Rails 4 IN ACTION

Ryan Bigg
Yehuda Katz
Steve Klabnik

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Rails 4 in Action MEAP version 11
Revised Edition of Rails 3 in Action

Copyright 2013 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

brief contents

Chapter 1 Ruby on Rails, the framework

Chapter 2 Testing saves your bacon

Chapter 3 Developing a real Rails application

Chapter 4 Oh CRUD!

Chapter 5 Nested resources

Chapter 6 Authentication

Chapter 7 Basic access control

Chapter 8 Fine-grained access control

Chapter 9 File uploading

Chapter 10 Tracking state

Chapter 11 Tagging

Chapter 12 Sending email

Chapter 13 Designing an API

Chapter 14 Deployment

Chapter 15 Alternative authentication

Chapter 16 Basic performance enhancements

Chapter 17 Rack-based applications

Appendix A Why Rails?

Appendix B Tidbits

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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. Great developers are happy developers.

If you're wondering who uses Rails, well there's plenty of companies out there. There's Twitter, Hulu, 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 pretty solid foundations of an application and then build on that 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 ("Matz") of Japan. Ruby 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.

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, with it being released under the MIT License[1].

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 1 The MIT license: http://en.wikipedia.org/wiki/MIT_License

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, through to bug fixes, all the way up to adding new features to the framework.

This book is written for version 4.0.0 of the framework, which is the latest version of Rails. If you've used Rails 3.2, you'll find that much feels the same, yet Rails has learned some new tricks, as well. There will be an appendix at the end of the book giving you a quick overview of what's new.

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. This generator 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 of this is already there for you, you won’t be spending your time configuring the way your application is laid out. It’s done for you.

How rapidly can you develop a Ruby on Rails application? Take the annual *Rails Rumble* event. This event brings together small teams of one to four developers around the world to develop Ruby on Rails[2] applications in a 48-hour period. Using Rails, these teams deliver amazing web applications in just two days. [3] Another great example of rapid development of a Rails application is the 20-minute blog screencast recorded by Yehuda Katz.[4] This screencast takes you from having nothing at all to having a basic blogging and commenting system.

Footnote 2 And now other Ruby-based web frameworks, such as Sinatra.

Footnote 3 To see an example of what has come out of previous Rails Rumbles, take a look at their alumni archive: http://r09.railsrumble.com/entries

Footnote 4 20-minute blog screencast: http://vimeo.com/10732081

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. The similarity between the applications is so close that working on different Rails applications is not tremendous. If and when you jump between Rails

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

applications, you don’t have to relearn how it all connects—it’s mostly the same. The Rails ecosystem may seem daunting at first, but Rails conventions allow even the new to seem familiar very quickly, smoothing the learning curve substantially.

The core features of Rails are split up into many different parts, such as *Active Record*, *Active Support*, *Action Mailer*, and *Action Pack*.[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 your business.

Footnote 5 These gems share the same version number as Rails, which means when you're using Rails 4.0, you're using the 4.0 version 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.

Ever wished for a built-in way of writing automated tests for your web application? Ruby on Rails has you covered with *Minitest*, part of Ruby’s standard library. It’s incredibly easy to write automated test code for your application, as you’ll see throughout this book. Testing your code saves your bacon in the long term, and that’s a fantastic thing. We touch on Minitest in the next chapter before moving on to RSpec, which is a testing framework that is preferred the majority of the community over Minitest and is a little easier on the eyes too.

In addition to testing frameworks, the Ruby community has produced several high-quality libraries (called RubyGems, or gems for short) for use in your day-to-day development with Ruby on Rails. Some of these libraries add additional functionality to Ruby on Rails; others provide ways to turn alternative markup languages such as Markdown and Textile into HTML. Usually, if you can think 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.

1.1.2 Common terms

You’ll hear a few common Ruby on Rails terms quite often. This section explains what they mean and how they relate to a Rails application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

MVC

The *M*odel-*V*iew-*C*ontroller (*MVC*) paradigm is not unique to Ruby on Rails but provides much of the core foundation for a Ruby on Rails application. This paradigm is designed to keep the logically 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 on where things should go. In Rails, a specific directory structure encourages all 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 sub-directories: 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 are 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 cannot.

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 *E*mbedded *Ru*b*y* (*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.

The assets, helpers, and mailers directories aren't part of the MVC paradigm, but they are important parts of Rails.

The *assets* directory is for the static assets of the application, such as JavaScript files, images, and Cascading Style Sheets (CSS) for making the application look

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

pretty. We look more closely at this in chapter 3.

The *helpers* directory is a place to put Ruby code (specifically, modules) that provide helper methods for just the views. These helper methods can help with complex formatting that would otherwise be messy in the view or is used in more than one place.

Finally, *mailers* is a home for the classes of our application that deal with sending email. In previous versions of Rails, these classes were grouped with models but have since been given their own home. We look at them in chapter 11.

REST

MVC in Rails is aided by *Re*presentational *S*tate *T*ransfer (*REST*)[6], a routing paradigm. REST is the convention for *routing* in Rails. When something adheres to this convention, it's said to be *RESTful*. Routing in Rails refers to how requests are routed within the application itself. You benefit greatly by adhering to these conventions, because Rails provides a lot of functionality around RESTful routing, such as determining where a form can submit data.

Footnote 6 http://en.wikipedia.org/wiki/Representational_state_transfer

1.1.3 Rails in the wild

One of the most well-known sites that runs Ruby on Rails is GitHub. Github is 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 a million repositories 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!

You don't have to build huge applications with Rails, either. There is a Rails application that was built for the specific purpose of allowing people to review 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 chapters for the book and leave notes on each element in the book, 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

1.2 Developing your first application

We covered the theory behind Rails and showed how quickly and easily you can develop an application. Now it's your turn to get an application going. This application will be a simple application that can be used to track items that have been purchased, tracking just the name and the price for an item. In the next section, you'll learn how to install Rails and use the scaffold generator that Rails comes with.

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 you use RVM (http://rvm.io) to install Ruby and RubyGems. It is a favored solution for many in the community because it is simple to get started with. You can install it by following the instructions on the https://rvm.io/rvm/install/ page. If you prefer a different tool, such as `chruby` or `rbenv`, that works fine as well. These options are a bit more complex to get started with, but some developers prefer them. Whichever way you choose, please don't install from your package manager, if you're on Linux. Installing from a package management system such as Ubuntu's Aptitude has been known to be broken.[7] After installing RVM, you must run this command to install a 2.0.0 version of Ruby:

Footnote 7 Broken Ubuntu Ruby explained here:
http://ryanbigg.com/2010/12/ubuntu-ruby-rvm-rails-and-you/

```
$ rvm install 2.0.0
```

To use this version of Ruby, you would need to use `rvm use 2.0.0` every time you wished to use it or else set up a .rvmrc file in the root of your project, which is explained on the RVM site in great detail. Alternatively, you can set this

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

version of Ruby as the default with the command `rvm use --default 2.0.0`, and use `rvm use system` if you ever want to swap back to the system-provided Ruby install if you have one.

If you're on Windows, you can't use RVM. We would recommend the use of the Rails Installer program (http://railsinstaller.org) from Engine Yard, or installing the Ruby 2.0.0 binary from http://ruby-lang.org or http://rubyinstaller.org as an alternative to RVM.

Next, you need to install the rails gem. The following command installs both Rails and its dependencies. If you're using the Rails installer you will not need to run this command as Rails will already be installed.

```
$ gem install rails -v 4.0.0
```

Okay, let's check we've got everything. Type these commands, and check out the responses.

```
$ ruby -v
ruby 2.0.0p195 (2013-05-14 revision 40734) [x86_64-linux]
$ gem -v
2.0.2
$ rails -v
Rails 4.0.0
```

If you see something that looks close to this, you're good to go! These particular values are the ones that I'm using right now: as long as you have Ruby 2.0 or later, Rails 4.0 or later, and RubyGems 2.0 or later, everything should be fine.

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

1.2.2 Generating an application

With Rails now 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.

WARNING **Don't use reserved words for application naming**

You can call your application anything you wish, but it can't be given the same name as 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` from within the framework itself.

When you use an invalid application name, you'll see an error like this:

```
$ rails new rails
Invalid application name rails, constant Rails is already in use.
Please choose another application name.
```

The application that you're going to generate will be able to record purchases you have 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 of the directories and files generated here 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 of learning.

1.2.3 Starting the application

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

```
$ cd things_i_bought
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ bin/rails server
```

The `bin/rails server` (or `bin/rails s`, for short) starts a web server on your local address on port 3000 using a Ruby standard library web server known as WEBrick. It will say its "starting in development on http://0.0.0.0:3000," which indicates to us that the server will be available on port 3000 on all network interfaces of this machine[8]. To connect to this server, go to http://localhost:3000 in your favorite browser. You'll see the "Welcome aboard" page, which is so famous to Rails (Figure 1.1).

Footnote 8 This is what the `0.0.0.0` address represents. It is not an actual address, so to speak, and so `localhost` or `127.0.0.1` should be used.

Careful readers will notice that previously, we used the `rails` command to generate the application, and we use `bin/rails` here. Here's the rule of thumb: use `rails` to generate applications, and use `bin/rails` everywhere else. We'll talk more about the 'binstub' later on. For now, just know that most commands are prefixed with `bin/`.

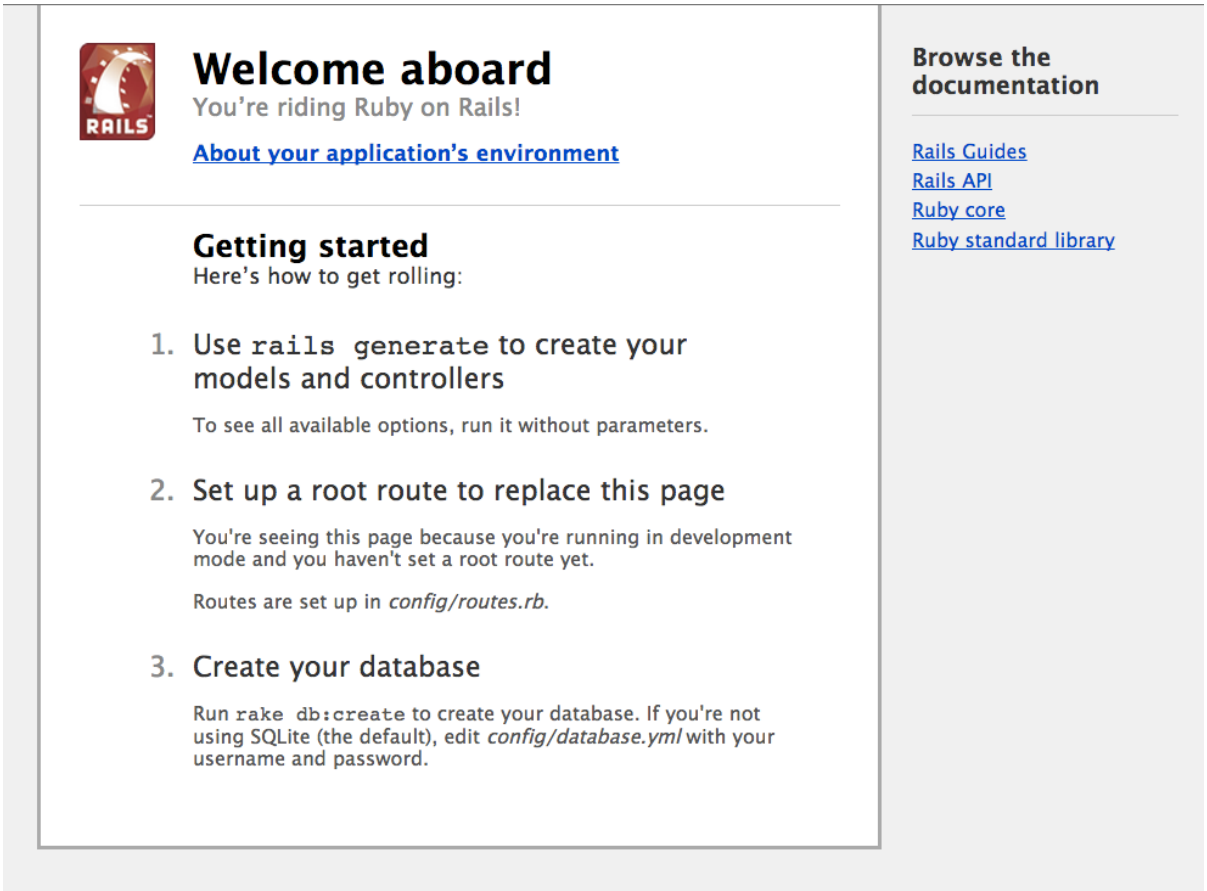

Figure 1.1 Welcome aboard!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

On the right-hand side of this page, there's four links to more documentation for Rails and Ruby. The first link will take you to the official guides page, which will give you great guidance that complements the information in this book. The second link will take you to the Rails API, where you can look up the documentation for classes and methods within Ruby. The final two links will take you to documentation about Ruby itself.

If you click About Your Application's Environment, you'll find your Ruby, RubyGems, Ruby on Rails, and Rack versions and other environmental data. One of the things to note here is that the output for Environment is Development. Rails provides three environments for running your application: *development*, *test*, and *production*. How your application functions can depend on the environment in which it is running. For example, in the development environment, classes are not cached, so if you make a change to a class when running an application in development mode, you don't need to restart the server, but the same change in the production environment would require a restart.

1.2.4 Scaffolding

To get started with this Rails application, you generate a *scaffold*. Scaffolds in Rails provide a lot of basic functionality and are generally used just as a temporary structure to get started, rather than for full-scale development. Let's generate a scaffold by running this command:

```
$ bin/rails generate scaffold purchase name:string cost:float
```

When you used the `rails` command earlier, it generated an entire Rails application. You can use this command inside of 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 `bin/rails g` as a shortcut to `bin/rails generate`.

The `scaffold` command generates a model, controller, views and tests based on the name passed after scaffold in this command. These are the three important parts needed for your purchase tracking. The model provides a way to interact with

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

a database. The controller interacts with the model to retrieve and format its information and defines different actions to perform on this data. The views are rendered by the controller and display the information collected within them.

Everything after the name for the scaffold are 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 a `name` and `cost` field, which are a string and a float, respectively.[9] To create this table, the scaffold generator generates what's known as a *migration*. Let's have a look at what migrations are.

Footnote 9 Usually you wouldn't use a float for storing monetary amounts because it can lead to incorrect rounding errors. Generally, you store the amount in cents as an integer and then do the conversion back to a full dollar amount. In this example, you use float because it's easier to not have to define the conversion at this point.

1.2.5 Migrations

Migrations are used in Rails as a form of version control for the database, providing a way to implement incremental changes to the schema of the database. They are usually created along with a model, or by running the migration generator. 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. When two developers are working on separate features of an application and both generate a new migration, this timestamp will stop them from clashing. Let's open the only file in db/migrate now and see what it does. Its contents are shown in the following listing.

Listing 1.1 db/migrate/[date]_create_purchases.rb

```
class CreatePurchases < ActiveRecord::Migration
  def change
    create_table :purchases do |t|
      t.string :name
      t.float :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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Inside the `change` method, you use database-agnostic commands to create a table. When this migration is run forward, it will create a table called "purchases", with a "name" column that's a string, a "cost" column that's a float, 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 is a feature that is built into Active Record. If there are fields present with these names (or "created_on" and "updated_on"), they will be automatically updated when necessary.

When the migration is reverted, Rails will know how to undo it because it is a simple table creation. The opposite of creating a table is to drop that table from the database. If the migration was more complex than this, you would need to split it out into two methods, one called `up` and one called `down` that would tell Rails what to do in both cases. Rails is usually smart enough to figure out what you want to do, but sometimes it's not clear and you will need to be explicit. You'll see examples of this in later chapters.

To run the migration, type this command into the console:

```
$ bin/rake db:migrate
```

This command runs the `up` part of this migration. Because this is your first time running migrations in your Rails application, and because you're using a SQLite3 database, Rails first creates the database in a new file at db/development.sqlite3 and then creates the `purchases` table inside that. When you run `bin/rake 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 want to roll back this migration, you'd use `bin/rake db:rollback`, which rolls back the latest migration by running the `down` method of the migration.[10]

Footnote 10 If you want to roll back more than one migration, use the `bin/rake db:rollback STEP=3` command, which rolls back the three most recent migrations.

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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

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 `bin/rake db:schema:load` command.

Footnote 11 where `[timestamp]` in this example is an actual timestamp formatted like `YYYYmmddHHMMSS`.

With your database set up with a `purchases` table in it, let's look at how you can add rows to it through your application.

1.2.6 Viewing and creating purchases

Ensure that your Rails server is still running, or start a new one up by running `bin/rails s` or `bin/rails server` again. Start your browser now and go to http://localhost:3000/purchases. You'll see the scaffolded screen for purchases, as shown in Figure 1.2.

Figure 1.2 Purchases

No purchases are listed yet, so let's add a new purchase by clicking New Purchase.

In Figure 1.3, you see two inputs for the fields you generated.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

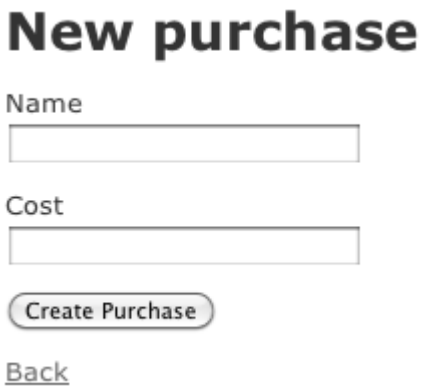

Figure 1.3 A new purchase

This page is the result of `new` action from the `PurchasesController` controller. 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 1.2 app/views/purchases/new.html.erb

```
<h1>New purchase</h1>

<%= render 'form' %>

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

This is an ERB file, which allows you to mix HTML and Ruby code to generate dynamic pages. The <%= beginning of an ERB tag indicates that the result of the code inside the tag will be output to the page. If you want the code to be evaluated but not output, you use the <% tag, like this:

```
<% some_variable = "foo" %>
```

If you were to use `<%= some_variable = "foo" %>` here, the `some_variable` variable would be set and the value output to the screen. By using <%, the Ruby code is evaluated but not output.

The `render` method, when passed a string as in this example, renders a *partial* . A partial is a separate template file that we can include into other templates to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

repeat similar code. We'll take a closer look at these in chapter 3.

The `link_to` method generates a link with the text of the first argument (`Back`) and with an `href` attribute specified by the second argument (`purchases_path`), which is simply `/purchases`. How this works is explained a little later on when we look at how Rails handles routing.

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

Listing 1.3 first half of app/views/purchases/_form.html.erb

```
                                                                    (1)
<%= form_for(@purchase) do |f| %>
  <% 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 |msg| %>
        <li><%= msg %></li>
      <% end %>
      </ul>
    </div>
  <% end %>
```

This half is responsible for defining the form by using the `form_for`(1) helper. The `form_for` method is passed one argument—an instance variable called `@purchase`—and with `@purchase` it generates a form. This variable comes from the `PurchasesController`'s `new` action which is shown in the following listing.

Listing 1.4 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, which initializes a new object of this model. By simply calling `new` on the model, it does not create a new record in the database. Instead, it only just initializes a new instance of the `Purchase` class.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `@purchase` variable is then automatically passed through to the view by Rails.

So far, all of 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_for` 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 did not pass the validation requirements set in the model. If any errors exist, they're rendered by the content inside this `if` statement. Validation is a concept covered shortly.

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

Listing 1.5 second half of app/views/purchases/_form.html.erb

```
<div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :cost %><br />
    <%= f.text_field :cost %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

Here, the `f` object from the `form_for` 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.

Let's fill in this form now and press the submit button. You should now see something similar to Figure 1.4

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

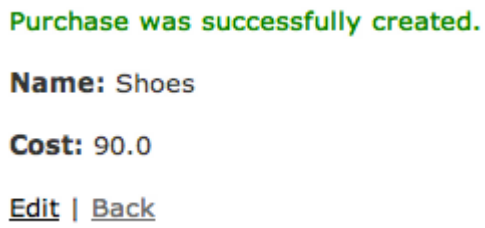

Figure 1.4 Your first purchase

What you see here is the result of your posting: a successful creation of a `Purchase`. Let's see how it got there. This submit button posts the data from the form to the `create` action, which looks like the following listing.

Listing 1.6 The create action of PurchasesController

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

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

Here, you use the `Purchase.new` you first saw used in the `new` action. But this time you pass it an argument of `purchase_params`, which is actually another method, defined below. That method calls `params`❶ (short for *parameters*) is a method that returns the parameters sent from your form in a `Hash` -like object. We'll talk more about why you need this little dance later, 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 12 The Rails word for fields.

Inside the `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 interprets what the `@purchase` object is and determines the path required is `purchase_path` because it's an object of the `Purchase` model. This path takes you to the `show` action for this controller. The `:notice` option passed to the `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 Figure 1.4

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` action's template again. We should note here that this doesn't call the `new` action/method again[13] but only renders the template.

Footnote 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 rerendering the template, you can display information about the object if the object is invalid.

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

1.2.7 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 above a certain other number. You're going to write your first code for this application and implement both of these things now.

Open up your `Purchase` model and change the whole file to what's shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.7 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 is greater than 0.

Let's test out these validations by going back to http://localhost:3000/purchases, clicking New Purchase, and clicking Create Purchase. You should see the errors shown in Figure 1.5

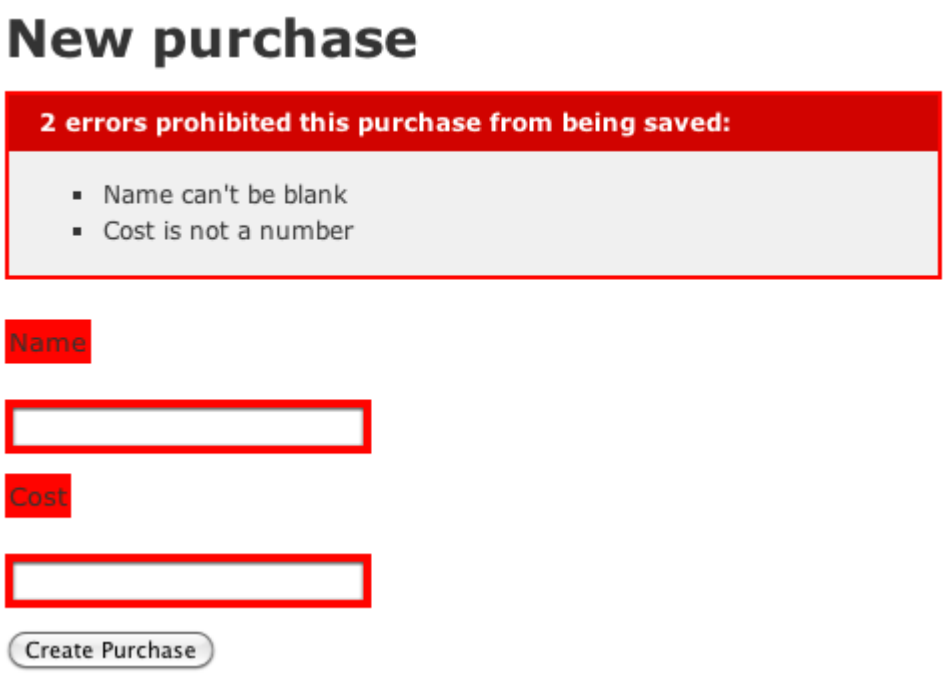

Figure 1.5 Errors on purchase

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, `-100` for the cost fields, and press Create Purchase. You should get a different error for the cost field now, as shown in Figure 1.6

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

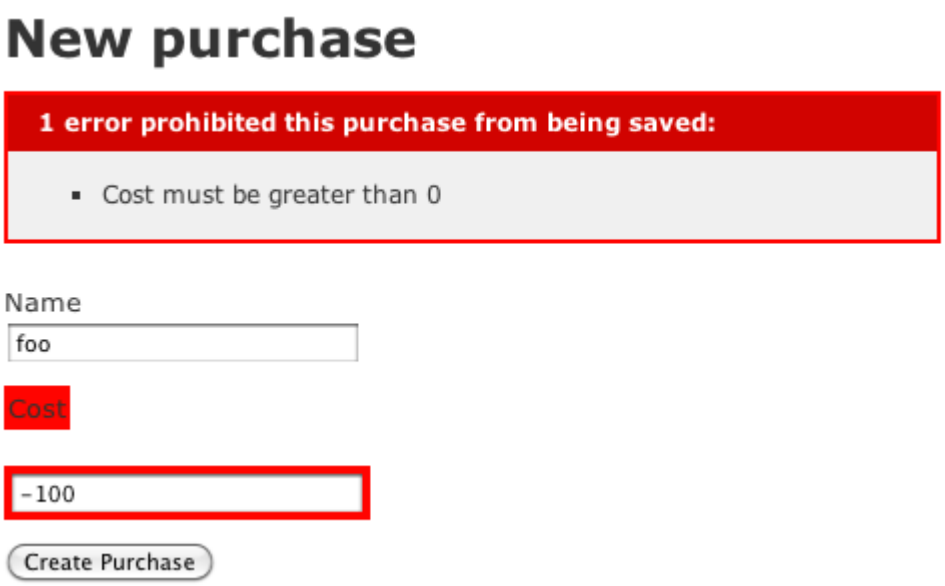

Figure 1.6 Cost must be greater than 0

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

1.2.8 Showing off

This action displays the content such as shown in Figure 1.7

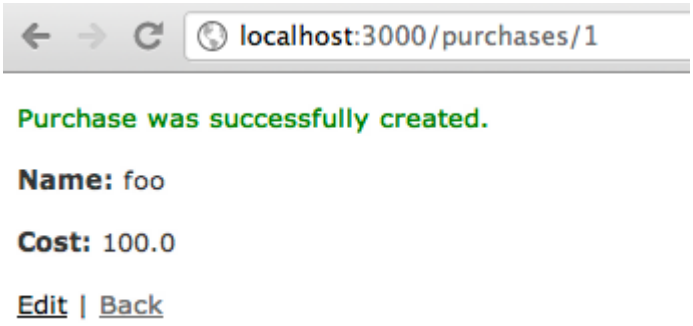

Figure 1.7 A single purchase

The number at the end of the URL is the unique numerical ID for this purchase. But what does it mean? Let's look at the view for this `show` action now, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.8 app/views/purchases/show.html.erb

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

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

<p>
  <b>Cost:</b>
  <%= @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 simply calling them as methods on your `@purchase` object. This object is defined in your `PurchasesController`'s `show` action, as shown in the following listing.

Listing 1.9 The show action of PurchasesController

```
def show
end
```

Or is it? It turns out that it's not actually defined here. There's a `before_action` defined:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.10 PurchasesController

```
class PurchasesController
  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
```

This code will get executed before every action given, 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 with `params[:id]` as the number on the end of the URL.

Going back to the view (Listing 1.8 app/views/purchases/show.html.erb) now, and at the end of this file is `link_to`, which generates a link using the first argument as the text for it and the second argument as the `href` for that URL. The second argument for `link_to` is a method itself: `edit_purchase_path`. This method is provided by a method call in config/routes.rb, which we now look at.

1.2.9 Routing

The config/routes.rb file of every Rails application is where the application routes are defined in a succinct Ruby syntax. The methods used in this file define the pathways from requests to controllers. If you look in your config/routes.rb while ignoring the commented-out lines for now, you'll see what's shown in the following listing.

Listing 1.11 config/routes.rb

```
ThingsIBought::Application.routes.draw do
  resources :purchases
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Inside the block for the `draw` method is the `resources` method. Collections of similar objects in Rails are referred to as *resources*. This method defines the routes and routing helpers (such as the `edit_purchase_path` method) to your purchases resources. Look at table 1.1 for a list of the helpers and their corresponding routes.

Table 1.1 Table 1.1 Routing helpers and their routes

Helper	Route
`purchases_path`	`/purchases`
`new_purchase_path`	`/purchases/new`
`edit_purchase_path`	`/purchases/:id/edit`
`purchase_path`	`/purchases/:id`

In this table, `:id` can be substituted for the ID of a record. Each routing helper has an alternative version that will give you the full URL to the resource. Use the `_url` extension rather than `_path` and you'll get a URL such as http://localhost:3000/purchases for `purchases_url`.

From this table, two of these routes will act differently depending on how they're requested. The first route, `/purchases`, takes you to the `index` action of `PurchasesController` if you do a GET request. GET requests are the standard type of requests for web browsers, and this is the first request you did to this application. If you do a POST request to this route, it will go to the `create` action of the controller. This is the case when you submit the form from the `new` view.

Let's go to http://localhost:3000/purchases/new now and look at the source of the page. You should see the beginning tag for your form looking like the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.12 The HTML source of app/views/purchases/new.html.erb

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

The two attributes to note here are the `action` and `method` attributes. The `action` dictates the route to where this form goes, and the `method` tells the form what kind of HTTP request to make.

How'd this tag get rendered in the first place? Well, as you saw before, the app/views/purchases/new.html.erb template uses the form partial from app/views/purchases/_form.html.erb, which contains this as the first line:

```
<%= form_for(@purchase) do |f| %>
```

This one simple line generates that form tag. When we look at the `edit` action shortly, you'll see that the output of this tag is different, and you'll see why.

The other route that responds differently is the `/purchases/{id}` route, which acts in one of three ways. You already saw the first way: it's the `show` action to which you're redirected (a GET request) after you create a purchase. The second of the three ways is when you update a record, which we look at now.

1.2.10 Updating

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. You should now see a page that looks similar to the `new` page, shown in Figure 1.8

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

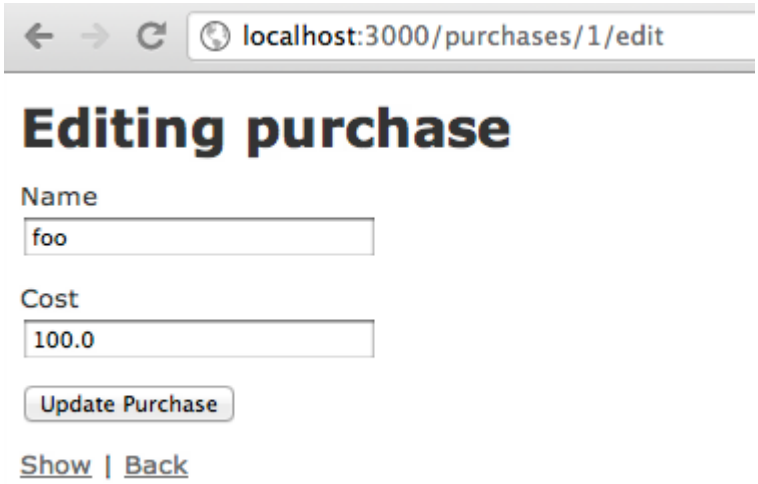

Figure 1.8 Editing a purchase

This page looks similar because it re-uses 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: you can use the same code for two different requests to your application. The template for this action can be seen in the following listing.

Listing 1.13 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 pre-existing object is found by the `edit` action in `PurchasesController`, shown in the next listing.

Listing 1.14 The edit action of PurchasesController

```
def edit
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Oops, it's not here! The code to find the `@purchase` object here is identical to what you saw earlier in the `show` action: it's set in the `before_action`.

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. Using this syntax, it will attempt to call the `purchase_path` method because the object has a class of `Purchase` and will pass the object along to that call, generating the URL.[14]

Footnote 14 This syntax is exceptionally handy if you have an object and are not 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`.

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 now, for example, by changing the cost from 100 to 10 and pressing Update Purchase. You now see the `show` page but with a different message, shown in Figure 1.9

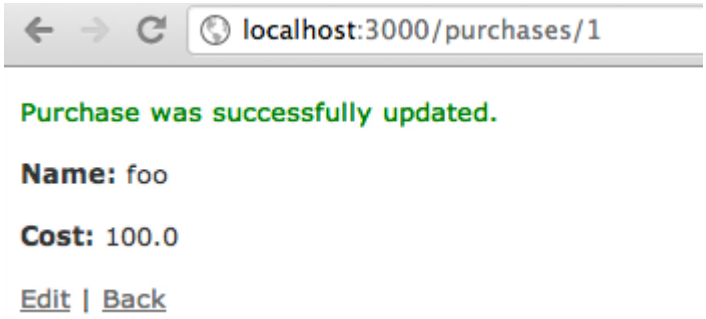

Figure 1.9 Viewing an updated purchase

Pressing Update Purchase brought you back to the `show` page. How did that happen? Press 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.15 The rendered HTML for app/views/purchases/edit.html.erb

```
    <form accept-charset="UTF-8"
          action="/purchases/2"
          class="edit_purchase"
          id="edit_purchase_2"
          method="post">
  <div style="margin:0;padding:0;display:inline">
    <input name="_method" type="hidden" value="patch" />
    <co id="ch01_613_1"/>
  </div>
...
```

This `form`'s `action` 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 the `/purchases/{id}` responds according to the method. By making a PATCH request to this route, you're taken to the `update` action in `PurchasesController`. Let's take a look at this in the following listing.

Footnote 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 dows not allow the `PATCH` method for form elements.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 1.16 The update action of PurchasesController

```
def update
  respond_to do |format|
    if @purchase.update(purchase_params)                    ❶

      format.html { redirect_to(@purchase,
                    notice: 'Purchase was successfully updated.') }
      format.json  { head :no_content }
    else
      format.html { render action: "edit" }
      format.json  { render json: @purchase.errors,
                            status: :unprocessable_entity }
    end
  end
end
```

Just as in the `show` and `edit` actions, you fetch the object first by using the `find` method. 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_attributes`❶ on the `@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_attributes` returns `true`, you're redirected back to the `show` action for this particular purchase by using `redirect_to`.

If the `update_attributes` 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 and setting the name to blank and then pressing Update Purchase. It should error exactly like the `create` method did, as shown in Figure 1.10

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

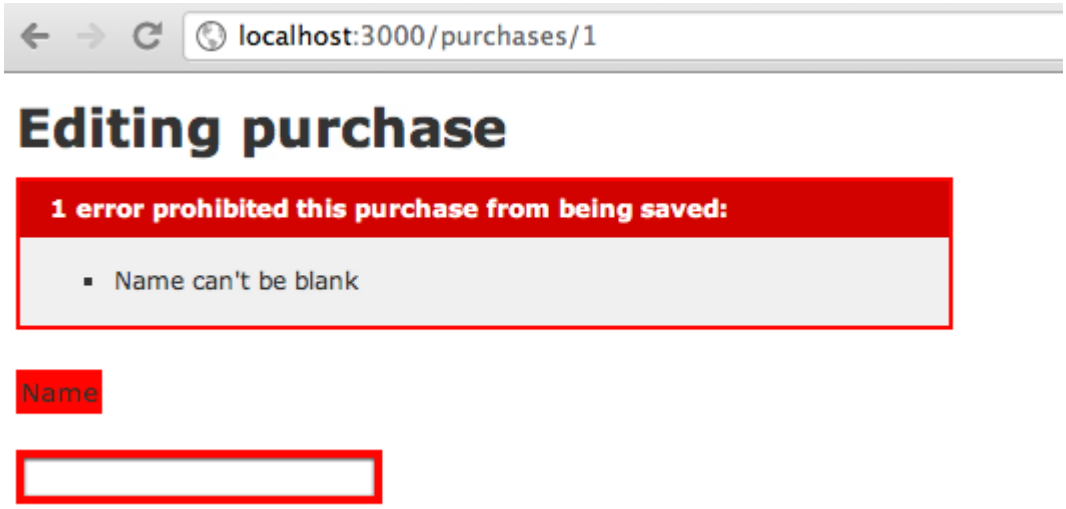

Figure 1.10 Update fails!

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

Now what would happen if, rather than update 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.

Footnote 16 Mac OS X dictionary

You can destroy a record by going to http://localhost:3000/purchases and clicking the Destroy link shown in Figure 1.11 and then clicking OK on the confirmation box that pops up.

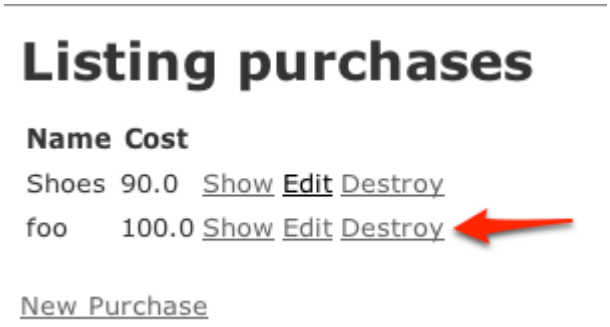

Figure 1.11 Destroy!

When that record's destroyed, you're taken back to the Listing Purchases page.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You'll see that the record no longer exists. You should now only have one record, as shown in Figure 1.12

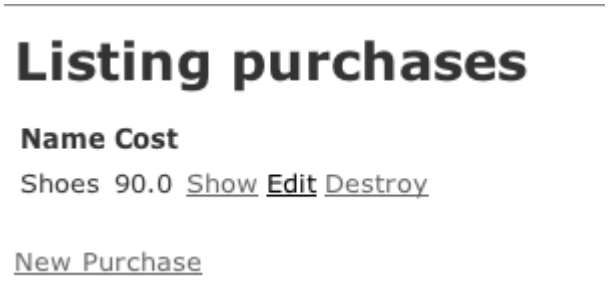

Figure 1.12 Last record standing

How does all of 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 1.17 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 %>
</table>

<br />

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

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`/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 the path is `/purchases/{id}/edit`.

The third `link_to` links seemingly to the `purchase` object exactly as the first, but it doesn't go there. The `:method` option on the end of this route specifies the method of `: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 and takes you to the `destroy` action in the `PurchasesController`. This action is shown in the following listing.

Listing 1.18 The destroy action of PurchasesController

```
def destroy
  @purchase.destroy                                          ❶
  respond_to do |format|
    format.html { redirect_to(purchases_url) }
    format.json  { head :no_content }
  end
end
```

Just as in the `show`, `edit`, and `update` actions shown earlier, this action finds the `@purchase` object by using `Purchase.find` and then destroys the record by calling `destroy`❶ on it, which permanently deletes the record. Then it uses `redirect_to` to take you to the `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, such as http://localhost:3000/purchases/1.

That wraps up our application run-through!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

1.3 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 get the bones of a Rails application going. From this basic skeleton, you can keep adding on 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 Groupon and GitHub—use Ruby on Rails. This clearly answers the question *Is Rails ready?* Yes, it very much is. A wide range of companies have built successful websites on the Rails framework, and a lot more will do so in the future.

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 all 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, you can explore the IRC channel on Freenode *#rubyonrails*, the mailing list *rubyonrails-talk* on Google Groups, 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 them. 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 it's continuously working with the most minimal effort possible, right?

Well, Ruby on Rails does that too. There are several testing frameworks for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Ruby and Ruby on Rails, and in chapter 2 we look at the two major ones: Minitest and RSpec.

Index Terms

bin/rails generate
bin/rails server
config/routes.rb
destroy, ActiveRecord::Base
errors, ActiveRecord::Base
find, ActiveRecord::Base
form_for
form_partial
form tag
link_to
Migrations
MVC
new, ActiveRecord::Base
notice
params
rails new command
redirect_to
redirect_to, :notice option
render, partial
REST
Routing, resources method
Routing helpers
save, ActiveRecord::Base
scaffold generator
update_attributes, ActiveRecord::Base
validates, :presence option
validates, :presence option

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

2 *Testing saves your bacon*

Chapter 1 presented an extremely basic layout of a Rails application and an example of the scaffold generator.[1] One question remains, though: how do you make your Rails applications maintainable?

Footnote 1 We won't use the scaffold generator for the rest of the book because people tend to use it as a crutch, and it generates extraneous code. There's a thread on the rubyonrails-core mailing list where people have discussed the scaffold generator's downsides: https://groups.google.com/forum/?fromgroups#!topic/rubyonrails-core/lkEqGjY_vcU

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, 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) do not. We're going to be doing it right from step one.[2]

Footnote 2 Unlike certain other books.

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 two testing tools -- Minitest and RSpec -- in a basic fashion so you can quickly learn their format.

By learning good testing techniques now, you've got 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.

A cryptic yet true answer to the question *Why should I test?* is "because you are

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

human.” Humans—the large majority of this book’s audience—make mistakes. It’s one of our favorite ways to learn. Because humans make mistakes, having a tool to inform them when they 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. We want you to save your bacon!

TDD and BDD also give you time to think through your decisions before you write any code. By first writing the test for the implementation, you are (or, at least, you should be) thinking through the implementation: the code you’ll write *after* the test and how you’ll make the test passes. If you find the test difficult to write, then perhaps the implementation could be improved. Unfortunately, there’s no clear way to quantify the difficulty of writing a test and working through it other than to consult with other people who are familiar with the process.

Once the test is implemented, you should go about writing some code that your test can pass. If you find yourself working backwards—rewriting your test to fit a buggy implementation—it’s generally best to rethink the test and scrap the implementation. Test first, code later.

TDD is a methodology consisting of writing a failing test case first (usually using a testing tool such as *Minitest*), then writing the code to make the test pass, and finally refactoring the code. This process is commonly called *red-green-refactor*. The reasons for developing code this way are twofold. First, it makes you consider how the code should be running before it is used by anybody. Second, it gives you an automated test you can run as often as you like to ensure your code is still working as you intended. We'll be using the Minitest tool for TDD.

BDD is a methodology based on TDD. You write an automated test to check the interaction between the different parts of the codebase rather than testing that each part works independently.

Two tools used for BDD when building Rails applications are *RSpec* and *Cucumber*, with this book heavily relying on RSpec and foregoing Cucumber[3].

Footnote 3 Cucumber was previously used in earlier editions of this book, but the community has drifted away from using it, as there are other tools (like Capybara, mentioned later) that provide a very similar way to test, but in a much neater syntax.

Let’s begin by looking at TDD and Minitest.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

2.1 Test-driven development basics

Automated testing is much, much easier than manual testing. Have you ever gone through a website and manually filled in a form with specific values to make sure it conforms to your expectations? Wouldn't it be faster and easier to have the computer do this work? Yes, it would, and that's the beauty of automated testing: you won't spend your time manually testing your code because you'll have written test code to do that for you.

On the off chance you break something, the tests are there to tell you the what, when, how, and why of the breakage. Although tests can never be 100% guaranteed, your chances of getting this information without first having written tests are 0%. Nothing is worse than finding out something is broken through an early-morning phone call from an angry customer. Tests work toward preventing such scenarios by giving you and your client peace of mind. If the tests aren't broken, chances are high (though not guaranteed) that the implementation isn't either.

You'll likely at some point face a situation in which something in your application breaks when a user attempts to perform an action you didn't consider in your tests. With a base of tests, you can easily duplicate the scenario in which the user encountered the breakage, generate your own failed test, and use this information to fix the bug. This commonly used practice is called *regression testing*.

It's valuable to have a solid base of tests in the application so you can spend time developing new features *properly* rather than fixing the old ones you didn't do quite right. An application without tests is most likely broken in one way or another.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

2.1.1 Writing your first test

The first testing library for Ruby was Test::Unit, which was written by Nathaniel Talbott back in 2000 and is now part of the Ruby core library. The documentation for this library gives a fantastic overview of its purpose, as summarized by the man himself:

> The general idea behind unit testing is that you write a *test method* that makes certain assertions about your code, working against a *test fixture*. A bunch of these *test methods* are bundled up into a *test suite* and can be run any time the developer wants. The results of a run are gathered in a *test result* and displayed to the user through some UI.
>
> -- —Nathaniel Talbott

The UI Talbott references could be a terminal, a web page, or even a light.[4]

Footnote 4 Such as the one GitHub has made: http://github.com/blog/653-our-new-build-status-indicator

In Rails 4, Test::Unit has been superseded by Minitest, which is a library of a similar style, but a more modern heritage. Minitest is part of the Ruby standard library.

A common practice you'll hopefully by now have experienced in the Ruby world is to let the libraries do a lot of the hard work for you. Sure, you *could* write a file yourself that loads one of your other files and runs a method and makes sure it works, but why do that when Minitest already provides that functionality for such little cost? Never re-invent the wheel when somebody's done it for you.

Now you're going to write a test, and you'll write the code for it later. Welcome to TDD.

To try out Minitest, first create a new directory called example and in that directory make a file called example_test.rb. It's good practice to suffix your filenames with `_test` so it's obvious from the filename that it's a test file. In this file, you're going to define the most basic test possible, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.1 example/example_test.rb

```
require 'minitest/autorun'

class ExampleTest < Minitest::Test
  def test_truth
    assert true
  end
end
```

To make this a Minitest test, you begin by requiring `minitest/autorun`, which is part of Ruby's standard library. This provides the `Minitest::Test` class inherited from on the next line. Inheriting from this class provides the functionality to run any method defined in this class whose name begins with `test`.

To run this file, you run `ruby example_test.rb` in the terminal. When this command completes, you see some output, the most relevant being the last three lines:

```
.

Finished tests in 0.000618s, 1618.1230 tests/s, 1618.1230 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

The first line is a singular period. This is Minitest's way of indicating that it ran a test and the test passed. If the test had failed, it would show up as an `F`; if it had errored, an `E`. The second and third lines provide statistics on what happened, specifically that there was one test and one assertion, and that nothing failed, there were no errors, and nothing was skipped. Great success!

The `assert` method in your test makes an assertion that the argument passed to it evaluates to `true`. This test passes given anything that's not `nil` or `false`. When this method fails, it fails the test and raises an exception. Go ahead, try putting `1` there instead of `true`. It still works:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Run options:

# Running tests:

.

Finished tests in 0.001071s, 933.7068 tests/s, 933.7068 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

In the following listing, you remove the `test_` from the beginning of your method and define it as simply a `truth` method:

Listing 2.2 example/example_test.rb, alternate truth test

```
def truth
  assert true
end
```

Minitest tells you there were no tests specified:

```
0 tests, 0 assertions, 0 failures, 0 errors, 0 skips
```

See no tests! Remember to always prefix Minitest methods with `test`!

2.1.2 Saving bacon

Let’s make this a little more complex by creating a bacon_test.rb file and writing the test shown in the following listing.

Listing 2.3 example/bacon_test.rb

```
require 'minitest/autorun'

class BaconTest < Minitest::Test
  def test_saved
    assert Bacon.saved?
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Of course, you want to ensure that your bacon[5] is always saved, and this is how you do it. If you now run the command to run this file, `ruby bacon_test.rb`, you get an error:

Footnote 5 Both the metaphorical and the crispy kinds.

```
NameError: uninitialized constant BaconTest::Bacon
```

Your test is looking for a constant called `Bacon` and cannot find it because you haven't yet defined the constant. For this test, the constant you want to define is a `Bacon` class. You can define this new class before or after the test. Note that in Ruby you usually must define constants and variables before you use them. In Minitest tests, the code is only run when it finishes evaluating it, which means you can define the `Bacon` class after the test. In the next listing, you follow the more conventional method of defining the class above the test.

Listing 2.4 example/bacon_test.rb, now with Bacon class

```
require 'minitest/autorun'

class Bacon
end

class BaconTest < Minitest::Test
  def test_saved
    assert Bacon.saved?
  end
end
```

Upon rerunning the test, you get a different error:

```
NoMethodError: undefined method `saved?' for Bacon:Class
```

Progress! It recognizes there's now a `Bacon` class, but there's no `saved?` method for this class, so you must define one, as in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.5 Bacon class inside example/bacon_test.rb

```
class Bacon
  def self.saved?
    true
  end
end
```

One more run of `ruby bacon_test.rb` and you can see that the test is now passing:

```
.

Finished tests in 0.000596s, 1677.8523 tests/s, 1677.8523 assertions/s.

1 tests, 1 assertions, 0 failures, 0 errors, 0 skips
```

Your bacon is indeed saved! Now any time that you want to check if it's saved, you can run this file. If somebody else comes along and changes that `true` value to a `false`, then the test will fail:

```
F

Finished tests in 0.000757s, 1321.0040 tests/s, 1321.0040 assertions/s.

  1) Failure:
test_saved(BaconTest) [test.rb:11]:
Failed assertion, no message given.
```

Minitest reports "Failed assertion, no message given" when an assertion fails. You should probably make that error message clearer! To do so, you can specify an additional argument to the `assert` method in your test, like this:

```
assert Bacon.saved?, "Our bacon was not saved :("
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Now when you run the test, you get a clearer error message:

```
  1) Failure:
test_saved(BaconTest) [bacon_test.rb:11]:
Our bacon was not saved :(
```

And that, my friend, is the basics of TDD using Minitest. While we don't use this method in the book, it's handy to know about because it establishes the basis for TDD in Ruby in case you wish to use it in the future. Minitest is also the default testing framework for Rails, so you may see it around in your travels. From this point on, we focus on just pure RSpec which you'll be using to develop your next Rails application.

2.2 Behavior-driven development basics

BDD is similar to TDD, but the tests for BDD are written in an easier-to-understand language so that developers and clients alike can clearly understand what is being tested. The tool that you'll be using for all BDD examples in this book is RSpec.

RSpec tests are written in a Ruby domain-specific language (DSL), like this:

```
describe Bacon do
  it "is edible" do
    expect(Bacon.edible?).to be_true
  end
end
```

The benefits of writing tests like this are that clients can understand precisely what the test is testing and then use these steps in acceptance testing;[6] a developer can read what the feature should do and then implement it; and finally, the test can be run as an automated test. With tests written in DSL, you have the three important elements of your business (the clients, the developers, and the code) all operating in the same language.

Footnote 6 A process whereby people follow a set of instructions to ensure a feature is performing as intended.

RSpec is an extension of the methods already provided by Minitest. You can even use Minitest methods inside of RSpec tests if you wish. But we're going to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

use the simpler, easier-to-understand syntax that RSpec provides.

2.2.1 Introducing RSpec

RSpec is a BDD tool written by Steven R. Baker and now maintained by Myron Marston and Andy Lindeman as a cleaner alternative to Minitest. With RSpec, you write code known as *specs* that contain *examples*, which are synonymous to the *tests* you know from Minitest. In this example, you're going to define the `Bacon` constant and then define the `edible?` method on it.

Let's jump right in and install the *rspec* gem by running `gem install rspec`. You should see something like the following output:

```
Successfully installed rspec-core-2.14.5
Successfully installed diff-lcs-1.2.4
Successfully installed rspec-expectations-2.14.2
Successfully installed rspec-mocks-2.14.3
Successfully installed rspec-2.14.1
```

You can see that the final line says the rspec gem is installed, with the version number specified after the name.

2.2.2 Writing your first spec

When the gem is installed, create a new directory for your tests called bacon anywhere you like, and inside that, create another directory called spec. If you're running a UNIX-based operating system such as Linux or Mac OS X, you can run the `mkdir -p bacon/spec` command to create these two directories. This command will generate a bacon directory if it doesn't already exist, and then generate in that directory a spec directory.

Inside the spec directory, create a file called bacon_spec.rb. This is the file you use to test your currently nonexistent `Bacon` class. Put the code from the following listing in spec/bacon_spec.rb.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.6 bacon/spec/bacon_spec.rb

```
describe Bacon do
  it "is edible" do
    expect(Bacon.edible?).to be_true
  end
end
```

You `describe` the (undefined) `Bacon` class and write an example for it, declaring that `Bacon` is edible. The `describe` block contains tests (examples) that describe the behavior of bacon. In this example, whenever you call `edible?` on `Bacon`, the result should be `true`. `expect` and `to` serve a similar purpose to `assert`, which is to assert that the object passed to `expect` matches the arguments passed to `to`. If the outcome is not what you say it should be, then RSpec raises an error and goes no further with that spec.

To run the spec, you run `rspec spec` in a terminal in the root of your bacon directory. You specify the spec directory as the first argument of this command so RSpec will run all the tests within that directory. This command can also take files as its arguments if you want to run tests only from those files.

When you run this spec, you get an `uninitialized constant Object::Bacon` error, because you haven't yet defined your `Bacon` constant. To define it, create another directory inside your Bacon project folder called lib, and inside this directory, create a file called bacon.rb. This is the file where you define the `Bacon` constant, a class, as in the following listing.

Listing 2.7 bacon/lib/bacon.rb

```
class Bacon
end
```

You can now require this file in spec/bacon_spec.rb by placing the following line at the top of the file:

```
require 'bacon'
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you run your spec again, because you told it to load bacon, RSpec has added the lib directory on the same level as the spec directory to Ruby's load path, and so it will find the lib/bacon.rb for your `require`. By requiring the lib/bacon.rb file, you ensure the `Bacon` constant is defined. The next time you run it, you get an undefined method for your new constant:

```
1) Bacon is edible
Failure/Error: expect(Bacon.edible?).to be_true
NoMethodError:
  undefined method `edible' for Bacon:Class
# ./spec/bacon_spec.rb:5:in `block (2 levels) in <top (required)>'
```

This means you need to define the `edible?` method on your `Bacon` class. Re-open lib/bacon.rb and add this method definition to the class:

```
def self.edible?
  true
end
```

Now the entire file looks like the following listing.

Listing 2.8 bacon/lib/bacon.rb

```
class Bacon
  def self.edible?
    true
  end
end
```

By defining the method as `self.edible?`, you define it for the class. If you didn't prefix the method with `self.`, it would define the method for an instance of the class rather than for the class itself. Running `rspec spec` now outputs a period, which indicates the test has passed. That's the first test—done.

For the next test, you want to create many instances of the `Bacon` class and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

have the `edible?` method defined on them. To do this, open lib/bacon.rb and change the `edible?` class method to an instance method by removing the `self.` from before the method, as in the next listing.

Listing 2.9 bacon/lib/bacon.rb

```
class Bacon
  def edible?
    true
  end
end
```

When you run `rspec spec` again, you get the familiar error:

```
1) Bacon is edible
   Failure/Error: expect(Bacon.edible?).to be_true
   NoMethodError:
     undefined method `edible?' for Bacon:Class
   # ./spec/bacon_spec.rb:5:in `block (2 levels) in <top (required)>'
```

Oops! You broke a test! You should be changing the spec to suit your new ideas before changing the code! Let's reverse the changes made in lib/bacon.rb, as in the following listing.

Listing 2.10 bacon/lib/bacon.rb

```
class Bacon
  def self.edible?
    true
  end
end
```

When you run `rspec spec`, it passes. Now let's change the spec first, as in the next listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.11 bacon/spec/bacon_spec.rb

```
describe Bacon do
  it "is edible" do
    expect(Bacon.new.edible?).to be_true
  end
end
```

In this code, you instantiate a new object of the class rather than using the `Bacon` class. When you run `rspec spec`, it breaks once again:

```
NoMethodError:
  undefined method `edible?' for #<Bacon:0x101deff38>
```

If you remove the `self.` from the `edible?` method, your test will now pass:

```
.

Finished in 0.00167 seconds
1 example, 0 failures
```

Now you can go about breaking your test once more by adding additional functionality: an `expired!` method, which will make your bacon inedible. This method sets an instance variable on the `Bacon` object called `@expired` to `true`, and you use it in your `edible?` method to check the bacon's status.

First you must test that this `expired!` method is going to actually do what you think it should do. Create another example in spec/bacon_spec.rb so that the whole file now looks like the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.12 bacon/spec/bacon_spec.rb

```
require 'bacon'

describe Bacon do
  it "is edible" do
    expect(Bacon.new.edible?).to be_true
  end

  it "expired!" do
    bacon = Bacon.new
    bacon.expired!
    expect(bacon).to_not be_edible
  end
end
```

If you run `rspec` again, your first spec still passes, but your second one fails because you have yet to define your `expired!` method. Let's do that now in lib/bacon.rb, as shown in the following listing.

Listing 2.13 bacon/lib/bacon.rb

```
class Bacon
  def edible?
    true
  end

  def expired!
    self.expired = true
  end
end
```

By running `rspec spec` again, you get an `undefined method` error:

```
NoMethodError:
  undefined method `expired=' for #<Bacon:0x101de6578>
```

This method is called by the following line in the previous example:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
self.expired = true
```

To define this method, you can use the `attr_accessor` method provided by Ruby, as shown in Listing 2.16; the `attr` prefix of the method means attribute. If you pass a `Symbol` (or collection of symbols) to this method, it defines methods for setting (`expired=`) and retrieving the attribute `expired` values, referred to as a *setter* and a *getter* respectively. It also defines an instance variable called `@expired` on every object of this class to store the value that was specified by the `expired=` method calls.

WARNING **`self.` prefix**
In Ruby you can call methods without the `self.` prefix. You specify the prefix because otherwise the interpreter will think that you're defining a *local variable*. The rule for setter methods is that you should always use the prefix.

Listing 2.14 attr_accessor for Bacon in bacon/lib/bacon.rb

```
class Bacon
  attr_accessor :expired
  ...
end
```

With this in place, if you run `rspec spec` again, your example fails on the line following your previous failure:

```
1) Bacon expired!
Failure/Error: expect(bacon).to_not be_edible
  expected edible? to return false, got true
# ./spec/bacon_spec.rb:11:in `block (2 levels) in <top (required)>'
```

Even though this sets the `expired` attribute on the `Bacon` object, you've still hardcoded true in your `edible?` method. Now change the method to use the attribute method, as in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 2.15 Bacon#edible? method

```
def edible?
  !expired
end
```

When you run `rspec spec` again, both your specs will pass:

```
..

Finished in 0.00191 seconds
2 examples, 0 failures
```

Let's go back in to lib/bacon.rb and remove the `self.` from the `expired!` method:

```
def expired!
  expired = true
end
```

If you run `rspec spec` again, you'll see your second spec is now broken:

```
1) Bacon expired!
Failure/Error: expect(bacon).to_not be_edible
  expected edible? to return false, got true
# ./spec/bacon_spec.rb:11:in `block (2 levels) in <top (required)>'
```

Tests save you from making mistakes such as this. If you write the test first and then write the code to make the test pass, you have a solid base and can refactor the code to be clearer or smaller and finally ensure that it's still working with the test you wrote in the first place. If the test still passes, then you're probably doing it right.

If you change this method back now:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def expired!
  self.expired = true
end
```

and then run your specs using `rspec`, you'll see that they once again pass:

```
..

2 examples, 0 failures
```

Everything's normal and working once again, which is great!

That ends our little foray into RSpec for now. You'll use it again later when you develop your application. If you'd like to know more about RSpec, *The RSpec Book: Behavior-Driven Development with RSpec, Cucumber, and Friends* (David Chelimsky et al., Pragmatic Bookshelf, 2010) is recommended reading.

2.3 Summary

This chapter demonstrated how to apply TDD and BDD principles to test some rudimentary code. You can (and should!) apply these principles to all code you write, because testing the code ensures it's maintainable from now into the future. You don't have to use the gems shown in this chapter to test your Rails application; they are just preferred by a large portion of the community.

You'll apply what you learned in this chapter to building a Rails application from scratch in upcoming chapters. You'll use RSpec and another tool called Capybara to build out acceptance tests that will describe the behavior of your application. Then you will go about implementing the behaviour of the applicaton to make these tests pass, and you'll know that you're doing it right when the tests are all green.

Let's get into it!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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, then refining the feature until it's viewed as complete before moving on to the next one.[1]

Footnote 1 More information about Agile can be found on Wikipedia: http://en.wikipedia.org/wiki/Agile_software_development.

For this example application, your imaginary client, who has limitless time and budget (unlike those 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: 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, then you can move on to the next prioritized chunk of work.

The first couple of features you develop for this application will be laying down the foundation for the application, enabling people to create projects and tickets. Later on, in Chapters 6 and 7, you implement authentication and authorization so that people can sign in to the application and only have access to certain projects. Other chapters cover things like adding comments to tickets and notifying users by email and file uploading.

BDD is used all the way through the development process. It provides the client

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

with a stable application, and when (not if) a bug crops up, you have a nice test base you can use to determine what is broken. Then you can fix the bug so it doesn't happen again, a process called *regression testing* (mentioned in chapter 2).

As you work with your client to build the features of the application using this behaviour-driven development technique, the client may ask why all of 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: Tests will make your code more maintainable; however it will not make your code bug-proof.

The tests also give you a clearer picture of what your client *really* wants. By having it all written down in code, you have a solid reference to point to if clients say they suggested something different. Story-driven development is simply BDD with emphasis on things a user can actually 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. However, rather than using scaffolding, as you did previously, you write this application from the ground up using the BDD (behavior-driven development) 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

3.1.1 The application story

Your client may have a good idea of the application he or she wants 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 we'll use RSpec and Capybara to develop these stories.

Start with the most basic story and ask your client how he or she wants 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:

```
describe "log in" do
  it "as a user" 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.

You may find it helpful to put these stories into a system such as Pivotal Tracker (https://pivotaltracker.com) so you can keep track of them. Pivotal Tracker allows you to assign points of difficulty to a story and then, over a period of weeks, estimate which stories can be accomplished in the next iteration on the basis of how many were completed in previous weeks. This tool is exceptionally handy to use when working with clients because the client can enter stories and then follow the workflow process. In this book, we don't use Pivotal Tracker because we aren't working with a real client, but this method is highly recommended.

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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[2]. To generate this application, run this command:

Footnote 2 Hey, at least *we* thought it was funny!

```
$ rails new ticketee
```

TIP **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:

- 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

You can't do all of this with 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 to the commit. Additionally, you can create branches for experimental features and work on those independently of the main code base 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.[3]

Footnote 3 http://progit.org/book/.

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 plugin 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 do not have Git already installed, GitHub's help site offers installation guides for Mac,[4] Linux,[5] and Windows.[6] The precompiled installer should work well for Macs, and the package distributed versions (APT, eMerge, etc.) work well for Linux machines. For Windows, the *msysGit* application does just fine.

Footnote 4 http://help.github.com/mac-set-up-git/. Note this lists four separate ways, not four separate steps, to install Git.

Footnote 5 http://help.github.com/linux-set-up-git/.

Footnote 6 http://help.github.com/windows-set-up-git/.

For an online place to put your Git repository, we recommend GitHub,[7] which offers free accounts. If you set up an account now, you can upload your code to GitHub as you progress, ensuring that you don't lose it if anything were to happen to you 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. [8] Once generate the key, copy the public key's content (usually found at ~/.ssh/id_rsa.pub) into the SSH Public Key field on the Signup page or, if you've already signed up, click the Account Settings link (Figure 3.1) in the menu at the top, select SSH Public Keys, and then click Add Another Public Key to enter it there (Figure 3.2).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 7 http://github.com.

Footnote 8 A guide for this process can be found at http://help.github.com/linux-key-setup/.

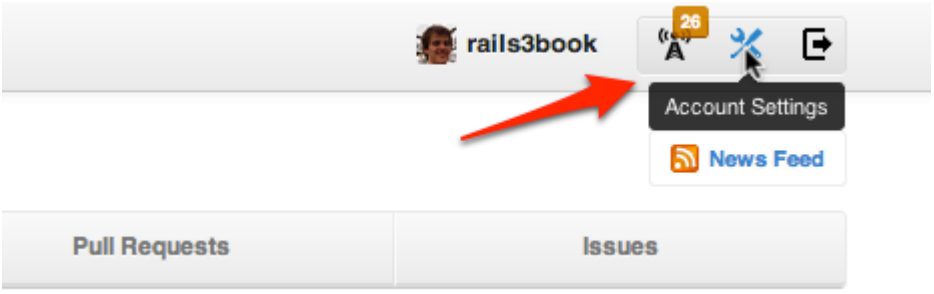

Figure 3.1 Click account settings

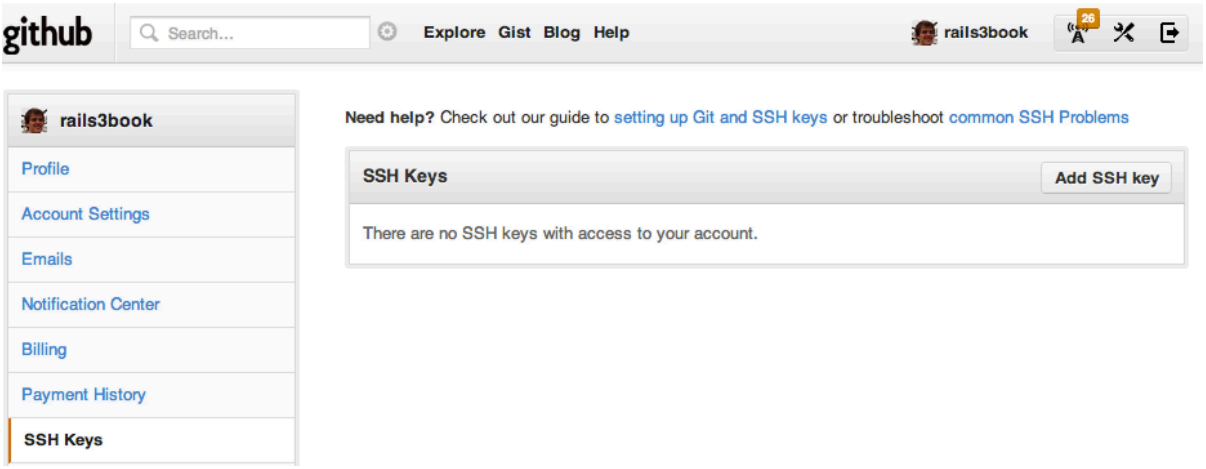

Figure 3.2 Add an SSH key

Now that you're set up with GitHub, click the New Repository button on the dashboard to begin creating a new repository (Figure 3.3).

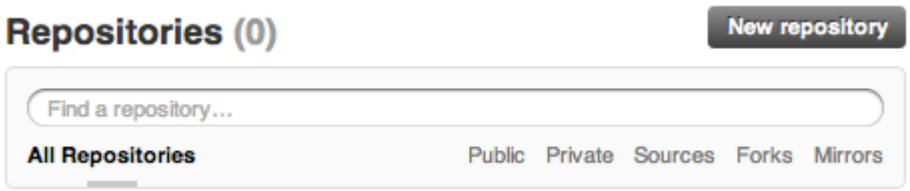

Figure 3.3 Create a new repository

On this page, enter the Project Name as *ticketee* and click the Create Repository button to create the repository on GitHub. Now you are on your project's page. Follow the instructions, especially concerning the configuration of your identity. In Listing 3.1, replace `"Your Name"` with your real name and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`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. The `git` commands should be typed into your terminal or command prompt.

Listing 3.1 Configuring your identity in GitHub

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

You already have a ticketee directory, and you're probably inside it. If not, you should be. To make this directory a git repository, run this easy command:

```
$ git init
```

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 as a checkpoint of 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 4 application"
```

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

```
Created initial commit cdae568: Generate the Rails 4 application
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
35 files changed, 9280 insertions(+), 0 deletions(-)
```

The `cdae568` 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 (additional line count), 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 an output similar to the following listing.

Listing 3.2 Viewing the commit log

```
commit cdae568599251137d1ee014c84c781917b2179e1
Author: Your Name <you@example.com>
Date:   [date stamp]

  Generate the Rails 4 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. [9] With that commit in your repository, you have something to push to GitHub, which you can do by running:

Footnote 9 The chances of this happening are 1 in 268,435,456.

```
$ git remote add origin git@github.com:yourname/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 github username]/ticketee.git` path, which connects to the repository using SSH.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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.

Listing 3.3 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:rails3book/ticketee.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.

As we go through the book, we will also `git push` just like you. You can compare your code to ours, by checking out our repository on GitHub: https://github.com/steveklabnik/ticketee

To roll back the code to a given point in time, check out `git log`:

```
commit e407ef5193de50b09f987f2cf4de0afd2be644b3
Author: Steve Klabnik <steve@steveklabnik.com>
Date:   Mon Sep 9 19:30:50 2013 -0700

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

commit d3e25028fbaeae6b79a68c9c08224c1692663f31
Author: Steve Klabnik <steve@steveklabnik.com>
Date:   Mon Sep 9 18:51:37 2013 -0700

    Only users with the 'change states'
                   permission can change states

commit 23729a46e6031ca2fabaf3c58b52c9bcbd9ccc78
Author: Steve Klabnik <steve@steveklabnik.com>
Date:   Mon Sep 9 18:40:50 2013 -0700

    Admins can now set a default state for tickets
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Each one 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/steveklabnik/ticketee/commits

Once you've found the commit with the right message, make note of the big long series of numbers and letters next to it: that's the commit's hash. Use this hash 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 promotes the *convention over configuration* line passionately, there's still some parts of the application that 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. The Bundler gem is responsible for everything to do with this file; it's the Bundler's job to ensure that all the gems are installed when your application is initialized. Let's look at the following listing to see how this looks inside (commented lines are removed for simplicity).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.4 Gemfile

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

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.0.0'

# Use sqlite3 as the database for Active Record
gem 'sqlite3'

# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.0'

# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'

# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'

# See https://github.com/sstephenson/execjs#readme for more
# supported runtimes
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'

# Turbolinks makes following links in your web application
# faster. Read more: https://github.com/rails/turbolinks
gem 'turbolinks'

# Build JSON APIs with ease.
# Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 1.2'

group :doc do
  # bundle exec rake doc:rails generates the API under doc/api.
  gem 'sdoc', require: false
end

# Use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

# Use unicorn as the app server
# gem 'unicorn'

# Use Capistrano for deployment
# gem 'capistrano', group: :development

# Use debugger
# gem 'debugger', group: [:development, :test]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 `4.0.0` of the `rails` gem. Bundler inspects the dependencies of the requested gem as well as all gem dependencies of those dependencies (and so on), then does what it needs to do to make them 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.

Chapter 2 focused on BDD and, as was more than hinted at, you'll be using it to develop this application. First, alter the Gemfile to ensure you have the correct gem for RSpec for your application. Add the lines from the following listing to the bottom of the file:

Listing 3.5 Gemfile - Adding rspec and capybara gems

```
group :test, :development do
  gem 'rspec-rails', "~> 2.14"
end

group :test do
  gem 'capybara', "2.1.0"
end
```

In the Gemfile, you specify that you wish to use the latest 2.x release of RSpec in the `test` and `development` groups. You put this gem inside the `development` group because without it, the tasks you can use to run your specs will be unavailable. 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.

With `rspec-rails`, you specified a version number with `~> 2.14`[10], which tells RubyGems you want RSpec 2.14 *or higher*, but less than RSpec 3.0. This means when RSpec releases 2.14.1 or 2.15 and you go to install your gems, RubyGems will install the latest version it can find rather than only 2.14.

Footnote 10 The ~> operator is called the *approximate version constraint*. You can read more about it on the RubyGems documentation page here: http://docs.rubygems.org/read/chapter/16#page74

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

A new gem is used in Listing 3.5: *Capybara*. Capybara is a browser simulator in Ruby that is used for *integration testing*, which you'll be doing in just a short while. This kind of testing ensures that when a link is clicked in your application, it goes to the correct page, or when you fill in a form and click the Submit button, an onscreen message tells you that the form's operation was successful.

Capybara also supports real browser testing by launching an instance of Firefox. You can then test your application's JavaScript, which you'll use extensively in chapter 9.

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*.

Development is used for your local application, such as when you're playing with it in the browser. In development mode, page and class caching is 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. We're not there yet.

Test 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.

Production is used when you finally deploy your application. This mode is designed for speed, and any changes you make to your application's classes are not effective until the server is restarted.

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

```
Bundler.require(:default, Rails.env)
```

To install these gems to your system, run `bundle update` at the root of your application. This command tells Bundler to ignore your `Gemfile.lock` and use your `Gemfile` to install all the gems specified in it. Bundler then will update the Gemfile.lock with the list of gems that were installed, as well as their versions. The next time `bundle` is run, the gems will be read from the Gemfile.lock file, rather than Gemfile. You commit this file to your repository so that when other people

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

work on your project and run `bundle install`, they get exactly the same versions that you have.

NOTE	**Ubuntu users** If you're running Ubuntu, you must install the `build-essential` package because some gems build native extensions and require the make utility. You may also have to install the `libxslt1-dev` package because the `nokogiri` (which will be used later) gem depends on it. You'll also need to install the `libsqlite3-dev` package to allow the `sqlite3` gem to install.

With the necessary gems for the application installed, you should run the `rspec:install` generator to set up a testing environment for the application:

```
$ bin/rails generate rspec:install
```

One more thing: it's sort of annoying to run `bundle exec rspec` every time we want to use Rspec. We can use bundler's 'binstubs' feature to generate stubs that eliminate the need for `bundle exec`. Run this:

```
$ bin/bundle binstubs rspec-rails
```

This will spit out some information about the stubs that `rspec-rails` supports:

```
rspec-rails has no executables, but you may want one
  from a gem it depends on.
railties has: rails
rspec-core has: autospec, rspec
```

Go ahead and generate a stub for `rspec` and `autospec` by typing `bin/bundle binstubs rspec-core`. If you look inside your `bin` directory, you should see both stubs there.

With this generated code in place, you should make a commit so you have

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 inside 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 3.6 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 like the following listing. It's common convention, but not mandatory, to call the environment's database [app_name]_[environment].

Listing 3.7 config/database.yml, PostgreSQL example

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You're welcome to change the database if you wish. Rails will go about its business. 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 switch your data over for you.

It's generally wise to use different names for the different database environments because if you use the same database in development and test, the database would be emptied of all data when the tests were run, eliminating anything you may have set up in development mode. You should never work on the live production database directly unless you are 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 3.8 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.

3.4 Beginning your first feature

You now have version control for your application, and you're hosting it on GitHub. You also cheated a little and got a pre-prepared stylesheet.[11]

Footnote 11 We wouldn't have a pre-prepared stylesheet in the real world, where designers would design at the same time we're developing features.

It's now time to write your first Capybara-based test, which isn't nearly as daunting as it sounds. We explore things such as models and RESTful routing while we do it. It'll be simple, promise!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

3.4.1 Creating projects

The CRUD (*c*reate, *r*ead, *u*pdate, *d*elete) acronym is something you 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 from your database throughout your application. The following section goes through the beginnings of generating a CRUD interface for a resource called *Project* by applying the BDD practices 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 continue to apply these practices to ensure 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 create a resource representing projects in your application by first writing a test for it, and then creating a controller and model, and a resource route. Then you add a validation to ensure that no project can be created without a name.

When you're done with this feature, you will have a form that looks like Figure 3.4.

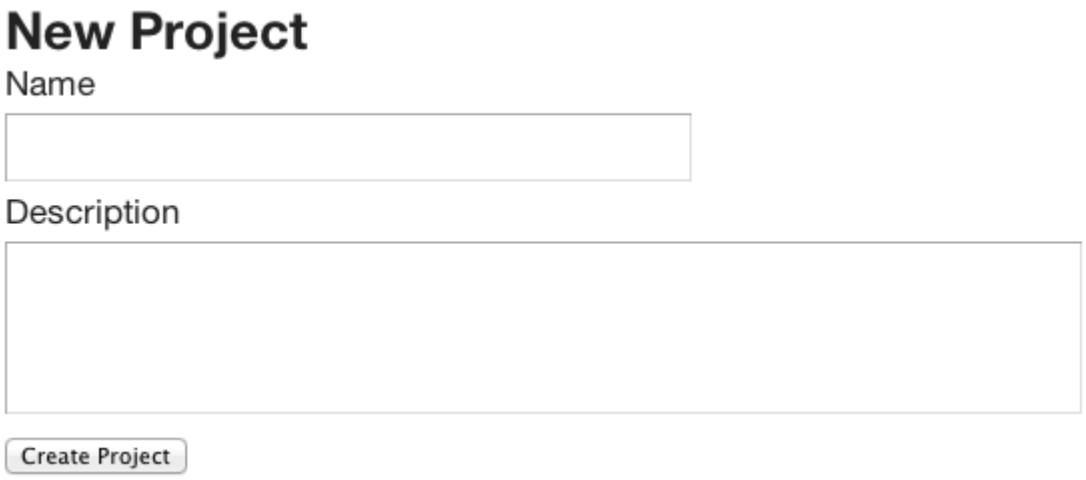

Figure 3.4 Form to create projects

First, you should create a new directory at spec/features and then in a file called spec/features/creating_projects_spec.rb you will put the test that would make sure that this feature works correctly when it is fully implemented. This code is the code shown in the following listing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.9 spec/features/creating_projects_spec.rb

```
require 'spec_helper'

feature 'Creating Projects' do
  scenario "can create a project" do
    visit '/'

    click_link 'New Project'

    fill_in 'Name', with: 'TextMate 2'
    fill_in 'Description', with: 'A text-editor for OS X'
    click_button 'Create Project'

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

To run this test, run `bin/rspec`. This command will run all of your specs, and display your application's first test's first failure:

```
1) Creating Projects can create a project
   Failure/Error: visit '/'
   ActionController::RoutingError:
     No route matches [GET] "/"
```

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

Rails is claiming that can't handle the `/` route and is throwing an exception. 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 content seen in the following listing (comments removed).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.10 config/routes.rb

```
Ticketee::Application.routes.draw do
end
```

The comments are good for a read if you're interested in the other routing syntax, but they're not necessary at the moment. To define a root route, you use the root method like this inside the block for the `draw` method:

```
Ticketee::Application.routes.draw do
  root "projects#index"
end
```

This defines a route for requests to `/` (the root route) to point at the `ProjectsController`'s `index` action. This controller doesn't exist yet, and so the test should probably complain about that if you got the route right. Run `bin/rspec` to find out.

```
Failure/Error: visit '/'
 ActionController::RoutingError:
   uninitialized constant ProjectsController
```

This error is happening because the root 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 cannot find it, you will get this error. To define this 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 inside 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:

```
$ bin/rails generate controller projects
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You may be wondering, why are we 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 projects controller, the most important of these being 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 your 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/controllers/projects_controller_spec.rb
invoke  helper
create    app/helpers/projects_helper.rb
invoke    rspec
create      spec/helpers/projects_helper_spec.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/projects.js.coffee
invoke    scss
create      app/assets/stylesheets/projects.css.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 are first created.

`invoke erb` signifies that the Embedded Ruby (ERB) generator was

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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.

`invoke rspec` shows that the RSpec generator was invoked during the generation also. This means that RSpec has generated a new file at spec/controllers/projects_controller_spec.rb, which you can use to test your controller; but not right now. [12]

Footnote 12 By generating RSpec tests rather than Test::Unit tests, a long-standing issue within Rails has been fixed. In previous versions of Rails, even if you specified the RSpec gem, all the default generators would still generate Test::Unit tests. With Rails, the testing framework you use is just one of a large number of configurable things in your application.

Finally, the assets for this controller are generated. There's two files generated here: app/assets/javascripts/projects.js.coffee and app/assets/stylesheets/projects.css.scss. The first file should contain any JavaScript related to the controller, written as CoffeeScript[13]. The second file should contain any CSS related to the controller, written using SCSS[14]. In the development environment, these files are automatically parsed into Javascript and CSS respectively.

Footnote 13 Coffeescript: http://coffeescript.org

Footnote 14 http://sass-lang.com/

Now, 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 `bin/rspec` again, it declares that the index action is missing:

```
Failure/Error: visit '/'
AbstractController::ActionNotFound:
 The action 'index' could not be found for ProjectsController
```

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.11 app/controllers/projects_controller.rb

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

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

```
ActionView::MissingTemplate:
Missing template projects/index, application/index
                 with {:locale=>[:en], :formats=>[:html],
                       :handlers=>[:erb, :builder, :raw,
                       :ruby, :jbuilder, :coffee]}.
Searched in:
  * ".../ticketee/app/views"
```

The error message isn't the most helpful to the untrained eye, but it's quite detailed. If you know how to put the pieces together, you can determine that it's trying to look for a template called projects/index or application/index, but it's not finding it. These templates are primarily kept at app/views, so it's fair to guess that it's expecting something like app/views/projects/index.

The extension of this file is composed of two parts: the *format* followed by the *handler*. In your output, you've got a handler of either `:erb` or `:builder` and a format of `:html`, so it's fair to assume from this that the file it's looking for is either index.html.erb or index.html.builder. Either of these file names for the `index` action's view is fine, but we'll use the first one, because we're wanting a HTML page rather than an XML document, which is what a builder template would provide.

The first part, *index*, is the name of the action; that's the easy part. The second part, *html*, indicates the format of this template. Actions in Rails can respond to different formats (using `respond_to`, which you saw in chapter 1); the default format is *html*. The third part, *erb*, indicates the templating language you're using,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

or the handler for this specific template. Templates in Rails can use different templating languages/handlers, but the default in Rails is ERB, hence the *erb* extension.

You could also create a file at app/views/application/index.html.erb to provide the view for the `index` action. This would work because the `ProjectsController` inherits from the `ApplicationController`. If you had another controller inherit from `ProjectsController`, you could put an action's template at app/views/application, app/views/projects or app/views/that_controller, and Rails would still pick up on it. This allows different controllers to share views in a simple fashion.

To generate this view, create the app/views/projects/index.html.erb file and leave it blank for now. You can run just the single spec for creating projects with `bin/rspec spec/features/creating_projects_spec.rb`. When you do, you get back to what looks like the original error:

```
Failure/Error: click_link 'New Project'
Capybara::ElementNotFound:
  Unable to find link "New Project"
```

Although this looks like the original error, the test is actually visiting the new root route for your application, making the first line in the test pass for real now. You've defined a homepage 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 navigating to 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 "Creating Projects" 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. Open app/views/projects/index.html.erb and put the link in by using the `link_to` method:

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This single line re-introduces two old concepts and one new one: ERB *output* tags, the `link_to` method (both of which we 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 are 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 a `<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 still complain of an `undefined method or local variable ,'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, as shown in the following listing.

Listing 3.12 resources :projects line, in config/routes.rb

```
Ticketee::Application.routes.draw do
  root "projects#index"

  resources :projects
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This is called a *resource* route, and it defines the routes to the seven *RESTful* actions in your projects controller. When something is said to be RESTful, it means it conforms to the Representational State Transfer (REST) architectural style[15]. Rails can't get you all the way there, but it can help. With Rails, this means the related controller has seven potential actions:

Footnote 15 http://en.wikipedia.org/wiki/Representational_state_transfer

- `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. Table 3.1 lists the routes, HTTP methods, and corresponding actions to make it clearer:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Table 3.1 Table 3.1 RESTful routing matchup

HTTP Method	Route	Action
GET	/projects	index
POST	/projects	create
GET	/projects/new	new
GET	/projects/:id	show
PATCH/PUT	/projects/:id	update
DELETE	/projects/:id	destroy
GET	/projects/:id/edit	edit

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.

To review the routes you've defined, you can run the `bin/rake routes` command and get output similar to that in table 3.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.13 bin/rake routes output

```
      Prefix Verb   URI Pattern                  Controller#Action
        root GET    /                            projects#index
    projects GET    /projects(.:format)          projects#index
             POST   /projects(.:format)          projects#create
 new_project GET    /projects/new(.:format)      projects#new
edit_project GET    /projects/:id/edit(.:format) projects#edit
     project GET    /projects/:id(.:format)      projects#show
             PATCH  /projects/:id(.:format)      projects#update
             PUT    /projects/:id(.:format)      projects#update
             DELETE /projects/:id(.:format)      projects#destroy
```

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 hardcoding 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 and generates the path according to this object. You'll see later how you can alter this path to be more user friendly, generating a URL such as `/projects/1-our-project` rather than the impersonal `/projects/1`.

The four paths mentioned earlier match up to the helpers in table 3.2.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Table 3.2 Table 3.2 RESTful routing matchup

URL	Helper
GET /projects	projects_path
GET /projects/new	new_project_path
GET /projects/:id	project_path
GET /projects/:id/edit	edit_project_path

Running `bin/rspec` now produces a complaint about a missing new action:

```
1) Creating Projects can create a project
Failure/Error: click_link 'New Project'
AbstractController::ActionNotFound:
  The action 'new' could not be found for ProjectsController
```

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

Listing 3.14 app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end

  def new
  end
end
```

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_link 'New Project'
ActionView::MissingTemplate:
  Missing template projects/new, application/new
              with {:locale=>[:en],
                    :formats=>[:html],
                    :handlers=>[:erb, :builder,
                                :raw, :ruby,
                                :jbuilder, :coffee]}.
Searched in:
   * ".../ticketee/app/views"
```

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. We will 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 `bin/rspec`.

```
Failure/Error: fill_in 'Name', :with => 'TextMate 2'
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` so you have an object to base the fields on. Change the `new` to this:

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

The `Project` constant is going to be a class, located at app/models/project.rb, thereby making it a *model*. A model is used to retrieve information from the database. Because this model inherits from Active Record, you don't have to set up anything extra. Run the following command to generate your first model:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ bin/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 this table and the specified fields. You can specify as many fields as you like after the model's name. They default to `string` type, so we didn't specify them. If we wanted to be explicit, we could use a colon, like this:

```
$ bin/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 to interact with a database what to do with those objects once they've been retrieved. A model is also the place where you define scopes (easy to use filters for database calls, done in Chapter 8) and associations (done in Chapter 5) and validations (seen later in this chapter), amongst other things. To perform any interaction with data in your database, you will be going through a model.[16]

Footnote 16 Although it is possible to perform database operations without a model within Rails, 99% of the time you'll want to use a model.

Migrations are effectively version control for the database. They are 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 are first defined. For example, the code shown in the following listing comes from the migration that was just generated:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.15 db/migrate/[date]_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration
  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 `bin/rake db:migrate`), it creates the table. When you roll the migration back (with `bin/rake 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:

```
class CreateProjects < ActiveRecord::Migration
  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 `self.up` method would be called if you ran the migration forward, and the `self.down` method if you ran it backwards.

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

```
class CreateProjects < ActiveRecord::Migration
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  def up
    remove_column :projects, :name
  end

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

This is because ActiveRecord 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.

The first line tells Active Record you want to create a table called `projects`. You call this method in 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, and the first argument is the name of that field. The `timestamps` method is special: it creates two fields, the `created_at` and `updated_at` datetime fields, which are by default set to the current time in coordinated universal time (UTC) 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:

```
$ bin/rake db:migrate
```

This command migrates 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 `bin/rake db:migrate` will migrate them in the order they were created.

With this model created and its related migration run on both the development and test databases, you can now run `bin/rspec` and get a little further:

```
Failure/Error: fill_in 'Name', with: 'TextMate 2'
Capybara::ElementNotFound:
  Unable to find field "Name"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Now you are back to the element not found error. To add this field to the `new` action's view, you can put it inside a form, but not just any form. A `form_for`, like in the following listing:

Listing 3.16 app/views/projects/new.html.erb

```
<h2>New Project</h2>
<%= form_for(@project) do |f| %>
  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>

  <p>
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  <%= f.submit %>
<% end %>
```

❶

So many new things! The `form_for` call ❶ allows you to specify the fields that belong to the object specified. In this case, all you're doing is specifying that the form should have a text field that belongs to the name attribute of the `@project` object.

The `form_for` method is passed the `@project` object as the first argument, and with this, the helper does more than simply place a form tag on the page. `form_for` inspects the `@project` object and creates a form builder specifically for that object. The two main things it inspects are (1) whether or not it's a new record and (2) what the class name is.

Determining what `action` attribute the form has (where the form sends to) is dependent on whether or not the object is a new record. A record is classified as new when it hasn't been saved to the database, and this check is performed internally to 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. In this case, it is /projects. Because the record is new, the path is /projects and the method for the form is `POST`. Therefore, a request is sent to the `create` action in `ProjectsController`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

After that part of `form_for` is complete, you use the block syntax to receive an `f` variable, which is a `FormBuilder` object. You can use this object to define your forms fields. The first element you define is a `label`. `label` tags correspond to their field elements on the page and serve two purposes in our application. First, they give users a larger area to click rather than just the field, radio button, or checkbox itself. The second purpose is so you can reference the label's text in the test, and Capybara will know what field to fill in.

TIP **Alternative label naming**

If you want to customize a label, you can pass the `label` method a second argument:

```
<%= f.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 id="project_name" name="project[name]"
       size="30" type="text">
```

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

Now, running `bin/rspec spec/features/creating_projects_spec.rb` once more, you can see that your spec is one step closer to finishing: the field fill-in step has passed.

```
Failure/Error: click_button 'Create Project'
AbstractController::ActionNotFound:
 The action 'create' could not be found for ProjectsController
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Capybara finds the label containing the `"Name"` text you ask for in your scenario and fills out the corresponding field. 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: "TextMate 2"
# or
fill_in "project[name]", with: "TextMate 2"
```

NOTE **Field selector preferences**
Some argue that using the id or name of the field is a better way because these attributes don't change as often as labels may. But to keep things simple, you should continue using the label name.

The spec is now complaining of a missing action called `create`. To define this action, you define the `create` method underneath the new method in the `ProjectsController`, as in the following listing:

Listing 3.17 create action of ProjectsController

```
def create
  @project = Project.new(params[:project])                    ❶

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

The `Project.new` method ❶ takes one argument, which is a list of attributes that will be assigned to this new `Project` object. The `params` method is available inside all controller actions and returns the parameters passed to the action, such as those from the form or query parameters from a URL, as a object.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 is:

```
{
  "commit"     => "Create Project",
  "action"     => "create",
  "project"    => {
    "name" => "TextMate 2",
    "description" => "A text-editor for OS X"
  },
  "controller" => "projects"
}
```

TIP **Inspecting params**

If you'd like to inspect the `params` hash at any point in time, you can put `p params` in any action and then run the action either by accessing it through `rails server` or by running a scenario that will run an action containing this line. This outputs to the console the `params` hash and is equivalent to doing `puts params.inspect`.

All the hashes nested inside this hash are also `HashWithIndifferentAccess` hashes. If you want to get the name key from the `project` hash here, you can use either `{ :name => "TextMate 2" }[:name]`, as in a normal `Hash` object, or `{ :name => "TextMate 2" }['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, 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 being `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 inside

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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.

When `new` receives this `HashWithIndifferentAccess`, it generates a new `Project` object with the *attributes* based on the parameters passed in. The `Project` object will have a `name` attribute set to the value from `params[:project][:name]`.

You call `@project.save` to save your new `Project` object into the *projects* table. Before that happens though, Rails will run all the data validations on the model, ensuring it's correct. At the moment, you have no validations on the model and 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's also a `HashWithIndifferentAccess`. These messages are stored in the session and are cleared at the completion of the next request. Here you set the `:notice` key 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 takes either an object, as in the `create` action, or a path to redirect to as a string. If an object is given, Rails inspects that object 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 `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 Redirected to /projects/1, which is the currently nonexistent `show` action.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

TIP **Combining redirect_to and flash**

You can combine the `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 do not wish to use either `notice` or `alert`, you must specify `flash` as a hash:

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

With the `create` action now established within your `ProjectsController`, the test should be getting a little further. Find out by running `bin/rspec`. You will see this error:

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

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.' This feature is new to Rails 4. We don't want to accept just any parameters: we want to accept the ones that we want, and no more. That way, someone can't mess around with our app by sending funky information in. Change your `ProjectsController` code for the create action to look like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
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
```

We now call the `require` method on our `params`, and we require that the `:project` key exists. We also allow it to have `:name` and `:description` entries. Finally, we wrap up that logic into a method so that we can use it in other actions, and make it private so we don't expose it as some kind of weird action!

Now that that's done, run `bin/rspec` again, and you'll get a new error:

```
Failure/Error: click_button 'Create Project'
AbstractController::ActionNotFound:
 The action 'show' could not be found for ProjectsController
```

The `show` action is responsible for displaying a single record's information. To retrieve a record, you need an ID to fetch. You know the URL for this page is going to 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 right now. Make sure it comes above the `private` declaration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.18 show action of ProjectsController

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

You pass the `params[:id]` object to `Project.find` here, which 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 cannot find a record matching that ID, it raises an `ActiveRecord::RecordNotFound` exception.

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

```
Failure/Error: click_button 'Create Project'
ActionView::MissingTemplate:
Missing template projects/show, application/show
            with {:locale=>[:en],
                  :formats=>[:html],
                  :handlers=>[:erb, :builder,
                              :raw, :ruby,
                              :jbuilder, :coffee]}.
Searched in:
   * ".../ticketee/app/views"
```

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

```
<h2><%= @project.name %></h2>
```

Now when you run the test again with `bin/rspec spec/features/creating_projects_spec.rb`, you see this message:

```
Failure/Error: expect(page).to have_content('Project has been created.')
  expected there to be text "Project has been created." in "TextMate 2"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This error message shows that the 'Project has been created.' text is not being displayed on the page. Therefore, you must put it somewhere, but where? 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 from anywhere.

The application layout is quite the interesting file:

```
<!DOCTYPE html>
<html>
<head>
  <title>Ticketee</title>
  <%= stylesheet_link_tag    "application", media: "all",
    "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application",
    "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</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_include_tag`, and `csrf_meta_tags`.

`stylesheet_link_tag` is for including stylesheets from the app/assets/stylesheets directory. Using this tag results in the following output:

```
<link data-turbolinks-track="true"
      href="/assets/application.css?body=1"
      media="all"
      rel="stylesheet" />
<link data-turbolinks-track="true"
      href="/assets/projects.css?body=1"
      media="all"
      rel="stylesheet" />
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `/assets` path is served by a gem called `sprockets`. In this case, the tag specifies the /assets/application.css path, and any route prefixed with `/assets` is served by Sprockets. Sprockets provides a feature commonly referred to as the *Asset Pipeline*. When files are requested through this, they're pre-processed and then served out to the browser.

There's also a second tag for our `projects.css` file. In development mode, Rails generates tags for all of your stylesheets and JavaScripts separately, for ease of debugging. If we run the application in production mode, we get something very different:

```
<link data-turbolinks-track="true"
      href="/stylesheets/application.css"
      media="all"
      rel="stylesheet" />
```

This single stylesheet is all of our stylesheets, concatenated and minified. That way, your users will load up all your styles on their first visit, and they'll be cached for the rest of their stay, increasing overall performance.

When the `/assets/application.css` asset is requested, Sprockets looks for a file beginning with application.css inside the asset paths for your application. The three asset paths it will search by default are `app/assets`, `lib/assets` and `vendor/assets`, in that order. Some gems will add to these lookup paths, and so you will be able to use assets from within them as well.

If the file has any additional extensions on it, such as a file called application.css.scss, Sprockets will look up a preprocessor for the `scss` extension and run the file through that, before serving it as CSS. You can chain together any number of extensions and Sprockets will parse the file for each extension, working right to left.

The `application.css` file that is being searched for here lives at `app/assets/application.css.scss` and is the stylesheet you created a short while ago. This has an additional `scss` extension on it, so will be preprocessed by the Sass preprocessor before being served as CSS by the `stylesheet_link_tag` call in the application layout.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

TIP	**Using Sass or SCSS** For your CSS files, you can use the Sass or SCSS languages to produce more powerful stylesheets. Your application depends on the `sass-rails` gem, which itself depends on `sass`, the gem for these stylesheets. We don't go into detail here because the Sass site covers most of that ground: http://sass-lang.com/. Rails automatically generates stylesheets for each controllers that uses Sass, as indicated by its .css.scss extensions. This final extension tells Sprockets to process the file using Sass before serving it as CSS.

`javascript_include_tag` is for including JavaScript files from the javascript directories of the Asset Pipeline. When the `application` string is specified here, Rails loads the app/assets/javascripts/application.js file, which looks like this:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

This file includes some Sprockets-specific code that will include the `jquery.js` and `jquery_ujs.js` files located in the `jquery-rails` gem that the application's `Gemfile` specifies as a dependency of the application. It also includes the JavaScript for Turbolinks, which is a feature that we'll discuss later. It compiles these three files, plus all the files in the `app/assets/javascripts` directory with the `//= require_tree .` into one superfile called `application.js`, which is referenced by this line in the output of your pages:

```
<script src="/assets/application.js" type="text/javascript"></script>
```

This file is also served through the `sprockets` gem. As with your CSS

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

stylesheets, you can use an alternative syntax called CoffeeScript (http://coffeescript.org), which provides a simpler JavaScript syntax that compiles into proper JavaScript. Just as with the Sass stylesheets, Rails generates CoffeeScript files inside app/assets/javascripts with the extension .js.coffee, indicating to Sprockets they are to be parsed by a CoffeeScript interpreter first, then served as JavaScript. We use CoffeeScript a little later, in chapter 9.

`csrf_meta_tags` is for protecting your forms from cross-site request forgery (CSRF)[17] attacks. It 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 submitted. If the key is invalid, an `ActionController::InvalidAuthenticityToken` exception occurs and the user's session is reset as a precaution.

Footnote 17 http://en.wikipedia.org/wiki/CSRF

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, value| %>
  <div class='flash' id='<%= key %>'>
    <%= value %>
  </div>
<% end %>
```

This code renders all the `flash` messages that get defined, regardless of their name and the controller that they come from. These lines will display the `flash[:notice]` that you set up in the `ProjectsController`'s `create`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

action. Run `bin/rspec` again and see that the test is now fully passing!

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

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

```
**.

Pending:
  Project add some examples to (or delete)
    /.../ticketee/spec/models/project_spec.rb
    # No reason given
    # ./spec/models/project_spec.rb:4
  ProjectsHelper add some examples to (or delete)
    /.../ticketee/spec/helpers/projects_helper_spec.rb
    # No reason given
    # ./spec/helpers/projects_helper_spec.rb:14

Finished in 0.31466 seconds
3 examples, 0 failures, 2 pending

Randomized with seed 24201
```

The key part is that 'or delete.' Let's delete those two files, since we're not using them yet.

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

Afterwards, run `bin/rspec` one more time:

```
.

Finished in 0.32084 seconds
1 example, 0 failures

Randomized with seed 26038
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 any process. Remember when you were learning to drive a car? You didn't drive like Michael Schumacher as soon as you got into the car. You learned by doing it slowly and methodically. As you progress, it becomes quicker, as all things do with practice.

3.4.3 Committing changes

Now you're at a point where all (just the one for now) your specs are running, and 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 revert all your changes by using:

```
$ git checkout .
```

WARNING **A warning about git checkout**

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 allows you to 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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ 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 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 inside spec/features/creating_projects_spec.rb, add the four lines shown in the following listing.

Listing 3.19 spec/features/creating_projects_spec.rb

```
project = Project.where(name: "TextMate 2").first

expect(page.current_url).to eql(project_url(project))

title = "TextMate 2 - Projects - Ticketee"
expect(page).to have_title(title)
```

The first line here uses two methods to find a project: `where` and `first`. `where` will give us all of the `Project`s that have a name "Textmate 2". The `first` chained onto our `where` will give us just the first one. You want to find the project that has just been created so you can use it later in the test. The second line ensures that you're on what should be the `show` action inside the `ProjectsController`. The third line finds the `title` element on the page by using Capybara's `find` method and checks using `have_content` that this element contains the content of "TextMate 2 - Projects - Ticketee". If you run `bin/rspec spec/features/creating_projects_spec.rb` now, you'll see this error:

```

```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: expect(find("title")).to have_content(title)
expected there to be text "TextMate 2 - Projects - Ticketee"
  in "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 = "TextMate 2 - Projects - Ticketee" %>
```

This sets up a `@title` instance variable in the template. Because the template is rendered before the layout, you are able to then use this variable inside the layout. However, if a page doesn't have a `@title` variable set, there should be a default title of just "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 will return `nil` as their values. 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 now pass when you run `bin/rspec`:

```
1 example, 0 failures
```

With this test now passing, you can change your code and have a solid base to ensure that whatever you change works as you expect. To demonstrate this point, change the code in your `show` to use a *helper* instead of setting a variable.

