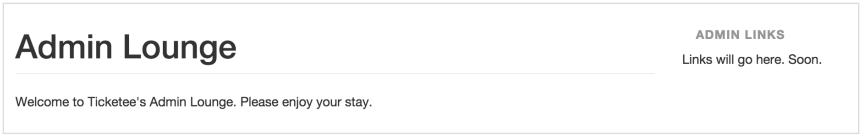
```
    Prefix  Verb   URI Pattern         Controller#Action
admin_root  GET    /admin(.:format)    admin/application#index
```

This root admin page is the first page our admins will see in the admin area, so we may as well make it a bit pretty. Open up the generated view for it, `app/views/admin/application/index.html.erb`, and replace its contents with the following.

Listing 158. Default content for the admin homepage

```
<div class="row">
  <div class="col-md-9">
    <header>
      <h1>Admin Lounge</h1>
    </header>

    <p>Welcome to Ticketee's Admin Lounge. Please enjoy your stay.</p>
  </div>

  <div class="col-md-3">
    <h5 class="text-uppercase mt-4">Admin Links</h5>
    <ul class="nav flex-column">
      <li>Links will go here. Soon.</li>
    </ul>
  </div>
</div>
```

Once we've done that, we can refresh the admin homepage and see a page that looks like this:



But anyone can access this page at the moment, you don't even need to be logged in. The next step is to make sure this page is only accessible to admin users, and we'll do that with a `before_action` that checks not only whether a user is signed in, but also whether the user is an admin. We'll start (as always) with a test.

8.2.2. Testing a namespaced controller

Testing this functionality is much better suited to a request spec, rather than the feature specs we've been using heavily up to this point. Feature specs are great for defining and testing a series of actions that a user can perform in your application, but request specs are much better for quickly testing singular points, such as whether a user can go to a specific action in a controller. You used this same reasoning back in chapter 4 to test what happens when a user attempts to go to a project that doesn't exist.

Let's take the file at `spec/requests/admin/application_request_spec.rb` and rename it to `spec/requests/admin/root_spec.rb`.

Then, let's open `spec/requests/admin/root_spec.rb`, and write the following example to ensure that non-signed-in and non-admin users can't access the `index` action. You can replace the current contents of the file.

Listing 159. spec/requests/admin/root_spec.rb

```
require 'rails_helper'

RSpec.describe "/admin", type: :request do
  describe "GET /" do
    let(:user) { FactoryBot.create(:user) }
    let(:admin) { FactoryBot.create(:user, :admin) }

    context "when signed in as a user" do
      before do
        login_as(user)
      end

      it "redirects away" do
        get "/admin"
        expect(response).to have_http_status(302)
      end
    end

    context "when signed in as an admin" do
      before do
        login_as(admin)
      end

      it "lets the admin in" do
        get "/admin"
        expect(response).to have_http_status(:success)
      end
    end
  end
end
```

With this test, you're testing that when a non-admin user makes a `GET` request to the `index` action of the `Admin::ApplicationController`, the response redirects them to the root path of the application and also sets a `flash[:alert]` message to "You must be an admin to do that."

When you run this test using `bundle exec rspec ticketee/spec/requests/admin/root_spec.rb`, it fails like this:

```
     Failure/Error: login_as(user)

     NoMethodError:
       undefined method `login_as' for
#<RSpec::ExampleGroups::Admin::GET::WhenSignedInAsAUser:...>
```

In order to fix this we will need to add this configuration to our `rails_helper.rb` file, underneath: `config.include Warden::Test::Helpers, type: :feature`.

```
config.include Warden::Test::Helpers, type: :request
```

This line adds the `login_as` helpers to our request tests. We've now got these helpers in both the feature tests and the request tests.

With this in place, when you run the tests again, it will fail like this:

```
1) Admin::ApplicationController non-admin users are not able to access the index action
    Failure/Error: expect(response).to redirect_to "/"

      Expected response to be a <3XX: redirect>, but was a <200: OK>
```

This error message tells you that although you expected to be redirected, the response was actually `200`, indicating a successful response. This isn't what you want; you want a redirect with a `302` response code! Now let's get it to pass.

The first step is to define a new method to be used as the `before_action` admin check on the `Admin::ApplicationController`. This method checks whether a user is an admin, and, if not, sets the "You must be an admin to do that" message and redirects the user back to the root path.

Define this new method in `app/controllers/admin/application_controller.rb` by placing the code from the following listing at the end of the class.

Listing 160. app/controllers/admin/application_controller.rb

```
class Admin::ApplicationController < ApplicationController
  ...

  private

  def authorize_admin!
    authenticate_user!

    unless current_user.admin?
      redirect_to root_path, alert: "You must be an admin to do that."
    end
  end
end
```

This method uses the `authenticate_user!` method provided by Devise to ensure that the user is signed in. If the user isn't signed in, they will be redirected to the sign in page. If we didn't use this method here, we would get an error when we call `admin?` on `current_user`, as the `current_user` method would return `nil`.

To call the `authorize_admin!` method, call `before_action` at the top of your `Admin::ApplicationController`.

```
class Admin::ApplicationController < ApplicationController
  before_action :authorize_admin!
  ...
```

Let's run the test again and see what happens:

```
1 example, 0 failures
```

Excellent! You have a controller that is only accessible by admin users of your application. With that done, you should ensure that everything is working as expected by running `bundle exec rspec`:

```
22 examples, 0 failures, 2 pending
```

Everything is still passing, but there are two pending tests:

```
# ./spec/helpers/admin/application_helper_spec.rb:14
# ./spec/views/admin/application/index.html.erb_spec.rb:4
```

These two tests were added when you ran `rails g controller admin/controller`. The first is a simple helper test, and the second is a view spec, which can be used to ensure that rendering a particular view works as intended.[59] You don't need these two particular tests, so delete both files. When you re-run `bundle exec rspec`, you should see this output:

```
20 examples, 0 failures
```

Let's commit this now:

```
$ git add .
$ git commit -m "Add admin namespace with application controller"
$ git push
```

8.2.3. Moving functionality into the admin namespace

Now that we have a working admin namespace, that only admin users can access, we can start moving functionality into it. Normal users shouldn't have the ability to create, update or delete projects, only admins should; so that functionality sounds like a good candidate to be moved.

We have tests that cover that functionality, so if we accidentally break it in the process of moving it, we'll know it straight away. To start with, the following tests should be moved into `spec/features/admin`:

- `spec/features/creating_projects_spec.rb`
- `spec/features/updating_projects_spec.rb`
- `spec/features/deleting_projects_spec.rb`

Re-run them, and they will still pass.

```
$ bundle exec rspec spec/features/admin
...

3 examples, 0 failures
```

You can make another `ProjectsController` inside the `Admin` namespace, which is where the moved controller actions will go. We can do that with this command:

```
$ rails g controller admin/projects
```

The only change that we'll need to make to this generated controller is to change what it inherits from - a default controller inherits from `ApplicationController`, but we want our admin controllers to inherit from `Admin::ApplicationController`. So inside the new controller, `app/controllers/admin/projects_controller.rb`, change the first line from:

```
class Admin::ProjectsController < ApplicationController
```

To:

```
class Admin::ProjectsController < Admin::ApplicationController
```

The actions that we'll be moving to this new controller are the `new`, `create` and `edit`, `update` and `destroy` actions of the existing `ProjectsController`. You can cut and paste those actions from the old controller to the new, so that your `ProjectsController` looks like this:

Listing 161. The new `ProjectsController`

```
class ProjectsController < ApplicationController
  before_action :set_project, only: [:show, :edit, :update, :destroy]

  def index
    @projects = Project.all
  end

  def show
  end

  private

  def set_project
    @project = Project.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    flash[:alert] = "The project you were looking for could not be found."
    redirect_to projects_path
  end

  def project_params
    params.require(:project).permit(:name, :description)
  end
end
```

And the `Admin::ProjectsController` looks like this:

Listing 162. The new `Admin::ProjectsController`

```
class Admin::ProjectsController < Admin::ApplicationController
  def new
    @project = Project.new
  end

  def create
    @project = Project.new(project_params)

    if @project.save
      flash[:notice] = "Project has been created."
      redirect_to @project
    else
      flash.now[:alert] = "Project has not been created."
      render "new"
    end
  end

  def edit
    @project = Project.find(params[:id])
  end

  def update
    @project = Project.find(params[:id])

    if @project.update(project_params)
      flash[:notice] = "Project has been updated."
      redirect_to @project
    else
      flash.now[:alert] = "Project has not been updated."
      render "edit"
    end
  end

  def destroy
    @project.destroy

    flash[:notice] = "Project has been deleted."
    redirect_to projects_path
  end
end
```

The `project_params` method will have to be duplicated into the `Admin::ProjectsController`, as it is used in both the `create` action of the `Admin::ProjectsController` as well as in the `update` action of the `ProjectsController`.

We also need to load the `@project` variable in the `edit`, `update` and destroy` actions of the `Admin::ProjectsController`, so let's copy over that `find_project` line from the (non-admin) `ProjectsController` class:

Listing 163. The final `Admin::ProjectsController`

```
class Admin::ProjectsController < Admin::ApplicationController
  before_action :set_project, only: [:show, :edit, :update, :destroy]


  def new
    @project = Project.new
  end

  def create
    @project = Project.new(project_params)

    if @project.save
      flash[:notice] = "Project has been created."
      redirect_to @project
    else
      flash.now[:alert] = "Project has not been created."
      render "new"
    end
  end

  def edit
  end

  def update
    if @project.update(project_params)
      flash[:notice] = "Project has been updated."
      redirect_to @project
    else
      flash.now[:alert] = "Project has not been updated."
      render "edit"
    end
  end

  def destroy
    @project.destroy

    flash[:notice] = "Project has been deleted."
    redirect_to projects_path
  end

  private
```

```
  def project_params
    params.require(:project).permit(:name, :description)
  end

  def set_project
    @project = Project.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    flash[:alert] = "The project you were looking for could not be found."
    redirect_to projects_path
  end
end
```

What else would we need to do? Oh of course, the views that these actions were rendering! Move the following views into `app/views/admin/projects`:

- `app/views/projects/_form.html.erb`
- `app/views/projects/new.html.erb`
- `app/views/projects/edit.html.erb`

You can re-run the admin specs with `bundle exec rspec spec/features/admin` now, to see what has been broken. We'll see five failures:

```
rspec ./spec/features/admin/creating_projects_spec.rb:14 # Users can create new projects with
valid attributes
rspec ./spec/features/admin/creating_projects_spec.rb:28 # Users can create new projects when
providing invalid attributes
rspec ./spec/features/admin/deleting_projects_spec.rb:4 # Users can delete projects
successfully
rspec ./spec/features/admin/editing_projects_spec.rb:12 # Users can edit existing projects
with valid attributes
rspec ./spec/features/admin/editing_projects_spec.rb:20 # Users can edit existing projects
when providing invalid attributes
```

All of these tests are complaining that their actions cannot be found. This is because we haven't updated the links in our views - the "New Project" and "Delete Project" links are still pointing at routes that reference actions that no longer exist.

You'll need to update your routes to point to the new actions, and you can also remove the routes for the old actions, to keep things tidy. Our old routes for projects looked like this:

Listing 164. Non-admin route definition for projects

```
resources :projects do
  resources :tickets
end
```

This defines the seven default RESTful routes for a `projects` resource: `index`, `new`, `create`, `show`, `edit`, `update` and `destroy`. If we want to remove some of those (because we've removed the actions), we can use the `only` option [60] when defining the routes:

Listing 165. Non-admin route definition for projects

```
resources :projects, only: [:index, :show] do
  resources :tickets
end
```

And we can add the new routes inside the admin namespace:

Listing 166. Admin route definition for projects

```
namespace :admin do
  root "application#index"

  resources :projects, except: [:index, :show]
end
```

This generates the following named routes:

```
            Prefix Verb   URI Pattern                          Controller#Action
    admin_projects POST   /admin/projects(.:format)            admin/projects#create
 new_admin_project GET    /admin/projects/new(.:format)        admin/projects#new
edit_admin_project GET    /admin/projects/:id/edit(.:format) admin/projects#edit
     admin_project PATCH  /admin/projects/:id(.:format)        admin/projects#update
                   PUT    /admin/projects/:id(.:format)        admin/projects#update
                   DELETE /admin/projects/:id(.:format)        admin/projects#destroy
```

So now we can edit the links to "New Project" and "Delete Project" in the `index` and `show` views, to point to the new named routes.

Listing 167. The new link to create a project in `app/views/projects/index.html.erb`

```
<li>
  <%= link_to "New Project", new_admin_project_path, class: "new" %>
</li>
```

Listing 168. The new links to edit + delete a project in `app/views/projects/show.html.erb`

```
<ul class="actions">
    <li><%= link_to "Edit Project", edit_admin_project_path(@project),
      class: "edit" %></li>

    <li><%= link_to "Delete Project", admin_project_path(@project),
      method: :delete,
      data: { confirm: "Are you sure you want to delete this project?" },
      class: "delete" %></li>
  </ul>
```

Even though these links are only accessible to admins in our application, they will still show up for non-admins. This is a problem we'll be fixing a little later in this chapter.

The creating projects feature

Let's try fixing these tests one-by-one now. We'll start with the test for creating projects. When we run it with `bundle exec rspec spec/features/admin/creating_projects_spec.rb`, we'll see a new error:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"

   ActionController::RoutingError:
     No route matches [POST] "/projects"
```

The test is trying to click the button on the form to create a project, but the route that the form is submitting to, no longer exists. This is because we just copied the old form partial - we should edit that form to submit to the new `create` action in the admin namespace instead. Open the admin project form in `app/views/admin/projects/_form.html.erb`, and modify the first line like so:

Listing 169. The start of the updated _form.html.erb partial

```
<%= bootstrap_form_with(model: [:admin, project], local: true, label_errors: true) do |form|
%>
```

For this `bootstrap_form_with`, you use the array form you saw earlier with `[@project, @ticket]`, but this time you pass in a symbol rather than a model object. Rails interprets the symbol literally, generating a route such as `admin_users_path` rather than `users_path`, which would be generated if we used `bootstrap_form_with @user` instead. You can also use this array syntax with `link_to` (seen earlier) and `redirect_to` helpers. Any symbol passed anywhere in the array for any of these methods is interpreted literally. The same goes for strings.

Now our specs for creating a project will pass:

```
2 examples, 0 failures
```

The updating projects feature

How about updating projects? When we run `bundle exec spec/features/admin/editing_projects_spec.rb`, we'll see the following error:

```
Failures:

  1) Users can edit existing projects with valid attributes
     Failure/Error: fill_in "Name", with: "Visual Studio Code Nightly"

     Capybara::ElementNotFound:
       Unable to find field "Name" that is not disabled
     # ./spec/features/admin/editing_projects_spec.rb:13:in `block (2 levels) in <top
(required)>'

  2) Users can edit existing projects when providing invalid attributes
     Failure/Error: fill_in "Name", with: ""

     Capybara::ElementNotFound:
       Unable to find field "Name" that is not disabled
     # ./spec/features/admin/editing_projects_spec.rb:21:in `block (2 levels) in <top
(required)>'
```

Our test isn't logging in as an admin user (or as any user), before attempting to delete a

project - that's why the text on the page includes "You need to sign in or sign up before continuing". We can sign in as an admin the same way we did in the "Creating Projects" spec, in the `before` block:

Listing 170. spec/features/admin/editing_projects_spec.rb

```
RSpec.feature "Admins can edit existing projects" do
  before do
    login_as(FactoryBot.create(:user, :admin))
    FactoryBot.create(:project, name: "Visual Studio Code")

    visit "/"
    click_link "Visual Studio Code"
    click_link "Edit Project"
  end

  ...
```

Re-running this feature will now show it correctly passing:

```
2 examples, 0 failures
```

The deleting projects feature

What about deleting projects? If you run `bundle exec rspec spec/features/admin/deleting_projects_spec.rb`, you'll get the following error:

```
1) Users can delete projects successfully
   Failure/Error: expect(page).to have_content "Project has been deleted."
     expected to find text "Project has been deleted." in "Ticketee\nHome Sign up Sign
in\nYou need to sign in or sign up before continuing.\nSign in\nEmail\nPassword\nRemember
me\nSign up Forgot your password?"
```

This is the same issue we saw earlier, again! We need to be signing in as an admin. To fix this test, we'll change the top of the feature to this:

Listing 171. spec/features/admin/deleting_projects_spec.rb

```
RSpec.feature "Users can delete projects" do
  before do
    login_as(FactoryBot.create(:user, :admin))
  end

  ...
```

And now the "deleting projects" spec will pass as well.

```
1 example, 0 failures
```

As always, run `bundle exec rspec` to make sure that nothing else is broken:

```
21 examples, 0 failures, 1 pending
```

Everything is passing, but we have another pending spec, coming from autogenerated code:

```
# ./spec/helpers/admin/projects_helper_spec.rb:14
```

You can delete this file as we are not using it at all. Re-run the specs to verify that everything is all green:

```
20 examples, 0 failures
```

Great! You've moved this admin-only functionality into the admin namespace, which restricts it so that non-admin users cannot access it. It's a good time to stop and commit your changes.

```
$ git add .
$ git commit -m "Only admins can create, edit or delete projects"
$ git push
```

You've restricted the controller actions by putting them into the namespace, but the links to perform these actions, such as "New Project" and "Delete Project", are still visible to users. You should hide (or protect) these links from users who aren't admins, because it's useless to show actions to people who can't perform them. Let's look at doing that.

8.3. Hiding links

In this section, you'll learn how to hide certain links, such as the "New Project" and "Delete Project" links, from users who have no authorization to perform those actions in your application.

If these links were available for the users to follow, then they would be told "You must be an admin to do that", thanks to the `before_action` that we set up in `Admin::ApplicationController`. This happens because these links link to `Admin::ProjectsController`, which inherits from `Admin::ApplicationController`. Therefore it's pointless to display these links to people who shouldn't be able to click them, so let's look at hiding them.

8.3.1. Hiding the "New Project" link

To begin, open a new file called `spec/features/hidden_links_spec.rb`. In this file, you'll write scenarios to ensure that the right links are shown to the right people. Let's start with the code for checking that the "New Project" link is hidden from regular users who are either signed out or signed in, but is shown to admins.

Listing 172. spec/features/hidden_links_spec.rb

```
require "rails_helper"

RSpec.feature "Users can only see the appropriate links" do
  let(:user) { FactoryBot.create(:user) }
  let(:admin) { FactoryBot.create(:user, :admin) }

  context "anonymous users" do
    scenario "cannot see the New Project link" do
      visit "/"
      expect(page).not_to have_link "New Project"
    end
  end

  context "regular users" do
    before { login_as(user) }

    scenario "cannot see the New Project link" do
      visit "/"
      expect(page).not_to have_link "New Project"
    end
  end

  context "admin users" do
    before { login_as(admin) }

    scenario "can see the New Project link" do
      visit "/"
      expect(page).to have_link "New Project"
    end
  end
end
```

In this spec, you first define two `let` blocks: one for a `user` and one for an `admin`. These create a non-admin user and an admin user, respectively, when they're called. You have three `context` blocks: one for each permutation of the scenario. In the first, you act as an anonymous user and check that there is indeed no "New Project" link on the page. In the second, you act as a regular user and again check that there's no "New Project" link on the page. In the third, however, you sign in as an admin; and when that happens, the "New Project" link should appear on the page.

When you run this feature using `bundle exec rspec spec/features/hidden_links_spec.rb`, you'll get some expected failure messages.

```
1) Users can only see the appropriate links anonymous users cannot see
   the New Project link
   Failure/Error: expect(page).not_to have_link "New Project"
     expected not to find link "New Project", found 1 match: "New
     Project"
   # ./spec/features/hidden_links_spec.rb:10:in `block (3 levels) in...

2) Users can only see the appropriate links regular users cannot see
   the New Project link
   Failure/Error: expect(page).not_to have_link "New Project"
     expected not to find link "New Project", found 1 match: "New
     Project"
   # ./spec/features/hidden_links_spec.rb:19:in `block (3 levels) in...
```

The first two scenarios from our new feature fail, of course, because you've done nothing yet to hide the link they're checking for. Open `app/views/projects/index.html.erb`, and change the "New Project" link to the following in order to work toward hiding it:

Listing 173. Only showing the "New Project" link to admins

```
<% admins_only do %>
  <ul class="actions">
    <li>
      <%= link_to "New Project", new_admin_project_path, class: "new" %>
    </li>
  </ul>
<% end %>
```

The `admins_only` method won't magically be there, so you'll need to define it. The method needs to take a block. If `current_user` is an admin, the method should run the code in the block; and if they're not, it should show nothing.

You'll want this helper to be available everywhere in your application's views, so the best place to define it is in `ApplicationHelper`. If you wanted it to be available only to a specific controller's views, you would place it in the helper that shares the name with the controller. To define the `admins_only` helper, open `app/helpers/application_helper.rb` and define the method in the module using this code:

Listing 174. app/helpers/application_helper.rb

```
def admins_only(&block)
  block.call if current_user.try(:admin?)
end
```

The `admins_only` method takes a block (as promised), which is the code between the `admins_only do` and `end` in your view. To run this code in the block, you call `block.call`, which only runs it if `current_user.try(:admin?)` returns `true`. This `try` method tries a method on an object: if the object is nil (as it would be if there is no user currently logged in), `try` gives up and returns `nil`, rather than raising a `NoMethodError` exception.

When you run this feature using `bundle exec rspec spec/features/hidden_links_spec.rb`, it passes because the links are being hidden and shown as required:

```
3 examples, 0 failures
```

Now that the "New Project" link hides if the user isn't an admin, let's do the same thing for the "Delete Project" link.

8.3.2. Hiding the edit and delete links

You need to add this `admins_only` helper to the "Edit Project" and "Delete Project" links on the project's `show` view as well, to hide this link from people who shouldn't see it. Before you do this, though, you should add further scenarios to cover this change to `spec/features/hidden_links_spec.rb`.

In order to test that the link works, you need to create a project during these tests. To enable that, define a `let` block with the two for users and admins in this file, using this line:

Listing 175. spec/features/hidden_links_spec.rb

```
RSpec.feature "Users can only see the appropriate links" do
  let(:project) { FactoryBot.create(:project) }
  ...
```

Now you can use this `project` method to define scenarios in the "anonymous users" context block, to ensure that anonymous users can't see the "Delete Project" link. Use the code from the following listing.

Listing 176. spec/features/hidden_links_spec.rb

```
context "anonymous users" do
  ...

  scenario "cannot see the Edit Project link" do
    visit project_path(project)
    expect(page).not_to have_link "Edit Project"
  end

  scenario "cannot see the Delete Project link" do
    visit project_path(project)
    expect(page).not_to have_link "Delete Project"
  end
end
```

Next, copy the scenario into the "regular users" `context` block:

Listing 177. spec/features/hidden_links_spec.rb

```
context "regular users" do
 ...

  scenario "cannot see the Edit Project link" do
    visit project_path(project)
    expect(page).not_to have_link "Edit Project"
  end

  scenario "cannot see the Delete Project link" do
    visit project_path(project)
    expect(page).not_to have_link "Delete Project"
  end
end
```

And finally, ensure that admin users can see the link by placing the code from this listing in the "admin users" `context`:

Listing 178. spec/features/hidden_links_spec.rb

```
context "admin users" do
  ...

  scenario "can see the Edit Project link" do
    visit project_path(project)
    expect(page).to have_link "Edit Project"
  end

  scenario "can see the Delete Project link" do
    visit project_path(project)
    expect(page).to have_link "Delete Project"
  end
end
```

With these latest changes, you should now have six new scenarios in the Hidden Links feature that check the visibility of "Edit Project" and "Delete Project" links.

- Two checking the links for anonymous users
- Two checking for regular users, and;
- Two checking for admins

Run this feature now with `bundle exec rspec spec/features/hidden_links_spec.rb` to see the new failures:

```
rspec ./spec/features/hidden_links_spec.rb:14
  # ... anonymous users cannot see the Edit Project link
rspec ./spec/features/hidden_links_spec.rb:19
  # ... anonymous users cannot see the Delete Project link
rspec ./spec/features/hidden_links_spec.rb:33
  # ... regular users cannot see the Edit Project link
rspec ./spec/features/hidden_links_spec.rb:38
  # ... regular users cannot see the Delete Project link
```

Again, we haven't done anything to hide the links, so the four tests here that expect the links not to be there are failing. To make these tests pass, change the links in `app/views/projects/show.html.erb` and wrap them in the `admins_only` helper, as shown in the following listing.

Listing 179. app/views/projects/show.html.erb

```
<% admins_only do %>
  <ul class="actions">
    <li><%= link_to "Edit Project", edit_project_path(@project),
      class: "edit" %></li>

    <li><%= link_to "Delete Project", admin_project_path(@project),
      method: :delete,
      data: { confirm: "Are you sure you want to delete this project?" },
      class: "delete" %></li>
  </ul>
<% end %>
```

We should now be hiding both the 'Edit Project' and 'Delete Project' links. A great way to know if this is the case is to run the test using `bundle exec rspec spec/features/hidden_links_spec.rb`. When you do, you should see this:

```
9 examples, 0 failures
```

All right, that was a little too easy, but that's Rails.

This is a great point to ensure that everything is still working by running all the tests with `bundle exec rspec`. According to the following output, everything's in working order:

```
29 examples, 0 failures
```

Let's commit and push that:

```
$ git add .
$ git commit -m "Only admins can see the links to create and delete
  projects"
$ git push
```

In this section, you defined a namespace and ensured that only users with the `admin` attribute set to `true` were able to access actions inside it. This is a great example of authorization.

Now that we have the namespace set up, we can start building new functionality inside it. We only have one admin user, the one we created in our seed data - it would be nice if our admin user had an interface for creating new users, or making existing users into admins as well.

8.4. Namespace-based CRUD

Now that only admins can access the `admin` namespace, you can create the CRUD actions for `Admin::UsersController` too, as you did for `TicketsController` and `ProjectsController` controllers. This will allow admin users to create new users in the application, without them needing to sign up first.

The first part of CRUD is the creation of a resource, so it would be a great idea to start with that. Begin by creating a new feature in a new file called `spec/features/admin/creating_users_spec.rb`. Use the code from the following listing for this new feature.

Listing 180. spec/features/admin/creating_users_spec.rb

```
require "rails_helper"

RSpec.feature "Admins can create new users" do
  let(:admin) { FactoryBot.create(:user, :admin) }

  before do
    login_as(admin)
    visit "/"
    click_link "Admin"
    click_link "Users"
    click_link "New User"
  end

  scenario "with valid credentials" do
    fill_in "Email", with: "newbie@example.com"
    fill_in "Password", with: "password"
    click_button "Create User"
    expect(page).to have_content "User has been created."
  end
end
```

When you run this feature using `bundle exec rspec spec/features/admin/creating_users_spec.rb`, the first couple of lines in the `before` block pass, but it fails due to a missing Admin link:

```
1) Admins can create new users with valid credentials
   Failure/Error: click_link "Admin"

   Capybara::ElementNotFound:
     Unable to find link "Admin"
```

This is a link that we'll have in the top navigation, linking to our admin area. Of course, you need this link for the feature to pass, but you want to show it only for admins. You can use the `admins_only` helper you defined earlier and put the link in `app/views/layouts/application.html.erb`, after the "Home" link:

Listing 181. The new 'admin' link in the top navigation

```
<ul class="navbar-nav mr-auto">
    <li class="nav-item <%= "active" if current_page?("/") %>">
      <%= link_to "Home", root_path, class: "nav-link" %>
    </li>
    <% admins_only do %>
      <li>
        <%= link_to "Admin", admin_root_path, class: "nav-link" %>
      </li>
    <% end %>
  <% unless user_signed_in? %>
    ...
```

This way, the link will only be shown to users who are admins. Now when you run the feature again using `bundle exec rspec spec/features/admin/creating_users_spec.rb`, you should get a little bit further:

```
1) Admins can create new users with valid credentials
   Failure/Error: click_link "Users"

   Capybara::ElementNotFound:
     Unable to find link "Users"
```

The admin homepage we created earlier doesn't have a link to "Users" in it. It sounds like a good candidate for the "Admin Links" menu that we have on the page!

Edit the `app/views/admin/application/index.html.erb` view and add a link to "Users" in the "Admin Links" menu:

Listing 182. Adding a link to manage users in the "Admin links"

```
<h5 class="text-uppercase mt-4">Admin Links</h5>
<ul class="nav flex-column">
  <li><%= link_to "Users", admin_users_path %></li>
</ul>
```

What exactly did we just link to? Nothing yet! Let's fix that.

8.4.1. The index action

We don't yet have an index page for Users, and it would make sense to have the link to create a new user in the header section on a Users page, like we did for projects and tickets.

So we'll create a `UsersController`, in our `Admin` namespace, and use it for the rest of this section. Run the following command to generate a new `Admin::UsersController`, with an `index` action prepopulated.

```
$ rails g controller admin/users index
```

This generates some odd output:

```
$ rails g controller admin/users index
      create  app/controllers/admin/users_controller.rb
       route  namespace :admin do
  get 'users/index'
  end
      invoke  erb
      ...
```

It's generated another admin namespace in your `config/routes.rb` file, with `get 'users/index'` inside it. But we already have an admin namespace, and we'll want to put a `users` resource in it so we can add the rest of the CRUD actions as well.

Replace the two `namespace :admin` declarations in your `config/routes.rb` with the following:

Listing 183. Creating a single `admin` namespace from the generated routes

```
namespace :admin do
  root "application#index"

  resources :projects, except: [:index, :show]
  resources :users
end
```

And we'll also have to tweak the controller that got generated, to make sure it extends from the `Admin::ApplicationController` we created earlier. Open `app/controllers/admin/users_controller.rb` and replace the first line with the following:

Listing 184. app/controllers/admin/users_controller.rb

```
class Admin::UsersController < Admin::ApplicationController
```

Now because the generator also generated a skeleton view, our test will get a bit further.

```
1) Admins can create new users with valid credentials
   Failure/Error: click_link "New User"

   Capybara::ElementNotFound:
     Unable to find link "New User"
```

Listing users

So what content should be in the admin user index? A list of all the users in the system might be a great start. Edit the `Admin::UsersController` again and load a list of users to display, in the `index` action:

Listing 185. app/controllers/admin/users_controller.rb

```
class Admin::UsersController < Admin::ApplicationController
  def index
    @users = User.order(:email)
  end
end
```

Next, you need to rewrite the template for this action, which lives at `app/views/admin/users/index.html.erb`, so it contains the "New User" link and lists all the

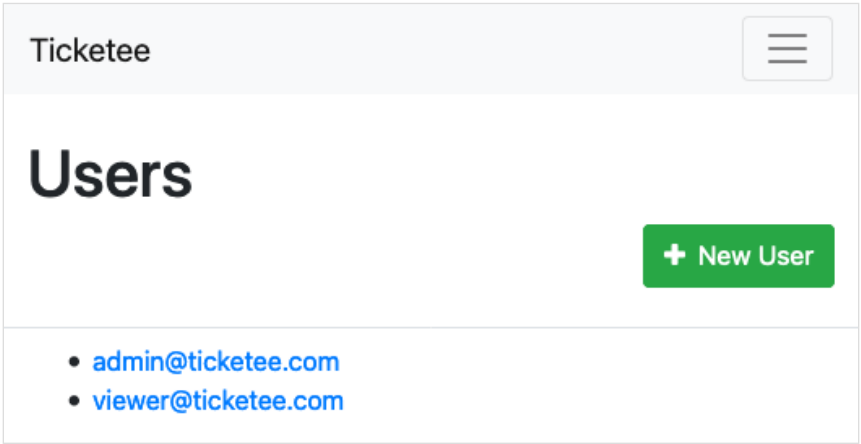
users loaded by the controller. Use the code from the following listing.

Listing 186. app/views/admin/users/index.html.erb

```
<header>
  <h1>Users</h1>

  <ul class="actions">
    <li>
      <%= link_to "New User", new_admin_user_path, class: "new" %>
    </li>
  </ul>
</header>

<ul>
  <% @users.each do |user| %>
    <li><%= link_to user.email, [:admin, user] %></li>
  <% end %>
</ul>
```

In this example, when you specify a `Symbol` as an element in the route for the `link_to`, Rails uses that element as a literal part of the route generation, making it use `admin_user_path` rather than `user_path`. You saw this in chapter 5 when you used it with `[:edit, project, ticket]`, but it bears repeating here.

With this view in place, we'll now have a very simple page that lists the users of our application:



When you run `bundle exec rspec spec/features/admin/creating_users_spec.rb` again, you're told the `new` action is missing:

```
1) Admins can create new users with valid credentials
   Failure/Error: click_link "New User"
   AbstractController::ActionNotFound:
     The action 'new' could not be found for Admin::UsersController
```

Great! This means that the test is able to navigate to the admin area, then to the `index` page for `UsersController`, and then it's able to click the "New User" link. We're getting through this feature pretty quickly.

8.4.2. The new action

Add the `new` action `Admin::UsersController` now by using this code:

Listing 187. The `new` action in `Admin::UsersController`

```
def new
  @user = User.new
end
```

And make the view for this action at `app/views/admin/users/new.html.erb`:

Listing 188. The `new` view for `Admin::UsersController`

```
<header>
  <h1>New User</h1>
</header>

<%= render "form", user: @user %>
```

Next, you need to create the form partial that's used in the `new` template, which you can do by using the following code. It must contain the `email` and `password` fields, which are the bare essentials for creating a user:

Listing 189. app/views/admin/users/_form.html.erb

```
<%= bootstrap_form_with(model: [:admin, user], local: true, label_errors: true) do |form| %>
  <%= form.text_field :email %>
  <%= form.text_field :password %>

  <%= form.primary %>
<% end %>
```

For this `bootstrap_form_with`, you use the array form you saw earlier with `[project, ticket]`, but this time you pass in a symbol rather than a model object. Rails interprets the symbol literally, generating a route such as `admin_users_path` rather than `users_path`, which would be generated if we used `bootstrap_form_with user` instead. You can also use this array syntax with `link_to` (seen earlier) and `redirect_to` helpers. Any symbol passed anywhere in the array for any of these methods is interpreted literally. The same goes for strings.

When you run the feature again with `bundle exec rspec spec/features/admin/creating_users_spec.rb`, you're told there's no action called `create`:

```
1) Admins can create new users with valid credentials
   Failure/Error: click_button "Create User"
   AbstractController::ActionNotFound:
     The action 'create' could not be found for Admin::UsersController
```

8.4.3. The create action

Let's create that action now by using this code:

Listing 190. The `create` action of `Admin::UsersController`

```
def create
  @user = User.new(user_params)

  if @user.save
    flash[:notice] = "User has been created."
    redirect_to admin_users_path
  else
    flash.now[:alert] = "User has not been created."
    render "new"
  end
end

private

def user_params
  params.require(:user).permit(:email, :password)
end
```

You've used this same pattern in a few controllers, so it should be old-hat by now.

There's now an "Admin" link in the top navigation menu that an admin can click, which takes them to the `index` action in `Admin::ApplicationController`. On the template rendered for this action (`app/views/admin/application/index.html.erb`) is a "Users" link that goes to the `index` action in `Admin::UsersController`. On the template for this action is a "New User" link that presents the user with a form to create a user. When the user fills in this form and clicks the "Create User" button, it goes to the `create` action in `Admin::UsersController`.

With all the steps implemented, our feature should now pass. Find out with a final run of `bundle exec rspec spec/features/admin/creating_users_spec.rb`:

```
1 examples, 0 failures
```

Great! Run `bundle exec rspec` to make sure everything's still working:

```
28 examples, 1 failure, 2 pending
```

What have we broken **this** time??

It turns out it isn't anything that we've broken - again it's auto-generated tests that we don't even want, in:

- `spec/helpers/admin/users_helper_spec.rb`,
- `spec/views/admin/users/index.html.erb_spec.rb` and
- `spec/requests/admin/users_request_spec.rb`.

You can delete all three of those files. Run `bundle exec rspec` again just to make sure:

```
30 examples, 0 failures
```

This is another great middle point for a commit, so let's do so now.

```
$ git add .
$ git commit -m "Add the ability to create users through the admin
  backend"
$ git push
```

Although this functionality allows you to create new users through the admin backend, it

doesn't let you create admin users. That's up next.

8.4.4. Creating admin users

We have only one admin user in our application right now: the `admin@ticketee.com` user that gets generated from the seed. What if we wanted to have more than just that? Sure, we could add more users to the seeds file, but that seems like hard work. Let's look at another way of doing it.

To create users who are admins, you can have a check box on the form you use to create a new user. When this check box is selected and the `User` record is saved, that user will be an admin.

To get started, let's add another scenario to the `spec/features/admin/creating_users_spec.rb` using the code from the following listing.

Listing 191. spec/features/admin/creating_users_spec.rb

```
scenario "when the new user is an admin" do
  fill_in "Email", with: "admin@example.com"
  fill_in "Password", with: "password"
  check "Is an admin?"
  click_button "Create User"
  expect(page).to have_content "User has been created."
  expect(page).to have_content "admin@example.com (Admin)"
end
```

When you run `bundle exec rspec spec/features/admin/creating_users_spec.rb`, it fails when it attempts to select the "Is an Admin?" check box.

```
1) Admins can create new users when the new user is an admin
   Failure/Error: check "Is an admin?"
   Capybara::ElementNotFound:
     Unable to find checkbox "Is an admin?" that is not disabled
```

You need to add this check box to the form for creating users, which you can do by adding the following code to the `simple_form_for` block in `app/views/admin/users/_form.html.erb`:

Listing 192. Adding a form field for setting the `admin` attribute

```
...
<%= form.password_field :password %>
<%= form.check_box :admin, label: "Is an admin?" %>
...
```

Because we've added a new field to this form, we will need to add it to the list of permitted parameters in `Admin::UsersController` by changing the `user_params` method to this:

Listing 193. Permitting the `admin` attribute in `Admin::UsersController`

```
def user_params
  params.require(:user).permit(:email, :password, :admin)
end
```

With this check box in place, when you re-run `rspec spec/features/admin/creating_users_spec.rb`, it can't find "admin@example.com (Admin)" on the page:

```
1) Admins can create new users when the new user is an admin
   Failure/Error: expect(page).to have_content "admin@example.com
   (Admin)"
     expected to find text "admin@example.com (Admin)" in "Ticketee
     Toggle navigation Home Admin Signed in as test2@example.com Sign
     out User has been created. Users New User admin@example.com..."
```

The problem is that only the user's email address is displayed: no text appears to indicate the person is an admin. To get this text to appear, change the display of the user in `app/views/admin/users/index.html.erb` from this

```
<li><%= link_to user.email, [:admin, user] %></li>
```

to this:

```
<li><%= link_to user, [:admin, user] %></li>
```

By not calling any methods on the `user` object and attempting to write it out of the view, you cause Ruby to call `to_s` on this method, which by default outputs something similar to the

following (which isn't human friendly):

Listing 194. Default `to_s` output on an ActiveRecord model

```
#<User:0xb6fd6054>
```

You can override the `to_s` method on the `User` model to provide the string containing the email and admin status of the user, by putting the following code in the class definition in `app/models/user.rb`, underneath the `devise` call:

Listing 195. Overriding `to_s` on an ActiveRecord model

```
def to_s
  "#{email} (#{admin? ? "Admin" : "User"})"
end
```

The `to_s` method will now output something like "user@example.com (User)" if the user isn't an admin or "admin@example.com (Admin)" if the user is an admin. With the `admin` field set and an indication displayed on the page regarding whether the user is an admin, the feature should pass when you run `bundle exec rspec spec/features/admin/creating_users_spec.rb`.

```
2 examples, 0 failures
```

This is another great time to commit; and again, run `bundle exec rspec` to make sure everything works:

```
31 examples, 0 failures
```

Good stuff. Push it:

```
$ git add .
$ git commit -m "Add the ability to create admin users through the admin
  backend"
$ git push
```

Now you can create normal and admin users through the backend. In the future, you may need to modify an existing user's details or delete a user, so we'll examine the updating and

deleting parts of the CRUD next.

8.4.5. Editing users

This section focuses on adding updating capabilities for `Admin::UsersController`. As usual, you start by writing a feature to cover this functionality. Place the file at `spec/features/admin/editing_users_spec.rb`, and fill it with the content from the following listing.

Listing 196. spec/features/admin/editing_users_spec.rb

```
require "rails_helper"

RSpec.feature "Admins can change a user's details" do
  let(:admin_user) { FactoryBot.create(:user, :admin) }
  let(:user) { FactoryBot.create(:user) }

  before do
    login_as(admin_user)
    visit admin_user_path(user)
    click_link "Edit User"
  end

  scenario "with valid details" do
    fill_in "Email", with: "newguy@example.com"
    click_button "Update User"

    expect(page).to have_content "User has been updated."
    expect(page).to have_content "newguy@example.com"
    expect(page).to_not have_content user.email
  end

  scenario "when toggling a user's admin ability" do
    check "Is an admin?"
    click_button "Update User"

    expect(page).to have_content "User has been updated."
    expect(page).to have_content "#{user.email} (Admin)"
  end
end
```

When we run the feature with `bundle exec rspec spec/features/admin/editing_users_spec.rb`, we'll see both of the scenarios fail this way:

```
1) Admins can change a user's details with valid details
   Failure/Error: visit admin_user_path(user)
   AbstractController::ActionNotFound:
     The action 'show' could not be found for Admin::UsersController
```

This failure is at the very start of the test, when visiting a user's details page in the admin area. We've got a link to the `show` action of `Admin::UsersController`, but the action isn't defined. Define the `show` action in `Admin::UsersController`, directly under the `index` action.

Grouping the different parts of CRUD is conventionally done in this order: index, show, new, create, edit, update, destroy. It's not a hard-and-fast rule, but consistency makes controllers easier to read and follow.

The `show` action can just be a blank action:

Listing 197. Adding the `show` action to `Admin::UsersController`

```
def show
end
```

The `show` action template requires a `@user` variable, so you should create a `set_user` method that you can call as a `before_action` in `Admin::UsersController`. This is just like the `set_project` and `set_ticket` methods, defined in `ProjectsController` and `TicketsController` respectively. Define this new `set_user` method under all the other methods already in the controller, because it will be a `private` method:

```
def set_user
  @user = User.find(params[:id])
end
```

You then need to call this method using a `before_action`, which should run before the `show`, `edit`, `update`, and `destroy` actions. Put this line at the top of your class definition for `Admin::UsersController`:

Listing 198. Defining the `before_action` in `Admin::UsersController`

```
class Admin::UsersController < Admin::ApplicationController
  before_action :set_user, only: %i(show edit update destroy)
  ...
```

With the `set_user` and `show` methods in place in the controller, what's the next step? Find out by running `bundle exec rspec spec/features/admin/editing_users_spec.rb` again. You will see this error:

```
1) Admins can change a user's details with valid details
   Failure/Error: visit admin_user_path(user)
     ActionController::MissingExactTemplate:
       Admin::UsersController#show is missing a template for request formats: text/html
```

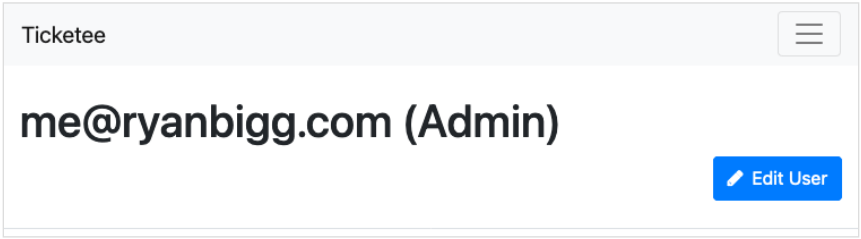
You can write the template for the `show` action to make this step pass. This file goes at `app/views/admin/users/show.html.erb`, and should use the following code:

Listing 199. app/views/admin/users/show.html.erb

```
<header>
  <h1><%= @user %></h1>

  <ul class="actions">
    <li>
      <%= link_to "Edit User", [:edit, :admin, @user], class: "edit" %>
    </li>
  </ul>
</header>
```

The `<h1>` at the top of this view will display the "user@example.com (User)" or "user@example.com (Admin)" text, and the "Edit User" link will allow us to navigate to the action where we can edit this user's details. This is a pretty boring view for now, but later on we might want to add details to it, like the list of projects a user belongs to, or their activity history on the site, or something like that.

Here's what it looks like without all that activity information now:



When you run `bundle exec rspec spec/features/admin/editing_users_spec.rb`, the line that visits the user's details page passes, and you're on to the next step:

```
1) Admins can change a user's details with valid details
   Failure/Error: click_link "Edit User"
   AbstractController::ActionNotFound:
     The action 'edit' could not be found for Admin::UsersController
```

Good; you're progressing nicely. You created the `show` action for `Admin::UsersController`, which displays information about a user to a signed-in admin user. Now you need to create the `edit` action so admin users can edit a user's details.

8.4.6. The edit and update actions

Add the `edit` action directly under the `create` action in your controller. It should be another blank method like `show`:

Listing 200. Defining an `edit` action in `Admin::UsersController`

```
def edit
end
```

With this action defined and the `@user` variable used in its view already set by the `before_action`, you now create the template for this action at `app/views/admin/users/edit.html.erb`. This template renders the same form partial as the `new` template:

```
<header>
  <h1>Edit User</h1>
</header>

<%= render "form", user: @user %>
```

OK, you've dealt with the current failure for the feature. Find out what's next with another run of `bundle exec rspec spec/features/admin/editing_users_spec.rb`. You should be told the `update` action doesn't exist:

```
1) Admins can change a user's details with valid details
   Failure/Error: click_button "Update User"
   AbstractController::ActionNotFound:
     The action 'update' could not be found for Admin::UsersController
```

Indeed it doesn't, so let's create it. Add the `update` action to `Admin::UsersController` underneath the `edit` action, as shown in the following listing. You don't need to set up the `@user` variable here because the `set_user before_action` does it for you.

Listing 201. app/controllers/admin/users_controller.rb

```
def update
  if @user.update(user_params)
    flash[:notice] = "User has been updated."
    redirect_to admin_users_path
  else
    flash.now[:alert] = "User has not been updated."
    render "edit"
  end
end
```

Looks like a standard update action. Let's re-run the test and see what happens now:

```
1) Admins can change a user's details with valid details
   Failure/Error: expect(page).to have_content "User has been updated."
     expected to find text "User has been updated." in "Ticketee Toggle
     navigation Home Admin Signed in as test1@example.com Sign out User
     has not been updated. Edit User Email PasswordPassword can't be
     blank Is an admin?"
```

We expected the user to be updated, but they weren't. Why not? The error message that tells us the actual content on the page gives us the answer (albeit a little hidden) - the user's password can't be blank.

The application has naively tried to take the data we submitted, which included an empty password field, and update the user with those details, which is triggering our validations that require a user to have a password. But why is the password blank? Because we don't store user passwords directly in the database - we only store hashed versions of them.[61]

In this case, if we submit a blank password, it should mean "don't change the user's current password". So we'll remove the field if it's blank, before we update the user. Above `if @user.update(user_params)`, insert this code:

```
if params[:user][:password].blank?
  params[:user].delete(:password)
end
```

Now the entire action looks like this.

Listing 202. app/controllers/admin/users_controller.rb, with blank password removal

```
def update
  if params[:user][:password].blank?
    params[:user].delete(:password)
  end

  if @user.update(user_params)
    flash[:notice] = "User has been updated."
    redirect_to admin_users_path
  else
    flash.now[:alert] = "User has not been updated."
    render "edit"
  end
end
```

When you run `bundle exec rspec spec/features/admin/editing_users_spec.rb` again, all the scenarios should pass:

```
2 examples, 0 failures
```

In this section, you added two more actions to `Admin::UsersController`: `edit` and `update`. Admin users can now update users' details as they please.

As always, run `bundle exec rspec` to make sure you didn't break everything. Just one quick run will show this:

```
33 examples, 0 failures
```

Done! Let's make a commit for this new feature:

```
$ git add .
$ git commit -m "Add ability for admins to edit and update users"
$ git push
```

With the updating done, there's only one more part to go for your admin CRUD interface: deleting users.

8.4.7. Archiving users

There comes a time in an application's life when you need to remove users from your app. Maybe they asked for their account to be closed. Maybe they were being pesky and you wanted to kick them out. Or maybe you have another reason to remove them. Whatever the case, having the functionality to remove users is helpful.

However, in Ticketee users have a trail of activity behind them - they can create tickets on projects. In the future, they'll also be able to take other actions in the system that we want to keep for posterity. Deleting users isn't the right action to take, then - we'll archive them instead, so we can still see everything they've done, but they can take no further part in the system, they can't even sign in anymore.

Keeping with the theme so far, you'll first write a feature for archiving users (using the following listing) and put it at `spec/features/admin/archiving_users_spec.rb`.

Listing 203. spec/features/admin/archiving_users_spec.rb

```
require "rails_helper"

RSpec.feature "An admin can archive users" do
  let(:admin_user) { FactoryBot.create(:user, :admin) }
  let(:user) { FactoryBot.create(:user) }

  before do
    login_as(admin_user)
  end

  scenario "successfully" do
    visit admin_user_path(user)
    click_link "Archive User"

    expect(page).to have_content "User has been archived"
    expect(page).not_to have_content user.email
  end
end
```

When you run this feature using `bundle exec rspec spec/features/admin/archiving_users_spec.rb`, you get right up to the first line in the scenario with no issue. Then it complains:

```
1) An admin can archive users successfully
   Failure/Error: click_link "Archive User"
   Capybara::ElementNotFound:
     Unable to find link "Archive User"
```

Of course, you need the "Archive User" link. Add it to `app/views/admin/users/show.html.erb` right under the "Edit User" link:

Listing 204. Adding a "Archive User" link to app/views/admin/users/show.html.erb

```
<li>
  <%= link_to "Archive User", [:archive, :admin, @user], method: :patch,
    data: { confirm: "Are you sure you want to archive this user?"},
    class: "delete" %>
</li>
```

This is the first time we've decided to stray away from Rails' default RESTful resources, and the seven default routes provided. Archiving doesn't fit in with the list of `index`, `show`, `new`, `create`, `edit`, `update` or `destroy` - so we'll make a new route for it, called `archive`. This is why we've specified the symbol `archive` in the URL for the link - `[:archive, :admin, @user]`. It's another example of the polymorphic routing that we saw earlier in section 5.1.

The HTTP method we've selected for the link is `patch`, the same HTTP method that the `update` action uses. We are, in effect, updating a user - but it's a very specific kind of update.

When we re-run the spec, we'll get a different error:

```
1) An admin can archive users successfully
   Failure/Error: visit admin_user_path(user)
   ActionView::Template::Error:
     undefined method `archive_admin_user_path' for ...
```

We can define this method as a member route on your `users` resource. A member route provides the routing helpers and, more importantly, the route itself to a custom controller action for a single instance of a resource.

`member` routes vs. `collection` routes

When you're looking at defining custom routes outside the normal seven RESTful routes, you'll come across these terms - member routes and collection routes. The difference can often be confusing for people learning Rails - which type do you use, and when?

Collection routes are typically used when you want to perform an action on a group of model instances. `index` is an example of a collection route - other examples might be `search`, or `autocomplete`, or `export`. These routes will generate URLs like "/projects/search" or "/projects/export".

Member routes are typically used when you want to perform an action on a single model instance. The `edit`, `update`, and `destroy` are all examples of member routes - they first find an instance of a model, and then take some action on it. Other examples might be `archive`, or `approve`, or `preview`. These routes will generate URLs like "/projects/1/archive" or "/projects/3/approve".

To define the new member route, change the `resources :users` line in the `admin` namespace inside `config/routes.rb` to the following:

Listing 205. Defining the `archive` member route for a `User` resource

```
namespace :admin do
  ...
  resources :users do
    member do
      patch :archive
    end
  end
end
```

Inside the `member` block here, you specify that each `user` resource has a new action called `archive` that can be accessed through a `PATCH` request. As stated previously, by defining the route in this fashion, you also get the `archive_admin_user_path` helper, which is what you've used in `app/views/admin/users/show.html.erb`.

You need to add the `archive` action next, directly under the `update` action in `Admin::UsersController`.

Listing 206. app/controllers/admin/users_controller.rb

```
def archive
  @user.archive!

  flash[:notice] = "User has been archived."
  redirect_to admin_users_path
end
```

We're not thinking about what it means to actually archive a user yet - we just want to call the `archive` method on the user and be done with it. We'll also need to modify the call to `before_action` in our `Admin::UsersController`, to add this new `archive` action to the list of actions it will run before. If we don't do this, the `@user` variable won't be instantiated correctly.

Listing 207. Running `set_user` before the `archive` action

```
class Admin::UsersController < Admin::ApplicationController
  before_action :set_user, only: %i(show edit update destroy archive)

  ...
```

When you run `bundle exec rspec spec/features/admin/archiving_users_spec.rb`, the error you get now is different:

```
1) An admin can archive users successfully
  Failure/Error: @user.archive!