Helpers are methods you can define in the files inside app/helpers, and they are made available in your views. Helpers are for extracting the logic from the views, as views should just be about displaying information. Every controller that comes

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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. Now open this file (app/helpers/application_helper.rb) and insert the code from the following listing.

Listing 3.20 app/helpers/application_helper.rb

```
module ApplicationHelper                                    (1)
  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 (*), (1) any arguments passed from this point will be available inside 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 the opposite keyword to `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 of `"title"`. Inside this content block, you join the parts together using a hyphen (-), meaning this helper will output something like `"TextMate 2 - Projects - Ticketee"`.

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

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

You don't need `Ticketee` here any more because the method puts it in for you. Let's replace the `title` tag line in app/views/layouts/application.html.erb with this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<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 will only be defined if `content_for(:title)` is called somewhere, like 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 just output the word `Ticketee`, and that becomes the title.

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

```
1 example, 0 failures
```

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

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

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 that they can fix it and attempt to create the project again.

With this in mind, you should add another test for ensuring that this happens to spec/features/creating_projects_spec.rb using the code from the listing:

Listing 3.21 spec/features/creating_projects_spec.rb

```
scenario "can not create a project without a name" 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 here are identical to the ones you placed inside the other scenario. You should eliminate this duplication by making your code DRY (*D*on't *R*epeat *Y*ourself!). This is another term you'll hear a lot in the Ruby world. 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 define a *before* block. For RSpec, `before` blocks will be run before *every* single test inside the file.

To DRY this file up, change spec/features/creating_projects_spec.rb to look like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 3.22 spec/features/creating_projects_spec.rb

```
require 'spec_helper'

feature 'Creating Projects' do
  before do
    visit '/'

    click_link 'New Project'
  end

  scenario "can create a project" do
    fill_in 'Name', with: 'TextMate 2'
    fill_in 'Description', with: 'A text-editor for OS X'
    click_button 'Create Project'

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

    project = Project.where(name: "TextMate 2").first

    expect(page.current_url).to eql(project_url(project))

    title = "TextMate 2 - Projects - Ticketee"
    expect(page).to have_title(title)
  end

  scenario "can not create a project without a name" 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 whole lot better! Now when you run `bin/rspec`, it will fail because it cannot see the error message that it's expecting to see on the page:

```
Failure/Error: expect(page).to
  have_content("Project has not been created.")
  expected there to be text "Project has not been created."
  in "Project has been created."
```

To get this test to do what you want it to do, you will need to add a validation. Validations are defined on the model and are run before the data is saved to the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 3.23 app/models/project.rb

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

The validates method usage here is exactly how you used it for the first time in the first chapter. 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, such as the `:uniqueness` key, which, 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.

WARNING **Beware race conditions with uniqueness validator**

The `validates :foo, uniqueness: true` validator works by checking to see if a record matching the validation criteria exists already. If this record doesn't exist, the validation will pass.

A problem arises if two connections to the database both make this check at almost exactly the same time. Both connections will claim that a record doesn't exist and therefore will allow a record to be inserted for each connection, resulting in nonunique records.

A way to prevent this is to use a database uniqueness index so the database, not Rails, does the uniqueness validation. For information how to do this, consult your database's manual.

While this problem doesn't happen all the time, it *can* happen, so it's something to watch out for.

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:

```
$ bin/rails console
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

or with it's shorter alternative:

```
$ bin/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 4.0.0)
irb(main):001:0>
```

At this prompt[18], 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:

Footnote 18 Although you may see something similar to `ruby-2.0.0:001 >` too, which is fine.

```
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 is not persisted (saved) in the database.

If you comment out or remove the validation from inside the `Project` class

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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: "2010-05-06 01:00:15",
             updated_at: "2010-05-06 01:00:15">
```

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[19] 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 it 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.

Footnote 19 The first argument for this method is the attributes. If there is no argument passed, then all attributes default to their default values.

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, showing the exception followed by a large stacktrace, which you can safely ignore for now. You are notified which validation failed. To stop it from failing, you must pass in a `name` attribute, and it will happily return a saved `Project` object:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
irb(main):002:0> Project.create!(name: "TextMate 2")
=> #<Project id: 1,
             name: "TextMate 2",
             description: nil,
             created_at: "[timestamp]",
             updated_at: "[timestamp]">
```

That's how to use `create` to test it in the console, but in your `ProjectsController`, you use the method shown in the following listing instead.

Listing 3.24 project creation inside ProjectsController's new action

```
@project = Project.new(project_params)
@project.save
```

`save` doesn't raise an exception if validations fail, as `create!` did, but instead returns `false`. If the validations pass, `save` returns `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`, as in the following listing.

Listing 3.25 create action for ProjectsController

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

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    flash[:alert] = "Project has not been created."          (1)

    render "new"
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Now if the `@project` object has a name attribute -- meaning that it is "valid" -- then `save` returns `true` and executes everything between the `if` and the `else`. If it isn't valid, then everything between the `else` and the following `end` is executed. In the `else`, you specify a different key ❶ 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 `bin/rspec spec/features/creating_projects_spec.rb` here, the line in the spec that checks for the "Project has not been created." message is now not failing, and so it's going 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 this test is failing again.

```
Failure/Error: expect(page).to have_content("Name can't be blank")
  expected there to be content "Name can't be blank" in ...
```

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

Directly under this `form_for` line, on a new line, insert the following into app/views/projects/new.html.erb to display the error messages for your object inside the 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 %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Error messages for the object represented by your form, the `@project` object, will now be displayed by the `each`. When you run `bin/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 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 alternatives -- 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, learned about the beautiful Bundler gem in the process, and then ran the installers for these gems so your application was then fully configured to use them. For instance, after running `bin/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, we 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 is broken by running `bin/rspec spec`, a command that runs all

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 just isn't any fun.

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

Index Terms

ActiveRecord::Base#create
ActiveRecord::RecordNotFound
Agile
app/assets
capybara
config/database.yml
content_for?
controller generator
create_table
created_at
CRUD
Displaying flash messages
DRY
Environments
flash
Git
git checkout
git commit
git config
git push
git remote add
git stash
HashWithIndifferentAccess
javascript_include_tag
label, FormBuidler
model generator
routing, resources
rspec-rails
Sass
Sprockets
Sprockets manifest file
stylesheet_link_tag
text_field, FormBuilder
timestamps
updated_at
Validators, presence
Validators, uniqueness
yield

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 just 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 be doing this using behavior-driven development the whole way through, continuing your use of the RSpec and Capybara gems. 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_girl, as well as a way to make 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 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 4.1 spec/features/viewing_projects_spec.rb

```
require 'spec_helper'

feature "Viewing projects" do
  scenario "Listing all projects" do
    project = FactoryGirl.create(:project, name: "TextMate 2")
    visit '/'
    click_link 'TextMate 2'
    expect(page.current_url).to eql(project_url(project))
  end
end
```

Now, to run this single test, you can run `bin/rspec spec/features/viewing_projects_spec.rb`. When you do this, you'll see the following failure:

```
Failure/Error: project = FactoryGirl.create(:project,
                                            name: "TextMate 2")
  NameError:
    uninitialized constant FactoryGirl
```

The `FactoryGirl` constant is defined by another gem: the `factory_girl` gem.

4.1.1 The Factory Girl

The `factory_girl` gem, created by thoughtbot,[1] 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. you can use to run our tests on.

Footnote 1 Thoughtbot's website: http://thoughtbot.com.

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

```
group :test do
  gem 'capybara', '2.1.0'
  gem 'factory_girl_rails', '~> 4.2.1'
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

To install, run `bin/bundle`. With the `factory_girl_rails` gem installed, the `FactoryGirl` constant will be defined. Run `bin/rspec spec/features/viewing_projects_spec.rb` again and you will see a new error:

```
Failure/Error: project = FactoryGirl.create(:project,
                                            name: "TextMate 2")
   ArgumentError:
     Factory not registered: project
```

When using Factory Girl, you must create factories. If a factory isn't registered with Factory Girl, you'll get the above error. To register/create a factory, create a new directory inside spec called factories and then inside that directory create a new file called project_factory.rb. Fill that file with the content from the following listing:

Listing 4.2 spec/factories/project_factory.rb

```
FactoryGirl.define do
  factory :project do
    name "Example project"
  end
end
```

(1)

When you define the factory inside this file, you give it a default name. The `name: name` part of this method call inside spec/features/viewing_projects_spec.rb changes the default name to the one passed in. You use factories here because you needn't be concerned about any other attribute on the `Project` object. If you weren't using factories, you'd have to use this method to create the object instead:

```
Project.create(name: name)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

While this code is about the same length as its `Factory` 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 to contain this new field. When you use factories, 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—now how about some practice? Let's see what happens when you run `bin/rspec spec/features/viewing_projects_spec.rb` again:

```
Failure/Error: click_link 'TextMate 2'
Capybara::ElementNotFound:
  Unable to find link "TextMate 2"
```

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 "TextMate 2" 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're going to do now. Open app/views/projects/index.html.erb and add the contents of the following listing underneath the first link.

Listing 4.3 app/views/projects/index.html.erb

```
<h2>Projects</h2>
<ul>
  <% @projects.each do |project| %>
    <li><%= link_to project.name, project %></li>
  <% end %>
</ul>
```

If you run the spec again, you get this error, which isn't helpful at first glance:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: visit '/'
 ActionView::Template::Error:
   undefined method `each' for nil:NilClass
 # ./app/views/projects/index.html.erb:3 ...
```

This error points at line 5 of your app/views/projects/index.html.erb file. From this you can determine that the error has something to do with the `@projects` variable. This variable isn't yet been defined, and because there's no `each` method on `nil`, you get this error. As mentioned in chapter 3, instance variables in Ruby return `nil` rather than raise an exception if they're undefined. Watch out for this in Ruby—as seen here, it can sting you hard.

To define this variable, open `ProjectsController` at app/controllers/projects_controller.rb and change the `index` method definition to look like the following listing.

Listing 4.4 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 `Array`-like object. Now that you've put all the pieces in place, you can run the feature with `bin/rspec spec/features/creating_projects_spec.rb`, and it should all pass:

```
1 example, 0 failures
```

The spec now passes. Is everything else still working, though? You can check by running `bin/rspec`. Rather than just running the one test, this command will run all the tests inside the spec directory. When you run this command, you should see this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
3 examples, 0 failures
```

All of the specs are passing, meaning all of the functionality you've written so far is working as it should. Commit and push this using:

```
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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 4.5 spec/features/editing_projects_spec.rb

```
require 'spec_helper'

feature "Editing Projects" do
  scenario "Updating a project" do
    FactoryGirl.create(:project, name: "TextMate 2")

    visit "/"
    click_link "TextMate 2"
    click_link "Edit Project"
    fill_in "Name", with: "TextMate 2 beta"
    click_button "Update Project"

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

If you remember, `FactoryGirl#create` will build us an entire object, and let us tweak the defaults. In this case, we're changing the title.

Also, it's common for tests to take this overall form: arrange, act, assert. 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.

In this story, you again use the `bin/rspec` command to run just this one feature: `bin/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:

```
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 this link underneath all the code currently in that file:

```
<%= link_to "Edit Project", edit_project_path(@project) %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `edit_project_path` method generates the link to the `Project` object, pointing at the `ProjectsController`'s `edit` action. This method is provided to you because of the `resources :projects` line in config/routes.rb.

If you run `bin/rspec spec/features/editing_projects_spec.rb` again, it now complains about the missing `edit` action:

```
The action 'edit' could not be found for ProjectsController
```

You should now define this action in your `ProjectsController`, underneath the `show` action, as in the following listing.

Listing 4.6 app/controllers/projects_controller.rb

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

As you can see, this action works in an identical fashion to the `show` action, where the ID for the resource is automatically passed as `params[:id]`. Let's work on DRYing[2] this up once you're done with this controller. When you run the spec again, you're told that the `edit` view is missing:

Footnote 2 As a reminder: DRY = Don't Repeat Yourself!

```
Failure/Error: click_link "Edit Project"
ActionView::MissingTemplate:
Missing template projects/edit, application/edit with
  {:locale=>[:en],
   :formats=>[:html],
   :handlers=>[:erb, :builder, :raw, :ruby, :jbuilder, :coffee]}.
  Searched in: * "/Users/steve/src/ticketee/app/views"
```

It looks like you need to create this template. The `edit` action's form is going

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

to be very similar to the form inside 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*.

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:

```
<%= form_for(@project) do |f| %>
  <% 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 %>

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

  <p>
    <%= f.label :description %><br />
    <%= f.text_field :description %>
  <%= f.submit %>
<% end %>
```

This first section is where we deal with all of the error handling for this form. Rails doesn't handle this for you because people need to heavily customize the HTML, but this will serve our needs for now.

Then create a new file called app/views/projects/_form.html.erb and put the code that you've just extracted from the new template into this new file. The `new` template should now use this partial to show the form. To do this, just put this line in app/views/projects/new.html.erb:

```
  <%= render "form" %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Now, you need to create the `edit` action's template. Create a new file at app/views/projects/edit.html.erb and put the content from this listing in it:

Listing 4.7 app/views/projects/edit.html.erb

```
<h2>Edit project</h2>

<%= render "form" %>
```

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 `bin/rspec spec/features/editing_projects_spec.rb`:

```
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 fails when the "Update Project" button is pressed, because it cannot find the `update` action inside the `ProjectsController`. To make this work, you're going to need to create that `update` action.

4.2.2 The update action

As the following listing shows, you can now define this `update` action underneath the `edit` action in your controller:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 4.8 app/controllers/projects_controller.rb

```
def update
  @project = Project.find(params[:id])                    (1)
  @project.update(project_params)

  flash[:notice] = "Project has been updated."
  redirect_to @project
end
```

Notice the new method here, `update`(1). 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 are valid. This method, like `save`, returns `true` if the update is valid or `false` if it is not.

Now that you've implemented the `update` action, let's see how the test is going by running `bin/rspec spec/features/editing_projects_spec.rb`:

```
1 example, 0 failures
```

What happens if somebody fills in the name field with a blank value? The user receives an error, just as in the `create` action. You should 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 are always going to be the same: A project needs to exist, then a user goes to the homepage, finds a project, clicks "Edit Project". Change spec/features/editing_projects_spec.rb so it looks like the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 4.9 spec/features/editing_projects_spec.rb

```
require 'spec_helper'

feature "Editing Projects" do
  before do
    FactoryGirl.create(:project, name: "TextMate 2")

    visit "/"
    click_link "TextMate 2"
    click_link "Edit Project"
  end

  scenario "Updating a project" do
    fill_in "Name", with: "TextMate 2 beta"
    click_button "Update Project"

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

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 underneath the one currently in this file:

Listing 4.10 spec/features/editing_projects_spec.rb

```
scenario "Updating a project with invalid attributes is bad" do
  fill_in "Name", with: ""
  click_button "Update Project"

  expect(page).to have_content("Project has not been updated.")
end
```

When you run `bin/rspec spec/features/editing_projects_spec.rb`, the filling in the "Name" works, but when the form is submitted the test doesn't see the "Project has not been

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

updated." message:

```
expected there to be content "Project has not been updated." in ...
```

Again, this error means that it was unable to find the text "Project has not been updated." on the page. This is because you haven't written any code to test for what to do if the project being updated is now invalid. In your controller, you should now 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 4.11 update action inside ProjectsController

```
def update
  @project = Project.find(params[:id])
  if @project.update(project_params)
    flash[:notice] = "Project has been updated."
    redirect_to @project
  else
    flash[:alert] = "Project has not been updated."
    render "edit"
  end
end
```

And now you can see that the feature passes when you rerun `bin/rspec spec/features/editing_projects_spec.rb`:

```
2 examples, 0 failures
```

Again, you should ensure everything else is still working by running `bin/rspec`; you should see this summary:

```
5 examples, 0 failures
```

Let's make a commit and push now:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git add .
git commit -m "You can now update a project."
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're going to need a feature to get going: a Delete Project link on the show page that, when clicked, prompts the user for confirmation.[3] You put the feature at spec/features/deleting_projects_spec.rb using the following listing.

Footnote 3 Although the test won't check for this prompt, due to the difficulty in testing JS confirmation boxes in tests.

Listing 4.12 spec/features/deleting_projects_spec.rb

```
require 'spec_helper'

feature "Deleting projects" do
  scenario "Deleting a project" do
    FactoryGirl.create(:project, name: "TextMate 2")

    visit "/"
    click_link "TextMate 2"
    click_link "Delete Project"

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

    visit "/"

    expect(page).to have_no_content("TextMate 2")
  end
end
```

When you run this test using `bin/rspec spec/features/deleting_projects_spec.rb`, the first couple of lines

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

will pass because they're just creating a project using Factory Girl, visiting the home page and then clicking the link to take us to the project page. The fourth line inside this scenario will fail however, with this message:

```
Failure/Error: click_link "Delete Project"
Capybara::ElementNotFound:
  Unable to find link "Delete Project"
```

To get this to work, you'll need to add a "Delete Project" link to the `show` action's template, app/views/projects/show.html.erb. You should put this on the line after the "Edit Project" link using this code:

```
<%= 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 `:confirm`.

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 we use the `DELETE` method, but let's review why. If all actions are available by GET requests, then anybody can send you a link to, say, the `destroy` action for one of your controllers, and if you click on that, 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 link from the site to make the proper request to delete this resource.

The `:confirm` option brings up a prompt, using JavaScript, that asks users if they're sure of what they clicked on. Because Capybara doesn't support JavaScript by default, this prompt is ignored, so you don't have to tell Capybara to click OK on the prompt—there is no prompt because Rails has a built-in fallback for users

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

without JavaScript enabled. If you launch a browser and follow the steps in the feature to get to this "Delete Project" link, and then clicked on the link, you would see the confirmation prompt. This prompt is exceptionally helpful for preventing accidental deletions.

When you run the spec again with `bin/rspec spec/integeration/deleting_projects_spec.rb`, it complains of a missing `destroy` action:

```
Failure/Error: click_link "Delete Project"
AbstractController::ActionNotFound:
  The action 'destroy' could not be found for ProjectsController
```

The final action you need to implement in your controller and we'll put it underneath the `update` action. This action is shown in the following listing.

Listing 4.13 destroy action inside ProjectsController

```
def destroy
  @project = Project.find(params[:id])                    ❶
  @project.destroy

  flash[:notice] = "Project has been destroyed."

  redirect_to projects_path
end
```

Here you call the `destroy` method ❶ on the `@project` object you get back from your `find` call. No validations are run here, 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. Once 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.

With this last action in place, your newest feature should pass when you run `bin/rspec spec/features/deleting_projects_spec.rb`:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
1 example, 0 failures
```

Great, let's see if everything else is running with `bin/rspec`:

```
6 examples, 0 failures
```

Great! Let's commit that:

```
git add .
git commit -m "Implement delete functionality for projects"
git push
```

Done! Now you have the full support for CRUD operations in pour `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 `bin/rails server` and try to navigate to http://localhost:3000/projects/not-here. You'll see the exception shown in Figure 4.1

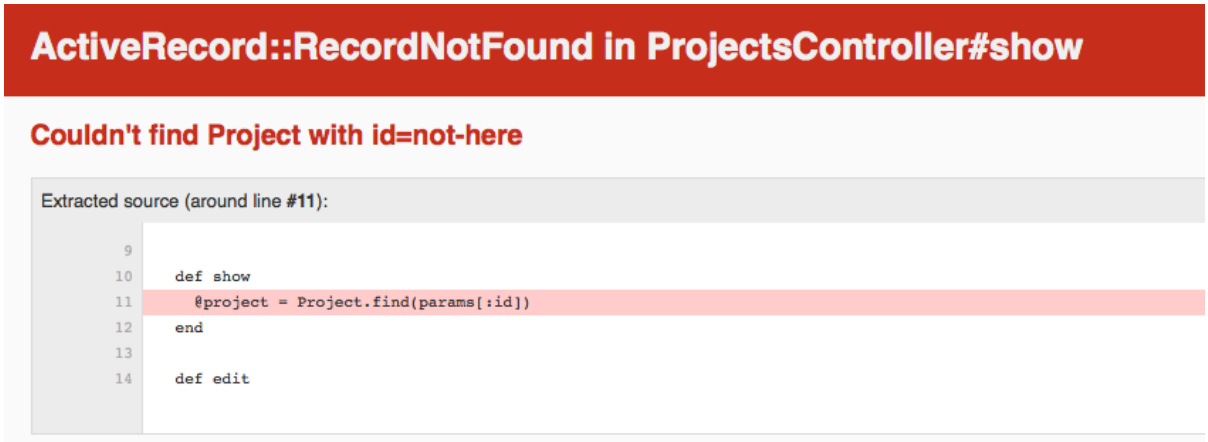

Figure 4.1 ActiveRecord::RecordNotFound exception

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `ActiveRecord::RecordNotFound` exception is Rails's way of displaying exceptions in development mode. Underneath 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 that Rails will show the complete exception information when it runs in the `development` environment.

If you were running in the *production* environment, you would see a different error because `consider_all_requests_local` in config/environments/production.rb is set to false. Let's try to reproduce this error now.

4.4.1 Visualizing the error

Stop any Rails server that is currently running and run these commands to start a new one in production mode:

```
bin/rake assets:precompile
bin/rake db:migrate RAILS_ENV=production
```

In order for the Rails production environment to work correctly, you must first compile the assets for the project using the `assets:precompile` rake task. This will go through all the assets of the application and compile them into their CSS and JS counterparts and then place these new files into public/assets so that they can be served by the web server that is running Rails. Not too relevant to what you're doing now, but necessary so that you can see what the `production` environment will do.

On the second line you must specify the RAILS_ENV environment variable to tell Rails you want to run the migrations on your production database. By default in Rails, the development and production databases are kept separate so you don't make the mistake of working with production data and deleting something you shouldn't when you're working in the development environment. This problem is also solved by placing the production version of the code on a different server from the one you're developing on. You only have to run the migration command when migrations need to be run, not every time you need to start your server.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

In the production environment, the Rails server is configured to not serve static assets itself. Instead, it will rely on the host server to serve the assets out of the public directory. In order for assets to be served correctly in the production environment while running a `bin/rails server` session, you will need to go into the config/environments/production.rb file and change this line:

```
config.serve_static_assets = false
```

To this:

```
config.serve_static_assets = true
```

This then tells Rails that you want to serve static assets from the public directory using Rails itself. Next, start the server running using the production environment by using this command:

```
bin/rails s -e production
```

You pass the `-e production` option to the `bin/rails server` command, which tells Rails to boot the server using the production environment.

Next, navigate to http://localhost:3000/project/not-here. When you do this, you will get the standard Rails 404 page (Figure 4.2), which, to your users, is unhelpful.

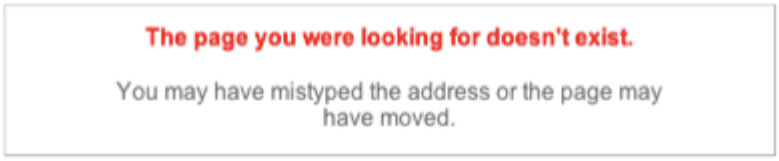

Figure 4.2 Page does not exist error

It's not the page that's gone missing, but rather the *resource* we're looking for isn't found. If users see this error, they'll probably have to click the Back button and then refresh the page. You could give users a much better experience by

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

dealing with the error message yourself and redirecting them back to the home page.

Before we move on, we want to un-do the stuff we just did: it's not a good idea to have Rails serve assets in production, and it's also not good to check compiled assets into source control. To back out your changes, just do this:

```
$ git add .
$ git reset --hard
```

Yay for git! This just adds our changes to the index, then resets our index to the last commit. Easy!

4.4.2 Handling the ActiveRecord::NotFound exception

To do so, you can rescue the exception and, rather than letting Rails render a 404 page, you 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 controller test rather than a feature test, because viewing projects that aren’t there is something a user *can* do, but not something they *should* do. Plus, it’s easier.

The file for this controller test, spec/controllers/projects_controller_spec.rb, was automatically generated when you ran the controller generator because you have the `rspec-rails` gem in your Gemfile. [4] Open this controller spec file and take a look. It should look like the following listing.

Footnote 4 The `rspec-rails` gem automatically generates the file using a Railtie, the code of which can be found at https://github.com/rspec/rspec-rails/blob/master/lib/rspec-rails.rb.

Listing 4.14 spec/controllers/projects_controller_spec.rb

```
require 'spec_helper'
describe ProjectsController do
end
```

In this controller spec, you want to test that you get 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, you put the following code inside the `describe` block:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
it "displays an error for a missing project" do
  get :show, id: "not-here"
  expect(response).to redirect_to(projects_path)
  message = "The project you were looking for could not be found."
  expect(flash[:alert]).to eql(message)
end
```

The first line *inside* this RSpec test—more commonly called an *example*—tells RSpec to make a GET request to the `show` action for the `ProjectsController`. How does it know which controller should receive the GET request? RSpec infers it from the class used for the `describe` block.

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 the `flash[:alert]` to contain a useful message explaining the redirection to the index action.

To run this spec, use the `bin/rspec spec/controllers/projects_controller_spec.rb` command. When this runs, you'll see this error:

```
Failure/Error: get :show, id: "not-here"
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 `bin/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 underneath the last action in the controller but before the end of the class.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 4.15 find_project method inside ProjectsController

```
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
```

This method has the `private` method before it so the controller doesn't respond to this method as an action. You already had `private` declared for the `project_params` method.[5]To call this method before every action, use the `before_action` method. Place these lines directly under the `class ProjectsController` definition:

Footnote 5 The lines for the flash[:alert] are separated into two lines to accommodate the page width of this book. You can put it on one line if you like. We won't yell at you.

```
before_action :set_project, only: [:show,
                                   :edit,
                                   :update,
                                   :destroy]
```

What does all this mean? Let's start with the `before_action`. `before_action`s 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 you want the `before_action` to run for. The `:except` option is the opposite of the `:only` option, specifying the actions you do not want the `before_action` to run for. The `before_action` calls the `find_project` method before the specified actions, setting up the `@project` variable for you. This means you can remove the following line from four of your actions: `show`, `edit`, `update`, and `destroy`:

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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

By doing this, you make the `show` and `edit` actions empty. If you remove these actions and run `bin/rspec` again, all the scenarios will still pass. 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, it's best to leave these in the controller so anyone who reads the code knows that the controller can respond to these actions. You can also remove the first line of `update` and `destroy` actions as well.

Back to the spec now: if you run `bin/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 `bin/rspec`. You should see this:

```
7 examples, 0 failures
```

Red-Green-Refactor! Now with that out of the way, let's commit and push that!

```
git add .
git commit -m "Redirect the users back to the projects
               page if they try going to a project that doesn't 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 `bin/rspec` and if something is broken that you've written a test for, you'll know about it. Now 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 whole 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 is still working exactly as it was 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 that 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 we do in the next chapter. We also cover nested routing and association methods for models. Let's go!

Index Terms

ActiveRecord::Base#update
before_action
before_action, :only option
link_to, :method
link_to, data: {confirm: "..."}

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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, defining routing for `Ticket` resources by creating a CRUD interface for them, scoped underneath the projects resource that you just created. In this chapter you'll see just 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 underneath the projects, you first develop the Capybara features and then implement the code required to make them pass. Nesting one resource under another involves additional routing, working with associations in Active Record, and using more `before_action`s. 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 title 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 the data entered is valid, just as you did with the `Project` model. This new form will look like Figure 5.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

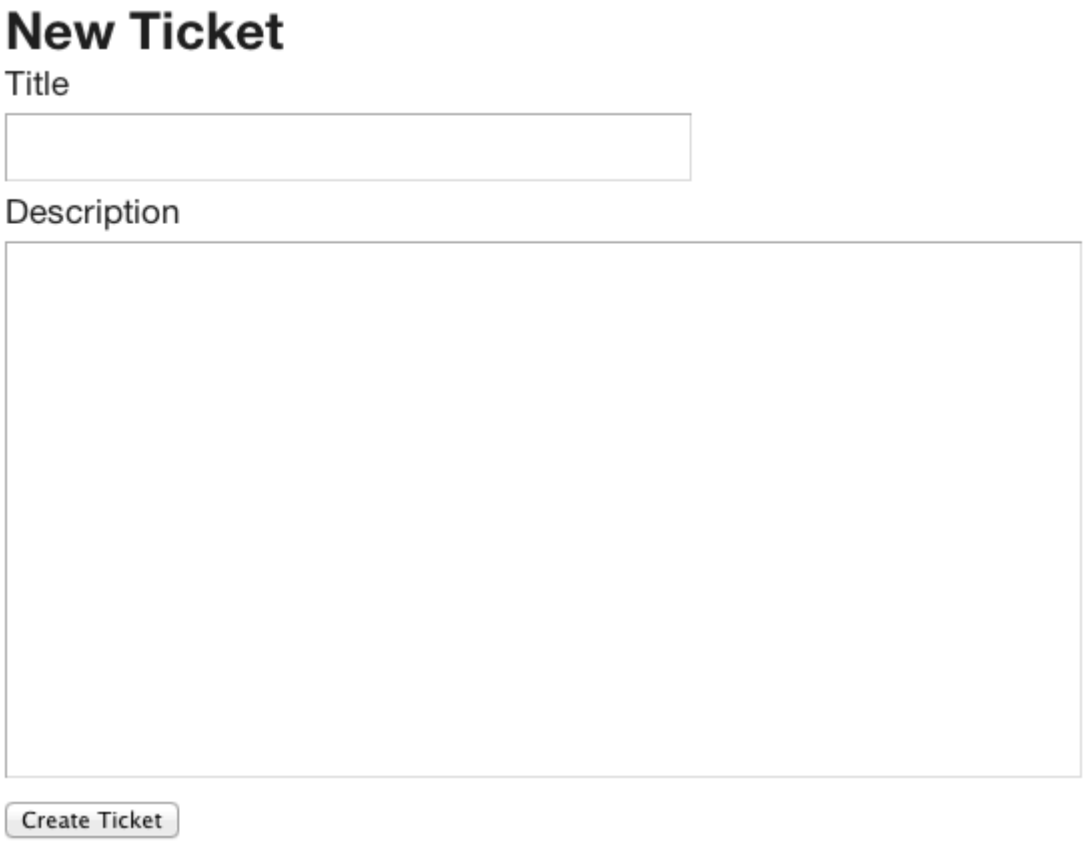

Figure 5.1 Form for creating new tickets

Start by using the code from the following listing in a new file.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 5.1 spec/features/creating_tickets_spec.rb

```
require 'spec_helper'

feature "Creating Tickets" do
  before do
    FactoryGirl.create(:project, name: "Internet Explorer")

    visit '/'
    click_link "Internet Explorer"
    click_link "New Ticket"
  end

  scenario "Creating a ticket" do
    fill_in "Title", 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 "Creating a ticket without valid attributes fails" do
    click_button "Create Ticket"

    expect(page).to have_content("Ticket has not been created.")
    expect(page).to have_content("Title can't be blank")
    expect(page).to have_content("Description can't be blank")
  end
end
```

You've seen `before` before, but this time, we use it to create the parent object. Our children need a parent, so it makes sense to build one before every test.

We want to make sure to test the basic functionality of creating a ticket. It's pretty straightforward: fill in the attributes, click the button, and make sure it works!

We should also test the failure case. Because we need to have a title and description, a failing case is easy: just click the "Create Ticket" button prematurely!

When you run this new feature using the `bin/rspec spec/features/creating_tickets_spec.rb` command, your `before` block fails, as shown in the following listing.

```
Failure/Error: click_link "New Ticket"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Capybara::ElementNotFound:
  Unable to find link "New Ticket"
```

You need to add this "New Ticket" link to the app/views/projects/show.html.erb template so that this line in the test will work. Add it underneath the "Delete Project" link.

```
<%= link_to "New Ticket", new_project_ticket_path(@project) %>
```

This helper is called a *nested routing helper*, and is just 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 takes one argument always, the `@project` object for which you want to create a new ticket: the object that 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

access it within the scope of a project too. Just as the `resources :projects` method gave you helpers to use in controllers and views, this nested one gives you the helpers (where `id` represents the identifier of a resource) shown in table 5.1.

Table 5.1 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`

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. The `:project_id` symbol here would normally be replaced by the project ID as well as the `:id` symbol, which would be replaced by a ticket's ID.

In the left column are the routes that can be accessed, and in the right, the routing helper methods you can use to access them. Let's make use of them by first creating your `TicketsController`.

5.1.2 Creating a tickets controller

Because you defined this route in your routes file, Capybara can now click the link in your feature and proceed before complaining about the missing `TicketsController`, spitting out an error followed by a stack trace:

```
Failure/Error: click_link "New Ticket"
 ActionController::RoutingError:
   uninitialized constant TicketsController
```

Some guides may have you generate the model before you generate the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

controller, but the order in which you create them is not important. When writing tests, you just following the bouncing ball, and if the test tells you it can't find a controller, then you generate the controller it's looking for next. Later, when you inevitably receive an error that it cannot find the `Ticket` model, as you did for the `Project` model, you generate that too. This is often referred to as *top-down design*[1].

Footnote 1 http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design

To generate this controller and fix this `uninitialized constant` error, use this command:

```
$ bin/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. Open app/controllers/tickets_controller.rb and add the `new` action, shown in the following listing.

Listing 5.2 new action, TicketsController

```
def new
  @ticket = @project.tickets.build
end
```

The `build` method simply instantiates a new record for the `tickets` association on the `@project` object, working in much the same way as the following code would:

```
Ticket.new(project_id: @project.id)
```

Of course, you haven't yet done anything to define the `@project` variable in `TicketsController`, so it would be `nil`. You should define the variable using a `before_action`, just as you did in the `ProjectsController`. Put

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the following line just under the `class` definition in app/controllers/tickets_controller.rb.

```
before_action :set_project
```

You don't restrict the `before_action` here because you want to have a `@project` to work with in all actions because the tickets resource is only accessible through a project. Underneath the `new` action, define the method that the `before_action` uses:

```
private
  def set_project
    @project = Project.find(params[:project_id])
  end
```

Where does `params[:project_id]` come from? It's made available through the wonders of Rails's routing, just as `params[:id]` was back in `ProjectsController`. It's called `project_id` instead of `id` because you could (and later will) have a route that you want to pass through an ID for a ticket as well as a project route, and the ticket id would be `params[:id]`. Now how about that `tickets` method on your `@project` object? Let's make sure it doesn't already exist by running `bin/rspec spec/features/creating_tickets_spec.rb`:

```
Failure/Error: click_link "New Ticket"
     NoMethodError:
       undefined method `tickets' for #<Project:0x007fc6222100a8>
```

No Rails magic here yet. We'll be getting into some of that with Active Record associations right now.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

5.1.3 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`, which you can use as follows inside app/models/project.rb:

```
has_many :tickets
```

As mentioned before, this defines the `tickets` method you need as well as the association. With the code `has_many` method called inside the `Project` model, you will now be able to get to all the tickets for any given project by simply 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, such as the `build` method, which you are 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 re-running `bin/rspec spec/features/creating_tickets_spec.rb`, you will get this:

```
Failure/Error: click_link "New Ticket"
NameError:
  uninitialized constant Project::Ticket (NameError)
```

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 objects of the `Ticket` model. This model is currently missing; hence, the error. You can add this model now with the following command

```
$ bin/rails generate model ticket title:string description:text \
  project:references
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This command is written as above so it will fit into the book. You can put it all on one line and it will work the same. The backslash at the end is just to tell your prompt not to run the command just yet, as there's more arguments to go.

The `project:references` part defines an *integer* column for the *tickets* table called `project_id` in the migration that creates the `tickets` table. It will also define 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:

```
class CreateTickets < ActiveRecord::Migration
  def change
    create_table :tickets do |t|
      t.string :title
      t.text :description
      t.references :project, index: true

      t.timestamps
    end
  end
end
```

As you can see here, this migration will also add an index on the table which will make lookups for tickets relating to a specific project faster than if there was no index.

The `project_id` column represents the project this ticket links to and is called a `foreign key`. The purpose of this field is to simply store the primary key of the project that 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".

You should now run the migration with `bin/rake db:migrate` and load the updated schema into your test database by running `bin/rake db:test:prepare`.

The `bin/rake 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 `bin/rake db:schema:load` task if you wish, which is better than running all the migrations on a large project again![2]

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 2 Large projects can have hundreds of migrations, which may not run due to changes in the system over time. It's best to just use the `bin/rake db:schema:load`.

The `bin/rake db:test:prepare` task performs a very similar task to `bin/rake db:schema:load`. It loads this schema into the test database, making the fields that were just made available on the development database by running the migration also now available on the test database.

Now when you run `bin/rspec spec/features/creating_tickets_spec.rb`, you're told the new template is missing:

```
Failure/Error: click_link "New Ticket"
Missing template tickets/new, application/new
  with {handlers: [:erb, :builder, :coffee],
        formats: [:html],
        locale: [:en]}.

  Searched in:
    * ".../ticketee/app/views"
```

A file seems to be missing! You must create this file in order to continue.

5.1.4 Creating tickets within a project

Create the file at app/views/tickets/new.html.erb and put the following inside:

```
<h2>New Ticket</h2>

<%= render "form" %>
```

This template renders a `form` partial, which will be relative to the current folder and will be placed at app/views/tickets/_form.html.erb, using the code from Listing 5.3

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 5.3 app/views/tickets/_form.html.erb

```
<%= form_for [@project, @ticket] do |f| %>                    ❶
  <% if @ticket.errors.any? %>
   <div id="error_explanation">
     <h2><%= pluralize(@ticket.errors.count, "error") %>
     prohibited this ticket from being saved:</h2>

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

  <p>
    <%= f.label :title %><br>
    <%= f.text_field :title %>
  </p>
  <p>
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </p>
  <%= f.submit %>
<% end %>
```

Note that `form_for` is passed an array of objects ❶ rather than simply:

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

This code indicates to `form_for` that you want the form to post to the nested route you're using. For the `new` action, this generate 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*[3].

Footnote 3 A great description of which can be found at http://ryanbigg.com/2012/03/polymorphic-routes

When you run `bin/rspec spec/features/creating_tickets_spec.rb` again, you're told the `create` action is missing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_button "Create Ticket"
AbstractController::ActionNotFound:
 The action 'create' could not be found for TicketsController
```

To define this action, put it directly underneath the `new` action inside `TicketsController` but before the `private` method. You'll also add the appropriate strong parameters helper method right below the `private`, as shown in the following listing:

Listing 5.4 create action, TicketsController

```
def create
  @ticket = @project.tickets.build(ticket_params)
  if @ticket.save
    flash[:notice] = "Ticket has been created."
    redirect_to [@project, @ticket]
  else
    flash[:alert] = "Ticket has not been created."
    render "new"
  end
end
private
  def ticket_params
    params.require(:ticket).permit(:title, :description)
  end
```

Inside this action, you use `redirect_to` and specify an `Array` here—the same array you used in the `form_for` 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 determine 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_for`, `@ticket` was new, so the route happened to be /projects/1/tickets.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You could have been explicit and specifically used `project_ticket_path` in the action, but using an array is DRYer.

When you run `bin/rspec spec/features/creating_tickets_spec.rb`, both scenarios report the same error:

```
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.

5.1.5 Finding tickets scoped by project

Currently, both of our scenarios are failing. :'(

Of course, now you must define the `show` action for your controller, but you can anticipate that you'll need to find a ticket for the `edit`, `update`, and `destroy` actions too and pre-empt those errors. You can also make this a `before_action`, just as you did in the `ProjectsController` with the `set_project` method. You define this finder underneath 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 scope of the project. Put the `before_action` at the top of your class, just underneath the one to find the project:

```
before_action :set_project
before_action :set_ticket, only: [:show, :edit, :update, :destroy]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The sequence here is important because you want to find the `@project` before you go looking for tickets for it. Then create the view template for this action at app/views/tickets/show.html.erb using this code:

```
<div id='ticket'>
  <h2><%= @ticket.title %></h2>
  <%= simple_format(@ticket.description) %>
</div>
```

The new method, `simple_format`, converts the line breaks[4] entered into the description field into HTML break tags (`<br>`) so that the description renders exactly how the user intends it to.

Footnote 4 Line breaks are represented as `\n` and `\r\n` in strings in Ruby rather than as visible line breaks.

Based solely on the changes that you've made so far, your first scenario should be passing. Let's see with a quick run of `bin/rspec spec/features/creating_tickets_spec.rb`:

```
Failure/Error: expect(page).to have_content("Ticket has not ...
   expected there to be content "Ticket has not been created." in [text]
...

2 examples, 1 failure
```

This means that you've got the first scenario under control and that 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.

5.1.6 Ticket validations

The second scenario fails because the `@ticket` that it saves is valid, at least according to your tests in their current state:

```
expected there to be content "Ticket has not been created" in "[text]"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You need to ensure that when somebody enters a ticket into the application, the `title` and `description` attributes are filled in. To do this, define the following validations inside the `Ticket` model:

Listing 5.5 app/models/ticket.rb

```
validates :title, presence: true
validates :description, presence: true
```

NOTE **Validating two fields using one line**

You could also validate the presence of both of these fields using a single line:

```
validates :title, :description, presence: true
```

It's just the author's 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 `bin/rspec 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! Let’s add this scenario to the spec/features/creating_tickets_spec.rb file:

```
scenario "Description must be longer than 10 characters" do
  fill_in "Title", with: "Non-standards compliance"
  fill_in "Description", with: "it sucks"
  click_button "Create Ticket"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  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 }
```

If you go into `bin/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)
```

That is the precise error message you are looking for in your new scenario, and if you're getting it on the console then that must mean that would appear in the app too. Find out by running `bin/rspec spec/features/creating_tickets_spec.rb` again.

```
3 examples, 0 failures
```

Alright, that one's passing now. Excellent! You should ensure that the rest of the project still works by running `bin/rspec` again. You will see this output:

```
12 examples, 0 failures, 2 pending
```

There looks to be two pending specs here, one located in spec/helpers/tickets_helper_spec.rb and the other in spec/models/ticket_spec.rb.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You don't need these right now, and so you should delete these two files. Once you've done that, re-running `bin/rspec` will output a lovely green result:

```
10 examples, 0 failures
```

Great! Everything's still working. 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 use the `show` action to create the functionality to view them individually.

When displaying a list of projects, you use the `index` action of the `ProjectsController`. For tickets, however, you use the `show` action because this page is currently not being used for anything else in particular. To test it, put a new feature at spec/features/viewing_tickets_spec.rb using the code from the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 5.6 spec/features/viewing_tickets_spec.rb

```
require 'spec_helper'

feature "Viewing tickets" do
  before do
    textmate_2 = FactoryGirl.create(:project,
                                    name: "TextMate 2")

    FactoryGirl.create(:ticket,
            project: textmate_2,
            title: "Make it shiny!",
            description: "Gradients! Starbursts! Oh my!")

    internet_explorer = FactoryGirl.create(:project,
                                           name: "Internet Explorer")
    FactoryGirl.create(:ticket,
            project: internet_explorer,
            title: "Standards compliance",
            description: "Isn't a joke.")

    visit '/'
  end

  scenario "Viewing tickets for a given project" do
    click_link "TextMate 2"

    expect(page).to have_content("Make it shiny!")
    expect(page).to_not have_content("Standards compliance")

    click_link "Make it shiny!"                                    (1)
    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! We'll go through it piece by piece in just a moment. First, let's examine the `within` (1) usage in your scenario. Rather than checking the entire page for content, this step checks the specific element using Cascading Style Sheets (CSS) selectors. The `#ticket` prefix finds all elements with an ID of `ticket` that contain an `h2` element with the content you specified. This content

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

should appear inside the specified tag only when you're on the ticket page, so this is a great way to make sure that you're on the right page and that the page is displaying relevant information.

When you run this spec with `bin/rspec spec/features/viewing_tickets_spec.rb` you'll see that it cannot find the ticket factory:

```
Failure/Error: Factory(:ticket,
ArgumentError:
  Factory not registered: ticket
```

Just like before when the project factory wasn't registered, you're going to need to create the ticket factory now. What this should do is create an example ticket with a valid title and description. To do this, create a new file called spec/factories/ticket_factory.rb and put the content from the following listing in it:

Listing 5.7 spec/factories/ticket_factory.rb

```
FactoryGirl.define do
  factory :ticket do
    title "Example ticket"
    description "An example ticket, nothing more"
  end
end
```

With the ticket factory now defined, the `before` block of this spec should now run all the way through when you run `bin/rspec spec/features/viewing_tickets_spec.rb`, and you'll see this error:

```
Failure/Error: expect(page).to have_content("Make it shiny!")
   expected there to be text "Make it shiny!" in ...
```

The spec is now attempting to see the ticket's title on the page, but it cannot see it at the moment, because you're not displaying a list of tickets on the project `show` template yet.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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, as shown in the following listing:

Listing 5.8 app/views/projects/show.html.erb

```
<ul id='tickets'>
  <% @project.tickets.each do |ticket| %>
    <li>
      #<%= ticket.id %> -
      <%= link_to ticket.title, [@project, ticket] %>
    </li>
  <% end %>
</ul>
```

TIP **Be careful about variables here**

If you use a `@ticket` variable in place of the `ticket` variable in the `link_to`'s second argument, it will be `nil`. You haven't initialized the `@ticket` variable at this point, and uninitialized instance variables are `nil` by default. If `@ticket` rather than the correct `ticket` is passed in here, 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 inside the block. The code inside this block runs for every single ticket. When you run `bin/rspec spec/features/viewing_tickets_spec.rb` it will pass 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 `bin/rspec`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You should see that everything is green.

```
11 examples, 0 failures
```

Fantastic! Push!

```
$ git add .
$ git commit -m "Implement displaying a list of relevant tickets.

You can see them on projects, and view a particular ticket."
$ git push
```

Now you can see tickets just for a particular project, but what happens when a project is deleted? The tickets for that project would not be 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 as 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, and you can do that by using the `:dependent` option on the `has_many` association for tickets defined in your `Project` model.

This option has three choices that all act slightly differently from each other. 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 will call `destroy` on them, then calls any destroy callbacks (such as any `has_many`'s in the `Ticket` model, which also have the `dependent` option)[5] the `ticket` objects have on them, any destroy callbacks for those objects, and so on. The

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

problem is that if you have a large number of tickets, `destroy` is called on each one, which will be slow.

Footnote 5 Or any callback defined with `after_destroy` or `before_destroy`.

The solution is the second value for this option:

```
has_many :tickets, dependent: :delete_all
```

This simply 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 have callbacks that 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`.

Finally, if you just 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 then 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

tracking application and instead of projects and tickets you had users and tasks. If you delete a user, you may want to reassign rather than delete the tasks associated with that user, in which case you'd use the `dependent: :nullify` option instead.

In your projects and tickets scenario, though, you 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:

```
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.

We haven't written any tests for this behavior, as it's quite simple and we'd basically be testing that we changed one tiny option. This is more of an internal implementation detail than it is customer-facing, and we're writing `feature` tests right now, not `model` tests. Let's check that we didn't break existing tests, commit, and push!

```
$ bin/rspec
$ git add .
$ git commit -m "Cull tickets when project gets destroyed"
$ git push
```

Let's look at how to edit the tickets in your application next.

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`.

The next feature you're going to implement is the ability to edit tickets. 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 hit an update button and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

expect to see two things: a message indicating that the ticket was updated successfully, and the modified data for that ticket.

With that in mind, you can write this feature using the code in the following listing, placing the code in a file at spec/features/editing_tickets_spec.rb.

Listing 5.9 spec/features/editing_tickets_spec.rb

```
require 'spec_helper'

feature "Editing tickets" do
  let!(:project) { FactoryGirl.create(:project) }
  let!(:ticket) { FactoryGirl.create(:ticket, project: project) }

  before do
    visit '/'
    click_link project.name
    click_link ticket.title
    click_link "Edit Ticket"
  end

  scenario "Updating a ticket" do
    fill_in "Title", 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!")
    end

    expect(page).to_not have_content ticket.title
  end

  scenario "Updating a ticket with invalid information" do
    fill_in "Title", 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! Its lesser brother, `let` (no bang), defines a new method with the same name as the symbol passed in, and that new method then evaluates the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

content of the block whenever that method is called. If you were using `let` (no bang), you would need to call the `project` method somewhere in the test, either in the `before` or `scenario` blocks, in order for it to create a project.

By using `let!`, it will automatically call the block and therefore the project and ticket that these `let!` definitions reference will be created. You can then refer to them later on in your tests, as you do in the `before` block underneath.

Just like we made a spec for creating a ticket, we need one for updating, as well. Updating specs are always complex, because you need to already have an existing object that's built properly, and then change it.

It's also a good idea to test the failure case. It looks pretty similar to the update case, but rather than try to factor out all the commonalities, we repeat ourselves. Some duplication in tests is okay; if it makes the test easier to follow, it's worth a litte bit of repetition.

When you run this feature using `bin/rspec spec/features/editing_tickets_spec.rb`, the first three lines in the `before` run just fine, but the fourth fails:

```
Failure/Error: click_link "Edit Ticket"
Capybara::ElementNotFound:
  Unable to find link "Edit Ticket"
```

To fix this, add the "Edit Ticket" link to the `TicketsController`'s `show` template, as that's where you've navigated to in your feature. Put it on the line underneath the `<h2>` tag in app/views/tickets/show.html.erb.

```
<%= link_to "Edit Ticket", [:edit, @project, @ticket] %>
```

Here is yet another use of the `Array` argument passed to the `link_to` method, but rather than passing all Active Record objects, you pass a `Symbol` first. Rails, yet again, works out from this `Array` what route you wish to follow. Rails interprets this array to mean the `edit_project_ticket_path` method, which is called like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
edit_project_ticket_path(@project, @ticket)
```

Now that you have an "Edit Ticket" link, you need to add the `edit` action to the `TicketsController`, as that will be the next thing to error when you run `bin/rspec spec/features/editing_tickets_spec.rb`:

```
Failure/Error: click_link "Edit Ticket"
AbstractController::ActionNotFound:
  The action 'edit' could not be found for TicketsController
```

5.3.1 Adding the edit action

The next logical step is to define the `edit` action in your `TicketsController`, which you can leave empty because the `set_ticket` before action does all the heavy lifting for you. Define the action to be a blank method, just so other people reading this code know that this controller deals with this action:

```
def edit
end
```

Again, you're defining the action here so that anybody coming through and reading your `TicketsController` class knows that this controller responds to this action. It's the first place people will go to determine what the controller does, because it is the *controller*. While the blank action is not necessary, it *is* good form, so that those coming to this project after you know that the action has been implemented.

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:

```
<h2>Editing a ticket in <%= @project.name %></h2>

<%= render "form" %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Here you re-use the `form` partial you created for the `new` action, which is handy. The `form_for` knows which action to go to. If you run the feature command here, you're told the update action is missing:

```
Failure/Error: click_button "Update Ticket"
AbstractController::ActionNotFound:
  The action 'update' could not be found for TicketsController
```

5.3.2 Adding the update action

You should now define the update action in your `TicketsController`, as shown in the following listing:

Listing 5.10 update action, TicketsController

```
def update
  if @ticket.update(ticket_params)
    flash[:notice] = "Ticket has been updated."

    redirect_to [@project, @ticket]
  else
    flash[:alert] = "Ticket has not been updated."

    render action: "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 inside the "Editing Tickets" feature will now pass when you run `bin/rspec spec/features/editing_tickets_spec.rb`:

```
2 examples, 0 failures
```

Now check to see if everything works with a quick run of `bin/rspec`:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
13 examples, 0 failures
```

Great! Let’s commit and push that.

```
$ git add .
$ git commit -m "We can now edit tickets."
$ 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_attributes`. You’ve got one more part to go: deletion.

5.4 Deleting tickets

We now reach the final story for this nested resource, the deletion of 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.

Let’s use the code from the following listing to write a new feature in spec/features/deleting_tickets_spec.rb.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 5.11 spec/features/deleting_tickets_spec.rb

```
require 'spec_helper'

feature 'Deleting tickets' do
  let!(:project) { FactoryGirl.create(:project) }
  let!(:ticket) { FactoryGirl.create(:ticket, project: project) }

  before do
    visit '/'
    click_link project.name
    click_link ticket.title
  end

  scenario "Deleting a ticket" do
    click_link "Delete Ticket"

    expect(page).to have_content("Ticket has been deleted.")
    expect(page.current_url).to eq(project_url(project))
  end
end
```

When you run this spec using `bin/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:

```
Failure/Error: click_link "Delete Ticket"
Capybara::ElementNotFound:
  Unable to find link "Delete Ticket"
```

You can add the "Delete Ticket" link to the app/views/tickets/show.html.erb file just under the "Edit Ticket" link, exactly like what you did with projects:

```
<%= link_to "Delete Ticket", [@project, @ticket], method: :delete,
  data: { confirm: "Are you sure you want to delete this ticket?"} %>
```

The `method: :delete` is specified again, turning the request into one headed for the `destroy` action in the controller. Without this `:method` option,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you'd be off to the `show` action because the `link_to` method defaults to the `GET` method. Upon running `bin/rspec spec/features/deleting_tickets_spec.rb` again, you're told a `destroy` action is missing:

```
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 5.12 destroy action, TicketsController

```
def destroy
  @ticket.destroy
  flash[:notice] = "Ticket has been deleted."

  redirect_to @project
end
```

With that done, your feature should now pass when `bin/rspec spec/features/deleting_tickets_spec.rb` is run again:

```
1 example, 0 failures
```

Yet again, check to see that everything is still going as well as it should by using `bin/rspec`. You've not changed much, and so it's likely that things will still be working. You should see this output:

```
14 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Commit and push!

```
$ git add .
$ git commit -m "Implement deleting tickets feature"
$ git push
```

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 to create new `Ticket` records that are scoped to a project.

In the next chapter, you will learn how to let users sign up and sign in to your application. You also implement a basic authorization for actions such as creating a project.

Index Terms

ActiveRecord, associations, find
ActiveRecord, build
ActiveRecord::Base, has_many
bin/rake db:schema:load
bin/rake db:test:prepare
foreign key
form_for, Array usage
simple_format

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

6 Authentication

You've created two resources for your Ticketee application: projects and tickets. 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. You don't want absolutely everybody creating or modifying projects, so you'll learn to restrict project creation to a certain subset of users. To round out the chapter, you'll create another CRUD interface, this time for the users resource, but with a twist.

There are lots of different ways that you can implement this functionality, and a balance needs to be struck. Authentication (as you'll see) deals with cryptography, and we don't know about you, but we're not experts in the field. So writing 100% of the code yourself probably isn't the best thing for you or your users. There are gems (such as Devise) that do everything you could ever want (and more), but we don't feel comfortable outsourcing such an important part of an application to a gem. Furthermore, you can often spend as much time customizing the more full-stack gems as you would writing authentication in the first place.

In this chapter, you'll build authentication with `ActiveModel::HasSecurePassword`. This comes with Rails (as part of `ActiveModel`) and is very, very minimal: it only handles the cryptography portion of authentication. The rest is up to you. That way, you can be safe, yet get exactly what you want: a win/win.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

6.1 Authentication basics

Let's get down to brass tacks. As you've seen, every time you have a concept in your project, you make a model for it: projects and tickets were both made so far. Because you have the concept of a user, let's make a model to represent it. You'll use the generator to make it:

```
$ bin/rails g model user name email password_digest
$ bin/rake db:migrate db:test:prepare
```

Easy! You may be wondering why you didn't include a type, like `email:string`. If you leave off the type, Rails will assume that you want a `string` type. In this case, this saved you typing `:string` three times! Every little bit counts.

So why `password_digest`? It's important that you don't save your users' passwords directly, because if someone gains access to your database, they could read all of your users' passwords. We hope we don't have to tell you that's bad. What to do?

Cryptographers call text that you want to encrypt *plaintext*. If you encrypt it using an encryption algorithm (a *cypher*), you get *cyphertext* out. You could do this, but then it'd be possible for someone to take your cyphertext and decrypt it and get plaintext back out. You need something else.

You don't care what the user types as their password; you just care that when they type it again, you can recognize that it's the same thing they typed when they signed up. It'd be nice if your transformation was irreversible as well, so the original passwords can't be recovered; cryptographers call this a *one-way function*. You'll use one of those one-way functions called a *hash function*. You'll save this *hash* (the output of a hash function) in your database instead. So, a user gives you their password, you hash it, and then you save the hash. When they log in next time, they give you their password, you hash it, and then you compare it to the hash you saved. That way, you know they typed the same password, but you don't save it to the database. Neat!

Rails comes with a helper method to help you properly hash your passwords: it's called `has_secure_password`. Let's get that going. First, you need to add a gem to your Gemfile. Uncomment the appropriate line so it looks like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
# To use ActiveModel has_secure_password
gem 'bcrypt-ruby', '~> 3.0.0'
```

Then you need to rebundle:

```
$ bin/bundle
```

Before you implement it in your model, let's write some tests to verify that everything is working as you'd expect. Normally we wouldn't consider testing something that is a feature of Rails; but authentication is an incredibly important part of your application, and you want to be sure it's working properly. We always write more tests than we might otherwise write for parts of our apps that are complex or important, like authentication, authorization, administrator functionality, payment systems, or anything that interacts with an external service.

spec/models/user_spec.rb already exists. Replace its content with this:

```
require 'spec_helper'

describe User do
  describe "passwords" do
    it "needs a password and confirmation to save" do
      u = User.new(name: "steve")

      u.save
      expect(u).to_not be_valid

      u.password = "password"
      u.password_confirmation = ""
      u.save
      expect(u).to_not be_valid

      u.password_confirmation = "password"
      u.save
      expect(u).to be_valid
    end

    it "needs password and confirmation to match" do
      u = User.create(
            name: "steve",
            password: "hunter2",
            password_confirmation: "hunter")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      expect(u).to_not be_valid
    end
  end

  describe "authentication" do
    let(:user) { User.create(
      name: "steve",
      password: "hunter2",
      password_confirmation: "hunter2") }

    it "authenticates with a correct password" do
      expect(user.authenticate("hunter2")).to be
    end

    it "does not authenticate with an incorrect password" do
      expect(user.authenticate("hunter1")).to_not be
    end
  end
end
```

Usually we'd write tests in small batches and work on them one by one. But because all this functionality is provided by `has_secure_password`, and we're just double-checking that the application uses it correctly, we expect all these tests to pass, so we wrote them all at once.

The first test is kinda long, so let's break it down. First you want to make sure a user without a username or password is invalid, so you assert that it's not valid immediately. Then you want to make sure you require a confirmation as well as a password; so you add a password with no confirmation, save, and assert that that's not valid either. Finally, you change that confirmation to be the same as the password and then assert that it's valid.

Some people prefer to write this as three separate tests. We don't think that's a bad thing to do, but we like the process here, where the user gets progressively more and more correct.

If you run the spec with `bin/rspec spec/models/user_spec.rb`, you should see this:

```
4 examples, 4 failures
```

You didn't write any code to implement this functionality, so it's expected to fail. Let's get them passing!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The usage of `has_secure_password` is simple. You need to add one line to your model—open app/models/user.rb, and do this:

```
class User > ActiveRecord::Base
  has_secure_password
end
```

That's it! As we mentioned before, Rails provides this functionality for you, but you have to tell Rails you want to use it. This is how you turn it on. Now run the tests:

```
$ bin/rspec spec/models/user_spec.rb
....

Finished in 0.06553 seconds
4 examples, 0 failures

Randomized with seed 28571

$ bin/rspec
..................

Finished in 1.03 seconds
18 examples, 0 failures

Randomized with seed 17704
```

Awesome! The specs tell the story: you have `password` and `password_confirmation` fields on your user, they have to match, and you get an `authenticate` method to check if the password is correct. Easy.

With your `User` model in shape, let's commit:

```
$ git add .
$ git commit -m "Make a User model with password."
$ git push
```

Well done. Now that you have a useful `User`, let's let your users sign up and create a `User`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

6.2 Signing up

Having users is nice, but if they can't sign up via your app, it's not much use. Let's write that interface. As always, you'll start with a feature test. You need a test to demonstrate that a user can sign up for an account in your system. Here it is, for spec/features/signing_up_spec.rb:

```
require 'spec_helper'

feature 'Signing up' do
  scenario 'Successful sign up' do
    visit '/'

    click_link 'Sign up'
    fill_in "Email", with: "user@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
```

You'll get this failure:

```
Failure/Error: click_link 'Sign up'
Capybara::ElementNotFound:
  Unable to find link "Sign up"
```

This should start to feel familiar by now. Do you know what you need to do? That's right: add a link! This time, it'll go in app/views/layouts/application.html.erb. Why? Because you want your users to be able to sign up on every page, and the application layout is the basis of all of your HTML:

```
<nav>
  <%= link_to "Sign up", new_user_path %>
</nav>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This fails because you haven't defined a route yet. Here's the failure:

```
Failure/Error: visit '/'
ActionView::Template::Error:
  undefined local variable or method `new_user_path'
    for #<#<Class:0x007fe08a7b70d8>:0x007fe08a7bd7f8>
```

As you saw before, routes are the ways you connect URLs to your controllers, models, and views. You know you need a route because you're making a new link and because it can't find a method ending in `_path`, which is one of the two endings that route helpers have. Open config/routes.rb, and add this:

```
resources :users
```

Now, because you defined your route, you get a different failure:

```
Failure/Error: click_link 'Sign up'
ActionController::RoutingError:
  uninitialized constant UsersController
```

The error says it can't find a controller, so let's make one for your `User`s. You could use the generator you've used before, but here's another little trick. What actions will you need? Well, you'll need `new` to show your form, you'll need `create` to make the user, and you'll need `show` to show your user's profile page. Turns out that the controller generator has a shortcut to create not just the controller, but also some actions. Try this on for size:

```
$ bin/rails g controller users new create show
create  app/controllers/users_controller.rb
 route  get "users/show"
 route  get "users/create"
 route  get "users/new"
invoke  erb
create    app/views/users
create    app/views/users/new.html.erb
create    app/views/users/create.html.erb
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
create    app/views/users/show.html.erb
...
```

Tons of stuff! A lot of this you've seen before, but now it's also making files in app/views. Before, you were just making the controllers, but now you're making views, routes, and actions as well. This is the advantage and disadvantage of certain generators: they can help you make stuff you don't want to make yourself, but they can also make too much. For example, run `bin/rspec` again:

```
Finished in 1.2 seconds
26 examples, 1 failure, 4 pending
```

Four pending? That's because the generator generated sample test files for your actions. It also generated a view for your `create` action, which you won't use; and finally, it added three routes you won't use. Let's get rid of all that:

```
$ rm -rf spec/views/users
$ rm spec/controllers/users_controller_spec.rb
$ rm -rf spec/helpers
$ rm app/views/users/create.html.erb
```

Don't forget to remove the three `get` routes from config/routes.rb as well. With all that done, you should see `18 examples, 1 failure` when you run `bin/rspec`.

OK, let's implement this spec. The error you're currently getting is

```
Failure/Error: fill_in "Email", with: "user@example.com"
Capybara::ElementNotFound:
  Unable to find field "Email"
```

This makes sense because you haven't put anything in your `new` view yet. Put this code in app/views/users/new.html.erb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<h1>Sign Up</h1>

<%= render "form" %>
```

That gives you the structure; now let's make the form. This goes in app/views/users/_form.html.erb:

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

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

  <p>
    <%= f.label :name, "Username" %><br>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :email %><br>
    <%= f.text_field :email %>
  </p>
  <p>
    <%= f.label :password %><br>
    <%= f.password_field :password %>
  </p>
  <p>
    <%= f.label :password_confirmation %><br>
    <%= f.password_field :password_confirmation %>
  </p>
  <%= f.submit "Sign up" %>
<% end %>
```

If you rerun your spec with `bin/rspec spec/features/signing_up_spec.rb`, you'll get this:

```
Failure/Error: click_link 'Sign up'
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
ActionView::Template::Error:
  First argument in form cannot contain nil or be empty
```

Ruby initializes instance variables to `nil` on first access, and because you have no controller action, you never set a `@user`. Let's fix that by adding a controller action in app/controllers/users_controller.rb:

```
def new
  @user = User.new
end
```

Rerun your tests with `bin/rspec spec/features/signing_up_spec.rb`:

```
Missing template users/create, application/create with...
```

You've seen this before, and it usually means you need to make a new view file. But this is your `create` action: you want to redirect elsewhere rather than render a page. So let's open app/controllers/users_controller.rb and code this up:

```
def create
  @user = User.new(user_params)

  if @user.save
    flash[:notice] = "You have signed up successfully."
    redirect_to projects_path
  else
    render :new
  end
end
```

And down under `private`, add this:

```
private
  def user_params
    params.require(:user).permit(:name,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
                                  :password,
                                  :password_confirmation)
  end
```

You've seen this before: it looks very similar to what you did for `Projects` and `Tickets`. Rrerun the test:

```
$ bin/rspec spec/features/signing_up_spec.rb
.

Finished in 0.31269 seconds
1 example, 0 failures

$ bin/rspec
..................

Finished in 1.06 seconds
19 examples, 0 failures
```

Bam! You're all green! Let's commit and push:

```
  $ git add .
  $ git commit -m "Implemented sign up."
  $ git push
```

Because you don't have a lot of functionality for your `Users`, you use `redirect_to projects_path`, because that will be more useful. Eventually, users will want to see a profile page or edit their information. You're getting pretty good at CRUD by now: you have `show`, `edit`, and `update` working. Here are some specs for you. Put them in spec/features/user_profile_spec.rb:

```
require 'spec_helper'

feature "Profile page" do
  scenario "viewing" do
    user = FactoryGirl.create(:user)

    visit user_path(user)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    expect(page).to have_content(user.name)
    expect(page).to have_content(user.email)
  end
end

feature "Editing Users" do
  scenario "Updating a project" do
    user = FactoryGirl.create(:user)

    visit user_path(user)
    click_link "Edit Profile"

    fill_in "Username", with: "new_username"
    click_button "Update Profile"

    expect(page).to have_content("Profile has been updated.")
  end
end
```

Give it a run with `bin/rspec spec/features/user_profile_spec.rb`. Once that's all working, your `bin/rspec` should say this:

```
.....................

Finished in 0.75249 seconds
10 examples, 0 failures
```

There is one little tricky part you might need help on. If you have a problem getting the last spec to pass, check this out. Otherwise, just read up, and see how your solution compares to ours.

The issue is with an error that looks like this:

```
1) Editing Users Updating a project
   Failure/Error: click_button "Update Profile"
   Capybara::ElementNotFound:
     Unable to find button "Update Profile"
```

If you've followed all the steps, you may be really confused. Let's check out where that button's code comes from, in app/views/users/_form.html.erb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= f.submit "Sign up" %>
```

Oh no! You hard-coded the button's text as `"Sign up"` in an earlier test. What to do? Well, if you haven't saved the user yet, you know they're signing up. If you have saved the user, you know they're editing their profile. Active Record provides a convenient method to do just this. Here are our modifications, which get you on to the next failure:

```
<% if @user.new_record? %>
  <%= f.submit "Sign up" %>
<% else %>
  <%= f.submit "Update Profile" %>
<% end %>
```

When you have 21 examples passing, it's time to commit:

```
$ git add .
$ git commit -m "Implement profile pages"
$ git push
```

That wasn't so bad, was it? You're done with authentication. It's time for authorization: now that users can sign up, let's allow them to sign in. No good having an account if you can't use it!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

6.3 Signing in

Now that users can sign up, it's time to let them sign in. Let's talk about *sessions*. HTTP is a stateless protocol, which means you have to resend your credentials on each request. This is for scalability purposes; if your servers had to keep track of clients, it would get complex. Clients know who they are and can tell you on subsequent requests. We call that set of requests a session, which Rails implements by default through cookies. The plan is this: when someone logs in with the correct credentials, you'll tell them to set a cookie with a certain value. On the next request, they will send that cookie along, and you use that information to authenticate them.

Let's get started. As always, you'll write a spec first. Put this in spec/features/sign_in_spec.rb:

```
require 'spec_helper'

feature "signing in." do
  scenario 'Signing in via form' do
    user = FactoryGirl.create(:user)

    visit '/'
    click_link 'Sign in'
    fill_in 'Name', with: user.name
    fill_in 'Password', with: user.password
    click_button "Sign in"

    expect(page).to have_content("Signed in successfully.")
  end
end
```

If you run this spec, you'll get this failure:

```
Failure/Error: click_link 'Sign in'
Capybara::ElementNotFound:
  Unable to find link "Sign in"
```

This should be getting easy at this point. You have a missing link, so let's add one. Put this new Sign In link in app/views/layouts/application.html.erb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<nav>
  <ul>
    <li><%= link_to "Sign up", new_user_path %></li>
    <li><%= link_to "Sign in", signin_path %></li>
  </ul>
</nav>
```

Now that you have more than one navigation link, it seems appropriate to put the links in a list. If you rerun your specs, you've gotten farther:

```
ActionView::Template::Error:
  undefined local variable or method `signin_path'
```

You know what this means: you need a new route. Put this in config/routes.rb:

```
get "/signin", to: "sessions#new"
```

This makes a new route. If you remember from the output of `bin/rake routes`, routes consist of a `Prefix`, `Verb`, `URI Pattern`, and `Controller#Action`. All four parts are in this declaration: the `get` is the `Verb`, the `"/signin"` is the `URI Pattern`, and the `to:` is the `Controller#Action`. The `Prefix` is inferred from the URL and is `"signin"`. Rerunning your specs, you get a new error:

```
ActionController::RoutingError:
  uninitialized constant SessionsController
```

Easy enough; you don't have a controller. Make one in app/controllers/sessions_controller.rb:

```
class SessionsController < ApplicationController
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

So easy. Now run the spec, `bin/rspec specs/features/sign_in_spec.rb`:

```
Failure/Error: click_link 'Sign in'
AbstractController::ActionNotFound:
  The action 'new' could not be found for SessionsController
```

Go ahead and make one, and then rerun the specs:

```
ActionView::MissingTemplate:
  Missing template sessions/new
```

Make a new file, app/views/sessions/new.html.erb, and put this in it:

```
<h1>Sign in</h1>
<%= form_for :signin, method: "POST" do |f| %>
  <p>
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.label :password %><br />
    <%= f.password_field :password %>
  </p>

  <%= f.submit "Sign in" %>
<% end %>
```

`:signin` makes your form link to `signin_path`, and `method:` changes the form to have `method="POST"` as well. `POST` is better than `GET` because you're possibly making a new session here, and therefore it's not idempotent or safe. Running your specs brings a new failure:

```
ActionController::RoutingError:
  No route matches [POST] "/signin"
```

This is true. Off to the routes file (config/routes.rb):

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
post "/signin", to: "sessions#create"
```

This is similar to your last map, except that the action is `create` and the method is `post`. Running your tests, you get another red:

```
AbstractController::ActionNotFound:
  The action 'create' could not be found for SessionsController
```

You know how to make a new action, right? Put this in it:

```
def create
  user = User.where(:name => params[:signin][:name]).first


  if user && user.authenticate(params[:signin][:password])

    session[:user_id] = user.id
    flash[:notice] = "Signed in successfully."

    redirect_to root_url
  else
    flash[:error] = "Sorry."
    render :new
  end
end
```

A few things are going on here. First, you use an Active Record method called `where` to find all the `User`s with a `name` that you got in your parameters. That gives you a `Relation`, so you call `first` on it to retrieve the first row that matches the query. You may be wondering why you're accessing `params` directly, when before (in chapter 3) you filtered them through strong parameters. The reason is that you're not doing any kind of mass assignment here; you're mapping one part of your `params` hash to one argument. Therefore, there's no problem. Second, there's this awkward `user &&` thing. If you don't have a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`User` record that matches your query, the `user` variable will be `nil`. So you make a check that it's not `nil` before you attempt to call `authenticate` to stop those pesky `NoMethodError on nil`'s.

Running your tests with `bin/rspec spec/features/sign_in_spec.rb`, you should see this:

```
.

Finished in 0.28416 seconds
1 example, 0 failures
```

Great! Let's run all the tests and then commit:

```
$ bin/rspec
$ git add .
$ git commit -m "Implemented signing in."
$ git push
```

That's all there is to it. With authentication finished: you can sign up and sign in.

6.4 Linking tickets to users

Now that users can sign up and sign in to your application, it's time to link a ticket with the user who created it automatically so it can be indicated to everyone who created a ticket. When you're done with this feature, a little indication will appear under a ticket's title in the app/views/tickets/show.html.erb view, as shown in figure 6.3.

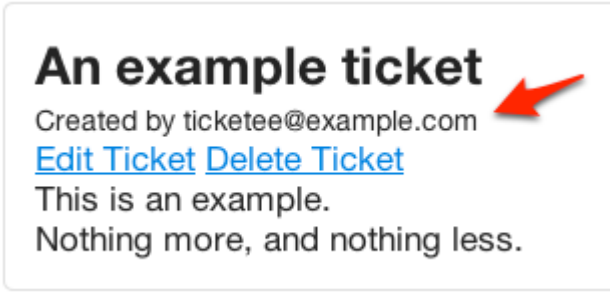

Figure 6.1 Ticket author indicator

That part is easy: you need a "Created by $USER" message displayed on the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

ticket page. The background work needed to make it function properly is a little more difficult, but you'll get through it.

You can test for this functionality by amending the "Creating a ticket" scenario in spec/features/creating_tickets_spec.rb to have the following final lines:

```
within "#ticket #author" do
  expect(page).to have_content("Created by sample@example.com")
end
```

When you run the feature using `bin/rspec spec/features/creating_tickets_spec.rb`, it fails on this new requirement like this:

```
Failure/Error: within("#ticket #author") do
Capybara::ElementNotFound:
 Unable to find css "#ticket #author"
```

This particular problem is easy to fix. All you need to do is add a new element to the app/views/tickets/show.html.erb page in the `#ticket` element that has an ID of `"author"`. Under the ticket title on this page, do exactly that:

```
<span id='author'></span>
```

That's all the scenario is asking for. Running `bin/rspec spec/features/creating_tickets_spec.rb` again should provide a better error message that has a *much* more complicated fix:

```
expected there to be content "Created by sample@example.com" in ""
```

That's better. Now the test is telling you the author can't be seen in that element on the page. You need to make sure the user is signed in before they can create a ticket; otherwise, you won't know who to make the owner of that ticket. When

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

users go to a project page and click the New Ticket link, they should be redirected to the sign-in page and asked to sign in, if they aren't already. Once they're signed in, they should be able to create the ticket. Change the `before` block of the `feature` in spec/features/creating_tickets_spec.rb to ensure that this process happens, using the code from the following listing.

Listing 6.1 spec/features/creating_tickets_spec.rb

```
before do
  project = FactoryGirl.create(:project)
  user = FactoryGirl.create(:user)                                    ❶


  visit '/'
  click_link project.name
  click_link "New Ticket"
  message = "You need to sign in or sign up before continuing."
  expect(page).to have_content(message)

  fill_in "Name", with: user.name
  fill_in "Password", with: user.password
  click_button "Sign in"

  click_link project.name
  click_link "New Ticket"
end
```

With these changes, you've added a call to the user factory ❶ under the one to the project factory, so a user exists. When the New Ticket link is clicked, you're checking that the user sees the message "You need to sign in or sign up before continuing" and that they can then sign in using the username and password for the user that was created. Once they're signed in, they should then see New Ticket on the page within a `<h2>` tag, indicating that they're back on the new ticket page.

What next? Well, run `bin/rspec spec/features/creating_tickets_spec.rb`, and follow the bouncing ball.

```
expected there to be text "You need to sign in or sign up...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The line that checks for the text “You need to sign in or sign up before continuing” fails because you’re not ensuring that the user is signed in before the `new` action in the `TicketsController`. To do so, you’ll write a `before_action`. This filter ensures that a user is signed in before they can perform a specific action. If the user isn’t signed in, they will be redirected to the sign in page and given the “You need to sign in or sign up before continuing” message. Here’s the implementation, in `TicketsController`:

```
before_action :require_signin!, except: [:show, :index]
```

This line ensures that users are authenticated before they go to any action in the controller that isn’t `index` or `show`, including the `new` and `create` actions.

By ensuring that a user is authenticated before they can create a ticket, there will always be a `current_user` object around that you can use to link a ticket to a user. When you run `bin/rspec spec/features/creating_tickets_spec.rb`, it now gets through the `before` block in this feature and proceeds to create a new ticket, failing again with this error:

```
Failure/Error: click_link "New Ticket"
NameError:
  undefined local variable or method `require_signin!' for ...
```

You didn’t write a method to make sure you’re signed in, you just called it. Ha! Thank goodness for tests! You could write the method in `TicketsController`, but this will be useful for things other than `Ticket`s. Put this code in app/controllers/application_controller.rb:

```
private
  def require_signin!
    if current_user.nil?
      flash[:error] =
      "You need to sign in or sign up before continuing."
      redirect_to signin_url
    end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  end
  helper_method :require_signin!

  def current_user
    @current_user ||= User.find(session[:user_id]) if session[:user_id]
  end
  helper_method :current_user
```

Now, rerunning the spec, you get a new error:

```
  expected there to be content "Created by ticketee@example.com" ...
```

This is happening because you aren't displaying any information about who the ticket belongs to. In order for that to happen, first you'll need to assign a user to the ticket when it has been created. Let's do that right now.

6.5 Attributing tickets to users

To link tickets to specific users, you alter the `build` line in your `create` action in `TicketsController` from this line

```
@ticket = @project.tickets.build(ticket_params)
```

to these two lines:

```
@ticket = @project.tickets.build(ticket_params)
@ticket.user = current_user
```

The `user=` setter method is used here because you shouldn't be able to assign the user via a parameter. You always want it to be the `current_user`, so you set it that way.

When you run `bin/rspec spec/features/creating_tickets_spec.rb`, it complains because this `user=` method isn't available on the `Ticket` object:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_button "Create Ticket"
NoMethodError:
undefined method `user=' for #<Ticket:...>
```

The `user=` method should be defined on the `Ticket` model by using a `belongs_to` association, because a ticket will always *belong to* a user. Do this by putting this line under the `belongs_to :project` line in app/models/ticket.rb:

```
belongs_to :user
```

By defining a `belongs_to :user` association on this model, calls to the `user` method on any `Ticket` object will attempt to look up a user record in the `users` table with an `id` field that matches the `user_id` field on the `tickets` table. You don't have a `user_id` field at the moment, so you'll need to add this for it to work. Run the migration generator again like this:

```
$ bin/rails g migration add_user_id_to_tickets user:references
```

Based solely on how you wrote the name of this migration, along with the `user_id:integer` at the end of it, Rails will understand that you want to add a particular column to the `tickets` table and that the column will be called `user_id` and will be of the integer type.

If you open the new migration file (it's the last one in the db/migrate directory), you'll see exactly what this migration generator output.

Listing 6.2 db/migrate/[timestamp]_add_user_id_to_tickets

```
class AddUserIdToTickets < ActiveRecord::Migration
  def change
    add_reference :tickets, :user, index: true
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You can close this file and then run the migration with `bin/rake db:migrate`, and prepare the test database by using `bin/rake db:test:prepare`.

SIDEBAR

Bash migrate alias

If you're using bash as a shell (which is probably the case if you're on a UNIX operating system), you can add an alias to ~/.bashrc or ~/.bash_profile to do both of these steps for you rather than having to type them out all the time:

```
alias migrate='bin/rake db:migrate && bin/rake db:test:prepare'
```

Then type `source ~/.bashrc` or `source ~/.bash_profile`, and the alias will be available to you in your current terminal window. It'll also be available in new terminal windows even if you didn't use `source`, because this file is processed every time a new bash session is started. If you don't like typing `source`, then `. ~/.bashrc` or `. ~/.bash_profile` will do.

You'll now be able to run `migrate` rather than `rake db:migrate; rake db test:prepare` each time you want to migrate both the development and test databases.

SIDEBAR

Make sure to run db:test:prepare

If you don't prepare the test database, the following error occurs when you run the feature:

```
And I press "Create Ticket"
undefined method 'name' for nil:NilClass (ActionView::Template::Error)
```

Watch out for that one.

Let's rerun `bin/rspec spec/features/creating_tickets_spec.rb` and see where it stands now:

```
Failure/Error:
  expect(page).to have_content("Created by sample@example.com")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  expected there to be content "Created by sample@example.com" ...
```

This is now failing because the author isn't displayed in the `span#author` element in app/views/tickets/show.html.erb. Change this element to this:

```
<span id='author'>Created by <%= @ticket.user.email %></span>
```

This small change should make the feature pass, because all the pieces are in place. Run `bin/rspec spec/features/creating_tickets_spec.rb`, and you should see this output:

```
3 examples, 0 failures
```

The "Created by $USER" line is appearing on the ticket page. Good job! Run `bin/rspec` as usual to ensure that you haven't broken anything:

```
Failed examples:

rspec ./spec/features/deleting_tickets_spec.rb:13
rspec ./spec/features/editing_tickets_spec.rb:14
rspec ./spec/features/editing_tickets_spec.rb:24
rspec ./spec/features/signing_in_spec.rb:8
rspec ./spec/features/viewing_tickets_spec.rb:20
```

Oops, it looks like you did. If you didn't have these tests in place, you wouldn't have known about this breakage unless you tested the application manually or guessed (or somehow knew) that your changes would break the application in this way. Let's see if you can fix it.

6.6 You broke something!

Luckily, the failed tests have the same error. The common error is this:

```
Failure/Error: click_link "Make it shiny!"
ActionView::Template::Error:
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  undefined method `email' for nil:NilClass
  # ./app/views/tickets/show.html.erb:7...
```

Whatever is causing this error is on line 7 of app/views/tickets/show.html.erb. It's this line:

```
<span id='author'>Created by <%= @ticket.user.email %></span>
```

Aha! The error is `undefined method 'email' for nil:NilClass`, and the only place you call `email` on this line is on the `user` object from `@ticket`, so you can determine that `user` must be `nil`. But why? Let's look at how to set up the data in the spec/features/viewing_tickets_spec.rb feature, as shown in the following listing.

Listing 6.3 spec/features/viewing_tickets_spec.rb

```
FactoryGirl.create(:ticket,
        project: textmate_2,
        title: "Make it shiny!",
        description: "Gradients! Starbursts! Oh my!")
```

The issue is happening because no user is assigned to the ticket in this setup. You should rewrite this feature to make it create a ticket and link it to a specific user.

6.7 Fixing the Viewing Tickets feature

The first step is to create a user you can link the ticket to, so change the ticket setup in the `before` block to this:

```
user = FactoryGirl.create(:user)
ticket = FactoryGirl.create(:ticket,
        project: textmate_2,
        title: "Make it shiny!",
        description: "Gradients! Starbursts! Oh my!")
ticket.update(user: user)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `update` method updates an attribute on an object and then saves the object. It's really a shortcut for this:

```
ticket.user = user
ticket.save
```

It's used in this situation to link the ticket to the user created, and should cause the scenario to pass. Run `bin/rspec spec/features/viewing_tickets_spec.rb`. You should see this output:

```
1 example, 0 failures
```

That looks like the feature is now unbroken, which is perfect. Let's fix up the other two using the same technique, beginning with the Editing Tickets feature.

6.8 Fixing the Editing Tickets feature

You can fix the Editing Tickets feature using a methodology similar to what you used for the Viewing Tickets feature. Change the `let!` blocks at the top of this feature to the code shown in the next listing.

Listing 6.4 spec/features/editing_tickets_spec.rb

```
let!(:project) { FactoryGirl.create(:project) }
let!(:user) { FactoryGirl.create(:user) }
let!(:ticket) do
  ticket = FactoryGirl.create(:ticket, project: project)
  ticket.update(user: user)
  ticket
end
```

In the `let!(:user)` block, it now reads as `FactoryGirl.create(:user)`, which is your usual `User` factory.

When you run the feature—unlike the Viewing Tickets feature—it doesn't pass,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

complaining that it can't find the field called `Title`. Uh oh:

```
Failure/Error: fill_in "Title", :with => ""
Capybara::ElementNotFound:
  Unable to find field "Title"
```

Back in the `TicketsController`, you restricted some of the actions by using the `before_action`:

```
before_action :require_signin!, except: [:index, :show]
```

This `before_action` restricts access to the `edit` and `create` actions for people who aren't signed in. To be able to edit a ticket, a user must first be signed in. In this feature, then, you should sign in as the user you create so you can edit this ticket. Change the first line of `before` to sign in as that user:

```
  sign_in_as!(user)
```

When you run this feature using `bin/rspec spec/features/editing_tickets_spec.rb`, you'll see that this new `sign_in_as!` method is undefined:

```
Failure/Error: sign_in_as!(user)
NoMethodError:
  undefined method `sign_in_as!' for ...
```

It's OK that it can't find this helper, because it's something you should define yourself. Create a new file at spec/support/authentication_helpers.rb, and use the code from the next listing to define the helper.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 6.5 spec/support/authentication_helpers.rb

```
module AuthenticationHelpers
  def sign_in_as!(user)
    visit '/signin'
    fill_in "Name", with: user.name
    fill_in "Password", with: user.password
    click_button 'Sign in'
    expect(page).to have_content("Signed in successfully.")
  end
end

RSpec.configure do |c|
  c.include AuthenticationHelpers, type: :feature
end
```

In this new file, you define an `AuthenticationHelpers` module that contains the single `sign_in_as!` method, which goes through the motions of signing in as a particular user. The final three lines includes this module into all specs in the spec/features directory, which means the spec/features/editing_tickets_spec.rb spec should now have access to this helper.

If that's the case, running `bin/rspec spec/features/editing_tickets_spec.rb` should pass:

```
2 examples, 0 failures
```

One more feature with this error to go: Deleting Tickets.

6.9 Fixing the Deleting Tickets feature

To fix the Deleting Tickets feature, replace these lines from spec/features/editing_tickets_spec.rb

```
let!(:project) { FactoryGirl.create(:project) }
let!(:user) { FactoryGirl.create(:user) }
let!(:ticket) do
  ticket = FactoryGirl.create(:ticket, project: project)
  ticket.update(user: user)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  ticket
end
```

with the following lines:

```
let!(:project) { FactoryGirl.create(:project) }
let!(:ticket) { FactoryGirl.create(:ticket, project: project) }
```

You also need to add the `sign_in_as!` helper to the beginning of the `before` block:

```
before do
  sign_in_as!(user)
  ...
end
```

The entire spec/features/deleting_tickets_spec.rb file should now look like this:

```
require 'spec_helper'

feature 'Deleting tickets' do
  let!(:project) { FactoryGirl.create(:project) }
  let!(:user) { FactoryGirl.create(:user) }
  let!(:ticket) {
    FactoryGirl.create(:ticket, project: project, user: user) }

  before do
    sign_in_as!(user)

    visit '/'
    click_link project.name
    click_link ticket.title
  end

  scenario "Deleting a ticket" do
    click_link "Delete Ticket"

    expect(page).to have_content("Ticket has been deleted.")
    expect(page.current_url).to eq(project_url(project))
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you run this file using `bin/rspec spec/features/deleting_tickets_spec.rb`, it passes:

```
1 example, 0 failures
```

There! You've fixed the last of the broken features with the same error … or at least you hope so. What happens when you run `bin/rpec`? This should be the result:

```
....................

Finished in 1.55 seconds
22 examples, 0 failures
```

All passing! Great—go ahead and commit:

```
$ git add .
$ git commit -m "Associate users to tickets
               when tickets are created"
$ git push
```

In this section, you've added a feature to associate users with the tickets they create. A user must be signed in before they can create a ticket, so the application can associate the two together. In implementing this feature, you broke a couple of seemingly unrelated features of your application that depended on the code being as it was. Shoring up these features did take a bit of time to get right, but it's worth it in order to have that feature safety net. This is the major reason why testing the features of your application the way you've been doing currently is a very good thing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

6.10 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 built a simple authentication system on top of `ActiveModel::SecurePassword`. You also tested the functionality 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 `bin/rails s`, visit http://localhost:3000, and play around, to get an idea of how it's looking right now. This is one of the interesting side effects of TDD; you've built an entire application, and you haven't even looked at it in the browser!

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.

Index Terms

before_action, :except option
update

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

7 Basic access control

As your application now stands, anybody, whether they're signed in or not, can create new projects. As you did for the actions in the `TicketsController`, you must restrict access to the actions in the `ProjectsController`. The twist here is that you'll allow 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 is not to be confused with *authentication*, which you implemented in chapter 6. Authentication is the process users go through to confirm their identity, while authorization is the process used by the system to determine which users should have access to certain things.

Later on this chapter, 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, then you would need to restrict access on a per-controller basis.

7.1 Turning users into admins

To restrict the creation of projects to admins, you're going to add an `admin` attribute to `User` objects. Only users who have this `admin` attribute set to `true` will be able to create projects. To test this, you must first alter the existing `before` in spec/features/creating_projects_spec.rb and insert a line to sign in as an admin user at the beginning of the `before` block:

```
before do
  sign_in_as!(FactoryGirl.create(:admin_user))
  ...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This line uses a `admin_user` factory that we haven't defined yet.. This factory will simply create a different "breed" of user; one that will eventually have permission to do everything in the system. To create this new factory, open spec/factories/user_factory.rb and make it look like this:

```
FactoryGirl.define do
  factory :user do
    name "username"
    email "sample@example.com"
    password "hunter2"
    password_confirmation "hunter2"

    factory :admin_user do
      admin true
    end
  end
end
```

This looks just like our `user` factory from Chapter 6, but it's inside of it! It also adds another attribute default, this time that `admin` should be true.

When you run `bin/rspec spec/features/creating_projects_spec.rb` you'll see that there is no `admin=` method defined for a `User` object:

```
  Failure/Error: sign_in_as!(FactoryGirl.create(:admin_user))
 NoMethodError:
   undefined method `admin=' for #<User:0x007fb194180a90>
```

Therefore the next logical step is to define a field in the database so that the attribute setter method is available.

7.1.1 Adding the admin field to the users table

You can generate a migration to add the `admin` field by running this command:

```
$ bin/rails g migration add_admin_to_users admin:boolean
```

You want to modify this migration so that when users are created, the `admin`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

field is set to `false` rather than defaulting to `nil`. Even though `nil` is 'falsey' in Ruby, it's more clear to make it explicitly false. `nil` means that they have no admin information, but they do: they're not an admin. Better to be explicit about things. To do this, open the freshly generated migration and change this line:

```
add_column :users, :admin, :boolean
```

to this:

```
add_column :users, :admin, :boolean, default: false
```

When you pass in the `:default` option here, the `admin` field defaults to `false`, ensuring that users aren't accidentally created as admins.

TIP **Jumping the gun**

If you jumped the gun and ran `bin/rake 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. Just run `bin/rake db:rollback` and that will 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 `bin/rake db:migrate` again!

Run `bin/rake db:migrate` and `bin/rake db:test:prepare` now so that the migration adds the `admin` field to the users table in both the development and test databases. When you run `bin/rspec spec/features/creating_projects_spec.rb` it will now run fully:

```
2 examples, 0 failures
```

Great! With the `admin_user` factory defined, you can go about using this to test the restriction of the acts of creating, updating, and destroying projects to only those users who are admins.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Before we do that, let's commit everything:

```
$ git add .
$ git commit -m "Added admin flag to user."
$ git push
```

7.2 Restricting actions to admins only

For this step, you will implement a `before_action` that checks not only whether the user is signed in but also whether the user is an admin. If the user isn't an admin, they shouldn't be able to perform any of the protected actions on that controller.

7.2.1 Testing admin-only controller access

Before you write this `before_action`, you will write a controller spec rather than an integration spec to test it. Integration tests are great for defining a set of actions that a user can perform in your system, but controller specs are much better for quickly testing singular points, such as whether or not a user can go to a specific action in the 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.

You want to ensure that all visits to the `new`, `create`, `edit`, `update`, and `destroy` actions are done by admins and are inaccessible to other users.

Create spec/controllers/projects_controller_spec.rb and add a set up for a new user:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.1 spec/controllers/projects_controller_spec.rb

```
require 'spec_helper'

describe ProjectsController do
  let(:user) { FactoryGirl.create(:user) }

  context "standard users" do
    before do
      sign_in(user)
    end

    it "cannot access the new action" do
      get :new

      expect(response).to redirect_to('/')
      expect(flash[:alert]).to eql("You must be an admin to do that.")
    end
  end
end
```

Here you use set up a user by using the `user` factory, and then sign in as this new user inside a `before` block using a new helper called `sign_in`, which we will write.

Inside the test, you are testing that when a user makes a `GET` request to the `new` action of the `ProjectsController` that the response should redirect them to the root path of the application and should also set a `flash[:alert]` message to "You must be an admin to do that."

When you run this test using `bin/rspec spec/controllers/projects_controller_spec.rb` it will fail like this:

```
Failure/Error: sign_in(user)
 NoMethodError:
   undefined method `sign_in' for ...
```

The `sign_in` method isn't yet available. Just like with the `email_spec` helper methods in the previous chapter, you'll need to include your own authorization helpers into RSpec's configuration too. Open

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

spec/support/authentication_helpers.rb, and add this chunk of code:

```
module AuthHelpers
  def sign_in(user)
    session[:user_id] = user.id
  end
end

RSpec.configure do |c|
  c.include AuthHelpers, type: :controller
end
```

By passing the `:type` option along here, the methods for `AuthHelpers` will only be made available on the controller specs inside the application, as that's the only place that you'll be needing these methods.

TIP **Including for models or views only**

You can specify `type: :model` as a filter if you want to include a module only in your model specs. If you ever write any view specs, you can use `type: :view` to include this module only in the view specs. Similarly, you can use `:features` for specs that reside in spec/features.

Going back to your spec, you make a request on the third line to the `new` action in the controller. The `before_action` that you haven't yet implemented should catch the request before it gets to the action; it won't execute the request but instead redirects the user to `root_path` and shows a `flash[:alert]` telling the user that they "must be an admin to do that".

If you run this spec with `bin/rspec spec/controllers/projects_controller_spec.rb`, it fails as you expect:

```
Failure/Error: response.should redirect_to('/')
  Expected response to be a <redirect>, but was <200>]>
```

This error message tells you that although you expected to be redirected, the response was actually a `200` response, indicating a successful response. This isn't

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

what you want; you want a redirect! 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 `ProjectsController`. This method checks whether a user is an admin, and if not, displays the “You must be an admin to do that” message and then redirects the user back to the root path.

First define this new method inside app/controllers/projects_controller.rb by placing the code from the following listing underneath the `destroy` method in `ProjectsController`:

Listing 7.2 app/controllers/projects_controller.rb

```
private

  def authorize_admin!
    require_signin!

    unless current_user.admin?
      flash[:alert] = "You must be an admin to do that."
      redirect_to root_path
    end
  end
```

This method uses the `require_signin!` method (that we wrote earlier) to ensure that the user is signed in. This was previously used in Chapter 6 as a `before_action`, but it can be called on its own like this. To refresh your memory, if the user isn't signed in when the `require_signin!` method is called, they are asked to sign in.

If the user *is signed in* but isn't an admin they are then shown the “You must be an admin to do that” message and redirected to the root path when `authorize_admin!` is called.

To call this method, call `before_action` at the top of your `ProjectsController` ensuring that it's above the `set_project` filter, like this:

```
before_action :authorize_admin!, except: [:index, :show]
before_action :set_project, only: [:show, :edit, :update, :destroy]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

With that in place, you can re-run the spec `bin/rspec spec/controllers/projects_controller_spec.rb`, which should now pass as the `before_action` is in place and will re-route non-admin users correctly:

```
1 examples, 0 failures
```

Great, now we know this is working for the new action, but does it work for `create`, `edit`, `update`, and `destroy`? You can replace the `"cannot access the new action"` example you just wrote with the code from the following listing.

Listing 7.3 spec/controllers/projects_controller_spec.rb

```
{ new: :get,
  create: :post,
  edit: :get,
  update: :put,
  destroy: :delete }.each do |action, method|
  it "cannot access the #{action} action" do
    sign_in(user)
    send(method, action, :id => FactoryGirl.create(:project))
    expect(response).to redirect_to(root_path)
    expect(flash[:alert]).to eql("You must be an admin to do that.")
  end
end
```

In this example, you call out to a currently-undefined `project` local variable or method in order to pass the `:id` parameter to some of these requests. The reason for doing this is because requests to the `edit`, `update` and `destroy` actions require an `id` parameter so that they can be correctly routed to.

The first line of this example has a hash. Its keys are all of the actions that you want to make sure are behind a protective `before_action`, and the values are the HTTP methods that you use to make those actions happen. You use each action here to give your examples dynamic names, and you use all of them them further down when you use the `send` method. The `send` method allows you to dynamically call methods and pass arguments to them. It's used here because for each key-value pair of the hash, the `action`[1] and `method` change. You pass in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the `:id` parameter because, without it, the controller can't route to the `edit`, `update`, or `destroy` actions. The `new` and `create` actions ignore this parameter.

Footnote 1 The action variable is a frozen string (because it's a block parameter), so you need to duplicate the object because Rails forces the encoding on it to be UTF-8.

The remainder of this spec is unchanged, and when you run `bin/rspec spec/controllers/projects_controller_spec.rb`, you should see all five examples passing:

```
5 examples, 0 failures
```

Now's a good time to ensure you haven't broken anything, so let's run `bin/rake spec`.

```
Failed examples:

rspec ./spec/features/deleting_projects_spec.rb:4
rspec ./spec/features/editing_projects_spec.rb:11
rspec ./spec/features/editing_projects_spec.rb:17
```

Oops. Three tests are broken. They failed because, for these features, you're not signing in as an admin user--in fact, as *any* user! -- which is now required for performing the actions in the scenario. You can fix these scenarios by simply signing in as an admin user.

7.2.2 Fixing three broken scenarios

For the spec/features/deleting_projects_spec.rb feature, add a new `before` block, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.4 spec/features/deleting_projects_spec.rb

```
feature "Deleting projects" do
  before do
    sign_in_as!(FactoryGirl.create(:admin_user))
  end
  ...
```

When you run this feature using `bin/rspec spec/features/deleting_projects_spec.rb` it'll once again pass:

```
1 example, 0 failures
```

Wasn't that just incredibly easy? The other scenarios, which both live in the same file, should be just as easy to fix.

For the spec/features/editing_projects_spec.rb, use the same line from inside the `before` block from Listing 7.4 again, putting it at the top of the already existing `before` block in this file:

```
before do
  sign_in_as!(FactoryGirl.create(:admin_user))
  ...
end
```

Now this feature should also pass when you run it with `bin/rspec spec/features/editing_projects_spec.rb`:

```
2 examples, 0 failures
```

That should be the last of it. Now when you run `bin/rake spec`, everything once again passes:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
26 examples, 0 failures
```

Great! Now that accessing the actions is restricted, let's make a commit here:

```
$ git add .
$ git commit -m "Restrict access to project
             actions to admins only"
$ git push
```

You've restricted the controller actions, but the links to perform these actions, such as "New Project" and "Edit Project" will still be visible to the users. You should hide these links from the users who are not admins, because it's useless to show actions to people who can't perform them.

7.3 Hiding links

Now you'll learn how to hide certain links, such as the "New Project" link, from users who have no authorization to perform those actions in your application.