  NoMethodError:
    undefined method `archive!' for #<User:0x00007f847dac3088>
```

What does it mean to actually archive a user? We could have a simple boolean field on our `User` model called `archived`, that we set to `true` if the user is archived, or `false` if they are not. Or alternatively, we could keep with the chronological tracking of events - we record and display when tickets were created, so perhaps we should record when users were archived instead. In keeping with Rails' convention for naming timestamp fields, let's add a new field called `archived_at` to our `User` model, that will store a timestamp of when a user was archived. Run the following command in your terminal:

```
$ rails g migration add_archived_at_to_users archived_at:timestamp
```

Again, Rails is smart enough to infer what we want to do, and will generate a migration that looks like this:

Listing 208. db/migrate/[timestamp]_add_archived_at_to_users.rb

```
class AddArchivedAtToUsers < ActiveRecord::Migration[6.1]
  def change
    add_column :users, :archived_at, :timestamp
  end
end
```

We don't need to modify this migration, it's good to run as-is. Run it with `rails db:migrate`.

Now we can look at filling in the missing method in our model. Remember, our last test failure was about a missing `archive` method on our `User` model. To mark a user as archived, what we would need to do is set the `archived_at` timestamp on the user, and then save it. We can do that by adding the following method to the `User` model located in `app/models/user.rb`:

Listing 209. Archiving a user

```
class User < ActiveRecord::Base
  ...

  def archive!
    self.update(archived_at: Time.now)
  end
end
```

Like we said, archiving is a very specific form of updating a user, so we can use the same `update` method, which will update the attributes and then save the changes.

When we re-run our archiving spec with `bundle exec spec features/admin/archiving_users_spec.rb`, it's nearly complete:

```
1) An admin can archive users successfully
   Failure/Error: expect(page).not_to have_content user.email
     expected not to find text "test2@example.com" in "Ticketee Toggle
     navigation Home Admin Signed in as test1@example.com Sign out User
     has been archived. Users New User test1@example.com (Admin)
     test2@example.com (User)"
```

We've archived the user, but they still appear in the list of users - we're not doing anything in

our `index` action to not show users that are archived. Our `index` action in our `Admin::UsersController` just looks like this:

Listing 210. Loading all users in the `index` action of `Admin::UsersController`

```
class Admin::UsersController < Admin::ApplicationController
  ...
  def index
    @users = User.order(:email)
  end
  ...
```

We can use a feature called scoping to limit the list of users that we show. Scopes are methods that you can define on your Active Record models, very similar to class methods - methods you call on the class itself, not an instance of the class. `order` above is an example of a class method.

Inside our `User` model, we can define a scope to only find users that aren't archived. Define it between the `devise` and `to_s` methods.

Listing 211. app/models/user.rb

```
class User < ActiveRecord::Base
  ...
  scope :active, lambda { where(archived_at: nil) }
  ...
```

Users that don't have an `archived_at` date must, by definition, not be archived. You can then alter the `index` action of your controller, to call this scope like so:

Listing 212. Using the `active` scope in the `index` action

```
class Admin::UsersController < Admin::ApplicationController
  ...
  def index
    @users = User.active.order(:email)
  end
  ...
```

The reason you write these scopes in your model is because the controller isn't responsible for knowing things like what defines an archived user - only the `User` model cares about the difference between archived and not archived.

You can write scopes for all kinds of things - for example, you could write scopes to find all users that have created more than one ticket, or to find users that have created tickets for a specific project. For now we've only written a very simple scope, as a demonstration.

Now when you run your archiving spec with `bundle exec rspec spec/features/admin/archiving_users_spec.rb`, it will pass happily:

```
1 example, 0 failures
```

There's one small problem with this feature, though: it doesn't stop you from archiving yourself!

8.4.8. Ensuring that you can't archive yourself

To make it impossible to archive yourself, you can add another scenario to `spec/features/admin/archiving_users_spec.rb`, as shown in the following listing.

Listing 213. spec/features/admin/archiving_users_spec.rb

```
scenario "but cannot archive themselves" do
  visit admin_user_path(admin_user)
  click_link "Archive User"

  expect(page).to have_content "You cannot archive yourself!"
end
```

When you run this feature with `bundle exec rspec spec/features/admin/archiving_users_spec.rb`, the first two lines of the scenario pass, but the third one fails—as you might expect, because you haven't added the message. Change the `archive` action in `Admin::UsersController` as follows.

Listing 214. app/controllers/admin/users_controller.rb

```
def archive
  if @user == current_user
    flash[:alert] = "You cannot archive yourself!"
  else
    @user.archive!
    flash[:notice] = "User has been archived."
  end

  redirect_to admin_users_path
end
```

Now, before the `archive` method does anything, it checks to see if the user attempting to be deleted is the current user and, if so, stops the process with the "You cannot archive yourself!" message. When you run `bundle exec rspec spec/features/admin/archiving_users_spec.rb` this time, the scenario passes:

```
2 examples, 0 failures
```

Great! With the ability to delete users implemented, you've completed the CRUD for `Admin::UsersController` and for the entire `users` resource. Make sure you haven't broken anything by running `bundle exec rspec`. You should see this output:

```
35 examples, 0 failures
```

Fantastic! Commit and push that:

```
$ git add .
$ git commit -m "Add feature for archiving users, including protection
  against self-archiving"
$ git push
```

8.4.9. Preventing archived users from signing in

There's just one last feature we need to build, as part of archiving users, and we alluded to it earlier - archived users should no longer be able to sign in to Ticketee.

To verify that this is the case, let's add another scenario to the "Signing in" feature we created

in chapter 6, in `spec/features/signing_in_spec.rb`.

Listing 215. Testing that archived users cannot sign in

```
RSpec.feature "Users can sign in" do
  ...

  scenario "unless they are archived" do
    user.archive!

    visit "/"
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: "password"
    click_button "Sign in"

    expect(page).to have_content "Your account has been archived."
  end
end
```

This looks very similar to the previous scenario for successful sign in, except we're calling `user.archive` before filling in the sign in form. When an archived user tries to sign in, we should show them a nice "Your account has been archived" message.

If you run this feature with `bundle exec rspec spec/features/signing_in_spec.rb`, your new scenario will fail:

```
1) Users can sign in unless they are archived
   Failure/Error: expect(page).to have_content "Your account has been archived."
     expected to find text "Your account has been archived." in
     "Ticketee Toggle navigation Home Signed in as test2@example.com
     Sign out Signed in successfully. Projects"
```

This is expected - we haven't yet configured our app to not allow archived users to sign in.

Devise determines if a user can sign in to our app with a method called `active_for_authentication?`. Each of the Devise strategies we listed in chapter 6 (`lockable`, `confirmable`, and so on) can add conditions to determine whether or not a user is able to sign in - for example, the `lockable` strategy will overwrite this `active_for_authentication?` method to return false if the user's record is locked. If the method returns false, then the user is not allowed to sign in.

We can write our own `active_for_authentication?` method in our `User` model, to not allow

authentication if a user is archived. We can do that with the following method, defined at the bottom of our `User` model:

Listing 216. app/models/user.rb

```
class User < ActiveRecord::Base
  ...
  def active_for_authentication?
    super && archived_at.nil?
  end
end
```

The call to `super` in this method will allow all of the other checks to take place, to make sure the user's account is unlocked, and confirmed, etc. If we left that out, we'd be stopping archived users from signing in, but we'd also be allowing locked users or unconfirmed users to sign in, as long as they weren't archived. Not good.

This looks like it's been way too easy, but that's the power of leveraging well-written gems. If we re-run our signing in spec, we'll see the following:

```
1) Users can sign in unless they are archived
   Failure/Error: expect(page).to have_content "Your account has been
   archived."
     expected to find text "Your account has been archived." in
     "Ticketee Toggle navigation Home Sign up Sign in Your account is
     not activated yet. Sign In Email Password Remember me Sign up..."
```

We've stopped the user from signing in! But we're not showing the right message back to the user, as to why their sign in failed.

To do this, we can overwrite another method provided by Devise, called `inactive_message`. This method will get called by Devise when `active_for_authentication?` returns `false`, and should return the translation key of the message that should be displayed to the user.

We haven't looked at translations and internationalization (I18n) yet in Rails, but the framework has a great system built in to allow your apps to be fully multilingual, and Devise has complete support for it. Let's define this `inactive_message` method below `active_for_authentication?` in our `User` model, to look like this:

Listing 217. Defining the message that gets displayed back to the user

```
def inactive_message
  archived_at.nil? ? super : :archived
end
```

If the user isn't archived (`archived_at` is `nil`), then there was some other reason why they couldn't log in - so we call `super` again. If the user's account is locked, or unconfirmed, this allows those strategies to supply the correct message, to let the user know why they couldn't log in.

But if `archived_at` isn't `nil`, then we return this `:archived` symbol. What does this symbol mean? If we re-run our specs after defining this method, we'll see what it does:

```
1) Users can sign in unless they are archived
   Failure/Error: expect(page).to have_content "Your account has been
   archived."
     expected to find text "Your account has been archived." in
     "Ticketee Toggle navigation Home Sign up Sign in
     translation missing: en.devise.failure.user.archived ..."
```

It uses the symbol to look up a translation, which we haven't defined. Devise provides a lot of its own translations, generated when you ran `rails g devise:install` - these translations are located in `config/locales/devise.en.yml`. If you look inside that file, you'll see a tree structure of YAML data:

Listing 218. The start of config/locals/devise.en.yml

```
en:
  devise:
    confirmations:
      confirmed: "Your email address has been successfully confirmed."
      ...
```

The keys on each level of the tree are added together to define the final translation key - the key shown above would be `en.devise.confirmations.confirmed`. Knowing this, we can define our missing `en.devise.failure.user.archived` key. There's already a section below `confirmations` in the file called `failure`, so inside that we can define new levels for `user` and `archived`, like so:

Listing 219. Defining a custom translation

```
en:
  devise:
    confirmations:
      ...
    failure:
      ...
      user:
        archived: "Your account has been archived."
    mailer:
      ...
```

Once you've defined this custom translation, you can re-run your spec with `bundle exec rspec spec/features/signing_in_spec.rb`:

```
2 examples, 0 failures
```

Fantastic! You've customized some of Devise's functionality to prevent archived users from signing in to Ticketee, and backed it up with tests. You've also learnt a little bit about how i18n works in Rails. Run `bundle exec rspec` to make sure your changes haven't broken anything else:

```
36 examples, 0 failures
```

Let's commit and push these changes.

```
$ git add .
$ git commit -m "Archived users cannot sign in to the app"
$ git push
```

With this final commit, you've created your admin section. It provides a great CRUD interface for users in this system so that admins can modify their details when necessary.

8.5. Summary

For this chapter, you dove into basic access control and added a field called `admin` to the `users` table. You used this `admin` field to allow and restrict access to a namespaced controller, as well as show and hide links.

Then you wrote the CRUD interface for the `users` resource under the `admin` namespace, including archiving users and then forbidding those archived users from signing in. You rounded out the chapter by not allowing users to delete themselves.

[57] https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STARTED.md#traits
[58] In Ruby, `nil` and `false` are both considered to be "falsey" and won't pass a conditional test (eg. `puts "foo" if false`). Everything else is considered "truthy", including things like `0`, `[]`, `""`, and `{}`. This can confuse people coming from other languages!
[59] Read more about view spec testing at https://www.relishapp.com/rspec/rspec-rails/v/3-0/docs/view-specs/view-spec.
[60] `only` works as a whitelist, listing the routes that should be generated. There is also a blacklist version to list the routes that should not be generated, called `except`. http://guides.rubyonrails.org/routing.html#restricting-the-routes-created
[61] For a brief writeup on why password hashing is so important, see http://security.blogoverflow.com/2011/11/why-passwords-should-be-hashed/

Chapter 9. File uploading

This chapter focuses on file uploading, the next logical step in a ticket-tracking application. Sometimes, when people file a ticket on an application such as Ticketee, they want to attach a file to provide more information for that ticket, because words alone can only describe so much. For example, a ticket description saying "This button should move up a bit" could be better explained with a picture showing where the button is now and where it should be. Users may want to attach any kind of file: a picture, a crash log, a text file, you name it. Currently, Ticketee has no way to attach files to the ticket: people would have to upload them elsewhere and then include a link with their ticket description.

By providing Ticketee with the functionality to attach files to the ticket, you give the project owners a useful context that will help them more easily understand what the ticket creator means. Luckily, Active Storage allows you to implement this feature easily.

Active Storage helps you attach files to Active Record objects. It comes with a local storage option for development and testing. Active Storage also connects with cloud services like Amazon AWS and allows you to upload your files to the cloud and then display them in the app. Active Storage is a gem that was added to the core collection of Rails gems in Rails 5.2. This gem provides similar functionality to other gems like Paperclip, Carrierwave and Dragonfly but is now part of Rails.

File uploading is also useful in other types of applications. Suppose you wrote a Rails application for a book. You could upload the chapters to this application, and then people could provide notes on those chapters. Another example is a photo gallery application that allows you to upload images of your favorite cars for people to vote on. It could also allow for video uploading to a website to share travel VLOGs with the world, like YouTube. File uploading has many different uses, and is a cornerstone of many Rails applications.

9.1. Attaching a file

You'll start off by letting users attach files when they begin creating a ticket. As explained before, files attached to tickets can provide useful context about what feature a user is requesting or can point out a specific bug. A picture is worth a thousand words, as they say. It doesn't have to be an image; it can be any type of file. This kind of information can be helpful when attempting to solve issues that tickets bring up.

To provide this functionality, you must add a file upload box to the new ticket page, which allows users to select a file to upload. When the form is submitted, the file is submitted along with it.

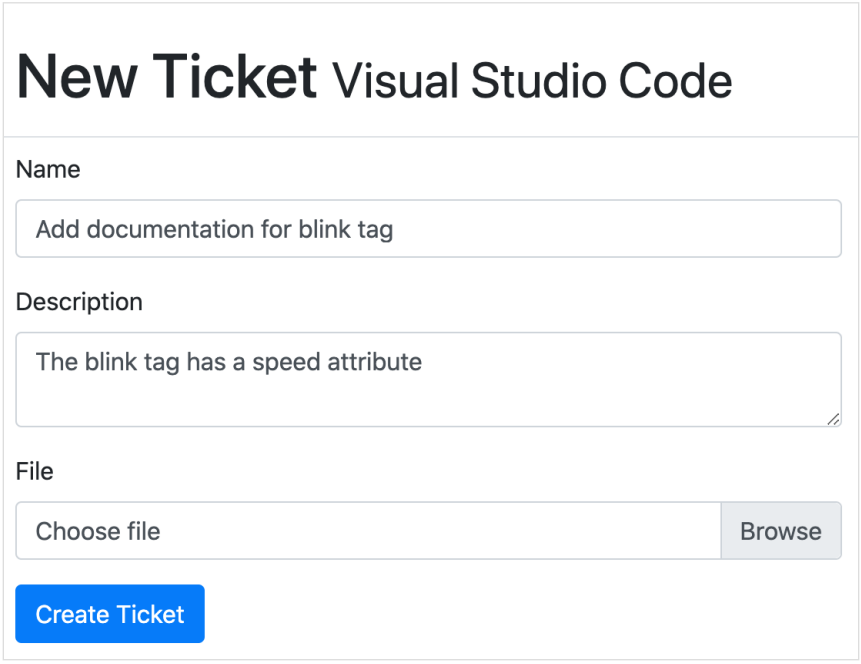


Figure 40. File upload example

9.1.1. A feature featuring files

You first need to write a scenario to make sure the functionality works. This scenario shows you how to deal with file uploads when creating a ticket. Users should be able to create a ticket, select a file, and upload it. Then they should be able see this file, along with the other ticket details, on the ticket's page. They may choose to click the filename, which would download the file. Let's test all this by adding a scenario at the bottom of `spec/features/creating_tickets_spec.rb` that creates a ticket with an attachment,[62] as shown in the following listing.

Listing 220. spec/features/creating_tickets_spec.rb

```
RSpec.feature "Users can create new tickets" do
  ...

  scenario "with an attachment" do
    fill_in "Name", with: "Add documentation for blink tag"
    fill_in "Description", with: "The blink tag has a speed attribute"
    attach_file "File", "spec/fixtures/speed.txt"
    click_button "Create Ticket"

    expect(page).to have_content "Ticket has been created."

    within(".ticket .attachment") do
      expect(page).to have_content "speed.txt"
    end
  end
end
```

This feature introduces a new concept: the `attach_file` method of this scenario, which attaches the file found at the specified path to the specified field. The path here is deliberately in the `spec/fixtures` directory because you may use this file for functional tests later. This directory would usually be used for test fixtures, but at the moment you don't have any.[63] Create the `spec/fixtures/speed.txt` file, and fill it with some random filler text like this:

```
The blink tag can blink faster if you use the speed="hyper" attribute.
```

Try running this feature using `bundle exec rspec spec/features/creating_tickets_spec.rb` and see how far you get. It fails on the `attach_file` line because the `File` field isn't available yet:

```
1) Users can create new tickets with an attachment
   Failure/Error: attach_file "File", "spec/fixtures/speed.txt"

   Capybara::ElementNotFound:
     Unable to find file field "File" that is not disabled
```

Add the "File" field to the ticket form partial directly under the `<%= form.text_area :description %>` field, using the code in the following listing.

Listing 221. app/views/tickets/_form.html.erb

```
<%= form.file_field :attachment, label: "File" %>
```

You call this field `attachment` internally, but the user will see "File". We're using attachment rather than file for naming because there's already a `File` class in Ruby. `Attachment` is good alternative you can use, and it describes well what you're doing - adding attachments to a ticket.

If you run `bundle exec rspec spec/features/creating_tickets_spec.rb` again, it fails with this error instead:

```
1) Users can create new tickets with an attachment
  Failure/Error:
  within(".ticket .attachment") do
    expect(page).to have_content "speed.txt"
  end

  Capybara::ElementNotFound:
    Unable to find css ".ticket .attachment"
```

You can see that the scenario failed because Capybara can't find the text in this element on `show` view of `TicketsController`: neither text nor element exist! So we'll add some output in the view, to show the details of any file that's been uploaded to the ticket. Spice it up by adding the file size there, too, as shown in the following listing.

Listing 222. app/views/tickets/show.html.erb

```
...
<%= simple_format(@ticket.description) %>

<% if @ticket.attachment.present? %>
  <h4>Attachment</h4>
  <div class="attachment">
    <p>
      <%= link_to @ticket.attachment.filename,
          rails_blob_path(@ticket.attachment, disposition: 'attachment') %>
      (<%= number_to_human_size(@ticket.attachment.attachment.blob.byte_size) %>)
    </p>
  </div>
<% end %>
```

If you run `bundle exec rspec spec/features/creating_tickets_spec.rb` again, it fails with this error instead:

```
1) Users can create new tickets with valid attributes
  Failure/Error: <% if @ticket.attachment.present? %>

  ActionView::Template::Error:
    undefined method `attachment' for #<Ticket:0x00007fa2cf5856a0>
    Did you mean?  attachment_changes
  # ./app/views/tickets/show.html.erb:33:in
`_app_views_tickets_show_html_erb___4548633037445968198_70168620071740'
  # ./spec/features/creating_tickets_spec.rb:17:in `block (2 levels) in <main>'
  # ------------------
  # --- Caused by: ---
  # NoMethodError:
  #   undefined method `attachment' for #<Ticket:0x00007fa2cf5856a0>
  #   Did you mean?  attachment_changes
  #   ./app/views/tickets/show.html.erb:33:in
`_app_views_tickets_show_html_erb___4548633037445968198_70168620071740'
```

It looks like we need to define an `attachment` method on our Ticket model. There is something that can help us with that.

9.1.2. Enter stage right, Active Storage

Uploading files is something many web applications need to allow, which makes it perfect functionality to put into a gem. The current best-of-breed gem in this area is Active Storage. Active Storage makes uploading files easy. When you need more advanced features, such as processing uploaded files or storing them in something like Amazon S3 rather than on your web server, Active Storage is there to help you, too. We will go into this in more detail in Chapter 13.

To setup Active Storage, you will need to run the `active_storage:install` generator:

Listing 223. Installing Active Storage

```
bin/rails active_storage:install
```

This generator creates a migration that will create two new tables in your application's database: `active_storage_attachments` and `active_storage_blobs`. Information from uploaded files, such as file names and file sizes, goes into the `blobs` table. From there, blobs

are associated with objects from your application via records in the `attachments` table.

To run the migration and add the new tables to your development environment's database, run the migrations:

```
rails db:migrate
```

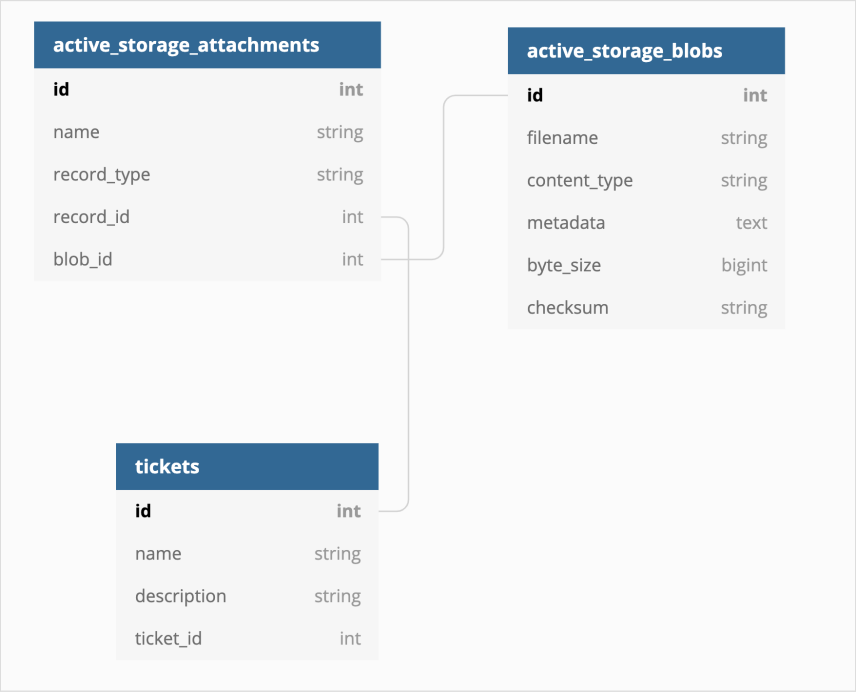


Figure 41. Database Diagram

Active Storage comes with some helpers that you can use to implement attachments in Rails. It provides a helper method `has_one_attached` that you can add into your model to link it to an attachment without needing to create another model to store the attachments. Rails will provide that model for us!

Let's look at our view code again:

Listing 224. app/views/tickets/show.html.erb

```
...
<%= simple_format(@ticket.description) %>

<% if @ticket.attachment.present? %>
  <h4>Attachment</h4>
  <div class="attachment">
    <p>
      <%= link_to @ticket.attachment.filename,
          rails_blob_path(@ticket.attachment, disposition: 'attachment') %>
      (<%= number_to_human_size(@ticket.attachment.attachment.blob.byte_size) %>)
    </p>
  </div>
<% end %>
```

To define a link to this attachment, we use <code>rails_blob_path</code>. This will give us a URL for this specific attachment. In this case, the URL for this file would be something like http://localhost:3000/rails/active_storage/blobs/eyJfcmFpbH…&#8203;. It’s quite long!

So, where is the route for these uploaded files defined? Well its not defined in the `routes.rb` but Rails has automatically defined some routes to serve the active_storage files.

You can see these routes by running this on the command line:

```
rails routes -c active_storage
```

This will show you that there are several routes. The one we are interested in is the `rails_service_blob` route. This route is used to download the file. It is slightly different though, we are using `rails_blob` but the route is `rails_service_blob`. Rails has defined a url_helper to make it easier for us which passes the signed_id and filename to the `rails_service_blob` route.

Under the filename, you display the size of the file using this code:

```
(<%= number_to_human_size(@ticket.attachment.attachment.blob.byte_size) %>)
```

The first `attachment` call here will give us the associated attachment. The second one will give

us access to the actual `ActiveStorage::Attachment` object, and then from there we can access information about the file itself with `blob`. An attachment's blob contains this `byte_size` field, which stores the size of the file in bytes. Bytes aren't that particularly useful for people to see! For instance, does 361704 bytes make more sense than 361kB? No!

To convert it to a human-readable output, (such as "361 Kilobytes"), you use the `number_to_human_size` Action View helper.

Finally, we're going to need to write some more code to give us access to that `attachment` method on `@ticket`. To do that, we can use the `has_one_attached` method in the `Ticket` model. This will define the methods that we need to work with this file. Add it to your `Ticket` model with this line:

Listing 225. Adding a file field to your `Ticket` model

```
class Ticket < ActiveRecord::Base
  ...
  has_one_attached :attachment
end
```

This `attachment` method is now defined, but it's not yet over!

9.1.3. Using Active Storage

You have one last thing to do: update your controller to allow you to pass in the uploaded attachment, when creating or editing a ticket. Change the `ticket_params` method in `app/controllers/tickets_controller.rb` to look like this:

```
class TicketsController < ApplicationController
  ...
  def ticket_params
    params.require(:ticket).permit(:name, :description, :attachment)
  end
  ...
end
```

This will allow the attachment to be submitted through on any form, such as the `new` form. If we do not make this change, files will not be attached to tickets!

With this final change and with the uploaded file's information now being output in

`app/views/tickets/show.html.erb`, this feature will now pass when you run `bundle exec rspec spec/features/creating_tickets_spec.rb`:

```
4 examples, 0 failures
```

Awesome! Your files are being uploaded and taken care of by ActiveStorage, which stores them at `public/uploads`. Let's see if your changes have brought destruction or salvation by running `bundle exec rspec`:

```
37 examples, 0 failures
```

Sweet salvation! Let's commit and push this:

```
$ git add .
$ git commit -m "Add the ability to attach a file to a ticket"
$ git push
```

Great! Now you can attach a file to a ticket.

There's still some work to do, though. What would happen if somebody wanted to add more than one file to a ticket? Let's look at how to do that next.

9.1.4. Implementing multiple-file upload

You have an interface for attaching a single file to a ticket, but no way for a user to attach more than one. Let's imagine your pretend client asked you to boost the number of file input fields on this page from one to three.

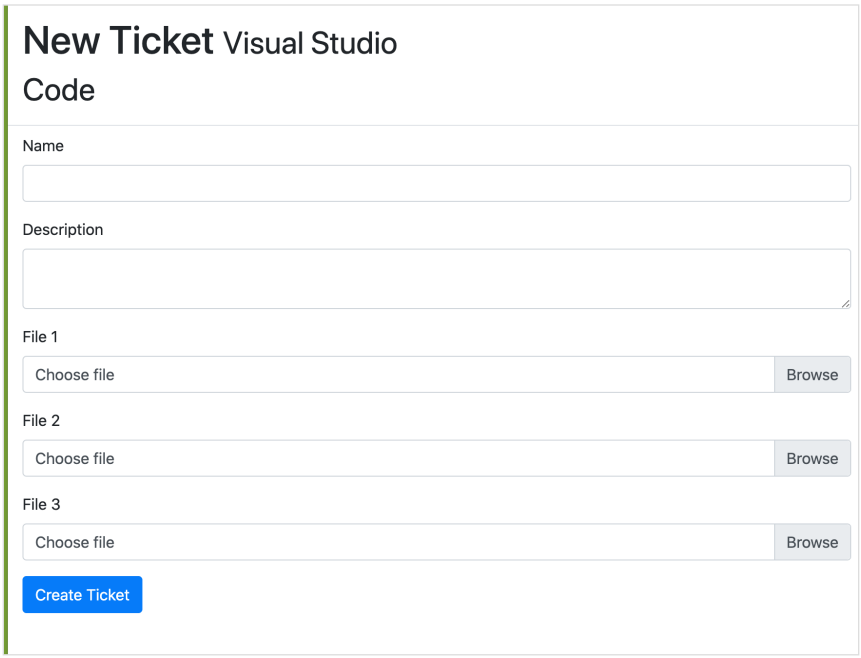


Figure 42. Mockup of multiple upload fields

To help create an interface to upload multiple files we will use the Dropzone JavaScript library. Dropzone is an open source javascript library that provided drag and drop, image previews and multiple file upload.

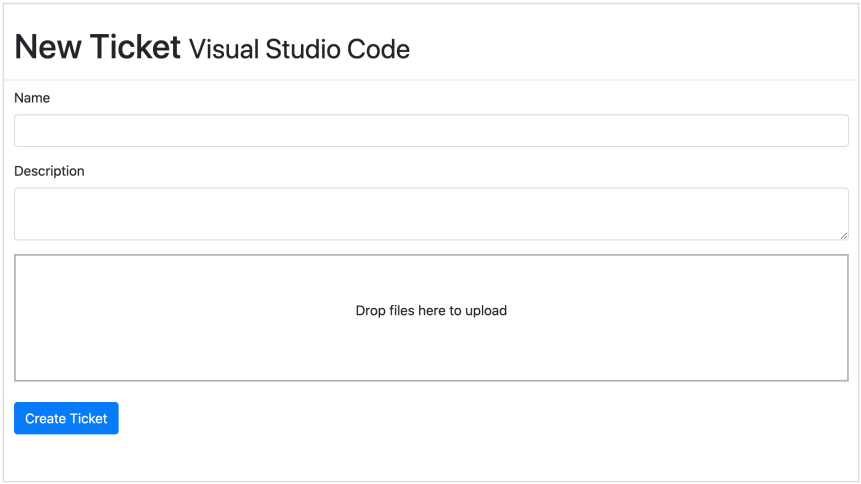


Figure 43. How the Dropzone interface will look

9.1.5. JavaScript testing

When you introduce JavaScript into your application, you have to run any scenarios that rely on it through another piece of software called Selenium WebDriver. Selenium WebDriver is a browser driver that was installed when the `selenium-webdriver` gem was installed, so you

don't have to do anything to set it up. Capybara without Selenium WebDriver won't run JavaScript because Capybara doesn't support JavaScript by itself. By running these JavaScript-reliant scenarios through Selenium WebDriver, you ensure that the JavaScript will be executed. One of the great things about this Selenium WebDriver and Capybara partnership is that you can use the same old, familiar Capybara steps to test JavaScript behavior in real browsers.

Capybara provides an easy way to trigger WebDriver testing. You tag a scenario (or feature) with the `js: true` option, and it launches a new web browser window and tests your code by using the same steps as standard Capybara testing, but in a real browser. Isn't that neat? Let's take the scenario for creating a ticket with an attachment from `spec/features/creating_tickets_spec.rb`, and update it by adding two additional file uploads. We'll also update the name of the scenario, so the entire scenario looks like the following listing.

Listing 226. File attachment scenario, spec/features/creating_tickets_spec.rb

```
scenario "with multiple attachments", js: true do
  fill_in "Name", with: "Add documentation for blink tag"
  fill_in "Description", with: "The blink tag has a speed attribute"

  attach_file("spec/fixtures/gradient.txt", class: 'dz-hidden-input', visible: false)
  attach_file("spec/fixtures/speed.txt", class: 'dz-hidden-input', visible: false)
  attach_file("spec/fixtures/spin.txt", class: 'dz-hidden-input', visible: false)

  click_button "Create Ticket"

  expect(page).to have_content "Ticket has been created."

  within(".ticket .attachments") do
    expect(page).to have_content "speed.txt"
    expect(page).to have_content "spin.txt"
    expect(page).to have_content "gradient.txt"
  end
end
```

The `js: true` option at the top of this scenario tells Capybara that the scenario should use JavaScript, so it should be run in a browser using Selenium WebDriver. Up until this point, our tests have been running in a simulated (pretend) browser. Now that we need some features of JavaScript, it's time to run them in a real browser. By default, the browser that will be used will be Firefox.

In this scenario, you attach three files to your ticket and assert that you see them within the `attachments` element, which was previously called `.ticket .attachment` but now has the pluralized name of `.ticket .attachments`.

We are now checking for the hidden input field that Dropzone creates instead of the file upload box.

We are using the `attach_file` helper here. This will attach the file that is supplied as the first argument to the class as specified. This method has several options for selecting this element. `visible:false` also needs to be added as this field is a hidden field and not normally picked up by the Web Driver unless specified.

Dropzone adds a hidden input on the page, as shown in the below image. This hidden input is where the image will be attached to and submitted in the form. When an image is uploaded with Dropzone it will append the metadata for that image into the hidden field. When you submit the form, rails will attach this file to the model using this metadata.

The dropzone hidden input

image::dz-hidden-input.png

Now run this single scenario using `bundle exec rspec spec/features/creating_tickets_spec.rb`. It should fail on the first `attach_file` step, it cannot find the new hidden field that Dropzone creates:

```
1) Users can create new tickets with multiple attachments
  Failure/Error: dropzone = find(".dz-hidden-input", visible: false)

  Capybara::ElementNotFound:
    Unable to find css ".dz-hidden-input"
```

Now we need to add a div in the code in place of the file upload. Dropzone will automatically find this div because of the `dropzone` and `uploader` class, and attach itself to this div when the page loads. We will write some Javascript later that configures how the uploader works and this will hook into these classes as well.

Add this code into your view below the other form elements to have somewhere for Dropzone to replace when it loads on the page:

Listing 227. File attachment div, app/views/tickets/_form.html.erb

```
<% upload_url = upload_file_project_tickets_path(project) %>
<div class="dropzone uploader mb-2" data-url="<%= upload_url %>"></div>
```

Now run the tests and we should get a lot of failures like so:

```
1) Users can create new tickets with multiple attachments
  Failure/Error: <div class="dropzone uploader" data-url="<%=
upload_file_project_tickets_path(project) %>"></div>

  ActionView::Template::Error:
    undefined method `upload_file_project_tickets_url' for
#<#<Class:0x00007fdbab8becf8>:0x00007fdbab8af0f0>
```

It looks like we are missing a url. Add this extra route inside `routes.rb`

Listing 228. config/routes.rb

```
resources :projects, only: [:index, :show, :edit, :update] do
  resources :tickets do
    collection do
      post :upload_file
    end
  end
end
```

We also need to add this new controller action. We are creating a new controller action here in order to upload the file seperately, and then attach it later when we create the ticket. This will also help us later with another feature.

Listing 229. app/controllers/tickets_controller.rb

```
def upload_file
  blob = ActiveStorage::Blob.create_and_upload!(io: params[:file], filename: params[
:file].original_filename)
  render json: { signedId: blob.signed_id }
end
```

Let's run that test again to see what we need to do next. We'll run it with this command: `bundle exec rspec spec/features/creating_tickets_spec.rb`

```
1) Users can create new tickets with multiple attachments
    Failure/Error: attach_file("spec/fixtures/gradient.txt", class: 'dz-hidden-input',
visible: false)

    Capybara::ElementNotFound:
      Unable to find file field nil with classes [dz-hidden-input] that is not disabled
    # ./spec/features/creating_tickets_spec.rb:46:in `block (2 levels) in <main>'
```

This is because we arent replacing our div with the Dropozone uploader yet. In order to install dropzone we will use yarn to add it as a javascript package. You can do this with the following command:

```
yarn add dropzone
```

The test is currently failing because the Dropzone file fields and uploader are not appearing on the page. To make the Dropzone uploader appear on the page, we need to add some JavaScript to our application. We can add the JavaScript to the `app/javascript` directory in our application. Create a new directory in here called `src`, we will use this folder to store all of our JavaScript files and then include them in our `application.js` file.

Listing 230. Dropzone, app/javascript/src/dropzone.js

```
const Dropzone = require("dropzone");
import "dropzone/dist/dropzone.css";

Dropzone.autoDiscover = false;

document.addEventListener("turbolinks:load", () => {
  const uploader = document.querySelector(".uploader");
  if (uploader) {
    const appendAttachment = (signedId) => {
      const input = document.createElement("input");
      input.type = "hidden";
      input.name = "attachments[]";
      input.value = signedId;
      uploader.parentElement.append(input);
    };

    const csrfTokenTag = document.querySelector('meta[name="csrf-token"]');
    const csrfToken = csrfTokenTag && csrfTokenTag["content"];

    let headers;
    if (csrfToken) {
      headers = {
```

```
        "X-CSRF-TOKEN": csrfToken,
      };
    } else {
      headers = {};
    }

    var myDropzone = new Dropzone(".uploader", {
      url: uploader.dataset.url,
      headers: headers,
    });

    myDropzone.on("success", (file, response) => {
      appendAttachment(response.signedId);
    });

     myDropzone.on("addedfile", function (file, xhr, formData) {
        var submit = document.querySelector("input[type=submit]");
        submit.disabled = true;
    });

    myDropzone.on("queuecomplete", function ( ) {
        var submit = document.querySelector("input[type=submit]");
        submit.disabled = false;
    });

  }
});
```

There is a lot to unpack in this JavaScript so let go through it. First up we are including the Dropzone library and CSS that is provided. We installed Dropzone using yarn and are requiring it in order to use it later in this file.

On the next line we turn autoDiscovery off. This is the feature we mentioned earlier where Dropzone will attach itself to the `.dropzone` css class as this time we are manually going to load it with the options we have specified.

On the next line we are listening for the "turbolinks:load" event before running the rest of the code. We have seen turboloinks earlier in Chapter 3. By waiting for this event we are making sure to load the uploader every time the turbolinks "loads" a new page. Either through a new page load or if someone clicks a link and turbolinks is only loading the content that is required when that link is clicked to speed up the page.

In order to replace the autoDiscover functionality of Dropzone, we need to find the div that we want to add it to, we are doing this here with `querySelector` and if we find this element,

we continue with the rest of the code.

With `const appendAttachment` we are creating a new hidden input on the page that will contain the signedId that will be sent to the `TicketsController` actions to attach the blobs to the `Ticket` instances. We will also need to ensure that this uploader sends the CSRF token (that we first mentioned in Chapter 3) in order to prevent CSRF attacks.

We are also disabling the submit button while the file is still being uploaded. This will stop a user from uploading a file and hitting submit before it was finished uploading. Our test will also fail if this is not set because it works at super human speed and will often click the button before the uploads have finished.

Finally, we need to initiate the Dropzone uploader with all of these options and add it to the page.

In order for this javascript code to run, we need to include it. Add it into application.js underneath require("bootstrap");

Listing 231. Application, app/javascript/packs/application.js

```
require("../src/dropzone");
```

By adding this line here, we will also be requiring Dropzone's CSS assets too. These will automatically be added to the <head> tag in the development and test environments, but we will also need them added in the production environment.

To make that happen, we will need to add this additional line to our application layout:

Listing 232. app/views/layouts/application.html.erb

```
<%= stylesheet_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
```

This line will load in any CSS that comes from Webpack, across all of our environments.

Now run the test.

```
Failure/Error: attach_file("spec/fixtures/gradient.txt", class: 'dz-hidden-input', visible:
false)

Capybara::FileNotFound:
  cannot attach file, spec/fixtures/gradient.txt does not exist
```

It's failing because we're now trying to attach three different files to the new file field, but we only have one file, named `speed.txt`.

We can fix the error by creating the other files that the spec is asking for - `spin.txt` and `gradient.txt` - within the same `spec/fixtures` folder. We'll populate them with some goofy content, for the fun of it, as well.

Listing 233. spec/fixtures/spin.txt

```
Spinning blink tags have a 200% higher click rate!
```

Listing 234. spec/fixtures/gradient.txt

```
Everything looks better with a gradient!
```

This is random filler meant only to provide some easily-distinguishable text if you ever need to reference it. On the next run of `bundle exec rspec spec/features/creating_tickets_spec.rb:43`, the error is now replaced with a new one:

```
1) Users can create new tickets with multiple attachments
  Failure/Error:
  within(".ticket .attachment") do
    expect(page).to have_content "speed.txt"
    expect(page).to have_content "spin.txt"
    expect(page).to have_content "gradient.txt"
  end

  Capybara::ElementNotFound:
    Unable to find css ".ticket .attachments"
```

This is because we have updated the test to check for multiple attachments but we arent displaying them yet.

In order to implement multiple upload we will need to change our model to have multiple attachments. We will also need to make it plural as it will not return and array of multiple

attachment. Change the `has_one_attached` line in the model to `has_many_attached` like so:

Listing 235. app/models/ticket.rb

```
has_many_attached :attachments
```

After we do this we will lose all of the existing attachments on tickets. To fix this we will need to create a data migration to fix the existing items. To update the exsiting attachments, we will use a rake task to update the existing attachments. Create this file: `lib/tasks/data_migrations.rake` with this contents:

Listing 236. lib/tasks/data_migrations.rake

```
namespace :data_migrations do
  desc "Migrate single to many attachments"
  task update_attachments: :environment do
    ActiveStorage::Attachment.where(name: "attachment").update_all(name: "attachments")
  end
end
```

This will load the existing attachments and then update the name to match the new model. Due to the way Active Storage data is saved, we dont need to make any other updates as multiple attachments are stored by adding a row for each attachment on a model instance.

Now run this task with the following command in the console:

```
rails data_migrations:update_attachments
```

Since we have changed it from attachment to attachments we will also need to update the stong params to allow the controller to update this value. Change this in the bottom of the controller like so:

Listing 237. app/controllers/tickets_controller.rb

```
class TicketsController < ApplicationController
  ...

  def ticket_params
    params.require(:ticket).permit(:name, :description, attachments: [])
  end
  ...
```

We also need to add a way to attach these attachments to the model, so add this line `@ticket.images.attach(params[:attachments])` in your model in the create method:

Listing 238. app/controllers/tickets_controller.rb

```
class TicketsController < ApplicationController
  ...

  def create
    @ticket = @project.tickets.build(ticket_params)
    @ticket.author = current_user
    if params[:attachments].present?
      @ticket.attachments.attach(params[:attachments])
    end
    if @ticket.save
  ...
```

While you're changing things, you may as well update `app/views/tickets/show.html.erb` to reflect what we plan to do with our attachments. Instead of a ticket having a single attachment accessible via `@ticket.attachment`, it will have multiple attachments accessible via `@ticket.attachments`. We can then iterate over the attachments, and print out the details of each.

Listing 239. app/views/tickets/show.html.erb

```
<% if @ticket.attachments.present? %>
  <h4>Attachment(s)</h4>
  <div class="attachments">
    <% @ticket.attachments.each do |attachment| %>
      <p>
        <%= link_to attachment.filename,
          rails_blob_path(attachment, disposition: 'attachment') %>
        (<%= number_to_human_size(attachment.blob.byte_size) %>)
      </p>
    <% end %>
  </div>
<% end %>
```

Now when you re-run the specs in `spec/features/creating_tickets_spec.rb`, you should see this output:

```
4 examples, 0 failures
```

You can re-run all of your specs with `bundle exec rspec` to make sure nothing else is broken:

```
37 examples, 0 failures
```

Awesome — let's commit and push this:

```
$ git add .
$ git commit -m "Add dropzone and support for multiple file uploads"
$ git push
```

One thing we are missing is the ability to update a ticket with new attachments. Lets add a new test for that:

Listing 240. spec/features/editing_tickets_spec.rb

```
scenario "with multiple attachments", js: true do
  click_link "Edit Ticket"
  attach_file("spec/fixtures/spin.txt", class: 'dz-hidden-input', visible: false)
  expect(page).to have_content "spin.txt"

  click_button "Update Ticket"

  expect(page).to have_content "Ticket has been updated."

  within(".ticket .attachments") do
    expect(page).to have_content "spin.txt"
  end
end
```

If you run this test this should now fail with:

```
Failure/Error:
  within(".ticket .attachments") do
    expect(page).to have_content "spin.txt"
  end

Capybara::ElementNotFound:
  Unable to find css ".ticket .attachments"
```

This is because the attachment was never uploaded.

In order to fix this we will need to add something in our update method to ensure the

attachments are updated. Underneath `@ticket.update(ticket_params)` add:

```
if params[:attachments].present?
  @ticket.attachments.attach(params[:attachments])
end
```

Now run the test again with `bundle exec rspec spec/features/editing_tickets_spec.rb` and it should pass.

```
3 examples, 0 failures
```

And all of the tests will still pass when we run `bundle exec rspec`:

```
38 examples, 0 failures
```

Time to commit this.

```
$ git add .
$ git commit -m "Add updating attachments on tickets"
$ git push
```

You're done with multiple file upload!

9.2. Summary

This chapter covered two flavors of file uploading: single-file uploading and multiple-file uploading. You first saw how to upload a single file by using ActiveStorage to handle the file when it arrives in your application.

After you conquered single-file uploading, you tackled multiple-file uploading. You installed the Dropzone JavaScript library and change Active Storage to support this and the uploading of multiple files.

After multiple-file uploading, you learned how to ensure that attachments are kept when a validation error is shown to users if they incorrectly enter data in to one of the other fields.

In the next chapter, you'll look at giving tickets a concept of state, which enables users to see which tickets need to be worked on and which are closed. Tickets will also have a default

state so they can be easily identified when they’re created.

[62] This attachment references the `blink` tag. Note that although the `blink` tag was once a part of HTML, you should never use it. The same goes for the `marquee` tag. We reference them in our text files to add some light humor to the scenario, not because documentation for these tags is a good idea.
[63] Nor will you ever, because factories replace them in your application.

Chapter 10. Tracking state

In a ticket-tracking application such as Ticketee, tickets aren't there to provide information about a particular problem or suggestion; rather, they're there to provide the workflow for it.

The general workflow of a ticket starts when a user files the ticket; the ticket will be classified as a "new" ticket. When the developers of the project look at this ticket and decide to work on it, they'll switch the state on the ticket to "open," and once they're done they'll mark it as "resolved." If a ticket needs more information, they'll add another state, such as "needs more info." A ticket could also be a duplicate of another ticket, or it could be something that the developers determine isn't worth working on. In cases such as these, the ticket may be marked as "duplicate" or "invalid."

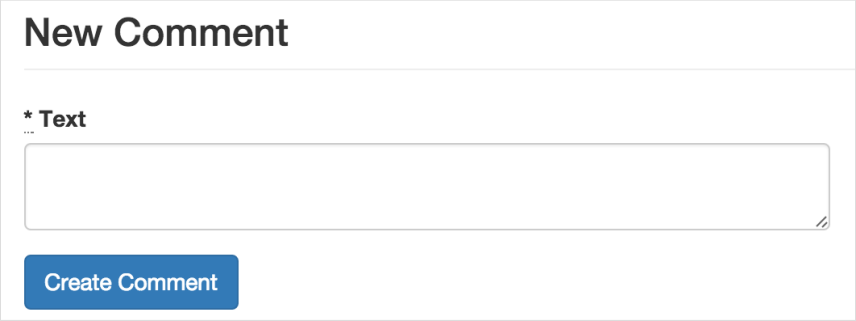
The point is that tickets have a workflow, and that workflow revolves around state changes. In this chapter, you'll allow the admin users to add states, but not to delete them. If an admin were to delete a state that was used, you'd have no record of that state ever existing. It's best that states not be deleted once they've been created and used on a ticket.

Alternatively, states you want to delete could be moved into an "archived" state so they couldn't be assigned to new tickets but still would be visible on older tickets.

To track the states, you'll let users leave a comment. Users will be able to leave a text comment about the ticket, and may also elect to change the state of the ticket by selecting another state from a drop-down list. But not all users will be able to leave a comment and change the state - you'll protect both creating a comment and changing the state from unauthorized access. By the time you're done with all of this, the users of your application will have the ability to add comments to tickets. Some users, depending on their permissions, will be able to change the state of a ticket through the comment interface. You'll begin by creating the interface through which a user will create a comment, and then you'll build the ability for the user to change the state of a ticket on top of that. Let's get into it.

10.1. Leaving a comment

The first step is to add the ability to leave a comment. Once you've done so, you'll have a simple form that looks like this figure:



To get started, you'll write a Capybara feature that goes through the process of creating a comment. When you're done with this feature, you'll have a comment form at the bottom of the `show` view for the `TicketsController`, which you'll then use as a base for adding a `state` select box later on. You'll put this feature in a new file at `spec/features/creating_comments_spec.rb` and put this test in it:

Listing 241. spec/features/creating_comments_spec.rb

```
require "rails_helper"

RSpec.feature "Users can comment on tickets" do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project) }
  let(:ticket) { FactoryBot.create(:ticket,
    project: project, author: user) }

  before do
    login_as(user)
  end

  scenario "with valid attributes" do
    visit project_ticket_path(project, ticket)

    within(".comments") do
      fill_in "Text", with: "Added a comment!"
      click_button "Create Comment"
    end

    expect(page).to have_content "Comment has been created."

    within(".comments") do
      expect(page).to have_content "Added a comment!"
    end
  end

  scenario "with invalid attributes" do
    visit project_ticket_path(project, ticket)
    click_button "Create Comment"

    expect(page).to have_content "Comment has not been created."
  end
end
```

Here you jump straight to the ticket's details page, fill in the comment box with the some text, and create your comment. The visiting of the ticket is inside the scenarios rather than in the `before` block, because you'll use this same feature for permission checking later on. Try running this feature now by running `bundle exec rspec spec/features/creating_comments_spec.rb`. You'll see this output:

```
1) Users can comment on tickets with valid attributes
   Failure/Error: fill_in "Text", with: "Added a comment!"
   Capybara::ElementNotFound:
     Unable to find field "Text"
   # ...
   # ./spec/features/creating_comments_spec.rb:17:in ...

2) Users can comment on tickets with invalid attributes
   Failure/Error: click_button "Create Comment"
   Capybara::ElementNotFound:
     Unable to find button "Create Comment"
   # ...
   # ./spec/features/creating_comments_spec.rb:27:in ...
```

The failing specs mean that you've got work to do! We'll start with the first scenario, the "happy" path. The label the spec is looking for is going to belong to the comment box underneath your ticket's information. Neither the label nor the field are there, and that's what's the scenario requires, so now's a great time to add them.

10.1.1. The comment form

Let's begin to build the comment form for the application. This comment form will initially consist of a single text field into which the user can insert their comment. Add a single line to the bottom of `app/views/tickets/show.html.erb` to render a comment form partial:

```
<%= simple_format(@ticket.description) %>


<%= render "comments/form", ticket: @ticket, comment: @comment %>
```

This line renders the partial from `app/views/comments/_form.html.erb`. The `app/views/comments` directory won't exist yet, so go ahead and create that first. Then, create this file:

Listing 242. app/views/comments/_form.html.erb

```
<header>
  <h3>New Comment</h3>
</header>

<%= bootstrap_form_with model: [ticket, comment], local: true do |f| %>
  <%= f.text_area :text %>
  <%= f.primary %>
<% end %>
```

This is pretty much the standard `simple_form_for`, except you use the `Array`- argument syntax again, which will generate a nested route. You need to do four things before this form will work.

1. You must define the `@comment` variable in the `show` action of your `TicketsController`. This will reference a new `Comment` instance, and give this `bootstrap_form_with` something to work with.
2. You need to create the `Comment` model, and associate it with your `Ticket` model. This is so you can create new comment records from the form, and associate them with the right ticket.
3. You need to define the nested resource in your routes, so that the `bootstrap_form_with` can make a POST request to the correct URL—one similar to `/tickets/1/comments`. The `bootstrap_form_with` will generate the URL by combining the classes of the objects in the array, and without this, it will give you an undefined method of `ticket_comments_path`.
4. You need to generate the `CommentsController` and the `create` action along with it, so that your form has somewhere to go when a user submits it.

Let's look at each of these in turn. First, you need to set up your `TicketsController` to use the `Comment` model for creating new comments, which you'll create shortly. To do this, you need to first build a new `Comment` object using the `comments` association on your `@ticket` object.

The comment model

The first step to getting this "Creating comments" feature is to define a `@comment` variable for the form, in the `show` action of your `TicketsController`. Open up `app/controllers/tickets_controller.rb` and add the line to define the `@comment` instance, as shown in the following listing.

Listing 243. app/controllers/tickets_controller.rb

```
class TicketsController < ApplicationController
  ...
  def show
    @comment = @ticket.comments.build
  end
  ...
```

This action will use the `build` method on the `comments` association for your `@ticket` object (which is set up by `before_action :set_ticket`) to create a new `Comment` object for the view's `form_with`.

Next, you'll generate the `Comment` model, so that you can define the `comments` association on your `Ticket` model. This model will need to have an attribute called `text` for the text from the form, a foreign key to link it to a ticket, and another foreign key to link to a user record, so we know who wrote the comment. You can generate this model using the following command:

```
$ rails g model comment text:text ticket:references author:references
```

This will generate a migration that looks like the following:

Listing 244. db/migrate/[timestamp]_create_comments.rb

```
class CreateComments < ActiveRecord::Migration[6.1]
  def change
    create_table :comments do |t|
      t.text :text
      t.references :ticket, null: false, foreign_key: true
      t.references :author, null: false, foreign_key: true

      t.timestamps
    end
  end
end
```

There's just one tiny change we need to make to this file before we can run it, and it's the same change we made to the migration for adding authors to tickets. By default, using `foreign_key: true` on `t.references :author` will mean that it tries to make a link to an `authors` table, which we don't have - an author is actually a user. So we need to remove the `foreign_key: true` from that line, and manually add another foreign key below. It should look

like this:

Listing 245. db/migrate/[timestamp]_create_comments.rb, with fixed foreign keys

```
class CreateComments < ActiveRecord::Migration[6.1]
  def change
    create_table :comments do |t|
      t.text :text
      t.references :ticket, index: true, foreign_key: true
      t.references :author, index: true,
        foreign_key: { to_table: :users }

      t.timestamps null: false
    end
  end
end
```

Once that change is made, you can run the migration for this model on your development database by running this familiar command:

```
$ rails db:migrate
```

With these done, your next stop is to add the `comments` association to the `Ticket` model. Add this line to `app/models/ticket.rb`, below the `has_many :attachments` line:

Listing 246. Defining the `comments` association for tickets

```
class Ticket < ActiveRecord::Base
  ...
  has_many :comments, dependent: :destroy
  ...
```

You don't need to add any associations to the `Comment` model - the model generator did that automatically when you said that `ticket` and `author` were references. But you should add a validation for the `text` field, to make sure the user actually enters some text for the comment. You can do this by adding a validation to `app/models/comment.rb`. It should now look like this:

Listing 247. Our complete `Comment` model

```
class Comment < ActiveRecord::Base
  belongs_to :ticket
  belongs_to :author

  validates :text, presence: true
end
```

This will help your second scenario pass, because it requires that an error message be displayed when you don't enter any text.

While we're here, we should change the `author` association to use the `User` class too:

```
class Comment < ApplicationRecord
  belongs_to :ticket
  belongs_to :author, class_name: "User"

  validates :text, presence: true
end
```

When you run your feature with `bundle exec rspec spec/features/creating_comments_spec.rb` at this midpoint, both specs will fail for the same reason - the `bootstrap_form_for` can't find the routing helper its trying to use:

```
1) Users can comment on tickets with valid attributes
   Failure/Error: visit project_ticket_path(project, ticket)
   ActionView::Template::Error:
     undefined method `ticket_comments_path' for ...

2) Users can comment on tickets with invalid attributes
   Failure/Error: visit project_ticket_path(project, ticket)
   ActionView::Template::Error:
     undefined method `ticket_comments_path' for ...
```

This is because you don't have a nested route for comments inside your tickets resource yet. To define one, you'll need to add it to `config/routes.rb`. Currently in your `config/routes.rb` you've got the tickets resource nested inside the projects resource with these lines:

```
resources :projects, only: [:index, :show, :edit, :update] do
  resources :tickets
end
```

This generates helpers such as `project_tickets_path`. But for your form, it's not important what project the comment is being created for, you only care about the ticket, so you can use `ticket_comments_path` instead. This means you'll need to define a separate non-nested resource for your tickets, and then a nested resource under that for your comments, as shown in the following listing.

Listing 248. config/routes.rb, now with nested `comments` resources

```
resources :projects, only: [:index, :show, :edit, :update] do
  resources :tickets
end

scope path: "tickets/:ticket_id", as: :ticket do
  resources :comments, only: [:create]
end
```

In this new code, we use `scope` which will define a routing scope for our new comments route.

This scope is defined with a `path` option that defines a path prefix for all the routes, turning what would be `/comments/` into `/tickets/:ticket_id/comments`.

The scope also uses `as`, which turns the routing helper from `comments_path` / `comments_url` into `ticket_comments_path` / `ticket_comments_url`.

The last three lines in the listing are the lines that define `ticket_comments_path`, which will make your form work. In general, it's a good idea to only go one level deep with the nesting of your resources. It's very rare that you need two levels of scoping information—one will usually do just fine.

With a route now defined, that error in your spec will be resolved. If you re-run your specs with `bundle exec rspec spec/features/creating_comments_spec.rb`, you'll get a different error:

```
1) Users can comment on tickets with valid attributes
   Failure/Error: click_button "Create Comment"
   ActionController::RoutingError:
     uninitialized constant CommentsController
```

So creating a `CommentsController` will be our next step.

10.1.2. The comments controller

The comments form you created is submitting a POST request to the `create` action of `CommentsController`, so now you need to make it. You can do this by running the following command:

```
$ rails g controller comments
```

A `create` action in this controller will provide the receiving end for the comment form, so you can add this now. You'll need to define a `before_action` in this controller as well, to load the `Ticket` object we'll be creating a comment for. Update your controller to what is shown in the following listing.

Listing 249. app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController
  before_action :set_ticket

  def create
    @comment = @ticket.comments.build(comment_params)
    @comment.author = current_user

    if @comment.save
      flash[:notice] = "Comment has been created."
      redirect_to [@ticket.project, @ticket]
    else
      flash.now[:alert] = "Comment has not been created."
      @project = @ticket.project
      render "tickets/show"
    end
  end

  private

  def set_ticket
    @ticket = Ticket.find(params[:ticket_id])
  end

  def comment_params
    params.require(:comment).permit(:text)
  end
end
```

Most of this controller should look familiar by now. It's long, but there's no new concepts in it.

In the `create` action, first note the `comment_params`. You only allow the `:text` key to make it through, and neither of the two associations, the author or the ticket. This is because the author should always be the `current_user`, and you can get the ID of the ticket from the URL, rather than trust some sort of input from the user.

Next, note that if the `@comment` saves successfully, you redirect back to the ticket's page by passing an `Array` argument to `redirect_to`, which compiles the path from the arguments passed in. This is like what `form_with` does to a nested route such as "/projects/1/tickets/2".

But if your `@comment.save` returns `false`, you'll actually be rendering the `tickets/show` view again, which belongs to an entirely different controller. That's okay, you're allowed to do that, but note that you're just rendering the specified view, not running the entire action, so you'll

need to set up all the right variables that the view needs to render. That's why we're defining `@project` to be the ticket's project - we don't need it in this action, but `tickets/show` needs it to render correctly.

Great, now what's failing next? Let's run our spec with `bundle exec rspec spec/features/creating_comments_spec.rb` again and see.

```
1) Users can comment on tickets with valid attributes
   Failure/Error: within(".comments") do
   Capybara::ElementNotFound:
     Unable to find css ".comments"
```

We're making a lot of progress! We're now able to create a comment in our application, and the "Comment has been created." text displays on the page too. All that is left to do is to show the comment itself.

We made the executive decision to display previous comments on the ticket inside a HTML element with the `class` of `comments`, but we haven't added that element to the `show` template yet.

Add this element to the `app/views/tickets/show.html.erb` template, by adding the following code above the spot where you render the `comment` form partial.

Listing 250. app/views/tickets/show.html.erb

```
<div class="comments">
  <header>
    <h3>Comments</h3>
  </header>

  <% comments = @ticket.comments.persisted.ordered %>
  <% if comments.any? %>
    <%= render comments %>
  <% else %>
    <p>There are no comments for this ticket.</p>
  <% end %>

  <%= render "comments/form", ticket: @ticket, comment: @comment %>
</div>
```

Here you create the element that the scenario requires: one with an `id` attribute of `comments`.

We've also introduced two new methods here, called `persisted` and `ordered`, on our relation of comments. The reason we need to do this is because we've added an unpersisted (not saved in the database) comment to the list, when we did `@ticket.comments.new`, to render with the comment form. We don't want to display that new comment in our list of all the posted comments, so we'll filter it out.

The `ordered` scope is so that we'll get comments back in the order that they were created. If we didn't do this, Active Record will return comments in any order.

To do this, we'll define these two new scopes in our `Comment` model. We first looked at scopes in chapter 8 when we look at filtering out archived users, we'll write another here to filter out unpersisted comments.

Inside the `Comment` model in `app/models/comment.rb`, add the following scope:

Listing 251. Defining a `persisted` scope

```
class Comment < ActiveRecord::Base
  ...
  scope :persisted, -> { where.not(id: nil) }
  scope :ordered, -> { order(created_at: :asc) }
  ...
```

It's another very simple scope, that simply selects all comments where the `id` field is not `nil`. Any persisted object will have a numerical ID, so this filters out non-persisted ones. Easy.

You also check if there are no comments by using the `any?` method from Active Record, and displaying an appropriate "no comments" message if this is the case. This will do a light query, similar to the following, to check if there are any comments:

```
SELECT 1 AS one FROM "comments" WHERE "comments"."ticket_id" = 1 LIMIT 1
```

If there are any records in the `comments` table with a `ticket_id` of "1", this just returns the number 1 to ActiveRecord. It also limits the result set to 1, which will stop looking after it finds the first comment of the ticket, resulting in a super-fast query. You also used `any?` back in chapter 9, when you checked if a ticket had any attachments.

You could use `empty?` here instead, but that would load the `comments` association in its entirety and then check to see if the array was empty. If there were a lot of comments, this approach would be slow. By using `any?`, you stop this potential performance issue from

cropping up.

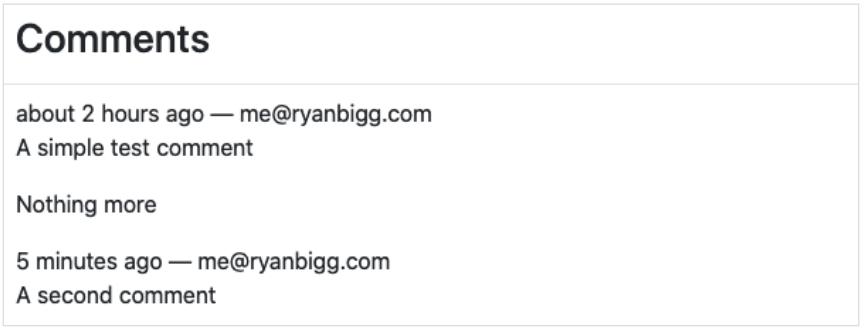
Inside this `div`, if there are comments, you call render and pass it the argument of `@ticket.comments.persisted`. The use of `render` in this form will cause Rails to render a partial for every single element in this collection, and to try to locate the partial using the first object's class name.

Objects in this particular collection are of the `Comment` class, so the partial Rails will try to find will be at `app/views/comments/_comment.html.erb`. This file doesn't exist yet, but you can create it and fill it with this content:

Listing 252. app/views/comments/_comment.html.erb

```
<div class="comment">
  <div class="author">
    <%= time_ago_in_words(comment.created_at) %> ago
    &mdash; <%= comment.author %>
  </div>
  <div class="text">
    <%= simple_format(comment.text) %>
  </div>
</div>
```

Here's what comments look like now:



But we can improve the look of this with some styling. It's not very clear at the moment where one comment finishes and another begins. Let's open up `app/assets/stylesheets/comments.scss` and add in this styling:

Listing 253. app/assets/stylesheets/comments.scss

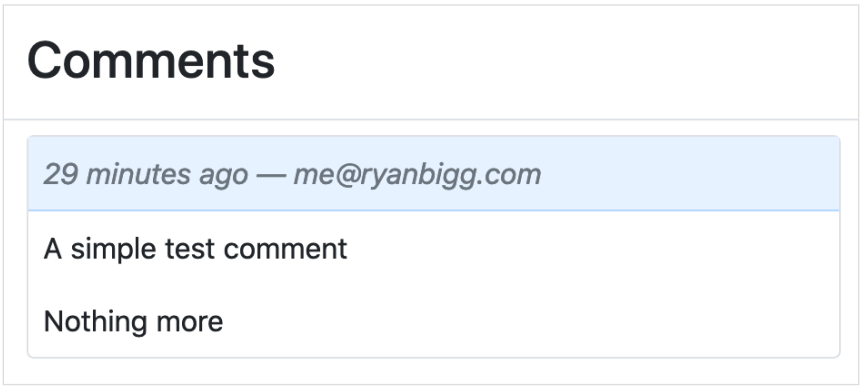
```
@import "bootstrap";

.comment {
  @extend .border, .rounded;
  border-width: 2px;

  .author {
    @extend .text-secondary, .font-italic, .p-2;

    background: lighten($blue, 45);
    border-bottom: 1px solid lighten($blue, 35);
  }

  .text {
    @extend .p-2;
  }

  p:last-of-type {
    margin-bottom: 0;
  }
}
```

We'll need to import this file into `application.scss` too:

Listing 254. app/assets/stylesheets/application.scss

```
@import "bootstrap";
@import "font-awesome";
@import "rails_bootstrap_forms";
@import "comments";
```

This styling will change our comment boxes into this by using Bootstrap's utility classes and colors:



Much better.

With the code in place not only to create comments but also display them, your feature should pass when you run it with `bundle exec rspec spec/features/creating_comments_spec.rb`:

```
2 examples, 0 failures
```

With this form added to the ticket's page, users are now able to leave comments on tickets.

Let's run all of our tests now to assert everything is working.

```
42 examples, 0 failures, 2 pending
```

The two pending tests in this output are from `spec/helpers/comments_helper_spec.rb` and `spec/models/comment_spec.rb`. You can go ahead and delete those two files, as they don't contain any useful tests. If you rerun `rspec`, you'll see this:

```
40 examples, 0 failures
```

Good stuff! Now commit and push this:

```
$ git add .
$ git commit -m "Users can now leave comments on tickets"
$ git push
```

This feature of your application is useful because it provides a way for users of a project to have a discussion about a ticket and keep track of it. Next up, you'll add a way to provide

additional context to this ticket by changing states.

10.2. Changing a ticket's state

States help standardize the way a ticket's progress is tracked. By glancing at the state of a ticket, a user can determine if that ticket needs more work or if it's complete, as shown below:

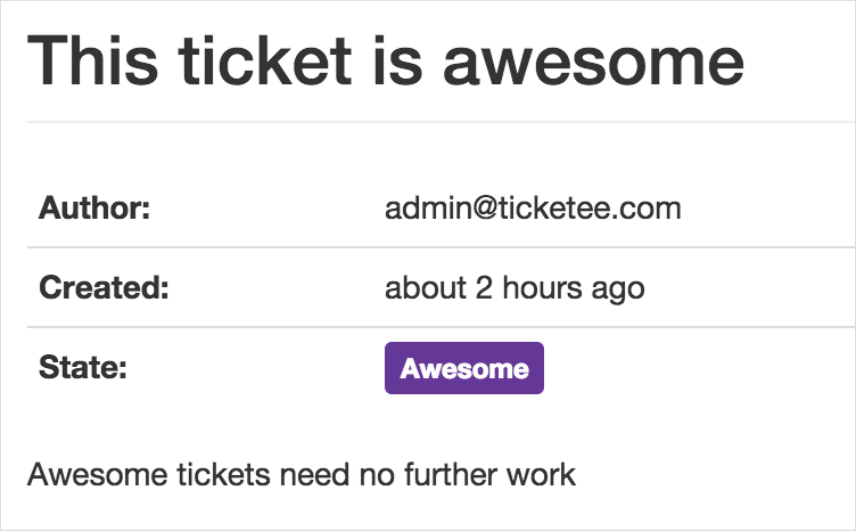


Figure 44. A ticket's state

To allow users to change a ticket's state, you'll add a select box on the comment form from which a state can be selected. These states will be stored in another table called `states`, and they'll be accessed through a `State` model. Later on, you'll give some users the ability to add states for the select box, and make one of them the default. First, though, you need to create the select box so that states can be selected. As usual, you'll start creating a comment that changes a ticket's state by writing another scenario. This scenario will go at the bottom of `spec/features/creating_comments_spec.rb` and it's shown in the following listing.

Listing 255. spec/features/creating_comments_spec.rb

```
RSpec.feature "Users can comment on tickets" do
  ...
  scenario "when changing a ticket's state" do
    visit project_ticket_path(project, ticket)

    within(".comments") do
      fill_in "Text", with: "This is a real issue"
      select "Open", from: "State"
      click_button "Create Comment"
    end

    expect(page).to have_content "Comment has been created."
    within(".ticket .attributes .state") do
      expect(page).to have_content "Open"
    end
  end
end
```

In this scenario, you go through the process of creating a comment, much like in the previous scenario in this file; only this time you select a state. This is the first part of the scenario, and you can expect it to fail because you don't have a state select box yet. After the comment is created, you should see the state appearing in the `.ticket .attributes .state` area. This is the second part of the scenario that will fail. When you run this scenario by running `bundle exec rspec spec/features/creating_comments_spec.rb`, it will fail like this:

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: select "Open", from: "State"
   Capybara::ElementNotFound:
     Unable to find select box "State"
```

As you can see from this output, the line that attempts to select "Open" from "State" can't find the select box because you haven't added it yet. With this select box, users of your application should be able to change the ticket's state by selecting a value from it, entering some comment text, and clicking the "Create Comment" button. Before you do all that, however, you need to create the `State` model and its related table, which is used to store the states.

10.2.1. Creating the State model

Right now you need to add a select box. When you're done, you'll have this:

The states in the select box will be populated from the database, because you want users to eventually be able to create their own states. For now, you'll define this `State` model to have a `name` field, as well as a `color` [64] field which will define the colors of the label for each state. (Later on, you'll add a `position` field, which you'll use to determine the sort order of the states in the select box on the comment form.) Create this `State` model and its associated migration by running this command:

```
$ rails g model state name:string color:string
```

But don't run the migration just yet! Before running the migration you just created, you'll need to define a way for states to link to comments and to tickets, but there are a couple of things worth mentioning beforehand. For comments, you want to track that a comment has changed the ticket's state, and for tickets, you want to track the current state of the ticket. You'll use references on the `Ticket` and `Comment` models for this, and, as a bonus, you can add these fields to the migration. You can also remove the `timestamps` call from within `create_table`, as it's not important when states were created or updated.

When you're done, the whole migration should look like the following listing:

Listing 256. db/migrate/[date]_create_states.rb

```
class CreateStates < ActiveRecord::Migration[6.1]
  def change
    create_table :states do |t|
      t.string :name
      t.string :color
    end

    add_reference :tickets,  :state, index: true, foreign_key: true
    add_reference :comments, :state, foreign_key: true
  end
end
```

In this migration, you use the `index: true` option on the reference to add a database index on the tickets table's `state_id` field. By adding an index on this field, you can speed up queries that search for tickets that have a particular value in this field.

The downsides of indexing is that it will result in slower writes and use more disk space, but the benefits far outweigh these. It's always important to have indexes on non-primary-key fields[65] that you do lookups on because of this great read-speed increase, as applications generally read from the database more often than write to it. Run this migration now by running this command:

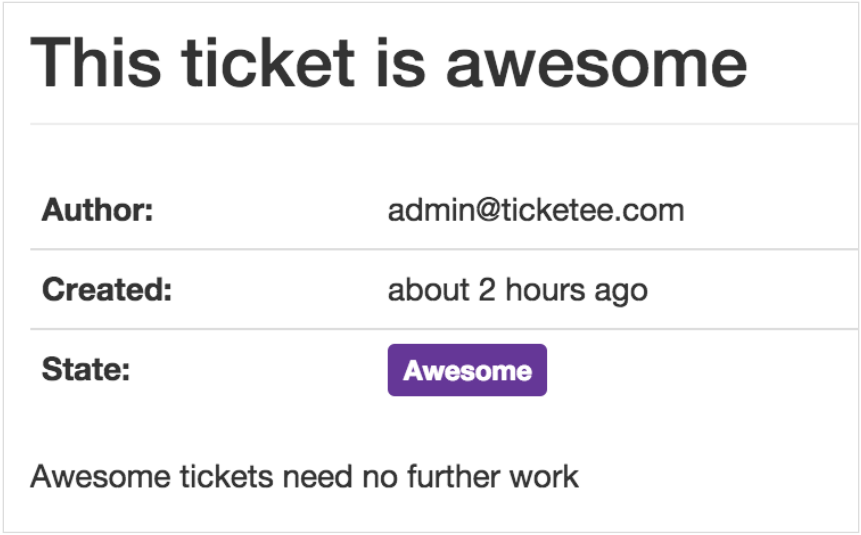
```
$ rails db:migrate
```

There you have it! The `State` model is up and running. You can now associate this class with the `Comment` class by adding this line to the top of the `Comment` model's definition:

```
class Comment < ActiveRecord::Base
  belongs_to :state, optional: true
  ...
```

We're using the `optional` option here, as not every comment will have a related state. Only those comments that change the state will have a state specified.

The `state` method provided by this `belongs_to` will be used shortly to display the state on the ticket page:



Before doing that, however, you'll need to add the select box for the state to the comment

form.

10.2.2. Selecting states

In your comment form partial, you can add the select box underneath the text box, as shown in the following listing.

Listing 257. app/views/comments/_form.html.erb

```
<%= bootstrap_form_with model: [ticket, comment], local: true do |f| %>
  <%= f.text_area :text %>
  <%= f.select :state_id, @states.map { |state| [state.name, state.id] } %>
  <%= f.primary %>
<% end %>
```

This `f.select` will provide that dropdown box, but we'll need to also set the `@states` variable before this will work. Let's set this now in the `show` action for `TicketsController`:

Listing 258. app/controllers/tickets_controller.rb

```
def show
  @comment = @ticket.comments.build
  @states = State.all
end
```

We will also need to add this to the `CommentsController's `create` action inside the `else` where we render the `"tickets/show"` page:

Listing 259. app/controllers/comments_controller.rb

```
def create
  @comment = @ticket.comments.build(comment_params)
  @comment.author = current_user

  if @comment.save
    flash[:notice] = "Comment has been created."
    redirect_to [@ticket.project, @ticket]
  else
    flash.now[:alert] = "Comment has not been created."
    @states = State.all
    @project = @ticket.project
    render "tickets/show"
  end
end
```

If we don't make this change, when we submit a blank comment the `"tickets/show"` page will not be able to render, as the `@states` variable will not be set.

For a ticket that has its state set to "New", the select box generated by `f.select :state_id` could look like this:

```
<select class="select optional form-control" name="comment[state_id]"
  id="comment_state_id">
  <option value="1" selected="selected">New</option>
  <option value="2">Open</option>
  <option value="3">Closed</option>
</select>
```

The first `option` tag in the `select` tag has an additional attribute: `selected`; when this attribute is set, that option will be the one selected as the default for the select.

Which `option` tag gets the `selected` attribute is determined by the `:selected` option for `f.select`. The value for this option is the corresponding value attribute for the `option` tag. For a bit more explanation, if this `comment` object had a `state_id` of 2, and the state with the ID 2 had the name "Rejected", then "Rejected" would automatically be selected in the `state` select box for the comment. Nifty!

With the select box in place, you're almost at the point where this scenario will pass. You can see how far you've gotten by running `bundle exec rspec spec/features/creating_comments_spec.rb`.

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: select "Open", from: "State"
   Capybara::ElementNotFound:
     Unable to find option "Open"
```

The "State" field is rendering correctly, but it's empty - there's no "Open" option to select. We don't have any states in the database, so we have to add one.

To do this, we'll add a line in our "changing state" scenario, to create a new `State` object:

Listing 260. spec/features/creating_comments_spec.rb

```
RSpec.feature "Users can comment on tickets" do
  ...

  scenario "when changing a ticket's state" do
    FactoryBot.create(:state, name: "Open")
    ...
```

For this to work, you'll need to define a `state` factory. Go ahead and do that in a new file called `spec/factories/states.rb` using the content from the following listing:

Listing 261. spec/factories/states.rb

```
FactoryBot.define do
  factory :state do
    name { "A state" }
  end
end
```

Now that the `state` factory is defined, when you rerun `bundle exec rspec spec/features/creating_comments_spec.rb`, the final scenario will fail with this error:

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: within(".ticket .attributes .state") do
   Capybara::ElementNotFound:
     Unable to find css ".ticket .attributes .state"
```

This output means it's looking for any element with the `id` attribute of `ticket` that contains any type of element with the `class` of `state`, but it can't find it. This should be easy to fix, we just need to display the state we're saving, on the page.

Rather than putting the state inside the `show` view of `TicketsController`, let's try putting it in a partial. This will allow you to reuse this code, to display a state wherever you need it in the future. Additionally, you can apply a dynamic class around the state so you can style it later.

Create the new partial at `app/views/states/_state.html.erb` and fill it with this content:

Listing 262. app/views/states/_state.html.erb

```
<span class="state state-<%= state.name.parameterize %>">
  <%= state -%>
</span>
```

To style the element, you need a valid CSS class name, which you can get by using the `parameterize` method. If, for example, you had a state called "Drop bears strike without warning!" and you used `parameterize` on it, all the spaces and characters that aren't valid in URLs would be stripped, leaving you with "drop-bears-strike-without-warning", which is a perfectly valid CSS class name. You'll use this generated class name later on, to style the state using the `color` attributes of the state.

You're now going to render this partial as part of the table displaying a ticket's attributes. Add another row at the bottom of the table in `app/views/tickets/show.html.erb`, using the following line:

Listing 263. Displaying the current state of a ticket

```
<table class='attributes'>
  ...

  <% if @ticket.state.present? %>
    <tr>
      <th>State:</th>
      <td><%= render @ticket.state %></td>
    </tr>
  <% end %>
</table>
```

You're using the short form of rendering a partial here again, and you conditionally render it if the ticket has a state. If you didn't have the `if` around the state, and the state was `nil`, this would raise an exception - the partial would try to call `nil.name.parameterize`, when generating a class name for the state.

Now that we have the name of the state on the page, let's re-run the spec. Have we missed anything?

```
1) Users can comment on tickets with valid attributes
   Failure/Error: visit project_ticket_path(project, ticket)
   ActionView::Template::Error:
     undefined method `state' for #<Ticket:0x007fa9c1dc6cd0>
```

All three of the specs are failing! Huh? Didn't we add a `state` association to the `Ticket` model? We added the right `state_id` field to the `tickets` table of the database, when we created the `states` table... but we didn't add the `state` association to the `Ticket` model.

Only comments have a state at the moment, not tickets. So you'll need to add the association between `Ticket` and `State`, in your `Ticket` model. This method should go directly below the `belongs_to :author`, line in `app/models/ticket.rb`:

Listing 264. A ticket has a state too!

```
class Ticket < ActiveRecord::Base
  ...
  belongs_to :state, optional: true
  ...
```

We're using `optional` again here, because tickets won't have states by default.

If you run the feature again with `bundle exec rspec spec/features/creating_comments_spec.rb`, it will still fail. Aww.

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: within(".ticket .attributes .state") do
   Capybara::ElementNotFound:
     Unable to find css ".ticket .attributes .state"
```

All these failures are so tiring, but at least we're running out of things that can go wrong as we fix them one at a time! The `.ticket .attributes .state` element still isn't displaying on the ticket's `show` view, which only happens if `@ticket.state` isn't present, ie. it's `nil`.

In our controller we're creating a new `Comment` object, and we think we're rightly saving the `state` we select, inside it. But in the view, we're displaying the `state` of the ticket object, not the comment we just created! How can we get the `state` saved on the ticket instance as well?

We can do this with a callback, defined as part of our `Comment` model, to set the ticket's status when a user changes it through the comment form.

A callback is a method that's automatically called either before or after a certain event occurs. We've seen and used controller callbacks before, when we wrote `before_action` and `after_action` methods. For models, there are before and after callbacks defined for the following events (where * represents either `before` or `after`):

- Validating (`*_validation`)
- Creating (`*_create`)
- Updating (`*_update`)
- Saving (`*_save`)
- Destroying (`*_destroy`)

You can trigger a specific piece of code or method to run before or after any of these events.

If we define a callback that occurs after a comment is created, we can then copy the state from the comment to the ticket itself. You can use the `after_create` method in your `Comment` model to do this.

Listing 265. app/models/comment.rb

```
class Comment < ActiveRecord::Base
  ...
  after_create :set_ticket_state
end
```

The symbol passed to the `after_create` method in listing 10.19 is the name of the method this callback will call. You can define this method at the bottom of your `Comment` model using the code from the following listing.

Listing 266. app/models/comment.rb with the callback fully defined

```
class Comment < ActiveRecord::Base
  ...
  after_create :set_ticket_state

  private

  def set_ticket_state
    ticket.state = state
    ticket.save!
  end
end
```

With this callback and associated method now in place, the associated ticket's state will be set to the comment's state after the comment is created. Great! But when you run your feature again by running `bundle exec rspec spec/features/creating_comments_spec.rb`, it still fails:

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: within(".ticket .attributes .state") do
   Capybara::ElementNotFound:
     Unable to find css ".ticket .attributes .state"
```

Wait, the same error? Even though we're displaying the ticket state on the page (if the ticket has one), we're not seeing it displayed? So the ticket doesn't have a state… what did we miss? This shouldn't be difficult to figure out, but you're probably sitting there thinking someone somewhere has made a terrible mistake.

The subtle bug is one that's caught us a couple of times in the past - we added a `state` field to our form, but we forgot to add it to the list of parameters we're permitting and whitelisting in our controller. That's really annoying. Luckily, Rails lets us configure how we want our app to respond when we have unpermitted parameters. By default, we'll get a little message in our log files - it looks like the following:

```
...
Started POST "/tickets/1/comments" for ::1 at [timestamp]
Processing by CommentsController#create as HTML
  Parameters: {"utf8"=>" ", "authenticity_token"=>"VEsOog...",
  "comment"=>{"text"=>"This is a test", "state_id"=>"2"}...
  Ticket Load (0.1ms)  SELECT  "tickets".* FROM "tickets" WHERE ...
Unpermitted parameter: state_id
  User Load (0.4ms)  SELECT  "users".* FROM "users" WHERE "users"....
   (0.1ms)  begin transaction
  SQL (0.4ms)  INSERT INTO "comments" ("text", "ticket_id", "author_...
  ...
```

It's really easy to miss, even if you're looking for it. We can make it much, much more obvious by changing a configuration option for our test environment. Open up the environment file that controls your `test` environment, in `config/environments/test.rb`, and add the following line at the bottom of the block:

Listing 267. Configuring the behaviour on unpermitted params

```
Rails.application.configure do
  ...

  config.action_controller.action_on_unpermitted_parameters = :raise
end
```

Rails will now throw a big nasty exception if we pass unpermitted params to our `params.permit` method, which we're doing right now. You can see what this looks like by re-running the "Creating comments" spec, with `bundle exec rspec spec/features/creating_comments_spec.rb`:

```
1) Users can comment on tickets with valid attributes
   Failure/Error: click_button "Create Comment"
   ActionController::UnpermittedParameters:
     found unpermitted parameter: state_id
```

Ka-booooom!

This sounds like a good setting to leave on in test mode, so leave that setting in your `test.rb`. Now we know what we need to do - permit the `state_id` parameter. Inside your `CommentsController` in `app/controllers/comments_controller.rb`, permit the `state_id` parameter in your `comment_params` method so it looks like this:

Listing 268. Now with bonus `state_id`

```
class CommentsController < ApplicationController
  ...
  def comment_params
    params.require(:comment).permit(:text, :state_id)
  end
end
```

That will clear up that unpermitted parameter error. It might clear up more errors too, let's re-run the spec again and see.

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: expect(page).to have_content "Open"
     expected to find text "Open" in "#<State:0x007feddd3b3670>"
```

We're now showing something, that looks like it might be a State, with a lot of added junk. By default, objects in Ruby have a `to_s` method that will output this ugly, inspected version of this object. By overriding this method in the model to call the `name` method, you can get it to display the state's name rather than its object output. We did this earlier on with the `User` class as well.

Open up your `State` model in `app/models/state.rb` and define a new `to_s` model that looks like this:

```
class State < ApplicationRecord
  ...
  def to_s
    name
  end
end
```

Great! This should mean that the last scenario in your "Creating comments" feature will pass. Run it with `bundle exec rspec spec/features/creating_comments_spec.rb` and find out.

```
3 examples, 0 failures
```

It's passing! This is a good time to ensure that everything is working by running `bundle exec rspec`.

```
42 examples, 0 failures, 1 pending
```

The one pending spec that's cramping your style is located in `spec/models/state_spec.rb`. You can delete this file, as it doesn't contain any useful specs.

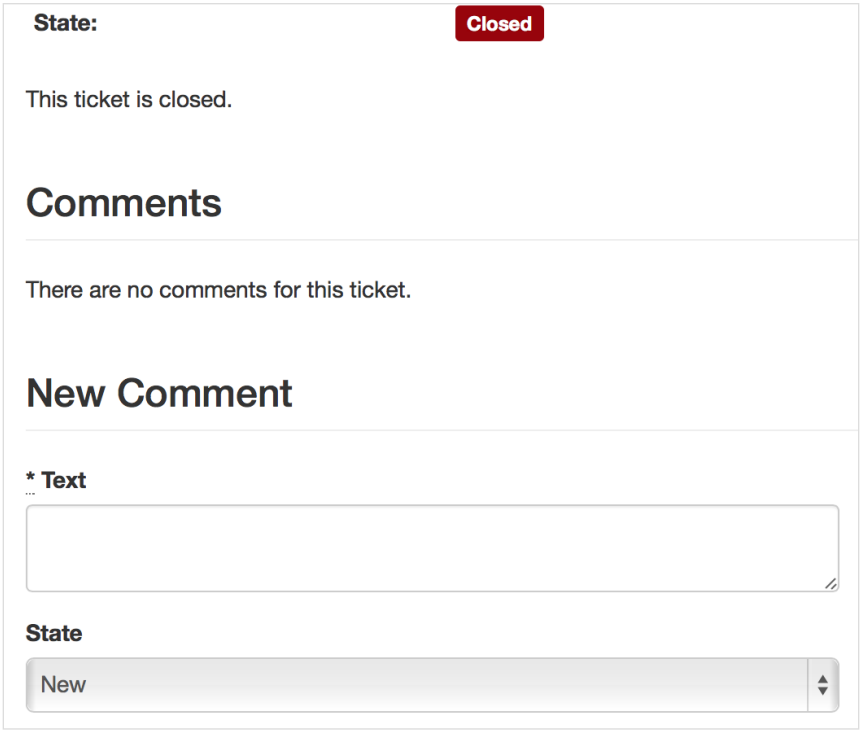
When you rerun `bundle exec rspec`, you'll see it's now lovely and green:

```
41 examples, 0 failures
```

Excellent, everything's fixed. Commit these changes now:

```
$ git add .
$ git commit -m "Tickets can have a state assigned when comments are
  created"
$ git push
```

We can now change a ticket's state by adding a new comment with a state, but then if we load the comment form again, the state form will be a little misleading:



We've marked the ticket as "Closed", but if we add a new comment just by typing some text and submitting it, then the state will be reset to "New"! It would be great if the default value

of the state field was the current state of the ticket, to prevent accidental changing of state. We'll look at fixing that next.

10.2.3. Setting a default state for a comment

At the moment, we're not setting which state should be set as the default value of the select box. We only have the following code in our comment form:

```
<%= f.select :state_id, @states.map { |state| [state.name, state.id] } %>
```

The form will automagically set the selected value of the state box to be whatever the `state_id` property of the comment is. By default, the `state_id` of a comment is `nil`, which we can verify by instantiating a new `Comment` instance in a Rails console:

Listing 269. Testing out code in `rails console`

```
irb(main):001:0> Comment.new
=> #<Comment id: nil, text: nil, ticket_id: nil, author_id: nil,
  created_at: nil, updated_at: nil, state_id: nil>
```

We can fix this by setting the `state_id` of the newly-built `Comment` object, when we build it in the `show` action of `TicketsController`. Open up the controller in `app/controllers/tickets_controller.rb`, and change the `show` action to set the `state_id`:

Listing 270. Setting the `state_id` of the comment on the "New Comment" form

```
def show
  @comment = @ticket.comments.build(state: @ticket.state)
end
```

This will change the default value that gets selected in the "New Comment" form, to be whatever the current state of the `@ticket` is. Great!

You've now got the ticket status updating along with the comment status. Let's add some seed states to our application, so we can see this functionality working in our browsers.

10.2.4. Seeding states

Seeds can be added to `db/seeds.rb`. Let's add some to the bottom of the file like so:

```
unless State.exists?
  State.create(name: "New", color: "#0066CC")
  State.create(name: "Open", color: "#008000")
  State.create(name: "Closed", color: "#990000")
  State.create(name: "Awesome", color: "#663399")
end
```

This will load four states into our system - "New", "Open", "Closed", and "Awesome"[66]. Load these seed states into your database by running the following:

```
$ rails db:reset
```

Now when you load up your Ticketee application in your browser, you'll be able to set states on your comments, making working with the next section a lot easier.

Now that we have statuses in the system, it would be handy to know what the timeline of status changes looks like. You can display this on the comment by showing a little indication of whether the state has changed during that comment. Let's work on adding this little tidbit of information to the comments right now.

10.3. Tracking changes

When a person posts a comment that changes the state of a ticket, you'd like this information displayed on the page next to the comment:

Figure 45. State transitions

By visually tracking this state change, along with the text of the comment, you can provide context as to why the state was changed. At the moment, you only track the state of the comment and then don't even display it alongside the comment's text; you only use it to update the ticket's status.

10.3.1. Ch-ch-changes

What you'll need now is some way of making sure that, when changing a ticket's state by way

of a comment, a record of that change appears in the comments area. A scenario would fit this bill and lucky for us you wrote one that fits almost perfectly. This scenario would be the final scenario ("Changing a ticket's state") in `spec/features/creating_comments_spec.rb`.

To check for the state-change text in your "Changing a ticket's state" scenario, you can add these lines to the bottom of the scenario:

```
scenario "when changing a ticket's state" do
  ...
  within(".comments") do
    expect(page).to have_content "state set to Open"
  end
end
```

If the ticket was assigned the "New" state, this text would say "state changed from New to Open", but because your tickets don't have default states assigned to them yet, the previous state for the first comment will be `nil`. When you run this scenario using `bundle exec rspec spec/features/creating_comments_spec.rb`, it will fail.

```
1) Users can comment on tickets when changing a ticket's state
   Failure/Error: expect(page).to have_content "state set to Open"
     expected to find text "state set to Open" in "This is a real
     issue less than a minute ago by test3@example.com (User)"
```

Good. You've got a way to test this state message that should appear when a comment changes the state of the ticket. Here's what these state transitions will look like:

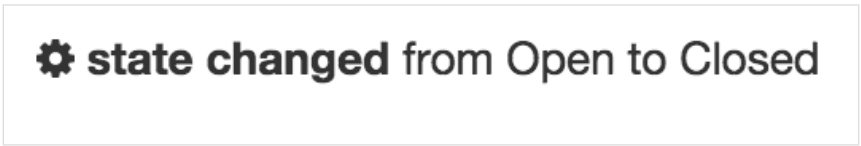


Figure 46. A state transition

With this little bit of information, users can see what comments changed the ticket's state, which is helpful for determining what steps the ticket has gone through to wind up at this point.

10.3.2. Displaying changes

When you display a comment that changes a ticket's state, you want to display this state

transition along with the comment. To get this text to show up, we'll change the comments area of `app/views/tickets/show.html.erb` to this:

Listing 271. app/views/tickets/show.html.erb

```
<div class="comments">
  <header>
    <h3>Comments</h3>
  </header>

  <% comments = @ticket.comments.persisted.ordered %>
  <% if comments.any? %>
    <% state = nil %>
    <% comments.each do |comment| %>
      <%= render comment %>
      <% if comment.state != state %>
        <%= render "state_change", previous_state: state, new_state: comment.state %>
        <% state = comment.state %>
      <% end %>
    <% end %>

  <% else %>
    <p>There are no comments for this ticket.</p>
  <% end %>

  <%= render "comments/form", ticket: @ticket, comment: @comment %>
</div>
```

Instead of using `<%= render comments %>` to loop through all the comments, we're now using an `each` instead. This `each` block will allow us to check the state of each comment, comparing it to the comment's state before it. If the state is different, then a new partial called `state_change` will be rendered, along with the comment itself.

Let's create this `state_change` partial too:

Listing 272. app/views/tickets/_state_change.html.erb

```
<div class="state_change">
  <% if previous_state %>
    <strong><i class="fa fa-gear"></i> state changed</strong> from
    <%= render previous_state %> to <%= render new_state %>
  <% else %>
    <strong><i class="fa fa-gear"></i> state set</strong> to <%= render new_state %>
  <% end %>
</div>
```

In this file, we need to take into account the fact where `previous_state` might be `nil`. The `state` variable back in `app/views/tickets/show.html.erb` is set to `nil` by default, because tickets do not have a state by default. If this is `nil` by the time it gets to the `state_change` partial, then we will say "State set to [new state]". If it's not nil, then we will say "State changed from [previous state] to [new state]"

We've used the idea of calling `render` on an object before, knowing that Rails will look up a partial based on the class of the object. We're passing a `State` object here, in both `previous_state` and `new_state`, so Rails will load the `app/views/state/_state.html.erb` partial we defined earlier, because both of these objects are `State` objects.