7.3.1 Hiding the New Project link

To begin, open a new file called spec/features/hidden_links_spec.rb. Inside 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 shown to admins. The code to do this is shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.5 spec/features/hidden_links_spec.rb

```
require 'spec_helper'

feature "hidden links" do
  let(:user) { FactoryGirl.create(:user) }
  let(:admin) { FactoryGirl.create(:admin_user) }

  context "anonymous users" do
    scenario "cannot see the New Project link" do
      visit '/'
      assert_no_link_for "New Project"
    end
  end

  context "regular users" do
    before { sign_in_as!(user) }
    scenario "cannot see the New Project link" do
      visit '/'
      assert_no_link_for "New Project"
    end
  end

  context "admin users" do
    before { sign_in_as!(admin) }
    scenario "can see the New Project link" do
      visit '/'
      assert_link_for "New Project"
    end
  end
end
```

In this spec, you first define two let blocks, one for `user` and one for `admin`. These create a non-admin user and an admin user respectively when they're called. There's 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. On 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 then there should be a "New Project" link on the page.

When you run this feature using `bin/rspec spec/features/hidden_links_spec.rb` the first thing you'll realise is that there is no `assert_link_for` or `assert_no_link_for` method defined:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: assert_no_link_for "New Project"
NoMethodError:
 undefined method `assert_no_link_for' for ...
```

That's because they were made up! But the intention of these methods is clear: they're supposed to check that a link either does or does not appear on the page. If you know the intention of a method, then writing it becomes much easier. You should define these methods now in a new file called spec/support/capybara_helpers.rb using the code from the following listing:

Listing 7.6 spec/support/capybara_helpers.rb

```
module CapybaraHelpers
  def assert_no_link_for(text)
    expect(page).to_not(have_css("a", :text => text),
      "Expected not to see the #{text.inspect} link, but did.")
  end

  def assert_link_for(text)
    expect(page).to(have_css("a", :text => text),
      "Expected to see the #{text.inspect} link, but did not.")
  end
end

RSpec.configure do |config|
  config.include CapybaraHelpers, :type => :feature
end
```

With this new file you define a module called `CapybaraHelpers` that includes the definitions for the two missing methods. Inside each method, you assert that within an `a` element on the page, there should or should not be the specified text. By using `have_css`, it will use a CSS selector to attempt to find a tag that matches the conditions.

With the two new methods defined, running `bin/rspec spec/features/hidden_links_spec.rb` should produce some actual failures:

```
1) hidden links anonymous users cannot see the New Project link
Failure/Error: assert_no_link_for "New Project"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
 Expected not to see the "New Project" link, but did.
  # ./spec/support/capybara_helpers.rb:3 ...
  # ./spec/features/hidden_links_spec.rb:11 ...

2) hidden links regular users cannot see the New Project link
   Failure/Error: assert_no_link_for "New Project"
     Expected not to see the "New Project" link, but did.
   # ./spec/support/capybara_helpers.rb:3 ...
   # ./spec/features/hidden_links_spec.rb:19 ...
```

The first two scenarios from the "Hidden links" feature fail, of course, because you've done nothing yet to hide the link that they're checking for! Open app/views/projects/index.html.erb and change the "New Project" link to the following in order to work towards hiding it:

```
<% admins_only do %>
  <%= link_to "New Project", new_project_path %>
<% end %>
```

The `admins_only` method isn't going to magically be there, so you'll need to define this yourself. The method will need to take a block, and then if the `current_user` is an admin it should run the code inside the block, and if they're not then it should show nothing.

You're going to want this helper to be available everywhere inside your application's views, and so the best place to define it would be inside the `ApplicationHelper`. If you only wanted it to be available to a specific controller's views, you would place it inside 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 inside the module using this code:

```
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 `do` and `end` in the call to it in your view. To run this code inside the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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, and if that method doesn't exist (as it wouldn't if `current_user` were `nil`), then it gives up and returns `nil`, rather than raising a `NoMethodError` exception.

When you run this feature using `bin/rspec spec/features/hidden_links_spec.rb`, it will pass because the links are now being hidden and shown as required:

```
3 examples, 0 failures
```

Now that you've got the "New Project" link hiding if the user isn't an admin, let's do the same thing for the "Edit Project" and "Delete Project" links.

7.3.2 Hiding the edit and delete links

You're going to need to add this `admins_only` helper to the "Edit Project" and "Delete Project" links on the projects `show` view as well to hide these links from the people who shouldn't see them. Before you do this, however, you should add further scenarios to cover these changes to spec/features/hidden_links_spec.rb.

In order to test that these links work, you're going to need to create a project during these tests. To enable that, define a `let` block underneath the two for users and admins inside this file using this line:

```
let(:project) { FactoryGirl.create(:project) }
```

Now you can use this `project` method to define scenarios inside the "anonymous users" `context` block to ensure that anonyomous users cannot see the "Edit Project" and "Delete Project" links by using the code from the following listing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.7 spec/features/hidden_links_spec.rb

```
context "anonymous users" do
  ...
  scenario "cannot see the Edit Project link" do
    visit project_path(project)
    assert_no_link_for "Edit Project"
  end

  scenario "cannot see the Delete Project link" do
    visit project_path(project)
    assert_no_link_for "Delete Project"
  end
end
```

Next, take these two scenarios and copy them into the "regular users" context block:

```
context "regular users" do
 ...
  scenario "cannot see the Edit Project link" do
    visit project_path(project)
    assert_no_link_for "Edit Project"
  end

  scenario "cannot see the Delete Project link" do
    visit project_path(project)
    assert_no_link_for "Delete Project"
  end
end
```

And then finally define two scenarios that ensure that admin users can see the links by placing the code from the following list inside the "admin users" `context`:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.8 spec/features/hidden_links_spec.rb

```
context "admin users" do
  ...
  scenario "can see the Edit Project link" do
    visit project_path(project)
    assert_link_for "Edit Project"
  end

  scenario "can see the Delete Project link" do
    visit project_path(project)
    assert_link_for "Delete Project"
  end
end
```

With these latest changes, you should now have 6 new scenarios inside the "Hidden links" feature, two checking the links for anonymous users, two checking for regular users and two checking for admins. Run this feature now with `bin/rspec spec/features/hidden_links_spec.rb` to see the new failures:

```
1) hidden links anonymous users cannot see the Edit Project link
   Failure/Error: assert_no_link_for "Edit Project"
     Expected not to see the "Edit Project" link, but did.
   # ./spec/support/capybara_helpers.rb:3 ...
   # ./spec/features/hidden_links_spec.rb:16 ..

2) hidden links anonymous users cannot see the Delete Project link
   Failure/Error: assert_no_link_for "Delete Project"
     Expected not to see the "Delete Project" link, but did.
   # ./spec/support/capybara_helpers.rb:3 ...
   # ./spec/features/hidden_links_spec.rb:21 ...
```

There are now four failing tests here, but only two shown above as the other two are nearly identical. These tests are failing because the "Edit Project" and "Delete Project" links are still visible to anonymous and signed in regular users. To make these tests pass change the links inside app/views/projects/show.html.erb and wrap the links in the `admins_only` helper, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.9 app/views/projects/show.html.erb

```
<% admins_only do %>
  <%= link_to "Edit Project", edit_project_path(@project) %>
  <%= link_to "Delete Project", project_path(@project),
         method: :delete,
         data: { confirm:
          "Are you sure you want to delete this project?"} %>
<% end %>
```

Now the links should be hidden from the users who are not admins. A great way to know if this is the case is to run the test using `bin/rspec spec/features/hidden_links_spec.rb`. When you run it, you should see this:

```
9 examples, 0 failures
```

All right, that was a little too easy, but that's just Rails.

This is a great point to ensure that everything is still working by running all the tests wiht `bin/rake spec`. According to the following output, everything's still in working order:

```
31 examples, 0 failures
```

Let's commit and push that:

```
git add .
git commit -m "Lock down specific projects controller
               actions for admins only"
git push
```

In this section, you ensured that only users with the `admin` attribute set to `true` can get to specific actions in your `ProjectsController`. This is a great

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

example of basic authorization.

Next, you learn to "section off" part of your site using a similar methodology and explore the concept of namespacing.

7.4 Namespace routing

While it's all fine and dandy to ensure that admin users can get to special places in your application, you haven't yet added the functionality to allow other admins to "promote" users to being admins themselves.

Since this functionality will only be provided to admins, it should go into its own namespace, imaginatively called "admin". The purpose of namespacing in this case is to separate a controller from the main area of the site so you can ensure that the only users accessing this particular controller (and any future controllers you create in this namespace) are admins.

7.4.1 Generating a namespaced controller

The first thing you will need is a namespaced controller, where the actions on users will be performed by admins. You can generate this namespaced controller by running this command:

```
$ bin/rails g controller admin/users index
```

When the `/` separator is used between parts of the controller, Rails knows to generate a namespaced controller called `Admin::UsersController` at app/controllers/admin/users_controller.rb. The views for this controller are at `app/views/admin/users`, and the spec is at spec/controllers/admin/users_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/users/index.html.erb for this action, as well as a route defined in config/routes.rb, like this:

```
get "users/index"
```

You won't be using this particular route, so you can remove it from `config/routes.rb`.[2]We do need an actual route though. Give this a try:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 2 The route is also incorrect. You generated a namespaced controller which should have a namespaced route as well (e.g. `get "/admin/users/index"`), so even if you did leave this route in the routes file, it wouldn't do anything particularly useful.

```
namespace :admin do
  resources :users
end
```

You've used `resources :users` before, but now we use it again: this time, it's inside a namespace. Running `bin/rake routes` will show you more about this route:

```
    admin_users GET    /admin/users(.:format)        admin/users#index
                POST   /admin/users(.:format)        admin/users#create
 new_admin_user GET    /admin/users/new(.:format)      admin/users#new
edit_admin_user GET    /admin/users/:id/edit(.:format) admin/users#edit
     admin_user GET    /admin/users/:id(.:format)      admin/users#show
                PATCH  /admin/users/:id(.:format)    admin/users#update
                PUT    /admin/users/:id(.:format)    admin/users#update
                DELETE /admin/users/:id(.:format)   admin/users#destroy
          users GET    /users(.:format)                users#index
                POST   /users(.:format)                users#create
       new_user GET    /users/new(.:format)            users#new
      edit_user GET    /users/:id/edit(.:format)       users#edit
           user GET    /users/:id(.:format)            users#show
                PATCH  /users/:id(.:format)            users#update
                PUT    /users/:id(.:format)            users#update
                DELETE /users/:id(.:format)            users#destroy
```

We've just included the relevant parts. You can now see that we have a double set of routes with `users` in them: one inside the namespace and one outside.

The difference between this controller and all the other controllers you have generated so far is that this controller should block requests from non-admin users and allow admin users. To ensure that this is what the controller does, you must write a spec for this newly generated controller.

7.4.2 Testing a namespaced controller

Open spec/controllers/admin/users_controller_spec.rb and write an example to ensure non-signed-in users can't access the `index` action, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.10 spec/controllers/admin/users_controller_spec.rb

```
require 'spec_helper'

describe Admin::UsersController do
  let(:user) { FactoryGirl.create(:user) }

  context "standard users" do
    before { sign_in(user) }

    it "are not able to access the index action" do
      get 'index'
      expect(response).to redirect_to('/')
      expect(flash[:alert]).to eql("You must be an admin to do that.")
    end
  end
end
```

In this new spec, you create a new confirmed user by using the `user` factory, and then sign in as them by using the `sign_in` method, provided by the `AuthHelpers` module that you had previously included in spec/spec_helper.rb. In the test, you attempt a `GET` request to the `index` action of `Admin::UsersController` and when that happens, the response should redirect you to the root path of the application and a `flash[:alert]` message should be set declaring that "You must be an admin to do that".

When you run this spec file using `bin/rspec spec/controllers/admin/users_controller_spec.rb`, it will make the request to the `index` action just fine, but precisely the opposite of what it should do:

```
Failure/Error: response.should redirect_to('/')
Expected response to be a <:redirect>, but was <200>
```

Normal users should not have access to the `index` action of `Admin::UsersController`, but should instead be sent away.. This test is failing because you have not implemented the `authorize_admin!` `before_action` inside the `Admin::UsersController`. Rather than

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

placing this `before_action` inside `Admin::UsersController`, what you should do is create a new controller that all controllers inside the "admin" namespace should inherit from, and then inside this controller put the `before_action` call.

Create a new file at app/controllers/admin/base_controller.rb and fill it with this code:

```
class Admin::BaseController < ApplicationController
  before_action :authorize_admin!
end
```

This file can double as an eventual homepage for the admin namespace and as a class that the other controllers inside the admin namespace can inherit from, which you'll see in a moment. You inherit from `ApplicationController` with this controller so you receive all the benefits it provides, like the `authorize_admin!` method and the typical controller functions provided by `ActionController::Base`.

One small thing that you will need to fix first is the availability of the `authorize_admin!` method. At the moment, this is locked away inside `ProjectsController`, but should be made available to all controllers of application, since you are now depending on it inside `Admin::BaseController`. Remove these lines from the end of `ProjectsController` inside app/controllers/projects_controller.rb:

```
def authorize_admin!
 require_signin!

 unless current_user.admin?
   flash[:alert] = "You must be an admin to do that."
   redirect_to root_path
 end
end
```

And then place them inside app/controllers/application_controller.rb inside the `ApplicationController` definition, underneath a `private` keyword:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
private

  def authorize_admin!
   require_signin!

   unless current_user.admin?
     flash[:alert] = "You must be an admin to do that."
     redirect_to root_path
   end
  end
```

Now the `authorize_admin!` method will be made available to all controllers that inherit from `ApplicationController`. One final thing to change is the inheritance of the `Admin::UsersController` class, so that it inherits from `Admin::BaseController`, meaning that the `authorize_admin` filter will run before all actions inside this controller. Open app/controllers/admin/users_controller.rb and change the first line of the controller from this:

```
class Admin::UsersController < ApplicationController
```

to this:

```
class Admin::UsersController < Admin::BaseController
```

Because `Admin::UsersController` now inherits from `Admin::BaseController`, the `before_action` from `Admin::BaseController` now runs for every action inside `Admin::UsersController`, and therefore in your spec, should pass. Run it with `bin/rspec spec/controllers/admin/users_controller_spec.rb` now, and you should see just that

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Excellent! Now 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 `bin/rake spec`:

```
34 examples, 0 failures, 2 pending
```

Great, everything is still passing, but there are two pending tests:

```
# ./spec/helpers/admin/users_helper_spec.rb:14
# ./spec/views/admin/users/index.html.erb_spec.rb:4
```

These two tests were added when you ran `bin/rails g controller admin/users`. The first test is a simple helper test, and the second is a *view spec* , which can be used to ensure that rendering that particular view works as intended. [3] You don't have a need for these two particular tests, so delete both of these files. When you re-run `bin/rake spec` you should see this output:

Footnote 3 Read more about view spec testing at https://www.relishapp.com/rspec/rspec-rails/v/2-9/docs/view-specs/view-spec

```
36 examples, 0 failures
```

Let's commit this now:

```
git add .
git commit -m "Add admin namespaced users controller"
git push
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

7.5 Namespace-based CRUD

Now that only admins can access this namespace, you can create the CRUD actions for the `Admin::UsersController` too, as you did for the `TicketsController` and `ProjectsController` controllers. This will allow admin users to create new users in the application, without them needing to sign up first. Along the way, you'll also set up a homepage for the admin namespace.

With the first part of CRUD being the "creation" of a resource, it would be a great idea to start with that. Begin by first creating a new directory for the admin features called spec/features/admin and then writing 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 7.11 spec/features/admin/creating_users_spec.rb

```
require 'spec_helper'

feature "Creating Users" do
  let!(:admin) { FactoryGirl.create(:admin_user) }

  before do
    sign_in_as!(admin)
    visit '/'
    click_link "Admin"
    click_link "Users"
    click_link "New User"
  end

  scenario 'Creating a new user' 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 `bin/rspec spec/features/admin/creating_users_spec.rb`, the first couple of lines in the `before` block will pass, but it will fail due to a missing "Admin" link:

```
Failure/Error: click_link "Admin"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Capybara::ElementNotFound:
  Unable to find link "Admin"
```

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 in the `if user_signed_in?` statement in the `nav` element:

```
<% if current_user %>
  Signed in as <%= current_user.email %>
  <% admins_only do %>
    <%= link_to "Admin", admin_root_path %>
  <% end %>
<% else %>
...
```

This way, the link will only be shown to users who are both signed in an admins. The next thing you will need to do is to define a namespaced root route.

7.5.1 Adding a namespace root

At the moment, `admin_root_path` doesn't exist. To define it, open config/routes.rb and define an `admin` namespace using this code:

```
namespace :admin do
  root :to => "base#index"
  resources :users
end
```

When you re-run the spec with `bin/rspec spec/features/admin/creating_users_spec.rb`, it fails because you don't have an `index` action for `Admin::BaseController`:

```
The action 'index' could not be found for Admin::BaseController
```

This action should render a view that provides links to special admin

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

functionality. Let's add that now.

7.5.2 The index action

Open app/controllers/admin/base_controller.rb and add the `index` action so the class definition looks like the following listing.

Listing 7.12 app/controllers/admin/base_controller.rb

```
class Admin::BaseController < ApplicationController
  before_action :authorize_admin!

  def index
  end
end
```

You define the action here to show users that this controller has an `index` action. The next step is to create the view for the `index` action by creating a new file at app/views/admin/base/index.html.erb and filling it with the following content:

```
<%= link_to "Users", admin_users_path %>
Welcome to Ticketee's Admin Lounge. Please enjoy your stay.
```

You needn't wrap the link in an `admins_only` here because you're inside a page that's visible only to admins. When you run the feature again using `bin/rspec spec/features/admin/creating_users_spec.rb`, you *don't* get a message saying `The action 'index' could not be found` even though you should. Instead, you get this:

```
Failure/Error: click_link "New User"
  Unable to find link "New User"
```

This unexpected output occurs because the `Admin::UsersController` inherits from `Admin::BaseController`, where you just defined an `index` method. By inheriting from this controller, `Admin::UsersController` also

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

inherits its views. When you inherit from a class like this, you get the methods defined in that class too. So really, there *is* an `index` action defined in `Admin::UsersController`. You can override the `index` action from `Admin::BaseController` by redefining it in `Admin::UsersController`, as in the following listing.

Listing 7.13 app/controllers/admin/users_controller.rb

```
class Admin::UsersController < Admin::BaseController
  def index
    @users = User.order(:email)
  end
end
```

Next, you will 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 gathered up by the controller, as shown in the following listing.

Listing 7.14 app/views/admin/users/index.html.erb

```
<%= link_to "New User", new_admin_user_path %>
<ul id='users'>
  <% @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 we used it with `[:edit, @project, ticket]`, but it bears repeating here.

When you run `bin/rspec spec/features/admin/creating_users_spec.rb` again, you're told the `new` action is missing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_link "New User"
AbstractController::ActionNotFound:
  The action 'new' could not be found for Admin::UsersController
```

7.5.3 The new action

Let's add the `new` action `Admin::UsersController` now by using this code:

```
def new
  @user = User.new
end
```

And let's create the view for this action at app/views/admin/users/new.html.erb:

```
<h2>New User</h2>
<%= render "form" %>
```

Next, you'll need to create the form partial that's used in the `new` template, which you can do by using the code from the following listing, Listing 7.15. It must contain the `email` and `password` fields, which are the bare essentials for creating a user.

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

  <p>
    <%= f.label :password %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    <%= f.password_field :password %>
  </p>
  <%= f.submit %>
<% end %>
```

For this `form_for`, 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 normally be generated. You can also use this array syntax with `link_to` (seen earlier) and `redirect_to` helpers. Any symbol passed anywhere in the array is interpreted literally.

When you run the feature once again with `bin/rspec spec/features/admin/creating_users_spec.rb`, you're told there's no action called `create`:

```
Failure/Error: click_button "Create User"
AbstractController::ActionNotFound:
  The action 'create' could not be found for Admin::UsersController
```

7.5.4 The create action

Let's create that action now by using this code:

```
def create
  params = user_params.dup
  params[:password_confirmation] = params[:password]
  @user = User.new(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 :action => "new"
  end
end

private
  def user_params
    params.require(:user).permit(:name,
                                 :password,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
                                    :password_confirmation)
  end
```

There's now an "Admin" link that an admin can click on, which takes them to the `index` action within `Admin::BaseController`. On the template rendered for this action (app/views/admin/base/index.html.erb) there's a "Users" link that goes to the `index` action within `Admin::UsersController`. On the template for *this* action, there's a "New User" link that presents the user with a form to create a user. When the user fills in this form and hits the "Create User" button, it goes to the `create` action inside `Admin::UsersController`.

There's only one tricky part about this: because we want to create a new user with just a password, we have to manually duplicate our parameters hash and add in a copy of the password as our `password_confirmation`. Normally, "don't make a user without a confirmation" is a great setting, but it makes for some slightly awkward code here. I think that's a pretty good tradeoff, though.

We need one more thing, though. Try to run your tests with `bin/rspec spec/features/admin/creating_users_spec.rb`:

```
Failure/Error: expect(page).to have_content("User has not...
  expected there to be text "User has not been created." in
  "User has been created. Signed in as sample@example.com
  Admin New User /admin/users/2 sample@example.com"
```

With all the steps implemented, both scenarios inside the "Creating Users" feature should now pass. Find out with a final run of `bin/rspec spec/features/admin/creating_users_spec.rb`.

```
1 examples, 0 failures
```

This is another great middle point for a commit, so let's do so now. As usual, you should run `bin/rake spec` to make sure everything's still working:

```
37 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Great! Let's push that.

```
git add .
git commit -m "Add the ability to create users
               through the admin backend"
git push
```

While this functionality allows you to create new users through the admin backend, it doesn't let you create *admin* users. That's up next.

7.5.5 Creating admin users

To create users who are an admin, you will need to be able to trigger this field by using a a check box on the user form. When this checkbox is checked and the `User` record is saved, that user will now 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 7.15 spec/features/admin/creating_users_spec.rb

```
scenario "Creating an admin user" 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")
  within("#users") do
    expect(page).to have_content("admin@example.com (Admin)")
  end
end
```

Now when you run `bin/rspec spec/features/admin/creating_users_spec.rb`, it will fail when it attempts to check the "Is an admin?" checkbox.

```
Failure/Error: check "Is an admin?"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Capybara::ElementNotFound:
  Unable to find checkbox "Is an admin?"
```

You want to add this check box to the form for creating users, which you can do by adding the following code to the `form_for` block inside app/views/admin/users/_form.html.erb:

```
<p>
  <%= f.check_box :admin %>
  <%= f.label :admin, "Is an admin?" %>
</p>
```

With this check-box in place, when you re-run `bin/rspec spec/features/admin/creating_users_spec.rb`, it will not be able to find "admin@example.com (Admin)" on the page, within the `#users` element.

```
  expected there to be content "admin@example.com (Admin)" in ...
```

The problem here is that only the user's email address is displayed: no text appears to indicate he or she is a user. To get this text to appear, change the line 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:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
#<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 inside the class definition in app/models/user.rb, underneath the `has_secure_password` line:

```
def to_s
  "#{email} (#{admin? ? "Admin" : "User"})"
end
```

The `to_s` method will now output something like "user@example.com (User)" if the user is not an admin, and "admin@example.com (Admin)" if the user is an admin. Now that the `admin` field is set and the indication of the user being an admin is displayed on the page, the feature should pass when you run `bin/rspec spec/features/admin/creating_users_spec.rb`. But it doesn't:

```
Failure/Error: expect(page).to have_content("admin@example.com (Admin)")
expected there to be text "admin@example.com (Admin)"
  in "(user) sample@example.com (Admin)"
```

What gives? Well, it says `"sample@example.com"`, but we were expecting `"admin@example.com"`. The `"sample@example.com"` admin is the one we're logged in as, not the one we made.

The answer is `strong_parameters`. Re-examine your app/controllers/admin/users_controller.rb:

```
def user_params
  params.require(:user).permit(:name, :password, :password_confirmation)
end
```

Yup. We're not allowing assignment of `admin`. Normally, this would be great,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

but we need to do this here! Since this controller is only accessable by admins, it's okay to assign this value. Let's change it to this:

```
def user_params
  params.require(:user).permit(:name,
                               :email,
                               :password,
                               :password_confirmation,
                               :admin)
end
```

```
2 examples, 0 failures
```

This is another great time to commit, and again, run `bin/rake spec` to make sure everything works:

```
38 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 examine the *updating* and *deleting* parts of the CRUD next.

7.5.6 Editing users

This section focuses on adding the updating capabilities for the `Admin::UsersController`.

As usual, you start by writing a feature to cover this functionality, placing the file at spec/features/admin/editing_users_spec.rb and filling it with the content

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

from the following listing.

Listing 7.16 spec/features/admin/editing_users_spec.rb

```
require 'spec_helper'

feature 'Editing a user' do
  let!(:admin_user) { FactoryGirl.create(:admin_user) }
  let!(:user) { FactoryGirl.create(:user) }

  before do
    sign_in_as!(admin_user)
    visit '/'
    click_link "Admin"
    click_link "Users"
    click_link user.email
    click_link "Edit User"
  end

  scenario "Updating a user's details" do
    fill_in "Email", with: "newguy@example.com"
    click_button "Update User"

    expect(page).to have_content("User has been updated.")

    within("#users") do
      expect(page).to have_content("newguy@example.com")
      expect(page).to_not have_content(user.email)
    end
  end

  scenario "Toggling user's admin ability" do
    check "Is an admin?"
    click_button "Update User"

    expect(page).to have_content("User has been updated.")

    within("#users") do
      expect(page).to have_content("#{user.email} (Admin)")
    end
  end
end
```

When you run this with `bin/rspec spec/features/admin/editing_users_spec.rb`, you'll get a curious error:

```
2) Editing a user Toggling user's admin ability
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_link user.email
Capybara::Ambiguous:
  Ambiguous match, found 2 elements matching link "sample@example.com"
```

Capybara is concerned about having two links on the page with the same name. If you think about it, we're making two users here, and since the factory declares our `User` like this:

```
FactoryGirl.define do
  factory :user do
    name "username"
    email "sample@example.com"
...
```

All of our users will have the same email. Seems bad. Let's write a test to fix it. Open up spec/models/user_spec.rb:

```
it "requires an email" do
  u = User.new(name: "steve",
               password: "hunter2",
               password_confirmation: "hunter2")

  u.save
  expect(u).to_not be_valid

  u.email = "steve@example.com"
  u.save
  expect(u).to be_valid
end
```

When you run this with `bin/rspec spec/models/user_spec.rb`, you'll get a failure:

```
Failure/Error: expect(u).to_not be_valid
  expected valid? to return false, got true
```

Awesome. This was expected: We don't think our user should be valid without an email, but they are. Luckily, we can fix this! Add this line to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

app/models/user.rb:

```
validates :email, presence: true
```

This validation ensures that we have to have an email for a user to be valid. But we still get the same failure when we run it:

```
Failure/Error: expect(u).to be_valid
  expected valid? to return true, got false
```

What gives?

This one gave me a pause when I saw it. However, I figured it out quickly due to a habit I've made: always read the error message. Let's read that again:

```
Failure/Error: expect(u).to be_valid
  expected valid? to return true, got false
# ./spec/models/user_spec.rb:18:in ...
```

Line 18? Which test is line 18? It's part of the block with line 5:

```
it "needs a password and confirmation to save" do
```

That's a totally different test! We need to fix every call to `User.new`. For example, on line 6:

```
# old
u = User.new(name: "steve")

# new
u = User.new(name: "steve", email: "steve@example.com")
```

Once we do this, we our tests pass again:

```
5 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Now that we have a passing test for our behavior, we need to bring our factory in line with our models. We're going to use a feature called "sequences." Update your spec/factories/user_factory.rb to look like this:

```
FactoryGirl.define do
  sequence(:email) {|n| "user#{n}@example.com" }

  factory :user do
    name "username"
    email { generate(:email) }
    password "hunter2"

    factory :admin_user do
      admin true
    end
  end
end
```

The `sequence` method provides a sequence of things: every time you call it, the number `n` gets bigger. So the first time, we'll get `user1@example.com`, the second time we'll get `user2@example.com`, and so on.

Now, let's re-run this feature using `bin/rspec spec/features/admin/editing_users_spec.rb`. You will discover the `show` action is missing:

```
Failure/Error: click_link user.email
AbstractController::ActionNotFound:
  The action 'show' could not be found for Admin::UsersController
```

This failure is happening when the link containing the user's email address is clicked on in the `before` block for the feature. Define the `show` action in the `Admin::UsersController`, directly under the `index` action, because grouping the different parts of CRUD is conventionaly done in this order: index, show, new, create, edit, update and destroy. Define the `show` action as a blank action:

```
def show
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `show` action template will require a `@user` variable, and so you should create a `set_user` method that you can call as a `before_action` in the `Admin::UsersController`. Define this new `set_user` method underneath all the other methods already in the controller, since it will be a `private` method:

```
private

  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`:

```
before_action :set_user, only: [:show, :edit, :update, :destroy]
```

With the `find_user` and `show` methods now in place in the controller, what's the next step going to be? Find out by running `bin/rspec spec/features/admin/editing_users_spec.rb` again. You will see this error now:

```
Failure/Error: click_link user.email
ActionView::MissingTemplate:
Missing template admin/users/show,
                 admin/base/show,
                 application/show
   with {:locale=>[:en],
         :formats=>[:html],
         :handlers=>[:erb,
                     :builder,
                     :coffee]}.
Searched in:
   * "...ticketee/app/views"]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

NOTE **Template inheritance**

Notice how in the error message above three different templates are listed: `admin/users/show`, `admin/base/show` and `application/show`. Rails is attempting to look for these three templates in exactly that order, but can't find any of them.

The reason why this happens was explained earlier, but is good to re-enforce here. The reason is because the `Admin::UsersController` inherits from `Admin::BaseController` and therefore inherits its templates inside app/views/admin/base as well. The `Admin::BaseController` inherits from `ApplicationController`, and so by inheritance both `Admin::BaseController` and `Admin::UsersController` also have the templates from inside the (imaginary) app/views/application directory as well.

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 uses the following code:

```
<h2><%= @user %></h2>
<%= link_to "Edit User", edit_admin_user_path(@user) %>
```

Now when you run `bin/rspec spec/features/admin/editing_users_spec.rb`, the line that clicks the link containing the user's email passes, and you're on to the next step:

```
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 the `Admin::UsersController`, which displays information for 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

7.5.7 The edit and update actions

Add the `edit` action directly underneath the `create` action in your controller. It should be another blank method like the `show` action.

```
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:

```
<h2>Editing a User</h2>
<%= render "form" %>
```

Okay, that's the current failure for the feature dealt with. Find out what's next with another run of `bin/rspec spec/features/admin/editing_users_spec.rb`. You should be told the `update` action doesn't exist:

```
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 your `Admin::UsersController`, as shown in the following listing. You don't need to set up the `@user` variable here because the `find_user before_action` does it for you:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.17 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[:alert] = "User has not been updated."
    render action: "edit"
  end
end
```

With this action in place, you need to delete the password parameters from `params[:user]` if they are blank. Otherwise, the application will attempt to update a user with a blank password and users wouldn't like that. Above `if @user.update(user_params)`, insert this code:

```
if params[:user][:password].blank?
  params[:user].delete(:password)
  params[:user].delete(:password_confirmation)
end
```

Now the entire action looks like the following listing.

Listing 7.18 app/controllers/admin/users_controller.rb

```
def update
  if params[:user][:password].blank?
    params[:user].delete(:password)
    params[:user].delete(:password_confirmation)
  end

  if @user.update_attributes(params[:user], :as => :admin)
    flash[:notice] = "User has been updated."
    redirect_to admin_users_path
  else
    flash[:alert] = "User has not been updated."
    render :action => "edit"
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you run `bin/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 your `Admin::UsersController`: `edit` and `update`. Admin users can now update users' details as they please.

As always, you'll want to run `bin/rake spec` to make sure you didn't break everything. Just one quick run.... oh no!

```
41 examples, 1 failure
```

What'd we do? Looking more closely at the failures, it has to do with emails. The new email code probably broke this: thank goodness for automated tests!

The issue is this:

```
expected there to be text "Created by sample@example.com" in
  "Created by user12@example.com Edit Ticket Delete Ticket My
  pages are ugly!"
```

Ahhh. Our test is looking for the wrong email address. We can fix it! Open up spec/features/creating_tickets_spec.rb, and check out the lines that failed:

```
expect(page).to have_content("Created by sample@example.com")
```

Ah. A hardcoded email. Let's save the one that we used in our setup. Change this line to this:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
expect(page).to have_content("Created by #{@email}")
```

And then change the very start of the `before` block:

```
before do
  project = FactoryGirl.create(:project)
  user = FactoryGirl.create(:user)
  @email = user.email
```

Great! Let's run that spec again, using `bin/rspec spec/features/creating_tickets_spec.rb`:

```
3 examples, 0 failures
```

Awesome! Now run `bin/rake spec` to ensure nothing was broken by these latest changes. You should see this output:

```
41 examples, 0 failures
```

Whew! Done! Let's make a commit for this new feature:

```
git add .
git commit -m "Add ability 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.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

7.5.8 Deleting users

There comes a time in an application's life when you need to delete users. Maybe they asked for their account to be removed. Maybe they were being pesky. Or maybe you have another reason to delete them. Whatever the case, having the functionality to delete users is helpful.

Keeping with the theme so far, you first write a feature for deleting users (using the following listing) and put it at spec/features/admin/deleting_users_spec.rb.

Listing 7.19 spec/features/admin/deleting_users_spec.rb

```
require 'spec_helper'

feature 'Deleting users' do
  let!(:admin_user) { FactoryGirl.create(:admin_user) }
  let!(:user) { FactoryGirl.create(:user) }

  before do
    sign_in_as!(admin_user)
    visit '/'

    click_link 'Admin'
    click_link 'Users'
  end

  scenario "Deleting a user" do
    click_link user.email
    click_link "Delete User"

    expect(page).to have_content("User has been deleted")
  end
end
```

When you run this feature using `bin/rspec spec/features/admin/deleting_users_spec.rb`, you get right up to the first line inside the scenario with no issue and then it complains about the second line:

```
  Unable to find link "Delete User"
```

Of course, you need the "Delete User" link! Add it to the `show` template at app/views/admin/users/show.html.erb right underneath the "Edit User" link:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= link_to "Delete User", admin_user_path(@user), method: :delete,
  data: { confirm: "Are you sure you want to delete this user?"} %>
```

You need to add the `destroy` action next, directly under the `update` action in `Admin::UsersController`, as shown in the following listing.

Listing 7.20 app/controllers/admin/users_controller.rb

```
def destroy
  @user.destroy
  flash[:notice] = "User has been deleted."
  redirect_to admin_users_path
end
```

When you run `bin/rspec spec/features/admin/deleting_users_spec.rb`, the feature passes because you now have the "Delete User" link and its matching `destroy` action:

```
1 example, 0 failures
```

There's one small problem with this feature, though: it doesn't stop you from deleting yourself!

7.5.9 Ensuring you can't delete yourself

To make it impossible to delete yourself, you must add another scenario to the spec/features/admin/deleting_users_spec.rb, shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 7.21 spec/features/admin/deleting_users_spec.rb

```
scenario "Users cannot delete themselves" do
  click_link admin_user.email
  click_link "Delete User"

  expect(page).to have_content("You cannot delete yourself!")
end
```

When you run this feature with `bin/rspec spec/features/admin/deleting_users_spec.rb`, the first two lines of this scenario pass, but the third one fails, as you might expect, because you haven't added the message! Change the `destroy` action in the `Admin::UsersController` to the following listing.

Listing 7.22 app/controllers/admin/users_controller.rb

```
def destroy
  if @user == current_user
    flash[:alert] = "You cannot delete yourself!"
  else
    @user.destroy
    flash[:notice] = "User has been deleted."
  end

  redirect_to admin_users_path
end
```

Now, before the `destroy` method does anything, it checks to see if the user attempting to be deleted is the current user and stops it with the `"You cannot delete yourself!"` message. When you run `bin/rspec spec/features/admin/deleting_users_spec.rb` this time, the scenario passes:

```
2 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Great! With the ability to delete users implemented, you've completed the CRUD for `Admin::UsersController` and for the users resource entirely. Now make sure you haven't broken anything by running `bin/rake spec`. You should see this output:

```
43 examples, 0 failures
```

Fantastic! Commit and push that!

```
$ git add .
$ git commit -m "Add feature for deleting users,
             including protection against self-deletion"
$ git push
```

With this final commit, you've now got your admin section created, and it provides a great CRUD interface for users in this system so that admins can modify their details when necessary.

7.6 Summary

For this chapter, you dove into basic access control and added a field called `admin` to the users table. You used `admin` to allow and restrict access to a namespaced controller.

Then you wrote the CRUD interface for the users resource underneath the admin namespace. This interface is used in the next chapter to expand on the authorization that you've implemented so far: restricting users, whether admin users or not, to certain actions on certain projects. We rounded out the chapter by not allowing users to delete themselves.

The next chapter focuses on enhancing the basic permission system you've implemented so far, introducing a gem called `cancan`. With this permission system, you'll have much more fine-grained control over what users of your application can and can't do to projects and tickets.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Index Terms

actions, controller generator
add_column, :default option
concat, ActionView::Base
form_for, Array with Symbol usage
have_css, Capybara
Namespaces
sign_in, test helper
try method

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

8 *Fine-grained access control*

At the end of chapter 7, you learned a basic form of authorization based on a Boolean field called `admin` on the `users` table. If this field is set to `true`, identifying admin users, those users can access the CRUD functions of the `Project` resource as well as an admin namespace where they can perform CRUD on the `User` resource.

In this chapter, you'll expand on authorization options by implementing a broader authorization system using a `Permission` model. The records for this model's table define the actions specified users can take on objects from your system, such as projects. Each record tracks the user who has a specific permission, the object to which the permission applies, and the type of permission granted.

The authorization implemented in this chapter is *whitelist authorization*. Under whitelist authorization, all users are denied access to everything by default, and you must specify what the user can do. The opposite is *blacklist authorization*, under which all users are allowed access to everything by default and you must block what they may not access. You use whitelist authorization for your application because you may have a large number of projects and want to assign a user to only one of them. Whitelist authorization involves fewer steps in restricting a user to one project.

A good way to think about whitelist authorization is as the kind of list a security guard would have at an event. If you're not on the list, you don't get in. A blacklist comparison would be if the security guard had a list of people who *weren't* allowed in.

This chapter guides you through restricting access to the CRUD operations of `TicketsController` one by one, starting with reading and then moving into creating, updating, and deleting. Any time a user wants to perform one of these

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

actions, they must be granted permission to do so, or added to "the list." During this process, you'll see another gem called CanCan, which provides some methods for your controllers and views that help you check the current user's permission to perform a specific action.

You'll first set up permissions through the Capybara features. Once you're finished with restricting the actions in your controller, you'll generate functionality in the backend to allow administrators of the application to assign permissions to users.

8.1 Restricting read access

A time comes in every ticket-tracking application's life when it's necessary to restrict which users can see which projects. For example, you could be operating in a consultancy where some people are working on one application and others are working on another. You want the admins of the application to be able to customize which projects each user can see.

First, you'll create a model called `Permission` that tracks which users have which permissions for which actions. But before you create that model, you must update one of your Viewing Projects features to make sure only users who have permission to view a project are able to do so.

8.1.1 Testing read-access restriction

Add a background, and change the scenario in this feature to set up a user with the correct permissions. Then make the user visit that project. Change the code in the scenario in this feature to what is shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.1 spec/features/viewing_projects_spec.rb

```
require 'spec_helper'

feature "Viewing projects" do
  let!(:user) { FactoryGirl.create(:user) }
  let!(:project) { FactoryGirl.create(:project) }

  before do
    sign_in_as!(user)
    define_permission!(user, :view, project)
  end

  scenario "Listing all projects" do
    visit '/'
    click_link project.name

    expect(page.current_url).to eql(project_url(project))
  end
end
```

❶

You've effectively rewritten a large portion of this feature, which is common practice when implementing such large changes. The `define_permission!` ❶ method defines a permission record. It's responsible for giving the specified user access to the specified object—a `Project` object, in this case. This method is currently undefined, so when you run `bin/rspec spec/features/viewing_projects_spec.rb`, it complains about that:

```
Failure/Error: define_permission!(user, :view, project)
NoMethodError:
 undefined method `define_permission!' for ...
```

To create this method, create a new file at spec/support/authorization_helpers.rb. In this file, put the following content.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.2 spec/support/authorization_helpers.rb

```
module AuthorizationHelpers
  def define_permission!(user, action, thing)
    Permission.create!(user: user,
                       action: action,
                       thing: thing)
  end
end

RSpec.configure do |c|
  c.include AuthorizationHelpers
end
```

With the `define_permission!` method defined, re-running this spec with `bin/rspec spec/features/viewing_projects_spec.rb` results in it now complaining about the missing `Permission` class:

```
Failure/Error: define_permission!(user, :view, project)
NameError:
  uninitialized constant AuthorizationHelpers::Permission
```

This model will be used to track what users have what kind of permission on certain objects. Therefore, it needs *four fields:* `user_id` to track the association with the user, `action` to track what kind of action this permission gives, and `thing_id` and `thing_type` to track the association with the object on which the permission is granted.

Two fields are used to track the `thing` association because a “thing” could be one of many things. It could be a project; it could be a ticket; it could be anything. This type of association is called a *polymorphic association* . You’ll see how the polymorphic association works when you define the actual association in the model in a short while.

8.1.2 Creating and using the Permission model

Create the `Permission` model by generating it using the following command, typed all on one line:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ bin/rails g model permission user_id:integer thing_id:integer
  thing_type:string action:string
```

With this model and its related migration, you can run `bin/rake db:migrate` and `bin/rake db:test:prepare` to set up the development and test databases. When you run your feature again with `bin/rspec spec/features/viewing_projects_spec.rb`, you get this error message:

```
Failure/Error: define_permission!(user, :view, project)
ActiveRecord::UnknownAttributeError:
  unknown attribute: user
```

You know better than Rails! This "attribute" is supposed to be an association for the user this permission relates to. Define this association using this line in app/models/permission.rb:

```
class Permission < ActiveRecord::Base
  belongs_to :user
end
```

Running the feature again with `bin/rspec spec/features/viewing_projects_spec.rb`, you'll see that it's now the attribute Rails doesn't know about.

```
Failure/Error: define_permission!(user, :view, project)
ActiveRecord::UnknownAttributeError:
  unknown attribute: thing
```

Define this association in the `Permission` model by using this line, placing it under the `user` association:

```
  belongs_to :thing, polymorphic: true
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This code represents a polymorphic association, which, as mentioned earlier, needs to associate with many types of objects. A polymorphic association uses the `thing_type` and `thing_id` fields to determine what object a `Permission` object relates to. Figure 8.1 illustrates how this association works.

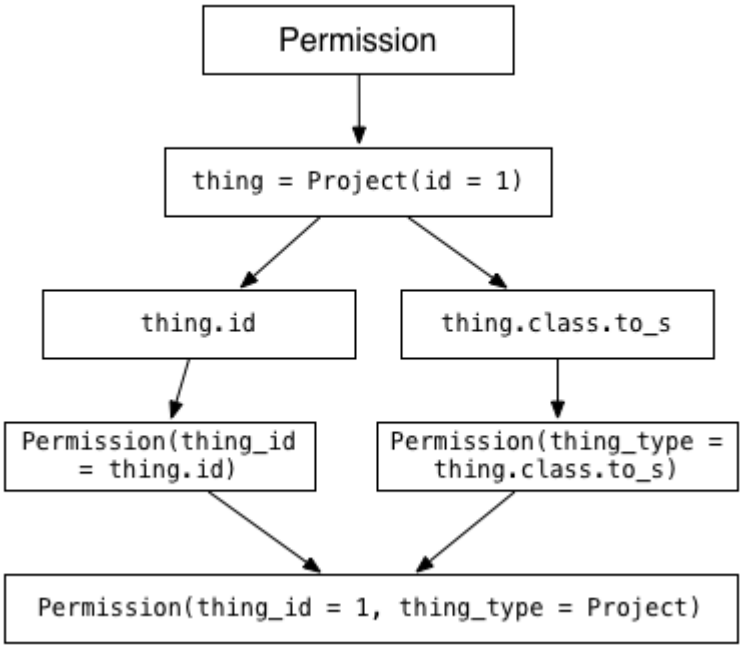

Figure 8.1 Polymorphic saving

When you assign an object to the `thing` polymorphic association, instead of saving `thing_id` as in a normal `belongs_to`, Rails also saves the `thing_type` field, which is the string version of the object's class, or `thing.class.to_s`. In this step of the application, the `thing_type` field is set to `"Project"` because you're assigning a `Project` object to `thing`. Therefore, the new record in the table has both `thing_type` and `thing_id` attributes set.

When Rails loads this object, it goes through the process shown in figure 8.2.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

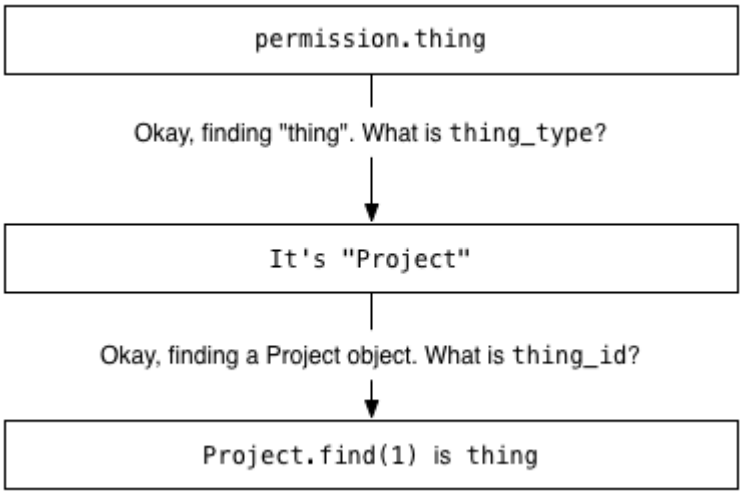

Figure 8.2 Polymorphic loading

Rails knows this is a polymorphic association because you told it in the `Permission` model by using the `polymorphic` option on the association, and it therefore uses the `thing_id` and `thing_type` fields to find the object. By knowing `thing_type`, Rails can figure out what model the association is and then use that model to load a specific object with the `id` of `thing_id`. Then boom, you've got a `Project` object.

Now when you run `bin/rspec spec/features/viewing_projects_spec.rb`, it passes:

```
1 example, 0 failures
```

The feature should pass with or without the new permission step, because, at the moment, the permission settings have no bearing on what projects a user can see.

The easiest way to specify which projects users can see is to restrict the scope of the projects the `show` action searches on so that projects the user doesn't have access to don't appear in this list. By default, a `find` on a model searches all records in the related table, but you can add a `scope` method to your model to allow you to search on restricted sets of records.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

8.2 Restricting by scope

To restrict the `show` action to certain record sets, you implement a *scope* on the `Project` model that returns only the projects with related `Permission` records that declare the user is authorized to read the projects. Before you scope down this `find`, you must write a spec to test that the `show` action in the `ProjectsController` really does scope down this find; and if the project can't be found, the controller should deny all knowledge of a project ever having existed.[1]

Footnote 1 For if you don't write a test, we will give you one of these: _

The spec goes in spec/controllers/projects_controller_spec.rb directly under the spec for testing that standard users can't access specified actions, but still in the `context` block for standard users. This spec is shown in the following listing.

Listing 8.3 spec/controllers/projects_controller_spec.rb

```
it "cannot access the show action without permission" do
  project = FactoryGirl.create(:project)
  get :show, id: project.id

  expect(response).to redirect_to(projects_path)
  expect(flash[:alert]).to eql("The project you were looking " +
                               "for could not be found.")
end
```

You use the same error message from the missing-project spec because you don't want to acknowledge to unauthorized users that the project they're looking for exists when they don't have permission to read it.

Now re-run `bin/rspec spec/controllers/projects_controller_spec.rb`:

```
Failure/Error: response.should redirect_to(projects_path)
  Expected response to be a <redirect>, but was <200>
```

The spec fails because you haven't yet scoped down the `Project.find` call in the `set_project` method, which is called using a `before_action` in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`ProjectsController`.

With a failing spec testing the nonexistent behavior, open app/controllers/projects_controller.rb and change the `set_project` method to look for projects the current user has access to so your spec will pass. But there's one problem: you're not restricting the `show` action to only users who are signed in.

You must make it so that the user has to sign in before accessing the `show` action, because you need to use the `current_user` method to check what permissions this user has access to in the `set_project` method. To do so, call the `require_signin!` method as a `before_action` in this controller, as you did for certain actions in `TicketsController`. Place this method above `set_project` to ensure that a user is authenticated before `set_project` does its job. The filters in `ProjectsController` should now look like this.

Listing 8.4 app/controllers/projects_controller.rb

```
before_action :authorize_admin!, except: [:index, :show]
before_action :require_signin!, only: [:show]
before_action :set_project, only: [:show, :edit, :update, :destroy]
```

Alter the `set_project` method to check the permissions of the project before letting authorized users see it or refusing unauthorized users access. Change the line that defines the `@project` variable from this

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

to this:

```
@project = Project.viewable_by(current_user).find(params[:id])
```

The `viewable_by` method doesn't exist yet; you'll define it in a moment. It will return a scope of only the projects the user is allowed to view. This scope has

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

exactly the same methods as an Active Record class, so you can treat it like one.

You can define this method using the `scope` class method in your `Project` model.

The `scope` method provides a method you can call on your class or on an association collection for this class that returns a subset of records. The following `scope` call, for example, defines a method called `admins`:

```
scope :admins, ->{ where(:admin => true) }
```

If you wanted to, you could call this `admins` method on your `User` model to return all the users who are admins:

```
User.admins
```

As you can see, manually specifying `where` isn't nearly as pretty as calling `User.admins`. This may seem like a contrived example, but trust us: it gets ugly when the conditions become more complex. Scopes are yet another great example of the DRY (Don't Repeat Yourself) convention seen throughout Rails. Because the `scope` method defines your scope's logic in one central location, you can easily change all uses of this scope by changing it in this one spot.

Scopes are also chainable. Imagine that you had another `scope` defined on your `User` model, such as the following:

```
scope :by_name, -> { order(:name) }
```

You can call this scope by itself

```
User.by_name
```

which returns all of your users, ordered by their name; no surprises there. The real magic comes when you chain both the `admins` and `by_name` scopes

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

together, like this

```
User.admins.by_name
```

or this:

```
User.by_name.admins
```

Rails builds up the queries by applying the scopes one at a time. Calling them in any order will result in the same query.

Let's define a `scope` now, along with the `permissions` association it needs to use. Put this scope under the validation lines in the `Project` model:

```
                                                    (1)
has_many :permissions, as: :thing

scope :viewable_by, ->(user) do
  joins(:permissions).where(permissions: { action: "view",
                                           user_id: user.id })
end
```

The `:as` option on the `has_many :permissions` association(1) links your projects to the `thing` association on the `Permission` objects. You need this association defined here because it's used by the `scope` below it.

The `joins` method joins the `permissions` table using a SQL `INNER JOIN`, allowing you to perform queries on columns from that table too. You do that with the `where` method, specifying a hash that contains the `permissions` key, which points to another hash containing the fields you want to search on and their expected values. This scope then returns all the `Project` objects containing a related record in the `permissions` table that has the `action` field set to `view` and the user ID equal to that of the passed-in user.

With this `scope` method in place, when you run this spec file again with `bin/rspec spec/controllers/projects_controller_spec.rb`, your tests (almost!) pass because you're scoping down the `find` in the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`set_project` method:

```
$ bin/rspec spec/controllers/projects_controller_spec.rb
FF.....

Failures:

  1) ProjectsController displays an error for a missing project
     Failure/Error: expect(response).to redirect_to(projects_path)
       Expected response to be a redirect to <http://test.host/projects>
         but was a redirect to <http://test.host/signin>.
       Expected "http://test.host/projects"
         to be === "http://test.host/signin".

  2) ProjectsController cannot access the show action without permission
     Failure/Error: expect(response).to redirect_to(projects_path)
       Expected response to be a redirect to <http://test.host/projects>
         but was a redirect to <http://test.host/signin>.
       Expected "http://test.host/projects"
         to be === "http://test.host/signin".
```

This failure was surprising to us. Then, by reading the error message, we realized something: “displays an error for a missing project” is an older test, and we added the requirement for a user to be signed in. To fix this, move the `before` block that contains your sign-in logic outside of the `context` and put it right after the `let`. Here’s our entire spec/controllers/projects_controller_spec.rb:

```
require 'spec_helper'

describe ProjectsController do
  let(:user) { FactoryGirl.create(:user) }
  before do
    sign_in(user)
  end

  it "displays an error for a missing project" do
    get :show, id: "not-here"

    expect(response).to redirect_to(projects_path)
    message = "The project you were looking for could not be found."

    expect(flash[:alert]).to eql(message)
  end

  context "standard users" do
    { new: :get,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      create: :post,
      edit: :get,
      update: :put,
      destroy: :delete }.each do |action, method|

      it "cannot access the #{action} action" do
        sign_in(user)

        send(method, action, :id => FactoryGirl.create(:project))

        expect(response).to redirect_to(root_path)
        expect(flash[:alert]).to eql("You must be an admin to do that.")
      end
    end
  end

  it "cannot access the show action without permission" do
    project = FactoryGirl.create(:project)
    get :show, id: project.id

    expect(response).to redirect_to(projects_path)
    expect(flash[:alert]).to eql("The project you were looking for
                                  could not be found.")
  end
end
```

Try running this one more time with `bin/rspec spec/controllers/projects_controller_spec.rb`:

```
7 examples, 0 failures
```

All right! The examples for this controller are now passing, but how about the feature—the one you wrote previously to test that users *can* access this `show` action if they have the correct permissions? This spec tested the negative, making sure a user without permission *can't* access this project.

With the code you just implemented, this feature should still pass as it did the last time you ran it. Let's find out by running `bin/rspec spec/features/viewing_projects_spec.rb`:

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Isn't that wonderful? You rewrote the feature, and it still passed! You've tested both the granted and denied facets of this particular permission by writing a feature and spec, respectively.

Now that you've implemented that little chunk of functionality and everything seems to be going smoothly, let's make sure the entire application is going the same way by running `bin/rspec`. Oh dear! You broke just about every feature in some way:

```
rspec ./spec/features/creating_tickets_spec.rb:24
rspec ./spec/features/creating_tickets_spec.rb:36
rspec ./spec/features/creating_tickets_spec.rb:44
rspec ./spec/features/hidden_links_spec.rb:50
rspec ./spec/features/hidden_links_spec.rb:55
rspec ./spec/features/editing_tickets_spec.rb:22
rspec ./spec/features/editing_tickets_spec.rb:35
rspec ./spec/features/editing_projects_spec.rb:14
rspec ./spec/features/editing_projects_spec.rb:21
rspec ./spec/features/viewing_tickets_spec.rb:25
rspec ./spec/features/deleting_projects_spec.rb:8
rspec ./spec/features/deleting_tickets_spec.rb:20
rspec ./spec/features/creating_projects_spec.rb:11
```

These features are all broken because you restricted the permissions on the `set_project` method, and these features depend on this functionality in one way or another. Let's fix them, from the top, one at a time.

8.3 Fixing what you broke

Currently, you have a whole bundle of features that are failing. When this happens, it may look like everything's broken (and maybe some things are on fire), but in reality it's not as bad as it seems. The best way to fix a mess like this is to break it into smaller chunks and tackle it one chunk at a time. The output from `bin/rspec` provided a list of the broken features: they're your chunks. Let's go through them and fix them, starting with the Creating Projects feature.

8.3.1 Fixing the Creating Projects feature

When you run `bin/rspec spec/features/creating_projects_spec.rb`, it fails because Capybara can't see the created-project flash message on the page:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
expected there to be content "Project has been created." in ...
```

If you look closely at the actual output Capybara shows you, you'll see that it contains a different message instead: "The project you were looking for could not be found." This is happening due to the way you've restricted visibility on the projects to only users with permission to view them. This restriction isn't taking into account that the user is signed in as an admin user and therefore should be able to see all projects, regardless of whether they have the `Permission` record defined.

The problem lies in the `set_project` method in `ProjectsController`. This method should not restrict the finding of projects when the user is an admin. To fix this problem, change how the `@project` variable is defined in the `set_project` method in app/controllers/projects_controller.rb from this

```
@project = Project.viewable_by(current_user).find(params[:id])
```

to this:

```
@project = if current_user.admin?
  Project.find(params[:id])
else
  Project.viewable_by(current_user).find(params[:id])
end
```

As you can see, this code won't scope the find using the `viewable_by` scope if the user is an admin, but it will if the user isn't. When you run `bin/rspec spec/features/creating_projects_spec.rb`, it should now pass:

```
2 examples, 0 failures
```

This change should fix a couple of other features projects as well, so rerun `bin/rspec` to find the ones that are still broken. You have a much shorter list

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

now:

```
rspec ./spec/features/deleting_tickets_spec.rb:20
rspec ./spec/features/creating_tickets_spec.rb:24
rspec ./spec/features/creating_tickets_spec.rb:44
rspec ./spec/features/creating_tickets_spec.rb:36
rspec ./spec/features/editing_tickets_spec.rb:35
rspec ./spec/features/editing_tickets_spec.rb:22
rspec ./spec/features/viewing_tickets_spec.rb:25
```

You reduced your failing scenarios from 13 to 7 (across 4 features), which is pretty good. Let's fix the first of these, the Creating Tickets feature.

8.3.2 Fixing the four failing features

Run the first feature with `bin/rspec spec/features/creating_tickets_spec.rb`. You'll see that it can't find the New Ticket link.

```
Failure/Error: click_link "New Ticket"
Capybara::ElementNotFound:
  Unable to find link "New Ticket"
```

This is the same problem as before: the user doesn't have permission to access that project. To fix this issue, you need to alter the beginning of the `before` block in spec/features/creating_tickets_spec.rb. You'll Change it so that permission to view the project is granted to the user. Replace these lines in this file

```
project = FactoryGirl.create(:project)
user = FactoryGirl.create(:user)
@email = user.email
```

with these lines:

```
project = FactoryGirl.create(:project)
user = FactoryGirl.create(:user)
define_permission!(user, "view", project)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
@email = user.email
sign_in_as!(user)
```

With these revisions, you change the first line to assign a new local variable called `project` so that it can be referenced later. On the second line, you swap out the two lines that previously created a new user and then confirmed them, replacing them with a single line that creates a new confirmed user. On the third line, you define a new permission for the user to view the project.

Because you're now signing in as a user, you no longer need these lines in the `before` block of this feature, so you should remove them:

```
message = "You need to sign in or sign up before continuing."
expect(page).to have_content(message)

fill_in "Email", with: "ticketee@example.com"
fill_in "Password", with: "password"
click_button "Sign in"
click_link project.name
click_link "New ticket"
```

All the pieces are in place now for this feature to work. When you run it again with `bin/rspec spec/features/creating_tickets_spec.rb`, all the scenarios should pass:

```
3 examples, 0 failures
```

One down, three to go. The next failing feature is Deleting Tickets.

It fails for the same reason as the previous feature: the user doesn't have access to the project to delete a ticket. Let's fix this now by putting the following line at the top of the `before` block in this feature:

```
before do
  define_permission!(user, "view", project)
  ...
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

That's a little too easy! When you run `bin/rspec spec/features/deleting_tickets_spec.rb`, this feature passes once again:

```
1 example, 0 failures
```

Next up is the Editing Tickets feature, which contains not one but two broken scenarios. The two scenarios in this feature, similarly to the Editing Projects scenario, are broken because the feature can't find a link:

```
Capybara::ElementNotFound:
  Unable to find link "Example ticket"
```

Again, the error occurs because the user doesn't have permission to access this particular project. You must specify that this user has access to this project in `before`, as you did for the Creating Tickets and Editing Tickets features. Add this line directly under the line that creates the project in `before`:

```
before do
  define_permission!(user, "view", project)
  ...
```

When you run `bin/rspec spec/features/editing_tickets_spec.rb`, both scenarios should pass:

```
2 examples, 0 failure
```

Great! You fixed another feature. The one remaining feature that fails is Viewing Tickets, which you fix the same way you fixed the previous features. Add this line again under where you create the TextMate 2 project, this time in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

spec/features/viewing_tickets_spec.rb:

```
define_permission!(user, "view", textmate_2)
```

You also need to add one for the Internet Explorer project:

```
define_permission!(user, "view", internet_explorer)
```

To sign in, add this line above the `visit` call in the `before` block:

```
sign_in_as!(user)
```

Running `bin/rspec spec/features/viewing_tickets_spec.rb`, you see that this feature is passing:

```
1 example, 0 failures
```

That was fast! All four failing features are fixed. Well, so you hope. You independently verified them, but run `bin/rspec` to make sure nothing else is broken:

```
46 examples, 0 failures, 1 pending
```

There's one pending spec from spec/models/permission_spec.rb. You can delete this file now, and then when you re-run `bin/rspec`, you'll see this output:

```
Pending:
  Permission add some examples to (or delete) ...
    # No reason given
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    # ./spec/models/permission_spec.rb:4

Finished in 2.64 seconds
45 examples, 0 failures, 1 pending
```

Ah. No reason to keep an empty spec around. Get rid of it with `rm spec/models/permission_spec.rb`, and then run your tests (with `bin/rspec`) one more time:

```
45 examples, 0 failures
```

Great! Everything's working again! Let's commit that:

```
$ git add .
$ git commit -m "Make projects only visible to users with
permission to see them"
$ git push
```

In these first two sections, you added the restriction on the `ProjectsController` that projects should be accessible only to users with `Permission` records with the action set to `view`. In the process, you broke a couple of features, but fixing them was simple.

But these changes only protect the actions in `ProjectsController` that use the `set_project` method and *not* those in `TicketsController`. Before you make changes in `TicketsController`, however, the links to all projects are still visible to all users through `ProjectsController`'s `index`, which is definitely something you should fix first.

8.3.3 One more thing

As described previously, the links to all projects are still visible to all users on the homepage. The way to fix it is to write a new scenario to test that this behavior is always present. You don't have to write the entire scenario, because you have a scenario you can *modify*, instead: the one in spec/features/viewing_projects_spec.rb.

To test that the links are hidden on the `index` action, change the first part of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

viewing_projects_spec.rb to look like this:

```
FactoryGirl.create(:project, name: "Hidden")
visit '/'
expect(page).to_not have_content("Hidden")
click_link project.name
```

This feature will now ensure that the user who doesn't have permission to view the first project no longer can see the second project. When you run this feature using `bin/rspec spec/features/viewing_projects_spec.rb`, it fails as expected:

```
Failure/Error: expect(page).to_not have_content("Hidden")
  expected there not to be text "Hidden" in...
```

To fix it, open app/controllers/projects_controller.rb and modify the `index` action to do exactly what the `set_project` method does: restrict the list of visible projects to just the ones the user has permission to view. You could reuse the code from the `set_project` method in the `index` action, but that isn't very DRY. Instead, extract the code from `set_project` and move it into the `Project` model. Take the code from the `set_project` method

```
@project = if current_user.admin?
  Project.find(params[:id])
else
  Project.viewable_by(current_user).find(params[:id])
end
```

and change it to this much shorter version:

```
@project = Project.for(current_user).find(params[:id])
```

The model is a better place than the controller for this logic because it's

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

operating on data from the database. Open app/models/project.rb, and define this new `for` class method using the following code. You'll also refactor the method down a smidgeon:

```
scope :for, ->(user) do
  user.admin? ? Project.all : Project.viewable_by(user)
end
```

The first line of this method uses a *ternary statement*, which is a shorter version of this:

```
if current_user.admin?
  Project.all
else
  Project.viewable_by(current_user)
end
```

This statement is useful when you have short conditional statements like this one, but it shouldn't be (ab)used for longer conditional statements. As a general rule of thumb, if the line for a ternary statement is longer than 80 characters, it's probably best to split it out over multiple lines for better readability.

In the `set_project` method, you can call `find` on what this new `for` method returns. And in the `index` method, you can use it in an identical fashion, like this:

```
def index
  @projects = Project.for(current_user)
end
```

Because you're referencing `current_user` in this action, you must modify the `before_action` line that references `require_signin!` to ensure that users are signed in before they visit this page. Let's change it to this:

```
before_action :require_signin!, only: [:index, :show]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you run the feature again with `bin/rspec spec/features/viewing_projects_spec.rb`, it passes:

```
1 example, 0 failures
```

Ensure that everything is working as it should by running `bin/rspec`. Oops! You broke one of the scenarios in the Signing Up feature, as shown by this output from the command you just ran:

```
Failure/Error: expect(page).to have_content("You have signed up...
  expected there to be text "You have signed up successfully." in...
```

You've fixed every other feature, but now the Signing Up feature is misbehaving.

Let's see why. When users sign up to your application, they're shown the "Please open the link to activate your account" message, as the scenario says they should; and they're also redirected to the root of your application. The problem lies with this final step: people are redirected to the root of the application, the `ProjectsController`'s `index` action, which is now locked down to require that users be authenticated before they can view this action. This is problematic, but it's fixable.

8.3.4 Fixing the Signing Up feature

The Signing Up feature is broken, and the problem lies solely with the latest changes you made to `ProjectsController`. When users sign up, they're sent to the `root_path` in the application, which resolves to the `index` action in `ProjectsController`. This controller has the `require_login!` method called before all actions in it, checking whether users are authenticated. If they aren't, they're redirected to the sign-in page.

You can see all this in action if you start your server using `rails server` and attempt to sign up. Rather than being properly shown the "Please open the link to activate your account" message, you'll see the Sign In page, as shown in figure

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

8.3.

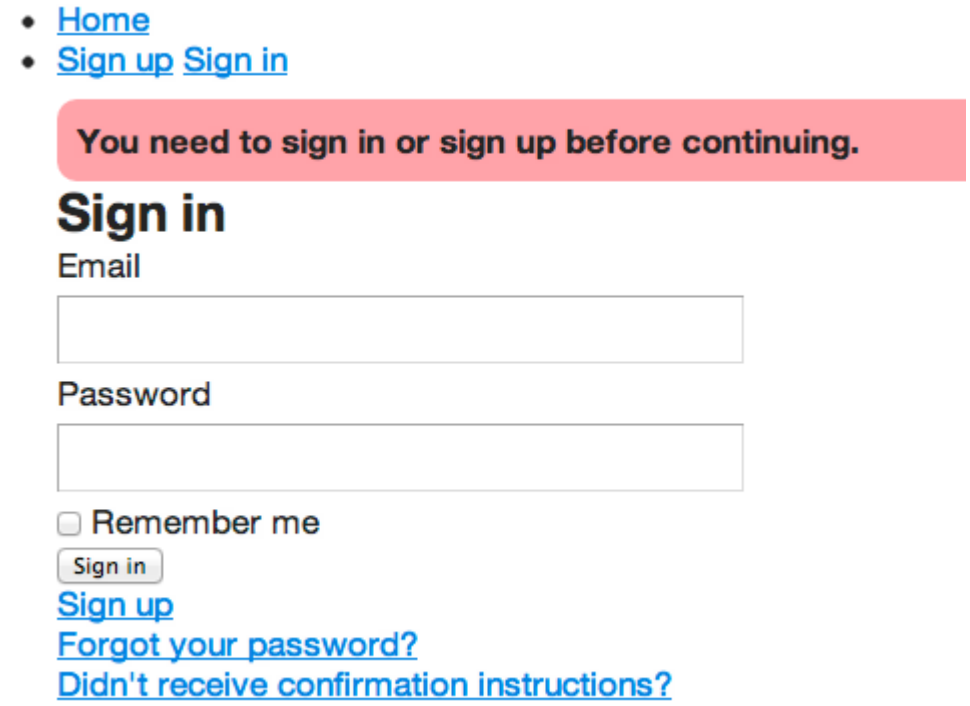

Figure 8.3 Sign In page

Lucky for you, this is easy to fix! You need to make sure you not only make a new account, but also sign in the user. Add a line between these two in app/controllers/users_controller.rb, the `create` action:

```
if @user.save
  session[:user_id] = @user.id
  flash[:notice] = "You have signed up successfully."
```

By putting the user's `id` in the session, you've logged them in. The Signing Up feature is probably fixed now, but the only true way to make sure it's working is to test it manually or run the feature. Running the feature is easier, so let's do that with `bin/rspec`:

```
45 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Everything is green. Awesome! This feature is passing again. Let's make a commit for that:

```
$ git add .
$ git commit -m "Don't show projects that a
 user doesn't have permission to see"
$ git push
```

You've limited the ability of users to take action on things in `ProjectsController` and fixed the Signing Up feature that broke because of the changes you made. But you haven't protected `TicketsController`. This is a problem because users who can't view a project will still be able to view its tickets, which could pose a security risk. A project's most vital assets (for now) are the tickets associated with it, and users who don't have permission to see the project shouldn't be able to see the associated tickets. Let's address this restriction next.

8.4 Blocking access to tickets

When implementing permissions, you have to be careful to ensure that all users who should have access to something do, and all users who *shouldn't* have access to something don't. All of `TicketsController`'s actions are still available to all users because it has no permission checking. If you leave it in that state, users who are unable to see the project can still make requests to the actions in `TicketsController`. They shouldn't be able to do anything to the tickets in a project if they don't have permission to view tickets for it. Let's implement permission-checking to remedy this problem.

To prevent users from seeing tickets in a project they're unauthorized to see, you must lock down the `show` action of `TicketsController`. To test that when you put this restriction in place, it's correct, write a spec in the spec/controllers/tickets_controller_spec.rb file, as you did for `ProjectsController`. Make it look like the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.5 spec/controllers/tickets_controller_spec.rb

```
require 'spec_helper'

describe TicketsController do
  let(:user) { FactoryGirl.create(:user) }
  let(:project) { FactoryGirl.create(:project) }
  let(:ticket) { FactoryGirl.create(:ticket,
                                    project: project,
                                    user: user) }

  context "standard users" do
    it "cannot access a ticket for a project" do
      sign_in(user)
      get :show, :id => ticket.id, :project_id => project.id

      expect(response).to redirect_to(root_path)
      expect(flash[:alert]).to eql("The project you were looking " +
                                   "for could not be found.")
    end
  end
end
```

This test sets up a project, a ticket, and a user who has no explicit permission to view the project and therefore shouldn't be able to view the ticket. You test this spec by signing in as the unauthorized user and trying to go to the `show` action for the ticket, which requires you to pass a `project_id` to help it find what project the ticket is in. The test should pass if the user is redirected to the `root_path` and if, upon the user seeing the `flash[:alert]`, the application denies all knowledge of this project ever having existed.

When you run this test using `bin/rspec spec/controllers/tickets_controller_spec.rb`, it fails because the user can still access this action:

```
Failure/Error: response.should redirect_to(root_path)
 Expected response to be a <redirect>, but was <200>
```

With this test failing correctly, you can work on restricting access to only the projects the user has access to. Open app/controllers/tickets_controller.rb and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

remove the `:except` option from the `require_signin!` filter, so it goes from this

```
before_action :require_signin!, :except => [:index, :show]
```

to this:

```
before_action :require_signin!
```

Now users should always be asked to sign in before accessing the `index` and `show` actions for this controller, meaning `current_user` will always return a `User` object.

You can reference the `current_user` method in `set_project` and use the `for` method to limit the project `find` scope to only the projects to which that user has access. Change the `set_project` method to the following example:

```
def set_project
  @project = Project.for(current_user).find(params[:project_id])
rescue ActiveRecord::RecordNotFound
  flash[:alert] = "The project you were looking " +
                  "for could not be found."
  redirect_to root_path
end
```

The rewritten `set_project` method retrieves a `Project` only if `current_user` has permission to view that project or is an admin. Otherwise, an `ActiveRecord::RecordNotFound` exception will be thrown and rescued, showing users “The project you were looking for could not be found.”

When you run the spec again with `bin/rspec spec/controllers/tickets_controller_spec.rb`, it passes because this user can no longer see this project and is shown the error:

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You scoped the project find for `TicketsController` the same way you did for `ProjectsController`, limiting it to only those projects to which the current user has access.

That's the end of that! Now ensure that all your specs and features are passing by running `bin/rspec`. You should see this output:

```
46 examples, 0 failures
```

In this section, you altered `TicketsController` so that only users with permission to access a project can see the tickets in it. Let's commit that:

```
$ git add .
$ git commit -m "Restrict reading tickets to correct project scope"
$ git push
```

Next you'll add a new permission that restricts who can create tickets in a project.

8.5 Restricting write access

Sometimes, when working on a project, you'll want to limit the creation of tickets to a certain person or a group of people, such as to only developers or only clients. For this, you want the New Ticket link to be hidden from people who don't have this permission, and you need both the `new` and `create` actions to reject such users.

You're lucky to already have the feature for creating tickets; you need to add a step to the `before` block declaring that the user can create tickets in the project. Place this step directly under the one declaring that the user can view the project. Open spec/features/creating_tickets_spec.rb, and modify the `before` so it contains these two lines:

```
define_permission!(user, "view", project)
define_permission!(user, "create tickets", project)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
```

With this permission defined, run `bin/rspec spec/features/creating_tickets_spec.rb`, and the entire feature passes:

```
3 examples, 0 failures
```

This feature passes regardless of whether the user has permission to create a ticket. You're basically in the same situation you faced with the Viewing Tickets feature: the feature would pass either way. So, like before, you can use a controller test to test that users can't create a ticket if they don't have permission to do so.

8.5.1 Blocking creation

Let's write the specs to test that users *with* permission to view the project but *without* permission to create tickets can't create tickets. Put the specs shown in the following listing in spec/controllers/tickets_controller_spec.rb in the standard users `context` block so all the examples are grouped nicely.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.6 spec/controllers/tickets_controller_spec.rb

```
context "with permission to view the project" do
  before do
    sign_in(user)
    define_permission!(user, "view", project)
  end

  def cannot_create_tickets!
    response.should redirect_to(project)
    message = "You cannot create tickets on this project."
    flash[:alert].should eql(message)
  end

  it "cannot begin to create a ticket" do
    get :new, project_id: project.id
    cannot_create_tickets!
  end

  it "cannot create a ticket without permission" do
    post :create, project_id: project.id
    cannot_create_tickets!
  end
end
```

You first set up the specs using a `before`, signing in as a user, and defining a permission for that user to view the project. Next, you define a method called `cannot_create_tickets!`, asserting that unauthorized users should be redirected to the project and shown an alert stating that they're not allowed to create tickets. Rather than duplicating these two lines in each spec where you want to check that a user receives the correct message, you call the `cannot_create_tickets!` method in that place. The two examples you just added ensure that unauthorized visitors to the `new` and `create` actions can't create tickets.

When you run this file with `bin/rspec spec/controllers/tickets_controller_spec.rb`, the specs fail, as you might expect:

```
Failure/Error: response.should redirect_to(project)
  Expected response to be a <redirect>, but was <200>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To make the spec pass, you need to implement the permission-checking on the `new` and `create` actions in `TicketsController`. To do this, place a `before_action` that runs before the `new` and `create` actions and that checks whether the current user has permission. If the user doesn't have permission, then this should redirect the user to `ProjectsController`'s `show` action and show the "You cannot create tickets on this project." error message.

Change the `before_action` calls at the top of app/controllers/tickets_controller.rb to include this new one to restrict the permissions:

```
before_action :require_signin!
before_action :set_project
before_action :set_ticket, only: [:show, :edit, :update, :destroy]
before_action :authorize_create!, only: [:new, :create]
```

This `authorize_create! before_action` is placed after the `authorize_admin!` and `set_project` `before_actions` because it uses the `current_user` object set up by the `authorize_admin!` method and the `@project` object from `set_project`.

Define the `authorize_create!` method like this under the `private` declaration in `TicketsController`:

```
def authorize_create!
  if !current_user.admin? && cannot?("create tickets".to_sym, @project)
    flash[:alert] = "You cannot create tickets on this project."
    redirect_to @project
  end
end
```

In this new filter, you call a new method called `cannot?`, which returns `true` or `false` to indicate whether the currently signed-in user can't or can do a particular action.

In this example, you use `to_sym` to create a symbolized version of a string, which is required because the `cannot?` method takes only symbols. You also check whether the user is an admin; if so, the user should be allowed to create

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

tickets. If you run the controller's spec again with `bin/rspec spec/controllers/tickets_controller_spec.rb`, the example fails because the `cannot?` method is undefined:

```
Failure/Error: get :new, project_id: project.id
undefined method 'cannot?' for #<TicketsController:0xb651244c>
```

Rails doesn't come with a `cannot?` method, but a gem called cancan (stylized as CanCan) does. This gem helps you tie together the `User` and `Permission` records. Let's install it now.

8.5.2 What is CanCan?

CanCan is a gem written by Ryan Bates of RailsCasts fame; it provides some nice helper methods (such as the `cannot?` method and its antithesis, `can?`) to use in controllers and views. The `can?` and `cannot?` methods use the same `permissions` table you created to check that a user has permission to perform a specific action on a specific object.

To install CanCan, add this line to your Gemfile:

```
gem 'cancan', '~> 1.6.10'
```

To install the CanCan gem, run `bin/bundle install`.

8.5.3 Adding abilities

When you run `bin/rspec spec/controllers/tickets_controller_spec.rb`, you get this output:

```
Failure/Error: post :create, :project_id => project.id
NameError:
 uninitialized constant Ability
```

This error occurs because CanCan is now defining the `cannot?` method for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the controller, which uses a CanCan built-in method called `current_ability`:

```
@current_ability ||= ::Ability.new(current_user)
```

`||=` sets `@current_ability` if it isn't already set. The `::` before `Ability` indicates to Ruby that you want the `Ability` at the root namespace. This allows you to have a module called `CanCan::Ability` and a class at `Ability` and to differentiate between the two. This example is trying to access `Ability`, which is a class that doesn't yet exist.

This new `Ability` class will provide the link between users and their permissions. Define it in a new file at app/models/ability.rb, exactly like the following listing.

Listing 8.7 app/models/ability.rb

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user.permissions.each do |permission|
    can permission.action.to_sym,
        permission.thing_type.constantize do |thing|
        thing.nil? ||
        permission.thing_id.nil? ||
        permission.thing_id == thing.id
      end
    end
  end
end
```

The `Ability` class's `initialize` method defines how `can?` and `cannot?` will act. In this example, you iterate over all the users' permissions and use the `can?` method to say that a user can perform a specific function. Users who shouldn't be able to perform that function won't have an entry in the `permissions` table for it. This is the *whitelist authorization* described at the beginning of the chapter.

When you run `bin/rspec spec/controllers/tickets_controller_spec.rb`, you get this error:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: post :create, project_id: project.id
NoMethodError:
  undefined method `permissions' for #<User:0x007fc6232e6210>
```

This error occurs because you haven't yet defined a `has_many` association on the `User` model to the `Permission` model. To do so, open app/models/user.rb and add this line in the class:

```
has_many :permissions
```

With this association in place, run `bin/rspec spec/controllers/tickets_controller_spec.rb`, and the whole spec file passes:

```
3 examples, 0 failures
```

Great! Now that the spec's passing, unauthorized users don't have access to the `new` and `create` actions. How about checking that those who have permissions can access these actions? Let's check the Creating Tickets feature. With this permission-checking in place, any user with the right permissions should still be able to create tickets. Run `bin/rspec spec/features/creating_tickets_spec.rb` to make sure. It should pass:

```
3 examples, 0 failures
```

Good—users without permission to create tickets can no longer do so.

Now that you've implemented this story, it's time to commit. As usual, you should ensure that everything is still working by running `bin/rspec`. Everything should pass:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
48 examples, 0 failures
```

Let’s commit the changes now:

```
$ git add .
$ git commit -m "Restrict creating tickets to only users
                 who have permissions to do it"
$ git push
```

In this section, you limited the creation of tickets to only those users who’re granted permission to do so by using the `Permission` class and the CanCan gem.

8.6 Restricting update access

You just learned how to restrict access to the creation of tickets; your next step is to restrict which users can update tickets. Thankfully, you can reuse the framework that’s already in place with CanCan to make this a cinch. You can also reuse the Editing Tickets feature to test the restriction.

For this feature, at spec/features/editing_tickets_spec.rb, you’ll set up a `Permission` that says the user you sign in as has permission to update tickets. To do this, write a step in the `before` directly under the other one that sets up read access:

```
define_permission!(user, "view", project)
define_permission!(user, "edit tickets", project)
```

When you run `bin/rspec spec/features/editing_tickets_spec.rb`, it all passes, as you expect. This covers the scenario in which the user *has* permission to update tickets; to cover the scenario in which the user doesn’t have permission, you need to write a couple of specs first.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

8.6.1 No updating for you!

In this section, you'll restrict updating of tickets in the same way you restricted creating tickets. You'll start by writing two examples: one to test the `edit` action and the other to test to the `update` action. In spec/controllers/tickets_controller_spec.rb, in the "with permission to view the project" context, define a `cannot_update_tickets!` method right under the `cannot_create_tickets!` method:

```
def cannot_update_tickets!
  expect(response).to redirect_to(project)
  expect(flash[:alert]).to eql("You cannot edit tickets " \
                               "on this project.")
end
```

Then, under the existing examples for ensuring that a user can't create a new ticket, put the specs ensuring that a user can't update a ticket, as shown in the following listing.

Listing 8.8 Update tests for spec/controllers/tickets_controller_spec.rb

```
it "cannot edit a ticket without permission" do
  get :edit, { project_id: project.id, id: ticket.id }
  cannot_update_tickets!
end

it "cannot update a ticket without permission" do
  put :update, { project_id: project.id,
                 id: ticket.id,
                 ticket: {}
               }
  cannot_update_tickets!
end
```

These two examples make requests to their respective actions and assert that the user is redirected away from them with an error message explaining why. With both these actions, you need to pass a `project_id` parameter so the `set_project` method can find a project and an `id` parameter so the `set_ticket` method can find a ticket. For the `update` action, you pass an

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

empty hash so `params[:ticket]` is set. If you didn't do this, you would get a confusing error in your test:

```
NoMethodError:
  undefined method 'stringify_keys' for nil:NilClass
```

This error would occur because the `update` call in the `update` action would be passed `nil`, as that's what `params[:ticket]` defaults to if you don't pass it in here. This error would happen only if the user had permission to update a ticket, which all users have for now (but not for long).

When you run this file using `bin/rspec spec/controllers/tickets_controller_spec.rb`, these two examples fail:

```
1) TicketsController standard users with permission to view
   the project cannot edit a ticket without permission
   Failure/Error: response.should redirect_to(project)
     Expected response to be a <redirect>, but was <200>

2) TicketsController with permission to view the project cannot update a
ticket without permission
     Failure/Error: put :update, { project_id: project.id,
     ActionController::ParameterMissing:
       param not found: ticket
```

Now you can implement this feature in your controller.

8.6.2 Authorizing editing

Before the `edit` and `update` actions are run, you want to check to see whether the user is authorized to run those actions. Write another `before_action` for `TicketsController`: the `before_action` list for this controller should now look like this.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.9 app/controllers/tickets_controller.rb

```
before_action :authorize_admin!
before_action :set_project
before_action :set_ticket, only: [:show, :edit, :update, :destroy]
before_action :authorize_create!, only: [:new, :create]
before_action :authorize_update!, only: [:edit, :update]
```

At the bottom of this controller class, define the new `authorize_update!` method:

```
def authorize_update!
  if !current_user.admin? && cannot?("edit tickets".to_sym, @project)
    flash[:alert] = "You cannot edit tickets on this project."
    redirect_to @project
  end
end
```

Now check whether the specs pass by running `bin/rspec spec/controllers/tickets_controller_spec.rb`:

```
5 examples, 0 failures
```

Wasn't that easy? The `edit` and `update` actions in `TicketsController` are restricted, like the `create` action. How's the feature going? Let's see if users with permission can still update tickets—run `bin/rspec spec/features/editing_tickets_spec.rb`:

```
2 examples, 0 failures
```

Just like that, you're finished restricting updating tickets to only some users.

Now run `bin/rspec` to make sure nothing is broken. Everything should be good:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
50 examples, 0 failures
```

Fantastic! Let's commit that:

```
$ git add .
$ git commit -m "Restrict ticket updating to only
               those who have permission"
$ git push
```

Good stuff. In this section, you learned how to restrict the `edit` and `update` actions using the permissions you implemented earlier. There's one last port of call for this restricting business: the `destroy` action.

8.7 Restricting delete access

The final action you restrict is the `destroy` action in the `TicketsController`. Again, you can reuse a feature to test this behavior: the Deleting Tickets feature.

As you did with the Creating Tickets and Updating Tickets features, you implement a step in the Deleting Tickets feature to give the user permission to delete tickets. Under the line that grants users permission to view the TextMate 2 project, put another one to grant them permission to delete tickets:

```
define_permission!(user, "view", project)
define_permission!(user, "delete tickets", project)
```

When you run this feature with `bin/rspec spec/features/deleting_tickets_spec.rb`, the whole thing passes because you already have the step that supports the different permissions you require:

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This feature ensures that anybody with permission can delete tickets for projects, but you need another spec to test that anybody *without* permission is prevented from deleting tickets.

To ensure that users without permission to delete tickets can't do so, write a spec (shown next) directly under the one for the `update` action in spec/controllers/tickets_controller_spec.rb:

```
it "cannot delete a ticket without permission" do
  delete :destroy, { project_id: project.id, id: ticket.id }

  expect(response).to redirect_to(project)
  message = "You cannot delete tickets from this project."
  expect(flash[:alert]).to eql(message)
end
```

You don't have to put the last two lines in their own method because you won't use them more than once. When you run this spec, it fails on the final line rather than on the third line:

```
1) TicketsController standard users with permission to view the project
   cannot delete a ticket without permission
     Failure/Error: flash[:alert].should eql(message)

     expected: "You cannot delete tickets from this project."
      got: nil
```

This error occurs because the `destroy` action is being processed, and it redirects the user to the project once it's complete. The spec doesn't know the difference between a redirect from within the action or within the `before_action`, nor should it.

To make this spec pass, define a new method called `authorize_delete!` at the bottom of `TicketsController`:

```
def authorize_delete!
  if !current_user.admin? && cannot?(:"delete tickets", @project)
    flash[:alert] = "You cannot delete tickets from this project."
    redirect_to @project
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

Then you can call this method in a `before_action` too:

```
before_action :authorize_delete!, only: :destroy
```

Now, when you run this spec using `bin/rspec spec/controllers/tickets_controller_spec.rb`, it's all passing:

```
6 examples, 0 failures
```

You're stopping users without permission; how goes your feature? Run `bin/rspec spec/features/deleting_tickets_spec.rb` to find out:

```
1 example, 0 failures
```

Great! With this last permission in place, all the actions in `TicketsController` are restricted to their appropriate users. Of course, you need to run all your specs with `bin/rspec`:

```
.................................................

Finished in 2.64 seconds
51 examples, 0 failures
```

Let's make a commit:

```
$ git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ git commit -m "Restrict destroy action to only people with permission"
$ git push
```

Because the controller's actions are restricted, the links associated with these actions should be hidden from users who are unable to perform these actions. Users who aren't able to create, update, or delete tickets shouldn't see the the related links.

8.8 Hiding links based on permission

To ensure that these links are hidden from those who shouldn't be able to see them but are still visible to admins (because admins should be able to do everything), you use spec/features/hidden_links_spec.rb. Start with the New Ticket link by adding this scenario in the "regular users" context:

```
scenario "New ticket link is shown to a user with permission" do
  define_permission!(user, "view", project)
  define_permission!(user, "create tickets", project)
  visit project_path(project)
  assert_link_for "New Ticket"
end
```

When a user has the permission to both view a project and create a ticket for that project, they should be able to see the New Ticket link on the project page. Simple. Here's another scenario, also in the "regular users" context: ensure that a user *without* permission can't see the link:

```
scenario "New ticket link is hidden from a user without permission" do
  define_permission!(user, "view", project)
  visit project_path(project)
  assert_no_link_for "New Ticket"
end
```

In this latest scenario, the user only has permission to view the project and has no permission to create a ticket. If they go to the project page, they shouldn't see a New Ticket link. Now for one more scenario, this time in the "admin users" context, to ensure that admins *can* view this new link, regardless of whether they have permission to:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
scenario "New ticket link is shown to admins" do
  visit project_path(project)
  assert_link_for "New Ticket"
end
```

These three scenarios test all three permutations of users who could possibly see this page. Users with permission and admins should be able to see the link, and users without permission shouldn't. When you run this feature with `bin/rspec spec/features/hidden_links_spec.rb`, the second scenario fails:

```
Failure/Error: assert_no_link_for "New Ticket"
  expected not to find css "a" with text "New Ticket", but...
```

This error occurs because the link is visible independently of whether the user has permission. With these scenarios in place, you can work on making them pass. You can wrap the New Ticket link in a helper method, similar to the `admins_only` helper used in chapter 6. Open app/views/projects/show.html.erb, and change the New Ticket link from this

```
<%= link_to "New Ticket", new_project_ticket_path(@project) %>
```

to this:

```
<% authorized?(:"create tickets", @project) do %>
  <%= link_to "New Ticket", new_project_ticket_path(@project) %>
<% end %>
```

Currently, this `authorized?` method is undefined. This is the method you need in views all across your application to determine whether the user has permission to see the specific action and whether that user is an admin. Because you'll use this helper everywhere, define it in app/helpers/application_helper.rb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def authorized?(permission, thing, &block)
  block.call if can?(permission.to_sym, thing) ||
                        current_user.try(:admin?)
end
```

This helper uses CanCan's `can?` method to check whether the user is authorized to perform this action. If so, then all is fine and dandy. If not, then the method checks to see whether `current_user` is set (it won't be set if the user isn't signed in); and if it is, the method checks to see whether that user is an admin by using the `try` method, which returns `nil`. If the method specified can't be found on `thing`, `try` is called. If it's found, then you use `block.call`, which runs the passed-in block and outputs the content to the view.

With this helper implemented, all three new scenarios should pass. Run `bin/rspec spec/features/hidden_links_spec.rb` to find out:

```
12 examples, 0 failures
```

Great! They're passing. Next let's implement another few for testing the Edit Ticket link for tickets. First up, add a new scenario to the "regular users" context in spec/features/hidden_links_spec.rb:

```
scenario "Edit ticket link is shown to a user with permission" do
  # this scenario needs the ticket created first to set correct
  permissions
  ticket
  define_permission!(user, "view", project)
  define_permission!(user, "edit tickets", project)
  visit project_path(project)
  click_link ticket.title
  assert_link_for "Edit Ticket"
end
```

Just as with the New Ticket link, here you define two permissions: one for the user to view the project and the other so that they can edit tickets in that project. You then go to the project's page and click a ticket's title, and you should see an Edit Ticket link on that page. Write another scenario that checks what happens

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

when the user doesn't have permission to edit tickets, placing it in the "regular users" context also:

```
scenario "Edit ticket link is hidden from a user without permission" do
  ticket
  define_permission!(user, "view", project)
  visit project_path(project)
  click_link ticket.title
  assert_no_link_for "Edit Ticket"
end
```

And then add one more for the admins of the application in the "admin users" context of spec/features/hidden_links_spec.rb, making sure they can always see the Edit Ticket link:

```
scenario "Edit ticket link is shown to admins" do
  ticket
  visit project_path(project)
  click_link ticket.title
  assert_link_for "Edit Ticket"
end
```

You need to call `ticket` in these scenarios as the first step, because if you visited the project before creating the ticket, there would be no link—the ticket wouldn't have been made yet!

When you run these scenarios using `bin/rspec spec/features/hidden_links_spec.rb`, the method `ticket` can't be found for any of them:

```
undefined local variable or method `ticket'
```

This method is *supposed* to return a `Ticket` object that you can use in your test, but it doesn't exist yet. This is pretty easy to fix. Define another `let` under the one for `project` at the top of spec/features/hidden_links_spec.rb, like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
let(:ticket) { FactoryGirl.create(:ticket, project: project,
                                          user: user) }
```

When you run this feature again with `bin/rspec spec/features/hidden_links_spec.rb`, the second scenario fails, as when you implemented the New Ticket link filtering:

```
Failure/Error: assert_no_link_for "Edit Ticket"
  Expected to not see the "Edit Ticket" link, but did.
```

This time, edit the file app/views/tickets/show.html.erb. Change the Edit Ticket link from this

```
<%= link_to "Edit Ticket", [:edit, @project, @ticket] %>
```

to this:

```
<%= authorized?("edit tickets", @project) do %>
  <%= link_to "Edit Ticket", [:edit, @project, @ticket] %>
<% end %>
```

With this one small change to use the `authorized?` helper to check for the permission to edit tickets for the current project, the Hidden Links feature now passes when you run `bin/rspec spec/features/hidden_links_spec.rb`:

```
15 examples, 0 failures
```

Great! You've got one last link to protect now: the Delete Ticket link on the tickets Show page. For this, you'll need to add another three scenarios. The first scenario will ensure that a user with permission can see the Delete Ticket link, and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

it should be placed in the “regular users” context:

```
scenario "Delete ticket link is shown to a user with permission" do
  ticket
  define_permission!(user, "view", project)
  define_permission!(user, "delete tickets", project)
  visit project_path(project)
  click_link ticket.title
  assert_link_for "Delete Ticket"
end
```

The second will ensure that a user without permission can’t see the Delete Ticket link. Also place it in the “regular users” context:

```
scenario "Delete ticket link is hidden from users without permission" do
  ticket
  define_permission!(user, "view", project)
  visit project_path(project)
  click_link ticket.title
  assert_no_link_for "Delete Ticket"
end
```

Finally, one more to check that admins can still see the Delete Ticket link—place it in the “admin users” context block:

```
scenario "Delete ticket link is shown to admins" do
  ticket
  visit project_path(project)
  click_link ticket.title
  assert_link_for "Delete Ticket"
end
```

When you run this feature with `bin/rspec spec/features/hidden_links_spec.rb`, the middle scenario fails again:

```
Failure/Error: assert_no_link_for "Delete Ticket"
expected not to find css "a" with text "Delete Ticket", but...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To fix it, open app/views/tickets/show.html.erb and wrap the Delete Ticket link in the warm embrace of the `authorized?` method, as you did with the Edit Ticket link:

```
<% authorized?("delete tickets", @project) do %>
  <%= link_to "Delete Ticket", [@project, @ticket], method: :delete,
  data: { confirm: "Are you sure you want to delete this ticket?"} %>
<% end %>
```

When you run `bin/rspec spec/features/hidden_links_spec.rb`, all 18 scenarios pass:

```
18 examples, 0 failures
```

Fantastic! Now you've stopped displaying links to the users who shouldn't see them and switched to displaying them only to people who should be able to see them.

What a whirlwind adventure! First you learned to check for permissions for all the actions in `TicketsController`, and then you learned to hide links from users in the views. Let's make sure everything is working by running `bin/rspec`:

```
60 examples, 0 failures
```

Great! Let's commit.

```
$ git add .
$ git commit -m "Restrict actions in TicketsController based
               on permissions and hide links"
$ git push
```

With all that done, you have the scaffold for setting up permissions but no

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

interface to it! There's currently no way in the system for a user (in particular, an admin) to set up the permissions on other users. You'll implement that next.

8.9 Assigning permissions

In chapter 7, you added an `admin` field to the `users` table and then triggered it through the admin backend by checking or unchecking a check box. You'll do the same thing with the permissions for projects. When you're finished, you'll see a permissions screen that allows you to pick and choose the permissions for users and projects.

You'll implement this screen one check box at a time because you must confirm that the permissions you assign through this interface work. Let's get started with the permission to view projects.

8.9.1 Viewing projects

In this section, you'll implement the foundations for assigning permissions through the admin backend, starting with the permission to view projects. Create a new spec/features/admin/assigning_permissions_spec.rb file, and begin with the code from the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.10 spec/features/admin/assigning_permissions_spec.rb

```
require 'spec_helper'

feature "Assigning permissions" do
  let!(:admin) { FactoryGirl.create(:admin_user) }
  let!(:user) { FactoryGirl.create(:user) }
  let!(:project) { FactoryGirl.create(:project) }
  let!(:ticket) { FactoryGirl.create(:ticket, project: project,
                                             user: user) }

  before do
    sign_in_as!(admin)

    click_link "Admin"
    click_link "Users"
    click_link user.email
    click_link "Permissions"
  end

  scenario "Viewing a project" do
    check_permission_box "view", project

    click_button "Update"
    click_link "Sign out"

    sign_in_as!(user)
    expect(page).to have_content(project.name)
  end
end
```

This scenario has two users: an admin user and a standard user. You sign in as the admin user, go to the permissions page, check the view permission for the project, click Update, and then sign out. Then you sign in as the user who was granted permission to test that permission, which you do in the next step. This ensures that the assigning of the permissions always works. For now, you're only testing the permission to view a project permission.

When you run `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, it fails when it tries to follow the Permissions link:

```
Failure/Error: click_link "Permissions"
Capybara::ElementNotFound:
  Unable to find link "Permissions"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

If you look at how the `before` gets to this point, you can see that it follows the Admin link, which leads to the admin dashboard; then goes to Users, to take you to the place where you can see users; and finally clicks a user's email, taking you to the `Admin::UsersController show` action. Therefore, you need to add the missing Permissions link to the app/views/admin/users/show.html.erb directly under the Delete User link:

```
<%= link_to "Permissions", admin_user_permissions_path(@user) %>
```

The path for this `link_to` (which isn't yet defined) takes you to the `Admin::PermissionsController`'s `index` action. To get this `link_to` to work, define that permissions are nested under users in config/routes.rb, and add the `admin` namespace in the definition using this code:

```
namespace :admin do
  root "base#index"
  resources :users do
    resources :permissions
  end
end
```

With these changes in the config/routes.rb file, the `admin_user_permissions_path` used in the `link_to` is now defined. When you run the feature using `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, you see there's more to be done for this step:

```
Failure/Error: click_link "Permissions"
  ActionController::RoutingError:
  uninitialized constant Admin::PermissionsController
```

Ah, of course! You must create the controller for this link.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

THE PERMISSIONS CONTROLLER

You can generate the `Admin::PermissionsController` file by running this command:

```
$ bin/rails g controller admin/permissions
```

Along with an app/controllers/admin/permissions_controller.rb file, this command generates other goodies, such as a helper and a directory for the views at app/views/admin/permissions. Before you go further, you must modify this file to make the class inherit from the right place so that only admins can access it. Open the file, and change the first line to this:

```
class Admin::PermissionsController < Admin::BaseController
```

This line makes the controller inherit from the `Admin::BaseController` class, which restricts all actions in this controller to only admin users.

When you run `bin/rspec spec/features/admin/assigning_permissions_spec.rb` again, you see an undefined method:

```
Failure/Error: check_permission_box "view", project
NoMethodError:
   undefined method `check_permission_box' ...
```

This method is used in the Assigning Permissions feature to check a specific project's permission check box. Unlike the other form elements you've seen previously, the check boxes on this page don't have labels associated with them and therefore can't be targeted that way. Instead, they must be targeted using either their `id` or `name` attribute: `permissions_1_view` or `permissions[1][view]`, respectively. The `id` attribute is slightly less typing, so you'll use that one.

The `check_permission` method uses two arguments in the Assigning

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Permissions feature:

```
  check_permission_box "view", project
```

The first argument is the permission name, in this case *view*. The second is a `Project` object. The method should take these two arguments and then check the related check box: that is, the check box for the *view* permission for that specific project. Define this new method in the `AuthorizationHelpers` module in spec/support/authorization_helpers.rb:

```
def check_permission_box(permission, object)
  check "permissions_#{object.id}_#{permission}"
end
```

You define a method like this rather than calling `check "permission_#{project.id}_view"` in the scenario so it's clear to other people (and your later self) reading the scenario exactly what this step in the scenario is intended to perform. When you run this feature again with `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, it gets a little further:

```
Capybara::ElementNotFound:
  Unable to find checkbox "permissions_1_view"
```

Obviously, you need to define the `index` action and view before you carry on. In this action, load all the permissions for the user you're currently looking at, and then, with the view, display a page from which an admin can choose what permissions this user has on each project. It would be helpful if this user was loaded by a `before_action`, because you'll need it for the action that updates the permissions later. With all this in mind, update the entire controller to resemble the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.11 app/controllers/admin/permissions_controller.rb

```
class Admin::PermissionsController < Admin::BaseController
  before_action :set_user

  def index
    @ability = Ability.new(@user)
    @projects = Project.all
  end

  private

    def set_user
      @user = User.find(params[:user_id])
    end
end
```

The new `Ability` instance created here lets you check the user's ability to perform specific actions on any project by calling this code:

```
@ability.can?(:view, @project)
```

This syntax may look similar to the syntax used in `TicketsController`—it is. In that controller, you used the `cannot?` method, which is the opposite of the `can?` method. These methods are added to the controller by CanCan and are shorter helper methods to do almost exactly what you did in this controller. The only difference is that you're not acting on the `current_user` here, so you must define an `Ability` object yourself and use that instead.

THE PERMISSIONS SCREEN

Now that you have the `index` action up, you need to make its view look like what is shown in figure 8.5. Create a new file at app/views/admin/permissions/index.html.erb, and fill it with the content from the next listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

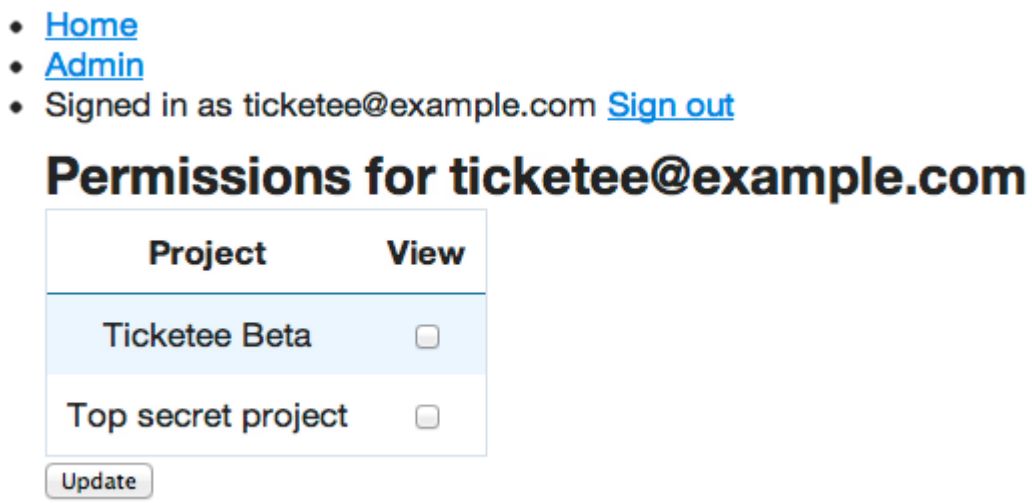

Figure 8.4 The Permissions screen

Listing 8.12 app/views/admin/permissions/index.html.erb

```
<h2>Permissions for <%= @user.email %></h2>
<%= form_tag admin_user_set_permissions_path, :method => :put do %>
  <table id='permissions' cellspacing='0'>
    <thead>
      <th>Project</th>
      <% permissions.each do |name, text| %>
        <th><%= text %></th>
      <% end %>
    </thead>
    <tbody>
      <% @projects.each do |project| %>
        <tr class='<%= cycle("odd", "even") %>'
            id='project_<%= project.id %>'>
          <td><%= project.name %></td>
          <% permissions.each do |name, text| %>
            <td>
              <%= check_box_tag "permissions[#{project.id}][#{name}]",
                @ability.can?(name.to_sym, project),
                @ability.can?(name.to_sym, project) %>
            </td>
          <% end %>
        </tr>
      <% end %>
    </tbody>
  </table>
  <%= submit_tag "Update" %>
<% end %>
```

This template defines the table discussed earlier. It provides a header row of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

permission titles, and rows for each project containing that project's name followed by check boxes for each of the permissions. In this view, you use the `form_tag`, which generates a form that points to the path specified by its first argument. You also use a `permissions` helper method, which isn't currently defined but will provide a list of permissions to iterate through to check on each project.

Right under where you use `permissions`, you use another helper method called `cycle`. This method is built in to Rails and cycles through its given arguments for each iteration of whatever it's encapsulated in; so when this page displays the first project, the method sets the class of that `tr` tag to `odd` and the second one to `even`. It cycles between these two classes until it runs out of projects to iterate through. This is how you can easily get different rows in this table to be styled differently. On the same line, you set the `id` attribute for the `tr` tag to be `project_[a project's id]`. This is so Capybara can locate the correct check box.

Before we look at how to define the `permissions` method, run `bin/rspec spec/features/admin/assigning_permissions_spec.rb`:

```
Failure/Error: click_link "Permissions"
  ActionView::Template::Error:
  undefined local variable or method
    `admin_user_set_permissions_path' for ...
```

You get an error because you haven't yet defined the route for the form. `Admin::PermissionsController` serves a different purpose than the standard REST controllers. For this controller, you use the `update` action to update a slew of permissions rather than a single one. To map to this action by using the `update` method, you must define another *named route* in your config/routes.rb file using the `put` method:

```
namespace :admin do
  root to: "base#index"
  resources :users do
    resources :permissions

    put "permissions", to: "permissions#set",
                       as: "set_permissions"
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

With this method, you define a new route for your application that will only respond to `PUT` requests to this route. The controller and action are defined using the `:to` symbol, and finally the method itself is given a name with the `:as` option.

When you run the feature again, this route method is defined, but the `permissions` method isn't:

```
And I follow "Permissions"
undefined local variable or method 'permissions' [...]
```

Great! It seems as though your page just requires this `permissions` helper method.

DEFINING A HELPER METHOD

Back in chapter 7, you defined a helper method called `admins_only` in `ApplicationHelper`, which allowed you to show links only for admin users. This time, you define the `permissions` method, which contains a list of permissions for which to display check boxes on this page. Because this method is specific to views from the `Admin::PermissionsController` controller, place it in app/helpers/admin/permissions_helper.rb and define it as shown in the following listing.

Listing 8.13 app/helpers/admin/permissions_helper.rb

```
module Admin::PermissionsHelper
  def permissions
    {
      "view" => "View"
    }
  end
end
```

This `permissions` method returns a hash containing only one key-value pair at the moment because you're testing only one particular check box. You use this method to display all the permissions you want to be configurable by admins, and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you revisit this method later to define more pairs. You use this method in your view twice; the first time, you iterate over it like this:

```
<% permissions.each do |name, text| %>
  <th><%= text %></th>
<% end %>
```

When you iterate over a `Hash` object with the `each` method, the key for the hash becomes the first block variable and the value becomes the second block variable; these variables change for each key-value pair of the `Hash` object. In this case, it renders headers for the table in this view. You use this helper later in the view too:

```
<% permissions.each do |name, text| %>
  <td>
    <%= check_box_tag "permissions[#{project.id}][#{name}]",
      1,
      @ability.can?(name.to_sym, project) %>
  </td>
<% end %>
```

Here you use just the key from the hash to define a uniquely identifiable name for this check box. The second argument is the value returned to the controller, which you use to determine whether this check box is selected. The third argument uses the `@ability` object to determine whether this check box is displayed as checked. By using this method, you get a tag like this:

```
<input id=\"permissions_1_view\"
  name=\"permissions[1][view]\"
  type=\"checkbox\"
  value=\"1\" />
```

You're given both the `id` and `name` attributes, which are generated from the first argument you passed to `check_box_tag`. The `id` attribute indicates not the permission's ID but the ID of the project you're determining the permission is for.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You'll use the `id` attribute shortly to select this check box using Capybara and the parsed-into-`params` version of the `name` attribute after that in your controller:

```
Failure/Error: click_button "Update"
AbstractController::ActionNotFound:
The action 'set' could not be found for Admin::PermissionsController
```

This action is a little tricky. Not only do you want it to add permissions for users, you also want to delete those permissions.[2] This action receives `params[:permissions]` in this `Hash` format from the form:

Footnote 2 Which is possibly why the action is called `set`.

```
{"1"=>{"view"=>"1"}}
```

The first key is the ID of the project, and the hash in contains the permissions for that project. If no check boxes are selected for that project, then no hash exists in `params[:permissions]` for it. Therefore, you use this hash to update the permissions that a user can do now, as shown next.

Listing 8.14 `set` action in app/controllers/admin/permissions_controller.rb

```
def set
  @user.permissions.clear
  params[:permissions].each do |id, permissions|
    project = Project.find(id)
    permissions.each do |permission, checked|
      Permission.create!(user: @user,
                         thing: project,
                         action: permission)
    end
  end
  flash[:notice] = "Permissions updated."
  redirect_to admin_user_permissions_path(@user)
end
```

You first clear all the user's permissions using the association method `clear`. Next, you iterate through all the key-value pairs in `params[:permissions]`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

and find the project for each one. Then you iterate through the `permissions` for the parameter and create a new permission for every project. Finally, you set a `flash[:notice]` and redirect back to the permissions page. When you run this feature, you'll see that the Sign Out link is missing:

```
no link with title, id or text 'Sign out' found ...
```

You didn't add this link in chapter 7 because you didn't need it, but in hindsight, you should have. Add this link now to app/views/layouts/application.html.erb, directly under the `Signed in as` text:

```
Signed in as <%= current_user.email %>
<%= link_to "Sign out", signout_path, method: :delete %>
```

This link now appears only to people who are signed in. When you re-run the Assigning Permissions feature with `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, everything should pass, but…

```
Failure/Error: sign_in_as!(admin)
ActionView::Template::Error:
 undefined local variable or method `signout_path' for...
```

Oh yeah! You haven't implemented sign out yet. Don't worry: it's very, very simple. All you have to do to log someone out is remove their ID from the session. Easy. First, though, let's fix that route to config/routes.rb:

```
delete "/signout", to: "sessions#destroy", as: "signout"
```

And implement the `destroy` action in app/controllers/sessions_controller.rb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def destroy
  session[:user_id] = nil
  flash[:notice] = "Signed out successfully."

  redirect_to root_url
end
```

OK, one last time, run `bin/rspec spec/features/admin/assigning_permissions_spec.rb`:

```
1 example, 0 failures
```

Great! You created a way for admins to choose which users can see which projects through an interface of check boxes and confirmed that users can see the project they have access to and can't see the projects they aren't authorized to see. Let's run all the tests with `bin/rspec` to make sure everything is working:

```
62 examples, 0 failures, 1 pending
```

The one pending test lives at spec/helpers/admin/permissions_helper_spec.rb. This file can be deleted because it doesn't contain any useful tests. Once it's deleted, another run of `bin/rspec` shows this result:

```
61 examples, 0 failures
```

All systems green! Let's make a commit before you go any further:

```
$ git add .
$ git commit -m "Add permissions screen for admins"
$ git push
```

Next, you'll implement this feature for the other permissions used.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

8.9.2 And the rest

With the foundation in place for this check box screen, you can add the rest of the permissions you've implemented. The next permission you implemented after the restriction of read access was the restriction of write access, restricting which users could and couldn't perform the `new` and `create` actions on `TicketsController`. Now that you have an interface for admins to assign permissions through the backend, you should ensure that they can assign the permission to create tickets and that users to whom they assign this permission can perform that action.

CREATING TICKETS

Open spec/features/admin/assigning_permissions_spec.rb, and add the scenario shown in the following listing right under the scenario currently in this file.

Listing 8.15 spec/features/admin/assigning_permissions_spec.rb

```
scenario "Creating tickets for a project" do
  check_permission_box "view", project
  check_permission_box "create_tickets", project
  click_button "Update"
  click_link "Sign out"

  sign_in_as!(user)
  click_link project.name
  click_link "New Ticket"
  fill_in "Title", with: "Shiny!"
  fill_in "Description", with: "Make it so!"
  click_button "Create"

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

Just as in your first scenario, you select the View check box for the project. Otherwise, the user wouldn't be able to see the project where the New Ticket link was. Then you select the Create Tickets check box, update the user's permissions, and sign out. Next, you sign in as that user and make sure you can do what you just gave that user permission to do. When you run this feature with `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, the step that checks the permission box for `create_tickets` fails because it can't find the check box:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: check_permission_box "create_tickets", project
Capybara::ElementNotFound:
  Unable to find checkbox "permissions_1_create_tickets"
```

Let's add this check box. Open app/helpers/admin/permissions_helper.rb, and add the permission to your hash, changing this method from

```
def permissions
  {
    "view" => "View"
  }
end
```

to

```
def permissions
  {
    "view" => "View",
    "create tickets" => "Create Tickets"
  }
end
```

Keep in mind that the key *must* match the intended `action` for the `Permission` object; the value is just a label for this permission. The action is what you use in your `authorized?` helpers around the application. When you add another key-value pair in this hash, the code in app/views/admin/permissions/index.html.erb automatically shows a check box for this new permission.

When you run this feature again with `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, it passes because this new check box is visible and the permission is applied correctly:

```
2 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Wasn't that a piece of cake? Let's move on to the next permission: updating tickets. Actually, let's do both updating tickets and deleting tickets at the same time.

THE DOUBLE WHAMMY

To show the world how great you are at developing this application, you'll next write two scenarios and get them both to pass at the same time. Add both of these scenarios to the end of spec/features/admin/assigning_permissions_spec.rb, as shown in the following listing.

Listing 8.16 spec/features/admin/assigning_permissions_spec.rb

```
scenario "Updating a ticket for a project" do
  check_permission_box "view", project
  check_permission_box "edit_tickets", project
  click_button "Update"
  click_link "Sign out"

  sign_in_as!(user)
  click_link project.name
  click_link ticket.title
  click_link "Edit Ticket"
  fill_in "Title", with: "Really shiny!"
  click_button "Update Ticket"

  expect(page).to have_content("Ticket has been updated")
end

scenario "Deleting a ticket for a project" do
  check_permission_box "view", project
  check_permission_box "delete_tickets", project

  click_button "Update"
  click_link "Sign out"

  sign_in_as!(user)
  click_link project.name
  click_link ticket.title
  click_link "Delete Ticket"

  expect(page).to have_content("Ticket has been deleted.")
end
```

The scenarios should be descriptive enough to understand—no particular magic is going on. When you run this feature using `bin/rspec spec/features/admin/assigning_permissions_spec.rb`, it can't

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

find the check boxes for both the updating and deleting permissions:

```
Failure/Error: check_permission_box "edit_tickets", project
  Unable to find checkbox "permissions_1_edit_tickets"
Capybara::ElementNotFound:
Failure/Error: check_permission_box "delete_tickets", project
Capybara::ElementNotFound:
  Unable to find checkbox "permissions_1_delete_tickets"
```

Of course! You have no check boxes for Capybara to check yet. Add them by changing the `permissions` method in app/helper/admin/permissions_helper.rb from this

```
def permissions
  {
    "view" => "View",
    "create tickets" => "Create Tickets"
  }
end
```

to this:

```
def permissions
  {
    "view" => "View",
    "create tickets" => "Create Tickets",
    "edit tickets" => "Edit Tickets",
    "delete tickets" => "Delete Tickets"
  }
end
```

By adding these two permissions, you display the check boxes that should make your scenarios all green. Let's run the feature with `bin/rspec spec/features/admin/assigning_permissions_spec.rb` to find out:

```
4 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

How great are you? Two features with one blow. Awesome stuff.

That's the final piece of the authorization puzzle. You now have a way for admins to assign permissions to users. But does everything work? Here's hoping! Run `bin/rspec` to find out:

```
62 examples, 0 failures
```

Awesome! Let's commit this:

```
$ git add .
$ git commit -m "Add creating, editing, updating and deleting
               tickets to assigning permissions interface"
$ git push
```

Now there's a way for admin users of your application to assign permissions to users so that they're able to view projects as well as create, edit, and update tickets for those projects. In doing so, you've learned how you can update multiple records at the same time.

Although it's great that you have an interface for assigning permissions, you don't have a way to do it without first having an admin user set up. You can set up an admin user manually through the console, or you can do so by creating *seed data*.

8.9.3 Seed data

Seed data are records created for the purpose of providing the minimal viable requirements to get an application running. Before Rails 2.2, many applications implemented such records by using plugins such as *seed_fu*, but since 2.2, seed data is built in.

Seed data allows you to create records for your application to provide a usable base if you or anybody else wants to get set up with the application quickly and easily. For your application's seed data, you'll create an admin user and an example project. From there, anybody using the admin user will be able to perform all the functions of the application.

Seed data lives under db/seeds.rb, and you can run this file by running

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`bin/rake db:seed`. The code for the `bin/rake` task is this:

```
load Rails.root + "db/seeds.rb"
```

The `load` method works in a fashion similar to `require`, loading and executing the code in the file. One difference, however, is that `load` expects the given string (or `Pathname`) to be the full path, with the extension, to the file.

First write a feature to ensure that when the seed data is loaded, you can sign in with the email admin@example.com and the password “password” and get to the Ticketee Beta project. Put this feature at spec/features/seeds_spec.rb, and write it as shown in the following listing.

Listing 8.17 spec/features/seeds_spec.rb

```
require 'spec_helper'

feature "Seed Data" do
  scenario "The basics" do
    load Rails.root + "db/seeds.rb"
    user = User.where(email: "admin@example.com").first!
    project = Project.where(name: "Ticketee Beta").first!
  end
end
```

It’s a pretty basic feature, but your seed file will be equally basic. When you run this feature using `bin/rspec spec/features/seeds_spec.rb`, it fails like this:

```
Failure/Error: user = User.where(email: "admin@example.com").first!
ActiveRecord::RecordNotFound:
  ActiveRecord::RecordNotFound
```

It can’t find this user because you haven’t yet created one for this scenario. This user should be created by the db/seeds.rb file. Open db/seeds.rb and add a couple of lines to create this user, set up the user as an admin, and confirm the user, as shown next.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 8.18 db/seeds.rb

```
admin_user = User.create(email: "admin@example.com",
                         name: "admin",
                         password: "password",
                         password_confirmation: "password",
                         admin: true)
```

Run `bin/rspec spec/features/seeds_spec.rb` to ensure that you can sign in as this user. If you can, you should see that the feature can't see the Ticketee Beta content:

```
Failure/Error: project = Project.where(name: "Ticketee Beta").first!
ActiveRecord::RecordNotFound:
  ActiveRecord::RecordNotFound
```

To get this last step of the scenario to pass, you must add the project to db/seeds.rb by putting this line in there:

```
Project.create(name: "Ticketee Beta")
```

Now your whole seeds file should look like the following listing.

Listing 8.19 db/seeds.rb

```
admin_user = User.create(email: "admin@example.com",
                         password: "password",
                         admin: true)

Project.create(name: "Ticketee Beta")
```

This is all you need to get this feature to pass. Let's run it with `bin/rspec spec/features/seeds_spec.rb` to make sure:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
1 example, 0 failures
```

Great! With this seeds file, you have data to put in the database so you can bootstrap your application. Let's run `bin/rake db:seed` to load this data. Start your application's server by typing `rails server` into a terminal, and then go to your server at http://localhost:3000 in your browser. Sign in as the admin user using the email admin@example.com and password "password". You should see the display shown in figure 8.6.

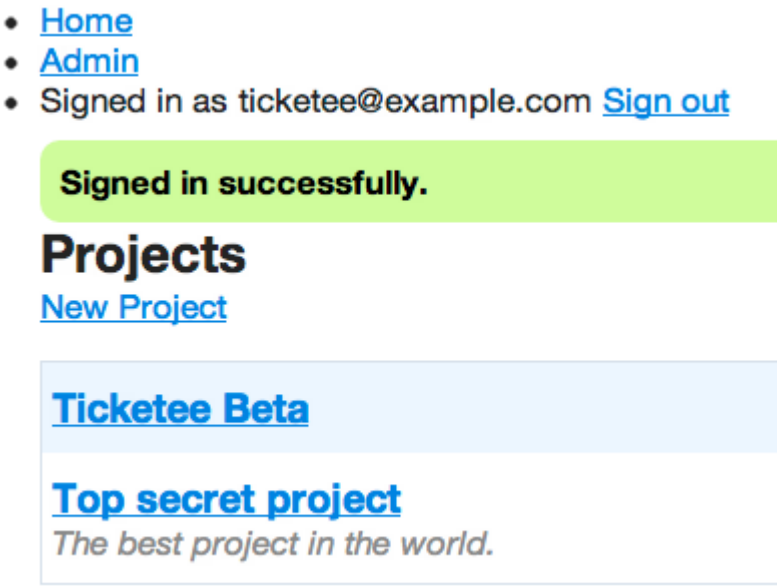

Figure 8.5 What admins see

When you're signed in as a user, you should be able to do everything from creating a new ticket to creating a new user and setting up user permissions. Go ahead and play around with what you've created so far.

When you're done playing, run `bin/rspec` for the final time this chapter:

```
65 examples, 0 failures
```

Everything's still green, which means it's time for another commit:

```
$ git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$ git commit -m "Add a seeds file"
$ git push
```

Now you're finished!

8.10 Summary

This chapter covered implementing authorization for your application and setting up a permissions-based system for both `ProjectsController` and `TicketsController`. You started with a `Permission` model, which you used in a `scope` on the `Project` model to show only the projects a user should be able to access.

Then you used the CanCan plug-in, which provided the `can?` and `cannot?` methods to use first in the controllers to stop users from accessing specified actions and then in the views, through the `authorized?` method, to stop users from seeing specified links.

You implemented a way for admins to change the permissions of a user through the admin backend of the system by displaying a series of check boxes. Here you used an `update` action that wasn't quite like the normal `update` action, and you had to define a custom-named route for it.

Finally, you learned how to set up seed data for your application so you have a solid base of objects to work from. Without using seed data, you'd have to manually set up the data not only for your local development environment but also for your production server, which can be tedious. Seed data saves you that effort. You also wrote a test for this data in the form of a feature that ensures that the data from the seed file is always created when the seed task is run.

In chapter 9, you'll learn how to attach files to tickets. File uploading is an essential part of any ticket-tracking application because files can provide that additional piece of context required for a ticket, such as a screenshot, a patch, or any type of file. You'll also learn some CoffeeScript to make it easy to upload any number of files, and strategies for testing JavaScript.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Index Terms

ActiveRecord::Base, has_many, :as option
ActiveRecord::Base, joins
ActiveRecord::Base, scope method
ActiveRecord::Base, where
authorized? helper
belongs_to, polymorphic option
CanCan, cannot?
clear, ActiveRecord::Base
cycle, helper method
db/seeds.rb
named routes
polymorphic associations
scopes
ternary statement
to_sym
try?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

9 File uploading

In chapter 8, you learned how to restrict access to specific actions in your application, such as viewing projects and creating tickets, by defining a `Permission` model that keeps track of which users have access to which actions.

Ticketee's getting pretty useful now. 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 the functionality to attach files to the ticket, you provide the project owners a useful context that will help them more easily understand what the ticket creator means. Luckily, there's a gem called CarrierWave that allows you to implement this feature easily.

Once you're familiar with CarrierWave, you'll change your application to accept multiple files attached to the same ticket using a JavaScript library called jQuery (which comes with Rails by default, through the jquery-rails gem) and some custom JavaScript code of your own. Because you're using JavaScript, you have to alter the way you test parts of your application. To test JavaScript functionality, you'll be using WebDriver,[1] which is a framework built for automatic control of web browsers. WebDriver is especially useful because you can use the same steps you use for standard Capybara tests and because Capybara

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

will take care of driving the browser. By running the tests inside the browser, you ensure the JavaScript on the page will be executed, and then you can run the tests on the results. Pretty handy!

Footnote 1 There's a great post explaining WebDriver on the Google Open Source blog: http://google-opensource.blogspot.com/2009/05/introducing-webdriver.html .

Finally, you'll see how you can restrict access to the files contained within the projects of your application so that confidential information isn't shared with people who don't have access to a particular project.

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. File uploading has many different uses and is a cornerstone of many Rails applications.

9.1 Attaching a file

We start off by letting users attach files when they begin creating a ticket. As explained before, files attached to tickets can provide useful context as to 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 context is key to solving tickets.

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. You use the CarrierWave gem to store the file inside your application's directory.

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 on 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,[2] as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 2 In this attachement, we reference the `blink` tag. Please note that although the `blink` tag was once a part of HTML, it should never be used. Same goes for the `marquee` tag. We reference them here in our text files to add some light humor to the scenario, not because documentation for these tags is actually a good idea.

Listing 9.1 spec/features/creating_tickets_spec.rb

```
scenario "Creating a ticket with an attachment" do
  fill_in "Title", 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 .asset") do
    expect(page).to have_content("speed.txt")
  end
end
```

In this feature you introduce 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, except that at the moment, you don't have any.[3] Create the spec/fixtures/speed.txt file now and fill it with some random filler text like this:

Footnote 3 Nor will we ever, as factories replace them in our application.

```
The blink tag can blink faster if you use the speed="hyper" attribute.
```

Try running this feature using `bin/rspec spec/features/creating_tickets.rb` and see how far you get. It will fail on the `attach_file` line because the `File` field isn't available yet.

```
Failure/Error: attach_file "File", "spec/fixtures/speed.txt"
Capybara::ElementNotFound:
  Unable to find file field "File"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Add the `File` field to the ticket form partial directly underneath the `p` tag for the `description` field using the code in the following listing.

Listing 9.2 app/views/tickets/_form.html.erb

```
<p>
  <%= f.label :asset, "File" %><br>
  <%= f.file_field :asset %>
</p>
```

We call this field `asset` internally, but the user will see File. The reason for this is explained a little later.

You have to do one other thing when using a `file_field`: add a bit to your `form_for` call:

```
<%= form_for [@project, @ticket], html: { multipart: true } do |f| %>
```

This attribute is needed on the generated HTML form tag in order to actually upload the files. Rails tries to be smart about it, but since we're adding them with JavaScript, Rails can't detect that we wanted it on there, so we have to add it ourselves.

If you run `bin/rspec spec/features/creating_tickets_spec.rb` again, it will fail with this error instead:

```
Failure/Error: within("#ticket .asset") do
Capybara::ElementNotFound:
  Unable to find css "#ticket .asset"
```

You can see that the scenario failed because Capybara can’t find the text within this element on the `TicketsController`’s `show` page: this text and this

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

element don't exist! You need to add this element for your scenario to go one step further, so add it underneath the spot in the `show` view where you currently have the following:

```
<%= simple_format(@ticket.description) %>
```

You must also wrap all the code in this view inside a `div` tag with the `id` attribute `ticket` and spice it up a little by adding the content type and file size there too, as shown in the following listing.

Listing 9.3 app/views/tickets/show.html.erb

```
<small>Created by <%= @ticket.user.email %></small>

<%= simple_format(@ticket.description) %>

<% if @ticket.asset.present? %>
  <h3>Attached File</h3>
  <div class="asset">
    <p>
      <%= link_to File.basename(@ticket.asset.path),          (1)
                  @ticket.asset.url %>
    </p>
    <p><small><%= number_to_human_size(@ticket.asset.size) %>
     </small></p>
  </div>
<% end %>
```

You use the `url` (1) method here with `link_to` to provide the user with a link to download[4] this file. In this case, the URL for this file would be something like http://localhost:3000/system/assets/1/original/file.txt.

Footnote 4 Some browsers open certain files as pages rather than downloading them. Modern browsers do so for .txt files and the like.

Where is this system route defined? Well, it's not a route. It's actually a directory inside the public folder of your application where CarrierWave saves your files.

Requests to files from the public directory are handled by the server rather than by Rails, and anybody who enters the URL in their browser can access them. This

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

is bad because the files in a project should be visible only to authorized users. You'll handle that problem a little later in the chapter.

Underneath the filename, you display the size of the file, which is stored in the database as the number of bytes. To convert it to a human-readable output, (such as "71 Bytes," which will be displayed for your file), you use the `number_to_human_size` Action View helper.

If you run `bin/rspec spec/features/creating_tickets_spec.rb` again, it will fail with this error instead:

```
Failure/Error: click_button "Create Ticket"
ActionView::Template::Error:
  undefined method `asset' for #<Ticket:0x007fb090a7c1a8>
```

Rather than running a migration to add an attribute by this name, you use the CarrierWave gem to handle it.

9.1.2 Enter stage right, CarrierWave

Uploading files is something that 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 CarrierWave. CarrierWave makes uploading files really 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, CarrierWave is there to help you too. To install CarrierWave, you need to add a line to the Gemfile to tell Bundler that you want to use this gem. Put this underneath the line specifying the CanCan gem, separated by a line because it's a different type of gem (CanCan has to do with users, CarrierWave has to do with files):

```
gem 'cancan', '1.6.9'

gem 'carrierwave', '0.8.0'
```

Next, you must run `bundle install` to install this gem.

With CarrierWave now installed, you can work on defining the asset attribute that your model wants. It's not really an attribute; the error message is misleading

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

in that respect. All it needs is a setter method (`asset=`) and it would be happy. However, you need this method to do more than set an attribute on this object; you need it to accept the uploaded file and store it locally. CarrierWave lets you define this fairly easily with its `has_attached_file` method. This method goes in the `Ticket` model, defines the setter method you need, and gives four application the ability to accept and process this file. Add it to your `Ticket` model with this line:

```
mount_uploader :asset, AssetUploader
```

Now this `asset=` method is defined, but it's not yet over!

9.1.3 Using CarrierWave

We also need to generate an uploader. Do this with `bin/rails generate uploader Asset`. You'll see a new file: app/uploaders/asset_uploader.rb. While we have an uploader, we need to connect it with our `Ticket`s, so we need a migration. Generate one with `bin/rails generate migration add_asset_to_tickets asset:string`. To add these columns to your development environment's database, run `bin/rake db:migrate`. Then run `bin/rake db:test:prepare` to add them to your test environment's database.

We have one last thing to do: update our controller to allow us to pass in assets. Change your `ticket_params` method in your app/controllers/tickets_controller.rb to look like this:

```
def ticket_params
  params.require(:ticket).permit(:title, :description, :asset)
end
```

With the file's information now being output in app/views/tickets/show.html.erb, this feature now passes when you run `bin/rspec spec/features/creating_tickets_spec.rb`:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
4 examples, 0 failures
```

Awesome! Your files are being uploaded and taken care of by CarrierWave, which stores them at public/uploads. Let's see if your changes have brought destruction or salvation by running `rake spec`.

```
64 examples, 0 failures
```

Sweet salvation! Let's commit but not push this just yet.

```
git add .
git commit -m "Add the ability to attach a file to a ticket"
```

Have a look at the commit output. It contains this line:

```
create mode 100644 public/uploads/ticket/asset/1/speed.txt
```

This line is a leftover file from your test and shouldn't be committed to the repository because you could be testing using files much larger than this. You can tell Git to ignore the entire public/system directory by adding it to the .gitignore file. Open that file now and add this line to the bottom:

```
public/uploads
```

This file tells Git which files you don't want versioned. The whole file should look like this now (minus all the comments):

```
/.bundle
/db/*.sqlite3
/log/*.log
/tmp
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
public/uploads
```

By default, the `.bundle` directory (for Bundler's configuration), the SQLite3 databases, the logs for the application, and any files in tmp are ignored. With public/system added, this directory is now ignored by Git too. You should also remove this directory from your latest commit, and thankfully, Git provides a way to do so by using these two commands:

```
git rm public/uploads/ticket/asset/1/speed.txt
git commit --amend --reuse-message HEAD
```

The first command removes the file from the filesystem and tells Git to remove it from the repository. The second command amends your latest commit to exclude this file, and it will be as if your first commit with this message never existed. The `--reuse-message HEAD` option uses the commit message of our latest commit. Let's push this change now:

```
git push
```

Great! Now you can attach a file to a ticket. There's still some work to do, however. What would happen if somebody wanted to add more than one file to a ticket? Let's take a look at how to do that now.

9.2 Attaching many files

You now 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 to three.

If you're going to add these three file input fields to your view, you need some more fields in your database to handle them. You could define four fields for each file upload field, but a much better way to handle this is to add another model.

Creating another model gives you the advantage of being able to scale it to not just three file input fields but more if you ever need them. Call this model `Asset`, after the name we gave to the `has_attached_file` in the `Ticket` model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you're done with this feature, you should see three file upload fields as shown in Figure 9.1

Figure 9.1 File upload boxes

You can create new instances of this model through the ticket form by using *nested attributes*. Nested attributes have been a feature of Rails since version 2.3, and they allow the attributes of any kind of association to be passed from the creation or update of a particular resource. In this case, you'll be passing nested attributes for a collection of new asset objects while creating a new `Ticket` model. The best part is that the code to do all of this remains the same in the controller.

9.2.1 Testing multiple file upload

Let's take the scenario for creating a ticket with an attachment from spec/features/creating_tickets_spec.rb and add two additional file upload fields so the entire scenario looks like the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 9.4 File attachment scenario, spec/features/creating_tickets_spec.rb

```
scenario "Creating a ticket with an attachment" do
  fill_in "Title", with: "Add documentation for blink tag"
  fill_in "Description", with: "The blink tag has a speed attribute"

  attach_file "File #1", Rails.root.join("spec/fixtures/speed.txt")
  attach_file "File #2", Rails.root.join("spec/fixtures/spin.txt")
  attach_file "File #3", Rails.root.join("spec/fixtures/gradient.txt")

  click_button "Create Ticket"

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

  within("#ticket .assets") do
    expect(page).to have_content("speed.txt")
    expect(page).to have_content("spin.txt")
    expect(page).to have_content("gradient.txt")
  end
end
```

In this scenario, you attach three files to your ticket and assert that you see them within the assets element, which was previously called `#ticket .asset` but now has the pluralized name of `#ticket .assets`.

Now run this single scenario using `bin/rspec spec/features/creating_tickets_spec.rb`. It should fail on the first "attach_file" step, because you renamed the label of this field.

```
Failure/Error: attach_file "File #1", "spec/fixtures/speed.txt"
Capybara::ElementNotFound:
  Unable to find file field "File #1"
```

9.2.2 Implementing multiple file upload

To get this step to pass, you can change the label on the field in app/views/tickets/_form.html.erb to `"File #1"`:

```
<p>
  <%= f.label :asset, "File #1" %>
  <%= f.file_field :asset %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
</p>
```

While you're changing things, you may as well change app/views/tickets/show.html.erb to reflect these latest developments. First, change the `if` around the asset field to use the `assets` method, because it'll need to check the `assets` of a ticket rather than the `asset`. This line in app/views/tickets/show.html.erb should change from this:

```
  <% if @ticket.asset.present? %>
```

To this:

```
  <% if @ticket.assets.present? %>
```

You also need to change the `h3` that currently reads `Attached File` so it reads `Attached Files` because now there's more than one file. You should also change the `div` that encapsulates your assets to have the `class` attribute of `assets`. These three changes mean that you now have these three lines in app/views/tickets/show.html.erb:

```
<% if @ticket.assets.exists? %>
  <h3>Attached Files</h3>
  <div class="assets">
```

When you call `present?` this time, it calls the `ActiveRecord::Base` association method, which checks if there are any assets on a ticket and returns `true` if there are. Although `assets` isn't yet defined, you can probably guess what you're about to do.

First, though, you need to change the lines underneath the ones you just changed to the following:

```
<% @ticket.assets.each do |asset| %>
  <p>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    <%= link_to File.basename(asset.asset.path), asset.asset.url %>
  </p>
  <p>
    <small><%= number_to_human_size(asset.asset.size) %></small>
  </p>
<% end %>
```

Here you switch to using the `assets` method and iterate through each element in the array, rendering the same output as you did when you had a single asset.

All of these changes combined will help your scenario pass, which is a great thing. When you run this feature again with `bin/rspec spec/features/creating_tickets_spec.rb`, the first attachment step inside the scenario passes, but the second fails:

```
Failure/Error: attach_file "File #2", Rails.root.join...
 Capybara::FileNotFound:
   cannot attach file, spec/fixtures/spin.txt does not exist
```

To fix this error, create a new file at spec/fixtures/spin.txt and put this content in it:

```
Spinning blink tags have a 200% higher click rate!
```

On the next run of `bin/rspec spec/features/creating_tickets_spec.rb` this error is replaced with a new one:

```
Failure/Error: attach_file "File #2", Rails.root.join...
Capybara::ElementNotFound:
  Unable to find file field "File #2"
```

You *could* add another field:

```
<p>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  <%= f.label :asset_2, "File #2" %>
  <%= f.file_field :asset_2 %>
</p>
```

But that's a messy way of going about it. The best way to handle this problem is by having some code to automatically present a number of fields -- say, three -- on the page to the user with an option to add more if they like. This is possible by using a `has_many` association for assets on the `Ticket` class, and by using *nested attributes*. To use nested attributes in the view, you use the `fields_for` helper. This helper defines the fields for an association's records, as many as you like. Let's remove the file field completely and replace it with the code from this listing:

Listing 9.5 app/views/tickets/_form.html.erb, asset fields

```
<% number = 0 %>                                                    ❶
<%= f.fields_for :assets do |asset| %>
  <p>
    <%= asset.label :asset, "File ##{number += 1}" %>               ❷

    <%= asset.file_field :asset %>
  </p>
<% end %>
```

Directly before the `fields_for` call, ❶ you set a local variable called `number`, which is incremented whenever you render a `label` ❷.

You use `fields_for` much in the same way you use `form_for`. You call `fields_for` on the `f` block variable from `form_for`, which tells it you want to define nested fields inside this original form. The argument to `fields_for`—`:assets` —tells Rails the name of the nested fields.

The file field inside this `fields_for` now has the `name` attribute of `ticket[assets][asset]` rather than simply `ticket[asset]`, meaning it will be available in the controller as `params[:ticket][:assets][:asset]`.

When you run this feature with `bin/rspec spec/features/creating_tickets_spec.rb`, the final scenario now fails because it still can't find the second file upload field:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
And I attach the file "spec/fixtures/spin.txt" to "File #2"
  cannot attach file, no file field with id, name,
  or label 'File #2' found (Capybara::ElementNotFound)
```

To make this appear, define an `assets` association in your `Ticket` model so the `fields_for` in your view can eventually provide `file_field`s for three new `Asset` objects. The defining of this association will also stop the other scenario that is currently failing in "Creating Tickets" from complaining about the missing `assets` method.

If this `assets` association is defined on the `Ticket` model and you've declared that your model accepts nested attributes for the association, `fields_for` iterates through the output from this method and renders the fields from `fields_for` for each element. This means that a file field will be rendered for every single `Asset` object in the `@ticket.assets` collection.

You can define this `assets` method by defining a `has_many` association in your `Ticket` model:

```
has_many :assets
```

Underneath this `has_many`, you also define that a `Ticket` model accepts nested attributes for assets by using `accepts_nested_attributes_for`:

```
accepts_nested_attributes_for :assets
```

This little helper tells your model to accept asset attributes along with ticket attributes whenever you call methods like `new`, `build`, or `update`. It has the added bonus of switching how `fields_for` performs in your form, making it reference the association and calling the attributes it defines `assets_attributes` rather than `assets`.

When you run our scenario with `bin/rspec spec/features/creating_tickets_spec.rb`, you see that Rails is now

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

basically *demanding* that there is an `Asset` class. So needy!

```
And I follow "New Ticket"
  uninitialized constant Ticket::Asset (ActionView::Template::Error)
```

We'd best get onto that then!

9.2.3 Using nested attributes

You used the term Asset rather than File throughout your application because of this model. You can't define a `File` model because there's already a `File` class in Ruby. Asset is an alternative name you can use. Rails uses it itself to describe static files, which is why you have an app/assets directory. To define this `Asset` constant in your application, you can run the model generator:

```
$ bin/rails g model asset
```

Each record for this model refers to a single file that has been uploaded to a ticket. Therefore, each record in the `assets` table must have the same `asset` field that each `tickets` record currently has. Storing the asset references in the `assets` table now makes the references in the `tickets` table irrelevant, so you should remove them. You should also add a relationship between the asset records and the ticket records by adding a `ticket_id` field to the `assets` table. Open the migration this generates and change it to the following listing to reflect these ideas.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 9.6 db/migrate/[date]_create_assets.rb

```
class CreateAssets < ActiveRecord::Migration
  create_table :assets do |t|
    t.string :asset
    t.references :ticket

    t.timestamps
  end

  remove_column :tickets, :asset
end
```

Run this migration with `bin/rake db:migrate` to migrate your development environment's database, and then run `bin/rake db:test:prepare` to migrate the test environment's database. When you run the feature again with `bin/rspec spec/features/creating_tickets_spec.rb`, your `File #1` field is once again missing!

```
And I attach the file "spec/fixtures/speed.txt" to "File #1"
Capybara::ElementNotFound:
  Unable to find file field "File #1"
```

You've gone backwards! Or so it seems.

As mentioned earlier, `fields_for` detects that the `assets` method is defined on your `Ticket` object and then iterates through each object in this collection while rendering the fields inside `fields_for` for each. When you create a new ticket in `TicketsController`'s `new` action, however, you don't initialize any assets for this ticket, so `assets` returns an empty array and no fields at all are displayed.

To get this action to render three file input fields, you must initialize three `Asset` objects associated to the `Ticket` object the form uses. Change your `new` action inside `TicketsController` to this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def new
  @ticket = @project.tickets.build
  3.times { @ticket.assets.build }
end
```

The final line of this action calls `@ticket.assets.build` three times, which creates the three `Asset` objects you need for your `fields_for`.

When you run our scenario again, the three fields are now available, but the scenario now fails because it can't find a file to upload:

```
Failure/Error: attach_file "File #3", "spec/fixtures/gradient.txt"
Capybara::FileNotFound:
 cannot attach file, spec/fixtures/gradient.txt does not exist
```

Create this gradient.txt file now inside the spec/fixtures directory and give it the following content:

```
Everything looks better with a gradient!
```

This text piece is random filler meant only to provide some text if you ever need to reference it. Let's run the scenario again with `bin/rspec spec/features/creating_tickets_spec.rb`:

```
Failure/Error: within("#ticket .assets") do
Capybara::ElementNotFound:
  Unable to find css "#ticket .assets"
```

Remember this from last time? It means we need `strong_parmeters` in our controller. Modify your `ticket_params` method in app/controllers/tickets_controller.rb:

```
def ticket_params
  params.require(:ticket).permit(:title, :description,
    assets_attributes: [:asset])
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

Tricky! `:asset` becomes `:assets_attributes`, and we also need to say that that array of attributes contains an inner `:asset`. This mirrors the structure of our models: a `Ticket` `has_many` `Asset`s. And each `Asset` has an uploaded file, named `asset`. Whew!

This should be all that is required to get the multiple asset uploading working. Find out by running `bin/rspec spec/features/creating_tickets_spec.rb`. You should see this output:

```
4 examples, 0 failures
```

Hooray, the scenario passed! In this section, you set up the form that creates new `Ticket` objects to also create three associated `Asset` objects by using nested attributes. This process was made possible by moving the responsibility of handling file uploads out of the `Ticket` model and into the associated `Asset` model. The `accepts_nested_attributes` call inside the `Ticket` model, as well as the `fields_for` call in app/views/tickets/_form.html.erb also played vital roles in getting this to work.

Let’s ensure that nothing is broken by running `bin/rake spec`.

```
63 examples, 0 failures, 1 pending
```

Yup, nothing's broken, but there's one pending spec that lives in spec/models/asset_spec.rb. Let's delete this file now. If you re-run `rake spec` you should see no more pending specs:

```
64 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Awesome, let's commit and push this.

```
$ git add .
$ git commit -m "Users can now upload 3 files at a time"
$ git push
```

Great. You're done with nested attributes! Earlier, it was mentioned that the files uploaded to your application are publicly available for anybody to access because these files are in the public directory. Any file in the public directory is served up automatically by any Rails server, bypassing all the authentication and authorization in your application. This is a bad thing. What if one of the projects in your application has files that should be accessed only by authorized users?

9.3 Serving files through a controller

You can solve this issue by serving the uploaded files through a controller for your application. Using a `before_filter` similar to the one you used previously in the `TicketsController`, this controller will check that the user attempting to access a file has permission to access that particular project.

When you implemented permission behavior before, you ensured that any unauthorized user would be blocked from accessing the resource you were trying to protect by writing a controller spec. You write this same kind of spec test for serving files.

9.3.1 Protecting files

You first need to generate the controller through which you'll serve the assets. Call it `files`, because `assets` is already reserved by the asset pipeline:

```
$ bin/rails g controller files
```

Now write a spec to ensure that unauthorized users can't see the files inside it. For this spec test, you must create two users, a project, a ticket, and an asset. The first user should have permission to read this project, and the second user shouldn't.

Open spec/controllers/files_controller_spec.rb now and add `let` definitions that set up your users, project, ticket, and asset inside the `describe` for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`FilesController`, as shown in the following listing.

Listing 9.7 spec/controllers/files_controller_spec.rb

```
require 'spec_helper'

describe FilesController do
  let(:good_user) { FactoryGirl.create(:user) }
  let(:bad_user) { FactoryGirl.create(:user) }

  let(:project) { FactoryGirl.create(:project) }
  let(:ticket) { FactoryGirl.create(:ticket,
                                    project: project,
                                    user: good_user) }

  let(:path) { Rails.root + "spec/fixtures/speed.txt" }
  let(:asset) do
    ticket.assets.create(asset: File.open(path))
  end

  before do
    good_user.permissions.create!(action: "view",
                                  thing: project)
  end
end
```

You used a `let` for setting up a project, two users, a ticket for this project, a path to the file that's served from the controller, and the asset for the ticket. This is the asset you'll be serving from the controller for this spec test.

You set up the permission in a `before` block because you won't be referencing it anywhere in your tests, so having it as a `let` block wouldn't work. You should use `let` blocks only when you're going to be referencing them inside your tests. If you need code set up beforehand, you should use a `before` block instead.

To serve the files from this controller, use the `show` action, using the `id` parameter to find the asset the user is requesting. When the application finds this asset, you want it to check that the user requesting the asset has permission to read the project this asset links to. The `good_user` object should be able to, and the `bad_user` object shouldn't. Now add the spec to test the `good_user`'s ability to download this asset by using the code from the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 9.8 spec/controllers/files_controller_spec.rb

```
context "users with access" do

  before do
    sign_in(good_user)
  end

  it "can access assets in a project" do
    get 'show', id: asset.id
    expect(response.body).to eql(File.read(path))
  end
end
```

If you're using Windows you may have to do this on the `response.body` line instead, as the line breaks on Windows are slightly different:

```
  expect(response.body.gsub!(/\r\n?/, "\n")).to eql(File.read(path))
```

In this example, you sign in as the `good_user` by using another `before` block. Then you assert that when this user attempts to get this asset through the `show` action, the user should receive it as a response. Write another `context` and spec for the `bad_user` too, as shown in the following listing.

Listing 9.9 spec/controllers/files_controller_spec.rb

```
context "users without access" do
  before do
    sign_in(bad_user)
  end

  it "cannot access assets in this project" do
    get 'show', id: asset.id
    expect(response).to redirect_to(root_path)
    expect(flash[:alert]).to eql("The asset you were looking for " +
                                 "could not be found.")
  end
end
```

Here you sign in as the `bad_user` and then deny all knowledge of the asset's

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

existence by redirecting to root and displaying an alert flash message. Let's run these specs now with `bin/rspec spec/controllers/files_controller_spec.rb`. Both examples complain:

```
ActionController::UrlGenerationError:
  No route matches {:id=>"1", :controller=>"files", :action=>"show"}
```

Well, that's no good. Now you need to define this `show` action.

9.3.2 Showing your assets

Open your `FilesController` file now and define the `show` action, along with a `before_filter` to set the `current_user` variable, which you'll need for permission checking. This code is shown in the following listing.

Listing 9.10 app/controllers/files_controller.rb

```
class FilesController < ApplicationController
  before_filter :require_signin!

  def show
    asset = Asset.find(params[:id])
    send_file asset.asset.path,
              filename:     asset.asset_identifier,
              content_type: asset.content_type
  end
end
```

In this action, you find the `Asset` object by using the `params[:id]` the action receives. Then you use the `asset` object in combination with the `send_file` method to send the file back as a response rather than a view in your application.

The first argument for `send_file` is the path to the file you're sending. The next argument is an options hash used to pass in the `filename` and `content_type` options so the browser receiving the file knows what to call it and what type of file it is.

To route requests to this controller, you need to define a route in your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

config/routes.rb file, which you can do with this line:

```
resources :files
```

Okay, we're ready to run our tests again. Check it out with `bin/rspec spec/controllers/files_controller_spec.rb`:

```
NoMethodError:
  undefined method `content_type' for #<Asset:0x007f819c3be500>
```

Right, we don't have a `content_type` method on our `Asset` class. We need to save the type when the file is uploaded, so we know what type to serve it with.

To do that, we need a few things: a new column in our database, as well as a way of storing it. Let's generate that migration:

```
$ bin/rails g migration add_content_type_to_assets content_type:string
$ bin/rake db:migrate db:test:prepare
```

It's so simple, we didn't even look at it before migrating. You should have seen some output like this:

```
==  AddContentTypeToAssets: migrating ======================
-- add_column(:assets, :content_type, :string)
   -> 0.0055s
==  AddContentTypeToAssets: migrated (0.0056s) =============
```

Anyway, now we need to save the content type when our asset is uploaded. To do this you can use a callback. A callback is a method that's called either before or after a certain event. For models, there are before-and-after callbacks for the following events (where * can be substituted for either `before` or `after`):

- Validation (`*_validation`)
- Creating (`*_create`)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

- Updating (`*_update`)
- Saving (`*_save`)
- Destruction (`*_destroy`)

We're able to trigger a specific piece of code or method to run before or after any of these events. The "Saving" item in the above list refers to when a record is saved to the database, which occurs when a record is created or updated. For your `Asset` model you'll want to define a callback that occurs before a record is about to be saved and for this you'll use the `before_save` method at the top of your `Asset` model, like this:

```
class Asset < ActiveRecord::Base
  mount_uploader :asset, AssetUploader

  before_save :update_content_type

  private

    def update_content_type
      if asset.present? && asset_changed?
        self.content_type = asset.file.content_type
      end
    end
end
```

When you run the specs for this controller again using `bin/rspec spec/controllers/files_controller_spec.rb`, the first spec passes, but the second one fails:

```
Failure/Error: response.should redirect_to(root_path)
Expected response to be a <redirect>, but was <200>
```

The `show` action doesn't redirect as this example expects because you're not doing any permission checking in your action, which is what this example is all about: "`users without access cannot access assets in this project`." To fix this problem, check that the user has permission to access this asset's project by using the CanCan helpers you used in chapter 8. you can use them in your `show` action now, as shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 9.11 app/controllers/files_controller.rb

```
def show
  asset = Asset.find(params[:id])
  if can?(:view, asset.ticket.project)
    send_file asset.asset.path,
      filename:     asset.asset_identifier,
      content_type: asset.content_type
  else
    flash[:alert] = "The asset you were looking for could not be found."
    redirect_to root_path
  end
end
```

Now when you rerun the specs with `bin/rspec spec/controllers/files_controller.rb`, you'll see that you're missing a method:

```
undefined method 'ticket' for #<Asset:0x000001043d1e18>
```

This method is a simple `belongs_to`, which you must define inside the `Asset` model:

```
belongs_to :ticket
```

When you rerun your specs, they both pass because the authorized user (`good_user`) can get a file and the unauthorized user (`bad_user`) can't:

```
2 examples, 0 failures
```

Great! Now you've begun to serve the files from `FilesController` to only people who have access to the asset's relative projects. There's one problem, though: all users can still access these files without having to go through the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`FilesController`. This is because these files still live in public/system, and therefore requests to them will be served by the webserver. These files need to move.

9.3.3 Public assets

People can still get to your files as long as they have the link provided to them because the files are still stored in the public folder. Let's see how this is possible by starting up the server using `bin/rails server`, signing in, and creating a ticket. Upload the spec/fixtures/spin.txt file as the only file attached to this ticket. You should see a ticket like the one in Figure 9.2

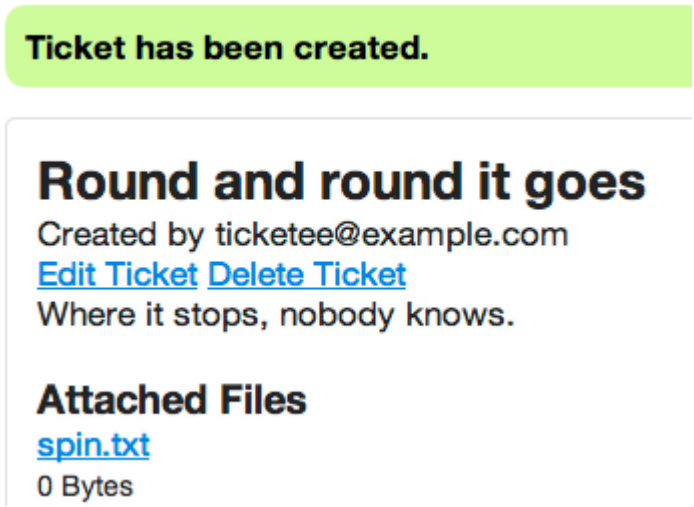

Figure 9.2 A ticket with spin!

Hover over the `spin.txt` link on this page and you'll see a link like this:

```
http://localhost:3000/system/assets/5/original/spin.txt?1282564953
```

As you saw earlier in this chapter, this link is not a route to a controller in your application but to a file inside the public directory. Any file in the public directory is accessible to the public. Sensible naming schemes rock!

If you copy the link to this file, sign out, and then paste the link into your browser window, you can still access it. These files need to be protected, and you can do that by moving them out of the public directory and into another directory at the root of your application called files. You should create this directory now.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

9.3.4 Privatizing assets

You can make these files private by storing them in the files folder. You don't have to move them there manually: you can tell CarrierWave to put them there by default by implementing the `store_dir` method on your `AssetUploader`. The generator provided us with a nice default implementation, let's change it to look like this:

```
def store_dir
  Rails.root + "uploads/#{model.class.to_s.underscore}/" + \
               "#{mounted_as}/#{model.id}"
end
```

Now try creating another ticket and attaching the spec/fixtures/spin.txt file. This time when you use the link to access the file, you're told there's no route. This is shown in Figure 9.3

Figure 9.3 No route!

The URL generated for this file is incorrect because we were linking to it wrong. Modify your app/views/tickets/show.html.erb:

```
<%= link_to File.basename(asset.asset.path), file_path(asset) %>
```

A great test to see if you can still upload assets correctly after this change is to run the scenario from spec/features/creating_tickets_spec.rb, which creates a ticket with three attachments. Run `bin/rspec spec/features/creating_tickets_spec.rb` now to see if this still works.

```
4 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Great! With this feature still passing, the files are being served through the `FilesController` controller correctly. You're done with implementing the functionality to protect assets from unauthorized access, so you should commit. First ensure that nothing is broken by running `rake spec`.

```
67 examples, 0 failures, 1 pending
```

The one pending spec here is in the file located at spec/helpers/files_helper_spec.rb. This file only contains this pending spec, so go ahead and delete it now. Another run of `rake spec` should give you this output:

```
66 examples, 0 failures
```

It's great to see everything is still in working order. Now commit and push your changes.

```
$ git add .
$ git commit -m "Uploaded Assets are now strictly served through
  the FilesController"
$ git push
```

By serving these files through the `FilesController`, you can provide a level of control over who can see them and who can't by allowing only those who have access to the asset's project to have access to the asset.

Inevitably, somebody's going to want to attach more than three files to a ticket, and then what? Well, you could add more fields until people stop asking for them, or you could be lazy and code a solution to save time. This solution entails putting an Add Another File link underneath the final file field in your form that, when clicked, add another file field. Users should be able to continue to do this ad infinitum. How do you implement this?

You use JavaScript. That's how.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

9.4 Using JavaScript

You started this chapter with only one file field, then moved to three after you realized users may want to upload more than one file to your application. While having three fields suits the purposes of many users, others may wish to upload yet more files.

You could keep adding file fields until all the users are satisfied, or you could be sensible about it and switch back to using one field and, directly underneath it, providing a link that, when clicked, adds another file field. Using this solution, you also clean up your UI a bit by removing possible extra file fields yet still allowing users to attach as many files as they like. This is where JavaScript comes in.

When you introduce JavaScript into your application, you have to run any scenarios that rely on it through another piece of software called WebDriver. WebDriver is a browser driver, which was installed when the Capybara gem was installed, so you don’t have to do anything to set it up. Capybara without WebDriver won’t run JavaScript because it doesn’t support it by itself. By running these JavaScript-reliant scenarios through WebDriver, you ensure the JavaScript will be executed. One of the great things with this WebDriver and Capybara partnership is that you can use the same old, familiar Capybara steps to test JavaScript behavior.

9.4.1 JavaScript testing

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 browser. Isn’t that neat? Let’s replace the "Creating a ticket with an attachment" scenario in this feature with the one from the following listing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 9.12 spec/features/creating_tickets_spec.rb

```
scenario "Creating a ticket with an attachment", js: true do
  fill_in "Title", with: "Add documentation for blink tag"
  fill_in "Description", with: "Blink tag's speed attribute"

  attach_file "File #1", "spec/fixtures/speed.txt"          ❶


  click_link "Add another file"
  attach_file "File #2", Rails.root.join("spec/fixtures/spin.txt")

  click_button "Create Ticket"

  expect(page).to have_content("Ticket has been created.")
                                                            ❷

  within("#ticket .assets") do
    expect(page).to have_content("speed.txt")
    expect(page).to have_content("spin.txt")
  end
end
```

The `js: true` tag at the top of this scenario tells Capybara that the scenario should use JavaScript, so it should be run in a browser using WebDriver. Also in this scenario, you've filled in only one file field ❶ because, as stated before, you're going to reduce the number of initial file fields to one. After filling in this field, you follow the Add Another File link that triggers a JavaScript event, which renders the second file field that you can then fill in. The rest of this scenario remains the same: ensuring that the ticket is created ❷ and that you can see the files inside the element with the class `assets`.

If you run this scenario with `bin/rspec spec/features/creating_tickets_spec.rb`, you'll see that signing in for this scenario is now not working:

```
Failure/Error: sign_in_as!(user)
expected there to be content "Signed in successfully." in "..."
```

That's strange. Why isn't this scenario working but the other ones still are? The only thing that was changed was the addition of the `js: true` option to the end

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

of the scenario. Well, that's exactly the problem! When you pass this option to a scenario, it will run the test using a real browser, and the real browser requires a real server to access. So what's actually happening here is that your test environment is now loading two Rails environments, one for normal testing and one for the JavaScript tests.

9.4.2 Cleaning the database

By itself, this isn't a big problem. But in combination with this line inside the spec/spec_helper.rb configuration:

```
config.use_transactional_fixtures = true
```

A perfect storm is created. What this line does is tells RSpec to run all the tests inside database transactions. This means that a new database transaction is begun at the same time as a test, and then a `ROLLBACK` command is issued to revert the database back to a clean state. This is why you can always test using the same database without having to clean out the data from inside it after the test run.

Due to this transaction, the data created by the test setup inside spec/features/creating_tickets_spec.rb will exist purely within this transaction. It's never committed to the database. So when the other Rails process spawns for the JavaScript testing and accesses the same database, it cannot see this test data and so this error happens.

The solution to this problem is to *not* use transactions within JavaScript tests. Instead, you should be using *data truncation*. Rather than running the tests inside a transaction block which is then rolled back at the end of it, data truncation involves an automatic wipe of the database at the end of the tests. To acheive this, you can use a gem called `database_cleaner`.

The `database_cleaner` gem can be installed into your application by adding this line inside the `test` group of your Gemfile:

```
gem 'database_cleaner', '1.0.1'
```

Run `bundle install` to install this gem now. To configure the gem, you should create a new file at spec/support/database_cleaning.rb and put the content from the following listing inside that file:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
RSpec.configure do |config|
  config.before(:suite) do
    DatabaseCleaner.strategy = :truncation
    DatabaseCleaner.clean_with(:truncation)
  end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end
end
```

You'll also need to change the `use_transactional_fixtures` line in spec/spec_helper.rb to set this option to `false`:

```
config.use_transactional_fixtures = false
```

With these changes, the database will now be truncated after every test run. When you run `bin/rspec spec/features/creating_tickets_spec.rb`, it fails because the "Add Another File" link doesn't yet exist:

```
And I follow "Add another file"
  no link with title, id or text 'Add another file' found
```

Before you fix it, however, let's make form render only a single asset field by changing this line in the `new` action in `TicketsController`:

```
3.times { @ticket.assets.build }
```

to this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
@ticket.assets.build
```

By building only one asset to begin with, you show users that they may upload a file. By providing the link to "Add Another File", you show them that they may upload more than one if they please. This is the best UI solution because you're not presenting the user with fields they may not use.

Now it's time to make the "Add Another File" link exist and do something useful!

9.4.3 Introducing jQuery

The "Add Another File" link, when clicked, will trigger an asynchronous call to an action, which renders a second file field. Every time this link is clicked, another file field will be added to the page.

For the "Add Another File" link to perform an asynchronous request when it's clicked, you can use the JavaScript framework called jQuery. This is already in use in your application, as your application's Gemfile references the `jquery-rails` gem, which provides the correct jQuery files. It's the task of your app/assets/javascripts/application.js file to include the two jQuery files Rails needs, which it does by using these two lines:

```
//= require jquery
//= require jquery_ujs
```

If you were to remove these two lines from application.js or if you were to remove the line that includes application.js in the application layout, things such as confirmation boxes on delete requests and asynchronous links would stop working. So please don't remove these lines! The line in app/views/layouts/application.html.erb is this one:

```
<%= javascript_include_tag "application" %>
```

It generates HTML like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<script src="/assets/application.js" type="text/javascript"></script>
```

The `/assets` path here is handled by the `sprockets` gem, which comes standard with Rails. When this route is requested, the Sprockets gem takes care of serving it. It begins by reading the `assets/javascripts/application.js` file, which specifies the following things:

```
//= require jquery
//= require jquery_ujs
//= require_tree .
```

The lines prefixed with `//` in this file (not shown in above example) are comments, but the lines prefixed with `//=` are *directives* that tell Sprockets what to do. These directives `require` the `jquery` and `jquery_ujs` files from the `jquery-rails` gem. The `jquery` file is the jQuery framework itself, while the `jquery-ujs` file provides *unobtrusive JavaScript* helpers for things such as the confirmation box that pops up when you click on a link that was defined using `link_to`'s `:confirm` helper. Again, if you removed this file this would stop working. Just be mindful of that, as it catches a lot of people out.

Rails has already required all the JavaScript files you need to get started here. Let's define the "Add Another File" link now.

9.4.4 Adding more files with JavaScript

You must add the "Add Another File" link to your tickets form at app/views/tickets/_form.html.erb. Put it underneath the `end` for the `fields_for` so it's displayed below existing file fields:

```
<%= link_to "Add another file",
  new_file_path,
  :remote => true,
  :update => "files",
  :position => "after"  %>
```

1. **Remote**
2. **Update**
3. **Position**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Here you use the `link_to` method to define a link, and you pass some options to it. The first option is `:remote => true`❶, which tells Rails you want to generate a link that uses JavaScript to make a background request, called an *asynchronous request*, to the server. More precisely, the request uses the JavaScript provided by the `jquery-ujs.js` file that comes with the `jquery-rails` gem.

This request then responds with some content, which is dealt with by the `:update` ❷ and `:position` ❸ options. The first option, `:update`, tells Rails to tell the JavaScript that handles the response that you want to insert the content from the background request into the element with the `id` attribute of `files`. The second, `:position`, tells it that you want to insert the content after any other content in the element, which would make your second file field appear after the first file field.

The element this updates doesn't currently exist, but you can easily create it by wrapping the `fields_for` inside a `div` with the `id` attribute set to `files`, as shown in the following listing.

Listing 9.13 app/views/tickets/_form.html.erb

```
<div id='files'>
  <%= f.fields_for :assets, :child_index => number do |asset| %>
    <p>
      <%= asset.label :asset, "File ##{number += 1}" %>
      <%= asset.file_field :asset %>
    </p>
  <% end %>
</div>
```

This `div` tag provides an element for your new `link_to` to insert a file field into. There's also been a change to the `fields_for` tag so that it uses the `:child_index` option to manually set the unique field identifier that is used for each of the file fields in this `fields_for` rendering. You'll see why this is done a little later on.

```
Failure/Error: attach_file "File #2", Rails.root.join...
Capybara::ElementNotFound:
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  cannot attach file, no file field with id, name,
  or label 'File #2' found
```

The "Add Another File" link currently uses the `new_file_path` helper, which generates a route such as `/files/new`. This route points to the `new` action in `FilesController`. This action isn't defined at the moment, so the feature won't work. Therefore, the next step is to define the action you need.

9.4.5 Responding to an asynchronous request

The job of the `new` action inside the `FilesController` is to render a single file field for the ticket form so users may upload another file. This action needs to render the fields for an asset, which you already do inside app/views/tickets/_form.html.erb by using these lines:

```
<p>
  <%= f.fields_for :assets, :child_index => number do |asset| %>
   <p>
     <%= asset.label :asset, "File ##{number += 1}" %>
     <%= asset.file_field :asset %>
   </p>
  <% end %>
</p>
```

To re-use this code for the `new` action in `FilesController`, move it into a partial located at app/views/files/_form.html.erb.

In app/views/tickets/_form.html.erb, you can replace the lines with this simple line:

```
<%= render partial: "files/form",
           locals: { number: number } %>                    (1)
```

When you pass the `:locals` (1) option to `render`, you can set local variables that can be used in the partial. Local variables in views are usable only in the views

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

or partials in which they're defined unless you pass them through by using this `:locals`. You pass through the number of your file field and the `asset` object provided by `fields_for :assets`.

To get the `new` action to render this partial, you can use the same code in the `new` action in `FilesController` but with a small change:

```
def new
  render partial: "files/form",
         locals: { number: params[:number].to_i }
end
```

Here you must pass the name of the partial using the `:partial` option so the controller will attempt to render a partial. If you left it without the option, the controller would instead try to render the template at app/views/files/form.html.erb, which doesn't exist.

Before this line, you need to set up the `asset` variable that you reference. Add these two lines directly above the first line inside the `new` action:

```
@ticket = Ticket.new
asset = @ticket.assets.build
```

Because the `Ticket` object for your form is only a new record, it isn't important precisely what object it is: all new `Ticket` objects are the same until they're saved to the database and given a unique identifier. You can exploit this by creating another `Ticket` object and building your new asset from it.

If you run the "Creating tickets" feature now using `bin/rspec spec/features/creating_tickets_spec.rb`, you will see this error come up:

```
Failure/Error: click_link "New Ticket"
ActionView::Template::Error:
  undefined local variable or method `f'
  for #<#<Class:0x007f89768bd330>:0x007f89788278d8>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This error is happening because in the new partial at app/views/files/_form.html.erb you are referencing the `f` variable without first defining it. This variable needs to represent a form builder object, and this can be done by wrapping the entire content in a `fields_for` block as shown in the following listing:

Listing 9.14 app/views/files/_form.html.erb

```
<%= fields_for @ticket do |f| %>
  <p>
    <%= f.fields_for :assets, child_index: number do |asset| %>
     <p>
       <%= asset.label :asset, "File ##{number += 1}" %>
       <%= asset.file_field :asset %>
     </p>
    <% end %>
  </p>
<% end %>
```

When you re-run `bin/rspec spec/features/creating_tickets_spec.rb`, you'll now see that's unable to see the second file field on this page:

```
Failure/Error: attach_file "File #2", Rails.root.join...
Capybara::ElementNotFound:
  Unable to find file field "File #2"
```

This is happening because the link to add another file field to the page is working, but the action of clicking that link is not returning any JavaScript that would modify the page. Let's fix this now using a little language called CoffeeScript.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

9.4.6 Sending parameters for an asynchronous request

The `number` variable indicates what file field you are up to, so you need a way to tell the `new` action in `FilesController` how many file fields are currently on the page. Previous versions of Rails had an option for this called `:with`, which has now, unfortunately, been removed. No matter, you can do it in JavaScript. It's better to put this code in JavaScript anyway, because it'll already be using some to determine the number to pass through. Rather than using pure JavaScript, you'll be using CoffeeScript, which comes with Rails but can be used with any other language. Let's learn some CoffeeScript now.

LEARNING COFFEESCRIPT

CoffeeScript is, in the words of its website, "a little language that compiles into JavaScript." It's written in a simple syntax, like this:

```
square = (x) -> x * x
```

This code compiles into the following JavaScript code:

```
var square;
square = function(x) {
  return x * x;
};
```

In the CoffeeScript version, you define a variable called `square`. Because this isn't yet initialized, it is set up using `var square;` in the JavaScript output. You assign a function to this variable, specifying the arguments using parentheses `(x)` and then specifying the code of the function after `->`. The code inside the function in this case is converted into literal JavaScript, making this function take an argument, multiply it by itself, and `return` the result.

Although this is a pretty basic example of CoffeeScript, it shows off its power. What you would write on four lines of JavaScript requires just one line of extremely easy-to-understand CoffeeScript.

Each time you generate a controller using Rails, a new file called app/assets/javascripts/[controller_name].js.coffee is created.[5] This file is created so

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you have a location to put CoffeeScript code that is specific to views for the relevant controller. This is really helpful in your situation, because you're going to use some CoffeeScript to tell your "Add Another File" link what to do when it's clicked.

Footnote 5 As long as you have the `coffee-rails` gem in your Gemfile.

Open app/assets/javascripts/tickets.js.coffee, and we'll build up your function line by line so you can understand what you're doing. Let's begin by putting this line first:

```
$(->
```

It seems like a random amalgamation of characters, but this line is really helpful. It calls the jQuery `$`[6] function and passes it a function as an argument. This line runs the function only when the page has fully loaded.[7] You need this because the JavaScript otherwise would be executed before the link you're going to be referencing is loaded. Let's add a second line to this:

Footnote 6 Aliased from the `jQuery` function: http://api.jquery.com/jquery/.

Footnote 7 For the meaning of "loaded," see this: http://api.jquery.com/ready

```
$ ->
  $('a#add_another_file').click(->
```

This line uses jQuery's `$` function to select an element on the page called `a`, which has an `id` attribute of `add_another_file` that will soon be your Add Another File link. This would happen only after the page is ready. After this, you call the `click` function on it and pass it a function that runs when you click on this link. Let's now add a third line:

```
$(->
  $('a#add_another_file').click(->
    url = "/files/new?number=" + $('#files input').length
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The double-space indent here indicates to CoffeeScript that this code belongs inside the function passed to `click`.[8] Here, you define a variable called `url`, which will be the URL you use to request a new file field on your page. At the end of this URL you specify the `number` parameter with some additional jQuery code. This code selects all the `input` elements inside the element on the page with the `id` attribute of `files` and stores them in an array. To find out how many elements are in that array, you call `length` on it. The URL for the first time you click on this link would now look something like `/files/new?number=1`, indicating that you already have one file field on your page.

Footnote 8 http://api.jquery.com/click/

Let's make the fourth line now:

```
$(->
  $('a#add_another_file').click(->
    url = "/files/new?number=" + $('#files input').length
    $.get(url,
```

This line is pretty simple; you call the jQuery function `$`, and then call the `get` [9] function on it, which starts an asynchronous request to the specified URL that is the first argument here, using the variable you set up on the previous line. Another line:

Footnote 9 http://api.jquery.com/jQuery.get/

```
$(->
  $('a#add_another_file').click(->
    url = "/files/new?number=" + $('#files input').length
    $.get(url,
      (data)->
```

This line is indented another two spaces again, meaning it is going to be an argument for the `get` function. This line defines a new function with an argument called `data`, which is called when the asynchronous request completes, with the `data` argument being the data sent back from the request. One more line:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
$(->
  $('a#add_another_file').click(->
    url = "/files/new?number=" + $('#files input').length
    $.get(url,
      (data)->
        $('#files').append(data)
```

This final line takes the `data` received from the request and `append`s[10] it to the end of the element that has the `id` attribute of `files` on this page. That's the one with the single file input field currently.

Footnote 10 http://api.jquery.com/append.

Finally, you need to close these functions you've defined, which you can do with three closing parentheses matching the levels of indentation, finalizing your code as this:

```
$(->
  $('a#add_another_file').click(->
    url = "/files/new?number=" + $('#files input').length
    $.get(url,
      (data)->
          $('#files').append(data)
    )
  )
)
```

That's all there is to it! When your server receives a request at `/assets/application.js`, the request will be handled by the Sprockets gem. The Sprockets gem will then combine the `jquery`, `jquery_ujs`, and the app/assets/javascripts/tickets.js.coffee into one JavaScript file, parsing the CoffeeScript into the following JavaScript:

```
(function() {
  $(function() {
    return $('a#add_another_file').click(function() {
      var url;
      url = "/files/new?number=" + $('#files input').length;
      return $.get(url, function(data) {
        return $('#files').append(data);
      });
    });
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  });

}).call(this);
```

You can see this JavaScript in action if you start up your server using `rails s` and then go to http://localhost:3000/assets/tackets.js.

In the `production` environment, this file is compiled upon the first request and then cached to save valuable processing time.

This is a little more verbose than the CoffeeScript and another great demonstration of how CoffeeScript allows you to write more with less. For more information and usage examples of CoffeeScript, see the CoffeeScript site: http://coffeescript.org .

Let's now give your link the `id` attribute that's required to get this working so we can move on.

PASSING THROUGH A NUMBER

Open your app/views/tickets/_form.html.erb and replace the code for your "Add Another File" link with this:

```
<%= link_to "Add another file", 'javascript:',
  id: "add_another_file" %>
```

This gives the element the `id` attribute you require. Let's witness this JavaScript in action now by running `rails server` to start up a server, signing in using the email address *ticketee@example.com* and the password *password*, and then creating a ticket on a project. Clicking on the "Add Another File" results in an error that you'll fix shortly. Click on it anyway. Afterwards, go back to the window where `rails server` is running.

This window shows information such as queries and results for every request, but you're only interested in the last request made. This request should begin with the following line:

```
Started GET "/files/new?number=1...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This line tells you that Rails has begun serving a `GET` request to the `/files/new` route with a bunch of URL parameters. Your `number` parameter is the first one in this example. The following lines show you the URL that was requested as well as what action and controller served this request:

```
Started GET "/files/new?number=1" for 127.0.0.1 at [timestamps]
  Processing by FilesController#new as */*
  Parameters: {"number"=>"1"}
```

The line you're most interested in is the third line:

```
Parameters: {"number"=>"1", ... }
```

This is the `params` hash output in a semi-human-readable format. Here you can see it has the `number` parameter, so you can use this inside the `new` action. With all this in mind, you can change how you render the partial in the `new` action inside `FilesController` to this:

```
render partial: "files/form",
       locals: { number: params[:number].to_i,
                 asset: asset }
```

You must convert the `number` parameter to an integer using the `to_i` method because it'll be a `String` when it comes from `params`. It needs to be a `Fixnum` so the partial can increment it by 1 to use that in the generation of the next id of the field.

Now if you refresh this page and attempt to upload two files, you should see that it works. Does your scenario agree? Let's find out by running `bin/rspec spec/features/creating_tickets_spec.rb`:

```
Failure/Error: attach_file "File #2", Rails.root....
Capybara::ElementNotFound:
 Unable to find file field "File #2"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Wait, what? But it was right there! You can even see the link to add a file right there! When we first saw this error, we were stumped. Big time. But that sentence before was the key: the link to add a file was there, but nothing happened when Capybara clicked on it. This generally means that the JavaScript isn't working. So, we fired up a web browser and a `bin/rails s` and checked it out...

And it worked. What in the world is going on here?

After thinking long and hard, and asking a few friends, we figured it out. The issue? Turbolinks.

Turbolinks is a gem that comes automatically installed with Rails 4. It attempts to speed up your site by overriding the links in your page with JavaScript.

So here's what's happening: When we loaded up the page in our browser, we went straight to the 'New Ticket' page. The page loads, our handler gets set, and it all works. When the tests run, they click through several pages in order to get to the 'New Tickets' page. This means Turbolinks handles the page loads, and therefore the browser's native `onload` event never gets sent, which means that the handler doesn't get attached, and so clicking on the link does nothing.

Now, Turbolinks provides its own event to replace the `onload` event, so that this kind of code can work properly. However, it's our opinion that Turoblinks is great to speed up mostly-server-side sites, but as soon as you start writing some JavaScript, it causes more problems than it's worth. Therefore, we're going to remove Turbolinks from our application, and the tests should pass.

Removing Turbolinks is really easy. There are three places you have to remove lines. First, remove the `require 'turbolinks'` line from your Gemfile. Run `bundle update` afterwards to update the lock file so it's truly gone. Then, remove the `//= require turbolinks` from your app/assets/javascripts/application.js. Finally, change these two lines in your app/views/layouts/application.html.erb to look like this:

```
<%= stylesheet_link_tag    "application", media: "all" %>
<%= javascript_include_tag "application" %>
```

With all of that done, try running `bin/rspec spec/features/creating_tickets_spec.rb` another time:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
4 examples, 0 failures
```

Yup, all working! Great. You've now switched the ticket form back to only providing one file field but providing a link called Add Another File, which adds another file field on the page every time it's clicked. You originally had implemented this link using the `:remote` option for `link_to`, but switched to using CoffeeScript when you needed to pass the `number` parameter through. A couple of other small changes, and you got it all working very neatly again!

This is a great point to see how the application is faring before committing. Let's run the tests with `rake spec`. You should see the following:

```
66 examples, 0 failures
```

Awesome! Let's commit it:

```
git add .
git commit -m "Provide an 'Add another file' link that uses Javascript
               so that users can upload more than one file"
git push
```

This section showed how you can use JavaScript and CoffeeScript to provide the user with another file field on the page using some basic helpers. JavaScript is a powerful language and is a mainstay of web development that has gained a lot of traction in recent years thanks to libraries such as the two we saw here, jQuery and CoffeeScript, as well as others such as Prototype and Raphael.

By using JavaScript, you can provide some great functionality to your users. The best part? Just as you can test your Rails code, you can make sure JavaScript is working by writing tests that use WebDriver.

9.5 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 adding the `file_field` helper to your view, making your form multipart, and by using the CarrierWave gem to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

handle the file when it arrives in your application.

After you conquered single file uploading, you tackled multiple file uploading. You offloaded the file handling to another class called `Asset`, which kept a record for each file you uploaded. You passed the files from your form by using nested attributes, which allowed you to create `Asset` objects related to the ticket being created through the form.

After multiple file uploading, you learned how to restrict which files are served through your application by serving them through a controller. By using a controller, you can use CanCan's `can?` helper to determine if the currently signed-in user has access to the requested asset's project. If so, then you give the user the requested asset using the `send_file` controller method. If not, you deny all knowledge of the asset ever having existed.

Finally, you used a JavaScript library called jQuery, in combination with a simpler way of writing JavaScript called CoffeeScript, to provide users with an "Add Another File" link which they could click every time they wanted to add another file to the form. jQuery does more than simple asynchronous requests, though, and if you're interested, the documentation[11] is definitely worth exploring.

Footnote 11 http://jquery.com.

In the next chapter, we 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.

Index Terms

accepts_nested_attributes_for
Callbacks, before_save
fields_for
file_field
git commit, --amend option
link_to, :position option
link_to, :remote option
link_to, :update option
number_to_human_size helper
present?, ActiveRecord::Base
render, :locals option
render, :partial option
send_file

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

10 Tracking State

In a ticket-tracking application such as Ticketee, tickets aren't there to provide information of a particular problem or suggestion; rather, they're there to provide the workflow for it. The general workflow of a ticket is that a user will file it and it'll 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, mark it as "resolved." If a ticket needs more information on it then 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 worthwhile putting in. In cases such as this the ticket may be marked as "duplicate" or "invalid," respectively.

The point is that tickets have a workflow, and that workflow revolves around state changes. We'll allow the admin users of this application to add states, but not to delete them. The reason for this is if an admin were to delete a state that was used then we'd have no record of that state ever existing. It's best if, once states are created and used on a ticket, that they can't be deleted.[1]

Footnote 1 Alternatively, these states could be moved into an "archive" state of their own so they couldn't be assigned to new tickets but still would be visible on older tickets.

To track the states we'll let users leave a comment. With a comment, users will be able to leave a text message about the ticket and may also elect to change the state of the ticket to something else by selecting it from a drop-down box. But not all users will be able to leave a comment and change the state. We will protect both creating a comment and changing the state.

By the time you're done with all of this, the users of your application will have the ability to add comments to your tickets. Some users, due to permission restriction, will be able to change the state of a ticket through the comment

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

interface.

We'll begin with creating that interface for a user to create a comment and then build on top of that the ability for the user to change the state of a ticket while adding a comment. Let's get into it.

10.1 Leaving a comment

Let's get started by adding the ability to leave a comment. When you're done you will have a simple form that looks like Figure 10.1

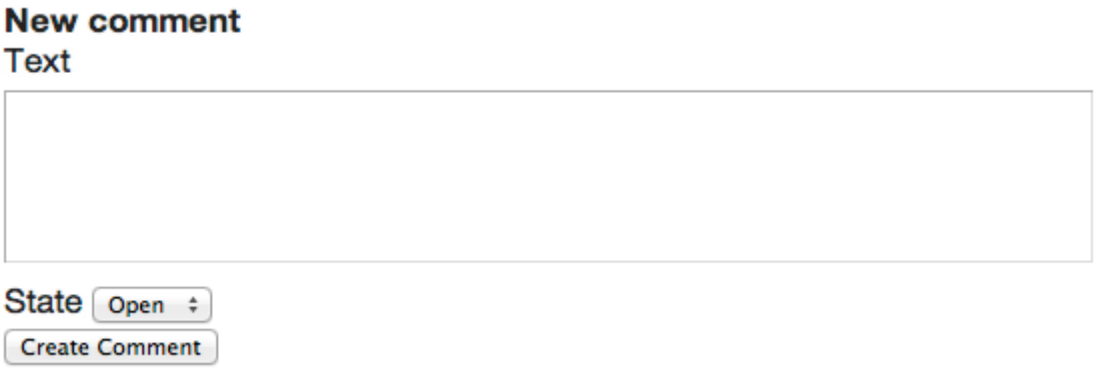

Figure 10.1 The comment form

To get started with this you'll write a Capybara feature that goes through the process of creating a comment. When you're done with this feature you will have a comment form at the bottom of the `show` action for the `TicketsController` which you'll then use as a base for adding your state drop-down box to later on. You'll put this feature in a new file at spec/integration/creating_comments_spec.rb and make it look like the following listing.

Listing 10.1 spec/integration/creating_comments_spec.rb

```
require 'spec_helper'

feature "Creating comments" do
  let!(:user) { Factory(:confirmed_user) }
  let!(:project) { Factory(:project) }
  let!(:ticket) { Factory(:ticket, :project => project, :user => user) }

  before do
    define_permission!(user, "view", project)

    sign_in_as!(user)
    visit '/'
    click_link project.name
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  scenario "Creating a comment" do
    click_link ticket.title
    fill_in "Text", :with => "Added a comment!"
    click_button "Create Comment"
    page.should have_content("Comment has been created.")
    within("#comments") do
      page.should have_content("Added a comment!")
    end
  end

  scenario "Creating an invalid comment" do
    click_link ticket.title
    click_button "Create Comment"
    page.should have_content("Comment has not been created.")
    page.should have_content("Text can't be blank")
  end
end
```

Here you navigate from the homepage to the ticket page by following the respective links, fill in the box with the label “Text,” and create your comment. You’ve put the link to the ticket inside the scenarios rather than the `before` because you’ll use this feature for permission checking later on. Let’s try running this feature now by running `bin/rspec spec/integration/creating_comments_spec.rb`. You'll see this output:

```
Failure/Error: fill_in "Text", :with => "Added a comment!"
Capybara::ElementNotFound:
  cannot fill in, no text field, text area or password field
  with id, name, or label 'Text' found
```

This failing step means that you’ve got work to do! The label it’s looking for is going to belong to the comment box underneath your ticket’s information. Both the label and the field aren't there, and that's what's the scenario requires, so now's a great time to add them.

10.2 The comment form

Let’s begin to build this comment form for the application, the same one you’ll eventually add a state select box to to complete this feature. This comment form will consist of a single text field with which the user can insert their comment.

Let’s add a single line to the bottom of app/views/tickets/show.html.erb to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

render a comment form partial:

```
<%= render "comments/form" %>
```

This line renders the partial from app/views/comments/_form.html.erb which you'll now create and fill with the content from the following listing.

Listing 10.2 app/views/comments/_form.html.erb

```
<strong>New comment</strong>
<%= form_for [@ticket, @comment] do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :text %><br>
    <%= f.text_area :text %>
  </p>
  <%= f.submit %>
<% end %>
```

Pretty much the standard `form_for` here, 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.

Firstly, you must define the `@comment` variable that references a new `Comment` instance in the `show` action inside `TicketsController` so this `form_for` has something to work with.

Secondly, has hinted before, you'll need to create the `Comment` model and associate this with your `Ticket` model so you can create new records from the data in the form and associate it with the right ticket.

Thirdly, you need to define the nested resource so that the `form_for` knows to POST to the correct URL, one similar to /tickets/1/comments. Without this we will run into an undefined method of `ticket_comments_path` when the `form_for` tries to generate the URL by combining the classes of the objects in the array for its `action`.

Finally, you'll 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.

Now set up your `TicketsController` to use the `Comment` model for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

creating new comments, which you'll create shortly afterwards. To do this, you need to first build a new `Comment` object using the `comments` association on your `@ticket` object.

10.3 The comment model

The first step to getting this feature to pass is to set up the `show` action in your `TicketsController` to define a `@comment` variable for the comment form. To do this, you'll change the `show` action, as shown in the following listing.

Listing 10.3 app/controllers/tickets_controller.rb

```
def show
  @comment = @ticket.comments.build
end
```

This will use the `build` method on the `comments` association for your `@ticket` object (which is set up by the `find_ticket before_filter`) to create a new `Comment` object for the view's `form_for`.

Next, you'll generate the `Comment` model so that you can define the `comments` association on your `Ticket` model. This model's going to 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. Let's generate this model using this command:

```
rails g model comment text:text ticket_id:integer user_id:integer
```

Then you'll run the migration for this model on both your development and test databases by running these familiar commands:

```
rake db:migrate
rake db:test:prepare
```

With these done, your next stop is to add the associations to the `Ticket` and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`Comment` models. For this, you add this line to app/models/ticket.rb directly under the `accepts_nested_attributes_for :assets` line:

```
has_many :comments
```

Add a validation to your `Comment` model to validate the presence of text for the records by adding this line to app/models/comment.rb:

```
validates :text, :presence => true
```

This will help your second scenario pass, because it requires that an error message is displayed when you don't enter any text. You'll also add a `belongs_to` association definition to this model too, given that you've a `user_id` column in your `comments` table:

```
belongs_to :user
```

While you're in this model, change the `attr_accessible` line from this:

```
attr_accessible :text, :user_id, :ticket_id
```

Into this:

```
attr_accessible :text
```

The `user_id` and `ticket_id` attributes should not be mass-assignable by users of this application, just the `text` attribute should be, since that's the only field available in the form.

When you run your feature with `bin/rspec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`spec/integration/creating_comments_spec.rb` at this mid-point you'll be told that it can't find the routing helper that `form_for` is trying to use:

```
Failure/Error: click_link ticket.title
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 do
  resources :tickets
end
```

This generates helpers such as `project_tickets_path`. But for your form it's not important what comment the project is being created for, so you use `ticket_comments_path` instead. This means you'll need to define a separate nonnested resource for your tickets and then a nested resource under that for your comments, as shown in the following listing.

Listing 10.4 config/routes.rb

```
resources :projects do
  resources :tickets
end

resources :tickets do
  resources :comments
end
```

The last three lines in Listing 10.4 are the lines we need in order for `ticket_comments_path` to be defined which will make your form work. With a route now defined you'll need to define the related controler for that route.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

10.4 The comments controller

Now finally we need to generate the `CommentsController` so that our form has somewhere to post to. We 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 we should add this now. We'll need to define two `before_filter`s in this controller. The first is to to ensure the user is signed in because we don't want anonymous users creating comments and another is to find the `Ticket` object. This entire controller is shown in the following listing.

Listing 10.5 app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController
   before_filter :authenticate_user!
   before_filter :find_ticket

   def create
     @comment = @ticket.comments.build(params[:comment])
     @comment.user = current_user
     if @comment.save
       flash[:notice] = "Comment has been created."
       redirect_to [@ticket.project, @ticket] <co id="ch10_v2_5_1"/>
     else
       flash[:alert] = "Comment has not been created."
       render :template => "tickets/show" <co id="ch10_v2_5_2"/>
     end
   end

   private

     def find_ticket
       @ticket = Ticket.find(params[:ticket_id])
     end
 end
```

In this action you use the `template` option of `render` when your `@comment.save` returns `false` to render a template of another controller. Previously you've used the `action` option to render templates that are for the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

current controller. By doing this, the `@ticket` and `@comment` objects will be available when the app/views/tickets/show.html.erb template is rendered.

If the object 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, like `form_for` does to a nested route similar to `/projects/1/tickets/2`.

But if the object doesn't save successfully you want it to render the template that `TicketsController`'s `show` action renders. You can do this by using the `render` method and passing it "tickets/show" . Keep in mind that the `render` method doesn't call the action, and so any code within the `show` method of `TicketsController` wouldn't be run. This is fine, though, as you're setting up the `@ticket` variable the template renders by using the `find_ticket` before filter in your controller.

By creating the controller you've now got all the important parts needed to create comments. Let's run this feature again by running `bin/rspec spec/integration/creating_comments_spec.rb` to see how you're progressing. You'll see that it's able to create the comment but it's unable to find the text within the `#comments` element on the page.

```
Failure/Error: within("#comments") do
  Capybara::ElementNotFound:
  Unable to find css "#comments"
```

This step is failing because you haven't added the element with the `id` attribute of "comments" to the `show` template yet. This element will contain all the comments for a ticket. Let's add it by adding the code from Listing 10.6 above the spot where you render the comment form partial.

Listing 10.6 app/views/tickets/show.html.erb

```
<h3>Comments</h3>
<div id='comments'>
  <% if @ticket.comments.exists? %> <co id="ch10_190_1"/>
    <%= render @ticket.comments.select(&:persisted?) %>
  <% else %>
    There are no comments for this ticket.
  <% end %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
</div>
<%= render "comments/form" %>
```

Here you create the element the scenario requires: one with an `id` attribute of `comments`. In this we check if there are no comments by using the `exists?` method from Active Record. This will do a light query similar to this to check if there are any comments:

```
SELECT "comments"."id" FROM "comments"
WHERE ("comments".ticket_id = 1) LIMIT 1
```

It only selects the “id” column from the comments table and limits the result set to 1, which results in a super-fast query to check if there’s any comments at all. We used `exists?` back in chapter 8 when we checked if a ticket had any assets. You could use `empty?` here instead, but that would load the `comments` association in its entirety and then check to see if the array is empty. If there were a lot of comments, then this would be slow. By using `exists?`, 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`. On the end of that call `select` on it.

You use `select` here because you don’t want to render the comment object you’re building for the form at the bottom of the page. If you left off the `select`, `@ticket.comments` would include this new object and render a blank comment box. When you call `select` on an array, you can pass it a block which it will evaluate on all objects inside that array and return any element which makes the block evaluate to anything that’s not `nil` or `false`.

The argument you pass to `select` is called a *Symbol-to-Proc* and is a shorter way of writing this:

```
{ |x| x.persisted? }
```

This is a new syntax versions of Ruby >= 1.8.7 and used to be in Active Support in Rails 2. It’s a handy way of writing a shorter block syntax if we’re only

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

looking to call a single method on an object.

The `persisted?` method checks if an object is persisted in the database by checking if it has its `id` attribute set and will return `true` if that's the case and `false` if not.

By using `render` in this form, Rails will render a partial for every single element in this collection and will 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, but you don't have this file right now. Let's create it and fill it with the content from the following listing.

Listing 10.7 app/views/comments/_comment.html.erb

```
<%= div_for(comment) do %>
  <h4><%= comment.user %></h4>
  <%= simple_format(comment.text) %>
<% end %>
```

Here you've used a new method, `div_for`. This method generates a `div` tag around the content in the block and also sets a `class` and `id` attribute based on the object passed in. In this instance, the `div` tag would be the following:

```
<div id="comment_1" class="comment">
```

The `class` method from this tag is used to style your comments so that they will look like Figure 10.2 when the styles from the stylesheet are applied.

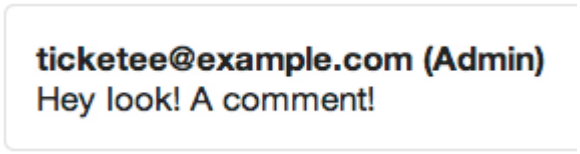

Figure 10.2 A comment

With the code in place not only to create comments but also display them, your feature should pass when you run it with `bin/rspec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`spec/integration/creating_comments_spec.rb`:

```
2 examples, 0 failures
```

Good to see. You're now giving users the ability to leave comments on a ticket. Before proceeding further, you should make sure that everything is working as it should by running `rake spec`, and you should also commit your changes. When you run the tests we'll see this output:

```
68 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 these two files now, as they don't contain any useful tests. If you re-run `rake spec` you'll see this:

```
66 examples, 0 failures
```

Good stuff! Let's only commit and push this:

```
git add .
git commit -m "Users can now leave comments on tickets"
git push
```

With this form added to the ticket's page, users are now able to leave comments on tickets. 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, we'll look at adding another way to provide additional context to this ticket by adding states.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

10.5 Changing a ticket's state

States provide a helpful way of standardizing the way that a ticket's progress is tracked. By glancing at the state of a ticket, a user will be able to determine if that ticket needs more work or if it's complete, as shown in Figure 10.3

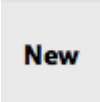

Figure 10.3 A ticket's state

To change a ticket's state, you'll add a drop-down box on the comment form where a user can select a state from a list of states. These states will be stored in another table called `states` and they'll be accessed through the `State` model.

Eventually, you'll let some users of the application have the ability to add states for the select box and make one of them the default. For now you'll focus on creating the drop-down box so that states can be selected.

As usual, you'll cover creating a comment that changes a ticket's state by writing another scenario. The scenario you'll now write will go at the bottom of spec/integration/creating_comments_spec.rb and it's shown in the following listing.

Listing 10.8 spec/integration/creating_comments_spec.rb

```
scenario "Changing a ticket's state" do
  fill_in "Text", :with => "This is a real issue"
  select "Open", :from => "State"
  click_button "Create Comment" <co id="ch10_234_1"/>
  page.should have_content("Comment has been created.")
  within("#ticket .state") do
    page.should have_content("Open")
  end
end
```

In this scenario you go through the process of creating a comment, much like in the first "Creating comments" scenario, only this time you select a state.❶ This is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the first part of the scenario that you can expect to fail because you don't have a state select box yet. After the comment is created, you should see the state appear in the "#ticket .state" area. This is the second part of the scenario that will fail.

When you run this scenario by running `bin/rspec spec/integration/creating_comments_spec.rb` it will fail like this:

```
Failure/Error: select "Open", :from => "State"
  Capybara::ElementNotFound:
  cannot select option, no select box with id, name,
    or label 'State' found
```

As you can see from this output, the "I select" step will attempt to select an option from a select box. In this case, it 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 pressing 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.

10.5.1 Creating the state model

Right now you need to add a select box. When you're done, you should have one that looks like Figure 10.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

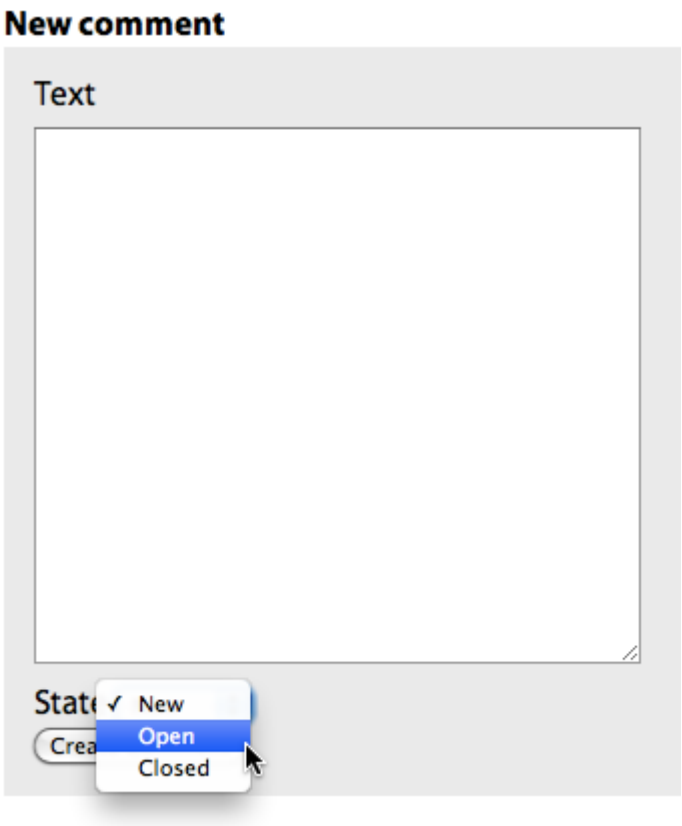

Figure 10.4 State select box

Before adding this select box, let's set the `TicketsController`'s `show` action up to return a collection of states that you can populate the drop select box with. You'll change the `show` action inside `TicketsController` to be like this now:

```
def show
  @comment = @ticket.comments.build
  @states = State.all
end
```

Here you call `all` on the `State` class, which doesn't exist yet. You'll be storing the states in a table because we'd like the 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 two other fields: `color` and `background` which will define the colors of the label for this state. Later on, you'll add a `position` field which

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you'll use to determine the sort order of the states in the select box on the comment form. Let's create this `State` model and the associated migration by running this command:

```
rails g model state name:string color:string background:string
```

Before running this migration, you'll need to define a way that states link to comments and to tickets, but there's a couple of things worth mentioning beforehand. For comments, you want to track the previous state so you can display that a comment has changed the ticket's state. For tickets, you want to track the current state of the ticket which you'll use a foreign key for.

With all of this in mind, let's add these fields to the migration now. You'll 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 10.9 db/migrate/[date]_create_states.rb

```
class CreateStates < ActiveRecord::Migration
  def change
    create_table :states do |t|
      t.string :name
      t.string :color
      t.string :background
    end

    add_column :tickets, :state_id, :integer
    add_index :tickets, :state_id

    add_column :comments, :state_id, :integer
  end
end
```

In this migration you use the `add_index` method 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 made that search for tickets that have a particular value in this field. The side-effect of indexing is that it will result in slower writes and more disk-space. It's always important to have indexes on nonprimary-key fields[2]

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

because of this great read speed increase, as applications generally read from the database more often than write to it.

Footnote 2 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.

Let's run this migration now by running these two commands:

```
rake db:migrate
rake db:test:prepare
```

There you have it! The `State` model is up and running. Let's now associate this class with the `Comment` class by adding this line to the top of the `Comment` model's definition:

```
belongs_to :state
```

The `state` method provided by this `belongs_to` will be used shortly to display the state on the ticket page, like Figure 10.5

Figure 10.5 A ticket's state

Before doing that, however, you'll need to add the select box for the state to the comment form.

10.5.2 Selecting states

In our comment form partial you'll add this select box underneath the text box, as shown in the following listing.

Listing 10.10 app/views/comments/_form.html.erb

```
<p>
  <%= f.label :state_id %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  <%= f.select :state_id , @states.map { |s| [s.name, s.id] } ,
  <co id="ch10_286_1"/>
                :selected => @ticket.state_id %> <co id="ch10_286_2"/>
</p>
```

Here you use a new method, `select` , which takes its first argument as the foreign-key *attribute* of your `Comment` object, not the association. You also use the `:state_id` value for the argument passed to the `label`, but Rails is smart enough to know the text for this should be "State". `select`'s second argument is a two-dimensional[3] `Array` which you create by using `map` on the `State` objects returned from the controller in the `@states` variable. The first element of each array is the value you want shown as an option in the select box to the user, whereas the second element is the value that's passed back to the controller.

Footnote 3 A 2-dimensional array is an array that contains arrays as elements.

Use the `:selected` option in the `select` call to select the current state of the ticket from the list. This value must match the value argument for one of the options in the select box, otherwise it will default to the first option.

Let's assume for a moment that you've got three states: "New," "Open," and "Closed." For a ticket that has its state set to "New," the select box generated by `f.select` would look like this:

```
<select id="comment_state_id" name="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, the `option` will be the one selected as the default option for the `select`. This is achieved by using the `:selected` option for `f.select`. The value for this option is the corresponding `value` attribute for the `option` tag. In this case it's the `state_id` of the `@ticket` object.