Go ahead now and add some comments to a ticket that make some state changes. And add some others that do not change the state at all. As you're adding these, you will probably see that things look a little squished:

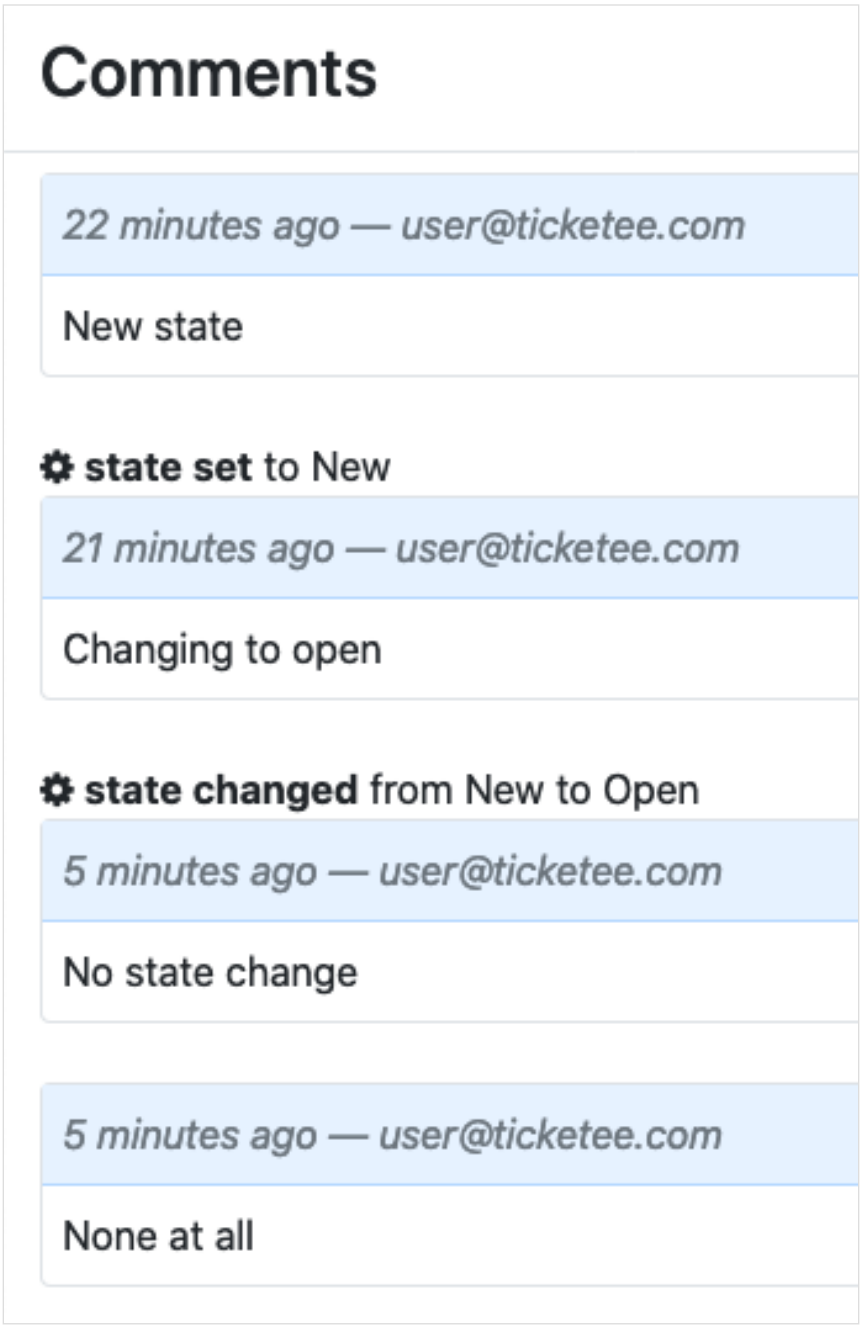


Figure 47. Squished states

The states are very close to the comments, and it's hard to see here what comment changed the state.

Let's fix this by changing the margin around comments and state changes. We can do this by first removing the bottom margin in `comments.scss`. Change this line in that file:

Listing 273. app/assets/stylesheets/comments.scss

```
.comment {
  @extend .border, .rounded;
  ...
}
```

To this:

Listing 274. app/assets/stylesheets/comments.scss

```
.comment {
  @extend .border, .rounded, .mb-2;
  ...
}
```

This will squish everything together even more!

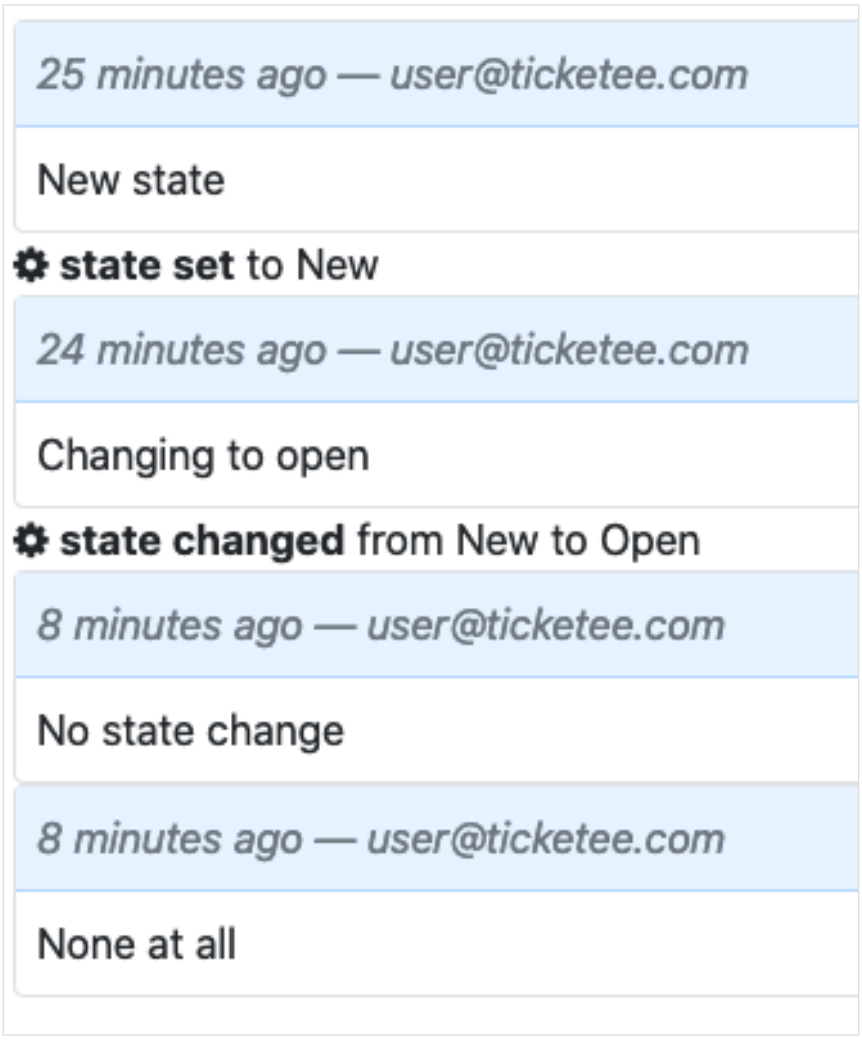


Figure 48. Squished comments & states

What we'd like to do here is to provide some spacing between state changes and comments. Let's add a new class for state changes to this CSS file:

Listing 275. app/assets/stylesheets/comments.scss

```
.state_change {
  @extend .my-2;
}
```

This will add a two-unit spacing above + below state changes. It will turn our list of comments into this:

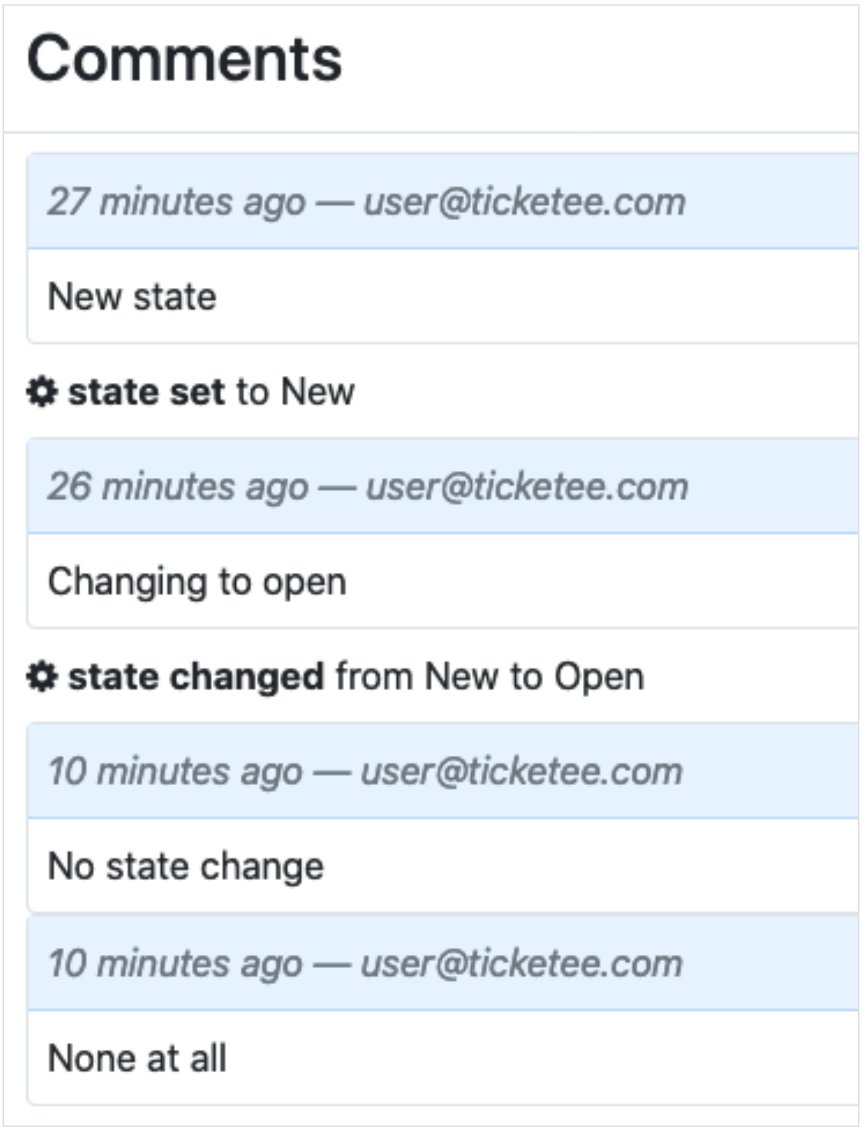


Figure 49. Not-so squished comments & states

Finally, we need to space out those comments. We need to make it so that two consecutive comments have a large gap in between them. We can do this with the + CSS pseudo-selector:

Listing 276. app/assets/stylesheets/comments.scss

```
.comment {
  ...

  + .comment {
    @extend .mt-4;
  }
}
```

This selector applies for consecutive comment elements only. Comment elements that are separated by a state change will not have this style applied.

This final change will leave our state changes and comments looking swell:

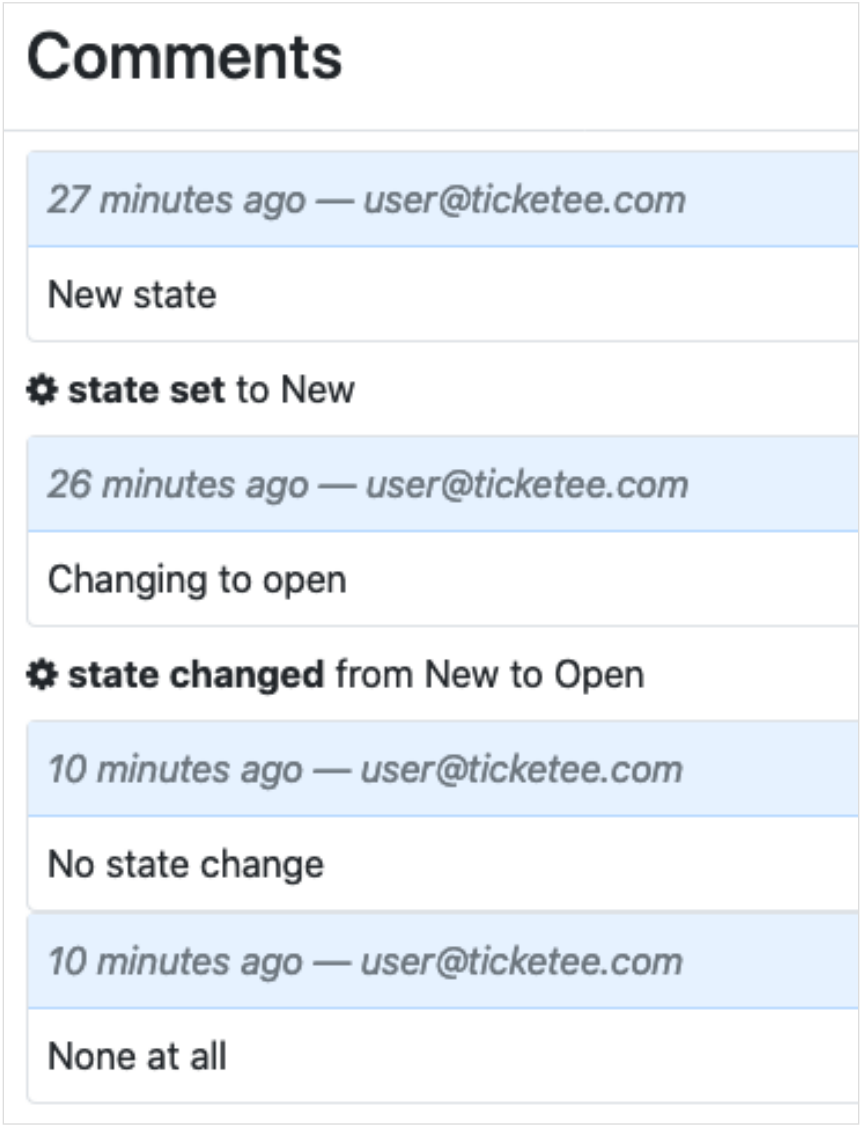


Figure 50. Great comments and state changes

Next, you can check to see if this is working by running your scenario using `bundle exec rspec spec/features/creating_comments_spec.rb`. It will pass:

```
3 examples, 0 failures
```

Excellent! You've got your application showing users what state a comment has switched the

ticket to. This is a good time to check that you haven't broken anything. When you run `bundle exec rspec`, you should see that everything is A-OK.

```
41 examples, 0 failures
```

You have state transition showing in your application, which is great to see. Commit and push this to GitHub.

```
$ git add .
$ git commit -m "Display a comment's state transition"
$ git push
```

Looking good, right? Wrong. Time to add some styles to our states.

10.3.3. Styling states

Ticketee's state styles could use a little work - at the moment you can't distinguish one state from another. Look below and gaze upon their ugliness.

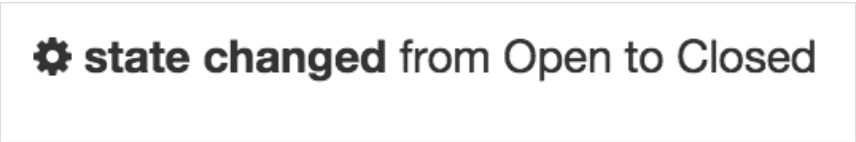


Figure 51. Ugly, ugly states

You could distinguish them by using the color you've specified in the attributes. Earlier, you wrapped the state name in a special `span` that will allow you to style these elements based on the class. For the "New" state, the HTML for the `span` looks like this:

```
<span class="state state-new">
  New
</span>
```

You can use the `state` class to add generic styles that will apply to all states, and the `state-new` class to apply the colors from that specific `State` record to this element. To do so, you'll need to dynamically define some CSS that will apply the colors.

The states in your system can change at anytime in the future, so you can't just put styles in `app/assets/stylesheets/application.css` for them. To get around this, we'll do two things:

- Add generic styles for the `state` class in `app/assets/stylesheets/tickets.scss`
- Add specific styles for our individual states in a `<style>` block in your application layout, so we can use the styles for states in every page.

"But wait," we hear you crying, "why can't we just make a new stylesheet like `states.css.erb`, load the states in it, and generate a stylesheet like that? Dirtying up our application layout is soooo ugly!"

Well it is ugly, you're right, but without a lot of Sprockets hackery, we don't really have a choice. We could create a stylesheet called something like `states.css.erb`, and fill it with the following code:

```
<% State.all.each do |state| %>
  .state-<%= state.name.parameterize %> {
    background-color: <%= state.color %>;
  }
<% end %>
```

If we included this file into our `application.css.scss`, it would work great... in development. We could even change the `State` instances in our database, or add new ones, the stylesheet would even be updated with the updated `State` styles.

However, when it comes to deployment, we'll be using Sprockets to compile stylesheets just once, at deploy time, so any changes we made to states after deployment would not be reflected on our site. That would be awful!

So while putting styles in our HTML is ugly, for now it's a necessary evil. We'll talk more about pre-compilation of stylesheets and other assets when we cover deployment in chapter 13.

The first step is easy - adding styles for the base `state` class. We can add these to `app/assets/stylesheets/tickets.scss` now:

Listing 277. Basic styles for our displayed states

```
@import "bootstrap";

.state {
  @extend .rounded, .px-2, .py-1;
  display: inline-block;
  font-size: 110%;
  color: white;
}
```

And then we will need to make sure these styles are added to our `application.css` when it's compiled. We do this by importing this `tickets.scss` file into `app/assets/stylesheets/application.css.scss` with this line:

Listing 278. app/assets/stylesheets/application.css.scss

```
@import "tickets";
```

The second step is a bit trickier - we'll have to iterate over our styles in our layout file in `app/views/layouts/application.html.erb`, and print out some CSS to apply to our individual styles. Add them at the bottom of the `<head>` tag, like in the following listing:

Listing 279. app/views/layouts/application.html.erb

```
<head>
  ...
  <%= javascript_include_tag 'application' %>
  <%= csrf_meta_tags %>
  <style>
    <% State.all.each do |state| %>
      .state-<%= state.name.parameterize %> {
        background-color: <%= state.color %>;
      }
    <% end %>
  </style>
</head>
```

With these few lines of code, your states should now be styled. If you visit a ticket page that has comments that have changed the state, you should see the state styled, as shown here:

Figure 52. States, now with 100% more style

While you're in the business of prettying things up, you can also add the state of your ticket to the listing on `app/views/projects/show.html.erb` so that users can easily glance at the list of tickets and see a state next to each of them. Add this to the left of the ticket name, so that the li element is as follows:

Listing 280. Displaying the state of tickets on the project page

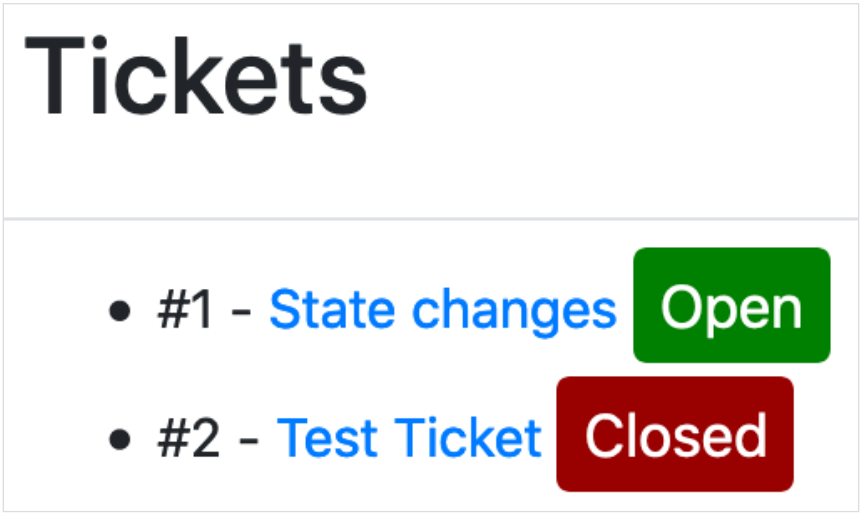
```
<li>
  #<%= ticket.id %> -
  <%= link_to ticket.name, [@project, ticket] %>
  <%= render ticket.state if ticket.state %>
</li>
```

We'll also add an extra margin to stop the ticket states pressing right up against each other, in `app/assets/stylesheets/tickets.scss`:

Listing 281. Separating out the ticket listing

```
.tickets li {
  @extend .mb-1;
}
```

That's looking a lot better!



You've completed all that you need to do to let users change the state of a ticket. They can select a state from the state select box on the comment form, and when they create a comment, that ticket will be updated to the new state. Right after the comment's text on the ticket page, the state transition is shown and (ideally) the comment's text will provide context for that change.

Commit this before we go any further:

```
$ git add .
$ git commit -m "Apply colour to states"
$ git push
```

Why did you add states in the first place? Because they provide a great way of standardizing the lifecycle of a ticket. When a ticket is assigned a "New" state, it means that the ticket is up for grabs. The next phase of a ticket's life is the "Open" state, which means that the ticket is being looked into or cared for by somebody. When the ticket is fixed, it should be marked as "Closed", perhaps with some information in the related comment about where the fix is located. We're not sure what you might use the "Awesome" state for, but we think you could use your imagination.

If you want to add more states than these four, you can't at the moment. But that would be a useful feature. Tickets could be marked as "Closed" for a few different reasons: one could be "Yes, this is now fixed" and another could be "No, I don't believe this is a problem." A third type could be "I couldn't reproduce the problem described."

It would be great if you could add more states to the application without having to add them to the state list in `db/seeds.rb`, wouldn't it? Well, that's easy enough. You can create an interface for the admin users of your application, to allow them to add additional states.

10.4. Managing states

Currently your application has only four states: "New", "Open", "Closed" and "Awesome". If you wanted to add more, you'd have to go into the console and add them there. Admins of this application should be able to add more states through the application itself, without using the console. They should also be able to rename states and delete them, but only if they don't have any tickets assigned to them. Finally, the admins should also be able to set a default state for the application, because no ticket should be without a state.

In this section, you'll start out by writing a feature to create new states, which will involve creating a new controller called `Admin::StatesController`. This controller will provide the admins of your application with the basic CRUD functionality for states, as well as the ability to mark a state as the default, which all new tickets will then be associated with.

We won't look at adding an `edit`, `update`, or `destroy` action to this controller, because we've covered that previously. You can add them yourself if you'd like some practice.

10.4.1. Adding additional states

You currently have the four default states that come from the `db/seeds.rb` file in your application. If the admin users of your application wish to add more, they can't. Not until you've created the `Admin::StatesController` controller and the `new` and `create` actions inside it. These will allow admin users to create additional states, which then can be assigned to a ticket.

But before you write any real code, you need to write a feature that describes the process of creating a state. Put it in a new file called `spec/features/admin/creating_states_spec.rb`, as shown in the following listing.

Listing 282. spec/features/admin/creating_states_spec.rb

```
require "rails_helper"

RSpec.feature "Admins can create new states for tickets" do
  before do
    login_as(FactoryBot.create(:user, :admin))
  end

  scenario "with valid details" do
    visit admin_root_path
    click_link "States"
    click_link "New State"

    fill_in "Name", with: "Won't Fix"
    fill_in "Color", with: "orange"
    click_button "Create State"

    expect(page).to have_content "State has been created."
  end
end
```

Here you sign in as an admin user and go through the motions of creating a new state. If you

run this new feature using the command `bundle exec rspec spec/features/admin/creating_states_spec.rb`, it will fail because it can't find the "States" link:

```
1) Admins can create new states for tickets with valid details
   Failure/Error: click_link "States"
   Capybara::ElementNotFound:
     Unable to find link "States"
```

The "States" link should take you to the `index` action of the `Admin::StatesController`, but it doesn't. That's because this link is missing from the admin home page, located at `app/views/admin/application/index.html.erb`. You can add this link now by adding the following line to this file, in the "Admin Links" list at the bottom:

Listing 283. Admin links with the "States" link added

```
<div class="col-md-3">
  <h5 class="text-uppercase mt-4">Admin Links</h5>
  <ul class="nav flex-column">
    <li><%= link_to "Users", admin_users_path %></li>
    <li><%= link_to "States", admin_states_path %></li>
  </ul>
</div>
```

The `admin_states_path` method won't be defined yet, but you can fix this by adding another `resources` line inside the `admin` namespace in `config/routes.rb`, like this:

Listing 284. config/routes.rb

```
namespace :admin do
  ...
  resources :states, only: [:index, :new, :create]
end
```

Remember, it's good practice to only define the routes you'll be using. We're only dealing with listing states, and creating new ones, so we only need the `index`, `new` and `create` actions.

With this line inside the `admin` namespace, the `admin_states_path` method (and its siblings) will be defined. Run the feature again with `bundle exec rspec spec/features/admin/creating_states_spec.rb` to see what you have to do next.

```
1) Admins can create new states for tickets with valid details
   Failure/Error: click_link "States"
   ActionController::RoutingError:
     uninitialized constant Admin::StatesController
```

Ah, that's right! You need to generate your controller. You can do this by running the controller generator:

```
$ rails g controller admin/states
```

This will generate an empty `Admin::StatesController`, in `app/controllers/admin/states_controller.rb` - enough to make that error go away. When you run the feature again, you'll be told that you're missing the `index` action from this controller:

```
1) Admins can create new states for tickets with valid details
   Failure/Error: click_link "States"
   AbstractController::ActionNotFound:
     The action 'index' could not be found for Admin::StatesController
```

Add this action to the `app/controllers/admin/states_controller.rb` file now, and make this controller inherit from `Admin::ApplicationController`. You know you'll be listing out all of the currently-available states in the system, in the view for this action, so you may as well load them up now too. After you're done, the whole controller class will appear as shown in the following listing.

Listing 285. app/controllers/admin/states_controller.rb

```
class Admin::StatesController < Admin::ApplicationController
  def index
    @states = State.all
  end
end
```

Next on the menu is defining the view for this action, in a brand new file to be located at `app/views/admin/states/index.html.erb`. This view must contain the "New State" link your feature will go looking for, and it should also include a list of states so that anyone looking at the page knows which states already exist. The code to do all this is shown in the following listing.

Listing 286. app/views/admin/states/index.html.erb

```
<header>
  <h1>States</h1>

  <ul class="actions">
    <li><%= link_to "New State", new_admin_state_path, class: "new" %></li>
  </ul>
</header>

<ul class="states">
  <% @states.each do |state| %>
    <li><%= render "states/state", state: state %></li>
  <% end %>
</ul>
```

We're copying the same format we have for all of our previous views - the header section at the top, with the "New State" link, and a list of all the existing states. We're reusing the `state` partial that we used to display the states with colours on the ticket page - but we have to tell Rails manually which partial to use, because now we're in the `admin` namespace (so Rails will only look for the partial in the `app/views/admin` folder).

We'll also add a little bit of CSS to make the states stand out from each other - at the moment they look like this:

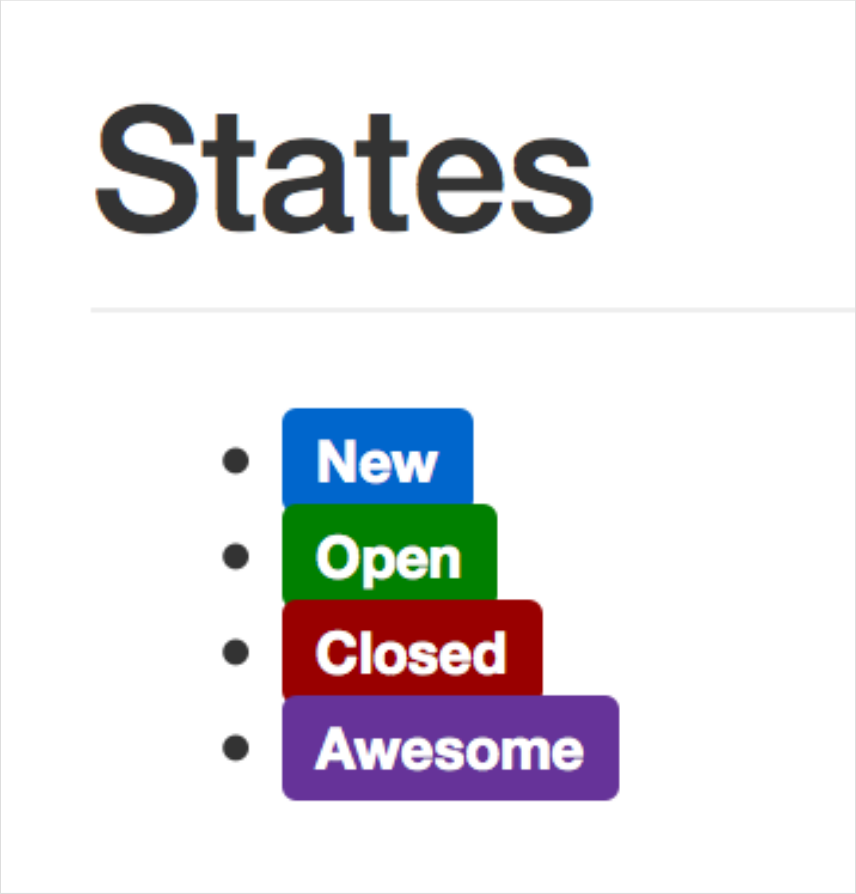


Figure 53. Smushed-up states

Which isn't very easy to read (or otherwise aesthetically pleasing). To separate them a little bit, we'll modify the selector at the bottom of `app/assets/stylesheets/application.css.scss` to add this new `states` list.

```
.tickets li, .states li {
  padding-bottom: 10px;
}
```

Now you'll have a pretty list of states displaying in your admin area. Nice. With this view now written, your feature will progress a little further, but whinge about the `new` action when you run `bundle exec rspec spec/features/admin/creating_states_spec.rb`:

```
1) Admins can create new states for tickets with valid details
   Failure/Error: click_link "New State"
   AbstractController::ActionNotFound:
     The action 'new' could not be found for Admin::StatesController
```

You should add the `new` action to `Admin::StatesController` if you want to continue any further. It should be defined as follows inside that controller:

```
class Admin::StatesController < Admin::ApplicationController
  ...
  def new
    @state = State.new
  end
end
```

You now need to create the view for this action at `app/views/admin/states/new.html.erb`, and fill it in with the following content:

```
<header>
  <h1>New State</h1>
</header>

<%= render "form", state: @state %>
```

You're using a form partial here again, because it's a best practice and also in case you ever want to use it for an `edit` action. In a new file for your partial, at `app/views/admin/states/_form.html.erb`, you can put the form that will be used to create new states. This form is pretty simple: it only needs a couple of text fields for the name and color, and a submit button to submit the form.

Listing 287. app/views/admin/states/_form.html.erb

```
<%= bootstrap_form_with model: [:admin, state], local: true do |f| %>
  <%= f.text_field :name %>
  <%= f.color_field :color %>

 <%= f.submit class: 'btn btn-primary' %>
<% end %>
```

Because the `state` variable rendered in the form is a new instance of the `State` model, the `submit` method will display a submit button with the text "Create State." That's just what your feature needs.

With this form partial done, your feature should run a little further. You should check this now by running `bundle exec rspec spec/features/admin/creating_states_spec.rb`.

```
1) Admins can create new states for tickets with valid details
   Failure/Error: click_button "Create State"
   AbstractController::ActionNotFound:
     The action 'create' could not be found for Admin::StatesController
```

Right. You'll need to create the `create` action too, which you'll define inside `Admin::StatesController`, as shown in the following listing.

Listing 288. app/controllers/admin/states_controller.rb

```
class Admin::StatesController < Admin::ApplicationController
  ...
  def create
    @state = State.new(state_params)
    if @state.save
      flash[:notice] = "State has been created."
      redirect_to admin_states_path
    else
      flash.now[:alert] = "State has not been created."
      render "new"
    end
  end

  private

  def state_params
    params.require(:state).permit(:name, :color)
  end
end
```

There's nothing here that you haven't seen many times over by now. With the `create` action defined in your `Admin::StatesController`, you'll now be able to run `bundle exec rspec spec/features/admin/creating_states_spec.rb` and have it pass.

```
1 example, 0 failures
```

Very good! By implementing a feature that lets the admin users of your site create states, you've provided a base on which you can build the other state-related features. You shouldn't have broken anything by making these changes, but it won't hurt to run `bundle exec rspec` to make sure. You should see the following:

```
43 examples, 0 failures, 1 pending
```

There's one pending spec inside `spec/helpers/admin/states_helper_spec.rb`. You can delete this file now. When you rerun `bundle exec rspec`, there should be this great green output:

```
42 examples, 0 failures
```

Celebrate with a glass of ice-cold refreshing Diet Coke[67], and commit these changes now:

```
$ git add .
$ git commit -m "Admins can create new states for tickets"
$ git push
```

With this base defined, you can move on to more exciting things than CRUD, such as defining a default state for your tickets. We do a lot of CRUD-building in Rails apps, hopefully it's becoming very natural to you by now.

10.4.2. Defining a default state

Adding a default state for the tickets in your application will provide a sensible way of grouping tickets that haven't yet been actioned in the system, making it easier for them to be found. The easiest way to track which state is the default is to add a boolean column called `default` to your `states` table, which is set to `true` if the state is the default, and `false` if not.

To get started, you need to write a feature that covers changing the default status. At the end of this feature, you'll end up with the `default` field in the `states` table, and then you can move on to making the tickets default to this state.

Create a new feature called `spec/features/admin/managing_states_spec.rb` and fill it with the content from the following listing.

Listing 289. spec/features/admin/managing_states_spec.rb

```
require "rails_helper"

RSpec.feature "Admins can manage states" do
  let!(:state) { FactoryBot.create :state, name: "New" }

  before do
    login_as(FactoryBot.create(:user, :admin))
    visit admin_states_path
  end

  scenario "and mark a state as default" do
    within list_item("New") do
      click_link "Make Default"
    end

    expect(page).to have_content "'New' is now the default state."
  end
end
```

In this scenario you've got one new method call, which you'll need to define for this feature to run. This new `list_item` method will need to return a selector for the list item with the specified content. This method is assisting the Capybara test in its job, and will be reusable across many tests, so we should define it in a new file called `spec/support/capybara_finders.rb`. Define it using the code from the following listing:

Listing 290. spec/support/capybara_finders.rb

```
module CapybaraFinders
  def list_item(content)
    find("ul:not(.actions) li", text: content)
  end
end

RSpec.configure do |c|
  c.include CapybaraFinders, type: :feature
end
```

This method simply takes some text, finds a list item on the page (that isn't an action link) with the specified content, and then returns it. Capybara will then use that element as the basis for all actions inside the block. It's great to write these little helper methods that really make your tests more readable, and it's so easy to do!

To load these helpers in `spec/support`, we need to uncomment this line in `spec/rails_helper.rb`:

Listing 291. spec/rails_helper.rb

```
# Dir[Rails.root.join('spec', 'support', '**', '*.rb')].each { |f| require f }
```

Just remove the hash at the start of this line, and then RSpec will load the helper files:

Listing 292. spec/rails_helper.rb

```
Dir[Rails.root.join('spec', 'support', '**', '*.rb')].each { |f| require f }
```

Now that this method is defined, let's see what the test says when it's run using `bundle exec rspec spec/features/admin/managing_states_spec.rb`.

```
1) Admins can manage states and mark a state as default
   Failure/Error: click_link "Make Default"
   Capybara::ElementNotFound:
     Unable to find link "Make Default"
```

The feature is failing now because it's found the right list item on the page, but there's no "Make Default" link in it. Let's fix that now - open the `app/views/admin/states/index.html.erb` view, and add a new "Make Default" link in for each list item.

Listing 293. Now with added "Make Default" links

```
<ul class="states">
  <% @states.each do |state| %>
  <li>
    <%= render "states/state", state: state %>
    <% if state.default? %>
      (Default)
    <% else %>
      <%= link_to "Make Default", make_default_admin_state_path(state), method: :patch %>
    <% end %>
  </li>
  <% end %>
</ul>
```

In this view, you have the states being displayed with a "(Default)" label next to them if they

are indeed the default state. If the state isn't the default, there's an option there to make it the default with the "Make Default" link.

When you run your feature again with `bundle exec rspec spec/features/admin/managing_states_spec.rb`, you'll find out that we haven't yet defined the `default?` method on our `State` instances.

```
1) Admins can manage states and mark a state as default
   Failure/Error: visit admin_states_path
   ActionView::Template::Error:
     undefined method `default?' for #<State id: 9, name: "New"...
```

We said earlier that we were going to do this with a boolean field on our `State` model - we need to set that up now. Generate a migration to create the new field with the following code:

```
$ rails g migration add_default_to_states default:boolean
```

Don't run this migration just yet. With the default column being a boolean field, it's going to need to know what its default value should be; either `true` or `false`. Edit the migration and define that the default value of the `default` attribute should be `false`:

Listing 294. db/migrate/[timestamp]_add_default_to_states.rb

```
class AddDefaultToStates < ActiveRecord::Migration[6.1]
  def change
    add_column :states, :default, :boolean, default: false
  end
end
```

With this small change, every `State` object that's created will have the `default` attribute set to `false` by default. Now run the migration using `rails db:migrate`.

Having a field named `default` will generate a getter and setter method, called `default` and `default=`, on our model for us. However, Rails will also create a convenience method called `default?` for us, because the field is a boolean. Our test will get a little further now that this method is defined, so let's run it again with `bundle exec rspec spec/features/admin/managing_states_spec.rb`:

```
1) Admins can manage states and mark a state as default
   Failure/Error: visit admin_states_path
   ActionView::Template::Error:
     undefined method `make_default_admin_state_path' for #<#< Class:...
```

We don't yet have this method defined. It should take you to a new `make_default` action in `Admin::StatesController`, much like `edit_admin_state_path` takes you to the `edit` action.

To define the new member route, change the `resources :states` line in the `admin` namespace inside `config/routes.rb` to the following:

Listing 295. Adding a new non-resourceful `make_default` route

```
Rails.application.routes.draw do
  namespace :admin do
    ...
    resources :states, only: [:index, :new, :create] do
      member do
        patch :make_default
      end
    end
    ...
```

With this member route now defined, your feature will complain that it's missing the `make_default` action when you re-run it with `bundle exec rspec spec/features/admin/managing_states_spec.rb`:

```
1) Admins can manage states and mark a state as default
   Failure/Error: click_link "Make Default"
   AbstractController::ActionNotFound:
     The action 'make_default' could not be found for
     Admin::StatesController
```

The `make_default` action will be responsible for making the state you've selected the new default state, as well as for setting the old default state to not be the default anymore. You can define this action inside `app/controllers/admin/states_controller.rb`, as shown in the following listing.

Listing 296. app/controllers/admin/states_controller.rb

```
class Admin::StatesController < Admin::ApplicationController
  ...
  def make_default
    @state = State.find(params[:id])
    @state.make_default!

    flash[:notice] = "'#{@state.name}' is now the default state."
    redirect_to admin_states_path
  end
  ...
end
```

Rather than putting the logic that changes the selected state to the new default inside the controller, you should place it in the model. To trigger a state to become the new default state, you'll call the `make_default!` method on it. It's a best practice to put code that performs functionality like this inside the model, so that it can be in any place that uses an instance of this model.

This `make_default!` method can be defined in the `State` model, as shown in the following listing.

Listing 297. app/models/state.rb

```
class State < ActiveRecord::Base
  def make_default!
    State.update_all(default: false)
    update!(default: true)
  end
  ...
```

There are two parts to this method - first we call `update_all` to make sure that all states in the system have their `default` value set to false (there should be only one, but it never hurts to make sure); and then we update the current state instance to have a `default` value of `true`.

When you run your feature again with `bundle exec rspec spec/features/admin/managing_states_spec.rb`, you'll get a happy ending to this feature:

```
1 example, 0 failures
```

Great to see! Let's make sure we haven't broken anything else in our specs by running `bundle`

`exec rspec`:

```
43 examples, 0 failures
```

Now that we know we haven't broken anything, let's commit and push our changes:

```
$ git add .
$ git commit -m "Admins can now set a default state for tickets"
$ git push
```

We now have a concept of having a default state, but at the moment we're not actually using the default state for anything. It would be great if our app auto-assigned this new state to newly-created tickets - we'll look at implementing that next.

10.4.3. Applying the default state

When a ticket is created now, the state of that ticket will be `nil` - we're not assigning a state anywhere. A state will only be assigned when a comment is created, which isn't great. We have a method of setting a default state in our system now - that state should be automatically assigned to newly-created tickets. Because this should be automatic functionality, every time anyone creates a ticket, we can implement it with another callback, this time in our `Ticket` model.

We'll start by modifying one of our existing specs, the "Creating tickets" spec, to make sure the default state gets assigned. Add a new definition of a default state to the top of the spec, like so:

Listing 298. spec/features/creating_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new tickets" do
  let!(:state) { FactoryBot.create :state, name: "New", default: true }
  ...
```

And then we can verify that the state is assigned to the ticket, after we create it:

```
...
scenario "with valid attributes" do
  ...
  within(".ticket") do
    expect(page).to have_content "State: New"
    expect(page).to have_content "Author: #{user.email}"
  end
  ...
```

If we run this spec now with `bundle exec rspec spec/features/creating_tickets_spec.rb`, it will fail:

```
1) Users can create new tickets with valid attributes
   Failure/Error: expect(page).to have_content "State: New"
     expected to find text "State: New" in "Ticketee Toggle navigation..
```

The state isn't getting assigned correctly! Now we can implement the functionality, and **know** that it works, as long as our tests pass.

Let's implement a new callback at the bottom of our `Ticket` model, to be run before a ticket is created:

Listing 299. A new `before_create` callback in `app/models/ticket.rb`

```
class Ticket < ActiveRecord::Base
  ...
  before_create :assign_default_state

  private

  def assign_default_state
    self.state ||= State.default
  end
end
```

Is it really that simple? Well, no. If we run the specs again, we'll see we haven't defined the `State.default` method anywhere:

```
1) Users can create new tickets with valid attributes
   Failure/Error: click_button "Create Ticket"
   NoMethodError:
     undefined method `default' for State(id: integer, name: string,
     color: string, default: boolean):Class
```

`default` needs to be a new class method on the `State` model, to return the state with the field `default` set to `true`. We could have done this query directly in the callback, but that would be exposing the internals of the `State` model to the `Ticket` model. The `Ticket` model doesn't care that the default-ness of a `State` is decided by a boolean value - it just cares that it can ask the `State` model what the default state is, and then use it.

We can define this new `default` method at the top of the `State` model like so:

Listing 300. app/models/state.rb

```
class State < ActiveRecord::Base
  def self.default
    find_by(default: true)
  end

  ...
```

Is it really that simple? Well... yes. Let's run our spec with `bundle exec rspec spec/features/creating_tickets_spec.rb` again to make sure:

```
5 examples, 0 failures
```

Perfect! We're creating a new ticket, and the state that we designated as the default state is automatically being assigned to the ticket. Make sure we haven't broken anything else, by running `bundle exec rspec`:

```
43 examples, 0 failures
```

10.4.4. Setting a default state in seed states

There's just one last thing that we need to do with default states - we need to set one of the states we've defined in our seeds, as the default state.

In `db/seeds.rb` we earlier defined four states:

```
unless State.exists?
  State.create(name: "New", color: "#0066CC")
  State.create(name: "Open", color: "#008000")
  State.create(name: "Closed", color: "#990000")
  State.create(name: "Awesome", color: "#663399")
end
```

Logically, "New" should be the default state for all newly-created tickets. So you can update that line to also set that state as the default:

Listing 301. Marking one of the seed states as the default state

```
State.create(name: "New", color: "#0066CC", default: true)
```

Easy done! Now if you were to recreate your database from scratch, you'd have a correctly-set default state, and all tickets you created from then on would have their state set correctly.

Awesome. Commit and push your new changes:

```
$ git add .
$ git commit -m "Auto-assign the default state to newly-created tickets"
$ git push
```

10.5. Summary

We began this chapter by writing the basis for the work later on in the chapter: comments. By letting users posts comments on a ticket we can let them add further information to it, and tell a story with them.

With the comment base laid down we implemented the ability for users to be able to change a ticket's state when they post a comment. For this, we tracked the state of the ticket before the comment was saved and the state assigned to the comment so we could show transitions:

Figure 54. Replay: state transitions

We finished up by limiting the ability to change states to only those who have permission to do so, much like how we've previously limited the abilities of reading projects and creating tickets in previous chapters. While doing this, we saw how easy it was for somebody to download the source of our form and alter it to do their bidding, and then how to protect it from that.

In chapter 11, you'll add tags to your tickets. Tags are words or short phrases that provide categorization for tickets, making them easier for users to manage. Additionally, you'll implement a search interface that will allow users to find tickets with a given tag or state.

[64] This is **not** our preferred way of spelling "colour", but we're trying to please our audience.
[65] The primary key, in this case, is the `id` field, which is automatically created for each model by `create_table`. Primary key fields are, by default, indexed.
[66] The hexadecimal colour #663399 is also known as "rebeccapurple", in memory of Rebecca Meyer, beloved daughter of CSS guru Eric Meyer.
[67] Or your preferred beverage of choice.

Chapter 11. Sending email

In the previous chapter, you allowed users of Ticketee to leave comments on tickets. In this chapter, you'll begin to send emails to your users. When a user signs up to Ticketee, they use their email address as a way for the system to uniquely identify them. With this email address, you can send them updates about important events in the system, such as a new comment being added to a ticket.

Before you go about configuring your application to send emails into the real world, you'll add two more features to Ticketee. The first feature automatically subscribes a user to a "watchers" list for the tickets they create. Every time a user's ticket is updated by another user, the creator should receive an email. This is helpful, as it allows users to keep up to date with the tickets they've created. The second feature will allow users to add themselves to or remove themselves from the watchers list for a given ticket.

With these features in place, all users who are watching a ticket will be notified via email that a comment has been posted to that ticket, what the comment was, and about any state change that took place. If a user posts a comment to a ticket and they're not watching it, they'll automatically be added to this watchers list and receive notifications whenever anybody who's not them posts a comment on the ticket. They can unsubscribe later, by visiting the ticket page and removing themselves from the watchers list. Email is a tried-and-true solution for receiving notifications of events such as this.

The first thing you're going to do is set up a way for users to receive notifications when a comment is posted to a ticket they've created. This is that "watchers" list we talked about. Let's dive into creating the feature and code for this functionality now.

11.1. Sending ticket notifications

You want to provide users with the ability to watch a ticket. By watching a ticket, a user will be notified by email whenever a new comment is posted to the ticket. The email will contain the name of the user who updated the ticket, the comment text and a URL that will send the user right to the comment.

To test all this, you'll use the Email Spec[68] gem. This gem provides very useful RSpec helpers that allow you to easily verify that an email was sent during a test, and you'll be taking full advantage of the features that this gem provides in the feature that you'll be writing right

now.

11.1.1. Automatically watching a ticket

This feature will test that when a user creates a ticket, they're automatically added to the watchers list for that ticket. Whenever someone else updates this ticket, the user who created it (and later, anybody else watching the ticket) will receive an email notification.

Create a new file at `spec/features/ticket_notifications_spec.rb`, and fill it with the content from the following listing.

Listing 302. spec/features/ticket_notifications_spec.rb

```
require "rails_helper"

RSpec.feature "Users can receive notifications about ticket updates" do
  include ActiveJob::TestHelper

  let(:alice) { FactoryBot.create(:user, email: "alice@example.com") }
  let(:bob) { FactoryBot.create(:user, email: "bob@example.com") }
  let(:project) { FactoryBot.create(:project) }
  let(:ticket) do
    FactoryBot.create(:ticket, project: project, author: alice)
  end

  before do
    ticket.watchers << alice

    login_as(bob)
    visit project_ticket_path(project, ticket)
  end

  scenario "ticket authors automatically receive notifications" do
    fill_in "Text", with: "Is it out yet?"
    click_button "Create Comment"

    perform_enqueued_jobs

    email = find_email!(alice.email)
    expected_subject = "[Ticketee] #{project.name} - #{ticket.name}"
    expect(email.subject).to eq expected_subject

    click_email_link_matching(/projects/, email)
    expect(current_path).to eq project_ticket_path(project, ticket)
  end

  scenario "comment authors do not receive emails" do
    fill_in "Text", with: "Is it out yet?"
    click_button "Create Comment"

    perform_enqueued_jobs

    email = find_email(bob.email)
    expect(email).to be_nil
  end
end
```

In this feature, you set up two users: one called Alice and one called Bob. Both Alice and Bob are setup to be watchers on the ticket. This feature checks that when Bob leaves a comment

on the ticket that Alice has created, then Alice should receive an email because Alice is listed as a watcher on this ticket. In the second scenario, we make sure Bob doesn't receive an email. That's because Bob should know what comment they left!

In both scenarios we use the `perform_enqueued_jobs` helper that is included from `ActiveJob::TestHelper`. Our mailer will run as a background job, and so we need to tell Active Job—Rails' background job library—to run that job at the particular point. If we do not do that, then the calls to find emails will not work at all.

The `find_email!` method here is from the Email Spec gem, and it will open the most recent email sent to the specified email address (or raise an exception if it can't find one). The next couple of lines in the scenario will check that email to see if it contains the correct subject, including the project and ticket names. The final trick in the scenario is to click the link that matches some text in the email (using `click_email_link_matching`, surprise!), and then validate that the link goes to the ticket page for the correct ticket.

Speaking of Email Spec, let's install it. Add this to the `test` section of your `Gemfile`:

```
group :test do
  ...
  gem "email_spec", "~> 2.2.0", require: false
end
```

And run `bundle` to install it.

You'll also need a tiny amount of configuration, to include Email Spec's methods into your tests. Add this code to `spec/support/email_spec.rb`:

```
require "email_spec"

RSpec.configure do |config|
  config.include EmailSpec::Helpers
  config.include EmailSpec::Matchers
end
```

Without this, we couldn't use Email Spec and its helpers (such as `find_email!`) in our specs. It also includes some RSpec matchers that we'll look at using later when we actually want to test our emails.

When you run the ticket notifications feature using `bundle exec rspec`

`spec/features/ticket_notifications_spec.rb`, you'll see that the `watchers` association hasn't been defined yet:

```
1) Users can receive notifications about ticket updates ticket authors
   automatically receive notifications

    Failure/Error: ticket.watchers << alice

    NoMethodError:
      undefined method `watchers' for #<Ticket:...>
```

Let's start getting this test to pass by defining this association.

11.1.2. Defining the watchers association

The `watchers` method should return a collection of users who are watching a ticket, including (by default) the user who has created the ticket in the first place. This means that in your feature, Alice (the author) receives the email triggered by Bob's comment.

You'll use another `has_and_belongs_to_many` association to define the `watchers` collection, this time in your `Ticket` model. To define it, put this code inside the `Ticket` model:

Listing 303. Adding an association between a ticket and its watchers

```
class Ticket < ActiveRecord::Base
  ...
  has_and_belongs_to_many :watchers,
    join_table: "ticket_watchers",
    class_name: "User"
  ...
```

You pass the `:join_table` option, to specify a custom table name for your `has_and_belongs_to_many`. If you didn't do this, the table name would be inferred by Rails to be `tickets_users` [69], which doesn't really explain the purpose of this table as much as `ticket_watchers` does. You pass another option too, `:class_name`, which tells your model that the objects from this association are `User` objects. If you left this option out, Active Record would infer that you wanted the `Watcher` class instead, which doesn't exist.

You can create a migration that will create this table by using this command:

```
$ rails g migration create_join_table_ticket_watchers tickets users
```

That will create you a migration that looks like this:

Listing 304. db/migrate/[timestamp]_create_join_table_ticket_watchers.rb

```
class CreateJoinTableTicketWatchers < ActiveRecord::Migration[6.1]
  def change
    create_join_table :tickets, :users do |t|
      # t.index [:ticket_id, :user_id]
      # t.index [:user_id, :ticket_id]
    end
  end
end
```

This is exactly what we did to create the join table between tickets and tags in chapter 11. We have just one minor change though - Rails has picked up that we wanted a join table from the phrase `create_join_table` in the migration name, but it hasn't picked up the name we wanted for the table. We can do that by specifying the table name as an argument to the `create_join_table` method inside the migration, like so:

Listing 305. When specifying the table name

```
class CreateJoinTableTicketWatchers < ActiveRecord::Migration[6.1]
  def change
    create_join_table :tickets, :users, table_name: :ticket_watchers
      do |t|
      ...
```

We'll also uncomment the two other lines in this migration:

```
class CreateJoinTableTicketWatchers < ActiveRecord::Migration[6.1]
  def change
    create_join_table :tickets, :users, table_name: :ticket_watchers do |t|
      t.index [:ticket_id, :user_id]
      t.index [:user_id, :ticket_id]
    end
  end
end
```

By uncommenting these, we will speed up database queries on this table.

The final thing we will do is ensure only a single `ticket_id` and `user_id` combination can be inserted into this table by using a unique index:

```
class CreateJoinTableTicketWatchers < ActiveRecord::Migration[6.1]
  def change
    create_join_table :tickets, :users, table_name: :ticket_watchers do |t|
      t.index [:ticket_id, :user_id], unique: true
      t.index [:user_id, :ticket_id]
    end
  end
end
```

Save the modified migration, and then run it using `rails db:migrate`.

That's ticket watchers taken care of. However, when you run `bundle exec rspec spec/features/ticket_notifications_spec.rb`, you'll see that the email is not being sent:

```
1) Users can receive notifications about ticket updates ticket authors
   automatically receive notifications
   Failure/Error: email = find_email!(alice.email)
   EmailSpec::CouldNotFindEmailError:
     Could not find email   in the mailbox for alice@example.com.
      Found the following emails:

      []
```

When Bob updates the ticket, Alice doesn't receive an email yet. That's why you wrote the feature: so you can test the behavior that you're about to create! Let's look into how emails are sent by Rails.

You're not really sending emails

Before we carry on, we need to let you in on a little something: All of the emails you're about to create won't be sent to these addresses in the real world, so you don't have to worry about using real email addresses in your tests. How do we do this? Well, there's a setting inside `config/environments/test.rb` that goes like this:

```
config.action_mailer.delivery_method = :test
```

This setting tells Action Mailer to intercept any emails that you try to send, and store them in

the `ActionMailer::Base.deliveries` array.[70] You'll then read the emails out of this array using the helpers provided by Email Spec.

But for now, lets get Ticketee to send Alice an email. For this, you're going to use what's known as a service class.

11.1.3. Using service classes

A service class is a class that provides a simple interface to a piece of business logic, and is typically used to group together related actions and behaviour that should always occur together. If this sounds generic, that's because it is: Rails doesn't have any specific support for service classes. For some extra reading on service classes (sometimes also called service objects), we recommend the following links:

- https://netguru.co/blog/service-objects-in-rails-will-help
- http://blog.codeclimate.com/blog/2012/10/17/7-ways-to-decompose-fat-activerecord-models/ - This article is more generic, but covers a lot more than just service objects.

We'll be using a service class to create a comment, and then send out notifications to all of the watchers of the ticket.

Why don't we just use callbacks in our model, or send the notifications from the CommentsController?

This is a bit of a tricky subject. It would be very easy to say to ourselves "well, we want to send out some email every time we create a comment, so we can write an `after_create` callback in our `Comment` model to do this, right?"

It would be easy to do that, and it would work, but it would also be a bad idea for several reasons:

1) It violates the single-responsibility principle, and introduces tighter coupling

A `Comment` object in our system has one purpose - managing the data that it holds, the content, the state, etc. Introducing this callback that adds behaviour unrelated to this purpose couples this model with the other objects we would be referencing, such as the mailer. If we change something in the mailer, our model (which is just supposed to be saving data) could completely break in weird and wonderful ways.

2) It slows down your tests, and a lot of other things too

This is more a side-effect of the first downside. Think of all of the possible times you might want to create a comment, but not send out these notification emails - when experimenting in the Rails console, or when calling factories in tests, for example. Think of all the tests we've written that create comments - they would now have the added overhead of creating and trying to send emails, that we really don't care about. There's only one time we really want to send them - when a comment is created through the Ticketee web interface.

OK, so why not in the controller then?

1) Controllers are notoriously difficult to test.

In an ideal world, we would want to test this interaction - when we save a comment in the application, that all these emails get triggered - and we'd want to do it easily. Controller testing is not straightforward, and typically requires a lot of fiddly mocking and stubbing because a controller touches all of the different parts of the Rails stack, from the request to the response -

that's its job.

2) The logic isn't reusable if it's locked away in a controller

Later down the track we might want to use this notification logic elsewhere, for example, if we wrote a Rake task that automatically closed tickets that hadn't had any activity in a long time. We'd want to notify the watchers that the ticket was closed in this scenario too, meaning we'd have to duplicate all of the logic from the controller. If we encapsulate it in a service object, we can reuse it in our Rake task, or anywhere else we specifically decide we want to save and send notifications.

Now that we've decided that service objects are a good idea, how do we build one, and where do we put it? There's no predefined place to store them in a Rails application, but a `services` directory sounds like a logical place to put them. Make a new directory, `app/services`, and add a new file, `app/services/comment_notifier.rb`:

Listing 306. app/services/comment_notifier.rb

```
class CommentNotifier
  attr_reader :comment, :watchers

  def initialize(comment)
    @comment = comment
    @watchers = comment.ticket.watchers.excluding(comment.author)
  end

  def notify_watchers
    watchers.each do |user|
      CommentMailer
        .with(comment: comment, user: user)
        .new_comment
        .deliver_later
    end
  end
end
```

This class is a simple PORO (Plain Old Ruby Object) that wraps a `Comment` instance, and lets you manage any code associated with sending notifications out to users.

Let's use this new service in the `create` action of `CommentsController` now by changing that action to this:

Listing 307. Replacing comment creation with the `CommentCreator` service class

```
class CommentsController < ApplicationController
  ...

  def create
    @comment = @ticket.comments.build(comment_params)
    @comment.author = current_user

    if @comment.save
      comment_notifier = CommentNotifier.new(@comment)
      comment_notifier.notify_watchers
      flash[:notice] = "Comment has been created."
      redirect_to [@ticket.project, @ticket]
    else
      flash.now[:alert] = "Comment has not been created."
      @states = State.all
      @project = @ticket.project
      render "tickets/show"
    end
  end

  ...
```

You can see that the creation logic in the controller has been replaced with the new service class and its `build` method. It also unwraps the comment from the service class in the error case; you need to have the actual comment to display on the comment form, not the service class.

When you run `bundle exec rspec spec/features/ticket_notifications_spec.rb`, you'll now be told this:

```
1) Users can receive notifications about ticket updates ticket
   authors automatically receive notifications
    Failure/Error: CommentMailer.new_comment(comment, user).deliver

    NameError:
      uninitialized constant CommentNotifier::CommentMailer
      Did you mean?  CommentsHelper
```

This time, your feature is failing because it can't find the constant `CommentMailer`, which is going to be the class that you use to send out the emails notifying users to new activity on a ticket. To create this class, we'll use a part of Rails we haven't seen yet: Action Mailer.

11.1.4. Introducing Action Mailer

You need to define the `CommentMailer` class to send out ticket update notifications using the `CommentNotifier` service. You can generate this mailer by running the mailer generator.

A mailer is a class defined for sending out emails. To define your mailer, you'll run this command:

```
$ rails g mailer comment_mailer
```

When running this command, you'll see this output:

```
create  app/mailers/comment_mailer.rb
invoke  erb
create    app/views/comment_mailer
invoke  rspec
create    spec/mailers/comment_mailer_spec.rb
create    spec/mailers/previews/comment_mailer_preview.rb
```

The first thing the command generates is the `CommentMailer` class itself, defining it in a new file at `app/mailers/comment_mailer.rb`. Inside this class, you'll define (as methods) the different emails that you'll send out, beginning with the new comment email.

The next thing that's generated is the `app/views/comment_mailer` directory, which is used to store the templates for all emails belonging to the `CommentMailer`. Each method in the `CommentMailer` class will correspond to a different type of email sent out, and each type of email will have its templates in this directory.

The third thing that's generated is `spec/mailers/comment_mailer_spec.rb`, which you won't use right now because you've got your feature testing this notifier anyway.

The final thing is a mailer preview class, which we'll use a little later on in this chapter to preview what our emails look like.

Now that you have the `CommentMailer` class defined, what happens when you run your feature? Run it using `bundle exec rspec spec/features/ticket_notifications_spec.rb` and find out:

```
1) Users can receive notifications about ticket updates ticket authors
   automatically receive notifications
   Failure/Error: click_button "Create Comment"
   NoMethodError:
     undefined method `new_comment' for CommentMailer:Class
   # ...
   # ./app/services/comment_creator.rb:23:in ...
```

In the `CommentMailer` class, you need to define the `new_comment` method, which will build an email to be sent out when a comment is posted. This method needs to get the email address for all the watchers for comment's ticket and send an email to each of them. You can define the method like this:

Listing 308. app/mailers/comment_mailer.rb

```
class CommentMailer < ApplicationMailer
  def new_comment
    @comment = params[:comment]
    @user = params[:user]
    @ticket = @comment.ticket
    @project = @ticket.project

    subject = "#{@project.name} - #{@ticket.name}"
    mail(to: @user.email, subject: subject)
  end
end
```

This may look a little wrong because you're defining it as an instance method, even though the error complains about a class method. Rest assured, this `new_comment` method is truly the method that Action Mailer is looking for - this is a little bit of magic performed by Action Mailer for your benefit.

The `new_comment` method doesn't exist on the `CommentMailer` class, but it does exist on the instances. So what happens here is that the call to that method is caught by `method_missing`, which then initializes a new instance of this class and then, with some sleight of hand, ends up calling your `new_comment` method.

Metaprogramming with `method_missing`

`method_missing` is one of those special sauces that make Ruby such a wonderful programming language. It acts like a catch-all method in a class - if you call a method that the class doesn't know how to respond to, Ruby will first look for a `method_missing` method to handle this unexpected method call. If `method_missing` doesn't exist, then the typical `NoMethodError` will be raised.

With `method_missing`, you can truly create dynamic classes that respond to a wide variety of methods, based on things like database fields - Active Record uses `method_missing` extensively to define getter and setter methods for each of the fields in the model's database table.

For an contrived example of `method_missing` in the flesh, you could define a class like the following:

```
class StringLength
  def method_missing(arg)
    "'#{arg}' has #{arg.length} letters!"
  end
end
```

And then interact with your class like so:

```
> StringLength.new.a_word
=> "'a_word' has 6 letters!"
> StringLength.new.something_long
=> "'something_long' has 14 letters!"
> StringLength.new.does_it_respond_to_anything?
=> "'does_it_respond_to_anything?' has 28 letters!"
```

You probably won't use `method_missing` extensively in your Rails apps, but it's good to know it exists, and the kinds of dynamic interfaces it can create.

When the `new_comment` method is called in your `CommentMailer`, it will attempt to render a plain-text template for the email, which should be found at `app/views/comment_mailer/new_comment.text.erb`. (You'll define this template after you've got the method working.)

Mailers work like controller actions in that all instance variables are accessible to the rendered template, but local variables are not, so we assign a set of variables as instance variables so we can use them in the content of the email. After assigning the variables, you then use the `mail` method to generate a new email, passing the `to` and `subject` keys, which define where the email goes to as well as the subject for the email.

When you run `bundle exec rspec spec/features/ticket_notifications_spec.rb`, you'll see that the user now receives an email and therefore is able to open it, but the link you're looking for isn't there, which brings up this error:

```
1) Users can receive notifications about ticket updates ticket authors
   automatically receive notifications
   Failure/Error: click_button "Create Comment"
   ActionView::MissingTemplate:
    Missing template comment_mailer/new_comment with "mailer". Searched in:
      * "comment_mailer"
```

Methods defined within an Action Mailer class need to have a corresponding template that defines the content of the email, much like actions in controllers have templates.

Let's define a template for the `CommentMailer#new_comment` mailer method now.

11.1.5. An Action Mailer template

Templates for Action Mailer classes go in `app/views` because they serve an identical purpose to the controller views: they display a final, dynamic result to users. Once you have this template in place, the plain-text email a user receives will look like the image below. As you can see, you'll need to mention who updated the ticket and what they updated it with, and you'll also need to provide a link to the ticket.

Figure 55. Your first email

You can define a text template for your `new_comment` method at `app/views/comment_mailer/new_comment.text.erb` as follows.

Listing 309. app/views/comment_mailer/new_comment.text.erb

```
<%= project_ticket_url(@project, @ticket) %>
```

Wait, `text.erb`? Yes! This is the template for the plain-text version of the email, after all. Remember, the format of a view in Rails is the first part of the file extension, with the latter part being the actual file type. Because you're sending a text-only email, you use the text format here. This template is a little barren at the moment, but it's all that's required to get this feature working. You'll flesh it out in a little while.

When you run the test again, you'll see this error:

```
1) Users can receive notifications about ticket updates ticket authors
   automatically receive notifications
   Failure/Error: click_button "Create Comment"
   ActionView::Template::Error:
     Missing host to link to! Please provide the :host parameter, set
     default_url_options[:host], or set :only_path to true
   # ...
   # ./app/views/comment_mailer/new_comment.text.erb:1:in ...
```

This is the line that we just added! This line is causing our error because the view wants to know the hostname to use to construct the full URL for a ticket, but we haven't told the app what that hostname is. In the controller context the view knows which URL to use because it has a request to base this information off. Mailers don't have access to anything to do with the request, and this means that our mailer won't have that information.

We can fix this now by putting this line in `config/environments/test.rb`, inside the `Rails.application.configure` block:

```
Rails.application.configure do
  ...
  config.action_mailer.default_url_options = {
    host: "localhost:3000"
  }
  ...
```

This host will point any emails back to our local server, which will be good enough for testing purposes. We'll want to set something in `config/environments/production.rb` too, but we'll do that in the deployment chapter later on when we deal with a production deployment of

our application.

We will also need this same configuration in `config/environments/development.rb` for when we test emails in the development environment so let's add that in there too:

Listing 310. config/environments/development.rb

```
Rails.application.configure do
  ...
  config.action_mailer.default_url_options = {
    host: "localhost:3000"
  }
  ...
```

When we run our tests again, they'll now pass:

```
1 example, 0 failures
```

You've done quite a lot to get this simple, little feature to pass.

In the beginning, you created a service class called `CommentNotifier`. This service class wraps up all the logic around notifying users that a comment has been posted to a ticket they're watching.

We wouldn't know what notifications to send out if it wasn't for the `watchers` association that we added to the `Ticket` model. This association currently tracks only the user who created the ticket but eventually we'll change it so that anyone who comments on the ticket will automatically become a watcher too.

The actual process of notifying is done by the `CommentMailer` class. `CommentMailer` is an Action Mailer class that's responsible for sending out emails to the users of your application. In this file you defined the `new_comment` method which is responsible for collecting all the information to present to the mailer template, exactly like how an action in a controller collects all the information for a view.

Finally, you defined the template for the `new_comment` email at `app/views/comment_mailer/new_comment.text.erb`, and included the link that you click to complete the final step of your scenario. You had to also tell Action Mailer what host to use for its URL building.

This scenario completes the first steps of sending email notifications to your users. You should now run all your tests to make sure you didn't break anything, by running `bundle exec rspec`:

```
46 examples, 0 failures
```

Great to see everything still passing! The one pending spec is located in `spec/mailers/comment_mailer_spec.rb`, but rather than deleting the file, keep it and just delete the pending spec in it. You're going to be using that file in the next section. If you delete just the spec and rerun `bundle exec rspec`, you'll see this:

```
45 examples, 0 failures
```

You've added email ticket notifications to your application, so you should now make a commit saying just that and push it:

```
$ git add .
$ git commit -m "Added basic email ticket notifications"
$ git push
```

Now that you've got your application sending plain-text emails, you should flesh out the emails a bit more, giving them some content that tells the user why they're receiving the email, rather than just having them contain a link. To test that, you'll be using the mailer spec that was generated along with the mailer.

11.2. Testing with mailer specs

Now you're going to get down into the nitty-gritty of exactly how your mailer works, with mailer specs. Mailer specs are generally contained within the files that are generated along with the mailer, and they test intimate details about the mail that's sent out by those mailers, such as the body content.

In this section, you're going to learn how to write a mailer spec—you'll write one that checks that the email contains specific content.

The spec that you'll be writing now will make sure that when a user receives the email that it contains a phrase like "[user] has just updated the [ticket name] for [project name]", and the

content for the comment. You may already be familiar with these types of emails from services such as Facebook. To test this, you'll first need to create a project, and then a ticket for that project that belongs to a user. The test itself will create a comment and then check the email that's just been sent to ensure that it's got the correct content.

Write the content from the following listing into a new file at `spec/mailers/comment_mailer_spec.rb`. Yes, this is the same file that we just deleted, but we didn't need it then!

Listing 311. spec/mailers/comment_mailer_spec.rb

```
require "rails_helper"

RSpec.describe CommentMailer, type: :mailer do
  describe "new_comment" do
    let(:project) { FactoryBot.create(:project) }
    let(:ticket_owner) { FactoryBot.create(:user) }
    let(:ticket) do
      FactoryBot.create(:ticket,
        project: project, author: ticket_owner)
    end

    let(:commenter) { FactoryBot.create(:user) }
    let(:comment) do
      Comment.new(ticket: ticket, author: commenter,
        text: "Test comment")
    end

    let(:email) do
      CommentMailer
        .with(comment: comment, user: ticket_owner)
        .new_comment
    end

    it "sends out an email notification about a new comment" do
      expect(email.to).to include ticket_owner.email
      title = "#{ticket.name} for #{project.name} has been updated."
      body = email.default_part_body
      expect(body).to include title
      expect(body).to include "#{commenter.email} wrote:"
      expect(body).to include comment.text
    end
  end
end
```

At the beginning of this test, a whole bunch of things are set up. First, the test needs a project

and a ticket. That part's easy. The ticket needs to have an author associated with it so that there's someone to be notified when the comment notification goes out. Next, there needs to be a comment so that the mailer can act on something. The comment needs to have some text so that you can validate that it shows up in the email that's sent out.

Inside the test itself, you create a new mail message by calling the `new_comment` method and passing it the `comment` and `ticket_owner` objects. The ticket owner is passed here because that's the user that needs to be notified by this email. For the test, you assert that the `to` address for the email contains the ticket owner's email, that the body should contain a message saying that a ticket has been updated, and that the body contains the comment's text.

When you run this test with `bundle exec rspec spec/mailers/comment_mailer_spec.rb`, you'll see that the email body doesn't contain that specialized message:

```
1) CommentMailer new_comment sends out an email notification about a new
   comment

  Failure/Error: expect(email.body.to_s).to include title

  expected "http://localhost:3000/projects/1/tickets/1\n\n" to include
    "Example ticket for Example project has been updated."

  Diff:
  @@ -1,2 +1,2 @@
  -Example ticket for Example project has been updated.
  +http://localhost:3000/projects/1/tickets/1
```

This failure is happening because you haven't yet put the correct message inside the email; all it contains is a link. To put that message in the email and make it look a whole lot nicer, replace the link inside `app/views/comment_mailer/new_comment.text.erb` with this:

Listing 312. app/views/comment_mailer/new_comment.text.erb

```
Hello!

<%= @ticket.name %> for <%= @project.name %> has been updated.

<%= @comment.author.email %> wrote:

<%= @comment.text %>

You can view this ticket online by going to:
<%= project_ticket_url(@project, @ticket) %>
```

When you rerun `bundle exec rspec spec/mailers/comment_mailer_spec.rb`, you'll see that this spec is now passing because the email now contains the text that you're looking for:

```
1 example, 0 failures
```

Now that you've spruced up the text template for the email, users will receive relevant information about the comment notification, rather than just a link.

With that feature passing, this is a good spot for a commit. But before, lets run our tests with `bundle exec rspec`:

```
46 examples, 0 failures
```

All passing - excellent. Commit and push:

```
$ git add .
$ git commit -m "Give more details about the ticket in email body"
$ git push
```

We have now fleshed out our plain text email's content. It looks a bit more professional now. But how we can we know that for certain? We have a test that shows us that… but a test can only go so far. How do we get to see what an email actually looks like? Fortunately, Rails has a feature for that too: it's called a mailer preview and it's exactly what we're going to look at right now.

11.3. Previewing emails

When we want to look at how an email looks to a user, we can use Action Mailer's previews feature.

You may have noticed that when the generator for the `CommentMailer` was run, there was this line in the output:

```
create    spec/mailers/previews/comment_mailer_preview.rb
```

This file is where we'll be spending this part of this chapter. Currently, this file is almost empty!

Listing 313. spec/mailers/previews/comment_mailer_preview.rb

```
# Preview all emails at http://localhost:3000/rails/mailers/comment_mailer
class CommentMailerPreview < ActionMailer::Preview
end
```

This class inherits from `ActionMailer::Preview` and this `ActionMailer::Preview` provides us with the functionality that we'll use to preview emails.

To preview this particular email, we're going to need to build a few things first: a project, a ticket, a user and a comment. If we don't have these things, we can't construct a `new_comment` email. Let's change the code in the preview class to this:

Listing 314. spec/mailers/previews/comment_mailer_preview.rb

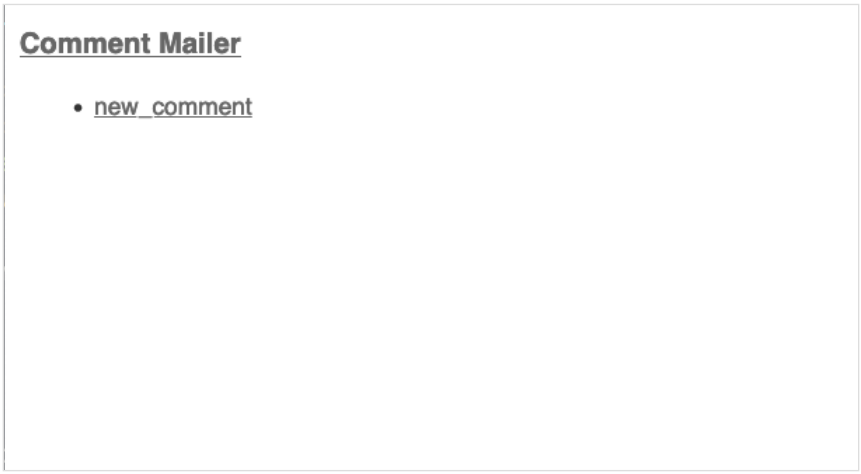
```
# Preview all emails at http://localhost:3000/rails/mailers/comment_mailer
class CommentMailerPreview < ActionMailer::Preview
  def new_comment
    project = Project.new(id: 1, name: "Example Project")
    ticket = project.tickets.build(id: 1, name: "Example Ticket")
    user = User.new(email: "user@ticketee.com")
    comment = ticket.comments.build(author: user, text: "Hello there")

    CommentMailer.with(comment: comment, user: comment.author).new_comment
  end
end
```

This class now contains a `new_comment` method, and that method defines how we generate a

`new_comment` email.

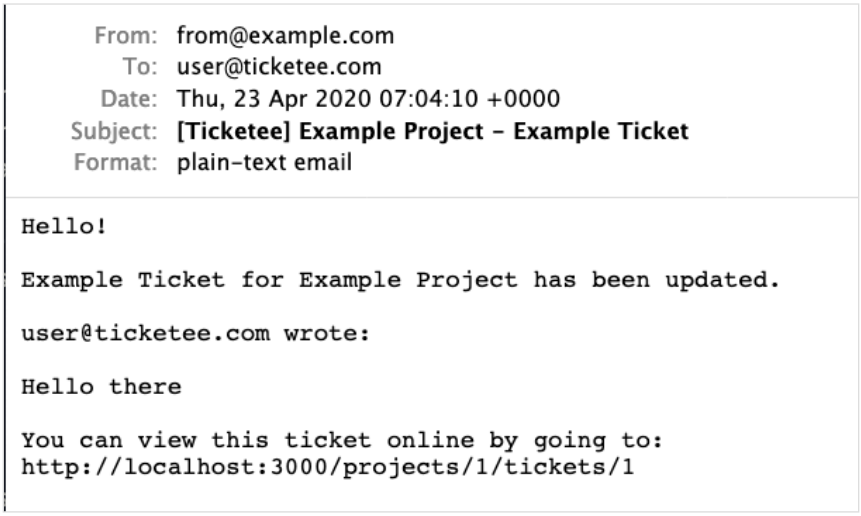
Inside the method, we instantiate four objects that our email will need: `Project`, `Ticket`, `User` and `Comment` objects. These aren't going to be saved to the database; they're going to just be built to display the preview and then our code will forget about them.

With this change to our preview code, we should now be able to go to http://localhost:3000/rails/mailers and see this preview listed:



This special `/rails/mailers` route is only available at `localhost:3000`. When we deploy our application to a production environment in the next chapter, users of our application will not be able to access this route.

When we click on `new_comment` here, we'll see our email appear:

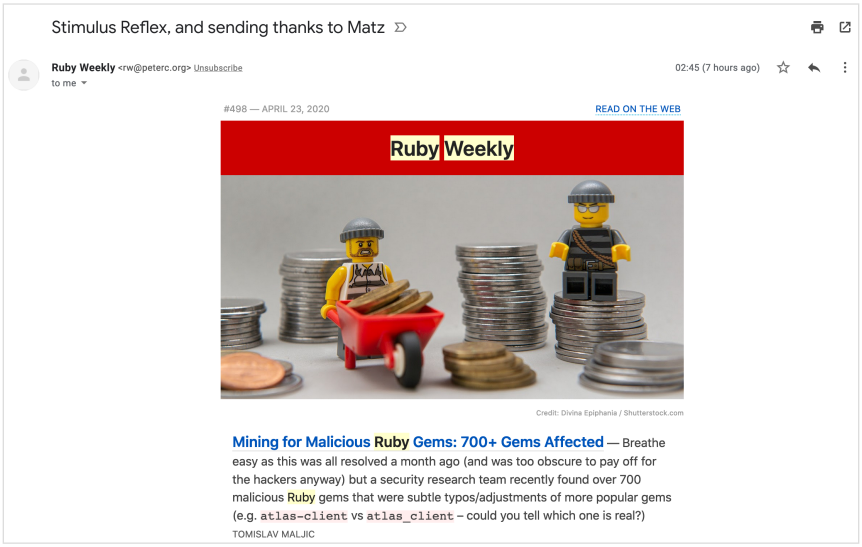


With this previewer, we can quickly and easily preview our emails.

This text email is looking a bit… plain. We could spice up this email by using a HTML format for the email, instead of a plaintext one. Let's now look at that.

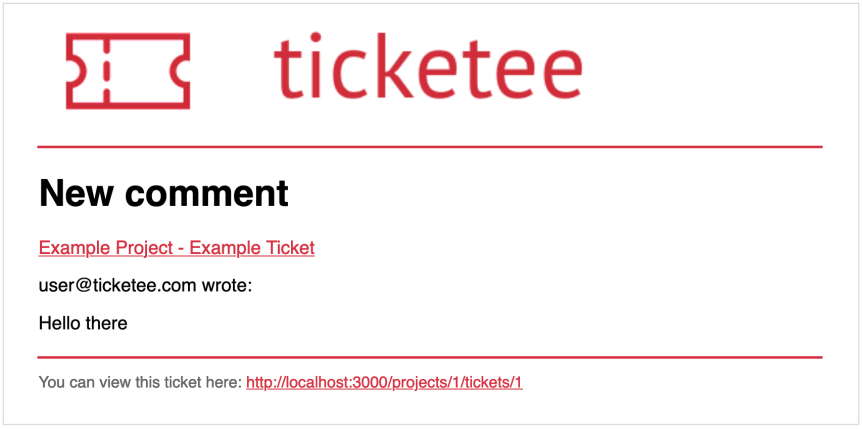
11.4. HTML emails

Ever wondered how people made emails look good? They probably did it with HTML emails. As an example, here's a Ruby Weekly email that uses HTML:



Doesn't that look much better than our plain text email?

We can accomplish the same sort of style upgrade with a few tricks of our own. What we'll end up with is this:



The first of these tricks is to create a HTML template for our email:

Listing 315. app/views/comment_mailer/new_comment.html.erb

```
<div class="wrapper">
  <table class="content">
    <thead>
      <tr>
        <td>
          <div class="logo">
            <%= link_to image_tag("logo.png"), root_url %>
          </div>
        </td>
      </tr>
    </thead>

    <% ticket_url = project_ticket_url(@project, @ticket) %>
    <tbody>
      <tr><td class="divider"></td></tr>
      <tr>
        <td>
          <h1>New comment</h1>

          <%= @ticket.name %> for <%= @project.name %> has been updated.

          <p>
            <%= link_to "#{@project.name} - #{@ticket.name}", ticket_url %>
          </p>

          <p><%= @comment.author.email %> wrote:</p>

          <%= simple_format @comment.text %>
        </td>
      </tr>
      <tr><td class="divider"></td></tr>
      <tr>
        <td class="footer">
          You can view this ticket here:
          <%= link_to ticket_url, ticket_url %>
        </td>
      </tr>
    </tbody>
  </table>
</div>
```

This HTML uses tables for layout. It's widely accepted that using tables for layout is considered bad practice: https://webmasters.stackexchange.com/questions/6036/why-arent-we-supposed-to-use-table-in-a-design/6037. However, HTML email templates need to support a lot of older HTML rendering engines (hello Outlook + friends), and so we need to

use this antiquated way of laying out HTML.

This HTML template starts out with a table that contains a logo, that we can add to `app/assets/images`. You can get this file from the example code on GitHub, or you could make up your own!

Underneath that logo, we output information about the comment. Rather than talking through it step-by-step, let's look at our email preview function now to see what this looks like. We can see this email when we go to http://localhost:3000/rails/mailers/comment_mailer/new_comment.html. Here's what it will look like:

New comment

Example Project - Example Ticket

user@ticketee.com wrote:

Hello there

You can view this ticket here: http://localhost:3000/projects/1/tickets/1

This email is looking a little better than our text based email, but it still needs some work. The font is the system default font, and there's no clear dividers showing up to separate out the text.

Let's look at how we can apply some CSS stylings to this email to make it look better!

11.4.1. Styling HTML emails

To style our email, we're going to use a stylesheet. That shouldn't come as a particularly big surprise given that we're using HTML and so of course we should be using CSS too, right?

Well, emails are kind of special. When we style emails, we need to put the CSS stylings inline with the elements themselves. Remember our nice clean HTML from the last section?

```
<div class="wrapper">
  <table class="content">
  ...
```

In order to support all email clients, we need to put the CSS styles inline:

```
<div class="wrapper" style="max-width: 600px; margin: 0 auto;">
  <table class="content" style="width: 100% !important; max-width: 100%;">
  ...
```

This looks like a lot of work and frankly, is quite ugly! But thanks to another gem built by the wonderful Ruby community, we will not have to write this inline CSS ourselves. This gem is called `premailer-rails`, and it will do the inlining of the CSS for us.

Let's go-ahead and add this gem now:

```
bundle add premailer-rails -v "~> 1.11"
```

The way this gem works is that before Rails sends out an email, the `premailer-rails` gem will read that email and look for any stylesheets included in the email. If there are some styles in a stylesheet, the premailer gem takes the styles out of that stylesheet and applies them to the relevant elements automatically. This allows us to keep our email HTML templates relatively clean, while still being able to support all kinds of email clients.

We don't have to configure this gem, we can just start using it. Let's add a new stylesheet just for emails to our application:

Listing 316. app/assets/stylesheets/emails.scss

```
$ticketee-red: #d63241;

body {
  font-size: 14px;
  font-family: "Helvetica";
}

a {
  color: $ticketee-red;
}

.wrapper {
  margin: 0 auto;
  max-width: 600px;
}

.content {
  width: 100% !important;
  max-width: 100%;
}

.footer {
  padding-top: 10px;
  font-size: 12px;
  color: #676767;
}

.divider {
  background: $ticketee-red;
}
```

This stylesheet will apply styles to the elements in our HTML email template. Before that can happen, we need to specify this stylesheet in the email layout template:

Listing 317. app/views/layouts/mailer.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <%= stylesheet_link_tag "emails" %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

We've added a line here: `<%= stylesheet_link_tag "emails" %>`. This will include the `emails.scss` file and style our emails.

This `app/views/layouts/mailer.html.erb` file contains the layout for every HTML email. To help with presenting our emails, let's move some of that markup from our `new_comment.html.erb` into this file:

Listing 318. app/views/layouts/mailer.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <%= stylesheet_link_tag "emails" %>
    <style>
      /* Email styles need to be inline */
    </style>
  </head>

  <body>
    <div class="wrapper">
    <table class="content">
      <thead>
        <tr>
          <td>
            <div class="logo">
              <%= link_to image_tag("logo.png"), root_url %>
            </div>
          </td>
        </tr>
      </thead>
      <tbody>
        <tr><td class="divider"></td></tr>
        <%= yield %>
      </tbody>
    </table>
  </body>
</html>
```

This will make it so that all of our emails have the same layout, across our application.

With these changes, we can reduce the code in `app/views/comment_mailer/new_comment.html.erb` to this:

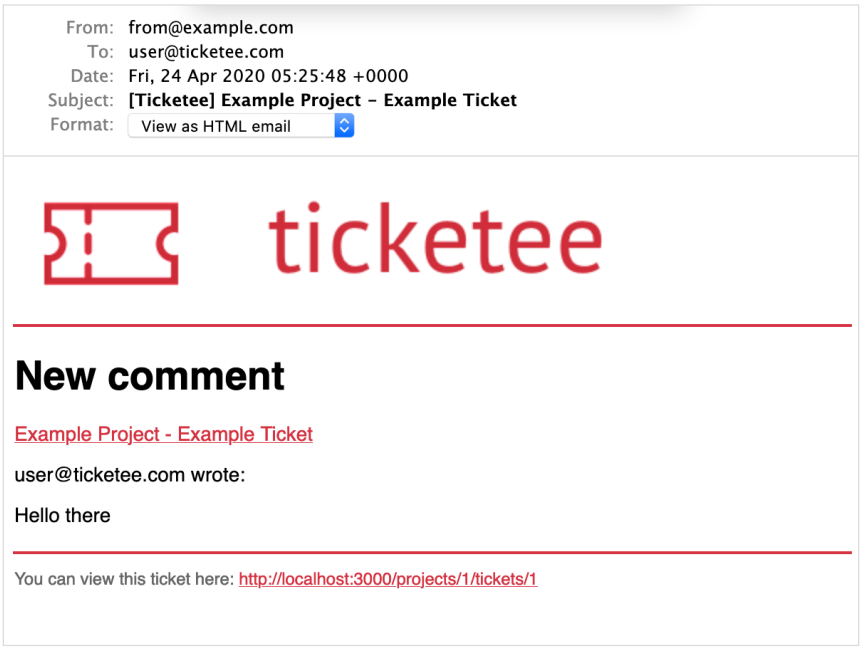
Listing 319. app/views/comment_mailer/new_comment.html.erb

```
<% ticket_url = project_ticket_url(@project, @ticket) %>
<tr>
  <td>
    <h1>New comment</h1>

    <%= @ticket.name %> for <%= @project.name %> has been updated.

    <p>
      <%= link_to "#{@project.name} - #{@ticket.name}", ticket_url %>
    </p>

    <p><%= @comment.author.email %> wrote:</p>

    <%= simple_format @comment.text %>
  </td>
</tr>
<tr><td class="divider"></td></tr>
<tr>
  <td class="footer">
    You can view this ticket here:
    <%= link_to ticket_url, ticket_url %>
  </td>
</tr>
```

That's better!

Now before we can preview what this email looks like, we will need to restart our Rails server because we've added a new gem to the `Gemfile`. Go ahead and do that now.

After you've restarted the server, refresh http://localhost:3000/rails/mailers/comment_mailer/new_comment and you'll now see the great looking HTML email:



If we bring up our web inspector in our browser by right clicking on the email and then going to Inspect Element, we'll see that the styles have been inlined into the template itself by the `premailer-rails` gem.

Our work here is now complete! We have been able to write a HTML template for the email and to have styles inlined.

This HTML template will be the default template shown to users when they receive our email. An important part of how email works is to point out here that when this email is sent out there'll be two parts to the email: a text version of this email and a HTML version of this email. Email clients will attempt to display the HTML one first. Some very old clients will only display the text one. So it's a good idea for us to support both, just in case.

The inlining of the styles was handled by the `premailer-rails` gem. This gem reads any stylesheets included into an email, and will automatically inline styles for us.

We were then able to see this in action by using the mailer preview that we built in the last section.

Now is a great time to make a commit, so let's do that:

```
git add .
git commit -m "Add HTML template for new_comment email"
git push
```

In this section, you've learned how to generate a mailer and create a mailer method for it, and now you're going to look at how you can let people subscribe to these emails. You're currently only subscribing the ticket's author to the list of watchers associated with this ticket, but other people may also wish to be notified of ticket updates. You can do this in two separate ways: through a watch button, and through automatic subscription when a user leaves a comment on a ticket.

11.5. Subscribing to updates

You want to provide other users with two ways to stay informed of ticket updates. The first approach will be very similar to the automatic subscription of the user who creates the ticket, but this time you'll automatically subscribe users who comment on a ticket. You'll reuse the same code that you used in the previous section to achieve this, but not in the way you might think.

The second approach will involve adding a watch button on the ticket page, which will display either "Watch" or "Unwatch", depending on whether the user is watching the ticket or not:

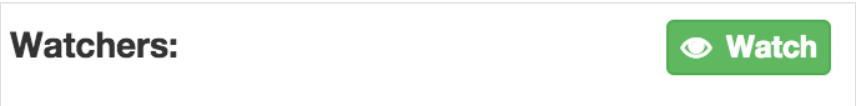


Figure 56. The watch button

We'll first look at implementing the automatic subscription when a user posts a comment to a ticket.

11.5.1. Testing comment subscription

The first feature you need to add will automatically subscribe users to a ticket when they create a comment on it. This is useful because users will likely want to keep up to date with tickets they've commented on. Later on, you'll implement a way for these users to opt out.

To automatically subscribe a user to a ticket upon adding a comment, you'll use `after_create`, just as you did in the `Ticket` model for the author of that ticket. But first you need to ensure

that this works!

You need to add another scenario to the "Ticket notifications" feature, that checks this flow works:

- Alice creates a ticket, and is therefore subscribed to ticket notifications
- Bob leaves a comment on the ticket, and is therefore subscribed too
- Alice replies to Bob's comment by leaving a comment of her own
  - As a result of this action, Bob should receive a notification

Let's add a new scenario bottom of the "Ticket notifications" feature, inside `spec/features/ticket_notifications_spec.rb`:

Listing 320. spec/features/ticket_notifications_spec.rb

```
RSpec.feature "Users can receive notifications about ticket updates" do
  ...

  scenario "comment authors are automatically subscribed to a ticket" do
    fill_in "Text", with: "Is it out yet?"
    click_button "Create Comment"

    perform_enqueued_jobs

    click_link "Sign out"

    reset_mailer

    login_as(alice)
    visit project_ticket_path(project, ticket)
    fill_in "Text", with: "Not yet - sorry!"
    click_button "Create Comment"

    perform_enqueued_jobs

    expect(page).to have_content "Comment has been created."
    expect(unread_emails_for(bob.email).count).to eq 1
    expect(unread_emails_for(alice.email).count).to eq 0
  end
end
```

In this scenario, you're already logged in as Bob (courtesy of the `before` block in this feature). With Bob, you create a comment on the ticket, and then sign out. Then you clear the email queue - Alice should have received one when Bob commented, but we want to make sure she

receives no more after this point. You then sign in as Alice and create a comment, which should trigger an email to be sent to Bob, but not to Alice, because the users shouldn't receive notifications for their own actions!

On the final lines for this scenario, we're checking the number of unread emails for each user - Bob should have one from Alice's comment, and Alice should have none because she shouldn't get notified about her own comment.

When you run this scenario using `bundle exec rspec spec/features/ticket_notifications_spec.rb`, you'll see that Bob never receives an email from that final comment left by Alice.

```
1) Users can receive notifications about ticket updates comment authors
   are automatically subscribed to a ticket
   Failure/Error: expect(unread_emails_for(bob.email).count).to eq 1

     expected: 1
          got: 0

     (compared using ==)
```

This is failing on the step that checks if Bob has an email. You can therefore determine that Bob isn't subscribed to receive comment update notifications, as he should have been when he posted a comment. You need to add any commenter to the watchers list when they post a comment so that they're notified of ticket updates.

11.5.2. Automatically adding the commenter to the watchlist

To keep comment authors up to date with tickets, you'll automatically add them to the watchers list for that ticket when they post a comment. You currently do this when users create a new ticket, so you can apply the same logic to adding them to the list when they create a comment.

Let's do this by adding an additional line to the `create` action of `CommentsController`

Listing 321. app/controllers/comments_controller.rb

```
def create
  @comment = @ticket.comments.build(comment_params)
  @comment.author = current_user

  if @comment.save
    comment_notifier = CommentNotifier.new(@comment)
    comment_notifier.notify_watchers
    unless @ticket.watchers.exists?(current_user.id)
      @ticket.watchers << current_user
    end
```

By using `<<` on the watchers association, you can add the creator of this comment to the watchers for this ticket. This should mean that when a comment is posted to this ticket, any user who has posted a comment previously, and not only the ticket creator, will receive an email.

An important thing to note here is that if the user was already a watcher of the ticket, they won't be added as a watcher a second time here thanks to the `watchers.exists?` call we make.

Now that a comment's owner is automatically added to a ticket's watchers list, that should be enough to get the new scenario to pass. Find out by rerunning `bundle exec rspec spec/features/ticket_notifications_spec.rb`.

```
3 examples, 0 failures
```

Perfect! Now users who comment on tickets are added to the watchers list automatically, and the user who posts the comment isn't notified if they are already on that list.

Did you break anything by implementing this change? Have a look-see by running `bundle exec rspec`:

```
47 examples, 0 failures
```

Every test that you have thrown at this application is still passing, which is a great thing to see. Commit this change:

```
$ git add .
$ git commit -m "Automatically subscribe users to a ticket when they
  comment on it"
$ git push
```

You now have automatic subscriptions for ticket notifications when a user creates a ticket or posts a comment to one, but currently there's no way to switch notifications off. To implement this, you'll add an "Unwatch" button that, when clicked, will remove the user from the list of watchers for that ticket.

11.5.3. Unsubscribing from ticket notifications

You need to add a button to the ticket page to unsubscribe users from future ticket notifications. When you're done here, the ticket page will look like this:

Figure 57. The watch button

Along with implementing the ability to turn off notifications by clicking this button, you'll also add a way for users to turn on notifications, using what will effectively be the same button with a different label. This button will toggle users' watching status, which will allow them to subscribe to ticket notifications without creating their own ticket or posting a comment.

You'll implement the on/off functionality simultaneously by writing a feature in a new file at `spec/features/watching_tickets_spec.rb`. Start with the code from the following listing.

Listing 322. spec/features/watching_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can watch and unwatch tickets" do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project) }
  let(:ticket) do
    FactoryBot.create(:ticket, project: project, author: user)
  end

  before do
    ticket.watchers << user
    login_as(user)
    visit project_ticket_path(project, ticket)
  end
end
```

In this example, you create a single user, a project, and a ticket. Because this user created the ticket, they're automatically subscribed to watching this ticket, so they should see the "Unwatch" button on the ticket page. You'll test this by putting the scenario in the following listing underneath the `before` in this feature.

Listing 323. Ticket watch toggling

```
RSpec.feature "Users can watch and unwatch tickets" do
  ...
  scenario "toggles between unwatching and watching a ticket" do
    within(".watchers") do
      expect(page).to have_content user.email
    end

    click_link "Unwatch"
    message = "You are no longer watching this ticket."
    expect(page).to have_content(message)

    within(".watchers") do
      expect(page).to_not have_content user.email
    end
  end
end
```

In this scenario, you check that a user is automatically subscribed to the ticket by asserting that their email address is visible in the `#watchers` element on the page. When that user clicks the "Unwatch" button, they'll be told that they're no longer watching the ticket, and their

email will no longer be visible inside `#watchers`.

To begin to watch a ticket again, all the user has to do is click the "Watch" button, which you can also test by adding the following code to this scenario:

```
scenario "toggles between unwatching and watching a ticket" do
  ...
  click_link "Watch"
  expect(page).to have_content "You are now watching this ticket."

  within(".watchers") do
    expect(page).to have_content user.email
  end
end
```

See? That's how you'll test the watching/not-watching function simultaneously! You don't need to post a comment and test that a user is truly watching this ticket; you can instead check to see if a user's name appears in a list of all the watchers on the ticket page, which will look like this:

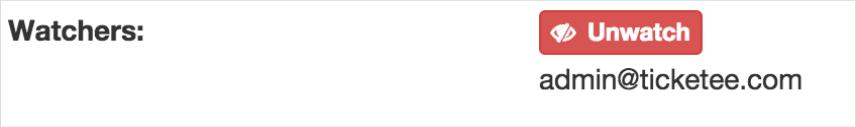


Figure 58. Who's watching

As usual, you'll see what you need to code right now to get your feature on the road to passing by running `bundle exec rspec spec/features/watching_tickets_spec.rb`. You'll see that it's actually the watchers list, indicated by Capybara telling you that it can't find that element:

```
1) Users can watch and unwatch tickets successfully
   Failure/Error: within("#watchers") do
   Capybara::ElementNotFound:
     Unable to find css "#watchers"
```

To get this feature to pass, you're going to need to add this new element with ID `watchers`. You can add it to `app/views/tickets/show.html.erb` at the bottom of the `attributes` table, by using the code in the following listing.

Listing 324. app/views/tickets/show.html.erb

```
<table class="attributes">
  ...