With the select box in place you're almost at a point where this scenario will be passing. Let's see how far you've gotten by running `bin/rspec spec/integration/creating_comments_spec.rb`. It won't be able to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

find the “Open” option in your select box.

```
Failure/Error: select "Open", :from => "State"
 Capybara::ElementNotFound:
   cannot select option, no option with text
   'Open' in select box 'State'
```

This is because you need to add a state to your database! Let’s add this line underneath the call to `define_permission!` in the `before` block of spec/integration/creating_comments_spec.rb to do this:

```
Factory(:state, :name => "Open")
```

For this to work, you will need to define a state factory. Go ahead and do that in a new file called spec/support/factories/state_factory.rb using the content from the following listing:

Listing 10.11 spec/support/factories/state_factory.rb

```
FactoryGirl.define do
  factory :state do
    name "A state"
  end
end
```

Now with the state factory defined, when you re-run `bin/rspec spec/integration/creating_comments_spec.rb`, the final scenario will fail with this error:

```
Failure/Error: click_button "Create Comment"
ActiveModel::MassAssignmentSecurity::Error:
  Can't mass-assign protected attributes: state_id
```

This error is happening because the `state_id` field is not mass-assignable for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

comment instances, but it should be. To make this happen, go into the `Comment` model and add this attribute to the `attr_accessible` list:

```
attr_accessible :text, :state_id
```

The `user_id` and `ticket_id` attributes have been removed here because they should not be mass-assignable by the form. This will fix the error that was causing the scenario to fail, so re-run it again with `bin/rspec spec/integration/creating_comments_spec.rb` to find out what to do next:

```
Failure/Error: within("#ticket .state") do
Capybara::ElementNotFound:
   Unable to find css "#ticket .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.

Rather than putting the state inside the `TicketsController`'s `show` template, put it in a partial. This is due to the fact that you'll be reusing this to display a state wherever we need it in the future. Additionally, we'll apply a dynamic class around the state so we can style it later on. Let's create a new partial at app/views/states/_state.html.erb and fill it with this content:

```
<div class='state state_<%= state.name.parameterize %>'>
  <%= state %>
</div>
```

To style the element you need a valid CSS class name. You can get one by using the `parameterize` method. If, for example, you had a state called "Drop bears strike without warning!," and used `parameterize` on it, all the spaces and non-url-valid characters would be stripped, leaving you with "drop-bears-strike-without-warning," which is a perfectly valid CSS class name.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You'll use this later on to style the state using the `color` and `background` attributes.

You're now going to render this partial underneath the "Created by" line on app/views/tickets/show.html.erb using the following line:

```
<%= render @ticket.state if @ticket.state %>
```

You're using the short form of rendering a partial here once again, and you conditionally render it if the ticket has a state. If you don't have the `if` at the end and the state is `nil`, this will raise an exception because it will try to determine the model name of `nil`.

To get this `state` method for your `Ticket`, you should add the association method to the model. This method should go directly above the `belongs_to :user` line in app/models/ticket.rb:

```
belongs_to :state
```

If you run the feature again with `bin/rspec spec/integration/creating_comments_spec.rb` it will fail because there's nothing shown in the `#ticket .state` element:

```
And I should see "Open" within "#ticket .state"
<false> is not true. (Test::Unit::AssertionFailedError)
```

This is because you're updating the state on the `Comment` object you're creating, not the associated `Ticket` object! You're trying to get the new state to display on the ticket object so that the users of the application can change the state of a ticket when they add a comment to it. For this to work, you'll need to define a *callback* in your `Comment` model.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

10.5.3 Callbacks

When a user selects a state from the drop-down box attached to the comment form on a ticket's page, you want that ticket's state to be updated with what that user picked.

To do this you can use a callback to set the ticket's status when you change it through the comment form. A callback is a method that's called either before or after a certain event. For models, there are before-and-after callbacks for the following events (where * can be substituted for either `before` or `after`):

- Validation (`*_validation`)
- Creating (`*_create`)
- Updating (`*_update`)
- Saving (`*_save`)
- Destruction (`*_destroy`)

We're able to trigger a specific piece of code or method to run before or after any of these events. The "Saving" item in the above list refers to when a record is saved to the database, which occurs when a record is created or updated. For your `Comment` model you'll want to define a callback that occurs after a record has been created and for this you'll use the `after_create` method at the top of your `Comment` model, as well as a `ticket` association, transforming this model into the code shown in the following listing.

Listing 10.12 app/models/comment.rb

```
class Comment < ActiveRecord::Base
  after_create :set_ticket_state

  belongs_to :ticket
  belongs_to :user
  belongs_to :state
  validates :text, :presence => true
end
```

While you're here, you can also set it up so that you can access the `project` association that the `ticket` association has in this model by using the `delegates` method:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
delegate :project, :to => :ticket
```

If you were to call the `project` method on a `Comment` object, this method will "delegate" the project method to the `ticket` object, making a call exactly like `ticket.project`. This makes your code shorter and will come in handy later on.

The symbol passed to the `after_create` method here is the name of the method to call for this callback. You can define this method at the bottom of your `Comment` model using the code from the following listing.

Listing 10.13 app/models/comment.rb

```
class Comment < ActiveRecord::Base
  ...
  private

    def set_ticket_state
      self.ticket.state = self.state
      self.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. When you run your feature again by running `bin/rspec spec/integration/creating_comments_spec.rb`, it still fails:

```
And I should see "Open" within "#ticket .state"
   Failed assertion, no message given. (MiniTest::Assertion)
```

Even though you're correctly assigning the state to the ticket, it still doesn't display as the state in the view. But why is this? You can attempt to duplicate this issue by running the server using the `rails server`. By visiting http://localhost:3000 we can follow the steps inside the scenario to attempt to duplicate the behavior you've seen in your feature.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Because you have no states in the development database, you won't be able to reproduce this problem right away. Yur feature uses the "Given there is a state called..." steps to define states, but you can't use these in your development environment. It would be better if you added seed data to your database because then you'll have a repeatable way of setting up the states in your application's database.

10.5.4 Seeding states

If you add some states to the db/seeds.rb file, users will be able to select them from the "State" drop-down box on the tickets page rather than leaving it blank and useless, much like it is now. With these states in the db/seeds.rb file, as mentioned before, you will have a repeatable way of creating this data if you ever needed to run your application on another server, such as would be the case when you put the application on another computer.

You're adding these files to the db/seeds.rb so you have some to "play around" with in the development environment of your application. You're attempting to figure out why, when a user picks "Open" from the state select box and presses "Create Comment," the state doesn't display on the ticket that should be updated.

When you go to the "Ticketee Beta" project to create a ticket and then attempt to create a comment on that ticket with the state of "Open," you'll see that there are no states (as shown in Figure 10.6).

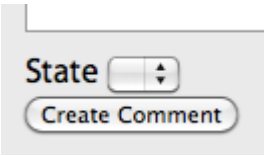

Figure 10.6 Oops! No states!

You should add a couple of states to your seeds file now; they'll be "New," "Open," and "Closed." Ideally, "New" will be the default state of tickets and you'll set this up a little later on. Before adding these states, let's add a couple of steps to your features/seed.feature to always ensure that your states are defined.

You'll extend this feature to go inside the "Ticketee Beta" project, create a ticket, and then begin to create a comment on that ticket. When it's on the comment creation screen, you'll check to see that all your states are in the state box. To do this, you'll modify the `scenario` block in this file to what's shown in the following listing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 10.14 spec/integration/seeds_spec.rb

```
scenario "The basics" do
  load Rails.root + "db/seeds.rb"
  user = User.find_by_email!("admin@ticketee.com")
  sign_in_as!(user)
  click_link "Ticketee Beta"
  click_link "New Ticket"
  fill_in "Title", :with => "Comments with state"
  fill_in "Description", :with => "Comments always have a state."
  click_button "Create Ticket"
  within("#comment_state_id") do
    page.should have_content("New")
    page.should have_content("Open")
    page.should have_content("Closed")
  end
end
```

The `#comment_state_id` element referenced here is the state select box for your comments. It has this id because it's inside the form for a comment, and the field is called `state_id`. Uou’re confirming that this state select box has got the three states you’re going to be seeding your database with. When you run this feature by running `bin/rspec spec/integration/seeds_spec.rb` it will fail because you don’t have your states yet:

```
Then I should see "New" within "#comment_state_id"
  <false> is not true. (Test::Unit::AssertionFailedError)
```

Let’s add these states to your db/seeds.rb file by using the lines shown in the following listing.

Listing 10.15 db/seeds.rb

```
State.create(:name       => "New",
             :background => "#85FF00",
             :color      => "white")

State.create(:name       => "Open",
             :background => "#00CFFD",
             :color      => "white")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
State.create(:name       => "Closed",
             :background => "black",
             :color      => "white")
```

If you try to run `rake db:seed` now, you'll see that this task was aborted:

```
rake aborted!
Validation failed: Email has already been taken

(See full trace by running task with --trace)
```

When a rake task aborts it means an exception has been raised. As the output above suggests, you can see the backtrace by running the same command with --trace: `rake db:seed --trace`. You'll now be given the complete backtrace of your rake task and can determine what broke. The first line of application related backtrace in the output provides you with a useful clue.

```
/home/you/ticketee/db/seeds.rb:1:in `<top (required)>'
```

It's the first line of db/seeds.rb that's causing the problem! This is the line that creates your admin user and it's rightly failing because you already have a user with the email address `admin@ticketee.com`. Let's comment out these first couple of lines as well as the line that creates the "Ticketee Beta" project, because you don't want two "Ticketee Beta" projects. The only line left uncommented in your seeds file should be the line you've just added. When you run `rake db:seed` again, it will run successfully. Let's uncomment these lines that you've just commented out.

With these states now defined inside db/seeds.rb your feature at spec/integration/seeds_spec.rb will pass when you run it using `bin/rspec spec/intgration/seeds_spec.rb`.

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Also, because you've now got your states seeding, you can go back to your server at http://localhost:3000 and create a comment on your ticket with any status because you're trying to figure out why the "Creating comments" feature is failing. After creating your comment you should see that the ticket's state doesn't display as simple text like "New," "Open," or "Closed," but rather as a standard `inspect` output, as shown in Figure 10.7

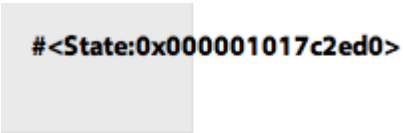

Figure 10.7 Ugly state output

Well, isn't that ugly and not user-friendly? It flows off the end of the ticket box! Thankfully, you can fix this by defining the `to_s` method in your `State` model to call the name method:

```
def to_s
  name
end
```

By default, objects in Ruby have a `to_s` method which will output the ugly version, the inspected version of this object, you saw earlier. By overriding this in the model to call the `name` method you'll get it to display the state's name rather than it's object output.

When you refresh the page in your browser you should see the correct state, as shown in Figure 10.8.

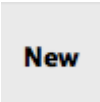

Figure 10.8 The correct state

Great! This should mean that the last scenario in your "Creating comments" feature will pass. Let's run it with `bin/rspec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`spec/integration/creating_comments_spec.rb:34` and find out.

```
1 example, 0 failures
```

Indeed it's passing! This is a good stage to ensure that everything is working by running `rake spec`. Blast, one of the features is failing!

```
Failed examples:

rspec ./spec/integration/creating_comments_spec.rb:28
```

A broken feature often means a broken part of your code, so you should investigate this before continuing. If there are thoughts of "it's only one failing feature," think again. At what point do you draw the line? One? Two? Three failing scenarios? Let's have a zero-tolerance policy on these and fix them when they break, before any problem could potentially appear to users.

10.5.5 Fixing creating comments

The entire reason why we write features before we write code is so that we can catch scenarios like this where something unexpectedly breaks. If we didn't have these scenarios in place, then we wouldn't be made aware of these scenarios until a user of our site stumbled across it. This isn't what we want. We want our users to assume that we're perfect.

You should look into why this feature is failing and fix it right away. This particular scenario is failing with this backtrace:

```
Failure/Error: click_button "Create Comment"
     ActionView::Template::Error:
       undefined method `map' for nil:NilClass
       # ./app/views/comments/_form.html.erb:11
```

Here it claims you're calling `map` on a `nil` object, and that it's on line 11 of app/views/comments/_form.html.erb. The line it's referencing is the following:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= f.select :state_id, @states.map { |s| [s.name, s.id] } %>
```

Alright, the only place where `map` is being called is on the `@states` variable, so it's pretty straightforward that `@states` is the `nil` object. But how did it come to be? Let's review this scenario, as shown in the following listing.

Listing 10.16 spec/integration/creating_comments_spec.rb:28

```
scenario "Creating an invalid comment" do
  click_button "Create Comment"
  page.should have_content("Comment has not been created.")
  page.should have_content("Text can't be blank")
end
```

This scenario tests that you're shown the "Text can't be blank" error when you don't enter any text for your comment. In this scenario, you press the "Create Comment" button, which submits your form, which goes to the `create` action in `CommentsController`. This action looks like the following listing.

Listing 10.17 app/controllers/comments_controller.rb

```
def create
  @comment = @ticket.comments.build(params[:comment])
  @comment.user = current_user
  if @comment.save
    flash[:notice] = "Comment has been created."
    redirect_to [@ticket.project, @ticket]
  else
    flash[:alert] = "Comment has not been created."
    render :template => "tickets/show"
  end
end
```

As you can see from this action, when the comment fails validation (when `@comment.save` returns `false`), then it rerenders the app/views/tickets/show.html.erb template. The problem with this is that, by rerendering this template, it calls the following line in the template:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= render "comments/form" %>
```

Which inevitably leads you right back to app/views/comments/_form.html.erb, the source of the problem. Therefore, you can determine that you need to set up the `@states` variable during the “failed save” part of your action and the best place for this is right after the `else` so that this part ends up looking like the following listing.

Listing 10.18 app/controllers/comments_controller.rb

```
else
  @states = State.all
  flash[:alert] = "Comment has not been created."
  render :template => "tickets/show"
end
```

Now that you’re correctly initializing your `@states` variable, this scenario will pass. Let’s run the whole feature now using `bin/rspec spec/integration/creating_comments_spec.rb`. You'll see that this is now passing:

```
3 examples, 0 failures
```

Awesome! Now let’s try re-running `rake spec`. That should be the last thing you need to fix in order to get everything back to all green. You should see the following output:

```
68 examples, 0 failures, 1 pending
```

The one pending spec that's cramping our style is located in spec/models/state_spec.rb. You can delete this file, as it doesn't contain any useful specs. When you re-run `rake spec` you'll see it's now lovely and green:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
67 examples, 0 failures
```

Excellent, everything's fixed. Let's commit these changes now:

```
git add .
git commit -m "When updating a comment's status,
               also update the ticket's status"
git push
```

It's great and all that you've now got the ticket status updating along with the comment status, but it would be handy to know what the timeline of a status change 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.

10.6 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, as shown in Figure 10.9

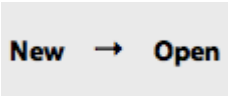

Figure 10.9 State transitions

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.

10.6.1 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, the "State: Open" text 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/integration/creating_comments_spec.rb.

To check for the state change text in your "Changing a ticket's state" scenario

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you'll add these lines to the bottom of it:

```
within("#comments") do
  page.should have_content("State: Open")
end
```

If the ticket was assigned the "New" state, this text would say "State: New ' Open," but because your tickets don't have default states assigned to them the previous state for the first comment will be `nil`. When we run this scenario by using `bin/rspec spec/integration/creating_comments_spec.rb` it will fail.

```
Failure/Error: page.should have_content("State: Open")
     expected there to be content "State: Open" in ...
```

Good, now you've got a way to test this state message that should be appearing when a comment changes the state of the ticket. Now, you'd like to track the state the ticket was at *before* the comment as well as the state of the comment itself. To track this extra attribute, you'll create another field on your `comments` table called `previous_state_id`. Before you save a comment, you'll update this field to be the current state of the ticket. Let's now create a new migration to add the `previous_state_id` field to your `comments` table by running the following command:

```
  rails g migration add_previous_state_id_to_comments \
  previous_state_id:integer
```

Again, Rails is pretty smart here and will use the name of the migration to infer that you want to add a column called `previous_state_id` to a table called `comments`. You only have to tell it what the type of this field is by passing `previous_state_id:integer` to the migration.

If you open up this migration now you'll see that it defines a `change` method which calls the `add_column` method inside it. You can see the entire migration

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

shown in the following listing.

Listing 10.19 Add previous_state_id to comments migration

```
class AddPreviousStateIdToComments < ActiveRecord::Migration
  def change
    add_column :comments, :previous_state_id, :integer
  end
end
```

It's done this way because Rails knows how to rollback this migration easily. It's a simple call to `remove_column` passing in the first two arguments in this method.

You don't need to do anything else to this migration other than run it. Do this now by running `rake db:migrate` and `rake db:test:prepare`. This field will be used for storing the previous state's id so that you can then use it to show a state transition on a comment, as pictured in Figure 10.10

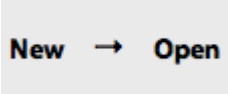

Figure 10.10 A state transition

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.

To use the `previous_state_id` field properly, you're going to need to add another callback to save it.

10.6.2 Another C-c-callback

To set this field before a comment is created you'll use a `before_create` callback on the `Comment` model. A `before_create` callback is triggered—as the name suggests—before a record is created, but *after* the validations have been run. This means that this callback will only be triggered for valid objects that are about to be saved to the database for the first time.

Put this new callback on a line directly above the `after_create` inside the `Comment` model because it makes sense to have all your callbacks grouped together and in the order that they're called in.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
before_create :set_previous_state
```

Call the `set_previous_state` method for this callback, which you'll define at the bottom of our `Comment` model just before the `set_ticket_state` method, like this:

```
def set_previous_state
  self.previous_state = ticket.state
end
```

The `previous_state=` method you call here isn't yet defined. You can define this method by declaring that your `Comment` objects `belongs_to` a `previous_state`, which is a `State` object. Let's put this line with the `belongs_to` in your `Comment` model:

```
belongs_to :previous_state, :class_name => "State"
```

Here you use a new option for `belongs_to`: `class_name`. The field in your `comments` table is called `previous_state_id` and so you call your association `previous_state`. To tell Rails what class this associated record is, you must use the `class_name` option, otherwise Rails will go looking for the `PreviousState` class.

With this `belongs_to` defined, you get the `previous_state=` method for free and so your callback should work alright. There's one way to make sure of this, and that's to attempt to display these transitions between the states in your view so that your feature will potentially pass. You'll now work on displaying these transitions.

10.6.3 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, add the following lines to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

app/views/comments/_comment.html.erb underneath the `h4` tag.

```
<%= render comment.previous_state %> &rarr;
        <%= render comment.state %>
```

This is almost correct, but there's a slight problem. Your callback will set the `previous_state` regardless of what the current state is and in this case you can end up with something like Figure 10.11.

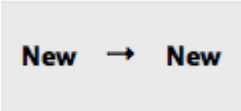

Figure 10.11 State transition from itself to itself

To stop this from happening, you can wrap this code in an `if` statement, like this:

```
<% if comment.previous_state != comment.state %>
  <%= comment.previous_state %> &rarr; <%= comment.state %>
<% end %>
```

Now this text will only show up when the previous state isn't the same as the current state.

You can go one step further and move this code into a helper. Views are more for displaying information than for deciding how it should be output, which should be left to the helpers and controllers. Move this code into the app/helpers/tickets_helper.rb because this partial is displayed from the `TicketsController`'s `show` template. The entire `TicketsHelper` should now look like the following listing.

Listing 10.20 app/helpers/tickets_helper.rb

```
module TicketsHelper
  def state_for(comment)
    content_tag(:div, :class => "states") do
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      if comment.state
        previous_state = comment.previous_state
        if previous_state && comment.state != previous_state
          "#{render previous_state} &rarr; #{render comment.state}"
        else
          render(comment.state)
        end
      end
    end
  end
end
```

In this example, you'll check to see if the comment has an assigned state and then if it has a previous state. If it has a previous state that isn't the assigned state then you show the state transition, otherwise you render the assigned state.

You can now replace the whole `if` statement in app/views/comments/_comment.html.erb with this single line:

```
State: <%= state_for(comment) %>
```

Now you'll check to see if this is working by running your scenario using `bin/rspec spec/integration/creating_comments_spec.rb`. It will now pass:

```
3 examples, 0 failures
```

Excellent! You've now got your application showing the users what state a comment has switched the ticket to. Now's a good time to check that you haven't broken anything. When you run `rake spec` you should see that everything is A-OK.

```
67 examples, 0 failures
```

Now we have state transition showing in your application neatly, which is great to see. Let's commit and push this to GitHub.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git add .
git commit -m "Display a comment's state transition"
git push
```

Currently, your styles aren't distinguishable. Look at Figure 10.12 and gaze upon their ugliness.

Figure 10.12 Ugly, ugly states

You could distinguish them by using the colors you've specified in the attributes. Earlier, you wrapped the state name in a special `div` which will allow you to style these elements, based on the class. For the "New" state, the HTML for the `div` looks like this:

```
<div class="state state_new">
  New
</div>
```

The `state_new` part of this you can use to apply the colors from the record to this element. To do this, you'll put a `style` tag at the top of your application's layout and dynamically define some CSS that will apply the colors.

10.6.4 Styling states

The states in your system can change at any point in time in the future and so you can't have set styles in public/stylesheets/application.css for them. To get around this little problem, put a `style` tag in your app/views/layouts/application.html.erb file, which will contain some ERB code to output the styles for the states. Directly underneath the `stylesheet_link_tag` line, put this code:

```
<style>
  <% for state in @states %>
    .state_<%= state.name.parameterize %> {
      background: <%= state.background %>;
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      color: <%= state.color %>;
    }
  <% end %>
</style>
```

You're going to need to define the `@states` variable in a place that will be accessible in all views of your application. This means you can't define it inside any controller other than `ApplicationController`. Lucky for you, this is like a normal controller and you can use a `before_filter` to load the states. Underneath the class definition for `ApplicationController` you can add this `before_filter`:

```
before_filter :find_states
```

Now you will define the method under the `authorize_admin!` definition:

```
def find_states
  @states = State.all
end
```

With these few lines of code, your states should now be styled. If you visit a ticket page that has comments which have changed the state, you should see a state styled, as shown in Figure 10.13

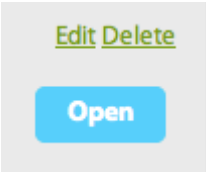

Figure 10.13 States, now with 100% more style

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. Let's add this

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

to the left of the ticket name so that the `li` element becomes:

```
<li>
  <%= render ticket.state if ticket.state %>
  #<%= ticket.id %> - <%= link_to ticket.title, [@project, ticket] %>
</li>
```

Now 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'll be able to select one 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 next to the comment's text on the ticket page there's a state transition shown and (ideally) the comment's text will provide context for that change.

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 this ticket is being looked into/cared for by somebody. When the ticket is fixed, then it should be marked as "Closed," perhaps with some information in its related comment relating where the fix is located.

If you want to add more states than these three default states, you can't at the moment. Tickets can have two different types of "Closed": 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." 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.

10.7 Managing states

Currently your application has only three states: "New," "Open," and "Closed." If you want 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, not the console. They should also be able to rename them 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.

You'll start out by writing a feature to create new states which will involve

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 tickets will then be associated with.

You're not going to look at adding an `edit`, `update` or `destroy` action to this controller because it's been covered previously and it should be left as an exercise to the reader.

10.7.1 Adding additional states

You've three default states from our db/seeds.rb file currently: "New," "Open," and "Closed." If the admin users of your application wish to add more, they can't. Not until you've created the `Admin::StatesController` and the `new` and `create` actions inside it. This will allow those users to create additional states which then can be assigned to a ticket.

You've this lovely `State` model, but no way for admins of the application to add any new records to it. What if they want to add more states? You'll create a new controller called `Admin::StatesController` and put a `new` and `create` action in it so that admins can create more states.

But before you write any real code, you'll write a feature that describes the process of creating a state. You'll put it in a new file called spec/integration/creating_states_spec.rb which is shown in the following listing.

Listing 10.21 spec/integration/creating_states_spec.rb

```
require 'spec_helper'

feature 'Creating states' do
  before do
    sign_in_as!(Factory(:admin_user))
  end

  scenario "Creating a state" do
    click_link "Admin"
    click_link "States"
    click_link "New State"
    fill_in "Name", :with => "Duplicate"
    click_button "Create State"
    page.should have_content("State has been created.")
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Here you sign in as an admin user and go through the motions of creating a new state. When you run this feature through using the command `bin/rspec spec/integration/creating_states_spec.rb`, it will fail because it can't find the "States" link:

```
Failure/Error: click_link "States"
Capybara::ElementNotFound:
  no link with title, id or text 'States' found
```

The "States" link should take you to the `Admin::StatesController`'s `index` action, but it doesn't. This is because this link is is missing from the admin home page, located at app/views/admin/base/index.html.erb. You can add this link now by adding the following line to this file:

```
<%= link_to "States", admin_states_path %>
```

The `admin_states_path` method won't be defined yet and you can fix this by adding another `resources` line inside the admin `namespace` in config/routes.rb like this:

```
namespace :admin do
  ...
  resources :states
end
```

With this line inside the admin namespace, the `admin_states_path` method (and its siblings) will be defined. Let's run the feature again with `bin/rspec spec/integration/creating_states_spec.rb` now to see what you have to do next.

```
Failure/Error: click_link "States"
ActionController::RoutingError:
  uninitialized constant Admin::StatesController
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Ah, that's right! You need to generate your controller. You can do this by running the controller generator:

```
rails g controller admin/states
```

When you run this feature again you'll be told that you're missing the `index` action from this controller:

```
And I follow "States"
  The action 'index' could not be found for Admin::StatesController
```

You'll add this action to the app/controllers/admin/states_controller.rb file now, as well as making this controller inherit from `Admin::BaseController`. After you're done, the whole controller class will appear as shown in the following listing

Listing 10.22 app/controllers/admin/states_controller.rb

```
class Admin::StatesController < Admin::BaseController

  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 our feature will go looking for, and it should also include a list of states so that anyone looking at the page know which states already exist The code to do all this is shown in the following listing.

Listing 10.23 app/views/admin/states/index.html.erb

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= link_to "New State", new_admin_state_path %>

<ul id='states'>
  <% for state in @states %>
    <li><%= state.name %></li>
  <% end %>
</ul>
```

With this view now written, your feature will now whinge about the `new` action when you run `bin/rspec spec/integration/creating_states_spec.rb`:

```
Failure/Error: click_link "New State"
AbstractController::ActionNotFound:
  The action 'new' could not be found for Admin::StatesController
```

Alright then, you should add the `new` action to `Admin::StatesController` if you want to continue any further. It should be defined like the following inside that controller:

```
def new
  @state = State.new
end
```

You'll 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:

```
<h1>New State</h1>
<%= render "form" %>
```

You're using a form partial here again because it's best practice and also just in case you ever wanted to use it for an `edit` action. In a new file for your partial at app/views/admin/states/_form.html.erb you'll put the form that will be used to create new states. This form is pretty simple: it only needs a text field for the name and a submit button to submit the form.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= form_for [:admin, @state] do |f| %>
 <p>
   <%= f.label :name %>
   <%= f.text_field :name %>
 </p>

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

Because the `@state` variable coming from the `new` is a new instance of the `State` model, the `submit` method will display a submit button with the text "Create State," just like your feature needs. Speaking of which, with this form partial done your feature should run a little further. You should check this now by running `bin/rspec spec/integration/creating_states_spec.rb` .

```
Failure/Error: click_button "Create State"
AbstractController::ActionNotFound:
  The action 'create' could not be found for Admin::StatesController
```

Right, so you'll need to create the `create` action too, whichyou'll define inside `Admin::StatesController` as shown in the following listing.

Listing 10.24 app/controllers/admin/states_controller.rb

```
def create
  @state = State.new(params[:state])
  if @state.save
    flash[:notice] = "State has been created."
    redirect_to admin_states_path
  else
    flash[:alert] = "State has not been created."
    render :action => "new"
  end
end
```

With the `create` action defined in your `Admin::StatesController` you'll now be able to run `bin/rspec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`spec/integration/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 to build the other state features upon. You shouldn't have broken anything by these changes but it won't hurt to run `rake spec` to make sure. You should see the following:

```
69 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 re-run `rake spec` there should be this great green output:

```
  68 examples, 0 failures
```

Great! Let's commit this now:

```
git add .
git commit -m "Add Admin::StatesController for managing states"
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.

10.7.2 Defining a default state

A default state for the tickets in your application will provide a sensible way of grouping tickets that are new to the system, making it easier for them to be found. The easiest way to track which state is the default state is to add a boolean column called `default` to your `states` table, which is set to true if the state is the default, false if not.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To get started, you'll write a feature that covers changing of the default status first. At the end of this feature, you'll end up with the `default` field in the `states` table and then you can move onto making the tickets default to this state. Let's create a new feature called spec/integration/managing_states_spec.rb and fill it with the content from the following listing.

Listing 10.25 spec/integration/managing_states_spec.rb

```
require 'spec_helper'

feature "Managing states" do
  before do
    load Rails.root + "db/seeds.rb"
    sign_in_as!(Factory(:admin_user))
  end

  scenario "Marking a state as default" do
    visit "/"
    click_link "Admin"
    click_link "States"
    within state_line_for("New") do <co id="ch10_740_1"/>
      click_link "Make Default"
    end

    page.should have_content("New is now the default state.")
  end
end
```

In this scenario you've got one new line❶, which you'll need to define for this feature to run. This new `state_line_for` method will need to return the CSS selector for the table row that contains the "Make Default" link for the specified state. This method is assisting the Capybara test in its job, and so should go into spec/support/capybara_helpers.rb. Define it using the code from the following listing, placing it inside the `CapybaraHelpers` module:

Listing 10.26 spec/support/capybara_helpers.rb

```
def state_line_for(state)
  state = State.find_by_name!(state)
  "#state_#{state.id}"
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This method simply takes the name of the state, finds the corresponding `State` object for it, and then gets the `id` of that attribute to put into a CSS selector, which would then be passed to `within` to perform the lookup.

Now that this method is defined, let's see what the test says when its run using `bin/rspec spec/integration/managing_states_spec.rb`.

```
Failure/Error: within state_line_for("New") do
Capybara::ElementNotFound:
  Unable to find css "#state_1"
```

The feature is failing because it cannot find the state element on the page. The reason that it can't do this is because the state elements on the page haven't yet been given an `id` attribute! This needs to happen, and the way to make it happen would be to alter the code within app/views/admin/states/index.html.erb to include this `id` attribute. We should also add the "Make Default" link. With that in mind, replace the code in app/views/admin/states/index.html.erb with the code from the following listing:

Listing 10.27 app/views/admin/states/index.html.erb

```
<%= link_to "New State", new_admin_state_path %>

<ul id='states'>
  <% for state in @states %>
    <li id='state_<%= state.id %>'>
      <%= state.name %>
      <%= link_to "Make Default",
            make_default_admin_state_path(state) %>
    </li>
  <% end %>
</ul>
```

In this view now, what you have is the states being displayed along with either a "(Default)" label next to them if they are indeed the state that is the default one. If the state is not the default state, then there's the option there to make it the default with the "Make Default" link instead.

When you run your feature again with `bin/rspec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`spec/integration/managing_states_spec.rb` you'll rightly be told that the `make_default_admin_state_path` method is undefined.

```
Failure/Error: click_link "States"
ActionView::Template::Error:
  undefined method `make_default_admin_state_path' for ...
```

This method should take you to the `make_default` action in the `Admin::StatesController`, much like `edit_admin_state_path` takes you to the `edit` action. You can define this method as a *member route* on your states 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. To define this, you'll change the `resources :states` line inside the admin namespace inside `config/routes.rb` into the following:

```
resources :states do
  member do
    get :make_default
  end
end
```

Inside the `member` block here, you define that each `state` resource has a new action called `make_default` on them that can be accessed through a `GET` request. As stated previously, by defining the route in this fashion you also get the `make_default_admin_state_path` helper which you use in app/views/admin/states/index.html.erb. With this member route now defined, your feature will now complain that it's missing the `make_default` action when you re-run it with `bin/rspec spec/integration/managing_states_spec.rb`:

```
Failure/Error: click_link "Make Default"
AbstractController::ActionNotFound:
  The action 'make_default' could not be
  found for Admin::StatesController
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The `make_default` action will be responsible for making the state you've selected the new default state, as well as 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 10.28 app/controllers/admin/states_controller.rb

```
def make_default
  @state = State.find(params[:id])
  @state.default! <co id="ch10_802_1"/>

  flash[:notice] = "#{@state.name} is now the default state."
  redirect_to admin_states_path
end
```

Rather than putting the logic to change the selected state to the new default inside the controller, you'll place it in the model. To trigger a state to become the new default state, you'll call the `default!`❶ method on it. It's 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 `default!` method can be defined in the `State` model, as shown in the following listing.

Listing 10.29 app/models/state.rb

```
def default!
  current_default_state = State.find_by_default(true)
  <co id="ch10_813_1"/>

  self.default = true
  self.save!

  if current_default_state <co id="ch10_813_2"/>
    current_default_state.default = false
    current_default_state.save!
  end

end
```

The `find_by_default` ❶ method here is a *dynamic finder* method from

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: click_link "Make Default"
NoMethodError:
 undefined method `find_by_default' for ...
```

```
rails g migration add_default_to_states default:boolean
```

```
add_column :states, :default, :boolean, :default => false
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

because you've got this `default` column allowing the whole process of making a state the default to complete.

```
1 example, 0 failures
```

Great to see! When a ticket is created now, the state of that ticket will default to the `State`, which is set to default. You should make "New" the default state in your application now by adding a `default` attribute from where you create it inside `db/seeds.rb` to the following:

```
State.create(:name       => "New",
             :background => "#85FF00",
             :color      => "white",
             :default    => true)
```

When this seeds file is run later on, you'll have a default state for your tickets so that they display properly in your tickets listing.

You should now commit these changes but, before that, you should make sure you haven't caused anything to break. Let's run `rake spec` to find out.

```
69 examples, 2 failures
```

Oops, there's two broken tests. But fortunately, they're broken in the same way:

```
1) Managing states Marking a state as default
  Failure/Error: load Rails.root + "db/seeds.rb"
  ActiveModel::MassAssignmentSecurity::Error:
    Can't mass-assign protected attributes: default
  # ./db/seeds.rb:8 ...
  # ./spec/integration/managing_states_spec.rb:5 ...

2) Seed Data The basics
  Failure/Error: load Rails.root + "db/seeds.rb"
  ActiveModel::MassAssignmentSecurity::Error:
    Can't mass-assign protected attributes: default
  # ./db/seeds.rb:8:in ...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  # ./spec/integration/seeds_spec.rb:5 ...
```

These tests are failing because you're now attempting to assign a protected attribute called `default` on line 8 of db/seeds.rb. This line is the line where you have now set up the "New" state with a new attribute called `default`, which is not declared to be mass-assignable in the `State` model.

To fix this problem, you *could* add the `default` attribute to the list of assignable attributes in the `State` model, but since it's only used in this one place, that's not a good idea. Instead, what would be better is to tell Rails that we don't care about mass-assignment protection in this one case. To do this, we can use the `without_protection` option on the `create` call. Replace the code in db/seeds.rb that creates the "New" state with these lines:

```
State.create({ :name       => "New",
               :background => "#85FF00",
               :color      => "white",
               :default    => true}, :without_protection => true)
```

The `without_protection` option will tell the `create` method to ignore any protection rules regarding the attributes and to just create the object as is. This should be enough to fix your tests, so check the status of them by running `rake spec` now.

```
```

There’s nothing broken, so it’s time to commit:

```
git add .
git commit -m "Admins can now set a default state for tickets"
git push
```

You’re so close to being done with states. So far, you’ve added the functionality for users to change the state through the comment form, to display the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

state transition on a comment, and (just recently) the ability for admins to be able to create new states and toggle which state is the default.

At the moment, any user is able to change the state of a ticket, which isn't a good thing. You'd only like some users to still have the ability to leave a comment but not to change the state and you'll look at creating this feature right now. This is the final feature you'll be implementing for states.

10.8 Locking down states

This feature is going to take a little more than hiding the state select box from the form; you're also going to need to tell the application to ignore the state parameter if the user doesn't have permission to change the state. You'll implement this one piece at a time, beginning with ensuring the state select box is hidden from those who should be unable to change the state.

10.8.1 Hiding a select box

In previous chapters you've seen how we can hide links from certain users by using the CanCan-provided `can?` view helper. You can use this helper to also hide the state field in your comment form from the users without the permission to change the state. Firstly, you'll write a Cucumber scenario to ensure that the state box is always hidden from these users.

You'll add this particular scenario to the bottom of the spec/integration/creating_comments_spec.rb because its operation is based around creating a comment. The scenario to ensure that you don't see this state field is a short and simple one:

```
scenario "A user without permission cannot change the state" do
  click_link ticket.title
  find_element = lambda { find("#comment_state_id") }
  find_element.should raise_error(Capybara::ElementNotFound)
end
```

This scenario contains two simple steps: navigating to the ticket and then attempting to locating the element with the `id` attribute of `comment_state_id`. When the `find` method cannot find an element, the `Capybara::ElementNotFound` exception is raised. With the `should raise_error` assertion here, you ensure that when this exception occurrs, it is captured by the assertion and checked. If the exception is what you say it should

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

be, then the test will pass. The call to the `find` method is wrapped in a lambda so that RSpec will rescue the exception rather than it bubbling up and making the test incorrectly fail when it is raised. When you run this scenario by running `bin/rspec spec/integration/creating_comments_spec.rb:50`, you will see it fail like this:

```
Failure/Error:
  find_element.should raise_error(Capybara::ElementNotFound)
  expected Capybara::ElementNotFound but nothing was raised
```

This test is correctly failing because the element is actually found on the page when it shouldn't be. If the element was *not* found on the page, then the `Capybara::ElementNotFound` exception would be thrown. The message for the test is a little cryptic, so it should be fixed. To do this, change the test to this:

```
scenario "A user without permission cannot change the state" do
  click_link ticket.title
  find_element = lambda { find("#comment_state_id") }
  message = "Expected not to see #comment_state_id, but did."
  find_element.should(raise_error(Capybara::ElementNotFound), message)
  <co id="ch10_813_1"/>
end
```

Inside the scenario on the second line, there's been a new variable added that is the error message that should be shown when the assertion on the final line of the scenario fails. On the final line, we use this variable by passing it as a second argument to `should`. This replaces the cryptic message with the custom message. When you re-run `bin/rspec spec/integration/creating_comments_spec.rb:50` you'll now see this error:

```
 Expected not to see #comment_state_id, but did.
```

That's definitely a step in the right direction. Now it's time to actually make this test pass. To do this, you'll use the `authorized?` method to check that the user

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<% authorized?(:"change states", @project) do %>
  <p>
    <%= f.label :state_id %>
    <%= f.select :state_id, @states.map { |s| [s.name, s.id] },
                            :selected => @ticket.state_id   %>
  </p>
<% end %>
```

```
1 example, 0 failures
```

```
cannot select option, no select box with id, name,
  or label 'State' found
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
define_permission!(user, "change states", project)
```

```
1 scenario (1 passed)
16 steps (16 passed)
```

10.8.2 Bestowing changing state permissions

```
scenario "Changing states for a ticket" do
  check_permission_box "view", project
  check_permission_box "change_states", project
  click_button "Update"
  click_link "Sign out"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  sign_in_as!(user)
  click_link project.name
  click_link ticket.title
  fill_in "Text", :with => "Opening this ticket."
  select "Open", :from => "State"
  click_button "Create Comment"
  page.should have_content("Comment has been created.")
  within("#ticket .state") do
    page.should have_content("Open")
  end
end
```

When you run this scenario with the command `bin/rspec spec/integration/admin/assigning_permissions_spec.rb:72` you'll see that it can't find the "Change States" select box for the project.

```
cannot check field, no checkbox ... 'permissions_1_change_states'
```

This is fine. You haven't added it to the list of permissions inside of app/helpers/admin/permissions_helper.rb yet. You can add this now by adding this key-value pair to the `Hash` object in the `permissions` method's hash:

```
"change states" => "Change States"
```

With this now added to the permissions hash, your scenario will move a little further toward success. When you rerun the scenario with `bin/rspec spec/integration/admin/assigning_permissions_spec.rb:72`, it will complain that it can't find the "Open" state from your select box:

```
Failure/Error: select "Open", :from => "State"
  Capybara::ElementNotFound:
  cannot select option, no option with text 'Open' in select box 'State'
```

Ah, not a problem! This means that the "Open" state isn't yet defined in your test database. You need to create this state in the `before` block for this feature.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You can do this by adding the following line at the bottom of the `before` block:

```
  State.create!(:name => "Open")
```

When you rerun this scenario using `bin/rspec spec/integration/admin/assigning_permissions_spec.rb:73`[4] it will now pass.

Footnote 4 If you get no scenario and no steps running for this feature, are you sure you're running the right line? Check to make sure you're running line 73, not 72, which is now a blank line.

```
1 example, 0 failures
```

That's good! Now admins are able to assign the "Change states" permission and users are able to see and touch the "State" select box on the comment form if they have this permission.

This is a great halfway point before you go diving into the final stage of this particular set of features to run your specs to ensure that nothing is broken. Let's run `rake spec` now and you should see that all your tests are passing:

```
71 examples, 0 failures
```

Yay! Everything is in working order which means you can commit and push these changes to GitHub:

```
git add .
git commit -m "Only users with the 'change states'
               permission can change states"
git push
```

The final piece of your states puzzle is to stop the state parameter from being set in your `CommentsController` if a user passes it through and doesn't have permission to set states. Firstly, you'll investigate how a user can fake this

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

10.8.3 Hacking a form

```
user = User.create!(:email => "test@example.com",
                    :password => "password")
user.confirm!
user.permissions.create({:thing  => Project.first,
                         :action => "view"},
                         :without_protection => true)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

page by going to “File” and then “Save” or “Save Page As,” and save this file in a memorable location. You’re going to be editing this saved file and adding in a state select box of your own.

Open this saved file in a text editor and look for the following lines:

```
<p>
  <label for="comment_text">Text</label><br>
    <textarea cols="40" id="comment_text"
      name="comment[text]" rows="20"></textarea>
</p>
```

These lines display the “Text” label and the associated `textarea` for a new comment. You’re able to add the state field underneath the text field ourselves by adding this code[5] to the page:

Footnote 5 Assuming you know -- or can at least guess -- the IDs of the states.

```
<p>
  <label for="comment_state">State</label><br>
  <select id="comment_state_id" name="comment[state_id]">
    <option value="1" selected="selected">New</option>
    <option value="2">Open</option>
    <option value="3">Closed</option>
  </select>
</p>
```

When you save this page you’ll now be able to choose a state when you open it in a browser. The `action` of the `form` tag on this page goes to http://localhost:3000/tickets/[id]/comments (where `[id]` is the id of the ticket this form will create a comment for) and this route will take you to the `create` action inside `CommentsController`.

Let’s open this saved page in a browser now, fill in the text for the comment with anything, and select a value for the state. When you submit this form, it will create a comment and set the state. You should see your comment showing the state transition, as shown in Figure 10.15.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Figure 10.15 Hacked state transition

Obviously, hiding the state field isn't a foolproof way to protect it. A better way to protect this attribute would be to delete it from the parameters before it gets to the method that creates a new comment.

10.8.4 Ignoring a parameter

If you remove the `state_id` key from the `comment` parameters before they're passed to the `build` method in the `create` action for `CommentsController` then this problem would not happen. You should write a *regression test*. Regression tests are tests that save you from causing regressions.

You'll now open spec/controllers/comments_controller_spec.rb and set up a project, ticket, state, and user for the spec you're about to write by putting the code from the following listing inside the `describe CommentsController` block.

Listing 10.32 spec/controllers/comments_controller_spec.rb

```
let(:user) { Factory(:confirmed_user) }
let(:project) { Project.create!(:name => "Ticketee") }

let(:ticket) do
  ticket = project.tickets.build(:title => "State transitions",
                                 :description => "Can't be hacked.")
  ticket.user = user
  ticket.save
  ticket
end

let(:state) { State.create!(:name => "New") }
```

The state you create will be the one you'll attempt to transition to in your spec, with the ticket's default state being not set, and therefore `nil`. The `user` you set up will be the user you use to sign in and change the state with. You need to set the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`user` attribute separate to the other ticket attributes because it is protected from being mass-assigned. This user has no permissions at the moment and so they won't be able to change the states.

Your spec needs to make sure that a change doesn't take place when a user who doesn't have permission to change the status of a ticket for that ticket's project submits a `state_id` parameter. You'll put this code, shown in the next listing, directly underneath the setup you just wrote.

Listing 10.33 spec/controllers/comments_controller_spec.rb

```
context "a user without permission to set state" do
  before do
    sign_in(:user, user)
  end

  it "cannot transition a state by passing through state_id" do
    post :create, { :comment => { :text => "Hacked!",
                                  :state_id => state.id },
                    :ticket_id => ticket.id }
                    ticket.reload <co id="ch10_1072_1"/>
    ticket.state.should eql(nil)
  end
end
```

This spec uses a `before` to sign in as the user before the example runs. Inside the example you use the `post` method to make a POST request to the `create` action inside `CommentsController` passing in the specified parameters. It's this `state_id` parameter that should be ignored in the action.

After the `post` method you use a new method: `reload` . When you call `reload` on an Active Record object it will fetch it again from the database and update the attributes for it. You use this because the `create` action acts on a different `Ticket` object and doesn't touch the one you've set up for your spec.

The final line here asserts that the `ticket.state` should be `nil`. When you run this spec by running `bin/rspec spec/controllers/comments_controller_spec.rb` this final line will be the one to fail:

```
Failure/Error: ticket.state.should eql(nil)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
 expected: nil
      got: #<State id: 1, name: "New" ...>
```

```
if cannot?(:"change states", @ticket.project)
  params[:comment].delete(:state_id)
end
```

```
1 example, 0 failures
```

```
72 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Great! You'll now commit and push this Github:

```
git add .
git commit -m "Protect state_id from users who do
               not have permission to change it"
git push
```

The `CommentsController` will now reject the `state_id` parameter if the user doesn't have permission to set it, thereby protecting the form from anybody "hacking" it to add a `state_id` field when they shouldn't.

The feature of protecting the `state_id` field from changes was the final piece of the state features puzzle. You've now learned how to stop a user from changing not only a particular record when they don't have permission to, but rather a specific field on a record.

10.9 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 (as shown in Figure 10.16)

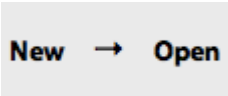

Figure 10.16 Replay: State transitions

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 will add *tags* to your tickets. Tags are words or short phrases that provide categorization for tickets, making them easier for users to manage.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Ind

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

11 Tagging

In Chapter 10, you saw how to give your tickets states ("New," "Open," and "Closed") 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 things such as iterations[1] or similar feature sets. If you didn't have 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.

Footnote 1 For example, by using a process such as Agile, feature sets, or any other method of grouping.

To manage tags, you'll set up a `Tag` model, which will have a `has_and_belongs_to_many` association to the `Ticket` model. You'll set up a *join table* for this association, which is a table that contains foreign key fields for each side of the association. 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. As you move forward in developing this association, note that, for all intents and purposes, `has_and_belongs_to_many` works like a two-way `has_many`.

You'll create two ways to add tags to a ticket. A text field for new tickets beneath the form's description field will allow users to add multiple tags by using a space to separate different tags, as shown in Figure 11.1.

Figure 11.1 The tag box

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Additional tags may also be added on a comment, with a text field similar to the one from the new ticket page providing the tagging mechanism. When a ticket is created, you'll show these tags underneath the description, as shown in Figure 11.2

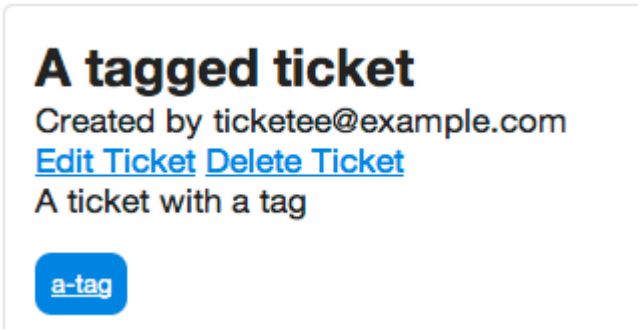

Figure 11.2 A tag for a ticket

When a user clicks on a tag, they'll be taken to a page where they can see all tickets with that particular tag. Alternatively, if the user clicks the little "x" next to the tag, that tag will be removed from the ticket. The actions of adding and removing a tag are both actions you'll add to your permission checking.

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 your first feature, adding tags to a new ticket.

11.1 Creating tags

Tags in this application will be extremely 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-existent `Tag` model.

11.1.1 Creating tags feature

You're going to add a text field beneath the description field on the new ticket page for this feature, like you saw earlier in Figure 11.1.

Figure 11.3 The tag box

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The words you enter into this field will become the tags for this ticket, and you should see them on the ticket page. At the bottom of spec/integration/creating_tickets_spec.rb, you'll add a scenario that creates a new ticket with tags, as shown in the following listing:

Listing 11.1 spec/integration/creating_tickets_spec.rb

```
scenario "Creating a ticket with tags" do
  fill_in "Title", :with => "Non-standards compliance"
  fill_in "Description", :with => "My pages are ugly!"
  fill_in "Tags", :with => "browser visual"
  click_button "Create Ticket"
  page.should have_content("Ticket has been created.")
  within("#ticket #tags") do
    page.should have_content("browser")
    page.should have_content("visual")
  end
end
```

When you run the "Creating a ticket with tags" scenario using `bin/rspec spec/integration/creating_tickets_spec.rb:58` it will fail, declaring that it can't find the "Tags" field. Good! It's not there yet.

```
Failure/Error: fill_in "Tag names", :with => "browser visual"
  Capybara::ElementNotFound:
  cannot fill in, no text field, text area or
  password field with id, name, or label 'Tag names' found
```

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_field`s that you've used previously, this one will not be tied to a database field.

To define this field, you'll put the following code underneath the `p` tag for the description in app/views/tickets/_form.html.erb:

```
<p>
  <%= f.label :tag_names %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  <%= f.text_field :tag_names %>
</p>
```

When you re-run this scenario again with `bin/rspec spec/integration/creating_tickets_spec.rb:58` it no longer complains about the missing "Tag names" field, telling you instead that it can't find the `tag_names` method on `Ticket` objects: ticket:

```
Failure/Error: click_link "New Ticket"
  ActionView::Template::Error:
  undefined method `tag_names' for #<Ticket:0x007ff1211eef28>
```

As said previously, the `tag_names` attribute isn't going to be tied to a database field, but instead will be a *virtual attribute*. A virtual attribute works just like a real attribute, except that it's not persisted to the database along with the normal attributes. Instead, it's constructed from other data within the model. To define this virtual attribute in your `Ticket` model, put this line underneath the `attr_accessible` call in app/models/ticket.rb:

```
attr_accessor :tag_names
```

You're grouping the `attr_accessor` and `attr_accessible` calls in the model because they're both defining actions on attributes of this model, and it's a good code organisation practice to group similar things together. The `attr_accessor` call defines virtual attributes in classes for Ruby, and so we can use this feature also in our Rails applications. The method will define a setter and a getter method for this attribute, performing the equivalent of this code:

```
def tag_names
  @tag_names
end

def tag_names=(names)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  @tag_names = names
end
```

The `attr_accessor` now will define the `tag_names` method that is sought after by the scenario. To make sure of this and to see what next to do, re-run the scenario with `bin/rspec spec/integration/creating_tickets_spec.rb:58`:

```
Failure/Error: click_button "Create Ticket"
ActiveModel::MassAssignmentSecurity::Error:
  Can't mass-assign protected attributes: tag_names
```

Ah, of course! You'll need to add `tag_names` to the list of accessible attributes inside the `Ticket` model also, because it's being passed in along with the `title` and `description` attributes as well. To do that, just turn this line inside app/models/ticket.rb:

```
attr_accessible :description, :title, :assets_attributes
```

Into this:

```
attr_accessible :description, :title, :assets_attributes, :tag_names
```

That'll be enough to get the scenario happy regarding that little problem, so if you re-run `bin/rspec spec/integration/creating_tickets_spec.rb:58`, you'll be told what needs fixing again:

```
Failure/Error: within("#ticket #tags") do
  Capybara::ElementNotFound:
  Unable to find css "#ticket #tags"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You'll now need to define this `#tags` element inside the `#ticket` 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.

11.1.2 Showing tags

You can add this new element, with its `id` attribute set to `tags`, to app/views/tickets/show.html.erb by adding this simple line underneath where you render the ticket's description:

```
<div id='tags'><%= render @ticket.tags %></div>
```

This creates the `#ticket #tags` element that your feature is looking for, and 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 steps do you take next? If you run your scenario again, you'll see that it cannot find the `tags` method for a `Ticket` object:

```
undefined method `tags' for #<Ticket:0x0..
```

This method is the `tags` method, which you'll be defining 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.

11.1.3 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:

```
has_and_belongs_to_many :tags
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This association will rely on a join table that doesn't yet exist called `tags_tickets`. The name 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.

When you re-run your scenario you're told that there's no constant called `Tag` yet:

```
uninitialized constant Ticket::Tag (ActionView::Template::Error)
```

In other words, there is no `Tag` model yet. You should define this now if you want to go any further.

11.1.4 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
```

The `timestamps` option passed here determines whether or not the model's migration is generated with timestamps. Because you've passed the value of `false` to this option, there will be no timestamps added.

Before you run this migration, however, you'll need to add the join table called `tags_tickets` to your database. The join table has two fields: one called `ticket_id` and the other `tag_id`. The table name is the pluralized names of the two models it is joining, sorted in alphabetical order. This table will have no primary key, as you're never going to look for individual records from this table and only need it to join the `tags` and `tickets` tables.

To define the `tags_tickets` table, put this code at the bottom of the `change` method of your db/migrate/[timestamp]_create_tags.rb migration:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
create_table :tags_tickets, :id => false do |t|
  t.integer :tag_id, :ticket_id
end
```

The `:id => false` option passed to `create_table` here tells Active Record to create the table without the `id` field, as the join table only cares about the link between tickets and tags, and therefore does not need a unique identifier.

Next, run the migration on your development database by running `rake db:migrate`, and on your test database by running `rake db:test:prepare`. This will create the `tags` and `tags_tickets` tables.

When you run this scenario again with `bin/rspec spec/integration/creating_tickets_spec.rb:58`, it is now satisfied that the `tags` method is defined and has now moved on to complaining that it can't find the "browser" tag within the `#ticket #tags` element on the ticket's page:

```
And I should see "browser" within "#ticket #tags"
  Failed assertion, no message given. (MiniTest::Assertion)
```

This failure is because you're not doing anything to associate the text from the "Tags" field to the ticket you've created. You need to parse the content from this field into new `Tag` objects and then associate them with the ticket you are creating, which you'll do right now.

11.1.5 Displaying a ticket's tags

You're now going to take the name for the tags that are passed in to the `tag_names` attribute for `Ticket` objects and turn them into objects of the `Tag` class. You're going to be doing this with an Active Record callback, just like you saw back in Chapter 10.

To make this happen, go into your `Ticket` model and put these lines inside the class definition, at the bottom:

```
before_create :associate_tags

private
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def associate_tags
  if tag_names
    tag_names.split(" ") each do |name|
      self.tags << Tag.find_or_create_by_name(name)
    end
  end
end
```

Before a new `Ticket` object is saved to the database, this new `associate_tags` method will go through all the `tag_names` associated with this object and associate new `Tag` objects with the ticket. It does this by using the dynamic finder `find_or_create_by_name`. The whole thing is wrapped in an `if`, because if `tag_names` is `nil`, you don't want it to attempt to parse the non-existant tags.

The `associate_tags` method you have just written will create the tags that you are 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 `bin/rspec spec/integration/creating_tickets_spec.rb:58`, you'll see this render is now failing with an error:

```
Missing partial tags/tag ...
```

This error is happening now because `@ticket.tags` actually contains some tickets, and the `render` call is attempting to render them. Just like back in Chapter 10 when we used this line:

```
<%= render @ticket.state %>
```

Rails will render a partial for the given objects based off the class name of the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

object. In the case of `@ticket.state` that class was `State`, and therefore the app/views/states/_state partial was used. When you iterate over a collection of objects, just as you're doing in the case of `@ticket.tags`, Rails will pick the first object from that collection and then render a partial for each of the objects based off the class of that first element. Therefore 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:

```
<span class='tag'><%= tag.name %></span>
```

By wrapping the tag name in a span with the `class` of tag, it will be styled as defined in your stylesheet (app/assets/stylesheets/application.css.scss). With this partial defined, this will stop the "Missing template" error from happening. When you run your scenario again with `bin/rspec spec/integration/creating_tickets_spec.rb:58` it should now pass:

```
1 example, 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 in figure 11.2, and is shown here again.

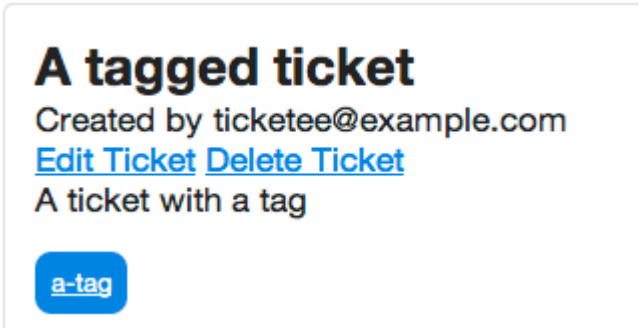

Figure 11.4 Look ma, a tag!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You'll now commit this change, but before you do you'll ensure that you haven't broken anything by running `rake cucumber:ok spec`.

```
74 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 since there's nothing else in that file, it's safe to delete it. So go ahead and do that now. After you're done, a re-run of `rake spec` will produce this lovely green output:

```
73 examples, 0 failures
```

Let's commit this change.

```
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 come about that would require another tag 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 add the tag when they comment.

11.2 Adding more tags

The tags for a ticket can change throughout the ticket's life; new tags can be added and old ones can be deleted. Let's look at how you can add more tags to a ticket after it's been created through the comments form. Underneath the comment form on a ticket's page, add the same tags field that you previously used to add tags to your ticket on the new ticket page. One thing you have to keep in mind here is that if someone enters a tag that's already been entered, you don't want it to show up.

You've got two scenarios to implement then: the first is a vanilla addition of tags to a ticket through a comment, and the second is a scenario ensuring that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

duplicate tags do not appear. Let's implement this function one scenario at a time. When you're done, you'll end up with this pretty picture (Figure 11.5.

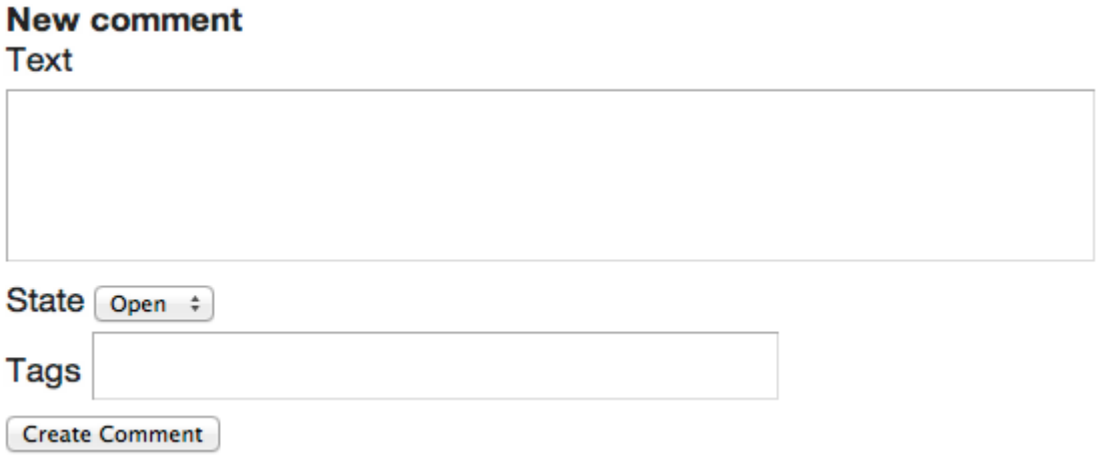

Figure 11.5 Comment form with tags

11.2.1 Adding tags through a comment

To test that users can add tags when they're creating a comment, you'll add a new scenario to the features/creating_comments.feature feature that looks like the following listing:

Listing 11.2 spec/integration/creating_comments_spec.rb

```
scenario "Adding a tag to a ticket" do
  click_link ticket.title
  within("#ticket #tags") do
    page.should_not have_content("bug")
  end

  fill_in "Text", :with => "Adding the bug tag"
  fill_in "Tags", :with => "bug"
  click_button "Create Comment"

  page.should have_content("Comment has been created.")
  within("#ticket #tags") do
    page.should have_content("bug")
  end
end
```

First, you ensure that you don't see this tag within `#ticket #tags`, to ensure you don't have a false positive. Next, you fill in the text for the comment so it's valid, add the word "tag" to the "Tags" field, and press the "Create Comment"

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

button. Finally, you ensure that the comment has been created and that the "bug" tag you entered into the comment form now appears in `#ticket #tags`.

When you run this scenario using `bin/rspec spec/integration/creating_comments_spec.rb:55`, it will fail because there is no "Tags" field on the ticket's page yet:

```
cannot fill in, no text field, text area or password
  field with id, name, or label 'Tags' found
```

You can fix this by taking these lines from app/views/tickets/_form.html.erb and moving them into a new partial at app/views/tags/_form.html.erb:

```
  <%= f.label :tag_names, "Tags" %>
  <%= f.text_field :tag_names %>
</p>
```

Replace the code you removed from app/views/tickets/_form.html.erb with this line:

```
<%= render "tags/form", :f => f %>
```

This new line will render your new app/views/tags/_form.html.erb partial, passing in the form building object, `f`, so that it's also available in that partial. In order to make the failing step in your scenario now pass, you'll re-use this same line now inside the `authorized?` block inside app/views/comments/_form.html.erb underneath the code you use to render the state select box.

Listing 11.3 Adding tags to app/views/comments/_form.html.erb

```
<% authorized?(:"change states", @project) do %>
  <p>
    <%= f.label :state_id %>
    <%= f.select :state_id, @states.map { |s| [s.name, s.id] },
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
                    :selected => @ticket.state_id   %>
  </p>
<% end %>

<%= render "tags/form", :f => f %>
```

When rendering the `tags/form` partial here, you'll be passing in the form builder object for a `Comment` object, not a `Ticket`. Not that the partial will mind; all it needs really is some kind of object that has a `tag_names` method on it and it's quite content.

When you re-run the scenario with `bin/rspec spec/integration/creating_comments_spec.rb:55`, you'll see this message:

```
Failure/Error: click_link ticket.title
ActionView::Template::Error:
  undefined method `tag_names' for #<Comment:0x007faf7dab8e78>
```

When defining the tag fields inside the form for a `Ticket` you came across this same problem. The problem back then was because there was no attribute -- real or virtual -- defined for `Ticket` objects. The problem you're facing is almost exactly the same, but this time it's for the `Comment` model. So open up app/models/comment.rb and make a call to `attr_accessor` right underneath the `attr_accessible` call in this model to define a virtual attribute for `tag_names`. You'll also need to add `tag_names` to the `attr_accessible` call in the model, so that you end up with these two lines:

```
attr_accessible :text, :state_id, :tag_names
attr_accessor :tag_names
```

This new `attr_accessor` call in your `Comment` model will define the `tag_names` method that the scenario is looking for and the addition to `attr_accessible` will make the `tag_names` To see what to do next, re-run the scenario. You will see this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: page.should have_content("bug")
 expected there to be content "bug" in ""
```

This scenario is not seeing the word "bug" within the content for the ticket's tags (which is empty), and so the scenario fails. This is because the code to associate a tag with a ticket isn't in the `Comment` model, as it is inside the `Ticket` model. To associate the tags from a comment with the relevant ticket when the comment is saved, you will use an `after_create` callback inside your `Comment` model:

```
after_create :associate_tags_with_ticket
```

You want to use an `after_create` here so that the tags aren't associated prematurely to a ticket, which they would be if you were using a `before_create`. For this callback to work, you will need to have the `associate_tags_with_ticket` method defined too. Define this method underneath the `set_previous_state` method inside the `Comment` model, like this:

```
def associate_tags_with_ticket
  if tag_names
    tags = tag_names.split(" ").map do |name| <co id="ch11_v2_21_1"/>
      Tag.find_or_create_by_name(name)
    end
    self.ticket.tags += tags <co id="ch11_v2_21_2"/>
    self.ticket.save
  end
end
```

This method is slightly different to the one found within the `Ticket` model. In this method, rather than iterating over each of the tag names and then adding a tag to the ticket for each tag name, you're using the `map` method instead . What this will do is iterate through each tag, find or create a `Tag` object for it, and then when it's done will return an array of `Tag` objects. This array is then added to the ticket's tags , and then the ticket is saved.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This should mean that now a comment's tags are associated with the ticket. Find out by running `bin/rspec spec/integration/creating_comments_spec.rb:55`.

```
1 example, 0 failures
```

Boom, that's passing! Good stuff. Now for the cleanup. Make sure you haven't broken anything else by running `rake spec` :

```
74 examples, 0 failures
```

With all the specs passing, it's commit time! In this section, you've created a way for your users to add more tags to a ticket when they add a comment, which allows your users to easily organize tickets into relevant groups after the ticket's creation. Let's commit this change now:

```
git add .
git commit -m "Users can add tags when adding a comment"
git push
```

With the ability to add tags when creating a ticket or a comment now available , you need to restrict this power to users with permission to manage tags. You don't want all users to create tags willy-nilly, as it's likely you would end up with an overabundance of tags[2]. Too many tags makes it hard to identify which tags are useful and which are not. People with permission to tag things will know that with great power, comes great responsibility.

Footnote 2 Such as the tags on the Rails Lighthouse account, on the bottom right-hand side of this page: https://rails.lighthouseapp.com/projects/8994-ruby-on-rails/overview

11.3 Tag restriction

Using the permissions system you built in chapter 8, you can easily add another type of permission: one for tagging. If a user has this permission, they will be able to add and (later on) remove tags.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

11.3.1 Testing tag restriction

When a user without permission attempts to submit a ticket or comment, the application should not tag the ticket with the tags they have specified. Add this restriction to the `CommentsController`, but first you'll write a controller spec to cover this behavior. Put the code from the following listing at the bottom of the block for `describe CommentsController` inside spec/controllers/comments_controller_spec.rb:

Listing 11.4 spec/controllers/comments_controller_spec.rb

```
context "a user without permission to tag a ticket" do
  before do
    sign_in(:user, user)
  end

  it "cannot tag a ticket when creating a comment" do
    post :create, { :comment => { <co id="ch11_327_1"/>
                      :text => "Tag!",
                      :tag_names => "one two"
                    },
                    :ticket_id => ticket.id
                  }
    ticket.reload <co id="ch11_327_2"/>
    ticket.tags.should be_empty
  end
end
```

In this test, you passing through the parameters to create a comment ❶, and then asserting that there are no tags created ❷. You need to reload the ticket here because the object that's been loaded for this spec by the setup won't be the same object that's modified in the controller spec. If the test is working as it should, then running `bin/rspec spec/controllers/comments_controller_spec.rb` will produce this error:

```
Failure/Error: ticket.tags.should be_empty
 expected empty? to return true, got false
```

Good! A failing test is a good start to a new feature. To make this test pass, you

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

should use the `can?` method in `CommentsController` to check the user's permission. Tto remove the `tag_names` parameter from the comment's parameters if the user is unable to tag, just like you did with the state parameter if they weren't able to change the state, you'll put these lines at the top of the `create` action, underneath the first `if` statement in it:

```
if cannot?(:tag, @ticket.project)
  params[:comment].delete(:tag_names)
end
```

The `create` action now has a lot of logic at the top of the method that is sanitizing the parameters. It's getting quite crowded in there! To make it easier to follow, move the two `if` statements checking for permissions out into a new private method for this class, like this:

```
def sanitize_parameters!
  if cannot?(:"change states", @ticket.project)
    params[:comment].delete(:state_id)
  end

  if cannot?(:tag, @ticket.project)
    params[:comment].delete(:tag_names)
  end
end
```

Then rather than having two `if` statements at the top of the `create` action, you can now call the `sanitize_parameters!` method, so that the `create` action is a little neater.

```
sanitize_parameters!
@comment = @ticket.comments.build(params[:comment])
@comment.user = current_user
...
```

If you were going to add an `update` action to this controller later on, this action could also use the `sanitize_parameters!` method.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you re-run the spec with `bin/rspec spec/integration/comments_controller_spec.rb`, it will pass because the user in the spec does not have permission to tag a project.

```
2 examples, 0 failures
```

Good! You have something in place to block users from tagging tickets when they create a comment. Now you're only missing the blocking code for tagging a ticket when it is being created. You can create a spec test for this too, this time in spec/controllers/tickets_controller_spec.rb. Underneath the "cannot delete a ticket without permission" example, add this example:

```
it "can create tickets, but not tag them" do
  Permission.create(:user => user,
                    :thing => project,
                    :action => "create tickets")
  post :create, :ticket => { :title => "New ticket!",
                             :description => "Brand spankin' new",
                             :tag_names => "these are tags"
                           },
               :project_id => project.id
  Ticket.last.tags.should be_empty
end
```

You can run this spec by running `bin/rspec spec/controllers/tickets_controller_spec.rb:62`, and you'll see that it fails:

```
Failure/Error: Ticket.last.tags.should be_empty
  expected empty? to return true, got false
```

Because there is no restriction on tagging a ticket through the `create` action, there are tags for the ticket that was just created, and so your example fails. For your `TicketsController`'s `create` action, you can do exactly what you did in the `CommentsController`'s create action and sanitize the parameters before

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

they're passed to where the object is created. To do this, make the beginning of the `create` action inside `TicketsController` look like this:

```
if cannot?(:tag, @project)
  params[:ticket].delete(:tag_names)
end
```

When you re-run your spec it will now pass:

```
1 example, 0 failures
```

Great, now you're protecting both the ways a ticket can be tagged. Because of this new restriction, the two scenarios which you created earlier to test this behavior will now be broken.

11.3.2 Tags are allowed, for some

When you run `rake spec` you'll see them listed as the only two failures:

```
Failing Scenarios:
rspec ./spec/integration/creating_comments_spec.rb:55
rspec ./spec/integration/creating_tickets_spec.rb:57
```

To fix these two failing scenarios, you'll use a new step, which you'll first put in the "Creating comments" feature. Underneath this line in the `before` for this feature:

```
define_permission!(user, "view", project)
```

Put this line:

```
define_permission!(user, "tag", project)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you re-run this scenario using `bin/rspec spec/integration/creating_comments_spec.rb` it will pass:

```
5 examples, 0 failures
```

One scenario down, one to go! The next one is the spec/integration/creating_tickets_spec.rb scenario. At the top of the feature, you can put the same line you used in the "Creating comments" feature, right under the "view" permission.

```
define_permission!(user, "tag", project)
```

This scenario too will now pass:

```
1 scenario (1 passed)
16 steps (16 passed)
```

Great! Only certain users can now tag tickets. Let's make sure that everything is still running at 100% by running `rake spec` again.

```
76 examples, 0 failures
```

In this section, you have restricted the ability to add tags to a ticket—whether through the new ticket or comment forms—to only users who have the permission to "tag." You've done this to restrict the "flow" of tags. Generally speaking, the people with the ability to tag should know only to create useful tags, so that the usefulness of the tags is not diluted. In the next section, you'll use this same permission to determine what users are able to remove a tag from a ticket.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

11.4 Deleting a tag

Removing a tag from a ticket is a helpful feature, because a tag may become irrelevant over time. Say that you've tagged a ticket as "v0.1" for your project, but that milestone is complete and the feature isn't yet and therefore needs to be moved to "v0.2." Without this feature, there will be no way to delete the old tag. Then what? Was this ticket for "v0.1" or "v0.2"? Who knows? With the ability to delete a tag, you have some assurance that people will clean up tags if they're able to.

To let users delete a tag, add a little "x" to the left of each of your tags, as shown in Figure 11.6

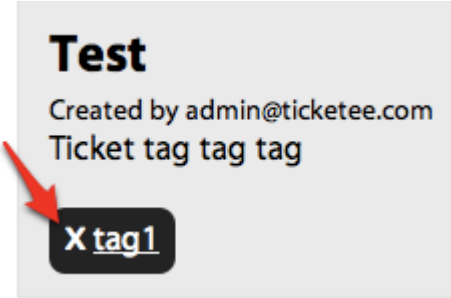

Figure 11.6 X marks the spot

When this little "x" is clicked, the tag will disappear through the magic of JavaScript. Rather than making a whole request out to the action for deleting a tag and then redirecting 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 delete tag action.

11.4.1 Testing tag deletion

To click this link using Capybara, you'll give the link around the "x" an id so you can easily locate it in your feature, which you'll now write. Let's create a new file at spec/integration/deleting_tags_spec.rb and put the code from the following in there.

Listing 11.5 spec/integration/deleting_tags_specs.rb

```
require 'spec_helper'

feature "Deleting tags" do
  let!(:user) { Factory(:confirmed_user) }
  let!(:project) { Factory(:project) }
  let!(:ticket) do
    Factory(:ticket,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
            :project => project,
            :tag_names => "this-tag-must-die",
            :user => user)
  end

  before do
    sign_in_as!(user)
    define_permission!(user, "view", project)
    define_permission!(user, "tag", project)
    visit '/'
    click_link project.name
    click_link ticket.title
  end

  scenario "Deleting a tag", :js => true do
    click_link "delete-this-tag-must-die"
    within("#ticket #tags") do
      page.should_not have_content("this-tag-must-die")
    end
  end
end
```

In this scenario, it's important to note that you're passing through the tags field as a field in the "created a ticket" step, just like the other fields. The "tags" field isn't in the tickets table. You'll get to that in a second.

In this feature, you create a new user and sign in as them. Then you create a new project called "Ticketee" and give the user the ability to view and tag the project. You create a ticket by the user and tag it with a tag called "this_tag_must_die". Finally, you navigate to the page of the ticket you've created.

In the scenario, you follow the "delete-this-tag-must-die" link, which will be the id on the link to delete this tag. When this link has been followed, you shouldn't see "this_tag_must_die", meaning that the action to remove the tag from the ticket has worked its magic.

When you run this feature using `bin/rspec spec/integration/deleting_tickets_spec.rb` you'll get this error:

```
Failure/Error: click_link "delete-this-tag-must-die"
Capybara::ElementNotFound:
  no link with title, id or text 'delete-this-tag-must-die' found
```

Alright, time to implement this bad boy.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

11.4.2 Adding a link to delete the tag

You need a link with the id of "delete-this-tag-must-die", which is the word "delete", followed by a hyphen and then the `parameterize`'d version of the tag's name. You last used the `parameterize` method back in Chapter 10 to provide a class name for states. This delete tag link needs to trigger an asynchronous request to an action that would remove a tag from a ticket. The perfect name for an action like this, if you were to put it in the `TicketsController`, would be "remove_tag". But because it's acting on a tag, a better place for this action would be inside a new controller called `TagsController`.

Before you go and define this action, let's define the link that your scenario is looking for first. This link goes into the tag partial at app/views/tags/_tag.html.erb inside the `span` tag:

```
<% if can?(:tag, @ticket.project) || current_user.admin? %>
  <%= link_to "x",
      remove_ticket_tag_path(@ticket, tag), <co id="ch11_409_1"/>
      :method => :delete,
      :remote => true <co id="ch11_409_2"/>
      :id => "delete-#{tag.name.parameterize}" %> <co id="ch11_409_3"/>
<% end %>
<%= tag.name %>
```

Here, you check that a user can "tag" in the ticket's project. If they can't tag, then you won't show the "x" to remove the tag. This is to prevent everyone from removing tags as they feel like it. Remember? With great power comes great responsibility.

You use the `:remote` option for the `link_to`, to indicate to Rails that you want this link to be an asynchronous request. This is similar to the "Add another file" button you provided in chapter 9, except this time you don't need to call out to any javascript to determine anything, you only need to make a request to a specific URL.

For the `:url` option here, you pass through the `@ticket` object to `remove_ticket_tag_path` so that your action knows what ticket to delete the tag from. Remember: your primary concern right now is disassociating a tag and a ticket, not completely deleting the tag.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Because this is a destructive action, you use the `:delete` method. You've used this previously for calling `destroy` actions, but the `:delete` method is not exclusive to the `destroy` action, and so you can use it here as well.

The final option, `:id` , lets you define the `id` for this link. You set that to be "delete", followed by a hyphen and then the name of your tag `parameterize`'d. For the tag in your scenario, this is the `id` that you'll use to click this link. Capybara supports following links by their internal text, the `name` attribute, or the `id` attribute.

When you run your feature with `bin/rspec spec/integration/deleting_tags_spec.rb`, you'll see that it reports the same error message at the bottom:

```
When I follow "delete-this-tag-must-die"
 no link with title, id or text 'delete-this-tag-must-die'
```

Ah! A quick eye would have spotted an error when the browser launched by WebDriver tried going to this page; it looks like Figure 11.7

Figure 11.7 Internal Server Error

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:

```
resources :tickets do
  resources :comments
  resources :tags do
    member do
      delete :remove
    end
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

By nesting the tags resource inside the ticket's resource, you are given routing helpers such as `ticket_tag_path`. With the `member` block inside the `resources :tags`, you can define further actions that this nested resource responds to. You'll define that you should accept a `DELETE` request to a route to a `remove` action inside the `TagsController`, which you should now create.

Before you add this action to the `TagsController`, you must first generate this controller by using:

```
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:

```
def remove
  @ticket = Ticket.find(params[:ticket_id])
  if can?(:tag, @ticket.project) || current_user.admin?
    @tag = Tag.find(params[:id])
    @ticket.tags -= [@tag] <co id="ch11_409_1"/>
    @ticket.save
    render :nothing => true
  end
end
```

In this action, you find the ticket based on the id passed through as `params[:ticket]`, and then you do something new. On the left side of `-=` you have `@ticket.tags`. On the right, an array containing `@tag`. This combination will remove the tag from the ticket, but will not delete the tag from the database.

On the second-to-last line of this action, you save the ticket minus one tag. On the final line you tell it to return nothing, which will return a `200 OK` status to your browser, signaling that everything went according to plan.

When you re-run your scenario with `bin/rspec spec/integration/deleting_tags_spec.rb` it will now successfully

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

click the link, but the tag is still there:

```
Failure/Error: page.should_not have_content("this-tag-must-die")
  expected content "this-tag-must-die" not to return anything
```

Your tag is unassociated from the ticket but not removed from the page, and 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.

11.4.3 Actually removing a tag

You're removing a tag's association from a ticket, but you're not yet showing people that it has happened on the page. If a request is made asynchronously, the format for that request will be `js`, rather than the standard `html`. For views, you've always used the `html.erb` extension, because HTML is all you've been serving. As of now, this changes. You're going to be rendering a `js.erb` template, which will contain JavaScript code to remove your element. Let's create the view for the `remove` action in a file called app/views/tags/remove.js.erb, and fill it with this content:

```
$('#delete-<%= @tag.name.parameterize %>').parent().remove();
```

This code will be run when the request to the `remove` action is complete. It uses the jQuery library's `$` function to locate an element with the `id` attribute of "delete-this-tag-must-die" and then calls the `parent()`[3] function to find the element wrapping the delete link, and then the `remove()`[4] on it, which will remove the tag from the page.

Footnote 3 http://api.jquery.com/parent

Footnote 4 http://api.jquery.com/remove/

Because you're now rendering a template for the `remove` action, you will need to remove the line from the `remove` action that tells the action to render nothing:

```
render :nothing => true
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

If you don't remove this line, then the action will not render the template and then the tag will not be removed. Without the line, the template will be rendered and the JavaScript inside it will be processed during the request.

When you run your feature using `bin/rspec spec/integration/deleting_tags_spec.rb`, you'll see that it now passes:

```
1 example, 0 failures
```

Awesome! With this feature done, users with permission to tag on a project will now be able to remove tags too. Before you commit this feature, let's run `rake spec` to make sure everything is ok.

```
78 examples, 0 failures, 1 pending
```

That's awesome too! There's one pending spec inside spec/models/tags_helper_spec.rb. You can delete this file now, and when re-running `rake spec` you'll see this now:

```
77 examples, 0 failures
```

That's awesome too! Commit and push this:

```
git add .
git commit -m "Add remove tag functionality"
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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

way for the 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.

11.5 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 in association with the "iteration_1" tag that had the state of "open". This helps users scope down the list of tickets that appear on a project page to be able to better focus on them.

There's a gem developed specifically for this purpose called Searcher [5] which you can use. This provides you with a `search` method on specific classes, which accepts a query like the one above and returns the records that match it.

Footnote 5 This gem is good for a lo-fi solution, but shouldn't be used in a high search-volume environment. For that, look into full text search support for your favorite database system.

11.5.1 Testing search

As usual, you should (and will) test that searching for tickets with a given tag works, which you can do by writing a new feature called spec/integration/searching_spec.rb and filling it with the content from Listing 11.6.

Listing 11.6 spec/integration/searching_spec.rb

```
require 'spec_helper'

feature "Searching" do
  let!(:user) { Factory(:confirmed_user) }
  let!(:project) { Factory(:project) }
  let!(:ticket_1) do
    Factory(:ticket,
            :title => "Create projects",
            :project => project,
            :user => user,
            :tag_names => "iteration_1")
  end

  let!(:ticket_2) do
    Factory(:ticket,
            :title => "Create users",
            :project => project,
            :user => user,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
            :tag_names => "iteration_2")
  end

  before do
    define_permission!(user, "view", project)
    define_permission!(user, "tag", project)

    sign_in_as!(user)
    visit '/'
    click_link project.name
  end

  scenario "Finding by tag" do
    fill_in "Search", :with => "tag:iteration_1"
    click_button "Search"
    within("#tickets") do
      page.should have_content("Create projects")
      page.should_not have_content("Create users")
      end
  end
end
```

In the `Background` 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 `bin/rspec spec/integration/searching_spec.rb` and it'll complain because there's no "Search" field on the page:

```
Failure/Error: fill_in "Search", :with => "tag:iteration_1"
Capybara::ElementNotFound:
  cannot fill in, no text field, text area or
  password field with id, name, or label 'Search' found
```

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. You'll add the field above the `ul` element for the tickets list, inside app/views/projects/show.html.erb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= form_tag search_project_tickets_path(@project),
             :method => :get do %>
  <%= label_tag "search" %>
  <%= text_field_tag "search", params[:search] %>
  <%= submit_tag "Search" %>
<% end %>
```

You've only used `form_tag` once, back in Chapter 8. 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_for` 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 use `submit_tag` for a submit button for this form.

The `search_project_tickets_path` method is undefined at the moment, which you'll see when you run `bin/rspec spec/integration/searching_spec.rb`:

```
undefined local variable or method `search_project_tickets_path' ...
```

Notice the pluralized "tickets" in this method. To define non-standard 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. This time, however, you want to act on a collection of a resource. This means that you use the `collection` method in config/routes.rb instead. To define this method, change these lines in config/routes.rb:

```
resources :projects do
  resources :tickets
end
```

Into these:

```
resources :projects do
  resources :tickets do
    collection do
      get :search
    end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  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` you have set up. When you run your feature again by using `bin/rspec spec/integration/searching_spec.rb`, you'll see that it's reporting that the `search` action is missing:

```
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]`, which is what you can use the Searcher gem for.

11.5.2 Searching by state with Searcher

The Searcher gem provides the functionality of parsing the labels in a query such as "tag:iteration_1" and determines how to go about finding the records that match the query. Rather than working like Google, where you could put in "iteration_1" and it would "know," you have to tell it what "iteration_1" means by prefixing it with "tag:". You use this query with the `search` method provided by Searcher on a configured model, and it will return only the records that match it:

```
Ticket.search("tag:iteration_1")
```

You'll use this method in the `search` action for `TicketsController` in a bit.

The first port of call to begin to use the Searcher gem is to add it to your Gemfile underneath `gem 'paperclip'`:

```
gem 'searcher', :git => "git://github.com/radar/searcher"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You're using the `:git` option to install the gem here, which is different to every other gem that you've installed so far. This option will clone the gem's git repository to your local machine and the load the gem from that repository, rather than the traditional method of installing it from RubyGems. This allows gem authors to update their gems code on GitHub and provide an alternative way to distribute their gems that isn't RubyGems.

To install this gem, you run `bundle install`. Now for the configuration. Searcher is configured by a `searcher` call in a model class, just as associations are setup by using `has_many` and friends. In app/models/ticket.rb directly above[6] the first `belongs_to`, put this code:

Footnote 6 Code from gems or plugins should go above any code for your models, as it may modify the behavior of the code that follows it.

```
searcher do
  label :tag, :from => :tags, :field => :name
end
```

The `:from` option tells Searcher what association this label should be searched upon, while the `:field` option tells it what field to perform a lookup on.

The `label` method is evaluated internally to Searcher and will result in a `by_tag` method being defined on your `Ticket` model, which will be used by the `search` method if you pass in a query such as "tag:iteration_1". This method will perform an SQL join on your tags table, returning only the tickets that are related to a tag with the given name.

With this configuration now in your model, you'll be able to define the `search` action directly underneath the `destroy` action in `TicketsController` to use the `search` method on `Ticket`:

```
def search
  @tickets = @project.tickets.search(params[:search])
end
```

Assign all the tickets retrieved with the `search` method to the `@tickets`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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:

```
<% @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:

```
  <% @tickets.each do |ticket| %>
```

With this change, you break the `ProjectsController`'s `show` action, because the `@tickets` variable is not defined there. You can see the error you would get when you run `ber spec/integration/viewing_tickets_spec.rb`:

```
You have a nil object when you didn't expect it!
You might have expected an instance of Array.
The error occurred while evaluating nil.each
```

To fix this error, you'll set up the `@tickets` variable inside the `show` action of `ProjectsController`, which you should place directly under the definition for the `index` action:

```
def show
  @tickets = @project.tickets
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you re-run `bin/rspec spec/integration/viewing_projects_spec.rb:`, you'll see that it now passes once again:

```
1 example, 0 failures
```

Great! With the insurance that you're not going to break anything now, you can render the app/views/projects/show.html.erb template in the `search` action of `TicketsController` by putting this line at the bottom of that action:

```
render "projects/show"
```

By rendering this template, you'll show a similar page to `ProjectsController#show`, but this time it will only have the tickets for the given tag. When you run your "Searching" feature using `bin/rspec spec/integration/searching_spec.rb` you'll see that it all passes now:

```
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 given tag. You prevented one breaking change by catching it as it was happening, but how about the rest of the test suite? Let's find out by running `rake spec`. You should see this result:

```
78 examples, 0 failures
```

Great! Let's commit this change now:

```
git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git commit -m "Add label-based searching for tags using Searcher"
git push
```

Now that you have tag-based searching, why don't you 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 the search term of "state:open tag:iteration_1". It's easy to implement.

11.5.3 Searching by state

Implementing searching for a state is incredibly easy now that you have the Searcher plugin setup 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. Let's change the code at the top of the feature in spec/integration/searching_spec.rb so that states are now specified for each of the tickets, replacing the two `let` blocks for the tickets with the code from the following listing:

Listing 11.7 spec/integration/searching_spec.rb

```
let!(:ticket_1) do
  state = State.create(:name => "Open")
  Factory(:ticket,
          :title => "Create projects",
          :project => project,
          :user => user,
          :tag_names => "iteration_1",
          :state => state)
end

let!(:ticket_2) do
  state = State.create(:name => "Closed")
  Factory(:ticket,
          :title => "Create users",
          :project => project,
          :user => user,
          :tag_names => "iteration_2",
          :state => state)
end
```

When the two tickets in this feature are created, there will be two states associated with these tickets also. 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

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

listing.

Listing 11.8 Finding by state scenario, for the 'Searching' feature

```
scenario "Finding by state" do
  fill_in "Search", :with => "state:Open"
  click_button "Search"
  within("#tickets") do
    page.should have_content("Create projects")
    page.should_not have_content("Create users")
  end
end
```

This should show any ticket with the "Open" state, and hide all other tickets. When you run this feature with `bin/rspec spec/integration/searching_spec.rb` you'll see that this is not the case. It can still see the "Create users" ticket.

```
Failure/Error: page.should_not have_content("Create users")
  expected content "Create users" not to return anything
```

When a user performs a search on only an undefined label (such as your "state" label), Searcher will return all the records for that table because it will completely ignore queries that it won't recognise. This is the behavior you are seeing right now, so it means that you need to define your `state` label in your model. Let's open app/models/ticket.rb and add this line to your `searcher` block:

```
label :state, :from => :state, :field => :name
```

With this label defined, your newest scenario will now pass when you re-run `bin/rspec spec/integration/searching_spec.rb`:

```
2 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You only had to add states to the tickets that were being created and tell searcher to search by states, and now this feature passes.

That's it for the searching feature! In it, you've added the ability for users to find tickets by a given tag and/or state. It should be mentioned that these queries can be chained, so a user may 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 per usual, commit your changes because you're done with this feature. But also per usual, you'll check to make sure that everything is A-OK by running `rake spec`:

```
79 examples, 0 failures
```

Brilliant, let's commit:

```
git add .
git commit -m "Users may now 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.

11.5.4 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 are shown all tickets for that specific tag. To test this functionality, you can add another scenario to the bottom of spec/integration/searching_spec.rb to test that when a user clicks on a ticket's tag, they are only shown tickets for that tag. The new scenario looks pretty much identical to this:

```
scenario "Clicking a tag goes to search results" do
  click_link "Create projects"
  click_link "iteration_1"
  within("#tickets") do
    page.should have_content("Create projects")
    page.should_not have_content("Create users")
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

When you run this last scenario using `bin/rspec spec/integration/searching_spec.rb`, you're told that it cannot find the "iteration_1" link on the page:

```
no link with title, id or text 'iteration_1' found
```

This scenario is successfully navigating to a ticket and then attempting to click a link with the name of the tag, only to not find the tag's name. Therefore, 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. Let's open app/views/tags/_tag.html.erb and change this simple little line:

```
<%= tag.name %>
```

Into this:

```
<%= link_to tag.name,
              search_project_tickets_path(@ticket.project,
             :search => "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. So you'll see that when you run `bin/rspec spec/integration/searching_spec.rb`, this new scenario will now pass:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
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. Let's make sure you didn't break anything with this small change by running `rake spec`. You should see this output:

```
80 examples, 0 failures
```

Great, nothing broke! Let's 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 before you have a good tagging system for your application.

11.6 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, and therefore 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[7].

Footnote 7 Like foursquare does

You first wrote the functionality for tagging a ticket when it was created, and then 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 the `parent()` and `remove()` functions from jQuery with the help of a `js` format template file, which is used specifically for JavaScript requests. This file allowed you to execute

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

JavaScript code when an AJAX request completes, and you used it to remove the tag 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 would implement some sort of help page that would demonstrate to the users how to use the search box, but that's another exercise for the reader.

Your final feature, based off the previous feature, allowed users to click a tag name and view all the tickets for that tag, and also showed how you can limit the scope of a resource without using nested resources.

In chapter 12, we'll look at how you can send emails to your users using Action Mailer. You'll use these emails to notify new users of new tickets in their project, state transitions, and new comments.

Index Terms

attr_accessor
create_table, id option
link_to, :remote option
link_to, :url option
migration generator, timestamps option
Routing helpers, additional parameters

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

12 Sending Email

In the previous chapter, you implemented tagging for your application, which allows users to easily categorize and search for 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. You then verify that the address is valid by sending the user a confirmation email. With a user's validated email address, you're able to send them updates for important events in the system, such as a ticket being updated.

Back in chapter 6, you changed a setting for the authentication engine Devise that caused Devise to send a confirmation email to a new user when they signed up. To test this setting, you used a gem called `email_spec`, which only tested that the emails were delivered in a test environment, and not in the real world. This is how Action Mailer (the Rails component responsible for email) acts[1] during a test environment.

Footnote 1 It defaults to not truly sending out the emails, but rather keeping track of them in a variable that you can access by using `ActionMailer::Base.deliveries`, or by using the methods found in `email_spec`

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 whenever that user creates a ticket. Every time this ticket is updated by another user, the creator of the ticket should receive an email. This is helpful, as it allows users to keep up-to-date with the tickets that they have created. The second feature will allow users to add or remove themselves from the watching 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 the ticket, what that comment was,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

and any state change that took place. This email message will additionally contain a link to the ticket and a link to unsubscribe from further notifications regarding the ticket. If a user posts a comment to a ticket and they're not watching it, then they will 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 if they wish by following the unsubscribe link in the email. Email is a tried-and-true solution to receiving notifications of events such as this.

Once that's all said and done, you'll work on sending emails through an actual server—Gmail—which will test that your application is able to send out emails into the real world and that you're doing everything you can to let your users receive them. Gmail is great for low-volume sending[2], but if you needed something with a larger capacity, other services such as SendGrid[3] or Mailchimp[4] are acceptable alternatives. While you don't look at how to use large-volume services in this chapter, it's always great to be aware of alternatives, should you ever need to scale up. To check for the emails on a Gmail account, you'll be using the (unofficial)[5] gmail gem.

Footnote 2 Gmail has a daily send limit of 200 emails

Footnote 3 http://sendgrid.com

Footnote 4 http://mailchimp.com

Footnote 5 As in, not sponsored by Google

After spending most of the chapter looking at how to *send* emails, you'll take a look at how to *receive* them using the Gmail gem and Action Mailer. When a user receives an email notifying them that a comment has been posted to a ticket, they will be able to send a reply that you can read using both the Gmail gem and Action Mailer. You'll also be able to create a new comment from their reply's text. Nifty stuff.

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. Let's dive into creating the feature and code for this functionality now.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

12.1 Sending ticket notifications

The next feature of your application will provide users with the ability to watch a ticket. You'll build off this functionality to notify users by email that a ticket has been updated any time somebody posts a comment to it. This email will contain the name of the user who updated the ticket, the comment text, a URL to the ticket, and finally a link to unsubscribe from all future ticket updates.

To test all this, you'll use the email_spec gem, which you first used back in chapter 6. 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 these steps in the feature that you'll be writing right now.

12.1.1 Automatically watching a ticket

This feature will initially test that a user automatically watches a ticket when they create it. Whenever someone else updates this ticket, the user who created it (and later, anybody else watching the ticket) will receive an email notification. You'll put this new feature in spec/integration/ticket_notifications_spec.rb and fill it with the content from Listing 12.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.1 spec/integration/ticket_notifications_spec.rb

```
require "spec_helper"
feature "Ticket Notifications" do
  let!(:alice) { Factory(:user, :email => "alice@example.com") }
  let!(:bob) { Factory(:user, :email => "bob@example.com") }
  let!(:project) { Factory(:project) }
  let!(:ticket) do
    Factory(:ticket,
            :project => project,
            :user => alice)
  end

  before do
    ActionMailer::Base.deliveries.clear <co id="ch12_28_1"/>

    define_permission!(alice, "view", project)
    define_permission!(bob, "view", project)

    sign_in_as!(bob)
    visit '/'
  end

  scenario "Ticket owner receives notifications about comments" do
    click_link project.name
    click_link ticket.name
    fill_in "comment_text", :with => "Is it out yet?"
    click_button  "Create Comment"

    email = find_email!(alice.email) <co id="ch12_28_2"/>
    subject = "[ticketee] #{project.name} - #{ticket.title}"
    email.subject.should include(subject)
    click_first_link_in_email(email) <co id="ch12_28_3"/>

    within("#ticket h2") do
      page.should have_content(ticket.title)
    end
  end
end
```

In this feature, you set up two users: one called Alice and one called Bob. At the top of the `before` for this feature, you need to clear all of Action Mailer's deliveries ❶, as it will otherwise contain the confirmation emails for the user. When the feature signs in as bob and leaves a comment on the ticket, then Alice should receive an email ❷.

The `find_email!` method here is from the `email_spec` gem, and wil open the last email for the specified email address or will raise an exception if it couldn't

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

find one. The next couple of lines in the scenario will check that email to see if it contains the correct subject and for the ticket notification email. The final trick in the scenario is to click the first link in the email using the `click_first_link_in_email` method ❸, and then validate that the link goes to a page that has the ticket title inside a `h2` element.

When you run this feature using `bin/rspec spec/integration/ticket_notifications_spec.rb`, you'll see that Alice is not yet receiving an email:

```
Failure/Error: email.should_not(be_nil ...
  Couldn't open email for alice@example.com
```

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!

TIP **You're not really sending emails**

These emails aren't actually sent to these addresses in the real world, but captured by Action Mailer and stored in `ActionMailer::Base.deliveries`. You then access these emails using the helpers provided by email_spec. There's a setting inside config/environments/test.rb that goes like this:

```
config.action_mailer.delivery_method = :test
```

By default, this setting is set to `:smtp`, which means that Action Mailer will attempt to connect to an SMTP server that is running on localhost. You don't have one of these set up yet, nor will you. Later on, you'll look at how you can actually send out "real world" emails from your application using a Gmail account.

The setting in config/environments/test.rb will tell Action Mailer to store all "sent" emails internally in `ActionMailer::Base.deliveries`.

To make "alice" receive an email, you're going to use what's known as an *observer*.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

12.1.2 Using observers

An observer is a class that sits outside the model, watching it for specific actions such as a save to the database. If new instances of the model are created, then the `before_create` and `after_create` methods in the observer will be called. Observers are handy if you have complex logic for your callbacks, or for sending out email. Hey, isn't that what you want to do? Indeed it is!

In this instance, your observer will be called `CommentObserver`. It's named like that because it will observe the `Comment` model. Observers will watch a model for specific changes and allow you to implement callback-like methods in them to order your application to do something when an action takes place in the model. While you could use a callback in a model, abstracting out code such as this to an observer is much better because it can lead to reduced code clutter in the model.

Let's now create a new folder at app/observers so that you can also reduce clutter in the app/models folder too. All the files inside the app directory are added to the load path, so they will be `require`'able by your application. Inside the app/observers folder you'll create a new file called comment_observer.rb that will hold the code for the observant observer. In this file you'll put this:

```
class CommentObserver < ActiveRecord::Observer
  def after_create(comment)
   (comment.ticket.watchers - [comment.user]).each do |user|
      Notifier.comment_updated(comment, user).deliver
    end
  end
end
```

This defines the observer that watches the `Comment` model and defines a method that will be called after a new `Comment` is saved to the database, more commonly known as `after_create` callback.

At the top of the `after_create` method, you get the list of watchers for a ticket and remove the user who has just made the comment from that list, as they shouldn't receive an email for a comment they just created!

The `Notifier` referenced inside the `after_create` is something you'll create in a little while. Consider it similar to an Active Record object, but for handling emails instead. The `comment_updated` method will build an email for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

each of the users watching this ticket and `deliver` will send it out.

There's a little bit of configuration you must do before this observer is used, however. You must open `config/application.rb` and put this line inside the `Ticketee::Application` class definition:

```
config.active_record.observers = :comment_observer
```

By calling this method, you are telling Rails to load the `CommentObserver` class, which it will find without your help, as Rails will infer the name of the observer from the symbol passed in. When you run `bin/rspec spec/integration/ticket_notifications_spec.rb` you're told this:

```
Failure/Error: click_button  "Create Comment"
NoMethodError:
  undefined method `watchers' for #<Ticket:0x007fcc16d9cf80>
```

In this `after_create` method in your observer, you're calling the `watchers` method to get at the watchers for this ticket. It's failing because you haven't defined this association yet, so let's go ahead and do that now.

12.1.3 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, so that in your feature alice@ticketee.com receives the email triggered by bob@ticketee.com's comment.

Here you must do two things: firstly, define the `watchers` association, and secondly, add the ticket owner to the `watchers` list when the ticket is created.

You'll use another `has_and_belongs_to_many` association to define the `watchers` collection, this time in your`Ticket` model. To define it, you'll put this code inside the `Ticket` model, along with the other `has_and_belongs_to_many` for tags:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
has_and_belongs_to_many :watchers, :join_table => "ticket_watchers",
                                   :class_name => "User"
```

Here 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, then the table name would be inferred by Rails to be `ticket_users`, 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 imply that you wanted the `Watcher` class instead, which doesn't exist.

You can create a migration that can be used to create this table by using this command:

```
rails g migration create_ticket_watchers_table
```

Unfortunately, the `migration` won't read your minds in this instance, so you'll need to open it and change it to resemble Listing 12.2

Listing 12.2 db/migrate/[timestamp]_create_ticket_watchers_table.rb

```
class CreateTicketWatchersTable < ActiveRecord::Migration
  def change
    create_table :ticket_watchers, :id => false do |t|
      t.integer :user_id, :ticket_id
    end
  end
end
```

Remember: you need to specify the `id` option here so that your join table doesn't have a primary key.

Let's save and then run this file using `rake db:migrate`, and let's not forget to run `rake db:test:prepare` either. When you run `bin/rspec spec/integration/ticket_notifications_spec.rb` you'll see that the email still isn't being sent:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: email.should_not(be_nil, ...)
  Couldn't open email for alice@example.com
```

Now that you have your`watchers` method defined, you need to add the user who creates a ticket to the list of watchers for that ticket so that the observer will know who to notify once a comment has been posted. You can do this by using an `after_create` callback on your`Ticket` model like this:

```
after_create :creator_watches_me
```

To define the `creator_watches_me` method, you'll put the following code at the bottom of the `Ticket` class definition:

```
private
  def creator_watches_me
    if user
      self.watchers << user unless self.watchers.include?(user)
    end
  end
```

This method will now add the user of the ticket the list of watchers for this ticket whenever the `after_create` callbacks are triggered. This means that each `Ticket` object will now have a list of users who are watching the ticket, and then will be able to act on those eventually.

Now that you have the user who created the ticket watching it, your `CommentObserver` will have something to act on. Let's see what happens when you run `bin/rspec spec/integration/ticket_notifications_spec.rb` now:

```
And I press "Create Comment"
  uninitialized constant CommentObserver::Notifier (NameError)
```

This time, your feature is failing because it can't find the constant `Notifier`,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

which is actually going to be the class that you use to send out the notifications of new activity to your users. To create this class, you'll use Action Mailer.

12.1.4 Introducing Action Mailer

You need to define the `Notifier` mailer to send out ticket update notifications using your fresh-out-of-the-oven `CommentObserver`'s `after_create` method. You can do this 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 notifier
```

When running this command, you'll see this output:

```
create  app/mailers/notifier.rb
invoke  erb
create    app/views/notifier
invoke  rspec
create    spec/mailers/notifier_spec.rb
```

The first thing the command generates is the `Notifier` class itself, defining it in a new file at app/mailers/notifier.rb. This is done to keep the models and mailers separate. In previous versions of Rails, mailers used to live in the app/models directory, which led to clutter. By separating mailers out into their own folder, the codebase becomes easier to manage. Inside this class,you'll define (as methods) your different notifications that you'll send out, beginning with the comment notification. You'll get to that in just a minute.

The second thing that is generated is the `app/views/notifier` directory, which is used to store all the templates for your emails. The methods in the `Notifier` class will correspond to each of the files in this directory.

The final thing that is generated is the spec/mailers/notifier_spec.rb, which you won't use because you've got your feature testing this notifier anyway.

In app/mailers/notifier.rb you'll see this code:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
class Notifier < ActionMailer::Base
  default from: "from@example.com"
end
```

`ActionMailer::Base` defines helpful methods such as the `default` one, which you can use to send out your emails. [6] The `default` method here configures default options for this mailer and will set the "from" address on all emails to be the one specified. Let's change this now to be "ticketee@gmail.com".

Footnote 6 Action Mailer had a revamp with Rails 3, switching to be based off the new `mail` gem rather than the old`tmail` gem. `mail`'s syntax is much nicer, and won't crash when it parses a spam email, unlike `tmail`

Now that you have the `Notifier` class defined, what happens when you run your feature? Let's run it using `bin/rspec spec/integration/ticket_notifications_spec.rb` and find out:

```
undefined method `comment_updated' for Notifier:Class (NoMethodError)
  ./app/observers/comment_observer.rb:3:in `after_create'
```

In this class, you need to define the `comment_updated` method, which will build an email to send out when a comment is updated. This method needs to get the email address for all the watchers for `comment`'s ticket and send an email to each of them. To do this, you can define the method like this:

```
def comment_updated(comment, user)
  @comment = comment
  @user = user
  @ticket = comment.ticket
  @project = @ticket.project
  subject = "[ticketee] #{@project.name} - #{@ticket.title}"
  mail(:to => user.email, :subject => subject)
end
```

Even though you're defining this as an instance method (the error complains about a *class* method), the `comment_updated` method is truly the method that is used by Action Mailer to set up your email. This is a little bit of magic performed by Action Mailer for your benefit. [7]

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 7 By calling the method on the class, it's caught by `method_missing`, which then initializes a new instance of this class and then eventually ends up calling your`comment_update` method.

When this method is called, it will attempt to render a plain-text template for the email, which should be found at app/views/notifier/comment_updated.text.erb. You'll define this template after you've got the method working. You define a `@comment` instance variable as the first line of your method so that the object in `comment` will be available to your template.

You use the `mail` method to generate a new email, passing it a hash containing `to` and `subject` keys, which define where the email goes to as well as the subject for the email.

When you run `bin/rspec spec/integration/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 is not there, which brings up this cryptic error:

```
Failure/Error: click_first_link_in_email(email)
URI::InvalidURIError:
  bad URI(is not URI?):
```

It's not seeing the link because you have not set up any content for this email yet. The methods defined within an Action Mailer class need to have corresponding templates to them that define the content of the email, much like actions in controllers have (sometimes) had templates for them.[8] Let's define a template for the `comment_updated` mailer method now.

Footnote 8 As an example, the create, update and destroy actions for your controllers do not have corresponding templates. This is not necessarily saying that they should never have templates. These actions sometimes do have templates too.

12.1.5 An Action Mailer template

Templates for Action Mailer classes go in app/views because they serve an identical purpose as the controller views: they display a final, dynamic result to the users. Once you have this template in place, the plain-text email a user receives will look like Figure 12.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

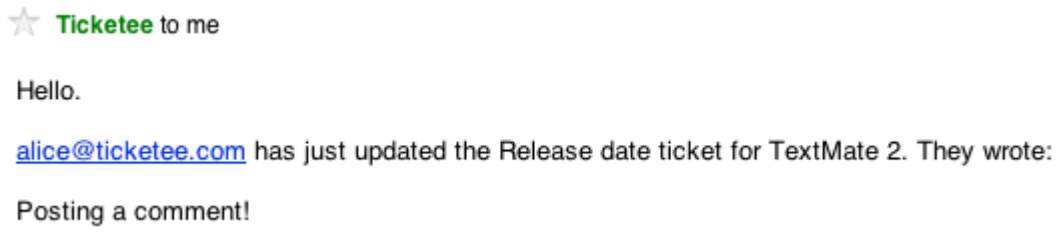

Figure 12.1 Your first email

As shown in the above figure, you'll need to mention who updated the ticket, what they updated it with, and provide a link to the ticket. Let's define a text template for your `comment_updated` method at app/views/notifier/comment_updated.text.erb, as shown in Listing 12.3.

Listing 12.3 app/views/notifier/comment_updated.text.erb

```
<%= project_ticket_url(@ticket.project, @ticket) %>
```

Wait, hold on! text.erb? Yes! This is the template for the plain-text version of this 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.

The template is the final part for your feature, yay! When you run `bin/rspec spec/integration/ticket_notifications_spec.rb` you'll see that it's now all passing:

```
1 example, 0 failures
```

You've done quite a lot to get this little simple feature to pass.

In the beginning you created an *observer* called `CommentObserver`, which watches the `Comment` model for any specific changes. You defined an `after_create` method on this, which took the `comment` object that was being updated and then called `Notifier.comment_updated`, passing along the `comment` object.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`Notifier` is an Action Mailer class that is responsible for sending out emails to the users of your application, and in this file you defined the `comment_updated` method called in your `CommentObserver` and set the `recipients` up to use the `comment` object's related ticket's watchers.

To define the `watchers` method, you used a `has_and_belongs_to_many` join table again. Your first experience using these was back in chapter 11, when you linked the `Ticket` and `Tag` models by setting one up on both of them. Back then, you used the `tags_tickets` table link the two. This is the default naming schema of a `has_and_belongs_to_many` join table in Rails. In the case of your ticket watchers, however, your method was called `watchers`, and so would look for a class called `Watcher` to determine where it should find your watchers. This was incorrect, so you told your association that your join table should be `ticket_watchers` and that the related model was`User`, not `Watcher`. You used the `:join_table` and `:class_name` methods for this.

Finally, you defined the template for the `comment_updated` email at app/views/notifier/comment_updated.text.erb and including the link that you click to complete the final step of your scenario.

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 `rake spec`:

```
82 examples, 0 failures, 1 pending
```

Great to see everything still passing! The one pending spec is located in spec/mailers/notifier_spec.rb, but rather than deleting the file, keep it and just simply delete the pending spec in it. We're going to be using that file in the next section. If you delete just the spec and re-run `rake spec` you'll see this:

```
81 examples, 0 failures
```

You've added email ticket notifications to your application, so you should now

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 the plain text emails out a bit more, making them have 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.

12.1.6 Testing with mailer specs

Now you're going to get down into the nitty-gritty of exactly how your mailer works. You're going to do this with *mailer specs*. Mailer specs are generally contained within the files that are generated along with the mailer, and test intimate details about the mail that is sent out by those mailers, such as body content. In this section, you're going to be learning how to write a mailer spec by writing one that checks that the email contains some content.

The test that you'll be writing now is to make sure that when a user receives the email and it contains a phrase like "[user] has just updated the [ticket title] for [project name]" and the content for the comment.[9]. To test it, 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.

Footnote 9 You may already be familiar with these types of emails from services such as Facebook.

To test that, write the content from Listing 12.4 into spec/mailers/notifier_spec.rb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.4 spec/mailers/notifier_spec.rb

```
require 'spec_helper'

describe Notifier do
  context "comment_updated" do
  let!(:project) { Factory(:project) } <co id="ch12_v2_12_1"/>
    let!(:ticket_owner) { Factory(:user) }
    let!(:ticket) { Factory(:ticket, :project => project,
                                     :user => ticket_owner) }
    <co id="ch12_v2_12_2"/>
    let!(:commenter) { Factory(:user) }
    let(:comment) do
      Comment.new({ <co id="ch12_v2_12_3"/>
        :ticket => ticket,
        :user => commenter,
        :text => "Test comment"
        }, :without_protection => true) <co id="ch12_v2_12_4"/>
    end

    let(:email) do
       Notifier.comment_updated(comment, ticket_owner)
    end

    it "sends out an email notification about a new comment" do
      email.to.should include(ticket_owner.email)
      title = "#{ticket.title} for #{project.name} has been updated."
      email.body.should include(title)
      email.body.should include("#{comment.user.email} wrote:")
      email.body.should include(comment.text)
    end
  end
end
```

At the beginning of this test, a whole bunch of things are setup. First, the test needs a project ❶ and a ticket. That part's easy. The ticket needs to have a user associated with it ❷ so that there's someone to be notified when the comment notification goes out. Next, there needs to be a comment ❸ so that the mailer can act on something. The comment needs to have some text so that it can be validated that it shows up in the email that is sent out.

When creating the comment, you pass it a `ticket` and `user` attribute. These attributes are typically not mass-assignable, because they're not listed within the `attr_accessible` call in the `Comment` model. To get around this, you pass a second argument to the `new` method which is a hash containing just the key `without_protection`, which references the value `true`. This will bypass the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

mass-assignment protection for this `Comment` creation and allow you to assign these attributes.

Inside the test itself, you create a new mail message by calling the `comment_updated` method ❹ and passing it the `comment` and `ticket_owner` objects. The ticket owner is passed here because that is 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 `bin/rspec spec/integration/notifier_spec.rb` you'll see that the email body doesn't contain that specialized message:

```
Failure/Error: email.body.should include(title)
  expected http://localhost:3000/projects/1/tickets/1
   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 have not yet put the special message inside the email; all it contains is just a link. To put that special message in the email and to make it look a whole lot nicer, replace the link inside app/views/notifier/comment_updated.text.erb with this:

```
Hello!

<%= @ticket.title %> for <%= @project.name %> has been updated.

<%= @comment.user.email %> wrote:

<%= @comment.text %>

You can view this ticket online by going to:
<%= project_ticket_url(@project, @ticket) %>
```

When you re-run `bin/rspec spec/mailers/notifier_spec.rb`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you'll see that this spec is now passing because the email now contains the text that you are looking for:

```
1 example, 0 failures
```

Now that you've spruced up the text template for the email, users will receive more relevant information about the comment notification, rather than just simply a link.

In this section, you've learned how to generate a mailer and create a mailer method to it, and now you're going to move into how you can let people subscribe to receive 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.

12.2 Subscribing to updates

You'll provide other users with two ways to stay informed of ticket updates. The first will be very similar to the automatic subscription of a user when they create 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 will be a "watch" button on the ticket page, which will display either "Watch this ticket" or "Stop watching this ticket," depending on if the user is watching the ticket or not, as shown in Figure 12.2

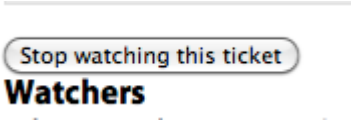

Figure 12.2 The watch button

You'll first look at implementing the automatic subscription when a user posts a comment to a ticket.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

12.2.1 Testing comment subscription

You'll now implement a feature to make users automatically watch a ticket when they create a comment on it. This is useful because your users will want to keep up-to-date with tickets that they have commented on. Later on, you'll implement a way for these users to opt-out.

To automatically subscribe a user to a ticket of a new comment, use an `after_create`, just as you did in the `Ticket` model for only the author of that ticket. But first, you need to ensure that this works!

You'll add another scenario to the "Ticket notifications" feature, but first let's consider the current flow. Here, a couple of diagrams help explain this process. First, let's look at Figure 12.3.

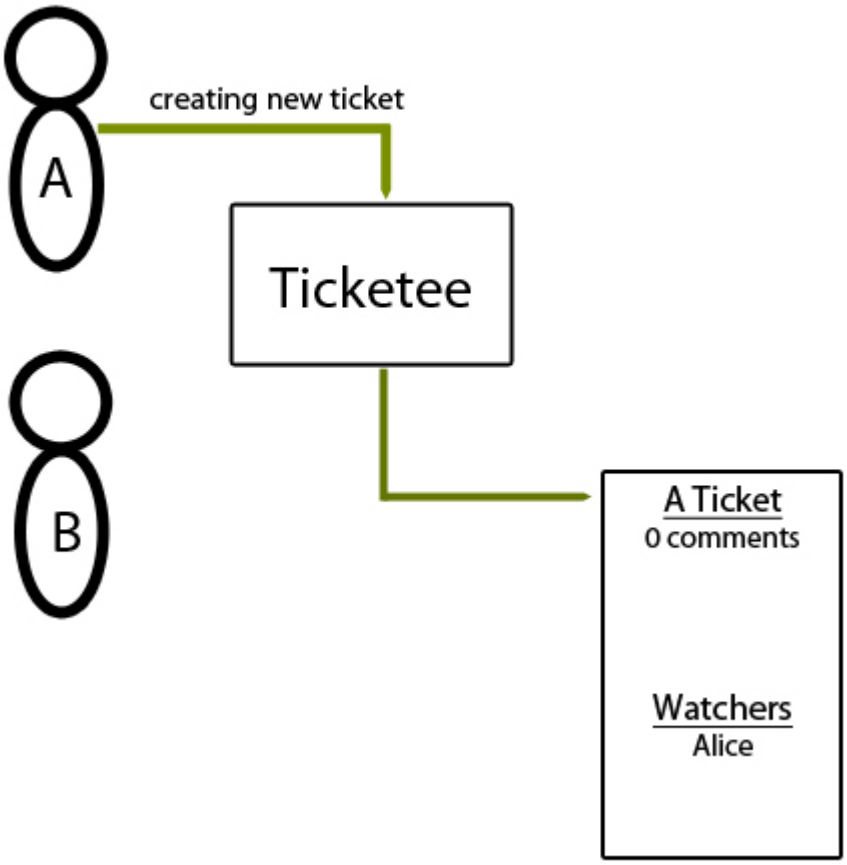

Figure 12.3 Alice creates a ticket

Here, alice@ticketee.com creates a ticket that will automatically subscribe her to be notified of any comments posted to it. Next, figure Figure 12.4

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

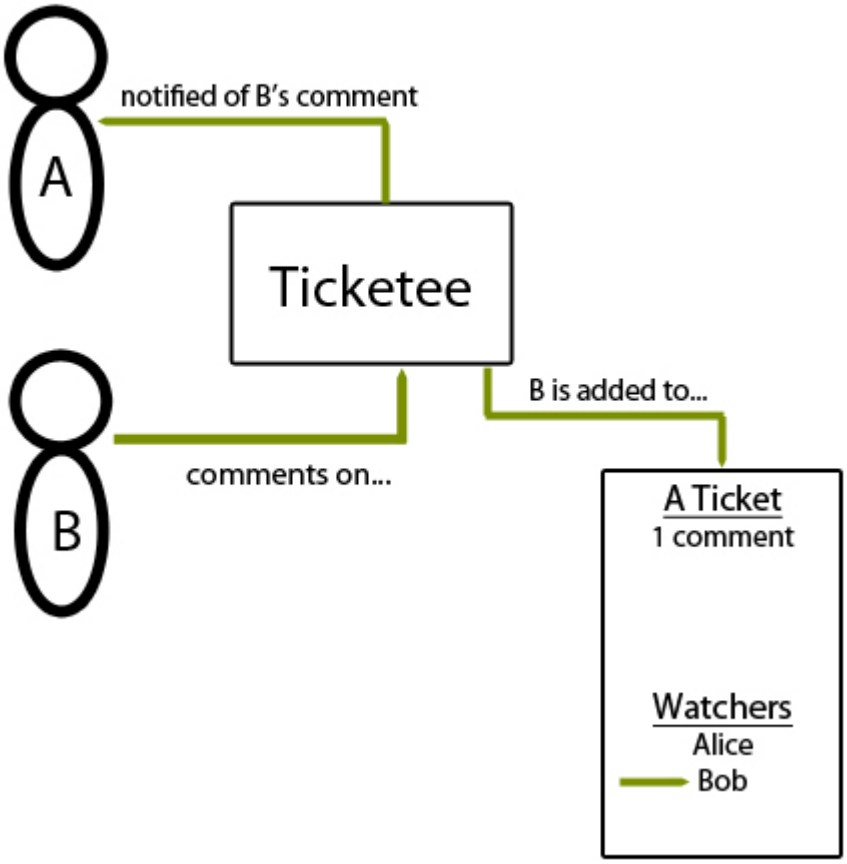

Figure 12.4 Bob comments on the ticket

Then bob@ticketee.com comes along and leaves a comment on the ticket, which should now subscribe bob@ticketee.com to these ticket updates. This is the feature that you'll code in a short while. After Bob has commented on the ticket, Alice receives a notification telling her that Bob has left a comment. Now that Bob is subscribed to the ticket, he should receive comment notifications every time somebody else--such as Alice--comments on the ticket, as shown in Figure 12.5

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

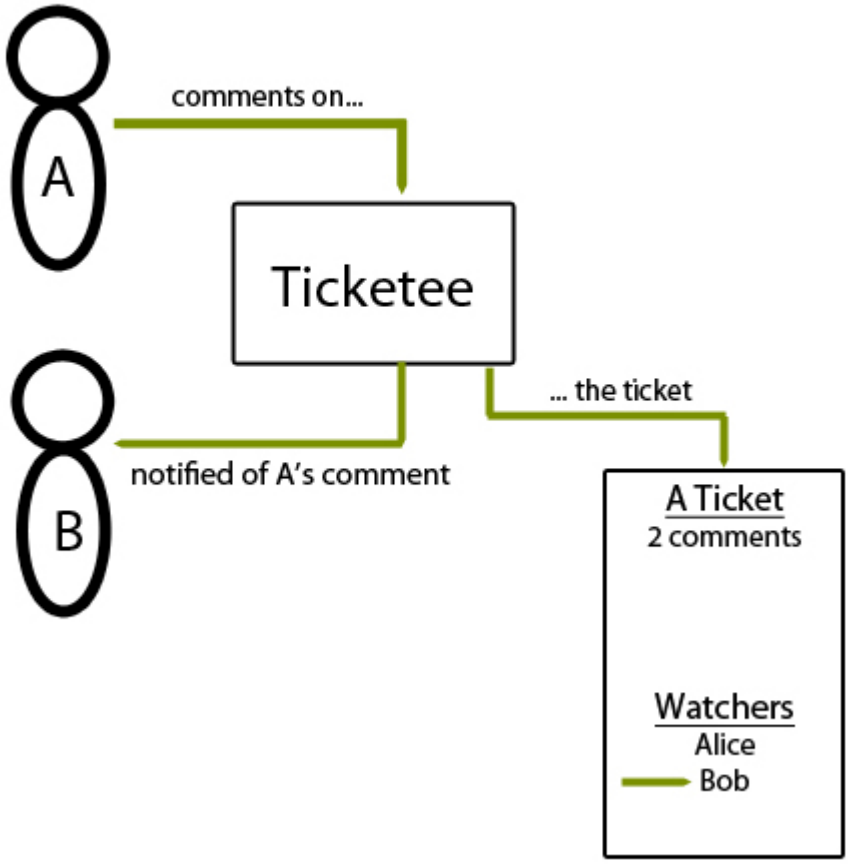

Figure 12.5 Alice comments on the ticket

In this case, alice@ticketee.com shouldn't receive a notification about a comment if she's the one posting it! With the scenario explained, you can write it in Capybara-form at the bottom of the "Ticket notifications" feature. Add the scenario from Listing 12.5 inside the feature of spec/integration/ticket_notifications_spec.rb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.5 Testing comment automatic subscription

```
scenario "Comment authors are automatically subscribed to a ticket" do
  click_link project.name
  click_link ticket.title
  fill_in "comment_text", :with => "Is it out yet?"
  click_button "Create Comment"
  page.should have_content("Comment has been created.")
  find_email!(alice.email)
  click_link "Sign out"

  reset_mailer

  sign_in_as!(alice)
  click_link project.name
  click_link ticket.title
  fill_in "comment_text", :with => "Not yet!"
  click_button "Create Comment"
  page.should have_content("Comment has been created.")
  find_email!(bob.email)
  lambda { find_email!(alice.email) }.should raise_error
  <co id="ch12_548_1"/>
end
```

In this scenario, you're already logged in as Bob (courtesy of the `before` in this feature). With Bob, you create a comment on the ticket, check that Alice receives an email and then sign out. Then you clear the email queue to ensure that Alice receives no emails 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 line for this scenario, the `find_email!` method would typically raise an exception and make the test fail, which is bad. You *want* it to raise an exception in this case because Alice should not receive an email. Therefore you wrap the method call in a `lambda` ❶ and use the `should raise_error` assertion from RSpec to validate that the method call does raise an exception.

When you run this scenario using `bin/rspec spec/integration/ticket_notifications_spec.rb:38` you'll see that Bob never receives an email from that final comment left by Alice.

```
Failure/Error: find_email!(bob.email)
  Could not find email  .
   Found the following emails:
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  []
```

This is failing on the step that checks if bob@ticketee.com has an email. You can therefore determine that bob@ticketee.com 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.

12.2.2 Automatically add a user to a watchlist

To keep users 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 people create a new ticket, and so you can apply the same logic to adding them to the list when they create a comment.

You can define another `after_create` callback in the `Comment` model by using this line:

```
after_create :creator_watches_ticket
```

Next, you need to define the method that this callback calls, which you can do by placing this code at the bottom of your `Comment` model:

```
def creator_watches_ticket
  ticket.watchers << 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.

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 re-running `bin/rspec spec/integration/ticket_notifications_spec.rb`.

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
1 example, 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? Let's have a look-see by running `rake cucumber:ok spec`. You should have this:

```
83 examples, 0 failures
```

Every test that you have thrown at this application is still passing, which is a great thing to see. Let's commit this change:

```
git add .
git commit -m "Automatically subscribe users
               to a ticket when they comment on it"
```

You now have automatic subscription for ticket notifications when a user creates a ticket or posts a comment to one, but currently there is no way to switch notifications *off*. To implement this, you'll add a "Stop watching" button that, when clicked, will remove the user from the list of watchers for that ticket.

12.2.3 Unsubscribing from ticket notifications

You'll 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 Figure 12.6.

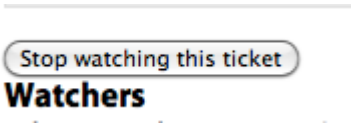

Figure 12.6 The watch button

Along with implementing the ability to turn *off* the notifications by clicking this button, you'll also add a way for the users to turn *on* notifications, using what will

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

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 1) creating their own ticket or 2) posting a comment.

You'll implement the "on" and "off" functionality simultaneously by writing a new feature in a new file at spec/integration/watching_tickets_spec.rb. Let's start with the code from listing 12.6.

Listing 12.6 Watching tickets feature setup

```
require 'spec_helper'

feature "Watching tickets" do
  let!(:user) { Factory(:confirmed_user) }
  let!(:project) { Factory(:project) }
  let!(:ticket) { Factory(:ticket, :project => project,
                                   :user => user) }

  before do
    define_permission!(user, "view", project)
    sign_in_as!(user)
    visit '/'
  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 and therefore they should see the "Stop watching this ticket" button on the ticket page, which you'll test by writing the scenario from listing 12.7 underneath the `before` inside this feature.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.7 Ticket watch toggling

```
scenario "Ticket watch toggling" do
  click_link project.name
  click_link ticket.title
  within("#watchers") do
    page.should have_content(user.email) <co id="ch12_645_1"/>
  end

  click_button "Stop watching this ticket"
  page.should have_content("You are no longer watching this ticket.")
  within("#watchers") do <co id="ch12_645_2"/>
    page.should_not have_content(user.email)
  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 ❶ on the page. When that user clicks the "Stop watching this ticket" button then they will be told that they're no longer watching the ticket, and their email will no longer be visible inside `#watchers` ❷

To begin to watch a ticket again, all the user has to do is press the "Watch ticket" button, which you can also test by adding the following code to this scenario:

```
click_button "Watch this ticket"
page.should have_content("You are now watching this ticket.")
within("#watchers") do
  page.should have_content(user.email)
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 right hand side of the ticket page, which will look like Figure 12.7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

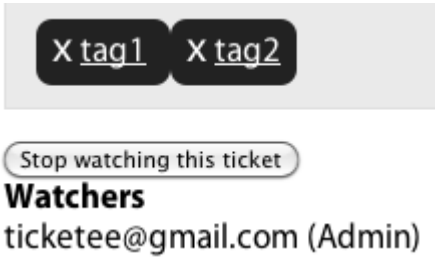

Figure 12.7 Who's watching

As usual, you'll see what you need to code *right now* to get your feature on the road to passing by running `bin/rspec spec/integration/watching_tickets_spec.rb`. You'll see that it's actually this watchers list, indicated by Capybara telling you that it can't find that element:

```
Failure/Error: within("#watchers") do
Capybara::ElementNotFound:
  Unable to find css "#watchers"
```

To get this feature to pass, you're going to need this element! You can add it to app/views/tickets/show.html.erb underneath the ticket `div`, and above the "Comments" `h3` tag by using the code from Listing 12.8

Listing 12.8 app/views/tickets/show.html.erb

```
<span id='watchers'>
  <strong>Watchers:</strong>
  <%= @ticket.watchers.map(&:email).to_sentence %>
</span>
```

You've created a `span` with the `id` attribute set to `watchers`, which is the element that your scenario looks for. In this `span` you collect all the watcher's emails using `map`, and then use `to_sentence` on that array. What this will do is turn the array of user's emails into a proper sentence, something such as "alice@example.com, bob@example.com and corey@example.com".

When you have this element and you run your feature again with `bin/rspec spec/integration/watching_tickets_spec.rb`, you'll see that your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

feature gets one step closer to passing by locating the user's email in the `#watchers` element, but it now can't find the "Stop watching this ticket" button:

```
Failure/Error: click_button "Stop watching this ticket"
Capybara::ElementNotFound:
  no button with value or id or text 'Stop watching this ticket' found
```

This button will toggle the watching status of the ticket of the current user, and the text will differ depending on if the user is or isn't watching this ticket. In both cases, however, the button will go to the same action. To get this next scenario to pass, you'll add the button to the `div#watchers` element you just created by using a helper, changing the first few lines of the element to this:

```
<span id='watcher'>
  <%= toggle_watching_button %>
```

This `toggle_watching_button` helper will only appear in views for the `TicketsController`, and so you should put the method definition in app/helpers/tickets_helper.rb inside the `TicketsHelper` module, using the code from listing 12.9 to define the method:

Listing 12.9 toggle_watching_button inside TicketsHelper

```
def toggle_watching_button
  text = if @ticket.watchers.include?(current_user)
    "Stop watching this ticket"
  else
    "Watch this ticket"
  end
  button_to(text, watch_project_ticket_path(@ticket.project, @ticket))
end
```

On the final line of this method, you use a new method: `button_to`. This method works in a similar fashion as `link_to` does, providing a user with an element to click on to go somewhere. In this case, the element is a button wrapped

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

in a form that points to the specified action. When the user clicks the button, it submits this form through a POST request, with the only parameter passed through being `params[:commit]`, which contains the text of the button.

Inside the `button_to`, you use a new route helper that you haven't defined yet. When you run `bin/rspec spec/integration/watching_tickets.feature` it will complain that this method is undefined when it tries to render the app/views/tickets/show.html.erb page:

```
And I follow "Release date"
  undefined method `watch_project_ticket_path' for ...
```

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 Listing 12.9

Listing 12.10 adding watch route to config/routes.rb

```
resources :projects do
  resources :tickets do
    collection do
      get :search
    end

    member do <co id="ch12_706_1"/>
      post :watch
    end
  end
end
```

The `button_to`'s purpose is to toggle the watch status of a single ticket, meaning you want to define a `member` route for your ticket resource. You put it inside the tickets resource, nested under the projects resource, because for your `watch` action you'll want to confirm that the person has permission to "view" this project. You define the route to the `watch` action with `post` because `button_to` generates a form by default, and a form's HTTP method will default to `POST`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

When you run your feature again using `bin/rspec spec/integration/watching` it will complain now because there is no `watch` action for your button to go to:

```
And I press "Stop watching this ticket"
  The action 'watch' could not be found for TicketsController
```

You're almost done! Defining this `watch` action is 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 you'll open app/controllers/tickets_controller.rb and define the action by using the code found in Listing 12.10

Listing 12.11 watch action inside TicketsController

```
def watch
  if @ticket.watchers.exists?(current_user)
    @ticket.watchers -= [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 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. This is because you can add this action to the list of actions that the `before_filter :find_ticket` runs on by changing these lines at the top of your controller:

```
before_filter :find_ticket,
              :only => [:show,
                        :edit,
                        :update,
                        :destroy]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To these lines:

```
before_filter :find_ticket,
              :only => [:show,
                        :edit,
                        :update,
                        :destroy,
                        :watch]
```

In this method you use the `exists?`, which will check if the given user is in the list of watchers. If they are, then you'll use `watchers -=` to remove a watcher from a ticket. If they aren't on the watchers list then you'll use `watchers` to add them to the list of watchers.

The `watch` action now defines the behaviour for a user to start and stop watching a ticket by pressing the button above the watchers list. When you run `bin/rspec spec/integration/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. Let's make sure that everything is working by running `rake cucumber:ok spec`. You should see the following output:

```
84 examples, 0 failures
```

Everything is still A-OK, which is good to see. Let's commit this change:

```
git add .
git commit -m "Add button so users can toggle
               watching on a ticket"
git push
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You've now got a way that a user can 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 going great in theoretically testing email, but you haven't yet configured your application to send out emails in the real world. Let's do that now.

12.3 Real world email

You've just created the beginnings of a way to send email in your application, but there's still a part missing: the SMTP server that receives your mail objects and then sends them out to their recipients. You could spend a lot of time configuring one yourself, but many people offer a free SMTP service, such as Gmail.[10] You'll use a Gmail account to send out tests of your emails, and you can use Action Mailer to connect to this service.

Footnote 10 SendGrid offers one too which you *would* use, but you're going to need to receive emails next and having a Gmail account will allow you to do that.

NOTE **Beware Gmail's daily send limit**

You wouldn't use Gmail to send or receive your emails if you were running a much larger application, but rather another webservice such as SendGrid. This is because Gmail has a limit of about 200 sent emails a day and if there are 200 tickets updated in a single day then it's goodbye email cap. Gmail is great for light email usage, but if you want to scale up your usage, SendGrid is one of the best options out there.

Action Mailer has a setting that you can use to set up your SMTP connection:

```
ActionMailer::Base.smtp_settings = {
  :username = "youraccount@example.com",
  :password = "yourpassword"
  ...
}
```

Before you dive into setting this up, you're going to need a feature to ensure that it always works. When you set up your application to send emails in the real world, it may work from the get-go, and you can test it manually by sending out emails in your application through `rails server`, but how do you ensure that it works all the time? The feature will provide that insurance.

When you're done here, you'll have your application hooked up to Gmail's

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

SMTP server so that you can send emails in the real world, and you'll have a Cucumber feature to ensure that it's never broken[11] Let's jump into it.

Footnote 11 That is to say, if you run all the tests and they all pass before you commit, then you know that your Gmail connection would be working, too.

12.3.1 Testing real world email

In this section, you'll create a feature in which you set up Action Mailer to send out emails to Gmail's SMTP service. You'll then update a ticket's comment, which should trigger the emails to be sent to the real world. Finally, you'll check the Gmail account (using the Mail gem on which Action Mailer is based), to make sure that the email was received. If it wasn't received, then the cause is most likely a configuration problem, such as an invalid password.

Let's write this new feature in a new file called spec/integration/gmail_spec.rb a bit at a time. You'll start with just these lines from Listing 12.11:

Listing 12.12 Gmail feature setup

```
require 'spec_helper'

feature "Gmail" do
  let!(:alice) { Factory(:confirmed_user) }
  let!(:me) { Factory(:confirmed_user,
                      :email => "youraccount@example.com") }

  before do
    ActionMailer::Base.delivery_method = :smtp
  end

  after do
    ActionMailer::Base.delivery_method = :test
  end
end
```

Here you're creating two new user accounts: one that's just a typical test user, and another user which *should* have your real email address listed. In the example above, we use "youraccount@example.com", as we don't want to bombard any real email address with real emails! If you don't have a Gmail address, sign up! It's free and will only take a minute.

Inside the `before` block for this feature, you set Action Mailer's delivery method to `smtp`. This means that it will send out *real* emails, delivering them via the SMTP protocol[12]There's some more setup you'll need to do to get this to send

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

out real emails, and you'll get around to doing that in just a bit. Let's fill out the feature a bit more first.

Footnote 12 http://en.wikipedia.org/wiki/SMTP

You need to set up a project that both Alice and you can see, and create a ticket on that project that is posted by you. In a short while, you'll get Alice to sign in and post a comment to this ticket, which should make an email appear in your inbox. You'll then check this email using the Mail gem. You'll now set up the project and ticket by putting these lines underneath the `let!` calls that are already inside spec/integration/gmail_spec.rb:

```
let!(:project) { Factory(:project) }
let!(:ticket) do
  Factory(:ticket, :project => project,
                   :user => me)
end
```

And then these lines to set up the permissions for the users in the `before` block of the feature:

```
define_permission!(alice, "view", project)
define_permission!(me, "view", project)
```

In these setup lines, you set up that Alice and your user object both have the "view" permission on the project. After this, you need a ticket that you've created so that Alice can post a comment to it and you can receive an email notification informing you of what Alice has posted.

Now you can get to the *meat* of your feature: the scenario itself. In this scenario, you want to log in as "alice@ticketee.com," visit the ticket that has been created in the setup and post a comment to it. After all that's said and done, you need to assert that your Gmail inbox has one new message. The code for the scenario should therefore look like Listing 12.12

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.13 Receiving a real-world email scenario

```
scenario "Receiving a real-world email" do
  sign_in_as!(alice)
  visit project_ticket_path(project, ticket)
  fill_in "comment_text", :with => "Posting a comment1"
  click_button "Create Comment"
  page.should have_content("Comment has been created.")

  ticketee_emails.count.should == 1
  email = ticketee_emails.first
  subject = "[ticketee] #{project.name} #{ticket.title}"
  email.subject.should == subject
  clear_ticketee_emails!
end
```

In this scenario, you walk through the process of signing in as Alice and creating a ticket as here. The assertion on the final line of this scenario will call the `ticketee_emails` method which will log into your Gmail account and check for emails from the application. Te `ticketee_emails` method is undefined at the moment and you'll define it in a short while.

When you run this feature using `bin/rspec spec/integration/gmail_spec.rb`, you'll see that your feature fails when Alice presses the "Create Comment" button:

```
And I press "Create Comment"
  Connection refused - connect(2) (Errno::ECONNREFUSED)
  .../net/smtp.rb:551:in `initialize'
```

Remember before how it was mentioned that Action Mailer would (by default) try to connect to an SMTP server running on `localhost`? That's what is happening here, because when a comment is updated, a notification will be attempted through an SMTP server running on localhost. If that server is not running, then you will see this "Connection refused" error. You didn't see this previously because it's only now that you've switched `delivery_method` to `:smtp` You don't have one running locally[13] so it's unable to connect. You can tell that it's now using SMTP, because the first line of the stacktrace points to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

net/smtp.rb in Ruby's standard library, which is what Action Mailer (by way of the Mail gem) uses to connect to SMTP servers. Therefore, you must change something in order to make this work once more.

Footnote 13 Unless you've got it set up from some other place that's not this book.

12.3.2 Configuring Action Mailer

To fix this error, you must tell Action Mailer to connect to your Gmail server so that it has a way to send out emails. You can create a new file in config/initializers that provides Action Mailer with the necessary information it needs. But what would this information be? Well, let's hop on over to Google's "Configuring other mail clients"[14] page, where you'll see the table from figure Figure 12.8

Footnote 14 http://mail.google.com/support/bin/answer.py?hl=en&answer=13287

Incoming Mail (POP3) Server - requires SSL:	`pop.gmail.com` **Use SSL**: Yes **Port**: 995
Outgoing Mail (SMTP) Server - requires TLS or SSL:	`smtp.gmail.com` (use authentication) **Use Authentication**: Yes **Port for TLS/STARTTLS**: 587 **Port for SSL**: 465
Account Name:	your full email address (including `@gmail.com` or `@your_domain.com`)
Email Address:	your email address (`username@gmail.com` or `username@your_domain.com`)
Password:	your Gmail password

Figure 12.8 Configuring other mail clients

You're trying to send email so you want to use the "Outgoing mail" section, which tells you to use "smtp.gmail.com" as the server. You'll connect to it using TLS, so you'll connect on port 587. The account name and password should be the gmail address and password for your email address. With these settings, you'll create a config/initializers/mail.rb file that looks like Listing 12.13

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.14 config/initializers/mail.rb

```
ActionMailer::Base.smtp_settings = {
  :user_name => "youraccount@gmail.com",
  :password => "password",
  :address => "smtp.gmail.com",
  :port => 587,
  :enable_starttls_auto => true
}
```

WARNING **Do not commit mail.rb to Git**

Because config/initializers/mail.rb contains sensitive data, we would recommend that you do not commit this file to your Git repository.

You can tell Git to completely ignore the file by adding a single line to another file called .gitignore at the root of your Rails application. That line is just the name of the file you want ignored:

```
config/initializers/mail.rb
```

Later on when you go to create a commit for all your changes, Git will acknowledge that you don't want this file committed to your Git repository and will leave it out.

With these settings now in this file, you can rerun `bin/cucumber features/gmail.feature` to see that it is now erroring out because `ticketee_emails` is not defined:

```
Failure/Error: ticketee_emails.count.should == 1
NameError:
  undefined local variable or method `ticketee_emails' ...
```

With this `ticketee_emails` method, you'll connect to Gmail using your email settings you've just specified in config/initializers/mail.rb and then check for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

an email with a subject beginning with "[ticketee]", which should be sent earlier in this scenario. You need to define this `ticketee_emails` method now to connect to Gmail.

12.3.3 Connecting to Gmail

You've now sent the email to the server, but you don't have any steps in place to read these emails from your Gmail account and check that one of the emails is from Ticketee. As you can now almost anticipate, there's a gem that can help you with this called quite simply "gmail". This gem will let you connect to a Gmail server using the username and password you just used to set up an SMTP connection, and also read the emails for that account. The code it uses looks like this:

```
Gmail.connect(username, password)
```

You'll also use this gem in the next major section, when you look at how you can receive emails into your application. It's a pretty neat gem, and it's got a great README, which can be seen at http://github.com/nu7hatch/gmail.

To install this gem, you must add it to the Gemfile by adding this line inside the group block for development and test, because you only want this gem used in those environments:

```
group :test, :development do
  gem 'gmail', '0.4.0'
  ...
end
```

Then you need to run `bundle install` to install this gem so that you can use it. When `bundle install` is finished running, create a new file called spec/support/gmail_helpers.rb and put the content from Listing 12.14 in that file:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.15 GmailHelpers

```
module GmailHelpers
  def gmail_connection
    settings = ActionMailer::Base.smtp_settings
    @gmail_connection ||= Gmail.connect(settings[:user_name],
                                        settings[:password])
  end

  def ticketee_emails
    gmail_connection.inbox.find(:unread,
                                :from => "Ticketee App")
  end

  def clear_ticketee_emails!
    ticketee_emails.map(&:delete!)
  end
end

RSpec.configure do |c|
  c.include GmailHelpers
end
```

In this new file, you've created a module called `GmailHelpers` and within it defined three helper methods that will assist you in your testing of gmail email sending. The first method, `gmail_connection` will establish a connection to Gmail using the settings from `ActionMailer::Base`. The second method will call the `inbox` method on this connection and then retrieve all the unread emails from the inbox that are from "Ticketee App". The third and final method gets all the emails and deletes them, leaving a pristine state for the next time the test is run. The final few lines of this file make these three methods available in the RSpec tests of your application.

Now that you've defined the `ticketee_emails` method that the feature was after (and two other utility methods), when you re-run `bin/rspec spec/integration/gmail_spec.rb` you'll see that this test cannot find the email:

```
Failure/Error: ticketee_emails.count.should == 1
expected: 1
     got: 0 (using ==)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

If you sign into your Gmail account, you'll see that there is actually an email there, as shown in Figure 12.9.

Figure 12.9 Gmail email

Sure enough, the email is there, but the "From" address on the email (on the left of the above image) says "me" rather than "Ticketee App", which is why the email is not being found. You will need to change the "From" address to display "Ticketee App" instead.

To make this change, you will need to alter the "From" address defined within your mailer. Currently this is defined as being "from@example.com". When the email is sent to Gmail, it recognises this is a fake address and so it will substitute it for your real one. Therefore you will need to use your real address and assign a name to the from field as well. Open up app/mailers/notifier.rb and change the `default` line from this:

```
default from: "from@example.com"
```

Into this:

```
from_address = ActionMailer::Base.smtp_settings[:user_name]
default from: "Ticketee App <#{from_address}>"
```

What this will do is send the email using the address as configured in your mail settings (config/initializers/mail.rb) and will set a display name of "Ticketee App", which is what the feature is looking for. When you run the feature again with `bin/rspec spec/integration/gmail_spec.rb` you'll see that it's now finding the email correctly:

The test sends out an email to your personal inbox using the settings provided by the Action Mailer configuration inside `config/initializers/mail.rb`, and then uses those same settings again to sign in to your Gmail account and validate that the email arrived successfully. Once it's validated that, then it clears out all the emails from Ticketee.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Everything should still be working now. You haven't changed anything that would have broken your existing features or specs, but it's still great practice to run them just to make sure. Let's do this by running `rake spec` now. You'll see the following output:

```
85 examples, 0 failures
```

Indeed, nothing is broken. Let's make a commit now:

```
git add .
git commit -m "Set up application to connect to Gmail to send emails"
git push
```

You've now got your application sending out emails in the real world using Gmail as the server. With these settings, the emails notifying users that tickets have had new comments posted to them, as well as the confirmation emails sent from Devise for new user sign ups, will be sent out through Gmail.

So you have the *sending* emails part of your application done, but what about if you wanted to let users reply to comments by replying to the email notification they receive in their inbox? That would be cool. To do this, you're going to need to figure out how you can receive emails with Rails.

12.4 Receiving emails

You'd now like to add a feature to Ticketee where users can reply to the email notifications for a new comment on a ticket, and by replying create a new comment on that ticket with their text. Many other applications do this by having an email such as this:

```
== ADD YOUR REPLY ABOVE THIS LINE ==
Bob has just updated the "Due date" ticket for "TextMate 2"
```

Text above the "ADD YOUR REPLY ABOVE THIS LINE" will be parsed out and turned into a new object. In Ticketee, this would be a comment.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

In the previous section, you learned how you could connect to a Gmail account to check to see if there was an email that had a subject beginning with "[ticketee]." You can use the same method in order to check for replies to your emails too, but you need to make one small modification.

To determine what ticket and project the reply is directed at, you need to tag the emails in a certain way. The best way to do this is to add the tags to the email addresses themselves, so that an email address with a tag looks like "ticketee+tag@gmail.com," where the "+tag" part of the email is "ignored" and the email arrives in "ticketee@gmail.com"'s mailbox. For your emails, you'll set a reply-to address such as "ticketee+61+23@gmail.com, where the first number is the project ID and the second number is the ticket ID.

You're not going to post comments straight from emails. Check to see if the user has permission to view the project where the ticket is, which means that they would be able to create a comment for that ticket too. If they're unable to post a comment to that ticket, you'll assume the user is trying to do something malicious and just ignore their email.

To parse these emails, you'll be using the `receive` method in an ActionMailer class, which takes an email object and allows you to process it.

A quick summary: you're going to use the Gmail gem to check for emails in your inbox that are replies to comment notifications and then parse them using Action Mailer into new `Comment` objects. If a user is restricted from viewing a project, then you'll ignore their emails.

Firstly, you'll want to check that the outgoing email contains the tag on the "from" address, so that when a user replies to it you know what project and ticket they're replying to.

12.4.1 Setting a reply-to address

By having a different "from" address set on the outgoing email, you'll be able to determine what project and ticket the user's reply comment should be created on. To ensure that all outgoing emails from the `comment_updated` method in `Notifier` have this set, you're going to write a simple test.

Let's now open spec/mailers/notifier_spec.rb and add in a new test that ensures that the Reply-To address for the email is correct. The code to do this can be seen in Listing 12.15.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 12.16 Reply-To test

```
it "correctly sets the Reply-To" do
  address = "youraccount+#{project.id}+#{ticket.id}@gmail.com"
  email.reply_to.should == [address]
end
```

Here you test that the `reply_to` for the email contains the project and ticket ids that are related to the comment you setup. You will need to replace "youraccount" in this example with your real GMail account name. With this information contained in the email address, you'll be able to know what project and ticket to create the comment for when a user replies to that email.

Now when you run `bin/rspec spec/mailers/notifier_spec.rb`, you'll see that it fails with this error:

```
expected "youraccount+1+1@gmail.com"
          got nil
```

Right then! This test is failing because there is currently no Reply-To address set for this email. A failing test is a great place to begin with, and now you need to fix it. Let's open up app/mailers/notifier.rb and add a `:reply_to` option to the `mail` call inside `comment_updated` method:

```
mail(:to => user.email, :subject => subject,
     :reply_to => "Ticketee App <youraccount+" +
     "#{@project.id}+#{@ticket.id}@example.com>")
```

This will change the from address on emails that go out to your users by tagging the addresses with the project and ticket id. When the user replies to this email, you can use this tag to find the project and ticket that you need to create a new comment on. Let's run `bin/rspec spec/mailers/notifier_spec.rb` again to see it now pass:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
2 examples, 0 failures
```

Now when a user replies to an email that they receive, they will be replying to this specially crafted email that contains the project and ticket ids, which can then be used to refer to the project and tickets so that the application can create comments on them. The next step is of course working on receiving these replies!

12.4.2 Receiving a reply

With the correct reply-to set, you can implement the feature responsible for creating new comments from email replies. You'll create a new class for dealing with incoming email and call it `Receiver`, placing it in app/mailers by running this command:

```
rails g mailer receiver
```

This will generate the mailer you'll use for receiving email, as well as the RSpec file that you can use to write the tests for the class in. To test this particular feature, you'll use a very similar set up to the spec/notifier_spec.rb test that you just wrote. This test needs to generate a comment and then a reply to the email you would receive from the comment. This new reply should have the same body as the original email, but prefixed with some text. This new text will become the new comment.

At this stage you only want to check that you can parse emails using this new class and a currently undefined method on it called `parse`. This method will take a `Mail::Message` object and create a new comment on a ticket.

```
require 'spec_helper'

describe Receiver do
    let!(:ticket_owner) { Factory(:user) }
    let!(:ticket) { Factory(:ticket, :project => project,
                                     :user => ticket_owner) }
    let!(:commenter) { Factory(:user) }
    let(:comment) do
      Comment.new({
        :ticket => ticket,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
        :user => commenter,
        :text => "Test comment"
      }, :without_protection => true)
    end

  it "parses a reply from a comment update into a comment" do
    email = Notifier.comment_updated(comment, ticket_owner)
```

With this email object, you can build a new `Mail::Message` "reply" to this email using these lines:

```
mail = Mail.new(:from => "user@ticketee.com",
                :subject => "Re: #{comment_email.subject}",
                :body => %Q{This is a brand new comment
                <co id="ch12_2490_1"/>
                    #{comment_email.body}
               },
               :to => comment_email.from)
```

With these lines, you're constructing a new reply using the body from the original email to generate a multi-lined string with "This is a brand new comment" before the body of the first email. The first line in this first email will eventually be "== ADD YOUR REPLY ABOVE THIS LINE ==", which is how you distinguish what should be the new content for the comment and what's just from the old email.

The final step for this spec is to actually parse the incoming email using the `Receiver` class, and to check that it changes the related ticket's comment count by 1.

```
lambda { Receiver.parse(mail) }.should(
   change(comment.ticket.comments, :count).by(1)
 )
```

Finish up this example by checking that the last comment for the ticket contains the text from the reply:

```
ticket.comments.last.text.should eql("This is a brand new comment")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The spec/mailers/receiver_spec.rb should now look like Listing 12.16

Listing 12.17 spec/mailers/receiver_spec.rb

```
require 'spec_helper'

describe Receiver do
  let!(:project) { Factory(:project) }
  let!(:ticket_owner) { Factory(:user) }
  let!(:ticket) { Factory(:ticket, :project => project,
                                   :user => ticket_owner) }
  let!(:commenter) { Factory(:user) }
  let(:comment) do
    Comment.new({
      :ticket => ticket,
      :user => commenter,
      :text => "Test comment"
    }, :without_protection => true)
  end

  it "parses a reply from a comment update into a comment" do
    original = Notifier.comment_updated(comment, ticket_owner)
  <co id="ch12_921_1"/>
    reply_text = "This is a brand new comment"
    reply = Mail.new(:from => "user@ticketee.com",
                     :subject => "Re: #{original.subject}",
                     :body => %Q{#{reply_text}
                    <co id="ch12_921_2"/>
                          #{original.body}
                     },
                     :to => original.reply_to)
    lambda { Receiver.parse(reply) }.should(
     change(ticket.comments, :count).by(1)
   )
   ticket.comments.last.text.should eql(reply_text)
  end
end
```

In this spec, you build a comment and reference the ticket for it and then build an email using the `comment_updated` method from `Notifier` ❶. From this email, you build a new `Mail` object and give it the content "This is a brand new comment" followed by the content of the original email. Passing this new email

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

message through the `parse` method on the `Reciever` is supposed to increase the comment count for the ticket by 1, and the final comment for the ticket should contain the same content as the reply email.

When you run this spec using `bin/rspec spec/mailers/receiver_spec.rb` you'll be told this:

```
Failure/Error: lambda { Receiver.parse(mail) }.should(
  undefined method `parse' for Receiver:Class
```

To make this spec parse, you need to define this method.This method should take a `Mail::Message` object, read out everything from the body of that object above the line "ADD YOUR REPLY ABOVE THIS LINE", and create a comment from it. Before you go about defining this method, take out the `default` definition at the top of this mailer because you won't be using it.

You can begin to define this method in app/mailers/receiver.rb like this:

```
def self.parse(email)
   reply_separator = /(.*?)\s?== ADD YOUR REPLY ABOVE THIS LINE ==/m
   comment_text = reply_separator.match(email.body.to_s)
```

Here you match the body of the email with the expected reply separator, getting back either a `MatchData` object (indicating the email is a valid reply to a comment), or `nil`. If you get back a valid reply then you do this:

```
if comment_text

   to, project_id, ticket_id =
     email.to.first.split("@")[0].split("+")
```

Here you take the list of `to` addresses for the email, get the first of them and then `split` it on the @ symbol. This separates the username and the domain name in your email. The username contains the project id and ticket id, which you get by calling `split` again, this time separating the individual elements by the + symbol.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Next, you need to find the relative project, ticket, and user for this email, which you can do using these lines all inside the `if` that you just opened:

```
project = Project.find(project_id)
ticket = project.tickets.find(ticket_id)
user = User.find_by_email(email.from[0])
```

Finally, you need to create the comment from the email body (stripping all extra spaces from it) and close the `if`, which is done with the following lines:

```
ticket.comments.create({
  :text => comment_text[1].strip, <co id="ch12_921_1"/>
  :user => user
}, :without_protection => true)
  end
end
```

The `[1]` here will get the first match for the `comment_text`, which will be the new comment's text, throwing `strip` on the end in case they've got a couple of extra spaces / lines between the comment text and the separator. You'll need to use the `without_protection` option here so that the user attribute can be assigned at the same time as the text attribute.

That's the final bit of code you need to get in the app/mailers/receiver.rb. The whole file should now look like this:

```
class Receiver < ActionMailer::Base
  default from: "from@example.com"

  def self.parse(email)
    reply_separator = /(.*?)\s?== ADD YOUR REPLY ABOVE THIS LINE ==/m
    comment_text = reply_separator.match(email.body.to_s)
    if comment_text
      to, project_id, ticket_id =
      email.to.first.split("@")[0].split("+")
      project = Project.find(project_id)
      ticket = project.tickets.find(ticket_id)
      user = User.find_by_email(email.from[0])

      ticket.comments.create({
        :text => comment_text[1].strip,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
        :user => user
        }, :without_protection => true)
    end
  end
end
```

When you run this spec again with `bundle exec rspec spec/mailers/receiver_spec.rb` it will still fail:

```
Failure/Error: lambda { Receiver.parse(mail) }.should(
  count should have been changed by 1, but was changed by 0
```

This is because your original comment notification doesn't have the reply separator, and therefore the `if` condition in the `parse` method you just wrote says "oh, can't find it then I'll just ignore this email," or something to that effect. So in order to get this to work, you must add that line to the comment notification. You can do this by opening up app/views/notifier/comment_updated.text.erb and adding this line to the beginning of both files:

```
== ADD YOUR REPLY ABOVE THIS LINE ==
```

Now when you run your spec once more with `bundle exec rspec spec/mailers/receiver_spec.rb`, it will pass because the `parse` method can now find the separator.

```
1 example, 0 failures
```

Great! Now you have a feature inside your application that will receive emails and translate them into comments for tickets. Alright, now that you've got that feature passing, does everything else still work? Let's find out by running `rake spec`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
87 examples, 0 failures
```

Good! Everything is still going great. Let's commit the new feature:

```
git add .
git commit -m "Add Receiver class to receive emails"
git push
```

Right, this feature isn't complete quite yet, as it only takes mail objects but doesn't actually do any of the fetching itself. You'll revisit this feature in chapter 15 and complete it there. This is a great start, however.

12.5 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 choose to subscribe or unsubscribe to a ticket. By doing this, all users, and not just those who created the ticket, can choose to receive emails when a ticket has had a comment posted to it. This way, all users can stay up to date on tickets they may be interested in.

Next, you made sure that you could actually send emails into the real world by connecting to a real Gmail account using Action Mailer's SMTP settings. You also ensured that when you send an email using the STMP setting, you can read it from the server by using the `gmail` gem.

By actually sending emails into the real world, you're bringing your application one step closer to being complete. Now you'll be able to put the application on a server and it should work just as it does in your tests. But, you're going to polish your application a little more before you do that.

The next chapter covers how you can use Rails to present your data to other developers so that they can create other applications or libraries to parse it into new and interesting formats.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Index Terms

ActionMailer, delivery_method
ActionMailer::Base, default
ActionMailer::Base, deliveries
ActionMailer::Base, mail
button_to
has_and_belongs_to_many, :join_table option
has_and_belongs_to_many :class_name option
Observers
Routing, member

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13 Designing an API

In the past chapters, you've created a great application that allows your users to manage projects and tickets through a web browser. In this chapter, you are going to create a way for your users to manage this content through what's known as an API ("Application Programming Interface"). Like its name implies, an API is a programming interface (for your application) that returns either JSON (JavaScript Object Notation[1]) or XML[2] data for its requests, which are the two most common formats for modern APIs to return information in. People can then create programs or libraries (referred to as "clients") to read and present this data in any way they see fit.

Footnote 1 http://json.org

Footnote 2 http://xml.org

One great example of how an API is used is Twitter. Twitter has had an API for an exceptionally long time now, and people have written Twitter clients for just about every operating system out there using this API. The functionality of the site is effectively the same, but people have come up with interesting ways of displaying and modifying that information, such as the Twitter for Mac clients.

There are many, many Rails sites out there that already provide an API interface, such as GitHub[3], and (as previously mentioned) Twitter[4]. Both of these APIs have exceptionally well-documented examples of how to use them and what can be done with them, which is a great way to convince people to start using your API. API documentation, however, is an exercise best left to the reader after this chapter's done.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 3 The octopi gem was built to interact with the GitHub API--you can see the source at http://github.com/fcoury/octopi

Footnote 4 The (non-official) twitter gem was built to interact with the API that Twitter provides; you can see the source of this gem at http://github.com/jnunemaker/twitter

APIs aren't unique to Rails. There are plenty of other sites out there that have implemented their own APIs, such as the StackExchange services that run on Microsoft.Net. Furthermore, APIs created in one language are not for exclusive use of API clients written in that specific language. For example, if you wanted to parse the data from the StackExchange API in Ruby, you could do that just fine.

In Rails, however, it's extremely easy to make a modern API for an application, as you'll see in this chapter. Our application will serve in two formats: JSON and XML.

Back to the Twitter and GitHub examples now, and one further thing to note is both of these APIs are also versioned. The point of this is that an API should be presented in a "frozen" state and should not be modified once it's considered stable. This way, a user is be able to use an API without fear of it changing and potentially breaking the code that they're using to parse the data from the API.

Twitter has a URL such as http://api.twitter.com/1/statuses/public_timeline.json that has the version as the first "part" of the URL after the site name and then the format as an extension at the end. Github's is slightly different, with a URL such as http://github.com/api/v2/json/repos/show/rails3book/ticketee having the version prefixed with a "v" as the second part of the URL, and the format as the part of the URL directly after. The "v" prefix here makes the version part of the URL clearer to those who are reading it, which is a good thing.

You're going to "borrow" ideas from both of these API designs, presenting your API at the base URL of /api/v1 and the format at the end of the URL, like most web requests. Our URLs will look like /api/v1/projects.json, which will be the URL to return a list of projects that the user is able to read. The reason for this versioning is so that you can always provide data that is predictable to your end users. If you wished to change the format of this data, you would create a new version namespace, which is the final thing we look at towards the end of this chapter. Really, these new version numbers can be whatever you wish, with "minor" versions such as "0.1" being the standard for unstable APIs, and major versions such as "1", "v1", or "1.0"being the standard for the stable, fixed APIs[5].

Footnote 5 Although logical (incremental) versioning is recommended to stave off questions such as "What were they thinking?!" and "Are they crazy?". This is often referred to as "Semantic Versioning," http://semver.org/

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To check what user is making a request to your API, you'll use a token-based authentication system, which is something that the Devise gem can be configured to provide. You'll require the token attribute to be passed through in every API request, as without it you cannot be sure of who is doing what with the API. You can then restrict things, such as the list of projects, based on the user's permissions. You can also use tokens to track the number of requests the user has performed on the API and then block them if they make more than 100 requests per hour, which is one of the final things you look at in this chapter. Twitter and GitHub both implement rate-limiting so that people do not spam the system with too many API requests.

Along the path of developing this API, you'll learn additional Rails goodies such as the `to_json` and `to_xml` methods, which will convert an object into JSON or XML representations of that object respectively, as well as the `respond_with` and `respond_to` controller methods, which are responsible for serving data in different formats.

When you're done with this chapter, you'll have a nice solid API that other developers can build upon. Other applications (be it Rails or not) will also be able to read the data from your application.

13.1 The projects API

To get started with Ticketee's API, you're going to write the projects part of it. In this section, we make extensive use of the `respond_with` and `respond_to` methods, which are new in Rails 3.

Before you go about implementing the first building blocks for your API, you'll learn about a module called `Rack::Test::Methods`, provided by the `rack-test` gem, which will allow you to easily test your API.

After that, you'll begin writing the API by creating the `index` action, which will be responsible for displaying all the projects in a JSON format. Next, implement token-based authentication so you can know who's who when they access the API. This will allow you to restrict the list of projects shown in the `index` action to only those that the user should see. Later on, you'll get this action to return XML as well as the JSON output.

With an API, you don't need to provide a `new` and `edit` actions, as this functionality should be provided by the client that is accessing the API. Instead,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you'll only write the "action"[6] parts of your API: the `create`, `show`, `update` and `destroy` actions. Along the way, you'll be restricting the `create`, `update` and `destroy` actions to administrators of the application.

Footnote 6 An absolutely terrible pun. Forgive us.

When learning about these actions, you'll see a lot of reference to HTTP status codes, which are the standard for all pages of the web. These status codes play a critical role in an API, providing key information such as if the request was successful (the 200 status code) or if the user is unauthorized (the 401 status code) to perform the request. These are standardized ways of quickly informing people of the result of their requests.

TIP **Cheat!**

There's a handy gem called `cheat` that provides cheat sheets for a number of things, including one for HTTP status codes. You can install this gem using the `gem install cheat` command and then bring up the cheat sheet for status codes using `cheat status_codes`.

However if you're on Windows this won't work as Cheat requires a function not found on your system. Instead, go to http://cheat.errtheblog.com/b where you can view the list of all the cheat sheets.

To begin writing this API, you'll need to define the routes to it. Without routes, making requests to /api/v1/projects.json will forever be fruitless. If you recall from this chapter's introduction, the API URL that you'll be using looks like /api/v1/projects.json. Previously, when you wanted URLs to be prefixed with a name (such as back in chapter 7), you used a `namespace` method for them. You're going to do the same thing here, except you'll use a namespace within another namespace. Let's open config/routes.rb and add the code from Listing 13.1 to the top of the routes definition.

Listing 13.1 config/routes.rb

```
Ticketee::Application.routes.draw do
  namespace :api do
    namespace :v1 do
      resources :projects
    end
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
...
```

This new route defines routes and routing helpers for the projects resources, such as /api/v1/projects, and `api_v1_projects_path` respectively. You're going to need to be serving content from this route, namely a list of projects. This list will be served in one of two forms: XML or JSON. Before you actually implement the code that makes these responses get served, you're going to write a new RSpec test that makes sure these routes return the right data. To help you along, you'll be using a feature provided by one of the dependencies of Rails: the `rack-test` gem.

This gem provides you with a module called `Rack::Test::Methods`, which contains methods such as `get`, `post`, `put` and `delete`. Look familiar? They should. They're the 4 basic HTTP methods that you use when making requests. The methods from `Rack::Test::Methods` take a path on which to make a request and then return a Rack response (an `Array`) that consists of three parts: the HTTP status code, the HTTP headers (in `Hash` form), and the body. The simplest Rack response would look something like this:

```
[200, {}, "Hello World!"]
```

The first element of this result is the HTTP status code, and in this case it indicates that your fictional response was 200, or in human-terms: OK. The second element contains no special HTTP headers, but you'll see these as you progress in the next section. Finally, the third element contains a string, which represents the body of this request, returning the string "Hello World!".

Using `Rack::Test::Methods`, you can initiate requests to your application's API that then return these Rack responses, and you can then use these responses to check that your API is responding in the correct way. You're purposely not using the standard RSpec controller tests here to make sure that the precise URLs are responding in the correct manner, instead of only testing that the actions are doing what you tell them.

Let's get into writing this initial test for your API.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13.1.1 Our first API

You're going to continue on the running theme of "test everything" with your API, and with the `rack-test` gem you've got the necessary tools to test what your API's URLs are doing. Rather than testing this in Cucumber, you're instead going to use RSpec, which provides a nicer DSL for testing your APIs. To begin with, you're going to create a new folder structure at spec/api/v1. You should name the apis directory as a plural for two reasons: the first being that it matches the consistency of the other directories in the spec directory and secondly, it may one day contain more than one API version. Then you'll create a new file called spec/api/v1/projects_spec.rb and you'll begin to fill it with the following:

```
require "spec_helper"

describe "/api/v1/projects", :type => :api do <co id="ch13_20_1"/>

end
```

There's much more code to come after this short snippet, but it's a pretty good start.

In the `describe` block here you pass through an option of `:type => :api`. What does this do? Well, you can use it to modify the behavior of this `describe` block in many ways, such as including modules. The `Rack::Test::Methods` module you're going to use needs to be included into each test. Rather than doing this manually, you can use this `:type` option to do the `include` for us, as well as some additional behavior. Let's open a new file at spec/support/api_helpers.rb and put the content from listing 13.2 inside.

Listing 13.2 spec/support/api_helpers.rb

```
module ApiHelper
  include Rack::Test::Methods <co id="ch13_458_1"/>

  def app <co id="ch13_458_2"/>
    Rails.application
  end
end

RSpec.configure do |c|
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  c.include ApiHelper, :type => :api <co id="ch13_458_3"/>
end
```

Here you define a module called `ApiHelper`, which you include into any test marked as an API test with the `:type` option . Inside the module, you use the `Rack::Test::Methods` module, which provides useful methods that you'll see throughout this chapter for making requests to your application, such as the `get` method (not yet shown). You define the `app` method here so that the `Rack::Test::Methods` knows which application to act on. With this done, let's go back to your test.

Inside the `describe` block here underneath this new method you're going to want to create a new user (an admin one at that, because later on you'll need it for the `create` and other actions) who you'll use to make this request to the API. You can create this admin by adding a `let` inside spec/api/v1/projects_spec.rb:

```
describe "/api/v1/projects", :type => :api do
  let!(:admin) { Factory(:admin_user) }
end
```

You'll need to set up Devise to include the `token_authenticatable` module so that you can authenticate API requests from users by using a token they provide with each request. This is so that you will know what projects to show to your users, as well as any other authorization criteria that you need to apply to this user's request. For example, only users with permission to create projects in the application should be able to do so through the API.

To implement the change that you need, go into the `User` model (app/models/user.rb) and change the `devise` call to be this:

```
devise :database_authenticatable, :registerable, :confirmable,
       :recoverable, :rememberable, :trackable, :validatable,
       :token_authenticatable
```

Next, generate a migration to add a field called `authentication_token` to the `users` table, which will be used to store this token. You'll need to add this

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

migration to both the development and test environments. To do this, run these three commands:

```
rails g migration add_token_to_users authentication_token:string
rake db:migrate
rake db:test:prepare
```

The migration generator is smart here and will know to add the `authentication_token` to the `users` table based off the name of the migration and the specified field at the end that you're passing through. The additional argument on the end tells Rails what type of field you'd like this to be.

With the migration created and run, you still need to add a callback to your `User` model, so that tokens are generated for users when they're created, or for when users are updated but don't have a token[7]. To do this, you'll put this line in your `User` model:

Footnote 7 A situation that is unlikely to happen (as you've got no serious users currently), but could potentially happen.

```
before_save :ensure_authentication_token
```

The `before_save` method here is run on a record whenever it is created or updated, as opposed to the `before_create` callback that you saw back in chapter 10, which only calls the specified method upon record creation.

With the callback to create the token in place, let's jump back to your spec and write a test to make a request with this token. Directly underneath the `let(:user)` in spec/api/v1/projects_spec.rb, you'll put the code from listing 13.3.

Listing 13.3 spec/api/v1/projects_spec.rb

```
require "spec_helper"

describe "/api/v1/projects", :type => :api do
  let!(:user) { Factory(:user) }
  let!(:token) { user.authentication_token }
  let!(:project) { Factory(:project) }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  before do
    user.permissions.create!(:action => "view", :thing => project)
  end

  context "projects viewable by this user" do
    let(:url) { "/api/v1/projects" }
    it "json" do <co id="ch13_126_1"/>
      get "#{url}.json", :token => token

      projects_json = Project.all.to_json
      last_response.body.should eql(projects_json)
      last_response.status.should eql(200)

      projects = JSON.parse(last_response.body)

      projects.any? do |p|
        p["name"] == project.name
      end.should be_true
    end
  end
end
```

You're using another `let!` to define a `token` method that, when called, will return the `authentication_token` for the user set up by the previous `let!`. You'll use this later for authenticating requests for your API. The `get` method you use here is provided by `Rack::Test::Methods` and simply makes a GET request with the provided URL. You put the URL in a `let` is because you don't want to repeat the URL too many times if you have multiple tests, and the `let` stops you from repeating yourself in your code.

After the request is done in the test, you ensure that the `last_response.status` returns `200`, which is the HTTP status code for `OK` and means the request was successful. The rest of this spec tests that the data contained within `last_response.body` contains the appropriate data. This `to_json` method will take the attributes for each project returned and turn them into JSON, resulting in an output such as:

```
[
  {
    "created_at":"[timestamp]",
    "id":1,
    "name":"Ticketee",
    "updated_at":"[timestamp]"
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
]
```

This output can then be read with a JSON parser by the receiving user, which is what you do on the line directly after this by using the `JSON.parse` method that is provided by the `json` gem. This method takes JSON string and converts it into a Ruby `Array` or `Hash`. On the final line of your spec, you check that there's anything in this array–anything at all–which returns `true` for `p["project"]["name"] == "Ticketee"`, to make sure that the project you've created really shows up. You need the first key, `project`, as this is how elements are returned in JSON response so that their types can easily be identified. If something does match for the `any?` method, then your test passes.

Let's see what happens when you run `bin/rspec spec/api/v1/projects_spec.rb` now:

```
Failure/Error: get "#{url}.json", :token => token
ActionController::RoutingError:
  uninitialized constant Api
```

You haven't yet defined any controllers for your API, and so this test is going to quite obviously fail. To make it pass, you'll need to define the full constant it will require, `Api::V1::ProjectsController`. This controller will be responsible for serving all the requests for your projects API.

13.1.2 Serving an API

To begin to define controllers for your namespace-within-a-namespace, you'll create a new file at app/controllers/api/v1/base_controller.rb. This file will serve as a base for all controllers within version 1 of your API, providing functionality (eventually) for authenticating and authorizing users, much like the `ApplicationController` currently does. In app/controllers/api/v1/base_controller.rb, you'll define the following:

```
class Api::V1::BaseController < ActionController::Base
  respond_to :json
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Eventually you'll put in the token authentication code into this, but for now you're only trying to get the example to pass. The `respond_to` method here sets up any inheriting controller to respond to `JSON` requests, such as the `ProjectsController` that you're about to create. To make your test pass, you need to return JSON data from the `index` action inside `Api::V1::ProjectsController`, which is much easier than it sounds. You can get the functionality you need from this controller by creating a new file at app/controllers/api/v1/projects_controller.rb and filling it with this content:

```
class Api::V1::ProjectsController < Api::V1::BaseController
  def index
    respond_with(Project.all)
  end
end
```

By inheriting from `Api::V1::BaseController`, the `Api::V1::ProjectsController` will inherit the `respond_to` definition from `Api::V1::BaseController`, as well as any future methods that you define. The `respond_with` method here inside the `index` action will return the a serialized representation of `Project.all` when you make a request to this controller. If you make a JSON request, the data will be serialized into json by calling `to_json` on the result. Rails knows to return JSON data back from any request with the format (that's the bit after the dot in api/v1/projects.json) of JSON. Rails handles all of this internally for us, which is nice of it to do.

Let's find out if this new controller and its only action make the spec pass with `bin/rspec spec/api/v1/projects_spec.rb`:

```
1 example, 0 failures
```

There you have it, your first API route and action are serving up data! Now you're going to need to restrict what this action returns to only the projects that the user can read, but you'll need to first authenticate a user based on their token, which is made easy with Devise.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13.1.3 API Authentication

Our next task is authenticating the user who's making the request in your API. The first step is to do something with the token parameter that gets passed through with your request. A sensible place to check this token would be in `Api::V1::BaseController`, as you want to authenticate for all controllers in the API (although there's only one, for now). For this authentication, you'll find if there's a user with the token passed in by using a `before_filter` like this in app/controllers/api/v1/base_controller.rb:

```
before_filter :authenticate_user

private
  def authenticate_user
    @current_user = User.find_by_authentication_token(params[:token])
  end

  def current_user
    @current_user
  end
```

To check and see if this is working, you'll alter your test in spec/api/v1/projects_spec.rb to generate another project, give the user read access to only that project, and check that the response from the API only contains that project. To do this, you'll add a new `before` to the "projects viewable by this user" `context` inside the spec, using the code from Listing 13.4

Listing 13.4 spec/api/v1/projects_spec.rb

```
context "projects viewable by this user" do

  before do
    Factory(:project, :name => "Access Denied")
  end

  ...
end
```

In the `before` block you create one project that the user should not have access to read. Inside the test itself, you're still using the `for` scope on the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`Project` class to get only the projects that the specified user has access to. Let's add a couple of more lines to your example now to check that this user cannot see the "Access Denied" project:

```
  projects.any? do |p|
    p["name"] == "Access Denied"
  end.should be_false
```

When you run this spec with `bin/rspec spec/api/v1/projects_spec.rb` you'll see that when the test is expecting `false`, it's actually getting `true`:

```
Failure/Error: projects.any? do |p|
 expected: false value
      got: true
```

This test is failing because it can still see the 'Access Denied' project within the returned results, even though we say it shouldn't.

To make this test pass, you're going to need to stop returning *all* the projects in the `index` action of `Api::V1::ProjectsController` and only return the projects that this user should be able to see. Let's now open app/controllers/api/v1/projects_controller.rb and change the `index` action to use the `for` method and pass in the `current_user`, rather than the `all` method:

```
def index
  respond_with(Project.for(current_user).all)
end
```

This will now return only the list of projects that the user should be able to see, which should be enough to get this test passing. You can find out with another quick run of `bin/rspec spec/api/v1/projects_spec.rb`:

```
Failure/Error: last_response.body.should eql(projects_json)
  expected: [two projects]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  got: [one project]
```

Oops, not quite! In the spec you're defininining the expected JSON result of this request as the variable `projects_json`, like this:

```
projects_json = Project.all.to_json
```

This is not correct because the JSON coming back from the controller is now defined like this inside the `index` action of `Api::V1::ProjectsController`:

```
respond_with(Project.for(current_user).all)
```

In the controller, the result is scoped using the `for` scope on the `Project` model, but in the spec it is not. To fix this, change the line in the spec that defines the `projects_json` to this:

```
projects_json = Project.for(user).all.to_json
```

When you re-run the test with `bin/rspec spec/api/v1/projects_spec.rb` it will now pass:

```
1 example, 0 failures
```

Great, now you've got your API finding a user based on the token that you've gathered in your spec. One thing you haven't tested for yet is: what happens when an invalid (or no) token is given? Well, you should return an error when that happens. This is the final change you'll be making before you make a commit, as it's been a little too long since you've last done that[8].

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 8 As a reminder: You should commit after every "safe" point so that if you stuff something up (it happens!) you won't have to rollback as much.

13.1.4 Error reporting

Something will inevitably go wrong in your application, and when that happens you're going to want to provide useful error messages to your users. One of the things that could go wrong in your API is that the user uses an invalid token to authenticate against your API. When a user makes a request with an invalid token, you should inform them of their mistake, which you can do by returning JSON that looks like this:

```
{ error: "Token is invalid." }
```

To test this behavior, you're going to make a request without a token and then fix up your `projects_spec.rb` test to pass in a token. You'll write your first test now in a new file at spec/api/v1/authentication_spec.rb, which will be filled with the content from Listing 13.5

Listing 13.5 No token test

```
require "spec_helper"

describe "API errors", :type => :api do

  it "making a request with no token" do
    get "/api/v1/projects.json", :token => ""
    error = { :error => "Token is invalid." }
    last_response.body.should eql(error.to_json)
  end

end
```

You're using `Rack::Test::Methods` in the spec again, and you've set up the token to be a blank string so `get` will pass this through as the token. Let's run this spec to make sure it's failing first with `bin/rspec spec/api/v1/authentication_spec.rb`:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Failure/Error: get "/api/v1/projects.json", :token => ""
NoMethodError:
  undefined method `admin?' for nil:NilClass
```

Yup, definitely looks like it's failing. Line 13 of app/models/project.rb attempts to call `admin?` on the `User` object passed in to the `for` method. If you attempt to make a request without a valid token, the call to `User.find_by_authentication_token` will return `nil`, resulting in the error you see here. You should check if the user has been found, and if not then you'll show the error. To make your `authenticate_user` method do this in app/controllers/api/v1/base_controller.rb, you'll change it to what is shown in Listing 13.6.

Listing 13.6 authenticate_user method in Api::V1::BaseController

```
def authenticate_user
  @current_user = User.find_by_authentication_token(params[:token])
  unless @current_user
    respond_with({:error => "Token is invalid." })
  end
end
```

If a user doesn't have a token that matches up with a user within the application then the `find_by_authentication_token` call will return `nil` and the response's body will be set to the JSON-form of `{ :error => "Token is invalid" }` and the user will see that response. Does this work? Let's find out with a quick run of `bin/rspec spec/api/v1/authentication_spec.rb`:

```
1 example, 0 failures
```

Booyah, it works! How about `bin/rspec spec/api/v1/projects_spec.rb` too?

```
1 example, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

All green there too, and so it's definitely time to do a commit now. You should run the customary checks before you commit by running `rake spec`:

```
89 examples, 0 failures
```

Great! Everything's still green. You'll commit and push the changes that you've made:

```
git add .
git commit -m "Implement token-based authentication API foundations"
git push
```

You've begun to implement the API now, and you've got the `/api/v1/projects` URL returning a list of the projects that a user can see. To check what user this is, you've implemented a basic token authentication using functionality built in to Devise.

There's still a little ways to go before you're done with the API. For starters, this API only serves JSON requests, and some people who use it may wish for the data to be returned in XML. You've also only got the one action in your controller, and you need to implement a way for people to create, update and delete projects through the API. Before you do that, you'll add in support for XML. This is incredibly easy to implement, as you'll soon see.

13.1.5 Serving XML

So far, you've been using the `respond_with` and `respond_to` methods to serve JSON responses. You can serve XML using these same methods while continuing to serve JSON. It's very, very easy. First of all, you're going to want to create a test to make sure that your new XML data is being returned correctly. You'll place this test in the "index" `context` for "projects viewable by this user" in spec/api/v1/projects_spec.rb using the code from Listing 13.7.

Listing 13.7 spec/api/v1/projects_spec.rb

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
it "XML" do
  get "#{url}.xml", :token => token
  last_response.body.should eql(Project.for(user).to_xml)
  projects = Nokogiri::XML(last_response.body)
  projects.css("project name").text.should eql(project.name)
end
```

In this spec you use the `nokogiri` gem[9] to parse the XML. Then you use the `css` method to find an element called `name` inside another called `project`, and then check to see if its `text` is equal to the name of your project, which it should be if everything works out fine. When you run `bin/rspec spec/api/v1/projects_spec.rb` this spec will fail:

Footnote 9 Nokogiri is a gem used to parse XML and HTML documents, written by Aaron Patterson, one of the Rails core contributors. A bit of trivia: Aaron is also a Ruby core committer too.

```
expected: "[projects XML]
got: " "

Diff:
 @@ -1,10 +1,2 @@
 -<?xml version="1.0" encoding="UTF-8"?>
 -<projects type="array">
 -  <project>
 -    <created-at type="datetime">[timestamp]</created-at>
 -    <id type="integer">1</id>
 -    <name>Example project</name>
 -    <updated-at type="datetime">[timestamp]</updated-at>
 -  </project>
 -</projects>
 +]]>
 ...
```

The diff here shows that the expected XML is nowhere to be found in the response, and instead you're getting back a final line of absolutely nothing. This is because your `Api::V1::BaseController` doesn't yet respond to XML requests. So now with a failing test you can go right ahead and change this controller to fix it. To make your API serve XML requests, you'll change this line in app/controllers/api/v1/base_controller.rb:

```
respond_to :json
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To this:

```
respond_to :json, :xml
```

This simple little change will now make your spec pass, which you can see by running `bin/rspec spec/api/v1/projects_spec.rb`

```
2 examples, 0 failures
```

Apologies if something harder was expected, but it really is this simple in Rails. You've only changed the API controller and spec, and it's all contained in itself, but even so it's still a good habit to run all the features and specs to make sure everything is fine.

```
90 examples, 0 failures
```

Green is good. Now you'll commit this change:

```
git add .
git commit -m "Support XML & JSON with /api/v1/projects"
git push
```

Now that you've got your first action of your API responding to both XML and JSON, why don't you make some more actions, like the `create` action for creating projects in Ticketee.

13.1.6 Creating projects

In this section, you're going to implement a new API action that will allow you to create projects. The route for this action was provided by this code in config/routes.rb:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
namespace :api do
  namespace :v1 do
    resources :projects
  end
end
```

You only need to implement the `create` action in your controller, which makes it all quite simple. When you make a request to this action and it passes validations, you will return the XML or JSON representation of the project that was created, along with a `201 Created` HTTP status code, which indicates that a new resource has been created[10]. If the creation of the project fails any validation, then Rails will return a 422 Unprocessable Entity HTTP Status Code[11], which will indicate that there are errors with the request. The body returned by this failing will contain those errors and look something like this:

Footnote 10 This isn't unique to Rails, but is rather part of RFC 2616: http://tools.ietf.org/html/rfc2616#section-10.2.2

Footnote 11 As described in RFC 4918, Section 11.2: http://tools.ietf.org/html/rfc4918#section-11.2

```
{"name":"can't be blank"}
```

It's then up to the people receiving the status back from the API to choose how to display this information.

To make this request to the `create` action, you need to make a `POST` request to the `/api/v1/projects` path, and to do this there's the `post` method provided by `Rack::Test::Methods` that you can use. You'll open up spec/api/v1/projects_spec.rb now and add in another `context` block under the first for checking that creating a project through JSON works, as shown in Listing 13.8.

Listing 13.8 Creating a project test

```
context "creating a project" do

  let(:url) { "/api/v1/projects" }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders. http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  it "successful JSON" do
    post "#{url}.json", :token => token,
                        :project => {
                          :name => "Inspector"
                        }

    project = Project.find_by_name!("Inspector")
    route = "/api/v1/projects/#{project.id}"

    last_response.status.should eql(201)
    last_response.headers["Location"].should eql(route)
    <co id="ch13_431_1"/>
    last_response.body.should eql(project.to_json)
  end
end
```

In the normal `create` action for the normal `ProjectsController` in your application, you're restricting creation of projects to admin users. You'll do this for this API action in a bit–you're only trying to get the most basic example going first. Here you again set up the `url` in a `let` so that you can re-use it for the other tests you'll implement later.

You begin your test by making a POST request using `post` (provided by the `Rack::Test::Methods` module, just like `get`) passing through the parameters of a project as the second argument in this method. Then you check that the status of the `last_response` is set to 201, which is the correct reply if the resource was created successfully. Next, check that the `Location` in the header is set to the correct API route, which would be something such as `http://example.com/api/v1/projects`. You'll find out why when you go to implement this action. On the final line of the spec, check that the `last_response.body` contains the JSON representation of the project that should have been created.

When you run `bin/rspec spec/api/v1/projects_spec.rb` this test will fail, because you've yet to implement the `create` action for its controller:

```
Failure/Error: post "#{url}.json", :token = token,
  The action 'create' could not be found for Api::V1::ProjectsController
```

You'll need to implement this action to make the spec pass. Let's open up app/controllers/api/v1/projects_controller.rb and add this action underneath your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`index` action, using the code shown in Listing 13.9.

Listing 13.9 Api::V1::ProjectsController, create method

```
def create
  project = Project.new(params[:project])
  if project.save
    respond_with(project, :location => api_v1_project_path(project))
  <co id="ch13_485_1"/>
  else
    respond_with(project)
  end
end
```

By attempting to save a project, your application will run the validations set up in the `Project` model. If this succeeds, then the status that will be returned will be `201` and you'll get back the proper representation (either JSON or XML) of the new project. On the final line of this action, you manually set the `Location` key in the headers by passing through the `:location` option so that it points to the correct URL of something such as `http://example.com/api/v1/projects/1`, rather than the Rails default of `http://example.com/projects/1`. People who are using your API can then store this location and reference it later on when they wish to retrieve information about the project. The URL that Rails defaults to goes to the user-facing version of this resource (/projects/1.json), which is incorrect.

If the project isn't valid (i.e. if the `save` method returns `false`), then you simply let Rails return a response that will contain the errors of the project, without having a custom location set.

This should be all that you need in order to get your spec to pass, so let's see what happens when you run `bin/rspec spec/api/v1/projects_spec.rb`.

```
3 examples, 0 failures
```

Great! Now you need to write a test to check that when you attempt to pass through a project with no name we're given a 422 status code and an error along

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

with it, indicating that the project wasn't created due to those errors. Directly underneath the previous test in spec/api/v1/projects_spec.rb you'll add this test shown in Listing 13.10.

Listing 13.10 Unsuccessful creation example

```
it "unsuccessful JSON" do
  post "#{url}.json", :token => token,
                      :project => {}
  last_response.status.should eql(422)
  errors = {"errors" => {
              "name" => ["can't be blank"]
            }}.to_json
  last_response.body.should eql(errors)
end
```

Naughty us, writing the test after the code is already there, but you can get away with it once. Let's run the spec and see how it goes now:

```
4 examples, 0 failures
```

Great success! With this URL working for valid and non-valid projects appropriately, you are now providing a way for your users to create a project through the API, and so it's time to make a commit:

```
git add .
git commit -m "Add API to create projects"
git push
```

Our next task is to restrict this action to only the admins of your application, like in the real `ProjectsController` controller. You want to limit the number of people who can change the projects to a select few who know what they're doing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13.1.7 Restricting access to only admins

In app/controllers/projects_controller.rb you've got this line, which restricts some actions to only admins:

```
before_filter :authorize_admin!, :except => [:index, :show]
```

As it says on the line, every action other than `index` or `show` has this filter run before it. This filter is defined in app/controllers/application_controller.rb like this:

```
def authorize_admin!
  authenticate_user!
  unless current_user.admin?
    flash[:alert] = "You must be an admin to do that."
    redirect_to root_path
  end
end
```

You're not going to be able to use this same `before_filter` for your API because the API doesn't return flash messages. You have to return errors in a lovely little JSON or XML format. This particular error, for example, is "You must be an admin." Also, redirection doesn't make sense here, because it wouldn't tell users *why* they were redirected. Therefore, you'll implement a different `authorize_admin!` method in your `Api::V1::BaseController` instead. You'll take the time, however, to write a test to check for this error occurring. Let's open a new file at spec/api/v1/project_errors_spec.rb and add a test that if you attempt to make a POST request to `api/v1/projects` using a token for a user who's *not* an admin, you get an error. Use the code from Listing 13.11.

Listing 13.11 spec/api/v1/project_errors_spec.rb

```
require "spec_helper"

describe "Project API errors", :type => :api do
  context "standard users" do
    let(:user) { Factory(:user) }

    it "cannot create projects" do
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      post "/api/v1/projects.json",
        :token => user.authentication_token,
        :project => {
          :name => "Ticketee"
        }
      error = { :error => "You must be an admin to do that." }
      last_response.body.should eql(error.to_json)
      last_response.status.should eql(401)
      Project.find_by_name("Ticketee").should be_nil
    end
  end
end
```

With this spec, you test that a normal user who's using a valid authenticity token cannot create a project through the API because they're not an admin. Instead, the API should return a response of "You must be an admin to do that." This response should have a `code` of 401, indicating an "Unauthorized" response. When you run this spec using `bin/rspec spec/api/v1/project_errors_spec.rb` it will not return the error like you expect:

```
  expected "{\"error\":\"You must be an admin to do that.\"}"
       got "{[project hash]}"
```

To make this error happen, you'll go into app/controllers/api/v1/base_controller.rb and underneath your `authenticate_user` method you'll add the `authorize_admin!` method shown in Listing 13.12.

Listing 13.12 authorize_admin method inside Api::V1::BaseController

```
def authorize_admin!
  if !@current_user.admin?
    error = { :error => "You must be an admin to do that." }
    warden.custom_failure! <co id="ch13_548_1"/>
    render params[:format].to_sym => error, :status => 401
    <co id="ch13_548_2"/>
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Here you use `warden.custom_failure!` to inform Warden (the Rack backend to Devise) that you're going to raise a custom 401 response. Without this, Devise would instead take over from this 401 response, showing a "You must be signed in to continue" error message.

You also use the `render` method in a unique manner here. You call it and pass in a hash with the key being a symbolized version of the format (in this case, `json`) and the value being the hash that contains the error. By calling `render` in this way, Rails will convert the hash to JSON, or if it were an XML request, to XML. The reason for doing it this way rather than using `respond_with` is because `respond_with` will attempt to do some weird behavior and doesn't work for POST requests, so you must work around that little issue.

By specifying the `status` option to the `render` here, your response's status code will be set to that particular status, which will let the code used to connect to the API know that the user is unauthorized to perform the specific request.

Now all you need to do is add this as a `before_filter` into app/controllers/api/v1/projects_controller.rb using this line:

```
before_filter :authorize_admin!, :except => [:index, :show]
```

With this line now in the `ProjectsController`, any request to any action that's not the `index` or `show` action will have the admin check run before it. If a user doesn't meet this criteria, then you will return the "You must be an admin to do that" error. These pieces of code should now be enough to get your test running, so let's find out with `bin/rspec spec/api/v1/project_errors_spec.rb`:

```
1 example, 0 failures
```

Now when people who aren't admins try to create a project, they will see the "You must be an admin to do that" error message returned from the API and the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

project won't be created. You can see this happen when you run the final `context` block of spec/api/v1/projects_spec.rb when you run `bin/rspec spec/api/v1/projects_spec.rb:45`

```
Failure/Error: project = Project.find_by_name!("Inspector")
ActiveRecord::RecordNotFound:
  Couldn't find Project with name = Inspector
```

Because this is now the case, you'll need to set up the user in the projects API examples to be an admin when they attempt to create a project, which you can do by putting this `before` after the beginning of the "creating a project" context block:

```
  before do
    user.admin = true
    user.save
  end
```

When you run `bin/rspec spec/api/v1/projects_spec.rb` all the examples will be passing:

```
4 examples, 0 failures
```

With this new authorization added you will make a commit, but before that you'll run a customary check to make sure everything is still alright by running `rake spec`. You should see this output:

```
93 examples, 0 failures
```

Great, so let's go ahead and commit this then:

```
git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git commit -m "Only admins are able to create projects through API"
git push
```

In the response for your `create` action, the headers point to a location (you customized) of a project, something such as http://example.com/api/v1/projects/1. Currently, this URL doesn't *go* anywhere because it needs the `show` action to exist. You should probably get on to that.

13.1.8 A single project

You've got a link (http://example.com/api/v1/projects/1) provided by your `create` action that doesn't go anywhere if people try to access it. Soon this URL will show a particular project's attributes through the `show` action in your `Api::V1::ProjectsController`. Within those attributes, you'll also show a `last_ticket` element, which will contain the attributes for the most recently updated ticket. To do this, you'll be using another option of `respond_with`, the `:methods` option. Using this option will change the output of each project resource in your JSON API to something like this:

```
{
  "project": {
    "created_at": "[timestamp]",
    "id": 1,
    "name": "Ticketee",
    "updated_at": "[timestamp]",
    "last_ticket": {
      "ticket": {
        "asset_updated_at": null,
        "created_at": "[timestamp]",
        "description": "A ticket, nothing more.",
        "id": 1,
        "project_id": 1,
        "state_id": null,
        "title": "A ticket, nothing more.",
        "updated_at": "[timestamp]",
        "user_id": 2
      }
    }
  }
}
```

Using the `last_ticket` method, people using the API will be able to discover when the last activity was on the project. You could add other fields such

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

as the comments too if you wished, but this example is kept simple for quality and training purposes.

To get started with this `show` action, you'll write a test in spec/api/v1/projects_spec.rb for it as shown in the following listing.

Listing 13.13 show action test for spec/api/v1/projects_spec.rb

```
context "show" do
  let(:url) { "/api/v1/projects/#{@project.id}"}

  before do
    Factory(:ticket, :project => @project)
  end

  it "JSON" do
    get "#{url}.json", :token => token
    project_json = @project.to_json(:methods => "last_ticket")
    last_response.body.should eql(project_json)
    last_response.status.should eql(200)

    project_response = JSON.parse(last_response.body)

    ticket_title = project_response["last_ticket"]["title"]
    ticket_title.should_not be_blank
  end
end
```

You're using the `project` method that was set up by the "projects viewable by this user" `context` block earlier to generate the URL to a `Project` resource, as well as using it to create a new ticket for this project so that `last_ticket` returns something of value. You take this URL and do a JSON request on it, and you expect to get back a JSON representation of the object with the `last_ticket` method being called and also returning data. Then you check that the response's status should be 200, indicating a good request, and finally you check that the last ticket title isn't blank.

To make this test pass you'll open app/controllers/api/v1/projects_controller.rb and add in the `show` action, as shown in Listing 13.14.

Listing 13.14 app/controllers/api/v1/projects_controller.rb

```
def show
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  @project = Project.find(params[:id])
  respond_with(@project, :methods => "last_ticket")
end
```

In this action, you find the `Project` based off the `params[:id]` value and then `respond_with` this object, asking it to call the `last_ticket` method. If this method is undefined (like it is right now), then the method will not be called at all. When you run this test with `bin/rspec spec/api/v1/projects_spec.rb`, you'll see this error:

```
Failure/Error: ticket_title = last_response ...
  You have a nil object when you didn't expect it!
  You might have expected an instance of Array.
  The error occurred while evaluating nil.[]
```

The error occurs because you're attempting to call the `[]` method on something that is `nil`, and it's really likely that the something is the `last_ticket` hash contained within the project's response, which doesn't exist yet because the method is not defined on `Project` instances. If a method is not defined but is specified as a method to be provided through a `respond_with` method, then that undefined method will be ignored.