  <tr>
    <th>Watchers:</th>
    <td class="watchers">
      <%= @ticket.watchers.map(&:email).to_sentence %>
    </td>
  </tr>
</table>
```

You've added another element to our list here with the `class` attribute set to `watchers`, which is the element that your scenario looks for. In this row, you collect all the watchers' emails using `map`, and then you use `to_sentence` on that array. What this will do is turn the array of user's emails into a proper sentence, such as "alice@example.com, bob@example.com and corey@example.com".

When you have this element and you run your feature again with `bundle exec rspec spec/features/watching_tickets_spec.rb`, you'll see that your feature gets one step closer to passing by locating the user's email in the `.watchers` element, but now it can't find the "Unwatch" button:

```
1) Users can watch and unwatch tickets successfully
   Failure/Error: click_link "Unwatch"
   Capybara::ElementNotFound:
     Unable to find link "Unwatch"
```

This button will toggle the current user's watching status for this ticket, and the text and appearance will differ depending on whether the user is or isn't currently watching this ticket. In both cases, however, the button will go to the same action. Because so much of the code will be duplicated for both buttons, we can add the buttons to the view by using a helper method, changing the first few lines of the element to this:

Listing 325. app/views/tickets/show.html.erb

```
<dt>Watchers:</dt>
<dd class="watchers">
  <%= @ticket.watchers.map(&:email).to_sentence %>
</dd>

<tr>
  <th>Watchers:</th>
  <td class="watchers">
    <%= toggle_watching_button(@ticket) %>
    <%= @ticket.watchers.map(&:email).to_sentence %>
  </td>
</tr>
```

This `toggle_watching_button` helper will only appear in views for the `TicketsController`, so you should put the method definition in `app/helpers/tickets_helper.rb` inside the `TicketsHelper` module. Use the code from the following listing to define the method.

Listing 326. `toggle_watching_button` **inside** `TicketsHelper`

```
module TicketsHelper
  ...
  def toggle_watching_button(ticket)
    text = if ticket.watchers.include?(current_user)
      "Unwatch"
    else
      "Watch"
    end
    link_to text, watch_project_ticket_path(ticket.project, ticket),
      class: text.parameterize, method: :patch
  end
end
```

On the last line of the helper, we've used `link_to` to create a HTML link, but we've specified that the `method` of the link is `:patch`. This tells Rails to create a `PATCH` request when the link is clicked, rather than the standard `GET` request.

Inside the `link_to`, you use a new route helper that you haven't defined yet. When you run `bundle exec rspec spec/features/watching_tickets_spec.rb`, it will complain that this method is undefined when it tries to render the `app/views/tickets/show.html.erb` page:

```
1) Users can watch and unwatch tickets successfully
   Failure/Error: visit project_ticket_path(project, ticket)
   ActionView::Template::Error:
     undefined method `watch_project_ticket_path' for #<#<Class:...
   # ./app/helpers/tickets_helper.rb:21:in `toggle_watching_button'
```

This route helper points to a specific action on a project's ticket. You can define it in `config/routes.rb` inside the `resources :tickets` block, which itself is nested inside the `resources :projects` block, as shown in the following listing.

Listing 327. Adding `watch` route to `config/routes.rb`

```
Rails.application.routes.draw do
  ...
  resources :projects, only: [:index, :show, :edit, :update] do
    resources :tickets do
      collection do
        post :upload_file
      end

      member do
        patch :watch
      end
    end
  end
  ...
```

The purpose of `link_to` is to toggle the watch status of a single ticket, so you want to define a `member` route for your ticket resource. You put it inside the `tickets` resource, nested under the `projects` resource.

When you run your feature again using `bundle exec rspec spec/features/watching_tickets_spec.rb`, it will complain now because there's no `watch` action for your button to go to:

```
1) Users can watch and unwatch tickets successfully
   Failure/Error: click_link "Unwatch"
   AbstractController::ActionNotFound:
     The action 'watch' could not be found for TicketsController
```

You're almost done! Defining this `watch` action is almost the last thing you have to do. This action will add the user who visits it to a specific ticket's watcher list if they aren't already

watching it, or remove them if they are. To define this action, open `app/controllers/tickets_controller.rb` and under the `search` action, insert the code in the following listing.

Listing 328. `watch` **action inside** `TicketsController`

```
class TicketsController < ApplicationController
  ...
  def watch
    if @ticket.watchers.exists?(current_user.id)
      @ticket.watchers.destroy(current_user)
      flash[:notice] = "You are no longer watching this ticket."
    else
      @ticket.watchers << current_user
      flash[:notice] = "You are now watching this ticket."
    end

    redirect_to project_ticket_path(@ticket.project, @ticket)
  end
  ...
```

The first thing you need to notice about this method is that you don't define the `@ticket` variable before you use it on the first line of this method. That's because you can add this action to the list of actions that the `before_action :set_ticket`, runs on by changing the call in your controller.

```
class TicketsController < ApplicationController
  ...
  before_action :set_ticket, only: %i(show edit update destroy watch)
  ...
```

At the top of this action we need to check that the user is authorized to view this ticket. We don't want users being able to watch tickets that they're not permitted to view! If we didn't have this `authorize!` call in place it wouldn't matter anyway, as Pundit would throw its `Pundit::AuthorizationNotPerformedError` exception.

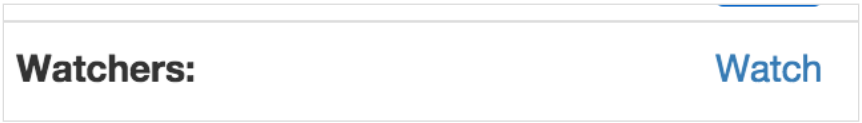
In this action, you use `exists?`, which will check if the given user is in the list of watchers. If they are, you'll use `watchers.destroy` to remove the watcher from the list. If they aren't on the watchers list, you'll use `watchers <<` to add them to the list of watchers.

The `watch` action now defines the behavior for a user to start and stop watching a ticket by clicking the button above the watchers list. When you run `bundle exec rspec`

`spec/features/watching_tickets_spec.rb`, it will pass:

```
1 example, 0 failures
```

Great! Now you have a way for users to toggle their `watch` status on any given ticket. When we load up a ticket's page in our browser, it could be a bit prettier, though:



Let's spend a little bit of time making the button a bit prettier, so it fits in with all of our other buttons and styles.

Because these styles will only be used on the ticket page, we should put them in `app/assets/stylesheets/tickets.scss`. Open that file, and put the following code at the bottom:

Listing 329. Styling the "Watch" and "Unwatch" buttons

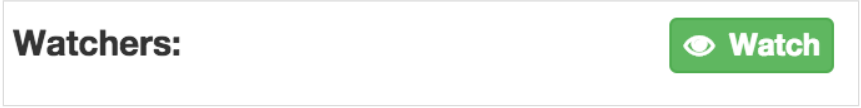
```
.watch,
.unwatch {
  @extend .btn, .btn-sm, .d-block;
  font-weight: bold;

  &:before {
    font-family: "FontAwesome";
    padding-right: 0.5em;
  }
}

.watch {
  @extend .btn-success;

  &:before {
    @extend .fa-eye;
  }
}

.unwatch {
  @extend .btn-danger;

  &:before {
    @extend .fa-eye-slash;
  }
}
```

We've styled the watch/unwatch button with Bootstrap's `btn` and `btn-xs` styles, so that they look the same as the tags and states also displayed in the `attributes` table. We've also given the two buttons different styles and colours - the "Watch" button is green and has an icon of an eye, and the "Unwatch" button is red with a crossed-out eye.

Because we're using the font-awesome icons here, we'll need to import the relevant CSS file for those icons at the top of this `tickets.scss` file:

```
@import "font-awesome";
```

When you refresh the page now, the buttons will be nicely styled:



Excellent! Make sure that everything else is still working by running `bundle exec rspec`. You should see the following output:

```
131 examples, 0 failures
```

Everything is still A-OK, which is good to see. Commit this change:

```
$ git add .
$ git commit -m "Add button so users can toggle watching on a ticket"
$ git push
```

You've now got a way for a user to start or stop watching a ticket. By watching a ticket, a user will receive an email when a comment is posted to the ticket. You're doing great in theoretically testing email, but you haven't yet configured your application to send emails out to the real world just yet. You'll see how to do that in the next chapter when we deploy the application to a production environment.

11.6. Summary

In this chapter, you learned how to send out your own kind of emails. Before that, however, you added two ways that users can subscribe to a ticket.

The first of these ways was an automatic subscription that occurred when a user created a ticket. Here, every time a comment was posted to a ticket, the owner of the ticket was notified through a simple email message.

The second of the two ways was to allow users to click a button to subscribe or unsubscribe to a ticket. This allowed all users, and not just those who created the ticket, to choose to receive emails when a comment is posted on the ticket. This way, all users can stay up to date on tickets they're interested in.

The next chapter involves deploying the application to a production environment and will build upon some of the things we built in this chapter. We'll even be sending real emails!

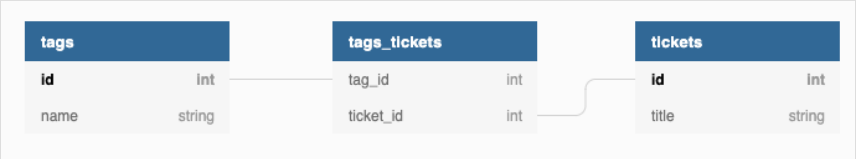
[68] https://github.com/email-spec/email-spec/
[69] Rails generates the table name by putting the two joined table names alphabetically, separated by an underscore.
[70] The default setting for `delivery_method` is `:smtp`. This will direct Action Mailer connect to an SMTP server running on `localhost` - that is, the machine Rails is running on. It is very unlikely you have an SMTP server running on your local machine.

Chapter 12. Tagging

In chapter 10, you saw how to give your tickets states (eg. "New", or "Open") so that their progress can be indicated. In this chapter, you'll see how to give your tickets tags. Tags are useful for grouping similar tickets together into iterations[71] or similar feature sets. Without tags, you could crudely group tickets together by setting a ticket's title to something such as "Tag - [name]." This method, however, is messy and difficult to sort through. Having a group of tickets with the same tag will make them much, much easier to find.

To manage tags, you'll set up a `Tag` model, which will have a `has_and_belongs_to_many` association to the `Ticket` model. This is a new type of association that we haven't seen before, one that combines both a `has_many` and a `belongs_to` association.

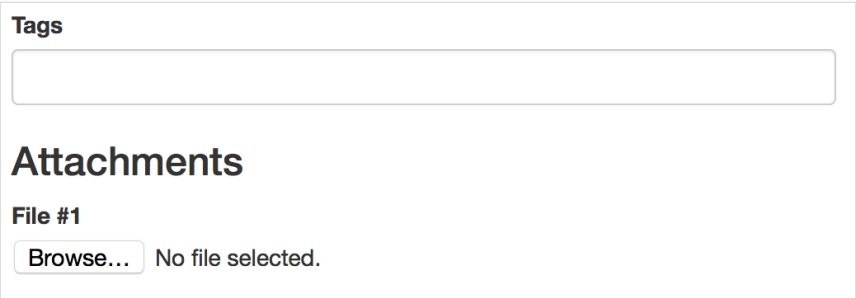
The way that a `has_and_belongs_to_many` association works is through a database feature called a join table. A join table's sole purpose is to join together the two tables whose keys it has. In this case, the two tables are the `tickets` and `tags` tables. Here's a small illustration showing the three tables that will be involved here:



The `has_and_belongs_to_many` association will allow us to store multiple tickets against a tag, as well have multiple tags on a ticket. It could be said that a "tag has and belongs to many tickets" or "a ticket has and belongs to many tags".

To add tags to our tickets, we're going to have an input field that will allow us to add multiple tags:

TODO: Show tags field in context of whole form, with multiple tags already added



Additional tags can also be added on a comment, with a text field providing the interface to add new tags. When a ticket is created, you'll show these tags underneath the ticket's

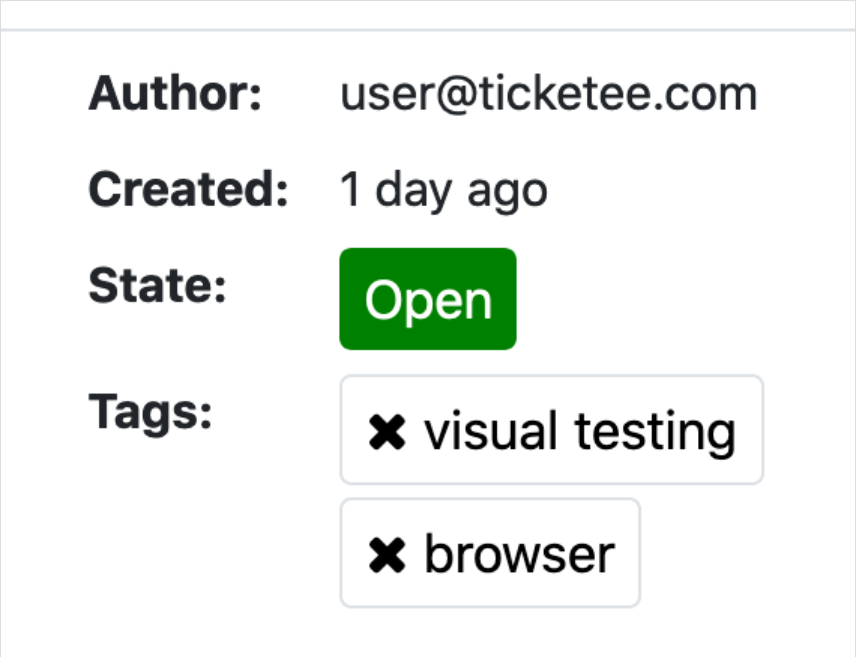


Figure 59. Tags for a ticket

When a user clicks on a tag, they'll be taken to a page where they can see all tickets with that particular tag. The little "X" next to the tag will allow users to remove tags quickly from an issue.

Finally, you'll implement a way to search for tickets that match a state, a tag, or both, by using a gem called Searcher. The query will look like `tag:"Iteration 1" state:Open`.

That's all there is to this chapter! You'll be adding tags to Ticketee, which will allow you to easily group and sort tickets. Let's dig into the first feature: adding tags to a new ticket.

12.1. Creating tags

Tags are useful for making similar tickets easy to find and manage. In this section, you'll create the interface for adding tags to a new ticket by adding a new field to the new ticket page and defining a `has_and_belongs_to_many` association between the `Ticket` model and the not-yet-existing `Tag` model.

12.1.1. The "Creating Tags" feature

First you're going to add a text field beneath the description field on the "New Ticket" page for tags, like you saw earlier.

The words you enter into this field will become the tags for this ticket, and you should see them on the ticket page. To do this, add a scenario that creates a new ticket with tags at the bottom of `spec/features/creating_tickets_spec.rb`, as shown in the following listing:

Listing 330. spec/features/creating_tickets_spec.rb

```
RSpec.feature "Users can create new tickets" do
  ...
  scenario "with associated tags" do
    fill_in "Name", with: "Non-standards compliance"
    fill_in "Description", with: "My pages are ugly!"
    fill_in "Tags", with: "visual testing, browser"
    click_button "Create Ticket"

    expect(page).to have_content "Ticket has been created."
    within(".ticket .tags") do
      expect(page).to have_content "browser"
      expect(page).to have_content "visual testing"
    end
  end
end
```

When we list tags for an issue, we'll separate them with a comma. This will allow us to specify multiple words for a tag such as our "visual testing" tag.

When you run the new spec using `bundle exec rspec spec/features/creating_tickets_spec.rb`, it will fail, declaring that it can't find the "Tags" field. Good! It's not there yet.

```
1) Users can create new tickets with associated tags
   Failure/Error: fill_in "Tags", with: "browser visual"
   Capybara::ElementNotFound:
     Unable to find field "Tags"
```

You're going to take the data from this field, process each word into a new `Tag` object, and then link the tags to the ticket when the ticket is created. You'll use a `text_field` tag to render the "Tags" field this way, but unlike the `text_fields` that you've used previously, this

one will not be tied to a database field.

To define this field, put the following code underneath the `description` field in `app/views/tickets/_form.html.erb`:

Listing 331. app/views/tickets/_form.html.erb

```
<%= bootstrap_form_with(model: [project, ticket], local: true, label_errors: true) do |form|
%>
  <%= form.text_field :name %>
  <%= form.text_area :description %>
  <% upload_url = upload_file_project_tickets_path(project) %>
  <div class="dropzone uploader mb-2" data-url="<%= upload_url %>"></div>

  <div class="form-group">
    <%= label_tag :tag_names, "Tags" %>
    <%= text_field_tag :tag_names, "", class: "form-control" %>
  </div>

  <%= form.primary %>
<% end %>
```

We're using a `text_field_tag` here, as `tag_names` isn't an attribute on our tickets. If we attempted to pass this through with `name` and `description`, it would raise an exception because `tag_names` is not a permitted attribute in the `TicketsController`. We work around this by using `text_field_tag`, which will make the tags available as `params[:tag_names]` in the action that receives this form's data.

Because we're not using the form builder here—the `form` object made available by `bootstrap_form_with`, we need to manually write our label and input, along with the correct CSS classes from Bootstrap. This will make our tag name field appear correctly on our form:

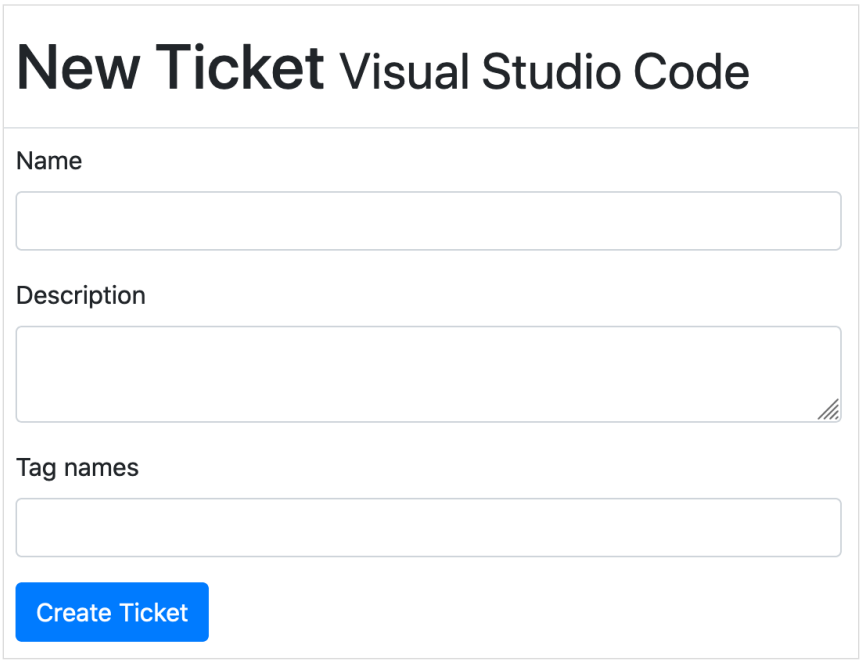


Figure 60. Ticket form with tags

When you rerun this scenario with `bundle exec rspec spec/features/creating_tickets_spec.rb`, it no longer complains about the missing "Tags" field, telling you instead that it cannot find the tags within the attributes of a ticket:

```
1) Users can create new tickets with associated tags
   Failure/Error: within(".ticket .attributes .tags") do
   Capybara::ElementNotFound:
     Unable to find css ".ticket .attributes .tags"
```

This test is looking for tags here on our tickets' show page:

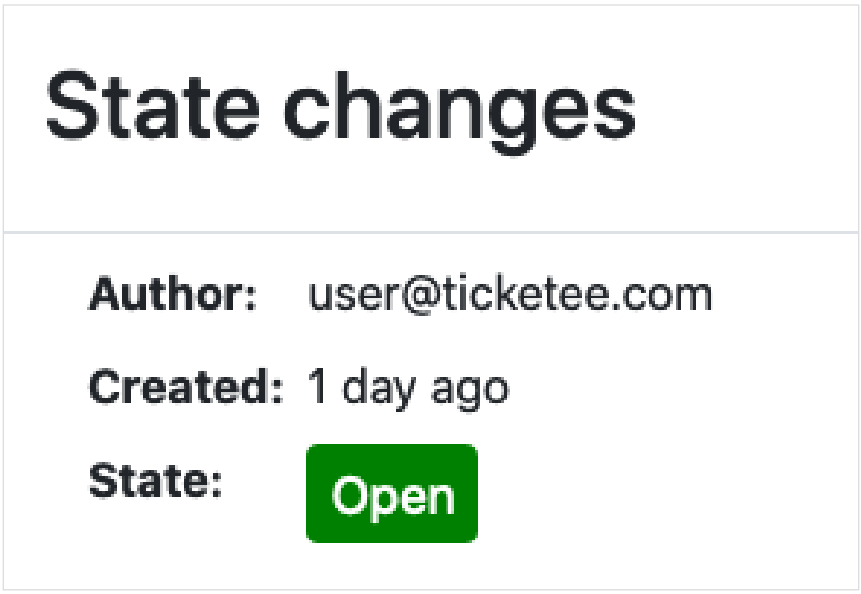


Figure 61. Tags are missing from here

You now need to define this `.tags` element inside the `.ticket .attributes` element on the ticket's page, so that this part of the scenario will pass. This element will contain the tags for your ticket, which your scenario will assert are actually visible.

12.1.2. Showing tags

You can add this new element, with its `class` attribute set to `tags`, to `app/views/tickets/show.html.erb` by adding these lines underneath the state:

Listing 332. app/views/tickets/show.html.erb

```
<table class="attributes">
  ...


  <% if @ticket.tags.any? %>
    <tr>
      <th>Tags:</th>
      <td class="tags"><%= render @ticket.tags %></td>
    </tr>
  <% end %>

  <tr>
    <th>Watchers:</th>
    <td class="watchers">
      <%= toggle_watching_button(@ticket) %>
      <%= @ticket.watchers.map(&:email).to_sentence %>
    </td>
  </tr>
</table>
```

This creates the `.ticket .attributes .tags` element that your feature is looking for, and it will render the soon-to-be-created `app/views/tags/_tag.html.erb` partial for every element in the also-soon-to-be-created `tags` association on the `@ticket` object.

So which of these two do you create next? Do you create the partial? Or do you create the `tags` association? If you run your scenario again, you'll see that it can't find the `tags` method for a `Ticket` object:

```
1) Users can create new tickets with associated tags
   Failure/Error: click_button "Create Ticket"
   ActionView::Template::Error:
     undefined method `tags' for #<Ticket:0x007fb2b7f40f58>
```

Looks like our test wants the association, so let's create that first. You'll be defining this `tags` method with a `has_and_belongs_to_many` association between `Ticket` objects and `Tag` objects. This method will be responsible for returning a collection of all the tags associated with the given ticket, much like a `has_many` would. The difference is that this method works in the opposite direction as well, allowing you to find out what tickets have a specific tag.

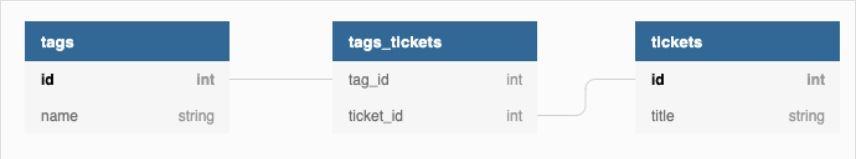
12.1.3. Defining the `tags` association

You can define the `has_and_belongs_to_many` association on the `Ticket` model by placing this line after the `has_many` definitions inside your `Ticket` model:

```
class Ticket < ActiveRecord::Base
  ...
  has_many :comments, dependent: :destroy
  has_and_belongs_to_many :tags
  ...
```

This association will rely on a join table that doesn't yet exist, called `tags_tickets`. By default, Rails will assume that the name of this join table is the combination, in alphabetical order, of the two tables you want to join. This table contains only two fields—one called `ticket_id` and one called `tag_id`—which are both foreign keys for tags and tickets. The join table will easily facilitate the union of these two tables, as it will have one record for each tag that links to a ticket, and vice versa.

Here's that table diagram from earlier again to help you visualise what this association looks like in our database:



We're also passing the `uniq: true` option to our association, meaning that only unique tags will be retrieved for each ticket. You could list the tag `bug` twice when creating a ticket, like `bug bug`, and two records will be created in the join table, but when you ask for a ticket's tags, you'll only get one `bug` tag back.

When you rerun your scenario, you'll be told that there's no constant called `Tag` yet:

```
1) Users can create new tickets with associated tags
   Failure/Error: click_button "Create Ticket"
   ActionView::Template::Error:
     uninitialized constant Ticket::Tag
```

In other words, there is no `Tag` model yet. You should define this now if you want to go any further.

12.1.4. The `Tag` model

Your `Tag` model will have a single field called `name`, which should be unique. To generate this model and its related migration, run the `rails` command like this:

```
$ rails g model tag name:string --timestamps false
```

We're not wanting to track when tags were created or updated, so we've passed the `timestamps` option to this migration, with the value of `false`. This will skip adding the `t.timestamps null: false` line inside our migration.

This is what that migration will look like:

Listing 333. db/migrate/[timestamp]_create_tags.rb

```
class CreateTags < ActiveRecord::Migration[6.1]
  def change
    create_table :tags do |t|
      t.string :name
    end
  end
end
```

Before you run that migration, we'll generate another one, for the join table for tags and tickets. The Rails migration generator can do this for you, with the following command:

```
$ rails g migration create_join_table_tags_tickets tag ticket
```

The actual name you give the migration is irrelevant; it just has to have the words "join table" in it, like ours does. We've specified the two models that should be joined together in this join table, and it will generate a migration with the following contents:

Listing 334. db/migrate/[timestamp]_create_join_table_tags_tickets.rb

```
class CreateJoinTableTagsTickets < ActiveRecord::Migration[6.1]
  def change
    create_join_table :tags, :tickets do |t|
      t.index [:tag_id, :ticket_id], unique: true
      t.index [:ticket_id, :tag_id]
    end
  end
end
```

Uncomment the two commented-out lines, so that the two indexes will be created on the table. Add `unique: true` to one of these `index` lines, so that only unique combinations of `tag_id` and `ticket_id` will be allowed.

This migration will create the join table, and create two indexes that will speed up lookups going both ways - looking up tags for a ticket, and looking up tickets for a tag. You can now run the two migrations that we have just created:

```
$ rails db:migrate
```

This will create the `tags` and `tags_tickets` tables specified in your two most recently created migrations.

When you run this scenario again with `bundle exec rspec spec/features/creating_tickets_spec.rb`, it's now satisfied that the `tags` method is defined, and it has now gone back to claiming it can't find the `.ticket .attributes .tags` element on the page:

```
1) Users can create new tickets with associated tags
   Failure/Error: within(".ticket .attributes .tags") do
   Capybara::ElementNotFound:
     Unable to find css ".ticket .attributes .tags"
```

This failure is because you're not doing anything to associate the text from the "Tags" field to the ticket you've created. This means that there are no tags to display, so this element isn't being displayed. Let's fix this up by processing that `tag_names` field in the controller.

12.1.5. Displaying a ticket's tags

You're now going to take the names of the tags that are passed in to the `tag_names` parameter and turn them into objects of the `Tag` class.

To make this happen, let's go into the `create` action in `TicketsController` and add some extra code to parse a ticket's tags:

Listing 335. app/controllers/tickets_controller.rb

```
def create
  @ticket = @project.tickets.build(ticket_params)
  @ticket.author = current_user
  if params[:attachments].present?
    @ticket.attachments.attach(params[:attachments])
  end
  @ticket.tags = params[:tag_names].split(",").map do |tag|
    Tag.find_or_initialize_by(name: tag.strip)
  end
```

This new code will take whatever's in `params[:tag_names]`, split it up by its commas, and then use `Tag.find_or_initialize_by` to either find an existing `Tag` object or build a new one that matches the tag's name.

This code will will create the tags that you're displaying on the `app/views/tickets/show.html.erb` view, by using the `render` method:

```
<%= render @ticket.tags %>
```

When you run this scenario again by running `bundle exec rspec spec/features/creating_tickets_spec.rb`, you'll see this render is now failing with an error:

```
1) Users can create new tickets with associated tags
   Failure/Error: click_button "Create Ticket"
   ActionView::Template::Error:
     Missing partial tags/_tag with {:locale=>[:en], :formats=>...
```

This error is happening now because `@ticket.tags` actually contains some tickets, and the `render` call is attempting to render them. That's just like back in chapter 10 when you used this line:

```
<%= render @ticket.state %>
```

Rails will render a partial for the given objects based on the class name of the object. In the case of `@ticket.state`, that class was `State`, so the `app/views/states/_state` partial was used.

When you iterate over a collection of objects, as you're doing with `@ticket.tags`, Rails will pick the first object from that collection and then render a partial for each of the objects based on the class of that first element. This partial is going to live in `app/views/tags/_tag.html.erb`, because the class for the first object is `Tag`.

The next step is to write the tag partial that your feature has complained about. Put the following code in a new file called `app/views/tags/_tag.html.erb`:

Listing 336. Adding a HTML representation of a tag

```
<div class="tag">
  <%= link_to "<span></span>".html_safe, "#", class: "remove",
    title: "remove" %>
  <%= tag.name %>
</div>
```

By wrapping the tag name in a `div` element with the class of `tag`, we can easily add some styles to it. We know the tags will also need a link in the future, from the requirements we stated earlier (being able to remove a tag from a ticket), so we've added in a dummy link now, just with a `href` of `#`.

Let's add some styles for our new tags. We'll leverage some more Bootstrap styles, by adding the following to `app/assets/stylesheets/tickets.scss`:

Listing 337. Adding a CSS representation of a tag

```
.tag {
  @extend .state, .mr-1, .mb-1, .border;
  color: black;
  font-size: 85%;

  a {
    color: black;
    &.remove {
      font-family: "FontAwesome";
      @extend .fa-close;
      text-decoration: none;
    }
  }
}
```

This style extends the `.state` style higher up in this file, which means that our tags will have very similar styles to the states. We override a few things here, like setting the background to Bootstrap's `$blue` colour, and we add some styling for the removal button.

Defining this partial will stop the "Missing template" error from happening. When you run your scenario again with `bundle exec rspec spec/features/creating_tickets_spec.rb`, it should now pass:

```
7 examples, 0 failures
```

Great! This scenario is now complete. When a user creates a ticket, they are now able to assign tags to that ticket, and those tags will display along with the ticket's information on the `show` action for `TicketsController`. The tag display was shown earlier:

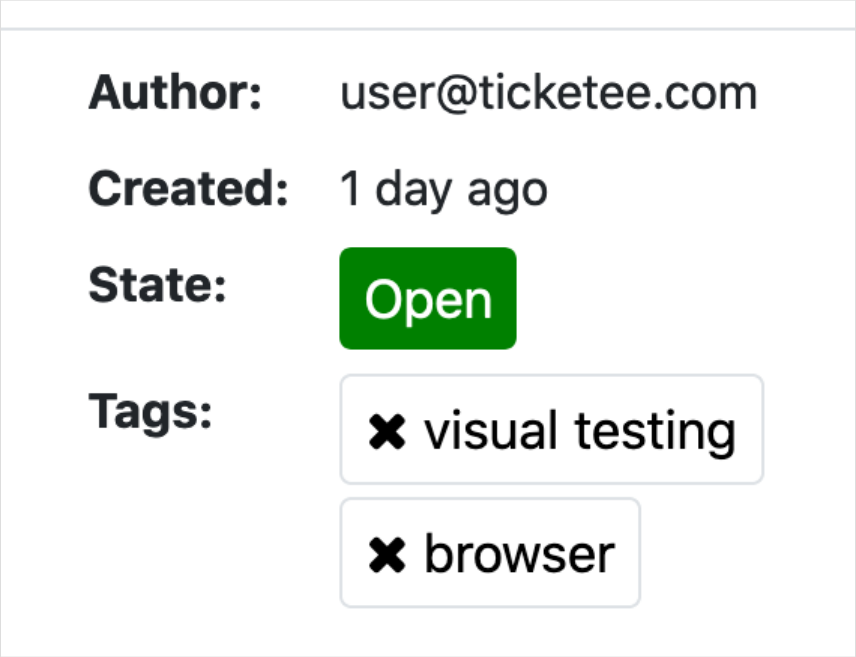


Figure 62. Look ma, a tag!

You can now commit this change, but before you do, ensure that you haven't broken anything by running `bundle exec rspec`.

```
50 examples, 0 failures, 1 pending
```

Good to see that nothing's blown up this time. There's one pending spec located in `spec/models/tag_spec.rb`, and because there's nothing else in that file, it's safe to delete it. Go ahead and do that now. After you're done, a rerun of `bundle exec rspec` will produce this lovely green output:

```
49 examples, 0 failures
```

Yay! Commit your changes.

```
$ git add .
$ git commit -m "Users can tag tickets upon creation"
$ git push
```

Now that users can add a tag to a ticket when that ticket is being created, you should also let them add tags to a ticket when they create a comment as well. When a ticket is being

discussed, new information may arise that will require another tag to be added to the ticket and group it into a different set. A perfect way to let your users do this would be to let them see the list of tags and to be able to edit them when they're editing the details of a ticket.

12.2. Editing tags

We're now going to make it so that users can update a tag's tickets when they're updating a ticket. When a user edits a ticket, they'll be able to see the existing tags for that ticket and they'll be able to add new tags:

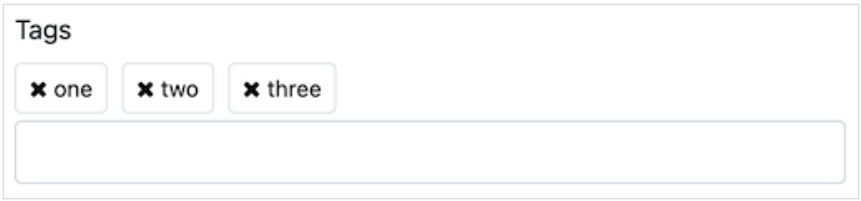


Figure 63. Adding additional tags

When a user clicks "Update Ticket", our application should preserve the existing tags for that ticket and add in the new tags.

12.2.1. Displaying existing tags

So with that in mind, let's start out by writing a test that makes sure that existing tags are displayed on the form:

Listing 338. spec/features/editing_tickets_spec.rb

```
context "when the ticket has tags" do
  before do
    ticket.tags << FactoryBot.create(:tag, name: "Visual Testing")
    ticket.tags << FactoryBot.create(:tag, name: "Browser")
  end

  it "sees existing tags on edit form" do
    click_link "Edit Ticket"
    within(".tags") do
      expect(page).to have_content("Visual Testing")
      expect(page).to have_content("Browser")
    end
  end
end
```

This test uses a new factory called tag to create new tags for our ticket. Let's set up this factory:

Listing 339. spec/factories/tags.rb

```
FactoryBot.define do
  factory :tag do
    sequence(:name) { |n| "Tag #{n}" }
  end
end
```

This factory will generate unique tag names using Factory Bot's `sequence` method, unless we specify them.

Now that we have this factory, we can set about running our test. Let's run it now with `bundle exec rspec spec/features/editing_tickets_spec.rb`. This test will fail because it cannot find the `.tags` element on the tickets form.

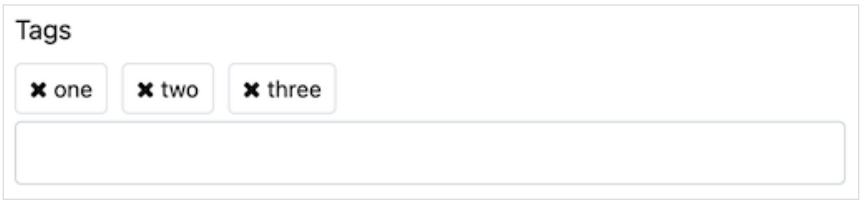
```
  1) Users can edit existing tickets when the ticket has tags sees existing tags on edit form
     Failure/Error:
       within(".tags") do
         expect(page).to have_content("Visual Testing")
         expect(page).to have_content("Browser")
       end

     Capybara::ElementNotFound:
       Unable to find css ".tags"
     # ./spec/features/editing_tickets_spec.rb:41
```

Let's go over to `app/views/tickets/_form.html.erb` and change the `form-group` element for the tags to include the existing tags for a ticket:

Listing 340. app/views/tickets/_form.html.erb

```
<div class='form-group'>
  <div><%= label_tag :tag_names, "Tags" %></div>
  <%= render @ticket.tags %>
  <%= text_field_tag :tag_names, "", class: "form-control" %>
</div>
```

This change will make the existing tags appear above the tag field:



In order for our test to be able to find this element, we will also need to change the `class` of this `div` to include `tags` as well:

Listing 341. app/views/tickets/_form.html.erb

```
<div class='form-group tags'>
  <div><%= label_tag :tag_names, "Tags" %></div>
  <%= render @ticket.tags %>
  <%= text_field_tag :tag_names, "", class: "form-control" %>
</div>
```

These two changes should be enough to get our test to run. Let's try it with `bundle exec rspec spec/features/editing_tickets_spec.rb`:

```
3 examples, 0 failures
```

Great! We can now see tags appearing on the edit screen for a form. Here's what users will now see:

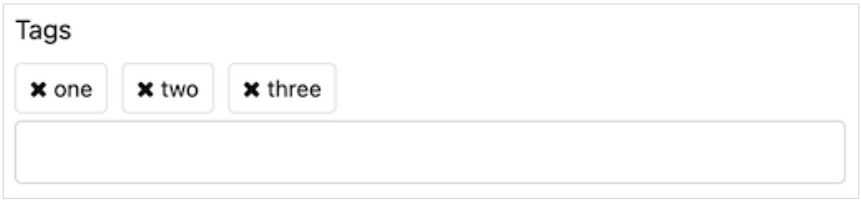


Figure 64. Adding additional tags

Let's move onto handling new tags.

12.2.2. Adding tags to an existing ticket

If you use the application right now to edit a ticket and add additional tags through the "Tags" field on the form... nothing happens! This is because we don't have any logic to handle tags in the `update` action of `TicketsController`. The only place where this logic exists currently is in the `create` action:

Listing 342. app/controllers/tickets_controller.rb

```
@ticket.tags = params[:tag_names].split(",").map do |tag|
  Tag.find_or_initialize_by(name: tag.strip)
end
```

We need the same sort of logic within the `update` logic to so that new tags can be added to an existing ticket. Before we get there though, let's write a test that asserts the behaviour we want:

Listing 343. spec/features/editing_tickets_spec.rb

```
context "when the ticket has tags" do
  before do
    ticket.tags << FactoryBot.create(:tag, name: "Visual Testing")
    ticket.tags << FactoryBot.create(:tag, name: "Browser")
  end

    ...

  it "can add new tags to a ticket" do
    click_link "Edit Ticket"
    fill_in "Tags", with: "regression, bug"
    click_button "Update Ticket"
    expect(page).to have_content("Ticket has been updated.")

    within(".ticket .attributes .tags") do
      expect(page).to have_content("Visual Testing")
      expect(page).to have_content("Browser")
      expect(page).to have_content("regression")
      expect(page).to have_content("bug")
    end
  end
end
```

This new test asserts that when we edit a ticket and add two new additional tags to it, that we will will then see these listed alongside the existing tags for the ticket.

When we run this test with `bundle exec rspec spec/features/editing_tickets_spec.rb` we'll see that it fails to see these new tags:

```
1) Users can edit existing tickets when the ticket has tags can add new tags to a ticket
    Failure/Error: expect(page).to have_content("regression")
      expected to find text "regression" in "Visual Testing\nBrowser"
```

The test is currently only seeing the tags that the ticket has. Now that we've been able to reproduce the behaviour we saw in the application by writing a test, let's set about fixing this.

Let's go into `app/controllers/tickets_controller.rb` and change the top of the `update` action to process tags too:

Listing 344. app/controllers/tickets_controller.rb

```
def update
  if @ticket.update(ticket_params)
    @ticket.tags << params[:tag_names].split(",").map do |tag|
      Tag.find_or_initialize_by(name: tag.strip)
    end
```

This code is almost the same as the code in the `create` action, but in this code we're using `<<` to add the tags that we find to the list of existing tags.

When we run our test again, we will see that this code has fixed our test:

```
4 examples, 0 failures
```

That was easy! But is this the best we can do? The code is the same between the `update` and `create` actions. Let's spend a bit of time tidying this up.

12.2.3. A quick refactor

When we're writing tests and we have them all passing, we should take the time to re-visit our code and to see if anything can be tidied up. This is a process referred to by a mantra: Red, Green, Refactor.

- **First** you write a failing test.
- **Then** you make it green by making it pass.
- **Finally** you look at the code and see if it can be improved.

The test is there in place to ensure that the behaviour of the code remains the same, even

through a refactor. It provides us an assurance of that fact.

So with that in mind, let's do the Refactor step now by moving this common logic out of the `create` and `update` actions of `TicketsController` into another method at the bottom of our controller:

Listing 345. app/controllers/tickets_controller.rb

```
private

def processed_tags
  params[:tag_names].split(",").map do |tag|
    Tag.find_or_initialize_by(name: tag.strip)
  end
end
```

This method wraps the shared logic that we have in `create` and `update`. We can now change both of these actions to use this `processed_tags` method instead:

Listing 346. app/controllers/tickets_controller.rb

```
def create
  @ticket = @project.tickets.build(ticket_params)
  @ticket.author = current_user
  @ticket.tags = processed_tags
  ...
end

...

def update
  if @ticket.update(ticket_params)
    @ticket.tags << processed_tags
    ...
  end
  ...
end
```

This really tidies up our `create` and `update` actions! Let's make sure that these two actions are still working. First, we'll run `bundle exec rspec spec/features/creating_tickets_spec.rb`:

```
5 examples, 0 failures
```

That's great! We're still able to create tickets successfully in our application.

Next up, let's see what `bundle exec rspec spec/features/editing_tickets_spec.rb` does:

```
5 examples, 0 failures
```

Those examples are still passing too. This means that our refactoring was successful!

This finishes up our work with adding tags to existing tickets. Let's make a commit here:

```
git add .
git commit -m "Make tags attachable to existing tickets"
git push
```

12.3. Deleting a tag

We've now been able to add tags when creating a new ticket, and when editing an existing ticket, but what if we wanted to remove a tag?

Removing a tag from a ticket is helpful because a tag may become irrelevant over time. Say you tagged a ticket as "v0.1" for your project, but that milestone is complete and the ticket isn't, so it needs to be moved to "v0.2." You need a way to delete the old tag. With the ability to delete tags, you have some assurance that people will clean up tags when they need to.

To let users delete a tag, we added a placeholder cross to the left of each of the tags:

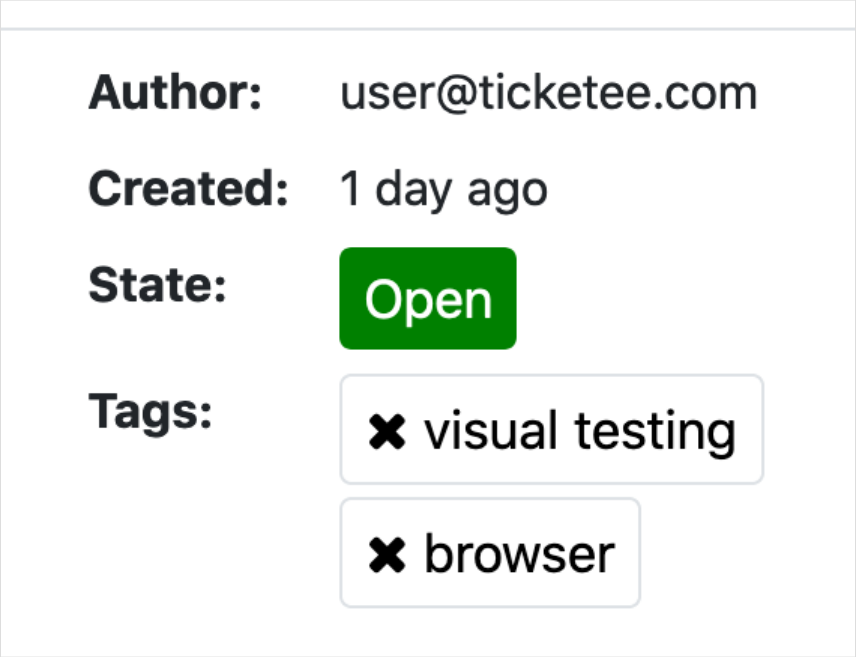


Figure 65. X marks the spot

Right now, this cross doesn't do anything. That changes in this part of the chapter!

When this cross is clicked, the tag will disappear through the magic of JavaScript. Rather than making a request to the action to delete the tag and then redirect back to the ticket page, the JavaScript will remove the tag's element from the page and make an asynchronous behind-the-scenes request to the controller action to delete the tag.

12.3.1. Testing tag deletion

We'll start by writing a feature to verify what should happen when we delete a tag from a ticket. Create a new file at `spec/features/removing_tags_spec.rb` and put the following code in it.

Listing 347. spec/features/removing_tags_spec.rb

```
require "rails_helper"

RSpec.feature "Users can remove unwanted tags from a ticket" do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project) }
  let(:ticket) do
    FactoryBot.create(:ticket,
      project: project,
      author: user,
      tags: [FactoryBot.create(:tag, name: "This Tag Must Die")]
    )
  end

  before do
    login_as(user)
    visit project_ticket_path(project, ticket)
  end

  scenario "successfully", js: true do
    within tag("This Tag Must Die") do
      click_link "Remove tag"
    end
    expect(page).to_not have_content "This Tag Must Die"
  end
end
```

In this feature, you create a new user and sign in as them. Then you create a new project and give the user the role of "manager" on it, which means the user can modify the tags of any of the project's tickets. There's a ticket already created with a tag name called "This Tag Must Die", and the final couple of lines delete that tag and verify that it's gone.

When you run this feature using `bundle exec rspec spec/features/removing_tags_spec.rb`, you'll get this error:

```
1) Users can remove unwanted tags from a ticket successfully
   Failure/Error: within tag("This Tag Must Die") do
   NoMethodError:
     undefined method `tag' for #<RSpec::ExampleGroups::...
```

Alright. Time to implement this bad boy.

12.3.2. Adding a link to delete the tag

We've used this method called `tag` in our spec, but the method doesn't exist. This method should be a finder - we're calling it to find an element on the page, to then scope all of our future calls by. We want to use this method to find the particular HTML element for the tag in question; we can then click the little cross to delete the tag, inside that element.

We've written finders for Capybara before - inside the file `spec/support/capybara_finders.rb`. There's one there called `list_item`, we'll add another for `tag`. Underneath the `list_item` method, define another one to find tags:

Listing 348. Adding another Capybara finder

```
module CapybaraFinders
  ...
  def tag(content)
    find("div.tag", text: content)
  end
end
```

This will find the `div` element with the class of `tag` that we defined earlier, to display our tags nicely. And we're filtering the list of tags by only selecting the one with the right text - the name of the tag, in our case "This Tag Must Die".

The "remove" link inside that tag needs to trigger an asynchronous request to an action that will remove a tag from a ticket. This request will be asynchronous so that a user removing a tag doesn't have to wait for a full-page refresh before they can carry on doing whatever they were doing before.

Let's change the link to remove a tag to this:

Listing 349. app/views/tags/_tag.html.erb

```
<div class="tag">
  <%= link_to "<span></span>".html_safe,
    ticket_remove_tag_path(ticket, tag), method: :delete, remote: true,
    class: "remove", title: "Remove tag" %>
  <%= tag.name %>
</div>
```

You've also filled out the contents of the link, so it's no longer a dummy link to "#". You use the

`:remote` option for the `link_to` to indicate to Rails that you want this link to be an asynchronous request—a request that will happen in the background and one that will not cause the page to reload.

Because this is a destructive action, you use the `:delete` method. You've used this previously to call destroy actions, but the `:delete` method is not exclusive to the destroy action, so you can use it here as well.

The final option, `:title`, lets you define the title for this link. You've set that to be "remove", and this is what Capybara will use to find the actual link to click it. Capybara supports following links by their internal text, or their `name`, `id` or `title` attributes.

When you run your feature with `bundle exec rspec spec/features/removing_tags_spec.rb`, you'll see that it fails with this error:

```
1) Users can remove unwanted tags from a ticket successfully
   Failure/Error: Unable to find matching line from backtrace
   ActionView::Template::Error:
     undefined local variable or method `ticket' for ...
   # ./app/views/tags/_tag.html.erb:2:in ...
```

Line 2 in our `_tag.html.erb` partial refers to the `ticket` variable, which doesn't exist within our partial. We need to have a reference to the ticket for the link, otherwise we won't know what ticket to remove the tag from!

In the parent `app/views/tickets/show.html.erb` view we have a `@ticket` variable, to display the details of the current ticket - we should pass that variable into the partial as a local.

Inside `app/views/tickets/show.html.erb`, modify the line that renders the tags for a ticket, to also provide the `@ticket` variable to the partial:

Listing 350. Part of `app/views/tickets/show.html.erb`, for displaying tags

```
<table class="attributes">
  ...
  <% if @ticket.tags.any? %>
    <tr>
      <th>Tags:</th>
      <td class="tags"><%= render @ticket.tags, ticket: @ticket %></td>
    </tr>
  <% end %>
</table>
```

The other place where need to make this change is in `app/views/tickets/_form.html.erb` where we do the same rendering:

Listing 351. app/views/tickets/_form.html.erb

```
<%= render @ticket.tags, ticket: @ticket %>
```

The `@ticket` variable will now be available inside the tag partial as the local variable `ticket`. Now when you re-run your spec with `bundle exec rspec spec/features/removing_tags_spec.rb`, you'll get a different error:

```
1) Users can remove unwanted tags from a ticket successfully
   Failure/Error: Unable to find matching line from backtrace
   ActionView::Template::Error:
     undefined method `ticket_remove_tag_path' for ...
   # ./app/views/tags/_tag.html.erb:4:in ...
```

This error is coming up because you haven't defined the route to the `remove` action yet. You can define this route in `config/routes.rb` inside the `resources :tickets` block, morphing it into this:

Listing 352. Defining a route for removing tags from tickets

```
Rails.application.routes.draw do
  ...
  scope path: "tickets/:ticket_id", as: :ticket do
    resources :comments, only: [:create]
    delete "tags/remove/:id", to: "tags#remove", as: :remove_tag
  end
  ...
end
```

We've added a new route here using the `delete` method. This method defines a route that will go an action called `remove` inside a controller called `TagsController`. We'll create that controller in a moment. The `as` option on the end here makes our routing helper `ticket_remove_tag_*`.

Now we'll be able to render the page without error, the route we've provided will be recognized by Rails and generate a URL that looks something like "/tickets/1/tags/2/remove". Now when you re-run the test, you'll get a different error:

```
1) Users can remove unwanted tags from a ticket successfully
   Failure/Error: Unable to find matching line from backtrace
   ActionController::RoutingError:
     uninitialized constant TagsController
```

The spec is loading the ticket page correctly, finding the "remove" link, and clicking it. Now we need a controller to process the actual removal of the tag from the ticket, so let's generate a new controller called `TagsController`, as that's what the error message is looking for.

```
$ rails g controller tags
```

Now that you have a controller to define your action in, open `app/controllers/tags_controller.rb` and define the `remove` action in it like this:

Listing 353. Removing a tag from a ticket

```
class TagsController < ApplicationController
  def remove
    @ticket = Ticket.find(params[:ticket_id])
    @tag = Tag.find(params[:id])

    @ticket.tags.destroy(@tag)
    head :ok
  end
end
```

In this action, you find the ticket based on the ID passed through as `params[:ticket_id]`, and the tag based on the ID passed through as `params[:id]`. You can then use `destroy`, which is a method provided by the `tags` association[72] to remove `@tag` from the list of tags in `@ticket.tags`. Finally, you can send the simple header `:ok`, which will return a `200 OK` status to your browser, signaling that everything went according to plan.

When you rerun your scenario with `bundle exec rspec spec/features/removing_tags_spec.rb`, it will successfully click the link, but the tag is still there:

```
1) Users can remove unwanted tags from a ticket successfully
   Failure/Error: expect(page).to_not have_content "This Tag Must Die"
     expected not to find text "This Tag Must Die" in "Ticketee Home ..."
```

Your tag is unassociated from the ticket but not removed from the page, so your feature is still failing. The request is made to delete the ticket, but there's no code currently that removes the tag from the page. Let's add that code now.

12.3.3. Removing the tag from the page

You're removing a tag's association from a ticket, but you're not yet showing people that it has happened, on the page. We're making the request to remove the tag asynchronously via JavaScript, so we can use JavaScript to hook into the lifecycle of that request and perform certain actions depending on the response.

We only want to remove the tag from the page if the request was a success, so we can use the `ajax:success` callback like we did before. Let's add this functionality to a new file called `app/javascript/src/tags.js`:

```
window.addEventListener("DOMContentLoaded", () => {
  const tags = document.getElementsByClassName("tags");
  const removeEls = tags[0].getElementsByClassName("remove");

  for (let removeEl of removeEls) {
    removeEl.addEventListener("ajax:success", () => {
      removeEl.parentElement.remove();
    });
  }
});
```

This JavaScript code first waits for the page to load (`DOMContentLoaded`). Once the page has loaded, it looks for any elements on the page with a `remove` class that exist inside of an element with a `tags` class. For each of these elements, this JS code adds an `ajax:success` event listener. When the request to remove a tag from a ticket completes, the function will be called to remove the tag from the page.

We need to require this file for this code to run, so let's go to `app/javascript/packs/application.js` and add in this line:

```
require("../src/tags");
```

When you run your feature using `bundle exec rspec spec/features/removing_tags_spec.rb`, you'll see that it now passes:

```
1 example, 0 failures
```

Awesome! With this feature done, users with permission to tag tickets of a project will now be able to remove tags too. Before you commit this feature, run `bundle exec rspec` to make sure everything is OK.

```
53 examples, 0 failures, 1 pending
```

That's awesome too! There's one pending spec inside `spec/helpers/tags_helper_spec.rb`. You can delete this file now, and if you rerun `bundle exec rspec`, you'll see this now:

```
52 examples, 0 failures
```

Three lots of awesomeness! Commit and push this:

```
$ git add .
$ git commit -m "Add the ability for users to remove tags from tickets"
$ git push
```

Now that you can add and remove tags, what is there left to do? Find them! By implementing a way to find tickets with a given tag, you make it easier for users to see only the tickets they want to see. As an added bonus, you'll also implement a way for users to find tickets for a given state, perhaps even at the same time as finding a tag.

When you're done with this next feature, you'll add some more functionality that will let users go to tickets for a tag by clicking on the tag name inside the ticket show page.

12.4. Finding tags

At the beginning of this chapter, there was mention of searching for tickets using a query such as `tag:"Iteration 1" state:open`. This magical method would return all the tickets associated with the "Iteration 1" tag that have the state of Open. This helps users scope down the list of tickets that appear on a project page so they can better focus on them.

There's a gem developed specifically for this purpose, called Searcher, which you can use. This gem is good for a lo-fi solution, but it shouldn't be used in a high-search-volume environment. For that, look into full text search support for your favorite database system[73]. This gem provides you with a search method on specific classes. It accepts a query like the one mentioned and returns the records that match it.

12.4.1. Testing search

As usual, you should (and will) test that searching for tickets with a given tag works. You'll do this by writing a new feature called `spec/features/searching_tickets_spec.rb` and filling it with the content from the following listing.

Listing 354. spec/features/searching_tickets_spec.rb

```
require "rails_helper"

RSpec.feature "Users can search for tickets matching specific criteria" do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project) }
  let!(:ticket_1) do
    tags = [FactoryBot.create(:tag, name: "Iteration 1")]
    FactoryBot.create(:ticket, name: "Create projects",
      project: project, author: user, tags: tags)
  end

  let!(:ticket_2) do
    tags = [FactoryBot.create(:tag, name: "Iteration 2")]
    FactoryBot.create(:ticket, name: "Create users",
      project: project, author: user, tags: tags)
  end

  before do
    login_as(user)
    visit project_path(project)
  end

  scenario "searching by tag" do
    fill_in "Search", with: %q{tag:"Iteration 1"}
    click_button "Search"
    within(".tickets") do
      expect(page).to have_link "Create projects"
      expect(page).to_not have_link "Create users"
    end
  end
end
```

In the setup for this feature, you create two tickets and give them two separate tags: `Iteration 1` and `Iteration 2`. When you look for tickets tagged with `Iteration 1`, you shouldn't see tickets that don't have this tag, such as the one that is only tagged `Iteration 2`.

Run this feature using `bundle exec rspec spec/features/searching_tickets_spec.rb` and it'll complain because there's no "Search" field on the page:

```
1) Users can search for tickets matching specific criteria searching by tag
   Failure/Error: fill_in "Search", with: "tag:Iteration 1"
   Capybara::ElementNotFound:
     Unable to find field "Search"
```

In your feature, the last thing you do before attempting to fill in this "Search" field is go to the project page. This means that the "Search" field should be on that page, so that your feature, and more importantly your users, can fill it out. Because we're searching tickets, we'll place the search form inside the `header` section that lists out the tickets, like so, inside `app/views/projects/show.html.erb`:

Listing 355. Adding a ticket search form to the project details page

```
<h2>Tickets</h2>

<ul class="actions">
  <li>
    <%= form_tag search_project_tickets_path(@project), method: :get,
      class: "form-inline" do %>
      <%= label_tag "search", "Search", class: "sr-only" %>
      <%= text_field_tag "search", params[:search], class: "form-control" %>
      <%= submit_tag "Search", class: "btn btn-outline-secondary ml-2" %>
    <% end %>
  </li>
  ...
```

This will reuse the same search styles on smaller screens as larger ones, as well as putting in some padding when the width of the form causes the "New Ticket" button to wrap to a second line.

This will give us a nice search form that looks like this:

(It won't look like that yet, because we haven't finished the feature. But it will when we're done.)

You've used `form_tag` before: this method generates a form that's not tied to any particular object, but still gives you the same style of form wrapper that `form_with` does. Inside the `form_tag`, you use the `label_tag` and `text_field_tag` helpers to define a label and input field for the search terms, and you use `submit_tag` for a submit button for this form. We've added a few of Bootstrap's classes like `sr-only`, `form-control` and `btn btn-default` to get the form to look nice.

The `search_project_tickets_path` method is undefined at the moment, as you'll see when you run `rspec spec/features/searching_tickets_spec.rb`:

```
1) Users can search for tickets matching specific criteria searching by
   tag
   Failure/Error: visit project_path(project)
   ActionView::Template::Error:
     undefined method `search_project_tickets_path' for ...
```

Notice the pluralized "tickets" in this method. To define nonstandard RESTful actions, you've previously used the `member` method inside of `config/routes.rb`. This has worked fine because you've always acted on a single resource, eg. wanting to remove a single tag from a ticket. This time, however, you want to act on a collection of a resource, so you use the `collection` method in `config/routes.rb` instead. To define this method, change these lines in `config/routes.rb`,

```
Rails.application.routes.draw do
  ...
  resources :projects, only: [:index, :show] do
    resources :tickets do
      collection do
        post :upload_file
      end

      member do
        patch :watch
      end
    end
  end
  ...
```

to these:

Listing 356. Adding our first `collection` route, for searching tickets

```
Rails.application.routes.draw do
  ...
  resources :projects, only: [:index, :show] do
    resources :tickets do
      collection do
        post :upload_file
        get :search
      end

      member do
        patch :watch
      end
    end
  end
  ...
```

The `collection` block here defines that there's a `search` action that may act on a collection of tickets. This `search` action will receive the parameters passed through from the `form_tag` that you've set up.

When you run your feature again by using `bundle exec rspec spec/features/searching_tickets_spec.rb`, you'll see that it's reporting that the `search` action is missing:

```
1) Users can search for tickets matching specific criteria searching by
   tag
   Failure/Error: click_button "Search"
   AbstractController::ActionNotFound:
     The action 'search' could not be found for TicketsController
```

Good! The job of this action is to find all the tickets that match the criteria passed in from the form as `params[:search]`.

12.4.2. Searching by tags

You want to be able to parse labels in a query such as "tag:Iteration 1" and find the records that match the query. Rather than working like Google, where you could enter "Iteration 1" and Google would "know" what you mean, you have to tell our searching code what "Iteration 1" means by prefixing it with "tag:". You'll use this query with the search method on a model, and it will return only the records that match it:

Listing 357. The syntax we want to end up with

```
Ticket.search("tag:Iteration 1")
```

Our form is submitting to a new `search` action, that it expects to find in our `TicketsController`. Let's define that method in `app/controllers/tickets_controller.rb` like so:

```
class TicketsController < ApplicationController
  ...
  def search
    if params[:search].present?
      @tickets = @project.tickets.search(params[:search])
    else
      @tickets = @project.tickets
    end
  end
  ...
```

First of all, we need to authorize this action. If we didn't do this, anyone could search for tickets in a project. Secondly, we assign all the tickets retrieved by the `search` method to the `@tickets` variable, which you would render in the `search` template if you didn't already have a template that was useful for rendering lists of tickets.

That template would be the one at `app/views/projects/show.html.erb`, but to render it you're going to make one small modification.

Currently this template renders all the tickets by using this line to start:

Listing 358. Rendering all tickets on a project

```
...
<ul class="tickets">
  <% @project.tickets.each do |ticket| %>
    ...
```

This line will iterate through each of the tickets in the project and do whatever is inside the block for each of those tickets. If you were to render this template right now with the `search` action, it would still return all tickets for the project, rather than the ones returned by the search query. You can get around this by changing the line in the template to read as follows:

Listing 359. Rendering a selected collection of tickets on a project

```
...
<ul class="tickets">
  <% @tickets.each do |ticket| %>
    ...
```

With this change, you break the `show` action of the `ProjectsController`, because the `@tickets` variable is not defined there. You can see the error you'd get by running `bundle exec rspec spec/features/viewing_tickets_spec.rb`:

```
1) Users can view tickets for a given project
   Failure/Error: click_link "Visual Studio Code"
   ActionView::Template::Error:
     undefined method `each' for nil:NilClass
```

This is a great thing about the tests we've written - we immediately know if we've broken existing functionality. To fix this error, we can also define a `@tickets` variable inside the `show` action of `ProjectsController`, underneath the `authorize` call in that action:

```
class ProjectsController < ApplicationController
  ...
  def show
    @tickets = @project.tickets
  end
  ...
```

When you rerun `bundle exec rspec spec/features/viewing_projects_spec.rb`, you'll see that it now passes:

```
1 example, 0 failures
```

Great! With the insurance that you're not going to break anything now, you can render that same `app/views/projects/show.html.erb` template in the `search` action of `TicketsController`, by putting this line at the bottom of that action, so that your controller action looks like this:

```
class TicketsController < ApplicationController
  ...
  def search
    if params[:search].present?
      @tickets = @project.tickets.search(params[:search])
    else
      @tickets = @project.tickets
    end
    render "projects/show"
  end
  ...
```

By rendering this template, you'll show a page similar to `ProjectsController#show`, but this time it will only have the tickets for the given tag.

Now that you're rendering properly, you can run the main searching spec again: `bundle exec rspec spec/features/searching_tickets_spec.rb`:

```
1) Users can search for tickets matching specific criteria searching by
   tag
   Failure/Error: click_button "Search"
   NoMethodError:
     undefined method `search' for #<Ticket::ActiveRecord_...
```

This error shows us that the test is getting to the `search` action of `TicketsController` and it's attempting to execute a search on the `Ticket` model using the `search` method. All of this is great!

Now that you have all the wiring set up, you just need to implement the actual functionality inside of your `Ticket` model. We'll be using the Searcher gem to do this, so first we need to install it. Let's add this gem now:

```
bundle add searcher -v '~> 6.0'
```

To add this `search` method to `Ticket`, we can use the `searcher` class method that the Searcher gem provides, placing it underneath our associations in `app/models/ticket.rb`:

```
class Ticket < ActiveRecord::Base
  ...
  searcher do
    label :tag, from: :tags, field: "name"
  end
  ...
```

The `searcher` method takes a block in which we can specify the labels that we can use for searching. The `from` option tells searcher that we want to look in the `tags` table, and the `field` option tells us we want to pick the `name` field from that table. This will allow us to use search queries like `tag:TestTag`, which will search for tickets are associated with a record in the `tags` table with the `name` field equal to "TestTag" - exactly what we're after.

When you run your "Searching" feature using `bundle exec rspec spec/features/searching_tickets_spec.rb`, it will now pass:

```
1 example, 0 failures
```

With this feature, users will be able to specify a search query, such as `tag:Iteration 1` to return all tickets that have that particular tag. You prevented one breaking change by catching it as it was happening, but how about the rest of the test suite? Find out by running `bundle exec rspec`. You should see this result:

```
53 examples, 0 failures
```

Great! Commit this change now:

```
$ git add .
$ git commit -m "Add tag-based searching for tickets"
$ git push
```

Now that you have tag-based searching, why not spend a little bit of extra time letting your users search by state as well? This way, they'll be able to perform actions such as finding all remaining Open tickets in the tag "Iteration 1" by using a search term of `state:Open tag:"Iteration 1"`. It's easy to implement.

12.4.3. Searching by state

Implementing searching for a state is incredibly easy now that you have the Searcher gem set up and have the search feature in place. As you did with searching for a tag, you'll test this behavior in the "Searching" feature. But first, you need to set up your tickets to have states.

Change the code at the top of the feature in `spec/features/searching_tickets_spec.rb` so that states are specified for each of the tickets, replacing the two `let` blocks for the tickets with the code from the following listing.

Listing 360. spec/features/searching_tickets_spec.rb

```
RSpec.feature "Users can search for tickets matching specific criteria" do
  ...
  let(:open) { State.create(name: "Open") }
  let(:closed) { State.create(name: "Closed") }

  let!(:ticket_1) do
    tags = [FactoryBot.create(:tag, name: "Iteration 1")]
    FactoryBot.create(:ticket, name: "Create projects",
      project: project, author: user, tags: tags, state: open)
  end

  let!(:ticket_2) do
    tags = [FactoryBot.create(:tag, name: "Iteration 2")]
    FactoryBot.create(:ticket, name: "Create users",
      project: project, author: user, tags: tags, state: closed)
  end
  ...
```

When the two tickets in this feature are created, there will be two states associated with these tickets. The next task is to write a scenario that will search for all tickets with a specific state. That scenario can be seen in the next listing.

Listing 361. Finding by state scenario, for the "Searching" feature

```
RSpec.feature "Users can search for tickets matching specific criteria" do
  ...

  scenario "searching by state" do
    fill_in "Search", with: "state:Open"
    click_button "Search"
    within(".tickets") do
      expect(page).to have_link "Create projects"
      expect(page).to_not have_link "Create users"
    end
  end
end
```

This should show any ticket with the "Open" state and hide all other tickets. When you run this feature with `bundle exec rspec spec/features/searching_tickets_spec.rb`, you'll see that this is not the case. It can still see the "Create users" ticket.

```
1) Users can search for tickets matching specific criteria searching by
   state
   Failure/Error: expect(page).to_not have_link "Create users"
     expected not to find link "Create users", found 1 match:
     "Create users"
```

This test is not working because we haven't enabled searching on states yet. We can fix this very easily by changing the `searcher` code in `app/models/ticket.rb`:

```
class Ticket < ActiveRecord::Base
  ...
  searcher do
    label :tag, from: :tags, field: "name"
    label :state, from: :state, field: "name"
  end
  ...
```

If you re-run the searching spec with with `bundle exec rspec spec/features/searching_tickets_spec.rb` you'll get a nice surprise:

```
2 examples, 0 failures
```

That's it for the searching feature! In it, you've added the ability for users to find tickets with a given tag and/or state. It should be mentioned that these queries can be chained, so a user can enter a query such as `tag:Iteration 1 state:Open`, and it will find all tickets with the "Iteration 1" tag and the Open state.

As usual, commit your changes because you're done with this feature. But also as usual, check to make sure that everything is A-OK by running `bundle exec rspec`:

```
54 examples, 0 failures
```

Brilliant, you can commit:

```
$ git add .
$ git commit -m "Add ability to search for tickets by state or tag"
$ git push
```

With searching in place and the ability to add and remove tags, you're almost done with this set of features.

12.4.4. Search, but without the search

The final feature for this chapter involves changing the tag name rendered in `app/views/tags/_tag.html.erb` so that when a user clicks on it, they're shown all tickets for that specific tag.

To test this functionality, you can add another scenario to the bottom of `spec/features/searching_tickets_spec.rb` to test that when a user clicks on a ticket's tag, they're only shown tickets for that tag. The new scenario looks pretty much identical to this:

```
RSpec.feature "Users can search for tickets matching specific criteria" do
  ...
  scenario "when clicking on a tag" do
    click_link "Create projects"
    within(".ticket .attributes .tags") do
      click_link "Iteration 1"
    end

    within(".tickets") do
      expect(page).to have_content "Create projects"
      expect(page).to_not have_content "Create users"
    end
  end
end
```

When you run this last scenario using `bundle exec rspec spec/features/searching_tickets_spec.rb`, you're told that it can't find the "Iteration 1" link on the page:

```
1) Users can search for tickets matching specific criteria when clicking
   on a tag
   Failure/Error: click_link "Iteration 1"
   Capybara::ElementNotFound:
     Unable to find link "Iteration 1"
```

This scenario is successfully navigating to a ticket and then attempting to click a link with the name of the tag, but it can't find any matching links. It's up to you to add this functionality to your app.

Where you display the names of tags in your application, you need to change them into links that go to pages displaying all tickets for that particular tag. We've isolated the code to display tags in one place - `app/views/tags/_tag.html.erb` - so that's the only place we need to change. Open the file, and change the output of the tag name from this:

Listing 362. Displaying the tag name

```
<%= tag.name %>
```

to this:

Listing 363. Displaying the tag name with a link to a prefilled-search for tickets

```
<%= link_to tag.name, search_project_tickets_path(ticket.project,
  search: %Q{tag:"#{tag.name}"}) %>
```

For this `link_to`, you use the `search_project_tickets_path` helper to generate a route to the `search` action in `TicketsController` for the current ticket's project, but then you do something different. After you specify which project to search with, using `ticket.project`, you specify options. These options are passed in as additional parameters to the route. Your search form passes through the `params[:search]` field, and your `link_to` does the same thing.

When you run `bundle exec rspec spec/features/searching_tickets_spec.rb`, this new scenario will now pass:

```
3 examples, 0 failures
```

This feature allows users to click a tag on a ticket's page to then see all tickets that have that tag. Make sure you didn't break anything with this small change by running `bundle exec rspec`. You should see this output:

```
55 examples, 0 failures
```

Great, nothing broke! Commit this change:

```
$ git add .
$ git commit -m "Users can now click a tag's name to go to a page
  showing all tickets for it"
$ git push
```

Users are now able to search for tickets based on their state or tag, as well as go to a list of all tickets for a given tag by clicking on the tag name that appears on the ticket's page. This is the final feature you needed to implement in order to have a good tagging system for your application.

12.5. Summary

In this chapter, we've covered how to use a `has_and_belongs_to_many` association to define a

link between tickets and tags. Tickets are able to have more than one tag, but a tag is also able to have more than one ticket assigned to it, so you use this type of association. A `has_and_belongs_to_many` could also be used to associate people and the locations they've been to[74].

You first wrote the functionality for tagging a ticket when it was created, and you continued by letting users tag a ticket through the comment form as well.

Next, we looked at how to remove a tag from the page using some JavaScript code that responded to a successful AJAX request, that then removed the tag element from the page.

You saw how to use the Searcher gem to implement label-based searching for not only tags, but states as well. Usually you'd implement some sort of help page that would demonstrate to users how to use the search box, but that's an exercise best left to the reader.

Your final feature, based on the previous feature, allowed users to click a tag name and view all the tickets for that tag, and it also showed how you can limit the scope of a resource without using nested resources.

[71] For example, by using a process such as Agile, feature sets, or any other method of grouping.
[72] For a full list of all the methods that the `tags` association gives you for free, check out http://guides.rubyonrails.org/association_basics.html#has-and-belongs-to-many-association-reference
[73] Or an external system, like Elasticsearch.
[74] Foursquare does this.

Chapter 13. Deployment

Developing applications is fun, but using them is more fun. Nobody can use your application until you make it available on the internet somewhere. In this chapter, we'll get you started on learning how to make this application available to the general public through a process we refer to as deployment.

Deployment is a big topic, enough for dedicated books on the subject alone. [75] This book can't possibly explain everything there is to know about deployment, so please think of this as an introduction.

One of the difficult things about deployment is that the process often relies on the details of how you've built your application. This means different projects have slightly different deployment processes. This book will show you what's needed to deploy Ticketee, but more complicated applications may have additional needs that the book won't cover here.

A wealth of tooling exists to assist you with getting your application into production. This chapter will build up your understanding of deployment slowly, so that you can understand all the parts. Although it's true that deploying Rails involves a bit more than FTPing a few files to a remote server[76], you'll have to confront these details eventually, no matter what your framework.

At the end of the chapter, we'll make our application send real world emails using the features that we built in the last chapter, using a service called Mailgun.

13.1. What is deployment?

Deployment is the act of placing your application on the internet. There's an entire process involved in getting your application off your computer and onto a server somewhere: setting up a server, transferring the code over, handling credentials and connections to other servers, and so on. However there are dedicated Platforms-as-a-Service (PaaS) options such as Heroku (http://heroku.com) which remove a lot of this pain.

You can also think of deployment as "moving your application into a different environment." Remember your `config/environments` directory? It has files for three different environments: development, test, and production. You use the development environment when you're playing with your application using `rails server`. You use the test environment every time you run your tests with `bundle exec rspec`. But you haven't used the production

environment yet. That's what this chapter is about.

Deploying to Heroku is as simple as `git push heroku master`. This pushes your code to Heroku rather than GitHub, and Heroku runs a series of steps to build your application in order to get it ready for production. The production environment is special, because it's the only one that other people can use.[77] Your development and test environments aren't shared; they're for your eyes only. Also, generally development and test environments are used on your local machine, whereas production environments are used on a server somewhere else. Production environments are often configured for maximum speed, whereas development and test environments are configured for maximum convenience.

With that in mind, let's go ahead and get Ticketee out there for everyone to try.

13.2. Simple deployment with Heroku

Heroku is a company that specializes in handling all the details of deployment for you. How convenient! Heroku started off specializing in hosting Rails applications, and it eventually added support for many other languages and frameworks. Heroku still pays extra attention to Ruby, though: the company even pays Matz[78] and two other members of the Ruby core team, to work on Ruby itself.

There is a catch, of course: that convenience comes at a cost. Heroku is more expensive than other hosting options. But for small apps, it has a service tier that is 100% free. That's what you'll be using here. It's not a lot, but it's enough to get you going with your first few deployments.

13.2.1. Signing up

You need two things to get set up with Heroku: an account and the Heroku CLI. Luckily, they're both easy to acquire. Getting an account with Heroku is as easy as going to https://www.heroku.com/, clicking the big "Sign Up For Free" button, putting in your name and email address, clicking "Create Free Account", and then following the instructions in the email you're sent. Once you're finished with that, it's time to get the Heroku CLI.

Heroku distributes a bunch of command-line tools that you can use to interact with your Heroku account. To get a copy, go to https://devcenter.heroku.com/articles/heroku-cli and follow the instructions. Pretty easy, eh?

You need to do one last bit of setup: link together the Heroku CLI and your account. There's an easy command-line way to do this:

```
$ heroku login
```

The `heroku login` command will ask you for the email and password you used to create the account on the website. It will then try to find your SSH keys. SSH stands for Secure Shell and is a program you'll use to log in to remote servers. If you haven't used SSH before, you won't have any keys. `heroku login` will ask you if you want to generate some new ones. If it asks you this, answer Yes. It will then upload the public half of your key to your Heroku account, allowing you to use the CLI to the fullest extent.

13.2.2. Provisioning an app

The first part of any deployment process is called provisioning. Provisioning is the process of requesting some kind of resource from a provider of that resource. In this case, you want to provision a new application on Heroku. Doing so creates all the internal accounting needed to give you access, and then you can deploy your code to that application.

The Heroku CLI gives you a simple way to provision a new application:

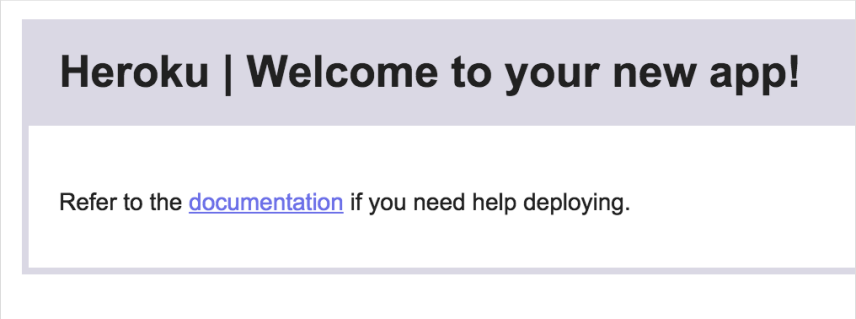
```
$ heroku apps:create
```

After running this command, you'll see some output that looks like this:

```
Creating app... done,   fast-escarpment-46140
https://fast-escarpment-46140.herokuapp.com/ |
https://git.heroku.com/fast-escarpment-46140.git
```

The names will be a bit different, because they're random. Every Heroku application must have a unique name, so `heroku apps:create` generates a random one for you. In this example, the app was christened `fast-escarpment-46140`.

The next line gives you two different URLs: one HTTP and one Git. The HTTP URL is a link that you can open in a web browser to view your application. When you open that URL you'll see this page:



Opening your Heroku app

If at any point you want to open your Heroku app, just run `heroku open` in the ticketee application directory. This will automatically open your app in a browser for you. You won't have to remember the URL this way!

The Git URL is a link to a Git repository that represents your application. Just like the `origin` Git remote often points to a repository on GitHub, the `heroku` Git remote points at Heroku. We'll use this in a little while.

Anyway, that's it for provisioning! You can see that you now have a new app if you visit your Heroku dashboard: https://dashboard.heroku.com/apps. Next, we're going to dive in a bit deeper into this app concept and how to best design your application for production.

13.3. Twelve-factor apps

There are many, many ways to build and deploy web applications. The number of options is overwhelming. After you deploy a few applications, you'll probably develop some opinions about how an application can be best designed to make it easy to get into the hands of your users.

Designed for production? Yep, the design decisions that you make in development can also affect production. For example, if you allow file uploads, your production setup must be able to accept the files, save them somewhere, and serve them back to users. If you add a job queue that requires Redis—as you would by using the Resque or Sidekiq gems—now your deployment strategy has to take Redis into account as well. As with any choice, you can make good choices and bad choices. Good choices will make deployment a breeze. Bad choices will make it difficult, complex, and scary.

Cloud providers like Heroku recommend a particular set of guidelines for building applications. These choices were made based on lessons learned while scaling hundreds of

Rails applications. Even if you don't use a provider like Heroku, the guidelines are still useful. These suggestions are boiled down into just 12 points, called the twelve-factor app [79]. The entire document is worth reading, because it makes a bunch of great points about deploying applications. We won't go over all 12 points here; the website does a good job of explaining itself. But we will go over one or two and show how they affect writing a Rails application.

13.3.1. Configuration

Section three of the twelve-factor app is about configuration. The rule is, "The twelve-factor app stores config in environment variables." By "configuration," it doesn't mean everything in your `config` directory: it means everything that varies between different deployments of an application.

How does this affect your Rails application? Back in chapter 11, you added some code for sending out emails. If this code contained your email credentials, then anyone with access to the code would know the credentials. It's a much safer idea to store these credentials on a protected production server. This is an application of the 12-factor configuration principle.

The configuration section in the 12-factor document also has a good rule of thumb to approximate this rule: "A litmus test for whether an app has all config correctly factored out of the code is whether the codebase could be made open source at any moment, without compromising any credentials." This is a great way to think about this problem, because it shows the strength of making this choice outside of a deployment context, too: if someone were to leak your source code, would your data still be safe? If a new employee joins your team, do they need the keys to the production data store on their first day? Ideally, secrets should be as secret as possible, not saved in a place everyone needs an exact copy of.

13.3.2. Processes

Section six is about processes. The sixth factor is, "Twelve-factor processes are stateless and share-nothing." Huh?

Imagine that you had two servers, each running a copy of your application. If these servers had their own copy of the Ticketee database, they'd have state and so would not be stateless. Also, they'd need to keep their data in sync, so you'd have to pass new data back and forth to keep everything consistent. Those are bad choices. If, instead, each server of yours saved no state of its own and wrote all of its state to a separate database server, you wouldn't need to share any data between copies of your application, and there would be no consistency

complexity. You could change from 2 servers to 200 servers, and your application wouldn't care. You could make your database server grow with your data, yet keep your app servers lean.

Heroku enforces this statelessness aspect, so your application needs to be designed to accommodate it. The first place this crops up in application design is often file uploads. Remember back in chapter 9, when you used Active Storage? You saved those files locally. If you deployed two copies of your application, some files would be on one server, and other files would be on another server. You're back to shared, stateful servers. That's why Active Storage provides an option to configure your application to use another backing store, such as Amazon S3.

The first thing you'll need is an Amazon S3 (Simple Storage Service) account. You can get one here: https://aws.amazon.com/free.

You'll need a credit card to sign up to Amazon S3. You won't be charged anything for the first year of service. If you don't want to sign up with a credit card, you can still continue this chapter. The only issue will be that file uploads won't persist between deploys for the Ticketee application.

If you're reading this section sometime after this book has been published, it may be that Amazon has changed their design. Please consult Amazon's own documentation on how to setup your S3 account.

After that, you need four bits of information: your access key, your secret key, your bucket name, and your region. You can get all of these credentials from your AWS dashboard. To create an access key, sign in to your Amazon AWS account, then click your name, then "Security Credentials". On this page, expand the "Access Keys" section and then create a new access key. You will be prompted to download the credentials. Do that, and then your access key and secret key will be in that file.

Now for the S3 bucket. An S3 Bucket is where Ticketee's uploaded files will be stored. To create the bucket, go to the S3 Management console (https://console.aws.amazon.com/s3/home) and click "Create Bucket", giving your bucket a name like "ticketee" so that you know what it's for. You can select whichever region you like, but typically you would select a region close to your users so that the files would be served faster for them.

Once you've created the bucket, getting the AWS name of the region you selected is a little

tricky. One way to do it on the S3 dashboard is to select your bucket, click "Properties", and expand "Static Website Hosting". The region you want is the part of the endpoint, in "s3-website-[region_name].amazonaws.com", eg. `us-east-1` or `ap-southeast-2`.

13.3.3. Configuring your S3 Credentials

Now that you have your S3 credentials and a bucket setup, we can securely store these credentials inside our Rails application by using a feature called Rails Credentials. To use this feature, we can use this command:

```
rails credentials:edit
```

This will open up a window in your editor with this code like this in it:

```
aws:
  # access_key_id: 123
  # secret_access_key: 345
  # Used as the base secret for all MessageVerifiers in Rails, including the one protecting
cookies.
  secret_key_base:
ae8691f631907afeef452c10dffb99204ae9a2804393b7d0d607b9b80b75f37a00b759a5b8bc49963ecfea11253e6
2c595e4a57e633a18ddc881f25b1e652ffe
```

This file is encrypted and stored at `config/credentials.yml.enc`. It's encrypted with a secure key, stored at `config/master.key`. This file is never uploaded to Git, instead it's stored on your own machine and it was generated when you built this application.

Lost your master key?

If you've lost `config/master.key`, never worry! You can get around this by removing your `config/credentials.yml.enc` file, and then re-running `rails credentials:edit`.

There's no problem with re-generating this file now. We haven't used it yet.

Let's change this file that we have open now to include our access key and secret access key:

```
aws:
  access_key_id: AKIA...
  secret_access_key: 86R...
  bucket: your-bucket-name-here
```

Close this file. This will automatically encrypt those credentials and save them to `config/credentials.yml`.

In order for Heroku to be able to read and decrypt these settings, we'll need to tell it about the key stored at `config/master.key`. We can do that with this command:

```
heroku config:add RAILS_MASTER_KEY=<contents of config/master.key>
```

Next, we will need to configure `config/storage.yml` to use these variables by adding this configuration to that file:

Listing 364. config/storage.yml

```
amazon:
  service: S3
  access_key_id: <%= Rails.application.credentials.dig(:aws, :access_key_id) %>
  secret_access_key: <%= Rails.application.credentials.dig(:aws, :secret_access_key) %>
  region: us-east-1
  bucket: <%= Rails.application.credentials.dig(:aws, :bucket) %>
```

Then, we will need to change some configuration in `config/environments/production.rb` to use this Active Storage configuration. Change this line in that file:

Listing 365. config/environments/production.rb

```
config.active_storage.service = :local
```

To this:

Listing 366. config/environments/production.rb

```
config.active_storage.service = :amazon
```

Finally, Active Storage will also need one extra gem in order to talk with Amazon. That gem is called `aws-sdk-s3`. Let's add to the `Gemfile` now:

```
gem 'aws-sdk-s3', require: false
```

Once we have configured the credentials, `config/storage.yml`, tweaked the `service` setting in `config/environments/production.rb` and added the `aws-sdk-s3` gem, our application will now be configured to upload files to S3 when it is running on Heroku.

Once you've made these changes, go ahead and commit all our work:

```
$ git add .
$ git commit -m "Configuring Active Storage for 12factor"
```

That's it! Active Storage will now work with Heroku and will upload files to S3.

You have another thing that needs to be configured, though: your database. You see, SQLite, the database you've been using, stores your data into a file on disk. That's excellent for development, but in production, it won't cut it. You can't have that shared state between servers! So rather than use SQLite, you'll be using Postgres for your production data store. Heroku comes with excellent Postgres support.

Setting up Postgres locally can be a pain, though. Wouldn't it be nice if you could use SQLite locally, but Postgres on Heroku? If you guessed that that was a leading question, you'd be correct. All you need to do is configure your gems. Find this line in your `Gemfile`:

```
# Use sqlite3 as the database for Active Record
gem "sqlite3", "~> 1.4"
```

And change it to this:

```
gem "sqlite3", "~> 1.4", group: [:development, :test]
gem "pg",      group:  :production
```

Using different databases in different environments considered harmful

On any kind of serious production application we highly recommend taking the time to set up PostgreSQL, along with the pg gem, in all environments.

If you use different databases in the different environments you may have a case where the code works locally and not on production, which can be disastrous.

We've only configured this way for the sake of brevity.

We don't want to install the pg gem in the development environment, for a couple of reasons:

- To install the pg gem locally, you need to have PostgreSQL installed - or at least, its development header files. On Linux its easy to install just the headers; on other operating systems, not so much.
- We're not going to be using PostgreSQL anyway!

So to install only the gems we want to use in development (and test), we can tweak the `bundle` command slightly:

```
$ bundle install --without=production
```

You only have to do this once; future `bundle` commands will remember this preference. At the end of the output from running `bundle`, you'll see the following:

```
Bundle complete! 28 Gemfile dependencies, 104 gems now installed.
Gems in the group production were not installed.
Use `bundle show [gemname]` to see where a bundled gem is installed.
```

It's that easy! Let's commit these changes:

```
$ git add .
$ git commit -m "Add Postgres for our production data store"
$ git push
```

Now you've got Postgres configured for your production database. You have one more thing to configure with regard to your statelessness: assets.

You see, a stateless app delegates things to a backing store as needed. Your application's assets—that is, its graphics and CSS and JavaScript files—are state. But they're not mutable state, in that you only mutate them when you make a new deploy.

Your Rails app is finally twelve-factor compliant. That wasn't too bad! As we said before, make sure to check out all 12 points, they can teach you a lot about how things work under the hood. But enough talk: let's get your application into production.

13.4. Deploying Ticketee

After all that fuss with getting your application ready, you may be sweating bullets when thinking about finally deploying it. With all that setup, the final process must be hard, right? Don't worry, we'll get through this together. Let's do the first step: sending your code up to Heroku. You can do this with the new `heroku` Git remote that `heroku create` added for you:

```
$ git push heroku master
```

This will push the latest `master` branch from our local Git repository to Heroku, rather than the usual `git push` operation of pushing it to GitHub.

When you do this, you see a bunch of output. It may take a minute or two to get going. It looks something like this:

```
Counting objects: 1256, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (883/883), done.
Writing objects: 100% (1256/1256), 144.56 KiB | 0 bytes/s, done.
Total 1256 (delta 773), reused 527 (delta 315)
Compressing source files... done.
Building source:

-----> Building on the Heroku-20 stack
-----> Ruby app detected
-----> Installing bundler 2.0.2
-----> Removing BUNDLED WITH version in the Gemfile.lock
-----> Compiling Ruby/Rails
-----> Using Ruby version: ruby-2.7.2
       Running: bundle install --without development:test --path ...
       Fetching gem metadata from https://rubygems.org/.............
       Fetching rake 13.0.1
       Installing rake 13.0.1
       Fetching concurrent-ruby 1.1.6
...
```

It goes on and on and on ... and eventually, you should see it succeed. It gives you a couple of warnings though - let's look at resolving those first.

As part of the output, you'll see some text like the following:

```
###### WARNING:
No Procfile detected, using the default web server.
We recommend explicitly declaring how to boot your server process via a Procfile.
https://devcenter.heroku.com/articles/ruby-default-web-server
```

And:

```
###### WARNING:
You set your `config.active_storage.service` to :local in production.
If you are uploading files to this app, they will not persist after
the app is restarted, on one-off dynos, or if the app has multiple
dynos. Heroku applications have an ephemeral file system. To persist
uploaded files, please use a service such as S3 and update your Rails
configuration.
```

These are both issues we can and should fix!

The `Procfile` will inform Heroku how to run our application, which can lead to better performance on Heroku.

Fixing the second and more word-y warning will ensure that any files that are uploaded to ticketee remain in place after we restart the application. An application on Heroku is restarted every time we perform a new deploy, so this one is definitely more important to fix.

Let's fix these in order.

13.4.1. Fixing deployment issues

We've successfully completed the first step of deployment to Heroku, but we received some warnings we can fix.

To fix the `Procfile` issue, we need to create a new `Procfile`

We don't need to do a lot of configuration for Puma, but we do need to create a new file called `Procfile`, in the root of our Rails application. We can put this configuration in it:

Listing 367. Defining a new `Procfile`

```
web: bin/rails s -p $PORT
```

This tells Heroku to start up our application's process the same way we do locally by running `bin/rails s`.

Let's commit these changes, and try deploying again.

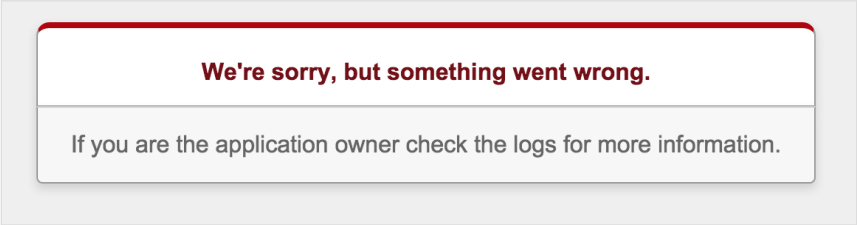
```
$ git add .
$ git commit -m "Add Procfile for Heroku"
$ git push
$ git push heroku master
```

When we push, we'll see that it found the `Procfile`:

```
-----> Discovering process types
       Procfile declares types -> web
       Default types for Ruby  -> console, rake, worker
```

And both of the warnings have gone! This is very good news. Open the URL it gives you in

your browser; it should look something like http://stark-chamber-2017.herokuapp.com/. Congratulations! Your application is now up and running. Let's go check on it by running `heroku open`.



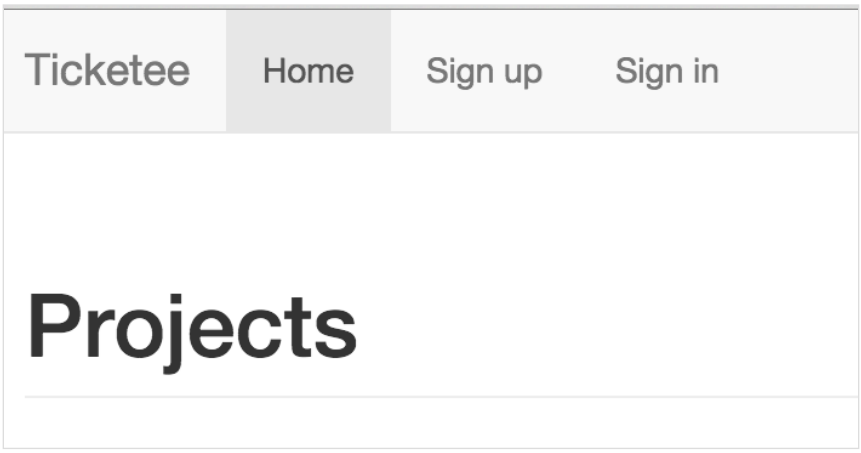
Oh dear! Our first production issue. What do we do in this case? To start with, we can consult the Heroku logs. The logs are the first place that we should go looking whenever anything goes wrong in our application. Let's open them up now with `heroku logs`. You'll get a lot of output, that looks similar to what you see when you run `rails server` in your own terminal. Looking through them, we can see this part:

```
...
[timestamp] app[web.1]: Started GET "/" for [IP address] at [timestamp]
[timestamp] app[web.1]:   Rendered projects/index.html.erb within
                          layouts/application (1.0ms)
[timestamp] app[web.1]:
[timestamp] app[web.1]: ActionView::Template::Error (PG::UndefinedTable:
                        ERROR:  relation "states" does not exist
[timestamp] app[web.1]: LINE 1: SELECT "states".* FROM "states"
...
```

This line is telling us that the `states` relation doesn't exist. In PostgreSQL, tables are called "relations"; so that would indicate that the `states` table is the thing that doesn't exist. How would this come to be? It would seem that we've forgotten to run our migrations on our production environment! To run the migrations on Heroku we can run this command:

```
$ heroku run rails db:migrate
```

When that's complete, refresh our application again. This time we'll see it running properly!



In order to sign in to the application, we can use the admin credentials. Well, at least we could do that if that user existed in our production database as well. To get that user created on Heroku, we can run another Rake command:

```
$ heroku run rails db:seed
```

As you might have guessed by now, we can prefix our normal `rake` commands with `heroku run` to run them in our Heroku application. This command will create the users, projects and states from the `db/seeds.rb` file of our application; which is just perfect for testing our application. Take a break and play around with it. You've come a long way!

13.4.2. Deploying is hard

Deploying is a new step in your process. You run your tests, commit, push to GitHub, and then push to Heroku:

Listing 368. Sample deployment process

```
$ bundle exec rspec
$ git add .
$ git commit -m "Some message"
$ git push origin
$ git push heroku
```

Isn't that kind of annoying? Six steps. Wouldn't it be nicer if you could just do this:

```
$ git add .
$ git commit -m "Some message"
$ git push
```

Then some magical… **thing** would handle running your tests and pushing to Heroku? You can do this with a technique called continuous deployment. You'll do that with one of our favorite services, GitHub Actions. Read on!

13.5. Continuous deployment with GitHub Actions

Before we get into the details of GitHub Actions, let's talk about what continuous deployment means. In a nutshell, continuous deployment means that every time you commit code, it ends up going into production. If that sounds a little extreme, that's because it is. Many people prefer to have a discrete time when they deploy. But if you limit your deployments to only doing them by hand, it's easy to not automate them as much, because you're already doing the work anyway. Furthermore, once you have fully automatic deployments, you can do all kinds of neat things. Continuous deployment can be a useful strategy, given the right developer mentality.

Here's the details about what GitHub will do when we push our code. GitHub will run the tests for our code, making sure it works. Just like what we've been doing throughout this book. It's that simple. It's impossible to forget to run your tests, because it's automatic.

If you’re building a library, GitHub can also automatically run your tests against multiple versions of Ruby, including JRuby and Rubinius. Furthermore, GitHub watches your pull requests and can tell you if a given pull request still passes all the tests after it’s been merged. One big example of this is https://github.com/ruby-i18n/i18n/actions?query=workflow%3ARuby:"the i18n project" which runs tests against a matrix of 6 different Ruby versions and 6 different Rails versions. This ensures that the i18n project will continue to work. Well, at least according to the tests!

13.5.1. Configuring GitHub Actions

Configuring GitHub actions for your Rails app is pretty simple. We need to create a new file at `.github/workflows/tests.yml` within our repository, and fill it out with certain steps that we want to do to run our build. Here's that file:

Listing 369. github/workflows/tests.yml