To define this method, open app/models/project.rb and add this method inside the class:

```
def last_ticket
  tickets.last
end
```

Why are you doing it this way? Well, `respond_with` doesn't let you chain methods, and so you'll work around this by defining a method that calls the chain in your model. When you run `bin/rspec spec/api/v1/projects_spec.rb`, this test will pass because the `last_ticket` method is now defined:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
1 example, 0 failures
```

Great! Now the `show` action is responding with data similar to this:

```
{
  "created_at": "[timestamp]",
  "id": 1,
  "name": "Ticketee",
  "updated_at": "[timestamp]",
  "ticket": {
    "asset_updated_at": null,
    "created_at": "[timestamp]",
    "description": "A ticket, nothing more.",
    "id": 1,
    "project_id": 1,
    "state_id": null,
    "title": "A ticket, nothing more.",
    "updated_at": "[timestamp]",
    "user_id": 2
  }
  }
}
```

How goes the rest of your application? Let's find out with a quick run of `rake spec`:

```
94 examples, 0 failures
```

Ok, that's good to see, time to make a commit:

```
git add .
git commit -m "Add API action for a single project with last ticket"
git push
```

Back in the main part of the application, you've got permissions on users that restrict which projects they can see. Currently in the API there is no such restriction, and so you need to add one to bring it in line with how the application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

behaves.

13.1.9 No project for you!

Currently, any user can see the details of any project through the API. The main application enforces the rule that users without permission to view a project are not able to do so. To enforce this rule in your API as well, you use the `find_project` method:

```
def find_project
  @project = Project.for(current_user).find(params[:id])
rescue ActiveRecord::RecordNotFound
  flash[:alert] = "The project you were looking for could not be found."
  redirect_to projects_path
end
```

Here you use the `for` method, which will return a scope for all projects viewable by the current user. By calling `find` on this scope, if the user doesn't have access to the project then an `ActiveRecord::RecordNotFound` exception will be raised. You then `rescue` this exception and lie to the user, telling them the project is mysteriously gone[12]. Much like the `authorize_admin!` method you ported over before, you can't set the flash notice or redirect here. Instead, you're going to have to present an API error like you did before[13].

Footnote 12 It's not really.

Footnote 13 While this may seem like repetition (which it is), it's part of the project's API and will help you understand the concepts better. Practice, practice, practice! It makes perfect prefects.

To test this new `before_filter :authorize_user`, you'll write a new test in spec/api/v1/project_errors_spec.rb where a user without permission on a project attempts to view it, only to be rebuked by the server with an error. This test should be placed inside the "standard users" `context` block, and is shown in Listing 13.15

Listing 13.15 cannot view projects test

```
it "cannot view projects they do not have access to" do
  project = Factory(:project)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  get "/api/v1/projects/#{project.id}.json",
    :token => user.authentication_token
  error = { :error => "The project you were looking for" +
                      " could not be found." }
  last_response.status.should eql(404)
  last_response.body.should eql(error.to_json)
end
```

When the user attempts to go to the show page, they should receive the error informing that the project has run away (or doesn't exist). The status code for this response should be 404, indicating the resource the user was looking for is not found. To make this work, you'll remove this line from the `show` action in app/controllers/api/v1/projects_controller.rb

```
project = Project.find(params[:id])
```

Then you'll put this line under the `authorize_admin!` filter inside this controller's class:

```
before_filter :find_project, :only => [:show]
```

Next, you need to add the `find_project` after the `show` action as a private method, as shown in Listing 13.16.

Listing 13.16 Api::V1::ProjectsController, find_project method

```
private

def find_project
  @project = Project.for(current_user).find(params[:id])
  rescue ActiveRecord::RecordNotFound
    error = { :error => "The project you were looking for " +
                        "could not be found."}
    respond_with(error, :status => 404)
end
```

Here you respond with the error message and set the status to 404 to tell the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

user that the project doesn't exist. When you run `bin/rspec spec/api/v1/project_errors_spec.rb` your spec will pass:

```
2 examples, 0 failures
```

You're now restricting the projects that a user can access to only the ones they have permission to view. If the API user doesn't have the permission, you'll deny all knowledge of the project and return a 404 status code. It's quite grand how this is possible in such few lines of easy-to-understand code.

You'll run all the specs now to make sure everything's rosy with `rake spec`. You should see that it's all green:

```
95 examples, 0 failures
```

Edging ever closer to 100 examples! A commit you shall make:

```
git add .
git commit -m "Restricting projects API show to only users
               who have permission to view a project"
git push
```

Currently you've got the `index`, `show`, and `create` actions implemented for your controller. What's missing? Well, you could say the `new`, `edit`, `update`, and `destroy` actions are, but you don't need the `new` and `edit` actions, because this should be handled on the client side of the API, not the server. It is the client's duty to present the new and edit dialogs to the user. Therefore, you only need to implement the `update` and `destroy` methods and then you're done with this API. So close!

13.1.10 Updating a project

To update a project in the API, people will need to make a POST request to the `/api/v1/projects/:id` URL with the project's new information contained in a `params[:project]` hash. Simple, really.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To test that this action works correctly, you'll add yet another spec to spec/api/v1/projects_spec.rb using the code from Listing 13.17

Listing 13.17 updating a project test

```
context "updating a project" do
  before do
    user.admin = true
    user.save
  end

  let(:url) { "/api/v1/projects/#{@project.id}" }
  it "successful JSON" do
    put "#{url}.json", :token => token, <co id="ch13_731_1"/>
                       :project => {
                          :name => "Not Ticketee"
                        }

    last_response.status.should eql(204)
    last_response.body.should eql("")

    project.reload
    project.name.should eql("Not Ticketee")
  end
end
```

At the top of this new `context` block, you've defined that the user is an admin again. You could wrap the `create` and `update` tests within another context that sets this flag too, but you'll do it this way for now.

You need to make a `put` request to this action for Rails to accept it, and you can do that by using the `put` method . Along with this request, you send the `token` and `project` parameters. The `project` parameter contains a new name for this project. Because it's a valid (non-blank) name, the response's status code will be 204 indicating that the request was processed and no content is to be returned. This request doesn't return an updated object, because what's the point? The client should be aware of the updates which have occurred, given it triggered them!

At the end of this spec, you use the `reload` method to find this object again from the database. This is because the object that the spec is working with will be a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

completely different Ruby object from the one in the `update` action in the controller. By calling `reload`, Rails will fetch the data for this object again from the database and update the object in the process.

To begin with writing this action in `Api::V1::ProjectsController`, you're going to need to first modify the `before_filter :find_project` line to include the `update` action, changing it from this:

```
before_filter :find_project, :only => [:show]
```

To this:

```
before_filter :find_project, :only => [:show, :update]
```

Now in the `update` action you'll have a project that you can work with because this `before_filter` will find it for us. Next, you'll write this action into the controller using the code from Listing 13.18

Listing 13.18 update action of Api::V1::ProjectsController

```
def update
  @project.update_attributes(params[:project])
  respond_with(@project)
end
```

Well isn't this quite a difference from your standard `update` actions? You only need to call `update_attributes` here, which will save the object and return a valid object in the format that you've asked for. If this object fails validation, the status code returned by `respond_with` will be 422, which represents an "Unprocessable Entity," and the body would contain only the validation errors that occurred. If the object is valid, `respond_with` will return a 200 status code, but an empty response. This is because the client should be aware of what changes it has made to the object, and so there's no need to send back the object.

So which is it? Does the `update` action work and return the 200 status code

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you want, or does it break? It's easy to find out: `bin/rspec spec/api/v1/projects_spec.rb`.

```
5 examples, 0 failures
```

All working, good stuff. You've now got a check that the `update` action responds correctly when a valid object is given, but what if invalid parameters are given instead? Well, the action should return that 422 response mentioned earlier. Although this is testing the already extensively tested[14] Rails behavior, you're making sure that this action always does what you think it should. No misbehaving allowed! You'll quickly whip up a spec for this, placing it right underneath the previous example, "successful JSON" that you wrote in spec/api/v1/projects_spec.rb. The code for it is shown in Listing 13.19

Footnote 14 It's tested within Rails itself

Listing 13.19 Unsuccessful updating test

```
it "unsuccessful JSON" do
  put "#{url}.json", :token => token,
                     :project => {
                       :name => ""
                      }
  last_response.status.should eql(422)
  errors = { :errors => { :name => ["can't be blank"] } }
  last_response.body.should eql(errors.to_json)
end
```

In this example, you attempt to set the project's name to a blank string, which should result in the 422 error you want to see. After you `reload` the project, the name should be the same. You should then get the 422 error as the response.

A quick run of `bin/rspec spec/api/v1/projects_spec.rb` should let you know if this is working:

```
7 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Indeed it is! Is everything else working? A run of `rake spec` will let you know:

```
97 examples, 0 failures
```

"97 examples, 0 failures" is exactly what you like to see. That means that your `update` action is all wrapped up, and now it's time for a commit:

```
git add .
git commit -m "Implement update action for projects API"
git push
```

You've now got 3/4ths of the CRUD of this API. You're able to create, read and update project resources. With updating, clients authorized with an admin's token can send through updates to the project resource, which will update the information in the application. The one remaining action you've got to implement is the `destroy` action, for making projects go bye-bye. You're almost home!

13.1.11 Exterminate!

You need to create the `destroy` action, which allows admins of Ticketee to delete projects through the API. To do this, API clients need to make a DELETE request to `/api/v1/projects/1.json` or `/api/v1/projects/1.xml`. Upon making this request, the specified project will be deleted–gone forever, exterminated!

You'll write the final example in the spec/api/v1/projects_spec.rb to make sure that people are able to delete projects using this route. You'll use the code from Listing 13.20 to do this.

Listing 13.20 deleting a project test

```
context "deleting a project" do
  before do
    user.admin = true
    user.save
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  let(:url) { "/api/v1/projects/#{project.id}" }
  it "JSON" do
    delete "#{url}.json", :token => token
    last_response.status.should eql(204)
  end
end
```

In this test you make a `DELETE` request as an admin to the `/api/v1/projects/1` route. Once this request is over, the HTTP status returned from it will be `204`, indicating that there's no content for this reply and that the deletion was a success. This is the same status code that was returned when you made a successful update request.

When you run `bin/rspec spec/api/v1/projects_spec.rb`, this spec will fail because the `destroy` action doesn't exist:

```
Failure/Error: delete "#{url}.json", :token => token
The action 'destroy' could not be found for Api::V1::ProjectsController
```

You need to add the `destroy` action to the `Api::V1::ProjectsController`, which you can do with this code:

```
def destroy
  @project.destroy
  respond_with(@project)
end
```

The `respond_with` here will respond with a 200 status code and an empty JSON or XML response, which indicates the object was successfully destroyed. But where does `@project` come from? Our `before_filter` should set this up, but it doesn't right now. Let's fix it by changing it from this:

```
before_filter :find_project, :only => [:show,
                                       :update]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To this:

```
before_filter :find_project, :only => [:show,
                                       :update,
                                       :destroy]
```

When you run `bin/rspec spec/api/v1/projects_spec.rb`, does it pass?

```
8 examples, 0 failures
```

It does, because you're great at what you do! That's the final piece of the projects API, and now people are able to create, read, update and delete projects through it. The rest of your specs probably pass because you didn't change anything outside the scope, but it's still good to do a check. Run `rake spec` now. For this command, you should see this output:

```
98 examples, 0 failures
```

All systems are go, so let's make a commit at this lovely point in time where everything is beautiful:

```
git add .
git commit -m "Projects can now be deleted through the API"
```

The entire projects API is now complete. What you've got at the moment is a solid base for version 1 of Ticketee's projects API. You'll now see how you can begin creating the nested API for tickets on a project.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13.2 Beginning the Tickets API

In this section you'll begin to create an API for tickets on a project. You're only going to be creating the part of the API to list tickets for now, as the remaining parts are similar in style to what you saw with the projects API. This section will give you a taste of how to work with nested resources within the context of an API.

The first part you're going to need for this API is two tests: one to make sure you can get XML results back from this API and another for JSON results. You're going to put these new tests in a new file at spec/api/v1/tickets_spec.rb, beginning with the setup required for both of these tests shown in Listing 13.21

Listing 13.21 beginning of spec/api/v1/tickets_spec.rb

```
require 'spec_helper'

describe "/api/v1/tickets", :type => :api do
  let!(:project) { Factory(:project, :name => "Ticketee") }
  let!(:user) { Factory(:user) }

  before do
    user.permissions.create!(:action => "view",
                             :thing => project)
  end

  let(:token) { user.authentication_token }
```

With this setup, you can begin the `context` for your `index` action and then in a `before` block, create 5 tickets for the project by using these lines after the `let!(:project)` line:

```
context "index" do
  before do
    5.times do
      Factory(:ticket, :project => project, :user => user)
    end
  end
```

Finally, you can write the XML and JSON tests by placing the code shown in Listing 13.22 inside the `context "index"` block you have written.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 13.22 spec/api/v1/projects_spec.rb

```
context "index" do
  ...

  let(:url) { "/api/v1/projects/#{project.id}/tickets" }

  it "XML" do
    get "#{url}.xml", :token => token
    last_response.body.should eql(project.tickets.to_xml)
  end

  it "JSON" do
    get "#{url}.json", :token => token
    last_response.body.should eql(project.tickets.to_json)
  end
end
```

You've defined the `let(:url)` here to point to the nested route for tickets of a given project. This URL is currently undefined and so when you run this test with `bin/rspec spec/api/v1/tickets_spec.rb`, you'll get told that the route you're requesting doesn't exist:

```
Failure/Error: get "#{url}.json", :token => token
     ActionController::RoutingError:
       No route matches [GET] "/api/v1/projects/1/tickets.json"
```

You can define this route easily inside config/routes.rb by changing these lines:

```
namespace :api do
  namespace :v1 do
    resources :projects
  end
end
```

To this:

```
namespace :api do
  namespace :v1 do
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    resources :projects do
      resources :tickets
    end
  end
end
```

Now you've got the `tickets` resource nested within projects for your API again. When you re-run this spec you'll be told this:

```
uninitialized constant Api::V1::TicketsController
```

You can create this controller by creating a new file at app/controllers/api/v1/tickets_controller.rb. This controller needs to first of all respond to both JSON and XML, and find the project for each request using a `before_filter`. You can begin to define this controller using the code from Listing 13.23

Listing 13.23 app/controllers/api/v1/tickets_controller.rb

```
class Api::V1::TicketsController < Api::V1::BaseController

  before_filter :find_project

  private
    def find_project
      @project = Project.for(current_user).find(params[:project_id])
      rescue ActiveRecord::RecordNotFound
        error = { :error => "The project you were looking for" +
                            " could not be found."}
        respond_with(error, :status => 404)
    end
end
```

In the beginning, you set up the controller to inherit from `Api::V1::BaseController` so that it inherits the basic behavior of your API, providing the controller with the `current_user` method which it uses

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

inside the `find_project` method. Then you define a `before_filter :find_project` that will find a project, providing that the user is able to access it. If the user cannot access it, then you `respond_with` a 404 error.

Underneath the `before_filter` in this controller, you need to define the `index` action to return a list of tickets for your project. You can do that with the code shown in Listing 13.24

Listing 13.24 app/controllers/api/v1/tickets_controller.rb

```
def index
  respond_with(@project.tickets)
end
```

That *feels* like you're getting too much for free, doesn't it? Rails is handling a lot of the actions here for us. When you run `bin/rspec spec/api/v1/tickets_spec.rb` specs again, your tests will now pass because you've got the controller defined correctly:

```
2 examples, 0 failures
```

This is a great start to generating a tickets API, and now with the skills you've learned a little earlier in this chapter you should be able to bash out the rest with little effort. Rather than covering that old ground again, it'll be left as an exercise to the reader.

Let's run all the tests with `rake spec` to make sure you didn't break anything:

```
100 examples, 0 failures
```

Nope, nothing broken there which is awesome to see. Oh and hey look at that, 100 examples! Nice work! Time for a commit:

```
git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git commit -m "Add beginnings of the V1 Tickets API"
git push
```

You should probably limit the number of tickets that are sent back through the API or, even better, cache the result. You'll see ways to do both of these things in chapter 16, and then you can apply them to the API when you feel it's right. For now, it would be fine for a project with a small amount of tickets, but if a project grew to something say, the size of the Rails project[15] then it would be problematic because your application would have to instantiate thousands of new `Ticket` objects per-request. That's no good.

Footnote 15 7,500 tickets, as of this writing

Now that you're versed in the Ways of the API, you can tackle potential problems with it. One of the potential problems with this API is that you'll have too many users accessing it all at once, which may cause performance problems with the application. To prevent, this you'll implement the rate of requests people can make to your server.

13.3 Rate limiting

When a server receives too many requests, it can seem unresponsive. This is simply because it is too busy serving existing requests to serve the hoard of incoming requests. This can happen when an application provides an API to which many clients are connecting. To prevent this, you'll implement rate-limiting on the API side of things, limiting users to only 100 API requests per hour.

The way you're going to do this is to add a new field to the users table that stores a count of how many requests the user has made per hour. To reset the user's count back to 0, you'll create a method that finds only the users who've made requests in the last hour and reset their counts.

13.3.1 One request, two request, three request, four

Currently in app/controllers/api/v1/base_controller.rb you've got code that only checks if the token specified is correct, and if so, assigns a user to the `@current_user` variable:

```
def authenticate_user
  @current_user = User.find_by_authentication_token(params[:token])
  unless @current_user
    respond_with({ :error => "Token is invalid." })
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  end
end
```

You'll now be able to do whatever you wish to this user object in an API request. First, you're going to make sure that it's incrementing the request count for a user whenever they make an API request. For this, you're going to need a field in the database to keep a track of user API requests. You'll generate a migration using this command:

```
rails g migration add_request_count_to_users request_count:integer
```

This migration will do exactly what you say it should do: add a field called `request_count` to the `users` table. You'll need to modify this migration slightly so that the field defaults to 0, which you can do by replacing this line in the new migration:

```
add_column :users, :request_count, :integer
```

with this:

```
add_column :users, :request_count, :integer, :default => 0
```

You can run these two commands to run this migration, and then you'll be on your way:

```
rake db:migrate
rake db:test:prepare
```

You can now write a test to make sure that the request count is going to be incremented with each request. You'll open a new file at spec/api/v1/rate_limit_spec.rb so that you can separate these tests from the others,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

as they are not part of the projects API or the errors from it. Into this file you'll put the code from Listing 13.25

Listing 13.25 spec/api/v1/rate_limit_spec.rb

```
require 'spec_helper'

describe "rate limiting", :type => :api do
  let(:user) { Factory(:user) }

  it "counts the user's requests" do
    user.request_count.should eql(0)
    get '/api/v1/projects.json', :token => user.authentication_token
    user.reload
    user.request_count.should eql(1)
  end
end
```

When you run this spec now with `bin/rspec spec/api/v1/rate_limit_spec.rb`, it's going to fail on the final line because the request count hasn't been incremented:

```
Failure/Error: user.request_count.should eql(1)

  expected 1
       got 0

  (compared using eql?)
```

Alright, now that you've got a failing test, you can make it work! To do that, you're going to need to make the request count for the user increment every time they make a request through the API. Open app/controllers/api/v1/base_controller.rb and add in a new method called `check_rate_limit` right underneath the `current_user` method. This method will just increment the `request_count` field for the current user and uses this code:

```
def check_rate_limit
  @current_user.increment!(:request_count)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
end
```

By calling the `increment!` method on the user object, the field specified will be incremented once. To call this method, you'll put it as another `before_filter` underneath the `authenticate_user` one at the top of this controller:

```
before_filter :check_rate_limit
```

That's all there is to it, and so it will pass when you run `bin/rspec spec/api/v1/rate_limit_spec.rb`:

```
1 example, 0 failures
```

Which is splendid. Before you run any more specs or make any commits, you'll do what you came here to do: limit some rates.

13.3.2 No more, thanks!

You've got the method called `check_rate_limit`, but it's not actually doing any checking right now, it's only incrementing. You should do something about this.

You'll begin by writing a test to check that people who reach the rate limit (of 100) receive a warning that tells them simply "Rate limit exceeded." You'll put this new test underneath the previous test you wrote in spec/api/v1/rate_limit_spec.rb using the code from Listing 13.26.

Listing 13.26 Rate limiting test

```
it "stops a user if they have exceeded the limit" do
  user.update_attribute(:request_count, 101)
  get '/api/v1/projects.json', :token => user.authentication_token
  error = { :error => "Rate limit exceeded." }
  last_response.status.should eql(403) <co id="ch13_900_1"/>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  last_response.body.should eql(error.to_json)
end
```

In this spec, you set the request count to be one over the 100 limit. If the user makes another request, they should see the "Rate limit exceeded" error. For the status in this spec, you're expecting the error to be set to a `403`, which would indicate a forbidden request that is perfectly in line with the *no, we're not going to let you make any more requests* theme you've got going on.

To make this work you'll change the `check_rate_limit` method in app/controllers/api/v1/base_controller.rb to what is shown in Listing 13.27

Listing 13.27 check_rate_limit method of Api::V1::BaseController

```
def check_rate_limit
  if @current_user.request_count > 100
    error = { :error => "Rate limit exceeded." }
    respond_with(error, :status => 403)
  else
    @current_user.increment!(:request_count)
  end
end
```

In this method, if the user's current `request_count` is greater than 100, then you respond with the "Rate limit exceeded" error and set the status code of the response to 403, meaning "Forbidden". If it's less than 100, then you'll increment their request count. This should be enough to make your spec pass now, so let's run `bin/rspec spec/api/v1/rate_limit_spec.rb` and find out if this is working:

```
2 examples, 0 failures
```

Our API is now limiting requests to 100 per user, but that's for all time right now, which isn't fun. So you're going to need a method that will reset the request count for all users who've made requests. It's the final step you need to complete the rate limiting part of your API.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

13.3.3 Back to zero

You need to reset the `request_count` of each user who's made a request to your API. This will be a method on the `User` model, and so you'll put its test in a new file as spec/models/user_spec.rb file, inside the `describe User` block, using the code from Listing 13.28

Listing 13.28 spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  it "resets user request count" do
    user = Factory(:user)
    user.update_attribute(:request_count, 42)
    User.reset_request_count!
    user.reload
    user.request_count.should eql(0)
  end
end
```

With this spec, you set a new user's request count to something other than 0. 42 is a random number[16], and you're quite fortunate for it to exist so that you can use it. The `reset_request_count!` method isn't defined, but as the remainder of the test implies, the user's request count should be 0. This test will fail when you run it with `bin/rspec spec/models/user_spec.rb` because the `reset_request_count!` method does not exist:

Footnote 16 Not really.

```
Failure/Error: User.reset_request_count!
NoMethodError:
 undefined method `reset_request_count!' for ...
```

As the `reset_request_count!` method is called on `User`, you'll be defining this method in app/models/user.rb. This method will need to reset the `request_count` attribute for all users, and can acheive this goal by using this code, placed above the `to_s` method inside the `User` class:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
def self.reset_request_count!
  update_all("request_count = 0", "request_count > 0")
end
```

You're placing this code right above the `to_s` method because it is best practice to place class methods (such as `reset_request_count!`) above instance methods in a model, as some instance methods may refer to class methods. Also, if everybody puts their code in logical places, then you won't be confused when you look at it, which is what Rails is all about.

The `update_all` method here will set the `request_count` on all user records (the first argument) that have a `request_count > 0` (the second argument), or a `request_count` greater than 0. No point resetting counts that are zero back to zero.

Now that the `reset_request_count!` method is defined, does it work as your test says it should? Well, let's run `bin/rspec spec/models/user_spec.rb`:

```
1 example, 0 failures
```

Cool, so now you've got the request count being reset for all users whenever this method is called. You'll take a look at calling this method automatically when we look at background jobs in chapter 16.

That completes everything you need to do for rate limiting in your API. Before you make a commit you'll run all the specs with the `rake spec` to see if the API is still working:

```
103 examples, 0 failures
```

All good! You can commit this now.

```
git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git commit -m "Add rate limiting to the V1 API"
git push
```

13.4 Versioning an API

13.4.1 Creating a new version

```
namespace :api do
  namespace :v1 do
  resources :projects do
    resources :tickets
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
namespace :api do

  namespace :v1 do
    ...
  end

  namespace :v2 do
    resources :projects do
      resources :tickets
    end
  end
end
```

Now you'll need to create a new `app/controllers/api/v2` directory, which you can do by copying `app/controllers/api/v1` into that location. You've now got version 2 of your API. You'll need to open these controllers and replace the multiple occurrences of `Api::V1` with `Api::V2`.

Strangely, version 2 of your API is, right now, identical to version 1 of your API. That's intentional: a new version of the API should be an improvement, not an entirely new thing. With this separation, you can modify version 2 of the API how you please, leaving version 1 alone.

Before deprecate the `name` field in your project responses, you'll write a test to make sure that this is gone. This test will now test version 2 of the API, and so you'll copy over the `spec/api/v1` directory to `spec/api/v2`, also replacing occurrences of `v1` in these files with `v2`. The test for the new `title` field will now go in spec/api/v2/projects_spec.rb and will test that the `projects viewable by this user` action returns projects with `title`, and not `name`, using the code from Listing 13.29 to replace the "JSON" example in the "index" `context`.

Listing 13.29 spec/api/v2/projects_spec.rb

```
context "projects viewable by this user" do

  before do
    Factory(:project, :name => "Access Denied")
  end

  let(:url) { "/api/v2/projects" }
  let(:options) { { :except => :name, :methods => :title } }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  it "JSON" do
    get "#{url}.json", :token => token

    body = Project.for(user).to_json(options)

    last_response.body.should eql(body)
    last_response.status.should eql(200)

    projects = JSON.parse(last_response.body)
    projects.any? do |p|
      p["title"] == project.title
    end.should be_true

    projects.all? do |p|
      p["name"].blank?
    end.should be_true
  end
  ...
```

At the beginning of this test, you need to pass the same options to `to_json` as you pass to `respond_with`, as the `respond_with` method generates the same output as `to_json`.

In the final lines of this test, you're checking that it's now `title` and not `name` that returns the correct project title, and that the `name` key on all projects are blank. You'll also need to change the XML test of this method to the code shown in Listing 13.30.

Listing 13.30 XML test for response of /api/v2/projects request

```
it "XML" do
  get "#{url}.xml", :token => token

  body = Project.for(user).to_xml(options)
  last_response.body.should eql(body)
  projects = Nokogiri::XML(last_response.body)
  projects.css("project title").text.should eql("Ticketee")
  projects.css("project name").text.should eql("")
end
```

When you run this test using `bin/rspec spec/api/v2/projects_spec.rb`, it's broken:

```
Failure/Error: last_response.body.should eql(body)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  expected "[{[ticket hash without name key]}]"
  got      "[{[ticket hash with name key]}]"
```

```
def index
  projects = Project.for(current_user)
  respond_with(projects, :except => :name, :methods => :title)
end
```

```
def title
  name
end
```

```
8 examples, 0 failures
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

output of it, and the text *says* that your original API (v1) shouldn't be effected, but was it? A great way to check is a quick run of `bin/rspec spec/api/v1`.

```
15 examples, 0 failures
```

Great, that's all working! A quick run of `rake spec` will confirm your suspicions that nothing is broken.

```
118 examples, 0 failures
```

Awesome stuff. Let's make a new commit:

```
git add .
git commit -m "Implement v2 of the API,
               renaming name to title for projects"
git push
```

Alright, so now you've got two versions of your API. Generally, there's much more than a single change in a new API version, but this is a good start. When you announce this API to the people who use your application, they can switch their libraries over to using it immediately, or, ideally, remain using the old version. After a while, you may elect to turn off the first version of the API, and you would do that by giving your users considerable notice such as a month, and then un-defining the routes and deleting the controllers and tests associated with that version. Out with the old and in with the new, as they say.

13.5 Summary

That wraps up Chapter 13, "Designing an API."

You've seen here how you can use the `Rack::Test::Methods` module, given to you for free by the `rack-test` gem, to test that requests to URLs provided by your application return valid API responses in JSON and XML formats. Users will then be able to make these same requests for their own applications or libraries to get at the data in your system. What they come up with

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

is up to their imagination. In this chapter, we only covered one aspect (projects) for your API, but with the knowledge found in this chapter you could easily create the other aspects for tickets, users, states or tags.

In the second section of this chapter you saw how you can limit the request rate to your API on a per-user basis. Users can make up to 100 requests to your system, and when they attempt to make their 101st the application denies them the request and provides a relevant error message. This is to deter people from excessively using the API, as you do not want your server to become overloaded immediately.

Lastly, you saw how you can generate a new version of your API so that you can introduce a change, or changes, so as to not break the previous version of the API. Once an API has been released to the public, its output shouldn't be modified, as this may affect the libraries referring to it. The easiest way to introduce these modifications is through a new version, which is what you did. Eventually, you may choose to deprecate the old API, or you may not. It's really a matter of personal choice.

Our application's at a pretty great point now and is ready for prime time! To show it off to the masses, it's best that you put the code on a computer dedicated to serving the application, rather than running it on some local hardware. In chapter 14, you'll deploy your application to an Ubuntu 10.10 box, learning about the core components to a deployment software stack as you go.

Index Terms

Callbacks, before_save
Rack::Test::Methods module
respond_to
respond_with
respond_with, :except option
respond_with, :methods option

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

14 Deployment

In this chapter we'll deploy your Ticketee application to a new Ubuntu install. Ubuntu is the preferred operating system for deploying Rails applications, mainly due to its simplicity of use and easy package management. You don't need to install another operating system on your computer, we'll be using a product called Oracle VM VirtualBox.

You'll set up this machine manually so that you can see how all the pieces fit together. There are automatic tools such Puppet[1], Chef[2], Babushka[3] and Gitpusshuten[4] that can do most, if not all, of this setup for us. To cover them all adequately in this chapter would turn the chapter into a book. Deployment is an enormous subject and different people have very different opinions of how it should be done. This chapter will give you an adequate taste of what parts are involved in setting up a server, but shouldn't be considered as the be all and end all of deployment. There are countless ways to skin this cat.

Footnote 1 http://puppetlabs.com

Footnote 2 http://opscode.com/chef/

Footnote 3 http://babushka.me

Footnote 4 http://gitpusshuten.com/

This chapter covers the following processes:

- Setting up a server
- Installing RVM and Ruby
- Creating a user account
- Deploying an application using Capistrano
- Setting up a database server using PostgreSQL
- Running the application using Nginx and Passenger
- Securing the server

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

While this isn't an exhaustive list of everything that needs to be done for deployment, it is a great start. When you're done, you're going to have a server that's able to receive requests and return responses from your application like a normal web server. Let's get into it.

14.1 Server setup

Your first step is to set up the Oracle VirtualBox software on your machine. This software is free, works on all of the main operating system variants, and provides a virtual box (or "environment") that you can run another operating system in while running your current operating system.

As an alternative to VirtualBox you could get a VPS with a paid service such as Linode[5]Slicehost[6] or Amazon EC2[7], which allows you to set up a box with Ubuntu (or any one of a few other operating systems) pre-installed. You could also use Heroku[8], which provides free hosting that has read-only access to the file-system[9]. Each of these services have in-depth guides, which should be used as a primary reference if you're going to take this path.

Footnote 5 http://linode.com

Footnote 6 http://slicehost.org

Footnote 7 http://aws.amazon.com/ec2/

Footnote 8 http://heroku.com

Footnote 9 This would cause the file upload part of Ticketee to fail as it requires write-access. To fix this, you would upload images to Amazon S3. Amazon S3 and Paperclip have good enough documentation that this should be easily figured out.

Either direction is fine. If the latter path is taken, jump straight to Section 2.

14.1.1 Setting up a server using VirtualBox

Oracle VirtualBox[10] is software that allows you to run another operating system inside your main operating system. Coupled with Vagrant[11]--a gem used for setting up VirtualBox servers--this is a perfect combination for getting an experimental server up and running. Vagrant will allow you to download an operating system image and then sett up VirtualBox in an exceptionally easy fashion.

Footnote 10 http://virtualbox.org

Footnote 11 http://vagrantup.com

To install VirtualBox, you must first download it from http://virtualbox.org and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

install it like a normal program.[12] After that, you need to install the vagrant gem, which you can do by running this command:

Footnote 12 However if you're on Windows XP you may encounter issues where it claims to have not been verified correctly. This is a known problem. If you skip the errors, it will still work.

```
gem install vagrant
```

Now that you've got VirtualBox and Vagrant, you can install Ubuntu using Vagrant. This is the operating system that you'll use to run your server. This file is pretty large (over 500 MB) and may not be good for some connections. As an alternative, you would recommend using a VPS, as suggested in the introductory text.

```
vagrant box add base http://files.vagrantup.com/lucid32.box
```

This command will download Ubuntu Lucid Lynx (10.04) which you can use as a perfectly-fine base to set up your server. To start up this server, you need to create a new folder called ubuntu (the name isn't important and could be anything), where the configuration of your server will be stored. You can then run `vagrant up` and `vagrant ssh` to boot it and connect to it through SSH. Altogether:

```
mkdir ubuntu
cd ubuntu
vagrant init
vagrant up
vagrant ssh
```

The `up` command will take a couple of minutes to run, but the `ssh` command should be instantaneous after that.

NOTE **Stopping your servers**
If at any point you wish to shut down your server, you can use the `vagrant halt` command.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This is how you connect to servers in the real world, except you would use a command such as this:

```
ssh username@some-server.somewhere.com
```

The `vagrant ssh` is a good-enough "analogy" to that. By running `vagrant ssh` you connect to your server as the user `vagrant`. This user has administrative (or more commonly referred to as "root") access on this box, and so you're able to install the packages that you need.

If you're using a non-Vagrant machine, you'll first need to set up a user for yourself rather than operating as the root user, as this can be dangerous [13]. To do this, use this command (replacing "user" with a username such as 'ryan'):

Footnote 13 For example, if you were running a script as the root user and that script attempted to delete the `/usr` directory, the command would execute. By executing commands as non-root, you save yourself some potential damage from malevolent scripts. This is because the user will only have access to some directories, rather than `root`, which has access to everything.

```
sudo useradd -d /home/user -m -s /bin/bash -g sudo user
```

This command will create a directory at /home/user and set this user's home path to that directory. This is done with the `-m` and `-d` options, respectively. Next, the command sets the user's shell to `/bin/bash`—which is the default shell of Unix operating systems—using the `-s` option. Near the end of this command, the `-g` option specifies that this user will be a part of the "sudo" group, which will let the user execute commands using `sudo`, a kind of super-user command. This part is important because you'll need these permissions to set up your server. At the end of the command, you specify the username of this new user.

Next, you need to set a password for this user, which you can do with this command:

```
passwd user
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You need to enter the new password for this user twice, and then you're done. You mustn't forget the password, otherwise you'll have to reset it as the `root` user.

With that done, let's switch into this user by using this command:

```
su user
```

Now you're all set up to go about installing the different software packages you'll need to get your server up and running.

14.1.2 Installing the base

The majority of software packages are installed on Ubuntu using a system called Aptitude. You can install packages from their source code too, if you wish (with the help of a package called `build_essential`, which contains the build tools you need). These Aptitude packages are downloaded from a package repository and then installed for you. To ensure that the list of these packages are up-to-date, run this command:

```
sudo aptitude update
```

This command goes through the list of sources, connecting to each of them and downloading a package list that is then cached by Ubuntu. When you go to install a package (your next step), Ubuntu will reference these lists to find the packages and the necessary dependencies for them.

Once this command is complete, continue these up by configuring RVM and creating a deploy user.

RVM is short for "Ruby Version Manager" and provides a simple way to install and maintain versions of Ruby on your system. You're going to be using it today to install a single Ruby version on your system, but it's good to learn it.

To get started, you're going to need to install some packages that will provide the necessary tools to use RVM. These packages are `build-essential`, `git-core` and `curl`. RVM uses these packages to build the Ruby version for your server. The `git-core` provides the base Git functionality that RVM uses to stay up to date, and is also used to deploy your application because you're hosting

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

it on GitHub. Finally, `curl` allows you to connect to a URL and download its content. You'll use this last package to pull down the script to install RVM.

To install these packages, run this command:

```
sudo aptitude -y install build-essential git-core curl
```

There are some additional packages required for Ruby itself and its gems. You can install these packages by running this command (all on one line):

```
sudo aptitude -y install openssl libreadline6 libreadline6-dev zlib1g
zlib1g-dev libssl-dev libyaml-dev libsqlite3-dev sqlite3 libpq-dev
libxml2-dev libbxslt-dev autoconf libc6-dev ncurses-dev automake
libtool bison libcurl4-openssl-dev
```

The `sudo` part of this command tells Ubuntu to run a command as a super-user (called `root`). This particular command will install the `build-essential` package, which contains helpful build tools that you'll need for Ruby and the `git-core` package containing the Git version control system as well as `curl`, which you'll need for installing RVM. With these packages installed, let's install RVM and a version of Ruby.

14.2 RVM and Ruby

You could install Ruby by downloading the package manually, extracting it, and then running the necessary commands yourself, but that's boring. You could also install it using the package manager that comes with Ubuntu, but the Ruby that it provides is old and has been known to be broken.

Wouldn't it be nice if there was a tool that would install Ruby for you? There is! It's called RVM!

14.2.1 Installing RVM

RVM provides several benefits over a standard Ruby compile, such as the ability to easily install and upgrade your Ruby install using commands like `rvm install 1.9.3` to install the latest release for Ruby 1.9.3. You can even choose to install Ruby when you install RVM, which is exactly what you're going to do in a bit. No digging for links on the http://ruby-lang.org site for you, no siree.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

There are a couple of ways you can install RVM. The first is a user-based install, installing it in an `.rvm` directory within the user's own directory. But, you're going to want to access gems at a system-level later on in this chapter, so it's best to install RVM at a system level.[14] To do this, run this command:

Footnote 14 To install RVM at a user level, just remove "sudo" from the command.

```
curl -L https://get.rvm.io | sudo bash -s stable --ruby
```

This script will take a long time to execute as it is installing both RVM and a new version of Ruby. Go grab a coffee, or a glass of water if glorious boosts of caffeine are not your thing.

With these packages installed, you'll experience minimum hassle when you install Ruby itself. To install Ruby using your current user, the user needs to be a part of the `rvm` group, which is a group created by the installation of RVM. To add your current user to this group, run this command, while remembering to replace 'user' with your actual username:

```
sudo usermod -a -G rvm user
```

The `-a` option here tells the command to append some groups to the list of groups that the user is in, and the `-G` option (like you saw before with `useradd`) specifies the group. You specify your username on the end of this command, telling it who you want this new group applied to.

To make the `rvm` command effective for all users, add a line to `/etc/profile`. Whenever new terminal sessions are launched, this file is read and run. Put a line in it using these commands:

```
sudo su
echo 'source "/usr/local/rvm/scripts/rvm"' >> /etc/profile
exit
```

The source command here will load the `/usr/local/rvm/scripts/rvm`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

file for each user whenever they start a new session. To make this change effective for the current session, exit out of the server completely using the `exit` command. Once back in, you should be able to run rvm and have it output the help information.

So now you've got the beginnings of a pretty good environment setup for your application, but you don't have your application on the server yet. To do this, you need to undertake a process referred to as "deployment." Through this process you'll put your application's code on the server and be one step closer to letting people use the application.

When you deploy, you'll use a user without root privileges to run the application, just in case. Call this user the same as your (imaginary) domain: `ticketeeapp.com`.

14.3 Creating a user for the app

You're calling this user `ticketeeapp.com` because if you wanted to deploy more than one application to your server, there will be no confusion as to which user is responsible for what. When you set up a database later on, this username will be the same as your database name. This is for convenience's sake, but also because the database will be owned by a user with the same name, allowing this account and none other (bar the database super user) to access it. It's all quite neat.

To begin to set up this user, run these commands:

```
sudo useradd ticketeeapp.com -s /bin/bash -m -d /home/ticketeeapp.com
sudo usermod -a -G rvm ticketeeapp.com
sudo chown -R ticketeeapp.com /home/ticketeeapp.com
sudo passwd ticketeeapp.com
```

We've used a couple of options to the `useradd` command. The `-s` option sets the shell for the user to `/bin/bash` (the standard shell found in most Unix based operating systems) and the `-d` option sets their home directory to `/home/ticketeeapp.com`, while the `-m` option makes sure that the user's home directory exists. The second command, `chown` (short for "change owner"), changes the owner of the `/home/ticketeeapp.com` directory to be the `ticketeeapp.com` user. The final command, `passwd`, prompts you to set a password for this user, which you should set to something complex (that you'll be able to remember) to stop people hacking your `ticketeeapp.com` user[15].

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 15 Even though this won't matter in a short while (when you turn off password authentication and switch to the more secure *key-based authentication*), it's still good practice to always secure any user account on any system with a strong password.

To make this account even more secure, you can switch to key-based authentication.

14.3.1 Key-based authentication

In this next step, you'll set up a key that will allow you to login as your user and `deploy` on your server without a password. This is called key-based authentication and requires two files: a private key and a public key. The private key goes on the developer's computer and should be kept private, as the name implies, because it is the key to gain access to the server. The public key file can be shared with anybody and is used by a server to authenticate a user's private key.

We'll use a key-based authentication for your server because it is incredibly secure versus a password authentication scheme. To quote the official Ubuntu instructions on this[16]:

> To be as hard to guess as a normal SSH key, a password would have to contain 634 random letters and numbers.
>
> -- OpenSSH Configuring

Footnote 16 https://help.ubuntu.com/community/SSH/OpenSSH/Configuring

Not many people today would be willing to use a password containing 634 random letters and numbers! Considering the average password length is 8 characters, this a vast improvement over password-based authentication.

We're going to enable this key-based authentication for both your current user and your `ticketeeapp.com`. For now, use the same key generated for use with GitHub; however, it's recommended that a *different* key be used for the server.

Public keys are stored at a file called .ssh/authorized_keys located in the user's home directory, the user being the user you will connect as through SSH. When the user attempts to connect to the server, the private and public keys are used to confirm the user's identity.[17] Because the chances of two users having the same public and private key are so astronomically high, it is generally accepted as a secure means of authentication.

Footnote 17 For a good explanation of how this process works, check this page: http://unixwiz.net/techtips/ssh-agent-forwarding.html#agent

In this instance, you'll create two of these ~/.ssh/authorized_keys files; one for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

each user. In each case, create the `~/.ssh` directory before creating the `authorized_keys`. Begin with the user you're currently logged in as.

TIP **If you're using Vagrant...**

Vagrant already has a ~/.ssh/authorized_keys file, so there's no need to recreate it. Overwriting this file may cause `vagrant ssh` to no longer work.

You will also need to forward the SSH port from the virtual machine launched by Vagrant to a local port in order to connect without using Vagrant. While you're here, forward the HTTP port (80) as well so that you can access it from the outside. Go into the Ubuntu directory that you created at the beginning of this chapter, open VagrantFile, and add this inside the `Vagrant::Config.run` block:

```
config.vm.forward_port "ssh", 22, 2200
config.vm.forward_port "http", 80, 4567
```

To connect to this server, use port 2200 for SSH and port 4567 for HTTP. When you use the `scp` command, the port can be specified using the `-P` (capital-p) option and `ssh` using `-p` (lowercase-p), with the port number specified directly after this option. In places where these commands are used, substitute `your-server` with `localhost` and `user` with `vagrant`.

Let's create the `~/.ssh` directory now using this command:

```
mkdir ~/.ssh
```

Now you need to copy over the public key from your local computer to the `~/.ssh` directory on the server, which you can do by running this command on your local system, again replacing 'user' with your actual username:

```
# NOTE: Run this on your *local* machine, not the server!
scp ~/.ssh/id_rsa.pub user@your-server:~/.ssh/[your_name]_key.pub
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

At this stage, you'll be prompted for a password, which is the complex one you set up a little earlier. Enter it here and the file will be copied over to the server.

Add this key to the `~/.ssh/authorized_keys` file *on the server* by using this:

```
cat ~/.ssh/[your_name]_key.pub >> ~/.ssh/authorized_keys
```

This command will append the key to ~/.ssh/authorized_keys if that file already exists, or create the file and then fill it with the content if it doesn't. Either way, you're going to have a ~/.ssh/authorized_keys file, which means that you'll be able to SSH to this server without using your complex password. If you disconnect from the server and then reconnect, you shouldn't be prompted for your password. This means that the authentication is working.

Finally, change the permissions on this ~/.ssh/authorized_keys file so that only the user it belongs to can read it:

```
chmod 600 ~/.ssh/authorized_keys
```

With that set, change into the application's user account by running `sudo su ticketeeapp.com` and run the same steps, beginning with `mkdir ~/.ssh` and ending with disconnecting and reconnecting without password prompt. Remember to change `user` in the `scp` command to be the `ticketeeapp.com` user this time around.

If both of these accounts are working without password authentication, then you may as well turn it off!

14.3.2 Disabling password authentication

You've just implemented key-based authentication on your system for both the accounts you have, thus removing the need for any kind of password authentication. To secure your server against possible password attacks, it's a good idea to turn off password authentication altogether.

To do this, open /etc/ssh/sshd_config using `sudo nano /etc/ssh/sshd_config`[18] and add `PasswordAuthentication no`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

where it would otherwise say `#PasswordAuthentication yes` (the # symbol indicates a commented line, just like Ruby). You can find this line by pressing Ctrl+W, typing in "PasswordAuth", and pressing enter. This configures your SSH server to not accept password authentication.

Footnote 18 nano is the basic editor that comes with Ubuntu

Towards the top of this file there's a line that says `PermitRootLogin yes`. Change this line to read `PermitRootLogin no` instead, so that it blocks all SSH connections for the `root` user, increasing the security further.

Lastly, quit `nano` by pressing Ctrl+X and then Y to confirm that you do want to quit and save. Next, you need to restart the SSH daemon by using this command:

```
sudo su
service ssh restart
exit
```

NOTE **Two files, different purposes**
There is also /etc/ssh/ssh_config, which is a little confusing... two files with nearly identical names. The file you just edited is the file for the SSH *server* (or daemon, hence the d at the end), while the ssh_config file is for the SSH client. Make sure you're editing the right one.

The server is now set up with key-based authentication, which completes the user setup part of this chapter. You can confirm that this is actually working by attempting to SSH to the machine from itself by running this command:

```
ssh localhost
```

This command should output `Permission denied (publickey)`. If you did not turn off password authentication properly, then you would be prompted for a password. Because password authentication is now disabled, it attempts to use a key to authenticate the request but no keys match, and therefore the request fails.

The next step is to install a sturdy database server where you can keep the data

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

for your application when it's deployed. At the moment (on your local machine) you're using the SQLite database for this. That's great for light development, but you probably want something more robust for your application, just in case it gets popular overnight[19]. That robust something is a database server called PostgreSQL.

Footnote 19 Chances are low, but this is more to demonstrate how to set it up with a different database server.

14.4 The database server

PostgreSQL is the relational-database[20] preferred by the majority of Rails developers. It will work perfectly with the Ticketee application because there's no SQLite3 specific code within the Ticketee application at the moment.[21]

Footnote 20 http://en.wikipedia.org/wiki/Relational_database. Constrasts the NoSQL term: http://en.wikipedia.org/wiki/NoSQL

Footnote 21 Some Rails applications are developed on specific database systems and may contain code that depends on that system being used. Be wary.

To install, use the `aptitude` command again:

```
sudo aptitude install postgresql-9.1
```

This will install the necessary software and commands for the database server, such as `psql` (used for interacting with the database server in a console), `createuser` (for creating a user in the system) and `createdb` (for creating databases on the server)[22]. You'll be using these commands to create a user and a database for your application.

Footnote 22 For more information about how to configure PostgreSQL, read about the pg_hba.conf file: http://www.postgresql.org/docs/9.0/static/auth-pg-hba-conf.html

14.4.1 Creating a database and user

To begin this, switch to the `postgres` user, which is another account that this `postgresql-8.4` install has set up. To switch into this user, use this command:

```
sudo su postgres
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This user account is the super user for the database and can perform commands such as creating databases and users, precisely what you want! Creating the database is easy enough, you only need to run `createdb` like this:

```
createdb ticketeeapp.com
```

Creating a user in PostgreSQL is a little more difficult, but (thankfully) isn't rocket science. Using the `createuser` command, answer no to all the questions provided:

```
$ createuser ticketeeapp.com
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

Create the database and the user in PostgreSQL with the same name so that when the system user account of `ticketeeapp.com` attempts to connect to this database they are automatically granted access. There is no need to configure this at all, which is most excellent. This process is referred to as *ident authentication*.

14.4.2 Ident authentication

Ident authentication works by determining if the user connecting has an account with an identical name on the database server. Your system's user account is named `ticketeeapp.com` and the PostgreSQL user you created is also named `ticketeeapp.com`. You can attempt to connect using the `psql` command from the `ticketeeapp.com` user, after first exiting from the `postgres` user's session:

```
exit
sudo su ticketeeapp.com
psql
```

If everything goes well, you should see this prompt:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
psql (9.1.4)
Type "help" for help.

ticketeeapp.com=>
```

This means that you're connected to the `ticketeeapp.com` database successfully. You can now execute SQL queries here if you wish. Exit out of this prompt by typing `\q` and pressing Enter.

That's all you need to do for your database for now. You've got a fully-functioning server, ready to accept your tables and data for your application. Now you need to give it what it needs! You can do this by putting the application on the server and running the `rake db:migrate` command, which will create the tables, and then `rake db:seed`, which will insert the basic data found inside db/seeds.rb.

We're not going to make you manually copy over this application, as this can get repetitive and boring. As programmers, we don't like repetitive and boring. One of your kind is called Jamis Buck, and he created a little tool called Capistrano to help automate the process of deploying your application.

14.5 Deploy away!

Capistrano is a gem originally created by Jamis Buck that is now maintained by Lee Hambley and additional volunteers, as well as the growing community that use it. It was initially designed for easy application deployment for Rails applications, but can now be used for other applications as well. Capistrano provides an easy way to configure and deploy versions of an application to one or many servers.

You'll use Capistrano to put your application's code on the server, automatically run the migrations, and restart the server after this has been completed. This action is referred to as a "deploy."

Before you leap into that however, you're going to set up a *deploy key* for your repository on GitHub.

14.5.1 Deploy keys

If your repository was private on GitHub, you would clone it with the url of `git@github.com:our_username/ticketee.git` and would need to authenticate with a private key. You shouldn't copy your private key to the server because if a malicious person gains access to the server they will also then have your private key, which may be used for other things.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To solve this particular conundrum, generate another private/public key pair just for the server itself and put the public key on GitHub to be a deploy key for this repository. This will allow the server to clone the repository.

To do this, run the following command as the `ticketeeapp.com` user on your server:

```
ssh-keygen -t rsa
```

Hit enter to put the key at the default `~/.ssh/id_rsa.pub` location. You can enter a password for it, however, if you do this you will be prompted for it on every deploy. It's really personal preference whether or not to do this.

This command will generate two new files: a public and private key. The private key should remain secret on the server and shouldn't be shared with any external parties. The public key, however, can be given to anybody. You're going to put this key on GitHub now.

Run the `cat` command to get the contents of the public key file, like this:

```
cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAA...
```

You should copy the output of this command into your clipboard. Your next step is to go to the repository on GitHub and click the "Admin" link in the bar in the view for the repository, shown in Figure 14.1

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

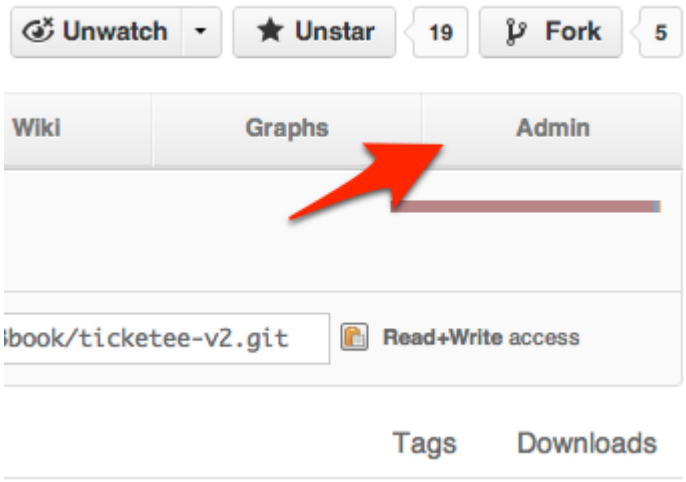

Figure 14.1 Admin button

From here, press the "Deploy Keys" link and then paste the key into the box, calling it "ticketeeapp.com" to keep up with your current naming scheme, shown in Figure 14.2

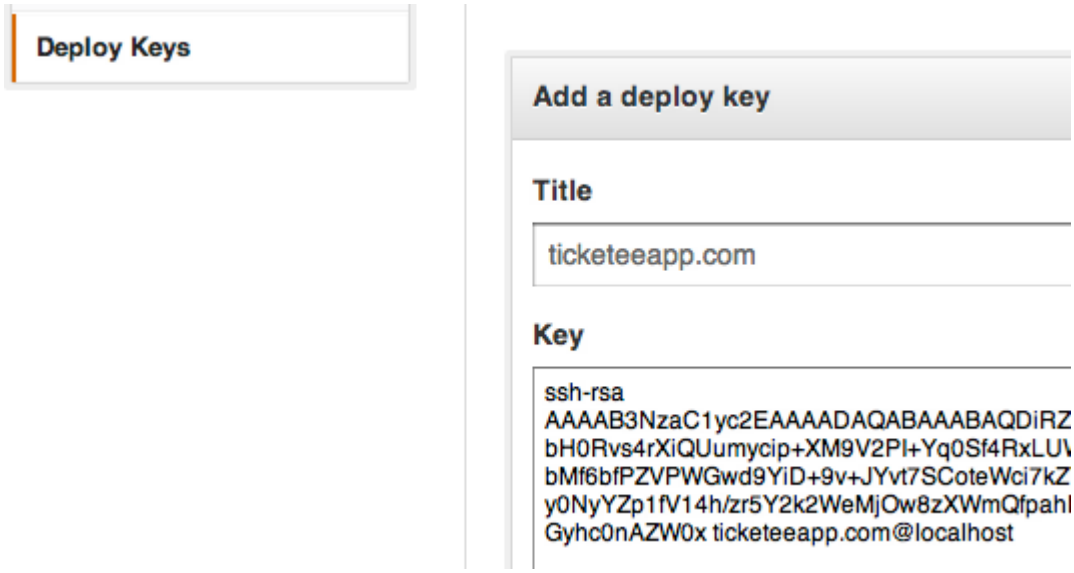

Figure 14.2 Paste in the key, and add a title

When you're done here, press the "Add Key" button, which will add the key you've specified to the list of deploy keys on GitHub. You should then be able to run a `git clone` command on the server using the private URL to clone your repository.

```
git clone git@github.com:[your_github_username]/ticketee.git ~/ticketee
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

If there is a `ticketee` directory at the current location of the server that contains the directories your application should contain, then this works. You can delete this directory now and you'll be putting the code on the server at another location using Capistrano.

Before that happens, you'll need to configure Capistrano.

14.5.2 Configuring Capistrano

To begin with, add the `capistrano` to your application's Gemfile using this code:

```
group :development do
  gem 'capistrano'
end
```

To install this gem, you (and other developers of your application) are able to run `bundle install`, which will keep those other developers up-to-date with all gems. Running `gem install capistrano` would only update them with Capistrano, and even then it may be a version that is incompatible with the one that you've developed.

When the gem is installed you can set it up inside a Rails application by running this command from the root of the application:

```
capify .
```

This will create two files: Capfile and config/deploy.rb. The Capfile is a file containing set up for Capistrano in the application and the following default code that will be used to load your Capistrano configuration:

```
load 'deploy' if respond_to?(:namespace) # cap2 differentiator
Dir['vendor/plugins/*/recipes/*.rb'].each { |plugin| load(plugin) }

load 'config/deploy' # remove this line to skip loading any ...
```

The final line of this file is the most important, as it loads the config/deploy.rb

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

file. This file contains the configuration for deploying your application. Everything in this file dictates how your application is deployed. We'll go through it with you line-by-line, beginning with these two lines:

```
set :application, "set your application name here"
set :repository,  "set your repository location here"
```

When you call `set` in Capistrano, it sets a variable you (or Capistrano itself) can reference later. The `application` variable here should be the name of your application and the `repository` variable should be the path to your application. Change these lines to this:

```
set :application, "ticketee"
set :repository,  "git@github.com:[your github username]/ticketee.git"
```

TIP **Deploying a branch**

When you deploy your application to the server, it will read from the `master` branch. If you'd like to change this, set the branch using this line in your configuration:

```
set :branch, "production"
```

You would also need to create this new branch in the GitHub repository called "production" with the `git checkout -b production` and `git push origin production` commands.

For a good branching model, check out this post: http://nvie.com/posts/a-successful-git-branching-model/.

On the next line of config/deploy.rb there's the `scm` setting:

```
set :scm, :subversion
```

You're going to use Git and not Subversion in this case, so change the line to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

this:

```
set :scm, :git
```

On the next few lines there are a couple of roles defined. These roles point to:

- web: The server or servers responsible for serving requests for your application
- app: The server or servers where the application's code is hosted.
- db: The server or servers where the database for the application is hosted.

Right now we won't worry about multiple server setups, focusing only on having everything on the one box. Your `web`, `app`, and `db` roles are all the same server in this instance. Therefore, you can replace those three lines with this:

```
role :web, "[your-server]"
role :app, "[your-server]"
role :db,  "[your-server]", :primary => true
```

Here you replace `[your-server]` with the address of the server, which is the same one that you've been SSH'ing to. If you're using Vagrant, this address is simply "localhost" and you'll need to add another line to specify the port:

```
set :port, 2200
```

Now that you've covered all the default options in config/deply.rb, you'll add some others to provide more information to Capistrano so that it can set up your application correctly.

The first two settings that you'll need to set up are the user and the path to which you'll deploy your application. Capistrano (unfortunately) can't guess what these are, and so you have to be explicit. The user will be the `ticketeeapp.com` user and the path will be `/home/ticketeeapp.com/apps`. Use the application name as the name of a sub-folder of that application so the application will be deployed into

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

/home/ticketeeapp.com/apps/ticketee. Underneath the `set :scm` line, put these settings:

```
set :user, "ticketeeapp.com"
set :deploy_to, "/home/ticketeeapp.com/apps/#{application}"
```

The `ticketeeapp.com` user doesn't have `sudo` privileges, so tell Capistrano not to use the `sudo` command by using this line:

```
set :use_sudo, false
```

When you deploy your application, you don't want to keep every single release that you've ever deployed on the server. To get rid of the old deploys (referred to as "releases"), put this in config/deploy.rb:

```
set :keep_releases, 5
```

This will keep the last five releases that you have deployed, deleting any releases that are older than that.

Next, you're going to need to tell it to use the `bash` prompt to send commands through, as you will need access to RVM during the deployment. To do this, put this line in config/deploy.rb:

```
default_run_options[:shell] = '/bin/bash --login'
```

Lastly, at the bottom of the file there are a couple of lines for defining `deploy:start`, `deploy:stop` and `deploy:restart` tasks for Passenger, which are commented out. Remove the comment hash from the beginning of these lines, transforming them to this:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
namespace :deploy do
  task :start do ; end
  task :stop do ; end
  task :restart, :roles => :app, :except => { :no_release => true } do
  path = File.join(current_path, 'tmp', 'restart.txt')
    run "#{try_sudo} touch #{path}"
  end
end
```

This defines a blank `start` and `stop` task and a full `restart` task for your `app` role. This task will run the `touch /home/ticketeapp.com/apps/ticketee/tmp/restart.txt` command, which will tell your server (not yet set up) to restart the application, causing your newly deployed code to be started up upon the next request.

With the Capistrano configuration done, you can run the `cap` command, passing in the name of a task to set up your application, such as `deploy:setup`. This task is one of a group of tasks that are provided by default with Capistrano. To see a list of these tasks, use the `cap -T` command.

There's two more things you'll need to do. The first is to tell Capistrano to precompile the assets for your application during the deployment. This step is easy and just requires an extra line at the top of the config/deploy.rb file, which you should put there now:

```
load "deploy/assets"
```

By loading this file, Capistrano will create a shared/assets directory at your deploy path which will be linked to the `public/assets` directory of the latest deploy. During a deploy, this script will tell Capistrano to run `rake assets:precompile` inside the application's directory to generate all the assets, placing them in public/assets.

The second thing is related to the first: Ubuntu does not come with a JavaScript runtime pre-installed, but Mac and Windows do. This is used during the asset precompilation step by execjs, a dependency of the Sprockets gem which provides the asset pipeline features. If you don't install a JavaScript runtime then you will get this error:

```
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Could not find a JavaScript runtime. See
https://github.com/sstephenson/execjs for a list of
available runtimes.
```

Fixing this problem is a simple matter of adding another gem to your Gemfile just for the production environment by using these lines:

```
group :production do
  gem 'therubyracer'
end
```

As long as this gem remains in the Gemfile, then the asset precompilation on the server will run just fine. Alternatively, you could install any one of the supported runtimes listed in the README for https://github.com/sstephenson/execjs. You will need to run `bundle install` to update your application's bundle, and then add both the Gemfile and Gemfile.lock files to Git with this command:

```
git add Gemfile*
gc -m "Added therubyracer and capistrano as dependencies"
git push
```

This is done so that the `therubyracer` dependency is available when the application is cloned onto the server, meaning the deploy will go smoother. Now that that's done, time to setup the environment on the server in preparation for your deploy.

14.5.3 Setting up the deploy environment

You'll now use the `deploy:setup` task, which will set up the folder where your application is deployed, `/home/ticketeeapp.com/apps/ticketee`, with some basic folders:

```
cap deploy:setup
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This command is in the same vein as the `rails new` command you've used previously because it sets up an identical, standard scaffold for every Capistrano. When this command runs, you'll see a large chunk of output that we'll now break down, one line at a time:

```
* executing `deploy:setup'
```

Capistrano tells you the name of the currently executing task, `deploy:setup`. The next line tells you what command it is about to execute.

```
* executing "mkdir -p /home/ticketeeapp.com/apps/ticketee
                      /home/ticketeeapp.com/apps/ticketee/releases
                      /home/ticketeeapp.com/apps/ticketee/shared
                      /home/ticketeeapp.com/apps/ticketee/shared/system
                      /home/ticketeeapp.com/apps/ticketee/shared/log
                      /home/ticketeeapp.com/apps/ticketee/shared/pids
```

These are the basic directories required for Capistrano. The first directory acts as a base for your application, containing several different sub directories, the first of which is `releases`. Whenever you deploy using Capistrano, a new release is created in the `releases` directory, timestamped to the current time using the same time format as migrations within Rails (e.g. 20110205225746, or the full year followed by two-digits each for the month, day, minute, hour and second, or `YYYYMMDDHHmmSS`). The latest release would be the final one in this directory.

The `shared` directory is the directory where files can be shared across releases, such as uploads from Paperclip, that would usually go in the `public/system` directory, which would now be placed in `shared/system`.

The `shared/log` directory is symbolically linked[23] to the current release's `log` directory when you run a deploy. This is so all logs are kept in the `shared/log` directory (rather than in each release) so that, if you choose to, you can go back over them and read them.

Footnote 23 http://en.wikipedia.org/wiki/Symbolic_link

The `shared/pids` directory is symbolically linked to the current release's `tmp/pids` up on deploy. This folder is used for process ids of any other parts of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

your application. At the moment, you don't have any of these and so this directory is of no major concern.

The next line after this makes these folders group writable with the `chmod` command:

```
chmod g+w /home/ticketeeapp.com/apps/ticketee
          /home/ticketeeapp.com/apps/ticketee/releases
          /home/ticketeeapp.com/apps/ticketee/shared
          /home/ticketeeapp.com/apps/ticketee/shared/system
          /home/ticketeeapp.com/apps/ticketee/shared/log
          /home/ticketeeapp.com/apps/ticketee/shared/pids
```

At the bottom of this command's output you can see what servers it will be executed on, with only your one server listed for now. It also tells you that the command is being executed and, faster than you can blink, that the command has finished. `chmod` isn't an intensive operation.

```
servers: ["your-server"]
[your-server] executing command
command finished
```

Once the `deploy:setup` Capistrano task has finished, you are returned to a console prompt. Now you can put the application on the server by deploying it!

14.5.4 Deploying the application

Capistrano has now been configured to deploy the Ticketee application and you've set up your server using the `cap deploy:setup` command, leaving it up to you now to deploy your code. Capistrano's `deploy` task will let you do this, and you can run this task with this command:

```
cap deploy
```

This command outputs an even larger output to `cap deploy:setup`, but again we'll go through it line by line. It's not really all that intimidating when it's

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

broken down into little chunks, really! The first output you'll see from a deploy is:

```
* executing `deploy'
* executing `deploy:update'
** transaction: start
* executing `deploy:update_code'
```

These first three lines tell you the tasks which are being executed. The `deploy` task is going to be executed because you asked Capistrano to do that. This task depends on the `deploy:update` task, and so it will run that first.

The `deploy:update` task begins a transaction (the third line in the above output), which is exceptionally helpful. If anything goes wrong in your deploy, Capistrano will rollback everything to the beginning of this transaction, deleting any code it's deployed. This transaction is a failsafe for your deploy.

The final part of the output is the `deploy:update_code` task, which is responsible for updating the application's code in your deployment environment. This task is responsible for the next chunk of output you see:

```
executing locally: "git ls-remote [git_path] HEAD"
 * executing "git clone -q [git_path] [release_path] &&
    cd [release_path] &&
    git checkout -q -b deploy [SHA1 hash] &&
    (echo [SHA1 hash] > [release_path]/REVISION)"
   servers: ["your-server"]
```

This task first runs `git ls-remote`, a lesser known Git command, locally (not on the server), which will get the current SHA for `HEAD`, the latest commit to the `master` branch, unless you set a `branch` in Capistrano's configuration.

The next thing Capistrano does is put the current revision in a file called `REVISION`. If you like, you can alter the layout of your application to read the value from this file and put it in your application's layout as a HTML comment so that when you do a deploy to the server, you can check this hash to see if it is the latest code.

The next couple of lines output from `cap deploy` are from the beginning of the `deploy:finalize_update` task:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
* executing "chmod -R g+w [release_path]"
  servers: ["localhost"]
  [localhost] executing command
  command finished
```

With this `chmod` command, Capistrano ensures that your new release's directory is group writable (`g+w`), allowing the user / group to make any modifications to this directory they like, barring all others.

Finally, the `deploy:finalize_update` then removes the `log`, `public/system` and `tmp/pids` directories and symbolically links the `shared/log`, `shared/system` and `shared/pids` directories (in your application's deployed path) to these paths respectively. It does that in this little series of commands:

```
* executing "rm -rf [release_path]/log
  [release_path]/public/system
  [release_path]/tmp/pids &&

  mkdir -p [release_path]/public &&
  mkdir -p [release_path]/tmp &&

  ln -s [shared_path]/log [release_path]/log &&
  ln -s [shared_path]/system [release_path]/public/system &&
  ln -s [shared_path]/pids [release_path]/tmp/pids
   servers: ["your-server"]
   [your-server] executing command
   command finished
```

Next, Capistrano will use the `find` command to `touch` every file in the `public/images`, `public/stylesheets` and `public/javascripts` to update their last modified time. This is so that when a user visits your site they get the latest image, stylesheet or javascript file rather than a cached file. It does this with this part of the output:

```
* executing "find [release_path]/public/images
                  [release_path]/public/stylesheets
                  [release_path]/public/javascripts
                  -exec touch -t [timestamp] {} ';'; true"
   servers: ["your-server"]
   [your-server] executing command
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
   command finished
```

The second-to-last step for the `deploy:update` task is to run the `deploy:symlink` task, which symbolically links the new release directory to the `current` folder within your deploy path (in this example, `/home/ticketeeapp.com/apps/ticketee/current`).

```
* executing `deploy:symlink'
* executing "rm -f [current_path] &&

  ln -s [release_path] [current_path]
   servers: ["your-server"]
   [your-server] executing command
   command finished
```

The last action of the `deploy:update` task is to commit the transaction that began at the start, meaning your deploy was successful:

```
** transaction: commit
```

The absolutely final thing the `deploy` task does is call `deploy:restart`, which will touch the `tmp/restart` file in your new application directory (`/home/ticketeeapp.com/apps/ticketee/current`), which would restart the server if you had one running:

```
* executing `deploy:restart'
* executing "touch [current_path]/tmp/restart.txt"
   servers: ["your-server"]
   [your-server] executing command
   command finished
```

And that's it! Our application is deployed for the first time; however, it's not quite ready for prime-time usage. For starters, the application's gems are not installed! On your development box you will do this by running the `bundle`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`install` task, but you're no longer in Kansas[24]or on your own development box for that matter. Bundler has some pretty slick integration with Capistrano, which will run `bundle install` when you deploy. This functionality is provided to you by a file that comes with the gem.

Footnote 24 Apologies for any Kansas-based readers out there. Let me assure you, you are still (most likely) in Kansas.

14.5.5 Bundling gems

You can trigger the `bundle install` task to happen (in a slightly different fashion from usual) when you do a deploy by requiring the `bundler/capistrano` file in the `config/deploy.rb` of your application, right at the top:

```
require 'bundler/capistrano'
```

You'll also need to require RVM's `capistrano` configuration so that when you do a deploy it can locate the `bundle` command (provided by a gem that was installed using an RVM-provided Ruby install), which it will need to run `bundle install`. At the top of `config/deploy.rb`, put these lines:

```
$:.unshift(File.expand_path('./lib', ENV['rvm_path']))
require 'rvm/capistrano'
```

The first line here adds the `lib` directory of RVM to the load path (represented in Ruby by `$:`). This is required so that this file knows where to find `rvm/capistrano`. Without it, it may fail.

Now that you're requiring `rvm/capistrano` when you run `cap deploy` again, you'll see this additional output just after the stylesheets, javascripts and images touching:

```
* executing `bundle:install'
* executing "ls -x /home/ticketeeapp.com/apps/ticketee/releases"
  servers: ["your-server"]
  [your-server] executing command
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  command finished
* executing "bundle install --gemfile [release_path]/Gemfile
  --path [shared_path]/bundle
  --deployment
  --quiet
  --without development test"
  servers: ["your-server"]
  [your-server] executing command
  command finished
```

Bundler's added a `bundle:install` task to your Capistrano configuration which runs after `deploy:finalize_update`. This task runs `ls -x` command at the beginning to get the last release's directory (`20110207202618`, in this case), which it then uses to specify the location of the `Gemfile` using the `--gemfile` flag passed to `bundle install`. Rather than installing the gems to a system location which may not be writable by this user[25], Bundler elects to install this to the `/home/ticketeeapp.com/apps/ticketee/shared/bundler` directory instead, specified by the `--path` flag.

Footnote 25 This directory would be located within /usr/local/rvm, which is only writable by members of the `rvm` group which this member is not a part of and thus, is unable to install any gems at a system-wide level.

The `--deployment` flag specifies that the repository must contain a `Gemfile.lock` file (meaning the gem versions are locked) and that the `Gemfile.lock` file is up-to-date according to the `Gemfile`. This is to ensure that you're running an identical set of gems on your server and local machines.

Lastly, the `--without` flag tells Bundler what groups to ignore. The `development` and `test` groups are ignored in this case, meaning gems specified in these two groups will not be installed at all.

With your application's gems installed, you're getting even closer to having an application running. When you deploy changes to your application, these changes may include new migrations, which will need to be run on the server after you do a deploy. You can deploy your code *and* migrate by running this lovely command:

```
cap deploy:migrations
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

After your code deploys, Capistrano will run the `rake db:migrate` task, which is of great use, as it sets up your database tables. You'll see output like this:

```
** [out :: [server]] (in [path_to_application])
** [out :: [server]] ==  CreateProjects: migrating ===
** [out :: [server]] -- create_table(:projects)
** [out :: [server]] -> 0.0012s
...
```

This indicates that the migrations have happened successfully. Unfortunately, this is in the wrong database! You spent all that time setting up a PostgreSQL server and it's gone ahead and instead used SQLite3. The nerve!

14.5.6 Choosing a database

To fix this, you can make a little change to your application's `Gemfile`. Rather than having `sqlite3` out there in the open and not in a group, switch it to only be used in development and test by moving it down into the `group :development, :test` block just underneath, so that it now looks like[26]:

Footnote 26 Generally, this is a bad idea. You should always develop on the same database system that you deploy on so that you don't run into any unexpected production issues. We're being "lazy" here because it's easier.

```
group :test, :development do
  gem 'gmail', '0.4.0'
  gem 'rspec-rails', '~> 2.9'
  gem 'sqlite3'
end
```

Then, inside the `production` group add the `pg` gem, like this:

```
group :production do
  gem 'therubyracer'
  gem 'pg'
end
```

The `pg` gem provides the PostgreSQL adapter that you need to connect to your PostgreSQL database server on your server. If you run `bundle install` now it will install this gem for you. Now you can make a commit for this small change

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

and push your changes:

```
git add Gemfile*
git commit -m "Add pg gem for PostgreSQL on the server"
git push
```

You haven't yet configured your production application to connect to PostgreSQL, which is somewhat of a problem. You would usually do this by editing the config/database.yml file in your application, but in this case you want to keep your development and production environments separate. Therefore, you'll set this up on the server.

Put this file in the shared directory of your application's deploy, so that all releases can just symlink it to config/database.yml inside the application itself. Connect to the server now with your user and then switch over to `ticketeeapp.com` using `sudo su ticketeeapp.com` so that you can add this file. Go into this shared directory now and open a new file for editing by running these commands as the `ticketeeapp.com` user:

```
cd /home/ticketeapp.com/apps/ticketee/shared
mkdir config
cd config
nano database.yml
```

Inside this file, put the database settings for the `production` environment of your application. These are as follows:

```
production:
  adapter: postgresql
  database: ticketeeapp.com
  min_messages: warning
```

You can exit out of `nano` by using Ctrl+X and then press Y to confirm your changes.

Your next step is to get this file to replace the config/database.yml that your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

application contains upon deployment. For this, define a new task at the bottom of config/deploy.rb in your application:

```
task :symlink_database_yml do
  run "rm #{release_path}/config/database.yml"
  run "ln -sfn #{shared_path}/config/database.yml
       #{release_path}/config/database.yml"
end
after "bundle:install", "symlink_database_yml"
```

This task will remove the current config/database.yml located at the `release_path` and will then link the one from `shared_path`'s config/database.yml into that spot. The final line that you have added tells Capistrano to run this task after the `bundle:install` task has been completed, meaning it will happen before anything else.

Now when you run `cap deploy:migrations` again, you'll see this additional output:

```
* executing `symlink_database_yml'
* executing "rm [release_path]/config/database.yml"
  servers: ["your-server"]
  [localhost] executing command
  command finished
* executing "ln -s [shared_path]/config/database.yml
             [release_path]/config/database.yml"
  servers: ["your-server"]
  [localhost] executing command
  command finished
```

It looks like your command is working! Another clue indicating this is the migration output just beneath. Check that the command is truly working by going onto the server as the `ticketeeapp.com` user and then going into the `/home/ticketeeapp.com/apps/ticketee/current` folder and running `rake RAILS_ENV=production db:seed` to load the default data into the production database. Then launch a PostgreSQL console by running the `psql` command. Inside this console, run `SELECT * FROM projects;`. You should see output like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
ticketeeapp.com=> SELECT * FROM projects;
 id |     name      |         created_at         | ...
----+---------------+----------------------------+ ...
  1 | Ticketee Beta | 2012-08-18 12:05:55.447643 | ...
(1 row)
```

The above output shows the data in the projects table that comes from db/seeds.rb, which means that your database configuration has been copied over and your database has been set up correctly.

Capistrano allows you to put the code on your server in a simple fashion. Once you make a change to your application, you can make sure that the tests are still passing, make a commit out of that, and push the changes to GitHub. When you're happy with the changes, you can deploy them to your server using the simple `cap deploy:migrations` command. This will update the code on your application, run `bundle install`, and then run any new migrations you may have added.

There's much more to Capistrano than this, and you can get to know more of it by reading the Capistrano Handbook[27] or by asking questions on the Capistrano Google Group at http://groups.google.com/group/capistrano.

Footnote 27 https://github.com/leehambley/capistrano-handbook/blob/master/index.markdown

To run this application and make it serve requests, you could use `rails server` like in development, but there's a couple of problems with this approach. For starters, it requires you to always be running a terminal session with it running, which is just hackish. Secondly, this process is only single-threaded, meaning it can only serve a single request at a time.

There's got to be a better way!

14.6 Serving requests

Rather than taking this approach, you're going to show you how to use the Passenger gem along with the nginx webserver to host your application. The benefit of this is that when a request comes into your server, it's handled by nginx and an nginx module provided by the Passenger gem, as shown in Figure 14.3

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

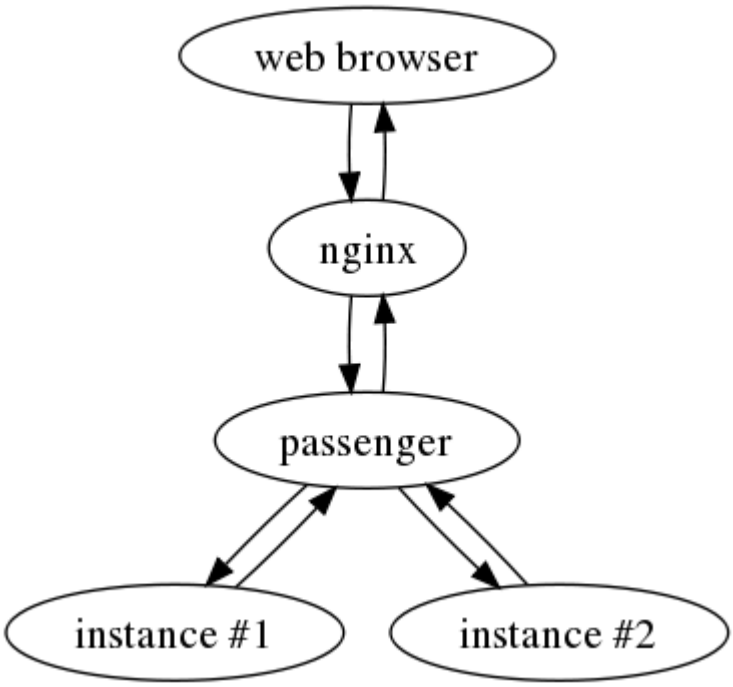

Figure 14.3 nginx request path

When the client sends a request to the server on port 80, nginx will receive it. nginx then looks up what is supposed to be serving that request and sees that Passenger is configured to do that, and so passes the request to Passenger.

Passenger manages a set of Rails instances (referred to as a "pool") for you. If Passenger hasn't received a request in the last five minutes, Passenger will start a new instance[28], passing the request to that instance, with each instance serving one request at a time. The more instances you have, the more (theoretical)[29] requests you can do. If there has been a request within that timeframe, then the request is passed to one of the instances in the pool already launched by a previous request[30].

Footnote 28 The `passenger_pool_idle_time` configuration option is responsible for this: http://www.modrails.com/documentation/Users%20guide%20Nginx.html#PassengerPoolIdleTime

Footnote 29 There's a hardware limit (when you run out of CPU and RAM) that will be reached if too many instances are started up. Things can get slow then.

Footnote 30 Passenger will scale up instances depending on the speed of requests coming to the application. The maximum number of application instances running at any one time by default is 6, and can be configured by the `passenger_max_pool_size` setting: http://www.modrails.com/documentation/Users%20guide%20Nginx.html#PassengerMaxPoolSize

Once Rails has done its thing, it sends the request back up the chain, going through Passenger to Nginx and then finally back to the client as the response,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

most commonly HTML but it could be anything, really. When you launch `rails server`, a request from the client is directly dealt with by the server, rather than being proxied through nginx.

In the `rails server` example, there's only one instance of your application serving requests and so it's going to be slower than having a collection of instances serving them, which is what automatically happens when you use Passenger. Additionally, nginx is super quick at serving files (like your CSS and JavaScript ones) and handles these requests itself, without Rails knowing about it. When you run `rails server`, it serves *every* request, and is definitely not "webscale", and therefore not suitable for a production environment. nginx and Passenger are designed for speed and reliability, and so you should feel pretty confident in using them.

Enough talk, let's get into this! you're going to install the `passenger` gem now, and it's nice enough to set up nginx for you too!

14.6.1 Installing Passenger

To install Passenger, as your user on the box (`vagrant ssh`, if Vagrant) you can run the same `gem install` you've been running all this time:

```
gem install passenger
```

Once this gem is installed, install nginx and the Passenger module by running this lovely command. The `-i` option "simulates initial login", meaning that the RVM script will run before this command, making it available:

```
sudo -i passenger-install-nginx-module
```

At the prompt, press 1 for the install process to download and compile nginx automatically. When prompted for a directory (`/opt/nginx`), hit enter. This'll be the directory where your server runs from. After this, nginx will be compiled and installed. This process takes a minute or two, so go grab something to eat or drink, or stretch.

Once it's done, you're told that Passenger inserted some configuration for you,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

wasn't that nice of it?

```
The Nginx configuration file (/opt/nginx/conf/nginx.conf)
must contain the correct configuration options in order for
Phusion Passenger to function correctly.

This installer has already modified the configuration file for you! The
following configuration snippet was inserted:

  http {
      ...
      passenger_root /usr/local/rvm/gems/ruby-1.9.3-p0/gems/pas...
      passenger_ruby /usr/local/rvm/wrappers/ruby-1.9.3-p0/ruby;
      ...
  }

After you start Nginx, you are ready to deploy any number of
Ruby on Rails applications on Nginx.
```

When you upgrade Passenger you'll need to edit the `passenger_root` line to point to the new version, and if you ever upgrade Ruby then you'll need to change the `passenger_ruby` line. Hit enter now to see the next bit of output, where you're told how to deploy a Rails application.

```
server {
     listen 80;
     server_name www.yourhost.com;
     root /somewhere/public;   # <--- be sure to point to 'public'!
     passenger_enabled on;
  }
```

This bit of configuration goes inside the /opt/nginx/conf/nginx.conf file. You can open this file with `sudo /opt/nginx/conf/nginx.conf`. It's already got a server block in there which is a default configuration for nginx that you can remove. In it's place, put the code from Listing 14.1 (based on the advice offered by Passenger).

Listing 14.1 /opt/nginx/conf/nginx.conf

```
server {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    listen 80;
    server_name your-server.com;
    root /home/ticketeeapp.com/apps/ticketee/current/public;
    passenger_enabled on;
}
```

You can now start the nginx server by running the `nginx` executable:

```
sudo /opt/nginx/sbin/nginx
```

You can make sure that requests to this server are working by accessing http://your-server or http://localhost:4567 if you're using Vagrant. You should see the sign-in page for your application, as shown in Listing 14.2

Figure 14.4 Sign in page for Ticketee

This means your web server is now working seamlessly with your application and everything's almost ready. If the operating system of the server restarts however, this nginx process will not. To fix this small problem, you need to create an *init script*.

14.6.2 An init script

An init script is a script that is run on startup (init) of the operating system and is usually used for launching applications or running commands. In Ubuntu, they reside in the /etc/init.d directory. Here, you're going to use one to start nginx. This script has already been prepared for you and you can download it using this command:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
sudo wget http://bit.ly/nginx-init-script -O /etc/init.d/nginx
```

This command will download the nginx init script and place it at `/etc/init.d/nginx`. This file won't automatically run on boot unless you tell Ubuntu it should, which you can do with these following commands:

```
sudo chmod +x /etc/init.d/nginx
sudo /usr/sbin/update-rc.d -f nginx defaults
```

If you were to reboot the deploy server right now, nginx would restart automatically along with the other services on the system. You don't need to do it now, but it's good to know that it'll start nginx when the server boots now.

There you have it, the application is now deployed onto your Ubuntu server using Capistrano and is running through the power of nginx and Passenger.

14.7 Summary

In this chapter we covered one of the many different permutations you can use to deploy a Rails application to a server. This one has covered the most commonly used software packages such as RVM, Capistrano, PostgreSQL, nginx and Passenger, and therefore it should be a great starting ground for anybody learning about deployment.

There's plenty of other tools out there such as Puppet[31]Chef[32]Babushka[33] and Gitpusshuten[34]Different people prefer different ways of doing similar things, and so there's a wide variety of choice out there. To cover everything within one chapter is just not possible.

Footnote 31 http://puppetlabs.com

Footnote 32 http://opscode.com/chef/

Footnote 33 http://babushka.me

Footnote 34 http://gitpusshuten.com/

You set up your server with Ruby 1.9.3 running your Rails 3.2.8 application. You began by installing the essential packages you needed, then installing RVM, followed by Ruby.

Afterwards, you set up a user with the same name as your application. This was

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

shortly followed by the locking down of SSH access on the machine: now nobody is able to access it with a password, as they need to have the private key instead. Disabling root access is just generally good practice. Nobody should ever need to use the root account on this machine, as everything can be managed by your user or the application's user.

Then we had you set up a database server using PostgreSQL, one of the most popular relational-datastores today. You discovered that giving your system user the same name as your database came in handy; PostgreSQL supports a kind of authentication that automatically grants a system user access to a database with the same name. That is of course provided a PostgreSQL user and database exist with that name. Very handy!

Second-to-last, you got down to the meat of the chapter: the first deployment of your application to your server using Capistrano. You saw that the config/deploy.rb file comes in handy, allowing you to specify the configuration of your deployment environment simply. With Capistrano, you distill everything you need to get your application's latest code onto the server down to one command: `cap deploy:migrations`. Every time you need to deploy, run this command and Capistrano (along with your configuration) will take care of the rest.

Finally, you set up nginx and Passenger to serve your application's requests, as well as the static assets of your application. Generally, this is the setup preferred by Rails developers, and so there's a lot of useful knowledge out there. An alternative to this setup would be to use the Apache web server instead of nginx. Both work suitably.

That's your application "done," really. From the first time you ran a test all the way up to deployment, you've covered a lot of important things within Rails. There's still much more to learn (which is why there's more chapters after this one), but right now readers should have a firm grasp of what the process of developing and deploying a Rails application is. In the next chapter, we show you how you can let people authenticate to your application through either Facebook or Twitter.

Index Terms

RVM (Ruby Version Manager)
Vagrant gem
VirtualBox

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

15 Alternative Authentication

Now that your application has been deployed to a server somewhere (or at least we've gone through the motions of doing that!), we're going to look at adding additional features to your application. One of these is OAuth authentication from services such as Twitter and GitHub.

When you sign into a website, you can generally use a couple of authentication methods. The first of these would be a username and password, with the username being forced to be unique. This method provides a solid way to identify what user has logged into the website, and from that identification the website can choose to grant or deny access to specific parts of the site. You have done this with your Ticketee application, except in place of a username, you're using an email address. An email address is an already unique value for users of a website that also allows you to have a way of contacting the user if the need arises. On other websites, though, you may have to choose a username (like with Twitter), or you could be able to use both a username and email to sign in, like with GitHub.

Entering your email address and a password[1] into every website that you use can be time consuming. Why should you be throwing your email addresses and passwords into every website?

Footnote 1 Ideally, a unique password per-site is best for added security. If one site is breached you do not want your password to be the same across multiple sites, as the attackers would gain access to everything.

Then along came OAuth. OAuth allows you to authenticate against an OAuth provider. Rather than giving your username/email and password to yet another site, you authenticate against a central provider, who then provides tokens for the different applications to read and/or write to the user's data on the application.

In this chapter you're going to be using the OAuth process to let users sign in to your Ticketee application using Twitter and GitHub. You'll not only see how easy

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

this is, but also how you can test to make sure that everything works correctly.

Rather than implementing this process yourself, you can use the OmniAuth gem in combination with the `devise` gem that you're already using. While this combination abstracts a lot of the complexity involved with OAuth, it's still helpful to know how this process works. Let's take a look now.

15.1 How OAuth Works

OAuth authentication works in a multi-step process. In order to be able to authenticate against other applications, you must first register your application with them. After this process is complete, you're given a unique key to identify your application and a secret passphrase, which is actually a hash. Neither of these should be shared. When your application makes a request to an OAuth provider, it will send these two parameters along as part of the request so the provider knows which application is connecting. Twitter's API documentation has a pretty good description of the process as an image, which you can see as Figure 15.1.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

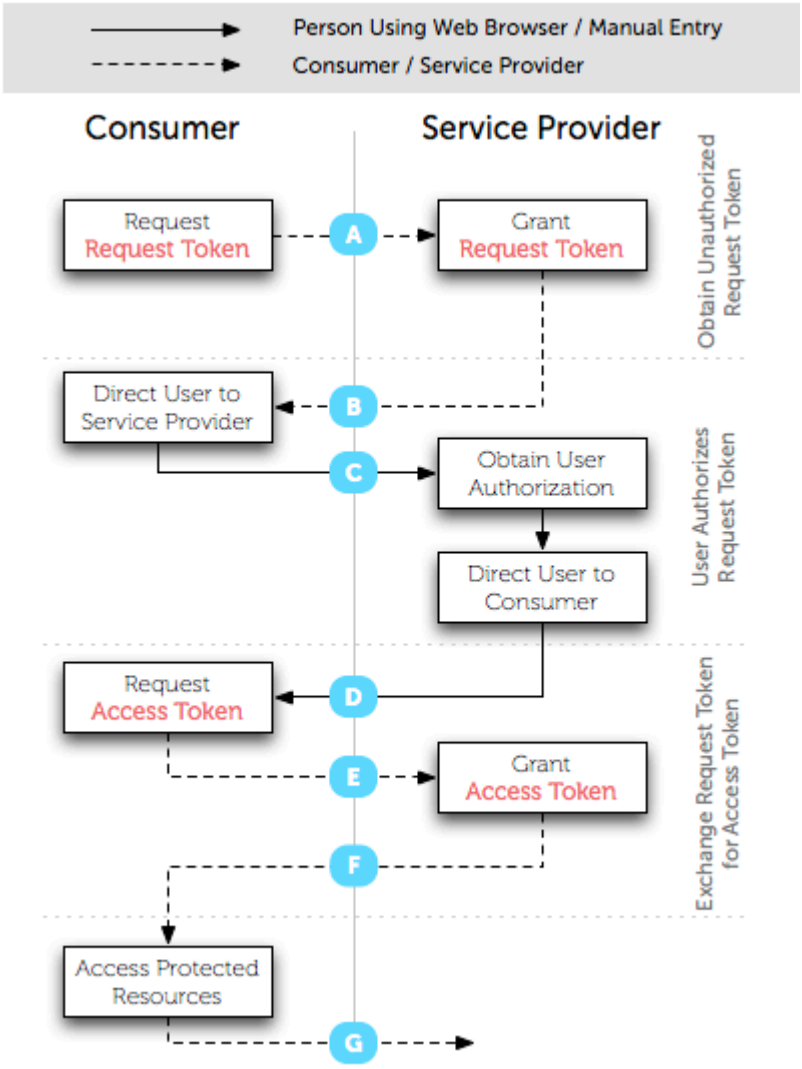

Figure 15.1 Twitter OAuth

First of all (not shown in the figure), a user initiates a request to your application (the "Consumer") to announce their intentions to login with Twitter (the "Service Provider"). Your application then sends that unique identifier and that secret key (given to us by Twitter when you register your application), and begins the authentication process by requesting a token (A). This token will be used as an identifier for this particular authentication request cycle.

The provider (Twitter) then grants you this token and sends it back to your application. Your application then redirects the user to the provider (B) in order to gain the user's permission for this application to access its data. When signing in with Twitter, your users would see something like Figure 15.2.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

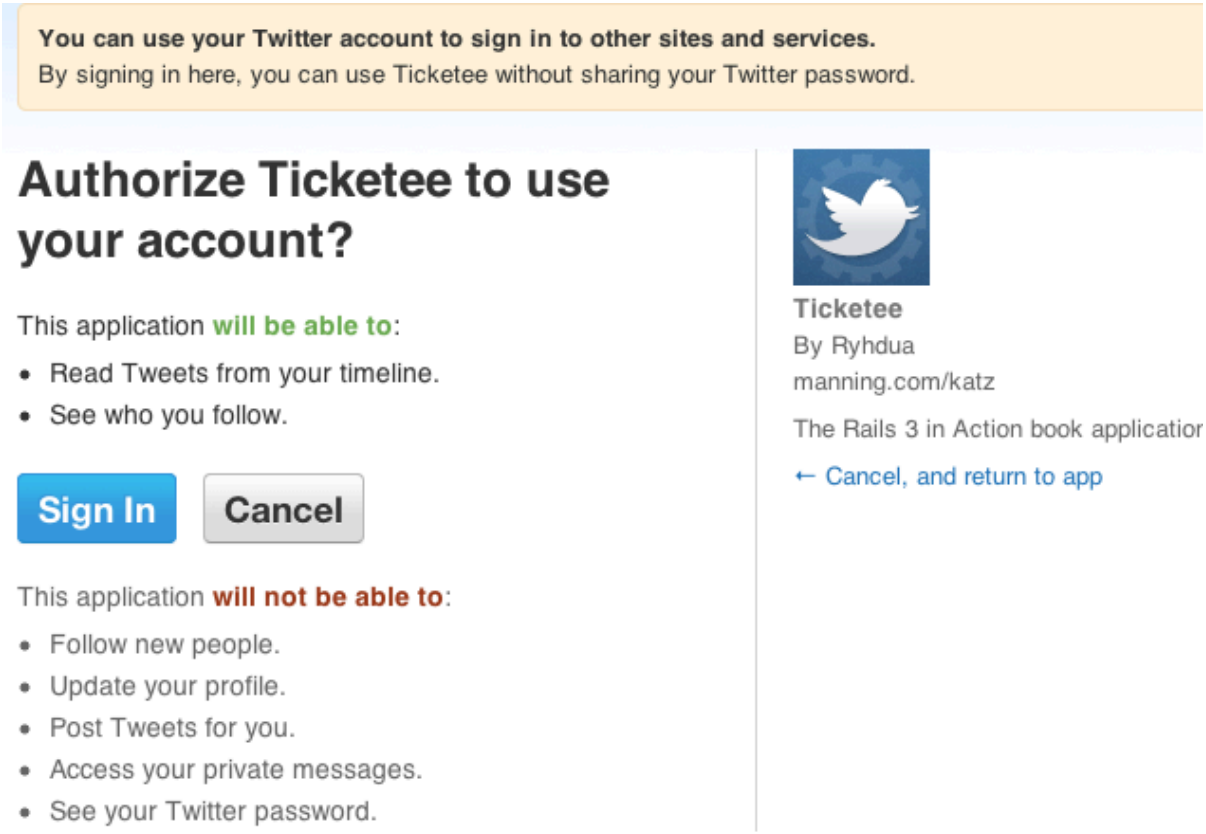

Figure 15.2 Twitter Authorization

The user can then choose to "Sign In" or "Cancel" on this screen. If they choose "Sign In," the application then has access to their data, which authorizes the request token you were given at the beginning. If they press "Cancel," it redirects the user back to the application without giving it access to the data.

In this case, we'll assume the user has pressed "Sign In." The user is then redirected back to your application from the provider, with two parameters: an `oauth_token` and a `oauth_verifier`. The `oauth_token` is the request token you were granted at the beginning, and the `oauth_verifier` is a verifier of that token. OmniAuth then uses these two pieces of information to gain an *access token*, which will allow your application to access this user's data. There's also additional data, such as the user's attributes, that gets sent back here. The provider determines the extent of this additional data.

This is just a basic overview of how the process works. All of this is covered in more extensive detail in Section 6 of the OAuth 1.0 spec, which can be found at http://oauth.net/core/1.0/.

In the case of your application, you're going to be letting users go through this process with the intention of using their authorization with Twitter to sign them in whenever they wish. After this process has been completed a first time, a user will

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

not be re-prompted to authorize your application (unless they have removed it from their authorized applications list), meaning the authorization process will be seamless for the user.

Let's see how you can use the OmniAuth gem to set up authentication with Twitter in your application.

15.2 Twitter Authentication

You're going to be using OmniAuth to let people sign in using Twitter and GitHub as OAuth providers. We'll begin with Twitter authentication and then move on to GitHub after.

15.2.1 Setting up OmniAuth

OmniAuth not only supports OAuth providers, but also supports OpenID, CAS and LDAP. You're only going to be using Twitter's OAuth authentication for now, which you can install in your application by putting this line in your Gemfile:

```
gem 'omniauth-twitter',
  :git => 'https://github.com/arunagw/omniauth-twitter.git'
```

The different parts of OmniAuth are separated out into different gems by an `omniauth-` prefix so that you can use some parts without including all the code for the other parts. In your Gemfile you're loading the `omniauth-oauth` gem, which will provide just the OAuth functionality you need.

Next, you need to tell Devise that your `User` model is going to be using OmniAuth. You can do this by putting the `:omniauthable` symbol at the end of the `devise` list in your app/models/user.rb so that it now becomes this:

```
devise :database_authenticatable, :registerable, :confirmable,
       :recoverable, :rememberable, :trackable, :validatable,
       :token_authenticatable, :omniauthable
```

With OmniAuth setup, you can now configure your application to provide a way for your users to sign in using Twitter. Twitter first requires you to register your application on its site.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

15.2.2 Registering an application with Twitter

You need to register your application with Twitter before your users can use it to login to your application. The registration process gives you a unique identifier and secret code for your application (called a "consumer key" and "consumer secret" respectively), which is how Twitter will know what application is requesting a user's permission.

The process works by a user clicking a small Twitter icon on your application, which will then redirect them to Twitter. If they aren't signed in on Twitter, they will first need to do so. Once they are signed in, they will then be presented with the authorization confirmation screen that you saw earlier, shown again in Figure 15.3.

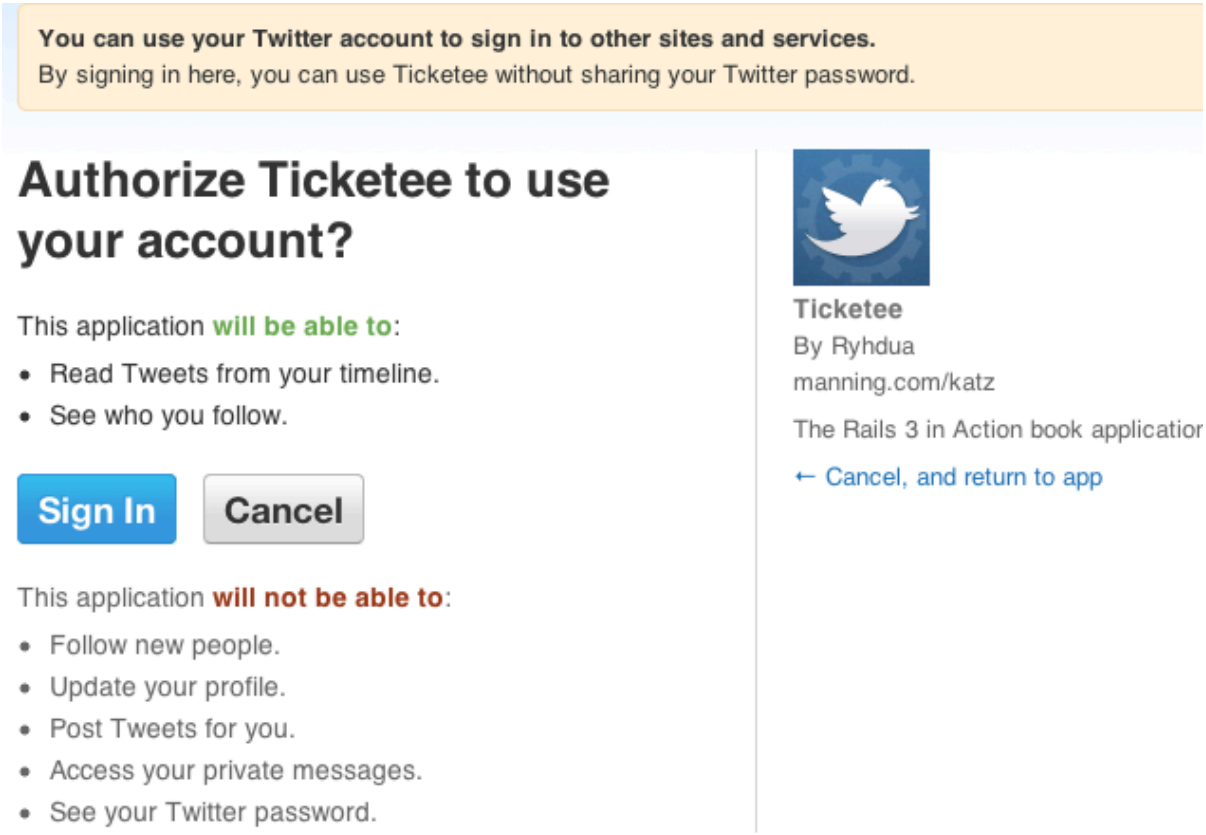

Figure 15.3 Twitter Authorization Request

On this screen you can see that Twitter knows what application is requesting permission for this user, and that the user can either choose to "Allow" or "Deny." By clicking "Allow," the user will be redirected back to your application and then signed in using code that you'll write after you've registered your application.