```
name: Tests

on: [push, pull_request]

jobs:
  test:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Install sqlite3
      run: sudo apt-get install libsqlite3-dev
    - name: Set up Ruby
      uses: ruby/setup-ruby@v1
      with:
        ruby-version: 2.7.2
    - name: Install RubyGems
      run: |
        bundle install --jobs 4 --retry 3
    - name: Install Node packages
      run: |
        yarn install
    - name: Compile assets
      run: |
        bundle exec rake assets:precompile
    - name: Run tests
      run: |
        bundle exec rspec
```

This file tells GitHub that in order to run our tests, it has to do a few things first:

1. It downloads our code, which it does with an action called `actions/checkout@v2`.
2. It has to install the `libsqlite3-dev` package with `apt-get`. If this package wasn't installed, the `bundle install` later on would fail.
3. Sets up Ruby 2.7.2 with the `ruby/setup-ruby@v1` action.
4. Installs the gems for the application with `bundle install`
5. Installs the Node packages with `yarn install`.
6. Compiles the application's assets so that they're available for the tests.

Then finally, it runs the tests.

These steps are very similar to what someone else would have to do if they were using our application.

We need to tweak one final thing for this to work correctly: we need to tell Capybara to use what's called a headless version of Firefox. By default, Capybara will run its JavaScript tests in a Firefox browser. You should've noticed this by now! When you run the tests, a Firefox window will open and that will be used by Capybara to ensure our JavaScript is performing correctly.

On GitHub Actions, we will not be able to open a Firefox browser in the same way. Instead, we'll have to use this "headless" version of Firefox, which is nowhere near as gory as it sounds. All "headless" means here is that we won't be seeing Firefox open up when the tests run on GitHub Actions.

To configure this, we need to add these lines to `spec/rails_helper.rb`:

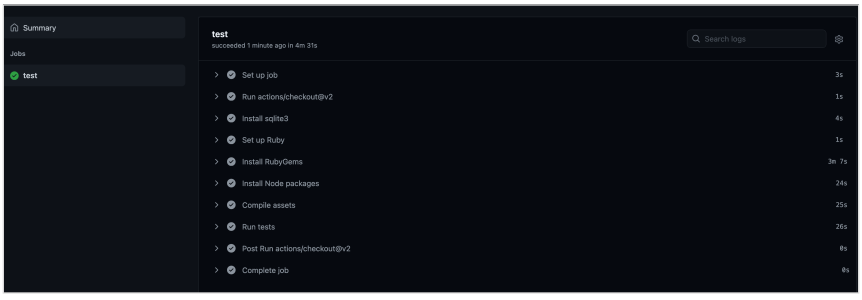
```
if ENV['CI']
  Capybara.javascript_driver = :selenium_headless
end
```

These lines tell Capybara to use the headless Selenium driver, which will still use the Firefox browser in the background, to run our JavaScript tests.

Let's commit and push these changes now:

```
git add .
git commit -m 'Configure Ticketee to run on Github Actions'
git push origin master
```

If you go to your GitHub repository for Ticketee and open the Actions tab, you'll see that the tests are then running automatically over there.



These tests will take a few minutes to run, just because it has to install your RubyGems and Yarn dependencies first, and that can take some time.

To speed it up, we can add this configuration to the GitHub Actions workflow file, directly underneath the checkout step:

Listing 370. .github/workflows/tests.yml

```
uses: actions/checkout@v2
- name: Get yarn cache directory path
  id: yarn-cache-dir-path
  run: echo "::set-output name=dir::$(yarn cache dir)"

- uses: actions/cache@v2
  name: Load Yarn Cache
  id: yarn-cache # use this to check for `cache-hit` (`steps.yarn-cache.outputs.cache-hit !=
'true'`)
  with:
    path: ${{ steps.yarn-cache-dir-path.outputs.dir }}
    key: ${{ runner.os }}-yarn-${{ hashFiles('**/yarn.lock') }}
    restore-keys: |
      ${{ runner.os }}-yarn-

- uses: actions/cache@v2
  name: Load Ruby Cache
  with:
    path: vendor/bundle
    key: ${{ runner.os }}-gems-${{ hashFiles('**/Gemfile.lock') }}
    restore-keys: |
      ${{ runner.os }}-gems-
```

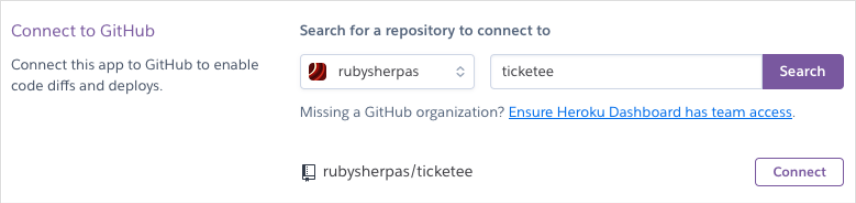
These additions will then make GitHub Actions cache the Yarn and Bundler dependencies in between each build, and it will speed it up significantly.

A lot of the power GitHub Actions come from its abilities to do things if your build passes or fails. One example is that when your build fails, GitHub can email your entire team to let you know that something is wrong. Another example is that GitHub can send a notification to Heroku to tell it to automatically deploy the application for us, without us having to run a single thing. Let's explore that now.

13.5.2. Continuous Deployment with Heroku

Configuring continuous deployment with Heroku is easy. We can go into our Heroku dashboard, find our Ticketee application, and then click the "Deploy" menu item at the top.

When we're on this screen, we can then choose to connect Heroku to GitHub:



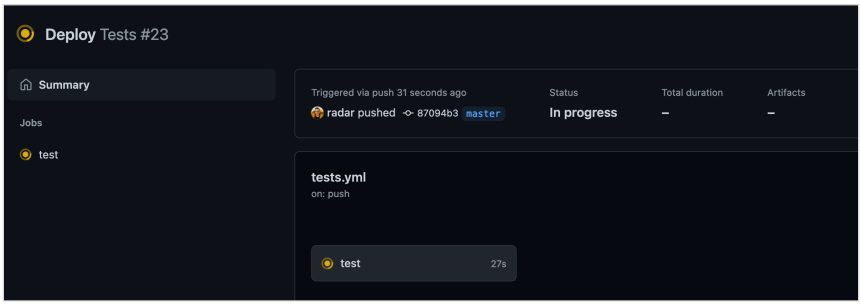
Once we've selected our repository there, we can then trigger automatic deployments from the master branch by first selecting "Wait for CI to pass before deploy" and then clicking the "Enable Automatic Deploys" button.

These two steps will mean that when we push code to our repository on GitHub, the tests will automatically run through GitHub Actions. If those tests pass, then Heroku will execute an automatic deployment for us.

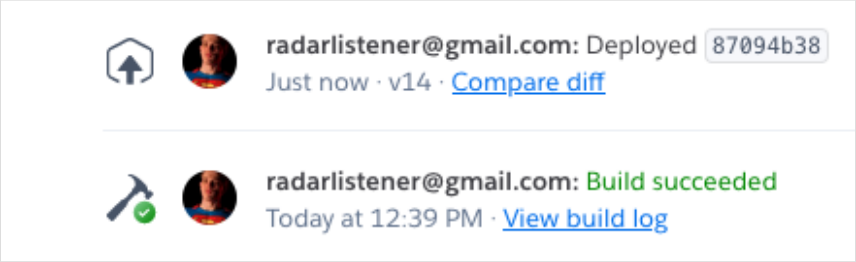
We can verify that this is working by making an empty commit on our repository and pushing that commit to GitHub.

```
$ git commit -m "deploy check" --allow-empty
$ git push origin master
```

Over on GitHub under the "Actions" tab in our repository, we'll see our tests running.



Once the tests have finished running, we can then go to over to our Heroku Dashboard, find the app again and then click the "Activity" tab. We'll see here that our application is deploying, and then is deployed successfully:



Hooray! We have now correctly setup our GitHub Actions and Heroku integration. If we push any new code to the `master` branch (which we will do in the next few chapters!) this code will automatically be deployed to Heroku... but only if the tests on Github Actions pass first! If the tests fail, no deployment will take place.

13.6. Background jobs

TODO: Write section about background jobs

13.7. Sending emails

The final thing that we need to do for our application in this chapter is to configure it to send out emails in the real world. We have the mailer code, and now what we need to do is to add the email configuration. To get there, we're going to use Heroku's addons feature to add a service called Mailgun to our application.

Mailgun is a service that lets users send emails either via traditional means (an SMTP server) or HTTP. It also provides many other features, none of which we're going to use right now. You're more than welcome to experiment with those features after we're done in this chapter, however.

Mailgun also provides a free plan on Heroku Addons, called "Starter". To setup Mailgun with our Heroku account, all we need to do is run this command in the terminal:

```
$ heroku addons:add mailgun:starter
```

When that completes successfully, you'll see a message like this:

```
Adding mailgun:starter on stark-chamber-2017... done, v17 (free)
Use `heroku addons:docs mailgun` to view documentation.
```

The documentation is good, but there's no need to read it as we'll cover the important parts right here, right now. By linking Mailgun to Heroku, a Mailgun account is automatically setup for you and its credentials are made available through environment variables in your application. If you run `heroku config` you can see them:

```
MAILGUN_API_KEY:              [redacted]
MAILGUN_SMTP_LOGIN:           [redacted]
MAILGUN_SMTP_PASSWORD:        [redacted]
MAILGUN_SMTP_PORT:            587
MAILGUN_SMTP_SERVER:          smtp.mailgun.org
```

We can use most of these environment variables to send emails through Mailgun by putting this code in `config/environments/production.rb`. Remember to specify your own application hostname as the `host` variable!

Listing 371. config/environments/production.rb

```
Rails.application.configure do
  ...
  ActionMailer::Base.delivery_method = :smtp

  host = "yourapp.herokuapp.com"

  ActionMailer::Base.smtp_settings = {
    port:           ENV['MAILGUN_SMTP_PORT'],
    address:        ENV['MAILGUN_SMTP_SERVER'],
    user_name:      ENV['MAILGUN_SMTP_LOGIN'],
    password:       ENV['MAILGUN_SMTP_PASSWORD'],
    domain:         host,
    authentication: :plain,
  }

  config.action_mailer.default_url_options = {
    host: host
  }
  ...
```

With this configuration, we're telling ActionMailer that we want to deliver emails with SMTP, and that the settings for SMTP are going to use the `MAILGUN_*` environment variables which are only available on Heroku for port, server, login and password. The `host` variable should be the root URL of your application on Heroku. The `default_url_options` bit at the end will tell ActionMailer what URL to use when it decides it wants to create a link.

Let's push this configuration up now:

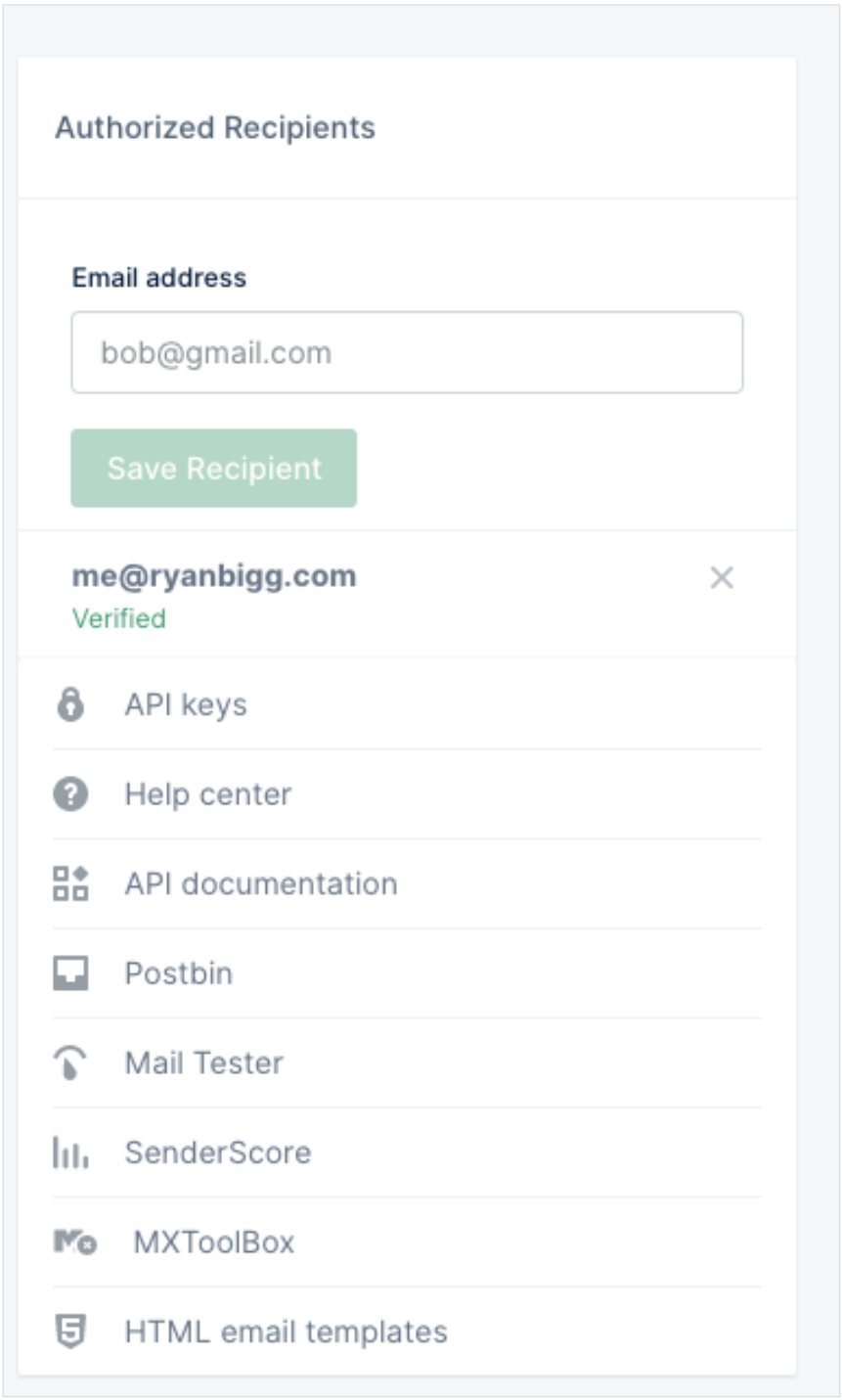
```
$ git add .
$ git commit -m "Add Mailgun configuration"
$ git push
```

Go over to Github Actions and wait for the build to complete. It will take a couple of minutes to run, but it's worth it for all it's doing: ensuring that our app is working by running our tests.

While that's happening, you'll need to add your email address to Mailgun's authorized recipients list. This is because we're currently using what Mailgun calls a "sandbox" plan—and it won't actually send emails unless we authorize our emails first.

To start, go over to your Heroku Dashboard and into your app. Then you'll need click on the Mailgun addon:

This will then take us to Mailgun itself. When you're here, go to "Sending" and then "Overview". On the right hand side of this screen will be the "Authorized Recipients" list. Add your email address there:



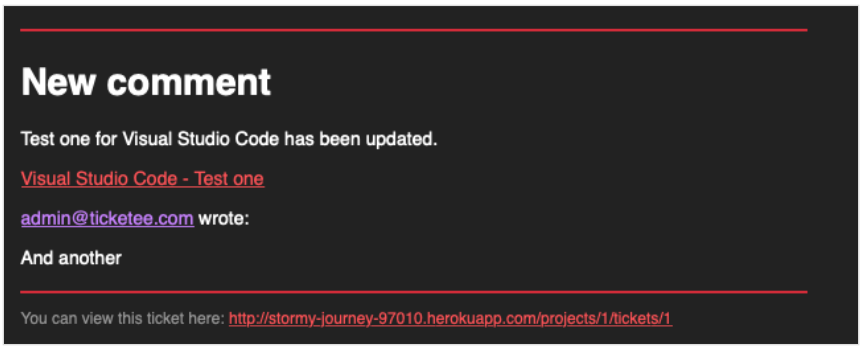
You will then receive an email from Mailgun where you will have to verify that the email is correct. Do that step now.

Once you've authorized your email address, Mailgun will be completely setup.

To test this feature we'll do the following on our production Heroku application:

1. Sign up with your real email address.
2. Create a ticket on any project.
3. Sign out from your real email address.
4. Sign in as "admin@ticketee.com".
5. Leave a comment on the ticket that you created.

After doing this little dance, you should see an email arrive in your inbox from Ticketee.



That's pretty great! Our production application is now completely setup.

13.8. Summary

In this short chapter, we've shown how we can deploy Ticketee to Heroku by either using `git push` or Github Actions and an automatic deploy. The automatic deploy path is better because it ensures that all our tests are passing beforehand and then does the pushing for us.

We also configured Mailgun to send emails to us. If we want to send emails to more than just ourselves, we'll need to pay for a real account on Mailgun. You can upgrade the account through Mailgun itself if you wish.

[75] Such as https://leanpub.com/deploying_rails_applications.
[76] Like in other Pretty Highly Prominent languages.
[77] Some projects will also have a staging environment for demonstrating new features to clients before they go into production. A staging environment is also usable by other people, but generally by a limited audience, not the general public. Staging should also have an identical configuration to production. Some advanced applications may have several staging environments, one for each new feature that is currently being developed.
[78] Remember, Yukihiro "Matz" Matsumoto is the creator of Ruby!
[79] http://12factor.net/

Appendix A: Installation Guide

Before you can get started building Rails applications, you'll need to spend some time setting up your development environment. This means installing Ruby, and Rails, on your operating system of choice.

There are various tools you can use to do so, but there's no "best" or "perfect" solution. This guide will cover the installation of our two preferred tools - **ruby-install**, to install different versions of Ruby from source, and **chruby**, to switch your installed versions of Ruby easily - and then using those tools to install Ruby, Rails, and then spin up a new working Rails app.

Windows

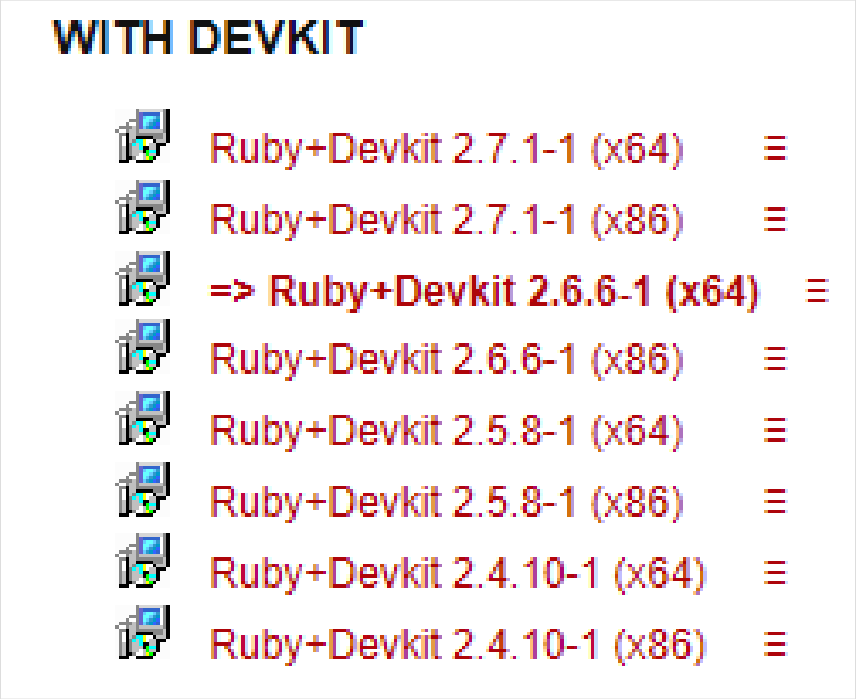
It's a bit tedious to setup Ruby and Rails on Windows. Windows doesn't have a package manager built in, and doesn't have support for some native gems. But Ruby development is still possible, and for the purposes of this book, it's perfectly fine to use.

This guide has been verified with Windows 10, Ruby 2.6.6 and Rails {rails_version}.

We'll be taking the approach of installing Ruby via RubyInstaller, and then installing DevKit that enables you to build many of the C/C++ extensions available.

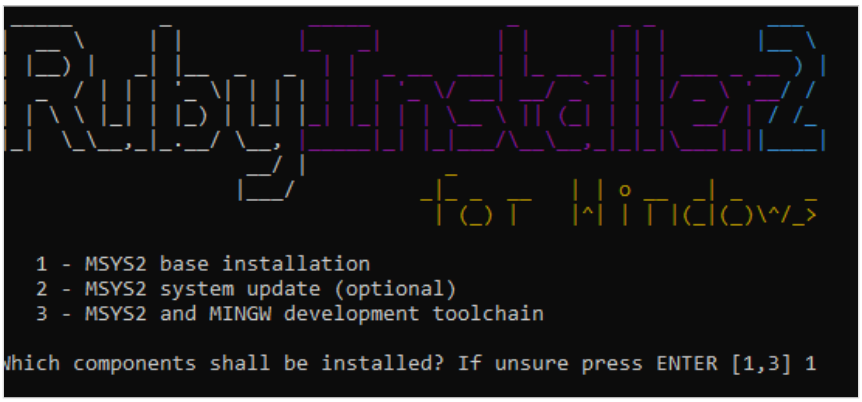
RubyInstaller

To start with, we're going to install Ruby using RubyInstaller. Visit the RubyInstaller website, and grab the latest version for your operating system - we're using the 'Ruby 2.6.6' version. It's the one in bold here:



Follow the instructions in the RubyInstaller, making sure to tick the checkbox that says "Add Ruby executables to your path" on the final step before completing the installation process.

The installer will then open a command prompt that prompts you to install the MSYS2 development tools:



For this, select the first option.

After installation, open the command prompt (which you can do by right-clicking on the Windows icon in the bottom left corner, and clicking "Command Prompt") and run `ruby -v`. You should see something like this:

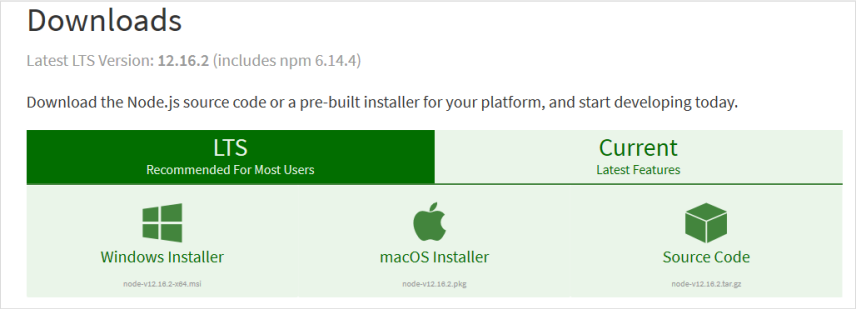
```
ruby 2.6.6p146 (2020-03-31 revision 67876) [x64-mingw32]
```

Node.js + Yarn

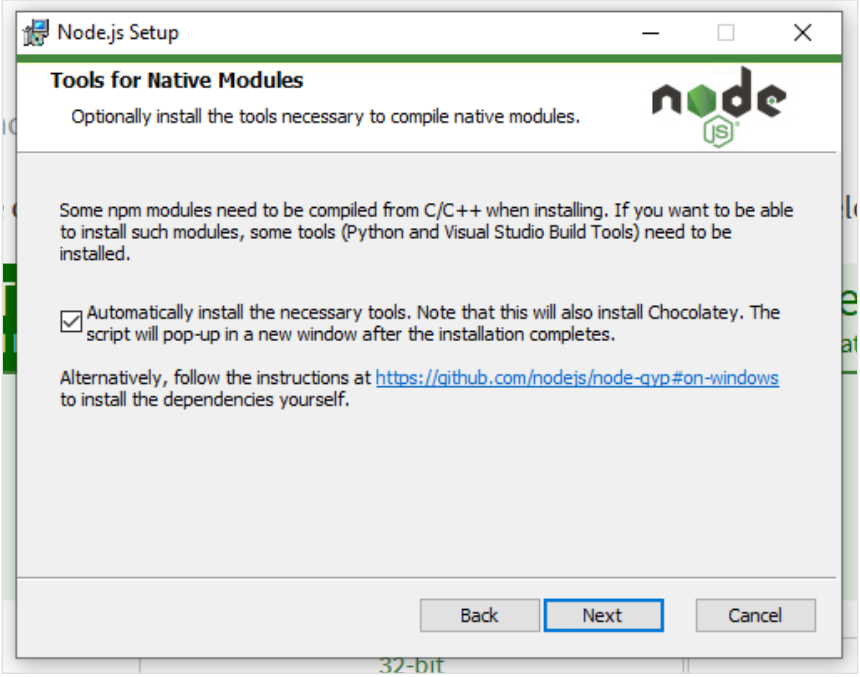
You'll also need to have another programming language called Node.js installed. Wait, why do you need another programming language to write Ruby apps? Well, way back when, many moons ago in Rails 3.1, the Rails core team introduced a new feature called the **asset pipeline**. The asset pipeline is a way to make your stylesheets, JavaScript files and other assets, much more efficient. It includes support for preprocessors like Sass, automatic concatenation and minification of files, appending of file digests to filenames to prevent misbehaving browsers from caching things they shouldn't, and much much more.

On top of this, more recent versions of Rails use a gem called Webpacker, which works in a similar manner.

To get the full power of the asset pipeline or Webpacker, you need to have Node installed. To get Node, go to https://nodejs.org/en/download, and select the "LTS" (Long Term Support) version:



Run this installer to install Node. During the setup, you'll be prompted to install additional tools:



Make sure you check this box!

Once the installer has finished, you'll be prompted to install those additional tools. Continue through these prompts until they're finished.

Open a new command prompt and then run `node -v`. If you see "v12.16.2", then Node has been installed.

Yarn

The next thing we need to install for Node is a package manager called Yarn. This is used by Rails to install files for Webpacker. We can install Yarn with this command:

```
npm install -g yarn
```

Verify this has been installed by running `yarn -v`. As of this time of writing, Yarn's version is 1.22.4. If you see any version number, then that is fine.

Git

New Rails applications start out as Git repositories, and so we will need to install Git.

If you do not do this step, when you generate a new Rails application it will only generate some of the files. You'll be missing a bunch, and your Rails app will not work.

You can do this by going to https://git-scm.com/download/win and downloading the "64-bit Git for Windows Setup" version. Run through this installer.

To verify Git has been installed, start the Command Prompt and run:

```
git --version
```

You should see something that looks like:

```
git version 2.26.2.windows.1
```

Rails

So now you have a working Ruby. What about Rails?

Ruby comes with its own package manager called RubyGems, and this is what we can use to install Rails. Gems are just little bundled-up packages of Ruby code, and Rails is just a gem. We can install Rails with this command:

```
$ gem install rails -v "~> 6.1.0" --no-document
```

This will install the latest version of the `6.1.x` branch of Rails. At the current and all of its dependencies. It'll take a while, as it figures out the dependencies, installs the gems, and then parses and installs documentation. (You can skip the documentation install by running the command with `--no-document`, eg. `gem install rails --no-document`. But hey, you might want it one day!)

When it's done, verify the installation by running `rails -v` - it should tell you that it's using a version starting with "6.0". Hooray!

Databases

However, there are a couple more things you will need to do, before starting an app. These are minor gotchas, that simply exist because Rails gives you a lot of choice in what libraries you use with your app.

One of those choices is of a database library. By default, when generating an app, Rails will try to configure the app to use SQLite, a simple file-based database system. This is a decent choice for learning, and will work out of the box on Windows.

However, once you get going with the application you may wish to switch to using a different database system, such as MySQL or PostgreSQL. These won't work out of the box on Windows - you'll need to fetch and install the database systems yourself.

If you wanted to use MySQL, you'll need to visit the MySQL website at http://mysql.com, and go to the Downloads page. There, under MySQL Community Edition, you can click through and download the MySQL Community Server. If you wanted to use PostgreSQL, you'll need to visit http://postgresql.org, and download the Windows binary package from the Downloads page.

Starting a Rails application

Once all that complicated setup is done, starting a new Rails app is trivial. Simply enter in your terminal:

```
$ rails new my_awesome_app
```

Which will create a new app using SQLite, in the `my_awesome_app` directory of your current folder. This will also install Once that's complete, you can start the Rails server:

```
$ cd my_awesome_app
$ rails server
```

Once Puma tells you it has loaded on port 3000, you can open up a browser and visit http://localhost:3000.

You should see "Yay! You're on Rails!"

Mac OS X

OS X is the second easiest of the three operating systems to install Ruby onto. The reason it's in second place is because it doesn't come with a package manager like most flavours of Linux do. Instead, you must elect to install a tool called Homebrew to manage these packages.

This guide has been verified with Mac OS X 10.15, but will **probably** work on earlier versions of OS X 10.x.

Homebrew

To start with, we are going to install Homebrew, which bills itself as "The missing package manager for OS X". The features that Homebrew provides along with its ease of use means that it has quickly gained status as the tool for managing packages on OS X. Follow the installation instructions at the bottom of http://brew.sh to install Homebrew. The instructions say to run this command in Terminal:

```
$ /bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install.sh)"
```

Go ahead and do that now.

This command will pull down the Homebrew installation script and run it using the version of Ruby which comes standard with every modern OS X install. Follow the prompts and allow the XCode tools to install. Once the XCode tools are installed, press any key in the Terminal to finish setting up Homebrew.

After we've installed Homebrew, we'll need to install two packages using it:

```
brew install openssl readline
```

These packages will be used when we setup (and use) Ruby a little later on.

asdf

Next, we'll need to install a tool that is used to manage the installation of different programming languages. We're going to use this tool to install both the Ruby and Node.js

programming languages.

Installation

To install asdf, you can go to https://asdf-vm.com/#/core-manage-asdf-vm and copy and paste the command at the top of the page. At the current time, that command is:

```
$ git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0.7.8
```

This will add some shell scripts to the `~/.asdf` directory. We can add these shell scripts to our `~/.bash_profile` file by running these commands:

```
$ echo ". $HOME/.asdf/asdf.sh" >> ~/.bash_profile
$ echo ". $HOME/.asdf/completions/asdf.bash" >> ~/.bash_profile
```

This will make it so that we have access to the `asdf` command in our terminal.

We will need to either restart our terminal, or reload this `~/.bash_profile` file to load in these changes:

```
$ . ~/.bash_profile
```

To verify that this has been installed correctly, run:

```
asdf --version
```

You should see this, or similar:

```
v0.7.8-4a3e3d6
```

Ruby

To install Ruby using `asdf`, we first need to add the `ruby` plugin to `asdf`. Do that with this command:

```
$ asdf plugin-add ruby
```

Next, we can install Ruby by running:

```
$ asdf install ruby 2.7.1
```

`asdf` and its Ruby plugin knows where to get the source code for Ruby 2.7.1, and also what dependencies are needed to compile it.

Once it's done all of that, it will start compiling Ruby, and you'll see lots and lots of random output as checks a lot of things and then compiles all of the necessary files. There doesn't appear to be an in-built way to silence this output, but you can just let it do its thing - go get a can of Diet Coke or something, it will take a few minutes.

When it's done, you'll see a nice happy:

```
Installed ruby-2.7.1 to /home/youruser/.asdf/installs/ruby/2.7.1
```

So you now have a ruby installed. How do you use it? The folder it installed to isn't part of your path, so calling `ruby` will have no effect.

We will need to set a current version of Ruby using `asdf`:

```
$ asdf global ruby 2.7.1
```

Now when we run `ruby -v`, we should see the current version of Ruby:

```
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
```

Node

Next up, we will need to install Node.js. This is used by the Webpacker gem, which is a dependency of Rails. The webpacker gem uses a JavaScript package called Webpack to compile the JavaScript assets for your application.

To install Node, we'll also use `asdf`. We first have to install the Node plugin:

```
$ asdf plugin-add nodejs
```

And we have to run this command so that `asdf` can verify Node installation files:

```
$ bash ~/.asdf/plugins/nodejs/bin/import-release-team-keyring
```

Once those two steps are done, we can install Node:

```
$ asdf install nodejs 12.16.2
```

Then we will need to configure `asdf` to use this version of Node:

```
$ asdf global nodejs 12.16.2
```

To verify that this has been installed correctly, run `node -v`. You should see this:

```
v12.16.2
```

Yarn

The next thing we need to install for Node is a package manager called Yarn. This is used by Rails to install files for Webpacker. We can install Yarn with this command:

```
npm install -g yarn
```

Verify this has been installed by running `yarn -v`. As of this time of writing, Yarn's version is 1.22.4. If you see any version number, then that is fine.

Rails

So now you have a working Ruby. What about Rails?

Ruby comes with its own package manager called RubyGems, and this is what we can use to install Rails. Gems are just little bundled-up packages of Ruby code, and Rails is just a gem.

```
$ gem install rails -v "~> 6.1.0" --no-document
```

This will install the latest from the 6.0.x branch of Rails and all of its dependencies. Whatever version this installs will be fine for this book. It'll take a while, as it figures out the dependencies, installs the gems, and then parses and installs documentation. (You can skip the documentation install by running the command with `--no-document`, eg. `gem install rails --no-document`. But hey, you might want it one day!)

When it's done, verify the installation by running `rails -v`. The output will look something like this:

```
Rails {rails_version}
```

Hooray!

Starting a new Rails app

Once all that complicated setup is done, starting a new Rails app is trivial. Simply enter in your terminal:

```
$ rails new my_awesome_app
```

Which will create a new app using SQLite, in the `my_awesome_app` directory of your current folder. This will also install Once that's complete, you can start the Rails server:

```
$ cd my_awesome_app
$ rails server
```

Once Puma starts and tells you it has loaded on port 3000, you can open up a browser and visit http://localhost:3000.

You should see "Yay! You're on Rails!"

Linux

Linux is perhaps the easiest of the three operating systems to get Ruby running on, but Rails is

a little more difficult. All Linux flavours come with decent package management built in, so getting the necessary pre-requisites is easy, safe, and guaranteed not to mess with software already installed on your system.

This guide has been verified with both Ubuntu 18.04, and presumes that your desktop is fully up to date, with all updates installed, but no other installed pre-requirements.

To make sure your system is up to date, run this command in a terminal:

```
$ sudo apt-get update
```

build-essential

The first thing we will need to install is a package called `build-essential`. This package includes a bunch of tools that will be used to install Ruby, and also by Ruby to compile C extensions. Without these tools, you won't be able to install or let alone use Ruby.

To install these tools, run this command:

```
sudo apt-get install build-essential openssl libssl-dev libsqlite3-dev
```

Git

The second thing we will need to install is Git. This will be used for version control for your Rails application, and is also used to setup a tool called `asdf` in a little while.

We can install this with:

```
$ sudo apt-get install git
```

To verify that this has been installed, run:

```
$ git version
```

You should see something like:

```
git version 2.17.1
```

asdf

Next, we'll need to install a tool that is used to manage the installation of different programming languages. We're going to use this tool to install both the Ruby and Node.js programming languages.

Installation

To install asdf, you can go to https://asdf-vm.com/#/core-manage-asdf-vm and copy and paste the command at the top of the page. At the current time, that command is:

```
$ git clone https://github.com/asdf-vm/asdf.git ~/.asdf --branch v0.7.8
```

This will add some shell scripts to the `~/.asdf` directory. We can add these shell scripts to our `.bashrc` file by running these commands:

```
$ echo ". $HOME/.asdf/asdf.sh" >> ~/.bashrc
$ echo ". $HOME/.asdf/completions/asdf.bash" >> ~/.bashrc
```

This will make it so that we have access to the `asdf` command in our terminal.

We will need to either restart our terminal, or reload this `~/.bashrc` file to load in these changes:

```
$ . ~/.bashrc
```

To verify that this has been installed correctly, run:

```
asdf --version
```

You should see this, or similar:

```
v0.7.8-4a3e3d6
```

Ruby

To install Ruby using `asdf`, we first need to add the `ruby` plugin to `asdf`. Do that with this command:

```
$ asdf plugin-add ruby
```

Next, we can install Ruby by running:

```
$ asdf install ruby 2.7.1
```

`asdf` and its Ruby plugin knows where to get the source code for Ruby 2.7.1, and also what dependencies are needed to compile it.

Once it's done all of that, it will start compiling Ruby, and you'll see lots and lots of random output as checks a lot of things and then compiles all of the necessary files. There doesn't appear to be an in-built way to silence this output, but you can just let it do its thing - go get a can of Diet Coke or something, it will take a few minutes.

When it's done, you'll see a nice happy:

```
Installed ruby-2.7.1 to /home/youruser/.asdf/installs/ruby/2.7.1
```

So you now have a ruby installed. How do you use it? The folder it installed to isn't part of your path, so calling `ruby` will have no effect.

We will need to set a current version of Ruby using `asdf`:

```
$ asdf gobal ruby 2.7.1
```

Now when we run `ruby -v`, we should see the current version of Ruby:

```
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
```

Node

Next up, we will need to install Node.js. This is used by the Webpacker gem, which is a dependency of Rails. The webpacker gem uses a JavaScript package called Webpack to compile the JavaScript assets for your application.

To install Node, we'll also use `asdf`. We first have to install the Node plugin:

```
$ asdf plugin-add nodejs
```

And we have to run this command so that `asdf` can verify Node installation files:

```
$ bash ~/.asdf/plugins/nodejs/bin/import-release-team-keyring
```

Once those two steps are done, we can install Node:

```
$ asdf install nodejs 12.16.2
```

Then we will need to configure `asdf` to use this version of Node:

```
$ asdf global nodejs 12.16.2
```

To verify that this has been installed correctly, run `node -v`. You should see this:

```
v12.16.2
```

Yarn

The next thing we need to install for Node is a package manager called Yarn. This is used by Rails to install files for Webpacker. We can install Yarn with this command:

```
npm install -g yarn
```

Verify this has been installed by running `yarn -v`. As of this time of writing, Yarn's version is 1.22.4. If you see any version number, then that is fine.

Rails

So now you have a working Ruby. What about Rails?

Ruby comes with its own package manager called RubyGems, and this is what we can use to install Rails. Gems are just little bundled-up packages of Ruby code, and Rails is just a gem.

```
$ gem install rails -v "~> 6.1.0" --no-document
```

This will install the latest from the 6.1.x branch of Rails and all of its dependencies. Whatever version this installs will be fine for this book. It'll take a while, as it figures out the dependencies, installs the gems, and then parses and installs documentation. (You can skip the documentation install by running the command with `--no-document`, eg. `gem install rails --no-document`. But hey, you might want it one day!)

When it's done, verify the installation by running `rails -v`. The output will look something like this:

```
Rails 6.0.2.1
```

Hooray!

Starting a new Rails app

Once all that complicated setup is done, starting a new Rails app is trivial. Simply enter in your terminal:

```
$ rails new my_awesome_app
```

Which will create a new app using SQLite, in the `my_awesome_app` directory of your current folder. This will also install Once that's complete, you can start the Rails server:

```
$ cd my_awesome_app
$ rails server
```

Once Puma starts and tells you it has loaded on port 3000, you can open up a browser and visit http://localhost:3000.

You should see "Yay! You’re on Rails!"

Appendix B: Why Rails?

A common question in the Ruby on Rails community from newcomers is "Why Ruby?" or "Why Rails?". In this Appendix, this question will be answered with a couple of key points about why people should be using Ruby on Rails over other frameworks, covering such things as the culture and community standards.

Ruby is an exceptionally powerful language that can be used for short scripts up to full-featured web applications, such as those built with Ruby on Rails. It's clean syntax and its focus on making programmers happy are two of the many major points that have generated a large community of people who use it. There's hobbyists who use it just for the sake of it right up to hardcore people who swear by it. However, Ruby (and by extension, Rails) should not be used as "golden hammers". Not all problems are solvable by Ruby or Rails, but the chance of running into one of these situations is extremely low. People who have used other languages before they have come to Ruby suggest that "Ruby just makes more sense"[80].

The speed of which that you can develop applications using Ruby on Rails is demonstrably faster than other languages. An application that has taken 4 months to build in Java could be done in 3 weeks in Rails, for example. This has been proven again and again. Rails even claims up front on http://rubyonrails.org that "Ruby on Rails is optimized for programmer happiness and sustainable productivity".

The Ruby and Rails communities have a consistent focus on self-improvement. Over the last couple of years we've seen developments such as the improvements from Rails 2 to Rails 3, Passenger (covered in Chapter 13) and Bundler. All of these have vastly improved the ease of development that comes naturally to Ruby. Other developments have focussed on other areas, such as the RSpec, Capybara and [81] gems (featured prominently in this book) which focus on making testing exceptionally easier for Ruby developers. By consistently improving, things are becoming easier and easier for Ruby developers every year.

Along the same vein of self-improvement is an almost zealot-like focus on testing, which is code that tests other code. While this may seem silly to begin with, it helps us make less silly mistakes and provides the groundwork for us to test the fixes for any bugs that come up in our system. Ruby, just like every other language, is no good at preventing buggy code. That's a human trait that is unavoidable.

The shift away from SVN to the wonderful world of distributed version control was also a major milestone, with GitHub (a Rails application!) being created in early 2008. Services such

GitHub have made it easier than ever for Ruby developers to collaborate on code across cultures. As an example of this, you only need to look at the authors of commits on the Rails project to see the wide gamut of people.

Don't just take it from us. Here's a direct quote from somebody who had only been using Rails for a few days:

> ```
> When I am programming with Ruby I think I'm making magic.
> ```
>
> — New person

While Ruby isn't quite the magic of fairy tales, you'll find young and old, experienced and not-so-experienced people all claiming that it's just a brilliant language to work with. As Yukihiro Matsumuto (the creator of the language) says: Ruby is designed to make programmers happy. Along the same lines, the Rails claim we saw earlier to be "optimized for programmer happiness and sustainable productivity" is not smoke and mirrors either. You can be extremely happy and productive while using Rails, compared with other frameworks.

Let's dive in a little deeper into the reasons why Rails (the framework) and Ruby (the language) are just so great.

Reason #1: The sense of community

The Rails community is like none-other on the planet. There is a large sense of togetherness in the community with people freely sharing ideas and code through services such as GitHub and RubyGems (see Reason #2). Such examples of this are the vibrant community on the Freenode IRC network (irc.freenode.net) where the main #rubyonrails channel is primarily used for asking questions about Rails. Anybody can come into the channel and ask a question and receive a response promptly from one of the other people who visit the channel. There's no central support authority, it's a group of volunteers who are voluntarily [82] their time to help strangers with problems, without asking for money or expecting anything else in return.

There's also a large support community focussed around Stack Overflow (http://stackoverflow.com) and other locations such as the Ruby on Rails Talk mailing list (http://groups.google.com/group/rubyonrails-talk) and Rails Forum (http://railsforum.org). Not to mention, there's also the RailsBridge (http://railsbridge.org) organisation which aims to bridge the gap between newbies and experienced developers.

All of these different areas of the internet share a common goal: be nice to the people who are asking for help. One mantra in the Ruby community is "Matz is nice always, so we are nice", often abbreviated to "MINASWAN". People in the Ruby and Rails communities are incredibly nice to everyone.

Another example of the excellent community around Ruby on Rails is the number of conferences and gatherings held worldwide. The smallest of them are the intimate hack sessions where people work together on applications and share ideas in a room. Slightly bigger and more organised than that are the events such as Railscamps (http://railscamps.org) which have about 150 people attend and run from Friday-Monday, with interesting talks given on the Saturdays and Sundays. The largest however is Railsconf, which has about 2,000 people in attendance.

There are hundreds of thousands, if not millions of people using Ruby on Rails today, building great web applications with it and building the best web framework community on the planet.

Reason #2: The speed and ease of development

The speed of how quickly you are able to develop a Ruby on Rails application is definitely one of the main reasons that people gravitate towards (and stick with) the framework.

One documented case of this is that of a team which had developed an application using a Java-based framework which took 4 months. When that application became difficult to maintain, alternative languages and frameworks were sought, with Ruby and Ruby on Rails found to fit the bill adequately. The team re-implemented all the features of the original Java-based application within 3 weeks, with less code and more beautiful code.

Ruby on Rails follows a paradigm known as "convention over configuration". This paradigm is adopted not only by Rails, but other modern web frameworks. Rails is designed in such a way that it takes care of the normal configuration that you may have to do with other frameworks, leaving it up to you to get down to coding real features for your application.

One example of this "convention over configuration" is the mapping between classes designed to interact with the database and the tables related to these classes. If the class is called Project then it can be assumed by Rails (and the people coding the application) that the related table is going to be called projects. However this can be configured using a setting in the class if that table name is not desired.

Reason #3: RubyGems

This third point is more about a general boon to the community of Ruby, but it plays a key role in developing Rails applications.

As we stated before, the culture of the Rails community is one of self-improvement. There's people who are consistently thinking of new ways to make other people's lives better. One of these ways is the RubyGems system which allows people to share libraries in a common format. By installing a gem, a user is able to use its code along with their own code. There's gems such as the json gem which is used for parsing JSON data, nokogiri for parsing XML and of course the Rails suite of gems.

Previously, gems were hosted on a system known as RubyForge which was unstable at times. In July 2009, Nick Quaranto, a prominent Rubyist, created the RubyGems site we know today: http://rubygems.org. This is now the primary nexus for hosting and downloading gems for the Ruby community, with RubyForge now playing second fiddle. The site Nick created provides an easy way for people to host gems that other people can use, freely. Now isn't that just awesome?

Working with gems on a project used to be tough. To find out what gems a project used they had to be listed somewhere and there were often times where the tools used to install these gems would not work; either by installing the wrong gems or simply refusing to work at all. Then along came Bundler. The Bundler gem provides a standardized way across all Ruby projects for managing gem dependencies. It's a gem to manage the gems that projects use. We can list the gems we want our project to use in a special file known as the Gemfile. Bundler can then interpret (when the bundle install is ran) the special syntax in this file to figure out what gems (and the dependencies, and their dependencies' dependencies) need installing and then goes about doing it. Bundler solves the gem dependency hell previously witnessed in the Ruby and Rails communities in a simple fashion.

In addition to this, having different Ruby versions running on the same machine used to be difficult and involve a lot of hacking around. Then another prominent Rubyist, Wayne E. Seguin, created a gem called RVM ("Ruby Version Manager") which allows for simplistic management of the different

All in all, RubyGems and it's ecosystem is very well thought-out and sustainable. It provides an accessible way for people to share their code in a free manner and serves as one of the foundations of the Ruby language. All of this work has been done by the exceedingly great

community which is made up of many different kinds of people, perhaps one day even including people who are reading this Appendix.

Reason #4: Emphasis on Testing

Within the Ruby and Rails community there's a huge focus on writing great, maintainable code. To help with this process, there's also a big focus on test driven development along with a mantra of "red/green/refactor". This mantra describes the process of test driven development: we write tests which are then failing (usually indicated by the color red), write the code to make those tests pass (indicated by green) and then clean up (refactor) the code in order to make it easier for people to know what it's doing. We covered this process in Chapter 2.

Because Ruby is interpreted rather than compiled like other languages, we cannot rely on compilers to pick up errors in our code. Instead, we write tests that describe functionality before it's implemented. Those tests will fail initially, but as we write that functionality those tests will pass. In some situations (such as tests written using the Cucumber gem), these tests can be read by people who requested a feature of our application and can be used as a set of instructions explaining exactly how this feature works. This is a great way to verify that the feature is precisely what was requested.

As we write these tests for the application we provide a safety net for if things go wrong. This collection of tests is referred to as a test suite. When we develop a new feature we have the tests that we wrote before the feature to prove that it's working as we originally thought it should.If we want to make sure that the code is working at some point in the future then we have the test suite to fall back on.

If something does go wrong and it's not tested, we've got that base to build upon. We can write a test for an unrealized situation—a process known as regression testing--and always have that test in the future to ensure that the problem does not crop up again.

[80] Quote attributed to Sam Shaw from Railsconf 2011
[81] A quick nod to the aruba gem: http://github.com/aslakhellesoy/aruba, which is used extensively to test RSpec and Cucumber's CLI (command-line interfaces), but can also be used to test other CLIs.
[82] Too much "volunteer" usage, perhaps. It was voluntary.