To register your application with Twitter, you need to go to http://dev.twitter.com and click the "Create an app" link.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

On this new page you need to fill in the name, description and URL fields. The name should be "[Your name]'s Ticketee" as it needs to be unique; the description can be anything, and the URL can be http://manning.com/bigg2. When you click create on this application, you'll see the consumer key and secret that you'll be using shortly, as shown in Figure 15.4.

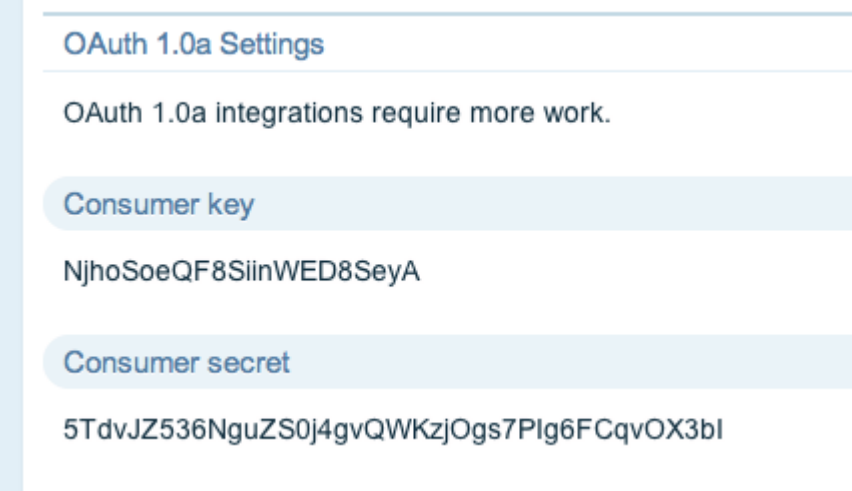

Figure 15.4 A brand new application!

This screen will show you the consumer key and the consumer secret which you will need to use for Omniauth. The other values on this page aren't important for you to know, as OmniAuth will take care of them for you.

You now need to set up your application to use this consumer key and consumer secret when authenticating with Twitter. You can do this in Devise's configuration file in your application, which is located at config/initializers/devise.rb. In this file, you'll see the following commented-out OmniAuth configuration:

```
# ==> OmniAuth
# Add a new OmniAuth provider. Check the wiki for more information
# on setting up on your models and hooks.
# config.omniauth :github, 'APP_ID', 'APP_SECRET',
#   :scope =&gt; 'user,public_repo'
```

This shows you how to add a new OmniAuth provider, using GitHub as an example. In this example, the `APP_ID` and `APP_SECRET` values would be the consumer key and consumer secret given to you by the provider. Set up a new provider for Twitter by putting these lines underneath the commented-out section:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
config.omniauth :twitter,
  '[consumer key]',
  '[consumer secret]'
```

This will configure Devise to provide OmniAuth-based authentication for Twitter, but you're not done yet. You need some way for a user to be able to initiate the sign in process with Twitter.

15.2.3 Setting up an OmniAuth testing environment

To provide a user with a way to sign in with Twitter, you'll add a small addition to your menu bar that lets people sign up and sign in using Twitter, as shown in Figure 15.5.

Figure 15.5 Sign in with Twitter

When a user clicks this button, your application will begin the OAuth process by requesting a request token from Twitter, and then using that token to redirect to Twitter. From here, the user will authorize your application to have access to their data on Twitter, and then they'll be redirected back to your application. It's the user being redirected back to your application that is the most important part. Twitter will send back the `oauth_token` and `oauth_verifier`, and then your application makes the request for the access token to Twitter. Twitter will then send back this access token and any additional parameters it sees fit, and you'll be able to access this information in a Hash format. For example, Twitter sends back the user's information in the response like this:

```
{
  ...
  "extra" => {
    ...
    "user_hash" => {
    "id" => "14506011"
    "screen_name" => "ryanbigg"
    "name" => "Ryan Bigg",
      ...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    }
  }
}
```

This is quite a stripped down version of the response you'll be getting back from Twitter, but it contains three very important values. The first is the unique Twitter-provided `id` of the user, the second is their Twitter username, and the third is their display name. Currently in Ticketee, you've been using the user's email to display who you're logged in as. Because Twitter doesn't send back an email address, you'll have to change where you'd usually display an email address to instead display the user's display name or screen name if they've chosen to sign in with Twitter.

First things first though: you need to have a link that a user can click to begin this process, and to make sure that the link is working you're going to need to write a feature. With this feature, you shouldn't always rely on being able to connect to your OAuth providers like Twitter. Instead, you should create fake responses (referred to as "mocks") for the requests you'd normally do. By doing this you can substantially speed up the rate at which your tests run, as well as not depend on something like connectivity, which is out of your control.

OmniAuth provides a configuration option for setting whether or not you're in a test mode, which will mock a response rather than making a call to an external service. This option is conveniently called `test_mode`. You can set this option at the bottom of your config/environments/test.rb like this:

```
OmniAuth.config.test_mode = true
```

With your test environment now set up correctly, you can write a feature to make sure that users can sign in with Twitter.

15.2.4 Testing Twitter Sign-in

Next, you can begin to write your feature to test Twitter authentication in a new file at spec/integration/twitter_auth_spec.rb as shown in Listing 15.1.

Listing 15.1 spec/integration/twitter_auth_spec.rb

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
require 'spec_helper'

feature 'Twitter Auth' do
  before do
    OmniAuth.config.mock_auth[:twitter] = {
      "extra" => {
        "user_hash" => {
          "id" => '12345',
          "screen_name" => 'twit',
          "display_name" => "A Twit"
        }
      }
    }
  end

  it "signing in with Twitter" do
    visit '/'
    click_link 'sign_in_with_twitter'
    page.should have_content("Signed in with Twitter successfully.")
    page.should have_content("Signed in as A Twit (@twit)")
  end
end
```

This is a simple little feature with a short, 3-line scenario. The code inside the `before` block here will create a fake response from Twitter, which will be used by Omniauth to identify the user who has signed in for this spec. These three fields will be stored in the database and can then be used to link users within Ticketee to users from Twitter.

When you run `bin/rspec spec/integration/twitter_auth_spec.rb`, you'll see that you're missing your link:

```
And I follow "sign_in_with_twitter"
  no link with title, id or text 'sign_in_with_twitter' found ...
```

Rather than have a link that reads "sign_in_with_twitter", you'll actually be giving the link an `id` attribute of "sign_in_with_twitter" and Capybara will still be able to find this link. The link itself is actually going to be a small button that you can get from https://github.com/intridea/authbuttons. You should download these images (just the 32x32px versions) and put them in the app/assets/images/icons directory of your application. Leave them named as they are.

To create this new link, open app/views/layouts/application.html.erb. This file

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

contains the layout for your application and is responsible for displaying the "Sign up" and "Sign in" links for your application if the user isn't signed in already. It's underneath these links that you want to display your little twitter icon, which you can do by making this small change to this file:

```
<%= link_to "Sign up", new_user_registration_path %>
<%= link_to "Sign in", new_user_session_path %>
<br>
Or use <%= link_to image_tag("icons/twitter_32.png"),
                   user_omniauth_authorize_path(:twitter),
                   :id => "sign_in_with_twitter" %>
```

With this link you use the downloaded icon as the first argument of `link_to` by using `image_tag`. The second argument to `link_to` is the routing helper method `user_omniauth_authorize_path` with the `:twitter` argument. This method is provided by Devise because you've told it your `User` model is `omniauthable`. This routing helper will go to a controller that is internal to Devise, as it will deal with the hand-off to Twitter.

When you run this spec again, the second step of your scenario will still fail, but this time with a different error:

```
And I follow "sign_in_with_twitter"
  The action 'twitter' could not be found
  for Devise::OmniauthCallbacksController
```

By default, Devise handles the callbacks from external services using the `Devise::OmniAuthCallbacksController`. Because different people will want this controller to perform differently, Devise provides a set of common functionality in this controller and expects you to subclass it to define the actions (like your `twitter` action) yourself. To do this, create a new controller for these callbacks by running this command:

```
rails g controller users/omniauth_callbacks
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This command will generate a new controller at app/controllers/users/omniauth_callbacks_controller.rb, but it's not quite what you want. You want this controller to inherit from `Devise::OmniauthCallbacksController`, and you also want it to have a `twitter` action. Before you do that, though, tell Devise to use this new controller for its callbacks. You can do this by changing these lines in your config/routes.rb file:

```
devise_for :users, :controllers => {
  :registrations => "registrations",
}
```

into this:

```
devise_for :users, :controllers => {
  :registrations => "registrations",
  :omniauth_callbacks => "users/omniauth_callbacks"
}
```

This will tell Devise to use your newly generated `users/omniauth_callbacks` controller rather than its own `Devise::OmniauthCallbacksController`, which you'll use as the superclass of your new controller. This `Devise::OmniauthCallbacksController` contains some code that will be used in case something goes wrong with the authentication process.

Now you need to define the `twitter` action in this new controller. This action is going to be called when Twitter sends a user back from having authorized your application to have access. Define this controller using the code from Listing 15.2.

Listing 15.2 app/controllers/users/omniauth_callbacks_controller.rb

```
module Users
 class OmniauthCallbacksController < Devise::OmniauthCallbacksController
    def twitter
      @user = User.find_or_create_for_twitter(env["omniauth.auth"])
      <co id="ch15_170_1"/>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
      flash[:notice] = "Signed in with Twitter successfully."
      sign_in_and_redirect @user, :event => :authentication
    end
  end
end
```

When a request is made to this action, the details for the user are accessible in the `env["omniauth.auth"]`❶ key, with `env` being the Rack environment of this request, which contains other helpful things such as the path of the request[2].

Footnote 2 Covered in much more detail in chapter 18.

You then pass these details to a currently undefined method called `find_or_create_for_twitter`, which will deal with finding a `User` record for this information from Twitter, or creating one if it doesn't already exist. You then set a `flash[:notice]` telling the user they've signed in and then use the Devise provided `sign_in_and_redirect` method to redirect your user to the `root_path` of your application, which will show the `ProjectsController`'s `index` action.

To make this action work you're going to need to define the `find_or_create_for_twitter` in your `User` model, which you can do using the code from Listing 15.3.

Listing 15.3 app/models/user.rb

```
def self.find_or_create_for_twitter(response)
  data = response['extra']['user_hash']
  if user = User.find_by_twitter_id(data["id"]) <co id="ch15_188_1"/>
    user
  else # Create a user with a stub password.
    user = User.new(:email => "twitter+#{data["id"]}@example.com",
    <co id="ch15_188_2"/>
                    :password => Devise.friendly_token[0,20])
    <co id="ch15_188_3"/>
    user.twitter_id = data["id"]
    user.twitter_screen_name = data["screen_name"]
    user.twitter_display_name = data["display_name"]
    user.confirm!
    user
  end
end
```

You've defined this class method to take one argument, which is the response

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you get back from Twitter. In this response, there's going to be the access token that you get back from Twitter that you don't care so much about, and also the `extra` key and its value that you do really care about. It's with these that the application then attempts to find a user based on the `id` key ❶ within the `response["extra"]["user_hash"]` (here as `data` to make it easier to type). If it can find this user, it'll return that object.

If it can't find a user with that `twitter_id` attribute, then you need to create one! Because Twitter doesn't pass back an email, you make one up ❷, as well as a password, ❸ using Devise's very helpful `friendly_token` method, which generates a secure phrase like `QfVRz8RxHx4Xkqe6uIqL`. The user won't be using these to sign in; Devise needs them so it can validate the user record successfully.

You have to do this the long way, because the `twitter_` prefixed parameters aren't mass-assignable due to your `attr_accessible` call earlier on in this model, so you must assign them manually one at a time. Store the id of the user so you can find it again if you need to re-authenticate this user, the `twitter_screen_name` and the `twitter_display_name`. Then you need to confirm and save the object, which you can do with the `confirm!` method, and finally you need to return the object as the final line in this `else` block.

These fields are not yet fields in your database, so you'll need to add them in. You can do this by creating a new migration using this command:

```
rails g migration add_twitter_fields_to_users
```

In this migration you want to add the fields to your table, which you can do by adding them to your migration, as shown in Listing 15.4

Listing 15.4 db/migrate/[timestamp]_add_twitter_fields_to_users.rb

```
class AddTwitterFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :twitter_id, :string
    add_column :users, :twitter_screen_name, :string
    add_column :users, :twitter_display_name, :string
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

With this migration set up, you can run it on your development and test databases with `rake db:migrate` and `rake db:test:prepare` respectively. Now when you run your spec again with `bin/rspec spec/integration/twitter_auth_spec.rb`, you'll see that your new `User` object is being created and that you can see the "Signed in with Twitter successfully." message:

```
Failure/Error: page.should have_content("Signed in as A Twit (@twit)")
  expected there to be content ...
```

The final check of your spec is now failing, but this is a pretty easy one to fix. You need to change where it would normally display a user's email to display something like "A Twit (@twit)" if the `twitter_id` attribute is set. To do this, define a new method in your `User` model above the `to_s` method, using the code from Listing 15.5

Listing 15.5 app/models/user.rb

```
def display_name
  if twitter_id
    "#{twitter_display_name} (@#{twitter_screen_name})"
  else
    email
  end
end
```

If the `twitter_id` attribute is set in this method, then you assume the `twitter_display_name` and `twitter_screen_name` attributes are set also and use those to display the twitter name. If it isn't set, then you'll fall back to using the `email` field instead. You'll be able to use this method later on to check if the `github_id` field is set and use the values for that instead[3].

Footnote 3 Alternatively, you could add a feature to let the user pick which one they would like to display.

Now you need to change the occurrences of where `user.email` is referenced to use the `display_name` method instead. The first occurrence of this is in

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

app/models/user.rb in your `to_s` method, which should now become:

```
def to_s
  "#{display_name} (#{admin? ? "Admin" : "User"})"
end
```

The rest of the occurrences are found in a handful of views throughout your application, and you'll need to fix these up now. The first of these is the first line of app/views/admin/permissions/index.html.erb, which should now become this:

```
<h2>Permissions for <%= @user.display_name %></h2>
```

Next, there's one in the application layout at app/views/layouts/application.html.erb:

```
Signed in as <%= current_user.email %>
```

This needs to become simply:

```
Signed in as <%= current_user %>
```

By placing an object like this in the view, the `to_s` method will be called on it automatically which is of course the `to_s` method in the `User` model.

Finally, you'll need to update the app/views/tickets/show.html.erb page in the same manner, changing this:

```
<%= @ticket.user.email %>
```

To this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
<%= @ticket.user.display_name %>
```

That's it! That's all the occurrences of calls to the `email` attribute in places where it's shown to users has been changed to `display_name` instead. So does this mean that your spec will now run? Find out with a quick run of `bin/rspec spec/integration/twitter_auth_spec.rb`.

```
1 example, 0 failures
```

All green, all good. Now users are able to sign up and sign in by clicking the Twitter icon in your application rather than providing you with their email and password. The first time a user clicks this icon, they'll be redirected off to Twitter, which will ask them to authorize your application to access their data. If they choose "Allow," they will be redirected back to your application. With the parameters sent back from the final request, you'll attempt to find a `User` record matching their Twitter ID or, if there isn't one, create one instead. Then you'll sign them in.

After that, when the user attempts to sign in using the Twitter icon, they'll still be redirected back to Twitter, but this time Twitter won't ask them for authorization again. Instead, Twitter will instantly redirect them back to your application; the whole process will seem pretty smooth, albeit with the delay that can normally be expected from doing two HTTP requests.

Go ahead, try launching `rails server` now and accessing the application at http://localhost:3000 by pressing the small Twitter icon on the sign in page. You'll be redirected off to Twitter, which deals with the authentication process before sending you back to the application.

Did you break anything? Let's see by running `rake spec`.

```
120 examples, 0 failures
```

Nope, it seems like everything is functioning correctly. Let's make a commit:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git add .
git commit -m "Add OmniAuth-driven support for signing in with Twitter"
```

With the work you've done in this section, users will now be able to easily sign into your application using Twitter. You can see this for yourself by starting a server using `rails s` and clicking the Twitter icon if you've got a Twitter account.

If your users don't have a Twitter account, then their only other choice at the moment is to provide you with their email address and a password, and that's not really useful to anyone who has a GitHub but not a Twitter account. So let's see how you can authenticate people using GitHub's OAuth next, while recycling some of the Twitter-centric code in the process.

15.3 GitHub Authentication

We've shown how you can let people authenticate using Twitter's OAuth. GitHub also provides this service, and the OmniAuth gem you're using can be used to connect to that too, in much the same way as you did with Twitter. Rather than re-doing everything that you did in the previous section again and changing occurrences of "twitter" to "github," you'll be seeing how you can make the code that you've written so far support both Twitter and GitHub in a clean fashion. When you're done, you're going to have a little GitHub icon next to your Twitter one so that people can use GitHub, Twitter or email to sign in, making your "sign in / sign up area" look like Figure 15.6

Figure 15.6 GitHub Login

As was the case with Twitter, your first step will be registering an application with GitHub.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

15.3.1 Registering and Testing GitHub Auth

To register an application with GitHub, you must first be signed in. Then you can visit https://github.com/settings/applications/new and fill in the form that it provides. After that, you'll need to copy the "Client Id" and "Client Secret" values and put them in your config/initializers/devise.rb file under your Twitter details, like this:

```
config.omniauth :github, "[Client ID]", "[Client Secret]"
```

With GitHub now set up in your application, you can now write the feature to ensure that its authentication is working. To begin testing your application's ability to authenticate users from GitHub, you're going to write a new spec at spec/integration/github_auth_spec.rb and fill it with the content from Listing 15.6.

Listing 15.6 spec/integration/github_auth_spec.rb

```
require 'spec_helper'
feature "GitHub Auth" do
  before do
    OmniAuth.config.mock_auth[:github] = {
      "extra" => {
        "user_hash" => {
          "id" => '12345',
          "email" => 'githubber@example.com',
          "login" => "githubber",
          "name" => "A GitHubber"
        }
      }
    }
  end

  it "can sign in withn GitHub" do
    visit '/'
    click_link "sign_in_with_github"
    page.should have_content "Signed in with GitHub successfully."
    page.should have_content "Signed in as A GitHubber"
  end
end
```

Although it may look like all you've done here is replace all the references to Twitter with GitHub... actually, that's precisely what you've done! This is because

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

there should be little difference in how the user interacts with your site to sign in with Twitter or GitHub. The differences should only be behind the scenes, as this is how a user would expect an application to behave[4].

Footnote 4 Also known as "Principle of least surprise" (POLS) or more colloquially, "keep it simple, stupid!" (KISS)

GitHub returns a similar hash to that of Twitter, containing an `extra` key with a `user_hash` key nested inside. Within this nested hash you've got the three parameters that you'll be storing on your end: the id, the login and a name.

When you run your feature with `bin/rspec spec/integration/github_auth_spec.rb`, you'll see that it can't find the button to sign in with GitHub on the page:

```
And I follow "sign_in_with_github"
  no link with title, id or text 'sign_in_with_github' found
```

This means that your `sign_in_with_github` link doesn't exist yet, so you're going to need to create it like you did with your `sign_in_with_twitter` link. You could do this by copying and pasting the Twitter link code underneath itself in app/views/layouts/application.html.erb, ending up with something like this:

```
Or use <%= link_to image_tag("icons/twitter_32.png"),
                   user_omniauth_authorize_path(:twitter),
                   :id => "sign_in_with_twitter" %>
<%= link_to image_tag("icons/github_32.png"),
                  user_omniauth_authorize_path(:github),
                  :id => "sign_in_with_github" %>
```

This code in your application layout is going to get ugly as you add providers, and it's quite a lot of duplication! What would be more sensible is moving this code into a helper method in a new file such as app/helpers/oauth_helper.rb, defining it as shown in Listing 15.7:

Listing 15.7 app/helpers/oauth_helper.rb

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
module OauthHelper
  def auth_provider(name)
    link_to image_tag("icons/#{name}_32.png"),
    user_omniauth_authorize_path(name),
    :id => "sign_in_with_#{name}"
  end
end
```

Then in place of the ugly code in your application layout, you'd put this instead:

```
Or use <%= auth_provider(:twitter) %> <%= auth_provider(:github) %>
```

How's that for simplicity? Well, you could make it even cleaner by accepting any number of arguments to your method, by turning it into this:

```
def auth_providers(*names)
  names.each do |name|
    concat(link_to(image_tag("icons/#{name}_32.png"),
    user_omniauth_authorize_path(name),
    :id => "sign_in_with_#{name}"))
  end
  nil
end
```

This helper uses the `concat` method to output the links to your view. If you didn't use this, it wouldn't render them at all. You could then write this in your application layout:

```
Or use <%= auth_providers(:twitter, :github) %>
```

Now isn't that way nicer? If at any time you want to add or remove one of the links, you only have to add or remove arguments to this method.

When you run this feature again with `bin/cucumber features/github_auth.feature` you'll see that you're on to the next error:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
The action 'github' could not be found
for Users::OmniauthCallbacksController
```

Like you did with Twitter, you're going to need to define a `github` action in the `Users::OmniauthCallbacksController`. This action will find or create a user based on the details sent back from GitHub, using a class method you'll define after in your `User` model. Sound familiar? You can duplicate the `twitter` action in this controller and create a new `github` action from it like this:

```
def github
  @user = User.find_or_create_for_github(env["omniauth.auth"])
  flash[:notice] = "Signed in with GitHub successfully."
  sign_in_and_redirect @user, :event => :authentication
end
```

When you run your feature again with `bin/rspec spec/integration/github_auth_spec.rb`, you'll see that it's now hitting your new `github` action, as it can't find a method that you use in it:

```
undefined method `find_or_create_for_github' for ...
(eval):3:in `github'
```

In this error output you're seeing that Rails is unable to find a `find_or_create_for_github` method on a class, which is the `User` class. You created one of these for Twitter, and unlike the provider links and the callback actions, you're not able to easily create a bit of smart code for your model. But, you can separate out the concerns of the model into separate files, which would make it easier to manage. Rather than filling your `User` model with methods for each of your providers, you'll separate this code out into another module and then extend your class with it.

You can do this by creating a new directory at app/models/user and placing a file called app/models/user/omniauth_callbacks.rb inside it. You should put the content from listing 15.10 inside this file.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 15.8 Listing 15.10 app/models/user/omniauth_callbacks.rb

```
class User < ActiveRecord::Base
  module OmniauthCallbacks
    def find_or_create_for_twitter(response)
      data = response['extra']['user_hash']
      if user = User.find_by_twitter_id(data["id"])
        user
      else # Create a user with a stub password.
        user = User.new(:email => "twitter+#{data["id"]}@example.com",
                        :password => Devise.friendly_token[0,20])
        user.twitter_id = data["id"]
        user.twitter_screen_name = data["screen_name"]
        user.twitter_display_name = data["display_name"]
        user.confirm!
        user
      end
    end
  end
end
```

In this file you define an `OmniauthCallbacks` module inside your `User` class. Inside this module, you've put the `find_or_create_for_twitter` method straight from your `User` model, except you've removed the `self` prefix to the method name. You can now go ahead and remove this method from the `User` model, making it temporarily unavailable.

By separating out the concerns of your model into separate modules, you can decrease the size of the individual model file and compartmentalize the different concerns of a model when it becomes complicated, like your `User` model has.

To make this method once again available, you need to extend your model with this module. You can do this by making the first two lines of your model into:

```
class User < ActiveRecord::Base
  extend OmniauthCallbacks
```

The `extend` method here will make the methods available for the module on the class itself as class methods.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

TIP **Where to extend**

It's generally a good idea to put any `extend` or `include` calls at the beginning of a class definition so that anybody else reading it will know if the class has been modified in any way. If an `extend` is buried deep within a model, then it can be difficult to track down where its methods are coming from.

By adopting a convention of putting things that can potentially seriously modify your class at the top of the class definition, you're giving a clear signal to anyone (including your future self who may have forgotten this code upon revisiting) that there's more code for this model in other places.

You can now define your `find_or_create_by_github` method in the `User::OmniauthCallbacks` module by using the code from listing 15.11.

Listing 15.9 app/models/user/omniauth_callbacks.rb

```
def find_or_create_for_github(response)
  data = response['extra']['user_hash']
  if user = User.find_by_github_id(data["id"])
    user
  else # Create a user with a stub password.  (1)
    user = User.new(:email => data["email"],
                    :password => Devise.friendly_token[0,20])
    user.github_id = data["id"]
    user.github_user_name = data["login"]
    user.github_display_name = data["name"]
    user.confirm!
    user
  end
end
```

You're lucky this time around, as the form of the data you get back from GitHub isn't too different to Twitter, coming back in the `response['extra']['user_hash']` key. In the case of other providers, you may not be so lucky. The form of the data sent back is not standardized, and so providers will choose however they like to send back the data.

Included in the data you get back from GitHub is the user's email address, which you can use (1) to create the new user, unlike with the `find_or_create_for_twitter` method where you had to generate a fake

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

email. The added bonus of this is that if a user wishes to sign in using either GitHub or their email, they would be able to do so after resetting their password.

The final lines of this method should be familiar; you're setting the `github_id`, `github_user_name` and `github_display_name` fields to store some of the important data sent back from GitHub. You're able to re-sign-in people who visit a second time from GitHub based on the `github_id` field you save. Finally, you confirm the user so that you're able to sign in as them.

With the `find_or_create_for_github` method defined, has your feature progressed? Find out with a run of `bin/cucumber features/github_auth.feature`:

```
And I follow "sign_in_with_github"
  undefined method `find_by_github_id' for ...
```

Ah, it would appear that you're not quite done! You need to define the github fields in your users table so that your newly added method can reference them. Go ahead and create a migration to do this now by running this command:

```
rails g migration add_github_fields_to_users
```

You can then alter this migration to add the fields you need by using the code from listing 15.12.

Listing 15.10 Listing 15.12 db/migrate/[timestamp]_add_github_fields_to_users.rb

```
class AddGithubFieldsToUsers < ActiveRecord::Migration
  def change
    add_column :users, :github_id, :integer
    add_column :users, :github_user_name, :string
    add_column :users, :github_display_name, :string
  end
end
```

Alright, you can now run this migration using `rake db:migrate` and `rake`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`db:test:prepare` to add these fields to your users table. Now you can run your feature again with `bin/cucumber features/github_auth.feature` to see this output:

```
Scenario: Signing in with GitHub
  Given I am on the homepage
  And I follow "sign_in_with_github"
  Then I should see "Signed in with Github successfully."
  Then I should see "Signed in as A GitHubber (githubber)"
    expected there to be content "Signed in as A Githubber"
```

The third step of your scenario is now passing, but the fourth is failing because you're not displaying the GitHub-provided name as the "Sign in as ..." line in your application. You can easily rectify this by changing the `display_name` method in app/models/user.rb to detect if the `github_id` field is set like it does already with the `twitter_id` field.

Underneath the display name output for the `if twitter_id` case in app/models/user.rb, add these two lines:

```
elsif github_id
  "#{github_display_name} (#{github_user_name})"
```

Transforming this whole method into this:

```
def display_name
  if twitter_id
    "#{twitter_display_name} (@#{twitter_screen_name})"
  elsif github_id
    "#{github_display_name} (#{github_user_name})"
  else
    email
  end
end
```

Now when you run `bin/cucumber features/github_auth.feature` again, you should see that it's all

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

passing:

```
1 scenario (1 passed)
5 steps (5 passed)
```

Now users are able to use GitHub to sign in to your site, as well as Twitter or their email address if they please. Make a commit for the changes that you've done, but first make sure everything's running with a quick run of `rake cucumber:ok spec`.

```
64 scenarios (64 passed)
746 steps (746 passed)
# and
56 examples, 0 failures
```

All systems green! Time to commit:

```
git add .
git commit -m "Add GitHub authentication support"
git push
```

Now you've seen how you can support another authentication provider, GitHub, along with supporting Twitter and email-based authentication too. To add another provider you'd only need to follow these 6 easy steps:

- Create a new client on the provider's website, which differs from provider to provider.
- Add the new client's information to config/initializers/devise.rb as a new provider.
- Write a test for your new provider to make sure that people can always use it to sign in.
- Add the provider icon to your listed providers in app/views/layouts/application.html.erb by passing another argument to the `auth_proviers` helper method that you defined in `OauthHelper`
- Add a callback to the `Users::OmniauthCallbacksController` by using the `provides` method. Again, passing another argument to this method is all you need.
- Define the `find_or_create_for_[provider]` method in the `User::OmniauthCallbacks` module.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Due to the flexibility offered by Devise and Omniauth, there's no provider-specific configuration you need to do: it all works beautifully. For a full-list of providers, check out the `omniauth` project on GitHub: https://github.com/intridea/omniauth.

Let's see for ourselves if GitHub's authentication is working by launching `rails server` again and going to http://localhost:3000 and clicking on the GitHub icon.

15.4 Summary

In this chapter you've seen how easy it is to implement authentication using two OAuth providers: Twitter and GitHub. You did this using the OmniAuth integration, which is available in Devise versions after 1.2.

For the Twitter section, you implemented the complete flow in a very simple manner using the features given to you by Devise, such as the routing helper, which initially sends a request off to the provider. Before OmniAuth came along, this process was incredibly tedious. It's truly amazing what OmniAuth offers you in terms of integrating with these providers.

When you got to the GitHub section, rather than copying and pasting the code you created for Twitter, you saw how you could reduce repetition in your code by using methods that iterate through a list of providers to display the icons or to provide callbacks.

Now that you've got multiple ways to allow people to sign in to your application, the barrier of entry is lowered because people can now choose to sign in with a single-click (after they've authorized the application on the relevant provider), rather than filling in the sign in form each time. You've also got a great framework in place if you want to add any more providers.

Your application is at a pretty good state now, but you've not yet made sure that it can perform as efficiently as possible. If thousands of users flock to your application, how can you code it in such a way to reduce the impact on your servers? In the next chapter, we look at how you can implement some basic performance enhancements to make your application serve requests faster, or even create a way where a request skips the application altogether.

Index Terms

Devise omniauthable module
Devise OmniAuth configuration
OmniAuth

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

16 *Basic performance enhancements*

When an application is written, it may be done in such a way that it will not perform ideally. A common situation is that an application with a small database will perform quickly because there is less data to retrieve, but starts to slow as the database grows larger. This problem can be fixed in many different ways.

The first way is to limit the amount of data retrieved in any one call to a fixed limit; a process known as *pagination*. At the moment, for example, you're not limiting the number of tickets shown in the `show` action of the `ProjectsController`. The more tickets that get added to a project, the slower the page that shows this data is going to perform because it will have to retrieve more data from the database and render it out to the page. By breaking the data down into a set of pages, you can show 50 tickets per page. This will lessen the load on your database, but not completely eliminate it. That would only be possible if you were to run no queries at all.

You could do exactly that if you cached the output of the page, or even just the part of the page that showed the list of tickets.

The first process involves saving a copy of the page in the public directory, which would then be used to serve this page. Any action on tickets, such as creating one, adding a comment, or changing a state would then wipe this cache and start afresh.

The second process is slightly different. Rather than storing the fragment as a file on the system, you will store it in memory and then access it through a key.

Finally, by adding indexes to key columns in your tables, such as foreign keys, you can greatly speed up the queries it runs too. If you had 10,000 tickets in your system and you wanted to find all the tickets which had `project_id` set to "123", an index would help speed up this process.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

We'll show you examples of all of these approaches in this chapter, beginning with pagination.

16.1 Pagination

We'll discuss two different kinds of pagination here. The first kind paginates the interface that users can see, as shown in figure 16.1.

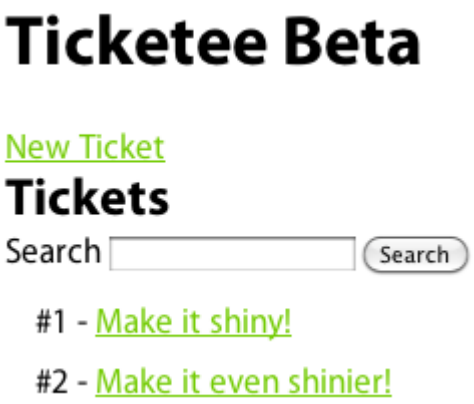

Figure 16.1 Tickets for a project

If this project had a thousand tickets, it wouldn't make sense to show all one thousand at a time. It would also be terribly slow, because the database would have to retrieve 1000 records. Rails would then have to instantiate 1000 `Ticket` objects, render 1000 tickets to the page, and send back that massive chunk of HTML.

The second kind of pagination has to do with your API. Back in chapter 12 you wrote the beginnings of the ticket API and we promised you we'd revisit it in this chapter. Inside the `Api::V1::TicketsController`'s `index` action you have this innocuous looking line:

```
respond_with(@project.tickets)
```

Again, if the database's `tickets` table contains 1,000 records for this project, it will have to send all of them to Rails. Rails will then have to instantiate 1,000 objects, parsing them all to JSON or XML before sending them off to the user. All of this would happen with each request, and if you were getting a lot of requests it would bring your application to its knees.

By paginating the result sets in both of these situations, you can change your

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

application to return only 50 tickets at a time, which would theoretically make your application respond 20 times faster than if it were returning 1,000 tickets. Let's begin by installing a gem called kaminari that will help you with pagination.

16.1.1 Introducing Kaminari

The Kaminari gem[1] is a new breed of pagination gem written by Akira Matsuda, and is considered the Rails 3 successor to the `will_paginate` gem[2], which was the favorite for a long time.[3]

Footnote 1 http://github.com/amatsuda/kaminari

Footnote 2 http://github.com/mislav/will_paginate

Footnote 3 Since this original writing, `will_paginate` has been updated to be Rails 3 compatible.

After you install this gem, you're given an interface on the models of your application, which allows you to make calls like this:

```
@project.tickets.page(2).per(50)
```

This call would ask for the second page of tickets, with each page containing 50 tickets. It's a very clean API. Those familiar with will_paginate will be used to a syntax like this:

```
@project.tickets.paginate(:per_page => 50, :page => 2)
```

The syntax is a little longer, but it's a little clearer what it's doing to those who are familiar with it. You'll use kaminari here just for something different. In your views, you can use the same `paginate` method, which is made available by both gems:

```
<%= paginate @tickets %>
```

This little helper generates the output shown in figure 16.2.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

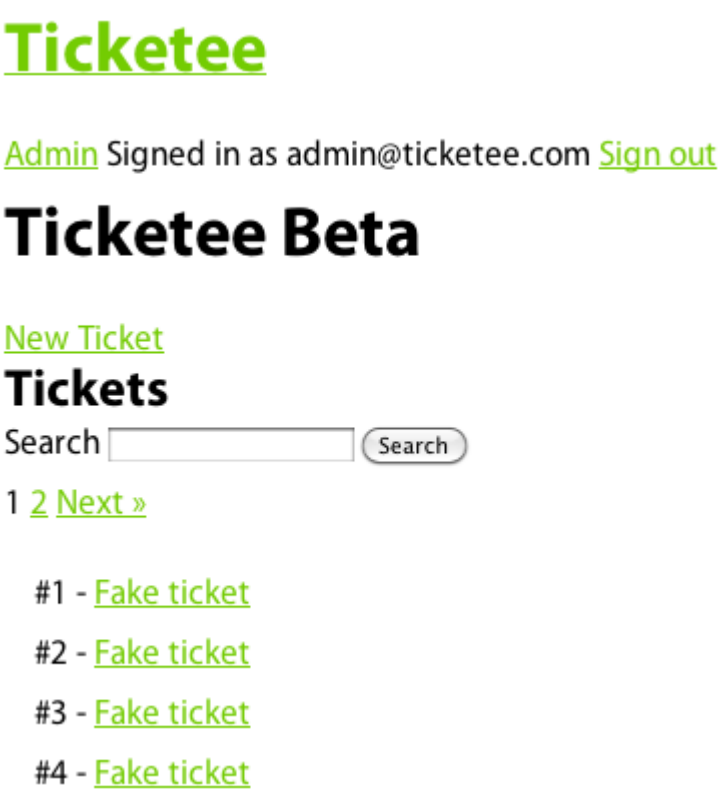

Figure 16.2 Pagination helper

To install this gem, add this line to your Gemfile underneath the `searcher` gem:

```
gem 'kaminari'
```

You'll then run the `bundle install` command to install the gem. With the gem installed, you can now begin to write a Cucumber feature to test that when you're on the tickets page with more than 50 tickets in the system you will see a pagination link somewhere on that page. You should be able to press "Next" and then see the next 50 tickets.

16.1.2 Paginating an interface

You're going to now implement paging for your tickets listing, showing 50 tickets at a time. Users will be able to navigate between pages by clicking the "Next" and "Prev" links. These two links will be provided by a helper from the `kaminari` gem.

Testing pagination

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To test this, we'll write a new scenario at spec/integration/paginating_tickets_spec.rb, shown in Listing 16.1. If we create 100 tickets in this feature, we'll see the pagination links and can then make sure they're working.

spec/integration/paginating_tickets_spec.rb

```
require 'spec_helper'

feature 'Paginating tickets' do
  let(:project) { FactoryGirl.create(:project) }
  let(:user) { FactoryGirl.create(:confirmed_user) }

  before do
    sign_in_as!(user)
    define_permission!(user, :view, project)

    @default_per_page = Kaminari.config.default_per_page
    Kaminari.config.default_per_page = 1

    3.times do |i|
      ticket = project.tickets.new
      ticket.title = "Test"
      ticket.description = "Placeholder ticket."
      ticket.user = user
      ticket.save
    end

    visit root_path
    click_link project.name
  end

  after do
    Kaminari.config.default_per_page = @default_per_page
  end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  it "displays pagination" do
    all(".pagination .page").count.should == 3
    within(".pagination .next") do
      click_link "Next"
    end
    current_page = find("*.pagination .current").text.strip
    current_page.should == "2"
  end
end
```

In this feature you use the FactoryGirl definition for a project you've used many times before to create a project, and then you set Kaminari's default per page to be a low value so that the pagination links are displayed after only a small amount of tickets. Then you create a handful of tickets for this project to ensure that the pagination links will actually appear. If you didn't have enough tickets in your project to warrant pagination then the links would not appear at all.

You then go through the motions of creating a user, giving them access to that project so that they can see into it, signing in as them, and then navigating to that project. On that project's page you should see the pagination links displaying two pages worth of pagination. When you click the "Next" link within the pagination element, you should be on the second page.

When you run `bin/rspec spec/integration/paginating_tickets_spec.rb` you'll see thois error:

```
Failure/Error: all(".pagination .page").count.should == 3
expected: 3
     got: 0 (using ==)
```

Implementing pagination helpers

Your step that checks for 2 pages of pagination wasn't able to see any at all, most likely because you aren't showing any right now! To fix this, you'll have to display the pagination link in app/views/projects/show.html.erb by putting this line above the `ul` that displays tickets:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  <%= paginate @tickets %>
```

This line will display the pagination links that your failing step currently requires. You're going to need to set up the `@tickets` variable for pagination in your controller so that these pagination links know what page you're on and that there are only 50 tickets displayed. You'll replace this line in the `show` action of app/controllers/projects_controller.rb:

```
@tickets = @project.tickets
```

With this line:

```
@tickets = @project.tickets.page(params[:page])
```

This `page` method will set `@tickets` to display only the tickets for the current page number, available in the `params[:page]` variable.

When you run your feature again with `bin/rspec spec/integration/paginating_tickets_spec.rb`, it will pass because you've now got your pagination links showing:

```
1 example, 0 failures
```

That's all there is to paginating a resource. You can also call the `page` and `per` methods on models themselves rather than associations; it was just in this case that you were calling it on an association.

Before you make a commit for this change, quickly make sure that everything's working by running `rake spec`.

```
Failed examples:

rspec ./spec/integration/searching_spec.rb:34
rspec ./spec/integration/searching_spec.rb:43
rspec ./spec/integration/searching_spec.rb:52
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Oh dear, it appears the feature in spec/integration/searching_spec.rb has been broken by your changes! Good thing that you've got a feature to catch these kinds of things.

Fixing broken scenarios

All three tests in this feature failed with the same error:

```
undefined method `current_page' for ...
```

This looks to be associated with the feature you just implemented, as it's trying to call a method called `current_page`. If you look a couple of lines down in the output, you'll see that there's a line in the stack trace that shows that this is from Kaminari:

```
...kaminari/helpers/action_view_extension.rb:21:in `paginate'
```

Okay, so it looks to be a problem coming from Kaminari, but why? Well, if you look even further down in the stacktrace for this error for somewhere in your application, probably from the app folder you'll come across this line:

```
./app/controllers/tickets_controller.rb:60:in `search'
```

So what's so great about this line? Well, this line renders the `projects/show` view:

```
render "projects/show"
```

Above that however, is the real culprit:

```
@tickets = @project.tickets.search(params[:search])
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You're not calling either `page` or `per` on your search results, and so it's not going to be paginating them. You're going to call the same methods you called back in the `ProjectsController`'s `show` action here so that you get paginated search results.

```
@tickets = @project.tickets.search(params[:search])
@tickets = @tickets.page(params[:page]).per(50)
```

With paginated search results, the feature in features/searching.feature will no longer complain when you run it with `bin/rspec spec/integration/searching_spec.rb`:

```
3 examples, 0 failures
```

Alright, so that one's passing. Let's see what happens now when you run `rake cucumber:ok spec` again.

```
122 examples, 0 failures
```

Great, time to make a commit with this new feature.

```
git add .
git commit -m "Add pagination for tickets"
git push
```

Seeing pagination for yourself

Here you've seen an easy way to add pagination links to resources in your application by using the Kaminari gem. You could have used the will_paginate gem and that would have worked just as easily. It's really up to personal preference. Pagination allows you to ease the load on the database server by returning only limited amounts of records per page, and also doesn't overwhelm the user with choices.

Let's see how this works in a browser before we continue. First, you'll need to create a hundred tickets for a project so that you can get two pages of pagination.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To do that, launch `rails console` and put in this code:

```
project = Project.first
100.times do |i|
project.tickets.create!(
:title => "Fake ticket",
:description => "Fake description",
:user => User.first
)
end
```

Next, type `exit` and hit enter to exit out of the console, then launch your application with `rails server`. You can login using the email and password you've set up in `db/seeds.rb`, which is "admin@ticketee.com" and "password" respectively. You can then click on the "Ticketee Beta" page and you should see a page like figure 16.3.

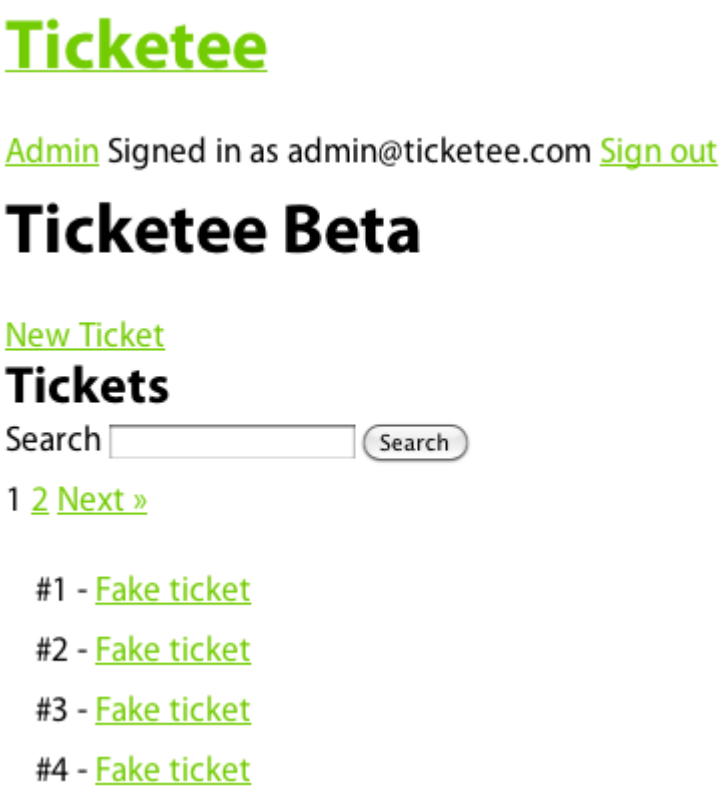

Figure 16.3 Paginated tickets

The pagination here shows that you're on the first page and that there's a second page you can go to, either by clicking the "2" link or the "Next" link. By clicking this link, the page switches to the second page of tickets and the URL now

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

becomes http://localhost:3000/projects/1?page=2. This `page` parameter is passed to the controller as `params[:page]` and then passed to the `page` method provided by Kaminari.

If you click the "1" link or the "Prev" link, you'll be taken back to the first page. All of that functionality was given to you by the `paginate` method in your views and the `page` call in your controller. You didn't have to code any of this yourself, which is great.

Next, we'll look at how you can add this same kind of pagination to the Tickets API in your API.

16.1.3 Paginating an API

You've easily set up pagination for your tickets on the interface that a user sees to ease the load on the database. However, for your tickets API you're still returning all the tickets for a project when they're requested, and therefore you'll run into the same problems you solved in the previous section.

Your API is different though. You can't provide a pagination link for the tickets returned by an API. Instead, you'll have to rely on people passing in a page number, which you'll then use to return that page of tickets.

To test this, you're going to go into your API spec file for tickets at spec/api/v2/tickets_spec.rb and you'll add another test. This one should assert that when you pass in a page parameter to your requests that you receive that page of tickets, rather than all of the tickets or a different page.

In your API you'll limit requests to 50 per response, however, you may choose to set this a little higher[4]. Therefore, you'll create 100 tickets, which should give you enough tickets to test that you can get the first and second pages of your API.

Footnote 4 200 seems to be a common number to use for API return objects per request

You'll add another `context` to spec/api/v2/tickets_spec.rb to test pagination, using the code shown in Listing 16.2.

Listing 16.2 spec/api/v2/tickets_spec.rb

```
context "pagination" do
  before do
    3.times do
      Factory(:ticket, :project => project, :user => @user)
    end

    @default_per_page = Kaminari.config.default_per_page
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    Kaminari.config.default_per_page = 1
  end

  after do
    Kaminari.config.default_per_page = @default_per_page
  end

  it "gets the first page" do
    get "/api/v2/projects/#{project.id}/tickets.json",
      :token => token,
      :page => 1

    last_response.body.should eql(project.tickets.page(1).to_json)
  end

  it "gets the second page" do
    get "/api/v2/projects/#{project.id}/tickets.json?page=2",
      :token => token,
      :page => 2

    last_response.body.should eql(project.tickets.page(2).to_json)
  end
end
```

In this new `context`, you'll create 100 tickets using the ticket factory, referencing the `@user` variable set up in the spec's `before` block and also pointing it at the `project` object set up near the top of this file. Your first test makes sure that you're getting back the first 50 tickets for the project, and the second test checks for the second 50.

When you run this test using `bin/rspec spec/api/v2/tickets_spec.rb:36`, it won't pass because you've not got the pagination in place yet:

```
expected [small array of JSON'ified tickets]
got [larger array of JSON'ified tickets]
```

You can easily fix this by changing this line in the `index` action of app/controllers/api/v2/tickets_controller.rb:

```
respond_with(@project.tickets)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

To this:

```
respond_with(@project.tickets.page(params[:page]))
```

When you rerun the pagination `context` with `bin/rspec spec/api/v2/tickets_spec.rb:35`, both tests will pass:

```
2 examples, 0 failures
```

Now users can go to `/api/v2/projects/:project_id/tickets.json` to get the first page of 50 tickets, or specify the `page` parameter by putting it on the end of the URL as a query parameter (i.e. `/api/v2/projects/:project_id/tickets.json?page=2`) to get to the second page of tickets.

You can now run `rake cucumber:ok spec` to check for any breakage:

```
124 examples, 0 failures
```

By paginating the number of tickets shown both on the interface and in the API, you can ease the load on the server and provide a better interface to your users at the same time.

Sometimes when you're coding your application you may inadvertently call queries that don't perform all that well. This could happen in a view if you were wanting to display all tags for each ticket as you iterated through them. In the next section, we take a look at how you can cause this problem to happen and at two ways to fix it.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

16.2 Database query enhancements

What would you do without database queries? Well, you'd have a boring application, that's for sure! But it's database queries that can be the biggest bottleneck for your application once it grows to a larger size. Having a page that--in the beginning--only ran five queries, and is now running 100 on each request will just not be webscale.

The most common place where performance degradation can occur in a Rails application is when an operation called "n+1 selects" takes place. Let's use your application as an example of this. Imagine that you have 50 tickets and want to display them all on the same page, but also along with these tickets you wanted to display all the tags for these tickets. Before you render this page, you know all the tickets but don't yet know what the tags are for the tickets. Therefore, you'd need to retrieve the tags as you are iterating over each of the tickets, generating another query to retrieve all the tags for each ticket.

This is the "N+1 selects" problem. You have an initial query for all of your tickets, but then `N` queries more, depending on the amount of tickets you're showing. This problem is not so much of a "big deal" now that you've got pagination, but it still can crop up.

16.2.1 Eager loading

In your app/views/projects/show.html.erb you can perform N+1 selects, asking for each ticket's tags just like in the example, by putting this line within the block where you iterate over each ticket:

```
<%= render ticket.tags %>
```

When you start your server using `rails server` and navigate to your first project's page, Rails will diligently run through each ticket in the `@tickets` array, performing a query for each one to find its tags. If you switch back over to the console, you'll see queries like this:

```
SELECT * FROM "tags"
INNER JOIN "tags_tickets" ON "tags".id = "tags_tickets".tag_id
WHERE ("tags_tickets".ticket_id = 1 )
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

There should be 50 of these little queries, and 50 adds up to a big number[5] when it comes to lots of requests hitting this page and running these queries. Fifty requests to this page would result in over 2,500 queries. Oh, your poor database server![6] It would be much better if you didn't have to run so many queries.

Footnote 5 When used in a function that uses squares, or even worse, cubes.

Footnote 6 Yes, they're made for this kind of thing, but that's not the point!

Thankfully, there's yet another thing in Rails that helps us be better programmers and better friends with our databases. This wonderful invention is known as *eager loading* and will allow you to run two queries to get all the tickets and all the tags, rather than one query for the ticket and N queries for all the tags for all the tickets.

There are two ways of doing this: you can use the `joins` or `includes` method when you attempt to grab all the tags for the tickets in app/controllers/projects_controller.rb. You're currently grabbing and paginating all the tickets for the current project using this line in the `show` action in `ProjectsController`:

```
@tickets = @project.tickets.page(params[:page])
```

The `@project.tickets` part of this line generates a query[7], but doesn't eager load the tags yet. To make it do this, you could use the `joins` method like this:

Footnote 7 But doesn't run it! When it gets to the view and you call `each` on it, then it runs.

```
@tickets = @project.tickets.joins(:tags).page(params[:page])
```

This line would generate an SQL query like this:

```
SELECT "tickets".* FROM "tickets"
INNER JOIN "tags_tickets"
ON "tags_tickets"."ticket_id" = "tickets"."id"
INNER JOIN "tags"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
ON "tags"."id" = "tags_tickets"."tag_id"
WHERE ("tickets".project_id = 1)
```

The `INNER JOIN` parts of the query here mean that it will find all records in the `tickets` table that have tags only. It will also return a ticket record for every tag that it has, so if one ticket has three tags it will return three tickets. This is somewhat of a problem, given that you're going to want to display all tickets regardless of if they are tagged or not, and you definitely don't want three of them appearing when only one should.

To fix this, use `joins` brother `includes`, switching the line in the `show` action to this:

```
@tickets = @project.tickets.includes(:tags).page(params[:page])
```

When you refresh the page, Rails will generate two queries this time around:

```
SELECT "tickets".* FROM "tickets"
WHERE ("tickets".project_id = 1)
LIMIT 50
OFFSET 0

SELECT "tags".*, t0.ticket_id as the_parent_record_id FROM "tags"
INNER JOIN "tags_tickets" t0 ON "tags".id = t0.tag_id
WHERE (t0.ticket_id IN (1,2,[...],49,50))
```

Rails has run the query to find all the tickets first, then another query to gather all the tags for all the selected tickets as the second query. This query doesn't care if tickets have tags or not, it will still fetch them.

Here you've seen a way to cause an N+1 query and how to stop it from happening. You can remove the line from app/views/projects/show.html.erb now, as you're done with this experiment.

This is just one way your database can be slow. Another is more insidious. It creeps in slowly over months of the application seemingly running fine and makes it progressively slower and slower. The problem is a lack of *database indexes*, and effects many Rails applications even today.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

16.2.2 Database indexes

Database indexes aren't a Rails feature, they're a feature of your own database that can greatly improve its performance when used correctly. The absence of database indexes may not seem like a problem immediately, but when you're dealing with larger datasets it becomes more and more of a problem. Take for example if you had 10,000 tickets with 2,300 of them belonging to Project A. To find all the tickets for Project A, your database sans indexes would have to do a *full table scan*, searching through each ticket and determining if it belonged to Project A or not. That's a problem, because the more records you have, the longer this scan is going to take.

Indexing the data in your databases allows you to perform fast lookups and avoid full table scans. Imagine that your database is a phonebook and that the names are in no particular order. In this situation, it would be difficult to find all people with a name such as "John Smith-McGee," because you'd have to scan the entire phone book to find out who has this name.

An index sorts this data into a logical order and allows for a much faster lookup. Ever seen how a phonebook that has the letter and the first name on the top left hand side, and maybe the same or a different letter on the top right-hand side, with another name? That's an index. That allows you to easily find names because you know that the letter A comes before B, and C after B and so on.

Indexes allow you to run much faster queries as you tell your database how to index the data. Although it may seem like premature optimization at this point, you're going to put an index on your `tickets` table to speed up finding collections of tickets for a project. It's common sense to have these from the beginning, because adding them onto large datasets will take a long time, as you'll need to work out how to index each record.

To add this index, create a new migration with this command:

```
rails g migration add_project_id_index_to_tickets
```

This will generate a file at db/migrate that ends with the name you've given it. You're going to need to open this file now and add in the index, as Rails cannot

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

(yet) read your mind. You'll add this index inside the `self.up` part of the migration using the `add_index` and remove it in the `self.down` method using `remove_index`, like this:

```
def change
  add_index :tickets, :project_id
end
```

Run this migration using `rake db:migrate db:test:prepare` to run it on the development and test environment databases. You'll see this line in the output:

```
-- add_index(:tickets, :project_id)
   -> 0.0015s
```

Just to reinforce the message: it's better to add the indexes when the database is first being designed, rather than at a later point because this "0.0015 seconds" could easily become whole seconds on a larger dataset. This index will now group your tickets into groups of `project_id` columns, allowing for much faster lookups to find what tickets belong to a specific project.

You want the absolute best performance you can get out of your database because it's a key point in your requests. Indexes and eager loading are the two most basic ways you can get better performance out of your database.

If your database is performing optimally and your pages still aren't loading fast enough, you'll need to look for alternative methods of speeding them up. Two of these methods are page and action caching, which allow you to store the output of a page to serve it up rather than re-processing the code and hitting the database again.

16.3 Page and action caching

Rails has several methods of caching pages. The first of these methods serves a request and then stores the output of that page in the public folder of your application so that it can be served without going through the Rails stack by the web server. This is known as *page caching*

You'd cache a page if that page took a long time to process, or if there were a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

lot of requests to it. If either of these situations happen, the performance of the web server can be degraded and requests can end up piling up.

By caching a page, you take the responsibility of processing and serving it off your Rails stack and put it on the (usually) more-than-capable web server[8].

Footnote 8 Such as Apache or Nginx, or any other HTTP server. Not Webrick. There are some things that Ruby's made for, and being a fast / stable HTTP server ain't one.

The first time a page is requested, you store it as a file in your application. The next time the request is made, that static page will be served rather than having the action processed again.

This first type of caching is great for pages that don't require authentication. For pages that *do* require authentication you'll need to use a different kind of caching called *action caching*. This type of caching runs the before filters on a request before it serves the cached page, and you'll see a great example of this in this section.

Let's take a look at the first kind of caching, plain ol' page caching.

16.3.1 Caching a page

You're going to cache the page that's rendered when a user looks at `ProjectsController`'s `show` action. By caching this particular page, Rails will serve the first request to this file and then save the output of the request to a new file at public/projects/:id.html. This public/projects directory will be created by Rails automatically. This process is shown in figure 16.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

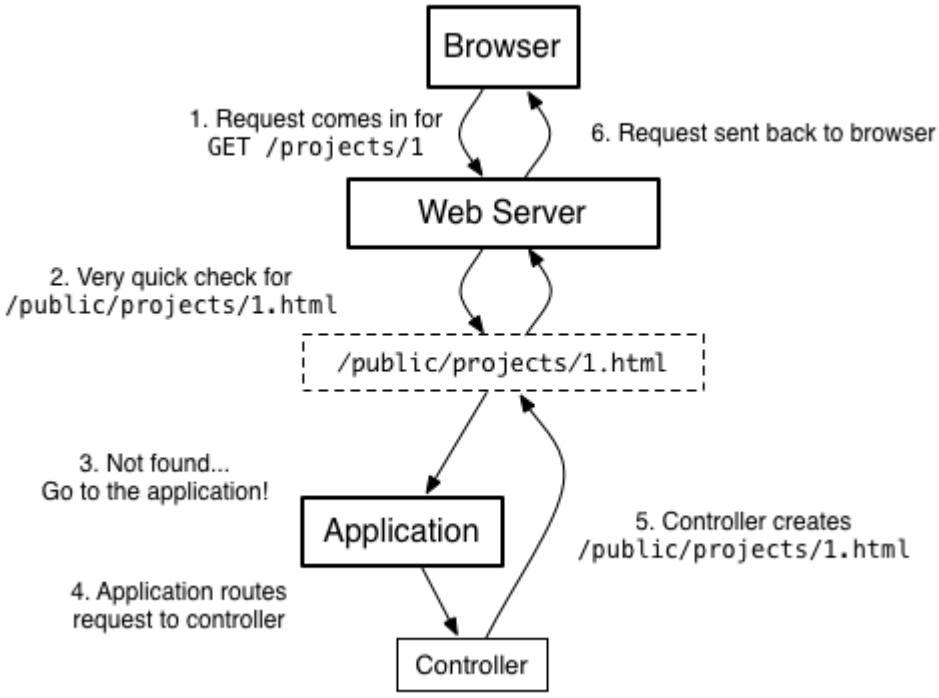

Figure 16.4 First request, no cached page

On the next request, due to how the web server is configured, it will serve the file rather than hit the Rails stack, as shown in figure 16.5.

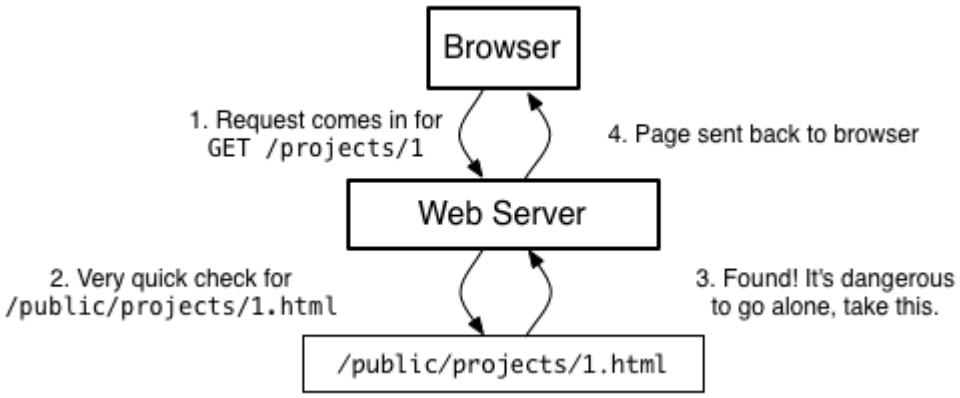

Figure 16.5 Subsequent requests, cached page

This is absolutely a faster request, regardless of how little goes on in an action in Rails. If a request doesn't have to go down that extra level in the stack it's going to save a great deal of time, and again: modern web servers are *built* to serve these static files.

One of the downsides of this is that it will not cache the GET parameter on the request, like your page numbers. Earlier when you used `rails server` to use your pagination, the URL became http://localhost:3000/projects/1?page=2. The

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

page that's cached doesn't have this parameter at the end, and so it will always display the first page, as that's what will be stored at public/projects/:id.html.

Regardless of this, you'll at least see how this method works. In your `ProjectsController`, underneath the `before_filter` lines, you can put this method to tell Rails to cache the page for the `show` action:

```
caches_page :show
```

In development mode, caching is turned off by default. Obviously, in development mode you don't care so much about caching, as all requests are going to be local and not on a heavy-load server. You can turn caching on by going into config/environments/development.rb and changing this line:

```
config.action_controller.perform_caching = false
```

To this:

```
config.action_controller.perform_caching = true
```

Without this option, you can still have `caches_page` in your controllers, it just won't do anything. With it turned on, your pages will be cached upon their first request.

Launch `rails server` again and this time go to http://localhost:3000/projects/1. In the server output, you'll see an additional line:

```
Write page /.../ticketee/public/projects/1.html (0.3ms)
```

This time, rather than simply processing your action and sending the response body back to the server, Rails will save the body in a new file in your application at public/projects/1.html. The next time this route is requested, because the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

public/projects/1.html page exists, it will be served by your web server, rather than Rails. A side-effect of this means that your request will not show up in the Rails console, but at least it will be served faster.

Let's reload the page now, it should be a little faster because it's serving that static page. If you press the next link on your pagination, you'll still be shown the first page. This is because the GET parameter was ignored, and the first page for this project's tickets was what was cached.

There's another problem too: this result is cached for *all* users of your application. At the top of the page, you'll be able to see the message that says "Signed in as admin@ticketee.com," as shown in figure 16.6.

Figure 16.6 Signed in as admin

To see this little issue in action, sign up as another user by first clicking the "Sign out" link in the application to sign up, then the "Sign up" link to be presented with a form to sign up. In this form, enter "user@ticketee.com" for the email and "password" for both the password and password confirmation fields. When you hit the "Sign up" button, this will create your new user account.

You currently require users to confirm their account through an email they receive, but because you're in development mode there will be no emails sent. To confirm this user, launch `rails console` now and run these commands:

```
user = User.find_by_email("user@ticketee.com")
user.confirm!
```

You'll also need to give this user access to the first project in your system, so that they can view the tickets too. To do this, run these couple of commands:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
project = Project.first
user.permissions.create!(:action => "view", :thing => project)
```

Alright, now that your user is confirmed and has access to this project, let's see what happens when you sign in with the email and password you used to sign up, "user@ticketee.com" and "password". At the top of the page you'll see that you're signed in as the new user, as seen in figure 16.7.

Figure 16.7 Signed in as a user

However, when you click the "Ticketee Beta" link to go to your first project, the page will change to saying that you're signed in as the "admin@ticketee.com" user again, as shown in figure 16.8.

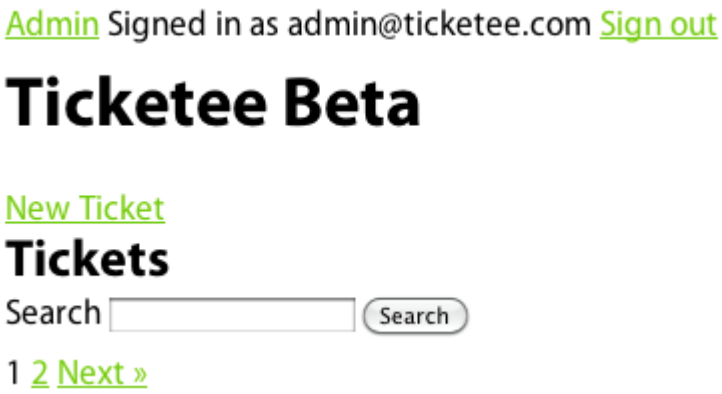

Figure 16.8 Still signed in as admin@ticketee.com?

You know better; you're actually signed in as the user! This is happening because Rails has cached the entire page, rather than just the tickets list. This page also ignores any kind of authorization you've set up in your controllers, making it available for every single person who wishes to access it, which is just a Very Bad Thing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

So it looks like that `caches_page` isn't going to work in this situation. This method is better for pages that don't have dynamic elements on them, such as the place at the top that displays the currently logged in user or the list of tickets.

This method has a brother called `caches_action` that will help you fix both the issue of the currently logged in user display message, as well as the issue of it only showing the first page of pagination.

16.3.2 Caching an action

Caching an entire page is helpful when you don't have authentication, but if you have authentication then it's better to cache the response of the action on a per-user basis. Caching an action involves caching the response for a particular session, so that when that user requests it again they'll be shown it again.

Caching a page is great for a page that's accessible by anybody, as the body would be served as a static file from the public folder by the web server. Caching an action is best used for actions that take a long time to process (you don't have any at the moment) and that require some interaction with the Rails stack, such as a `before_filter` that authenticates your user.

There's a third way of caching, and that's *fragment caching*, where you'd cache just a bit of a page at a time, rather than the entire result. Before you get on to using that, let's see what `caches_action` provides you.

NOTE	**Cleaning up after yourself** Before you do anything, you'll want to remove the old file that has been cached. To do this, delete the public/projects directory. Next time this page is requested, the cache will be recreated.

Let's replace this line in your `ProjectsController`:

```
caches_page :show
```

With this line:

```
caches_action :show
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

For this change to take effect, you only need to refresh the page at http://localhost:3000/projects/1 or actually visit it again if you've closed the browser since the last time. If you switch over to the terminal where your server is running, you won't see the line that says this:

```
Write page /.../ticketee/public/projects/1.html (0.3ms)
```

Rather, you'll see this line instead:

```
Write fragment views/localhost:3000/projects/1 (40.3ms)
```

This time, Rails has written a *fragment* rather than writing a page. In this case, the fragment is actually the entire page, but it is the page available only for this user. When you request this page again, you'll see this line in the server's output:

```
Read fragment views/localhost:3000/projects/1 (0.3ms)
```

Upon the second request here, Rails has found the fragment pertaining to this request and served that instead. Rather than saving these files into the public directory, Rails instead saves them to the tmp/cache directory. Files that are in the public directory are automatically served by your web server without hitting the Rails stack, but cached responses in tmp/cache are served by the Rails stack itself. This may seem counterintuitive at first, but it's really helpful if you want to alter what cache fragments are served to what user.

Currently, the fragment is written to a file such as tmp/cache/CC6/080/views%2Flocalhost%3A3000%2Fprojects%2F1. This location is simply a location in the tmp/cache folder with a hashed path, followed by the escaped name of `views/localhost:3000/projects/1`. It's with this name that Rails can retrieve this fragment and show it again.

But, you're still going to have the problem that both of your users are going to see the same page. Sign out of your current user and sign in as the other one. Once you visit this page again, you'll see you're still signed in as the first user! It's doing

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the same darn thing as `caches_page`!

But, as stated before, `caches_action` is different! It runs the `before_filters` of your controller and has one more special benefit: you can change the path of where this file is cached by using the `cache_path` option passed to `caches_action`. You can then set this option to be a `Proc` object, which means it will be evaluated before every request made to the action (or actions) you are caching. In this `Proc` object you'll have access to the current controller instance, meaning you'll have access to `current_user`. With this access, you'll be able to customise the path where the cache is kept so that you can cache the same page for different users.

To do this, change your `caches_action` line in your controller to these lines:

```
caches_action :show, :cache_path => (proc do
  project_path(params[:id], :user_id => current_user.id)
end)
```

Here, you've passed the `cache_path` option to `caches_action`. This is a `proc` object, and you need to wrap the value for this option in brackets otherwise Ruby will think the block is for the `caches_action` call.

This `Proc` object is evaluated within the context of an instance of this controller, and therefore you'll have access to the `params` and `current_user` methods usually available within an action or a `before_filter`. With these, you're building a string by combining the URL of the current project (provided to you by the helper `project_path`) and the `id` of `current_user`.

When you access this page again in the browser, Rails will re-process this action because the cache path has changed and then save the page in a new location. In the output for the server you'll see this new fragment has been written, indicated by this line:

```
Write fragment views/localhost:3000/projects/1/1 (4.4ms)
```

This time, the path to the file and the file itself have changed because you've

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

changed the URL of the page; it's now the cached version of this page currently for this user. When you sign out as your current user and sign in as the other user and navigate to this project's page, you'll see that the "Signed in" message at the top of the page is now the correct one, as shown in figure 16.9.

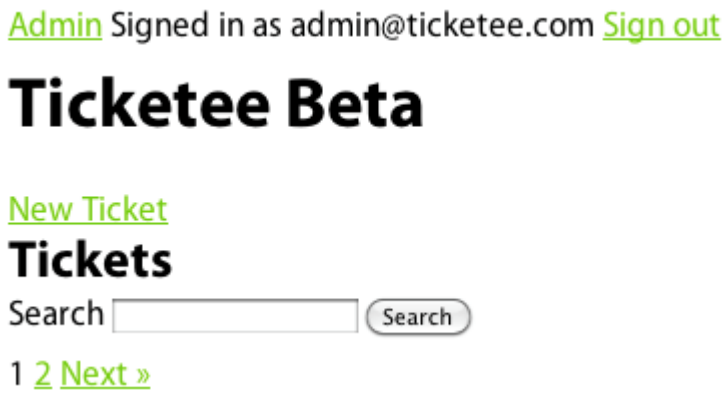

Figure 16.9 Signed in as admin for a cached page

This means that you've now fixed the problem where the same cached page was shown for all users, meaning that each of your users will see a slightly different version of this page. This is *almost* right, but not quite. When you click on the "Next" link for pagination, you'll still only be shown the first page. This is because much like `caches_page`, your `caches_action` also ignores the page parameter.

You can fix this, however, by changing the path generated for the cached page to contain the current page number. To do this, change this line in `caches_action`'s `cache_path` option in `ProjectsController`:

```
project_path(params[:id]) + "/#{current_user.id}"
```

To this:

```
project_path(params[:id]) +
"/#{current_user.id}/#{params[:page] || 1}"
```

The next time you request this page, it will again save a new version of it, this time outputting a line like this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Write fragment views/localhost:3000/projects/1/1/1
```

The first 1 here represents the project's id, the second represents the user's and the third represents the page number. This file is saved to a path such as tmp/cache/E62/3E0/views%2Flocalhost%3A3000%2Fprojects%2F1%2F1%2F1.

So in this section you've fixed the problem where all people would see that they were signed in as the first person who requested the page, as well as the case where only one page of your tickets were available. Now what happens when you update this page and the tickets change? These pages will still be cached and your new tickets or updates to them will not be shown!

You're going to need a way to clear this cache, to expire the fragments that are created when these events happen. Right now, the number one situation where that's going to happen is when you create a new ticket for a project. You can trigger this event to clear your cache by using a feature in Rails known as *cache sweepers*.

16.3.3 Cache sweepers

Cache sweepers are much like the observers you used back in chapter 11. In fact, the `ActionController::Caching::Sweeper` class *inherits* from ActiveRecord::Observer, effectively making them the same thing. The difference here is that you refer to the sweeper in the controller, telling it to run after certain actions have complete[9].

Footnote 9 It uses `after_filter` to do this, which can also be used to run other actions after a controller's action has been processed, just like a `before_filter` can be used to run actions before a controller's action runs.

In this case, whenever a ticket is created, updated or destroyed in a project, you'll want your application to clear out the cached pages because they would be out of date at that point. This is precisely what you can use a sweeper for. To call this sweeper, put this line underneath the `before_filter` calls in `TicketsController`:

```
cache_sweeper :tickets_sweeper, :only => [:create, :update, :destroy]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You put this line in your `TicketsController` because you want it to run after the `create`, `update` and `destroy` actions.

TIP **Alternatively, pass a constant**

Rather than passing the symbolized version of the name along to the `cache_sweeper` method, you can also alternatively pass along a class:

```
cache_sweeper TicketsSweeeper
```

This doesn't perform any different to when you pass in a symbol, but is really helpful if your sweeper was modularized:

```
cache_sweeper Ticketee::TicketsSweeper
```

You can't pass a modularized sweeper name as a symbol, and so the `cache_sweeper` method supports passing both a symbol and a constant reference as well.

Now when you go to a project in your application and attempt to create a new ticket on it, you'll get this error:

```
uninitialized constant TicketsSweeper
```

Rails is looking for the `TicketsSweeeper` constant, which is supposed to define the cache sweeping behaviour for your `TicketsController`, but can't find it because you haven't defined it yet. To define this, create a new folder at app/sweepers for this sweeper and its brethren to live[10]. In this directory you'll create a new file called app/sweepers/tickets_sweeper.rb and fill it with this content:

Footnote 10 Because it doesn't really belong in the controllers, helpers, models, observers or views directory, but is still a vital part of your application.

```
class TicketsSweeper < ActionController::Caching::Sweeper
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  observe Ticket
  def after_create(ticket)
    # expire fragment code goes here
  end
end
```

You'll get around to putting the expire fragment code in just a bit, but first a bit of explanation is needed. A sweeper looks and acts in much the same way as an observer. By calling the `observe` method at the top of the `TicketsSweeper`, you tell this sweeper to watch the `Ticket` class for changes. The `after_create` method here will be called after creation of a new `Ticket` object, but because you're in a sweeper, you'll have access to the controller's parameters also. With them, you can use what's usually available in the controller to expire the cached fragments.

To do this, you can call the `expire_fragment` method, passing it a regular expression. This regular expression will match all cached fragments for the ticket's project for all users, effectively wiping clean the slate for this project in terms of cached pages. Inside your `after_create` method you'll put this:

```
expire_fragment(/projects\/#{ticket.project.id}\/.*?/)
```

Now when you create a new ticket for a project, this `expire_fragment` method will be called. Let's try this out now, creating a new ticket by clicking the "New Ticket" link on a project's page and filling out the form. Once you've pressed the "Create Ticket" button on the form, you'll see this in the console:

```
Expire fragment (?-mix:projects\/1\/.*?) (327.3ms)
```

Rails has now gone through and expired all the fragments associated with this ticket's project. If you now go into tmp/cache and into any one of the directories there looking for a file, you shouldn't see any. The directories (with names like "E62" and "3E0") will still exist, but there aren't any files. This means that Rails has successfully cleared its cache of fragments for the project.

Let's get your sweeper to perform this same action when tickets are updated and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

destroyed. Move the `expire_fragment` call into another method and then call it in the `after_create`, `after_update` and `after_destroy` methods in `TicketsSweeper` using the code shown in Listing 16.3.

Listing 16.3 app/sweepers/tickets_sweeper.rb

```
class TicketsSweeper < ActionController::Caching::Sweeper
  observe Ticket
  def after_create(ticket)
    expire_fragments_for_project(ticket.project)
  end

  def after_update(ticket)
    expire_fragments_for_project(ticket.project)
  end

  def after_destroy(ticket)
    expire_fragments_for_project(ticket.project)
  end

  private

    def expire_fragments_for_project(project)
       expire_fragment(/projects\/#{project.id}\/.*?/)
    end
end
```

Now you have Rails caching the pages of tickets for all projects in your application and clearing that cache when tickets are updated. This is a great demonstration of caching on a per-user basis, even if your project page isn't that intensive. If you had a system resource (CPU / memory) intensive action in your application that required user customization like this, you could use this same method to cache that action to stop it from being hit so often, which would reduce the strain on your server.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

TIP **Expiring pages**

If you were still using `caches_page`, you wouldn't use `expire_fragment` to expire the cache files that were generated. Instead, you'd use `expire_page`, which can take a hash like this:

```
expire_page(:controller => "projects",
:action => "show",
:id => 1)
```

Or, better still would be to pass it the URL helper:

```
expire_page(project_path(1))
```

Even though you're not caching pages any more, it's still handy to know how to clear cached pages and fragments.

Let's make a commit now for this:

```
git add .
git commit -m "Add fragment caching to ticket listings on a project"
```

Another way to ease the load on the server side is to use the browser (client) side caching by sending back a `304 Not Modified` status from your Rails application. In the next section, we'll look at a Rails controller method that'll help you with this.

16.3.4 Client-side caching

There's one more method in the controller you're going to see in this section, and that's the `fresh_when` method. This method will send an `ETag`[11] header back with the initial request to a client and then the client's browser will cache that page with that ETag on the client's machine[12]. The ETag is the unique identifier for this page, or "entity," at the current point in time.

Footnote 11 The "E" stands for entity. More information is available on the Wikipedia page for this: http://en.wikipedia.org/wiki/HTTP_ETag

Footnote 12 If "Private Browsing" is turned on in the browser, this wouldn't happen.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

In this situation, you'll use this type of caching for the `show` action on a ticket in a project, meaning the URL will be something like /projects/1/tickets/2. The first request to this action after you're done will follow the steps shown in figure 16.10.

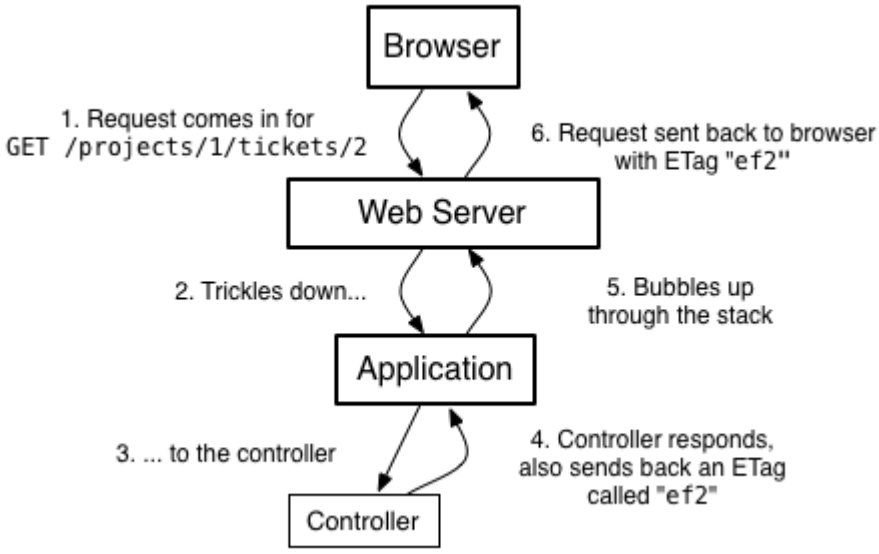

Figure 16.10 E-Tag caching

The next time the page is requested, the browser will send a request with the E-Tag it received on the first request to the server, this time in a `If-None-Match` header. The server then re-generates the E-Tag for the page that's been requested and compares it against the `If-None-Match` incoming header. If these two match, then the server will send back a `304 Not Modified` header, telling the browser to use its cached copy. This means that, rather than having the server re-render the view and its pieces, the client does all the hard work of just re-showing the initial page. This whole process is shown in figure 16.11.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

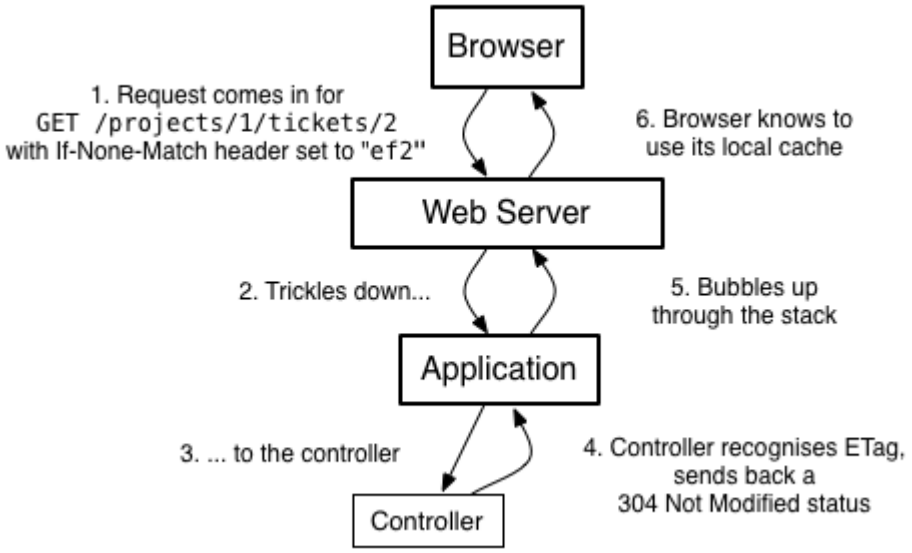

Figure 16.11 304 Not Modified response

Even though this goes through the same series of events both times, what happens in the controller is the clincher: by returning a `304 Not Modified`, you can respond with a lightweight response and get the user's browser to render the page, rather than having your application do it again.

For your ticket page, you're going to want your application to send back this status only when your ticket hasn't been updated. When a ticket's information, such as the title or description are updated, or when a comment is posted to the ticket, you'd want to send back a proper response rather than the `304 Not Modified` header. It's this timestamp that you're going to be using to determine if a page is either fresh or stale. A fresh page is one that's been recently updated, with a stale one being one that hasn't been.

You've got a column in your `tickets` table that you can use to determine if a ticket's been updated: the `updated_at` column. Each time a ticket's updated through your application, this field will be set to the timestamp automatically. But, when a comment is posted to the ticket, the `updated_at` field for the ticket will remain the same.

To fix this problem, you can configure the `Comment` model to touch the ticket object it's related to, which will update its `updated_at` timestamp. The way you do this is with an option on the `belongs_to` association in `Comment` called `touch`. Let's change the `belongs_to :ticket` line currently in app/models/comment.rb to this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
belongs_to :ticket, :touch => true
```

Whenever a comment is updated, created, or even destroyed, the related ticket's `updated_at` attribute will be updated. With the touch option, you can now confidently use this attribute to provide a reliable timestamp for your new form of caching. This particular form of caching uses a new method in your controllers called `fresh_when`.

To make the `show` action in `TicketsController` conditionally send back the `304 Not Modified`, put this at the bottom of the `show` method in app/controllers/tickets_controller.rb:

```
fresh_when :last_modified => @ticket.updated_at,
           :etag => @ticket.to_s + current_user.id.to_s
```

The `last_modified` option here sends back another header to the client: the `Last-Modified` header. This header is used by a browser to detect when the page was last updated, which provides a near-identical purpose to an ETag. A browser sends an `If-Modified-Since` header that contains the last `Last-Modified` time. If the server sees that the `Last-Modified` time is later than the `If-Modified-Since`, it will send a new copy of the page. Otherwise, it will send a `304 Not Modified` header.

The `:etag` option tells `fresh_when` to generate a new ETag for the resource. Until this resource changes, the ETag generated will be the same for each user. This wouldn't be the case if you didn't pass through the `current_user.id.to_s` to the ETag, but only for two user accounts accessed on the same computer. By using the `current_user`'s `id` attribute to seed the `etag` option, the tag will be different between users. How this ETag is generated differs from implementation to implementation; in Rails it's an MD5 hash, which is guaranteed uniqueness.

Even though these two options are nearly identical, some browsers may support one or the other. It's more of a way to cover your bases to pass through both headers, and it's a worthwhile thing to cover.

You can see this in action now if you attempt to visit a ticket's page. Your first

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

request will have a final line that says something like this:

```
Completed 200 OK in 486ms (Views: 200.4ms | ActiveRecord: 5.6ms)
```

In this instance, the views have been rendered and the entire procedure has taken 486ms. Rather than refreshing the page (because in some browsers, this triggers them to *not* send the `If-Modified-Since` or `If-None-Match` headers), you'll go back to the project's page and then click back on the same ticket again. This time in the server output you'll see this output:

```
Completed 304 Not Modified in 267ms
```

The server has sent back a `304 Not Modified` response in a slightly quicker time than your original request, mainly because it didn't have to re-render the views for the application and send back all that HTML.

This is another way to ease the load on your server, by getting the browser to deal with the page caching and serving, rather than the server.

That wraps up this section. You've made a small change here you should probably commit. You can do that by typing these commands into the terminal:

```
git add .
git commit -m "Add ETag and Last-Modified
               support to ticket show page"
```

You've now seen many different flavors of controller caching, ranging from caching pages and caching actions (actually, fragments), to getting the browser to take care of the hard part of the caching process (storing a file and expiring it). All of these caching methods deal with caching entire pages, so what's a Railer supposed to do if they want to cache only a bit of a page at a time? For that, you can tell Rails to cache just these parts using an aptly-named method: `cache`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

16.3.5 Caching page fragments

If part of a page takes a long time to render, then that's a problem. To fix this kind of problem, you can use fragment caching, which allows you to cache fragments of pages using the `cache` method in your views where appropriate. This method takes a block, like this:

```
<% cache do %>
  # some horribly long and complex thing
<% end %>
```

This way when Rails attempts to load the page and comes across your `cache` call, it will check to see if there's an available fragment for it otherwise will perform the code inside the block and then store it in tmp/cache, just like `caches_action` does for an entire page.

You don't have an actual use-case for this in your application at the moment, but you'll still use it just to see what it does. You're going to be using it back on the app/views/projects/show.html.erb view, meaning you're going to want to temporarily disable `caches_action` in `ProjectsController` for this action so that it doesn't cache the page before `cache` has a chance to run. You can do this by simply removing the lines in `ProjectsController`:

```
# caches_action :show, :cache_path => (proc do
#   project_path(params[:id]) +
#   "/#{current_user.id}/#{params[:page] || 1}"
# end)
```

In the app/views/projects/show.html.erb, the primary content that's going to be changing is the list of tickets, and so you'll want to cache that and leave out the rest. To do this, you'll wrap the whole list of tickets, including the pagination link above it, in a `cache` block, as shown in Listing 16.4

Listing 16.4 app/views/projects/show.html.erb

```
<% cache do %>
  <%= paginate @tickets %>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  <ul id='tickets'>
    <% @tickets.each do |ticket| %>
      <li>
        <%= render ticket.state if ticket.state %>
        #<%= ticket.id %> -
         <%= link_to ticket.title, [@project, ticket] %>
      </li>
    <% end %>
  </ul>
<% end %>
```

The next time you reload this page in your browser, you'll see this line in your server's output:

```
Write fragment views/localhost:3000/projects/1 (3.0ms)
```

Look familiar? It's exactly the same output generated by `caches_action`. The `cache` method that you just used assumes that it's only being used once per page and so will save it with the same path (more commonly referred to as the "cache key"). We had a problem with this initially, didn't we?

Yes, we did. It was saving the page name just fine, but it didn't care if you were on your first page of pagination or the last, it was always showing the first cached page. If you click on the "Next" link on your pagination, you'll find that you've regressed this behaviour accidentally. Not to worry, this is easy to fix. You need to tell your `cache` method that there's more than one type of this page. You can do that by passing a string containing the page number to the method to give it a unique name, or key. By making this key unique for each page, Rails will cache a list of tickets for each page rather than one for all.

To fix this, change the `cache` call in your app/views/projects/show.html.erb file to this:

```
<% cache "projects/#{@project.id}/#{params[:page] || 1}" do %>
```

When you refresh this page and switch back into the terminal where your server is running, you'll see this line output:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Write fragment views/projects/1/1 (3.3ms)
```

You've specified the key that the cache now uses to store the fragment and so you'll see that it's saved it as "views/projects/1/1" now, with the first 1 being the ID of your project and the second one being the page number. If you create, update or delete a ticket now, you'll see that this fragment gets cleared away.

```
Expire fragment (?-mix:projects\/1\/.*?) (1.9ms)
```

The next time you revisit the project's page you'll see that it rewrites the fragment again:

```
Write fragment views/projects/1/1 (1.5ms)
```

In this section, you've seen that fragment caching is useful for not only caching dynamic actions with `caches_action`, but also for caching small chunks of pages by using the `cache` method. The latter allowed you to cache a small fragment of the page rather than the entire page, which is great if you have a small chunk of the page that takes a long time to render. You didn't, but it's always good to know what tools are available if you come up against this particular beast.

With the `cache` method in the view, you don't have to set the `cache_path` for the user because you're only caching the part of the page that is user-agnostic. Everything else in either the layout or elsewhere in this view would be processed each time the page is requested, but the part you have cached will be retrieved from that cache and added to the output, rather than re-processed. All in all, this solution is more elegant than `caches_action`. Another commit is in order!

```
git add .
git commit -m "Implement tidier caching for the tickets
               list on the projects page"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

That covers all the major methods for basic caching in controllers and views. You've seen ways to cache entire pages and parts of pages as cached files on the filesystem. In a Rails application there may be a lot of reading and writing to the filesystem, which can cause degradation of performance, so storing these files on the filesystem may not be the best idea. A speedier way of doing this would be to store these files in memory by switching the cache store that Rails uses. You can do this by putting this line in one of your config/environments files, probably production.rb:

```
config.action_controller.cache_store = :memory_store
```

Rather than storing the fragments on the file system, Rails will now store them in memory along with the code for the application. The retrieval time is faster here, but comes at the cost losing the cache if the server was ever stopped. If you want something more persistent, you may choose to use either Memcached (http://memcached.org) or Redis (http://redis.io). We won't go into these in this chapter, as they exceed the boundaries of what would be considered "basic" performance enhancements.

In this section you've learned how to use view fragment caching to store parts of the view that may take a long time to process. This type of caching would store these fragments in the tmp/cache directory; they can be retrieved later on.

16.4 Background workers

There are other situations where requests can be slow for your application too. One of these cases would be if a ticket had a large number of watchers and a comment was posted to that ticket. The reason for this slowdown would be because Rails would have to iterate through all the watchers and send out the update notification email to each of them individually, using the feature that you developed in chapter 12.

Rather than having a user make the request to create a comment in the application, having the server process the email notifications, and then send a response back, you can take the long-running task of sending these emails and move it into a job that runs in a background.

This will work by having your `CommentObserver` add the task of sending these emails to a job queue that runs in the background. You'll then have a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

background process separate from your application that will run these jobs as it receives them. This way, the hard work is done behind the scenes and the user receives the request back almost as if nothing of consequence happened.

To make this happen, you'll use a gem called `delayed_job`. This gem will allow you to create a table in your database where the jobs that the background worker needs to work off will be stored. The gem will also provide you with the ability to start a worker process. To add this to your application you'll put this line in your Gemfile:

```
gem 'delayed_job'
```

Then you'll need to run `bundle install` to install it. Once you're done there, you can run this command, which will generate a migration to create the `delayed_jobs` table:

```
rails g delayed_job
```

You can now run this migration with `rake db:migrate db:test:prepare`. That's all that's needed to set up the gem itself.

Your next task is to create a job. A job is any object that responds to `perform`. This method needs to perform the action of sending out the email to all the watchers of the ticket, which is currently the responsibility of the `after_create` method in `CommentObserver`, which uses this code.

```
watchers = comment.ticket.watchers - [comment.user]
watchers.each do |user|
  Notifier.comment_updated(comment, user).deliver
end
```

You'll take this code out of the `after_create` method and replace it with code to enqueue your job to be performed, using a method given to you by the `delayed_job` gem:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
Delayed::Job.enqueue CommentNotifierJob.new(comment.id)
```

The `CommentNotifierJob` class here will actually be a `Struct` object. You can create the code by first creating a new directory called app/jobs and then a new file in it called comment_notifier_job.rb, using the code you stole from the `after_create` method as shown in Listing 16.5

Listing 16.5 app/jobs/comment_notifier_job.rb

```
class CommentNotifierJob < Struct.new(:comment_id)
  def perform
    comment = Comment.find(comment_id)
    watchers = comment.ticket.watchers - [comment.user]
    watchers.each do |user|
      Notifier.comment_updated(comment, user).deliver
    end
  end
end
```

In the `perform` method here, you find the comment based on the `comment_id` and then iterate through all the watchers of the comment's ticket who are not the commenter themselves, sending them each an email that the ticket has been updated with a new comment.

By enqueueing this job using the `Delayed::Job.enqueue` method, the `delayed_job` gem will store a *marshalled* format (actually a YAML string) of this object in the table, such as this:

```
--- !ruby/struct:CommentNotifierJob \ncomment_id: 1\n
```

When a worker reads this row, it will convert this marshalled object back into a real object and then call the `perform` method on it. The reason for making another class and using a `Struct` over using one such as the `Comment` is that a `Struct` object will always be lighter than a full-on class that inherits from `ActiveRecord::Base`. If you enqueued a `Comment` object instead, the result would be this:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
"--- !ruby/ActiveRecord:Comment  ...
```

This contains a lot of useless information that you don't care about when you're enqueuing the job, and so you should not use it. When enqueuing jobs, you should always try for the lightest possible solution so that the job is queued quickly.

Now when a comment is created, a job will be enqueued to notify the watchers of the relevant ticket. This job is actually a record in a table called `delayed_jobs` that the worker reads from, running each job one at a time and working them off the queue. When there are no more jobs, it will simply wait.

To make sure that this is working, you're going to write a test for it. The test should check that a job is enqueued when a comment is created and that the watchers of the comment's ticket are notified by email when the job is run. Primarily, this test will check the `perform` method in the `Comment` model, and so you'll put it in spec/models/comment_spec.rb, using the code shown in Listing 16.6

Listing 16.6 spec/models/comment_spec.rb

```
require 'spec_helper'

describe Comment do
  let(:user) { Factory(:user) }

  before do
    @ticket = Factory(:ticket)
    @ticket.watchers << user
  end

  it "notifies people through a delayed job" do
    Delayed::Job.count.should eql(0)
    ticket.comments.create!(:text =&gt; "This is a comment",
                            :user =&gt; ticket.user)
    Delayed::Job.count.should eql(1)

    Delayed::Worker.new.work_off!
    Delayed::Job.count.should eql(0)

    email = ActionMailer::Base.deliveries.last
    email.to.should eql(user.email)
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

At the beginning of the `describe Comment` block, you set up a user who will be the one to watch the ticket that you set up in the `before` block.

In the test itself you make reference to a `Delayed::Job` class, which is actually a model provided by the `delayed_job` gem which connects to the `delayed_jobs` table. You call `count` first up and make sure that's 0 because you don't want any jobs in the table before comments exist.

Next, you create a comment for the ticket, making it originate from the creator of the ticket (`ticket.user`). This way, you can be sure that the user you set up with the `let` block will receive the notification. After the comment has been created, there should be exactly one job in the table.

You then call `Delayed::Worker.new.work_off(1)` to create a new `Delayed::Worker` instance that will work off a single job on the queue and then finish[13]. When it's done, there will be no more jobs in the queue.

Footnote 13 The default of this method is 100 jobs

Finally, you check that the last email sent out (by referencing `ActionMailer::Base.deliveries`, which stores the emails that have been sent but only in the `test` environment) has gone to the user who should have been notified, indicating that the job has run successfully.

This test should pass automatically because you've already implemented the feature. You can see this by running `bin/rspec spec/model/comment_spec.rb`:

```
1 example, 0 failures
```

Great! Now when a comment is created it should be created at the same speed, independent of the number of watchers on a ticket. Although the number of watchers on a ticket would have to reach a high number before a problem like this would arise, it is still a perfect example of how you can use `delayed_job` to queue jobs in the background.

One final thing. You've seen how you can enqueue the jobs and work them off using the `Delayed::Worker#work_off` method, but that isn't quite the way you'd do it in the real world or in a production environment. There, you'd run a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

command like this:

```
script/delayed_job start
```

This command will start a single delayed job worker[14], which will check the database every five seconds for jobs and work them off as they come in. However, there is no monitoring in place for this and so it is advisable that a tool such as Monit or God is used to monitor this process and restart it if it happens to go down.

Footnote 14 Watch out: this loads the entire Rails environment again. On a low-memory system a large number of Rails instances and job workers can suck up all the RAM of the system. It is advised to take care when deciding how many of each process is running on a machine. If this is outside the bounds of the system, then perhaps it is time to upgrade.

You can stop this job runner by using this command:

```
script/delayed_job stop
```

If you're using `delayed_job` extensively, you may wish to start more than one worker, which you can do by passing in the `-n` option to the command, like this:

```
script/delayed_job -n 2 start
```

This particular example will start two workers rather than one. For more examples on how to use this gem, check out the README on https://github.com/collectiveidea/delayed_job.

That does it for background jobs. You've learned how to take things that could potentially slow down a request and move them into the background, allowing the Rails application to continue serving the request.

Let's make a commit for these changes now:

```
git add .
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
git commit -m "Ticket notifications are now a background job"
git push
```

Now you're done!

16.5 Summary

In this chapter you learned how to implement small, easy changes that help your application perform faster, beginning with pagination and ending with view fragment caching and delayed jobs.

By using pagination, you're able to lighten the load on the database by retrieving smaller sets of records at a time. This is the easiest way to lessen the load on your application's infrastructure.

Database queries are often the bottleneck in the application because they may inadvertently be performed in excessive amounts, or they may not be indexed in the correct manner. You saw in the beginning how to implement eager loading for your queries so that rather than doing more requests than necessary, Rails will load all the necessary objects in a second, separate query.

The second way to improve database performance is to use an index similar to the page titles in a phonebook, but for a database. If you had a large number of records in your database, the index would allow for speed increases in the lookups for records for that index.

If your database speed can't be enhanced any further, then the next stop is caching the resulting pages from your actions. You first attempted to use `caches_page` but found that it came with a couple of problems: the page was available to all users regardless of their authorization, showing "Signed in as x" where "x" was the first user who requested the page, and it completely ignored your page parameter. So you moved on to the `caches_action` method, which allowed you to pass an option called `cache_path` to define where your file was saved.

Then you learned that you can cache specific parts of a view using the simply-named `cache` method in them. This saves fragments of a view into the tmp/cache directory, allowing you to store the result of potentially computationally expensive parts of your view.

These are the basic concepts for enhancing the performance of your application. There is more you can do, like integrating with tools such as Memcached (http://memcached.org) or Redis (http://redis.io), and interacting with the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`Rails.cache` variable which gives you fine-grained control over the cache that Rails uses to store fragments and can be used to store other pieces of information.

For more information, we would recommend reading the official caching guide at http://guides.rubyonrails.org/caching_with_rails.html

Index Terms

ActionController::Caching::Sweeper
belongs_to, :touch option
cache_sweeper
Cache fragment
cache method
delayed_job gem
Delayed::Job
Delayed::Worker
Eager loading
ETags
fresh_when
includes
Kaminari
observe method
Pagination

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

17 Rack-based Applications

So far, this book has primarily focused on how to work with pieces of the Rails framework, such as application and engines. In this chapter, we'll look at how you can use Rack-based applications to respond more quickly than what you'd otherwise be capable of with your main application.

Rack is the underlying web server framework that powers the underlying request/response cycle found in Rails, but it isn't a part of Rails itself. It's completely separate, with Rails requiring the parts of Rack it needs. When your application is running, it's running through a web server. When your web server receives a request it will pass it off to Rack, as shown in Figure 18.1

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

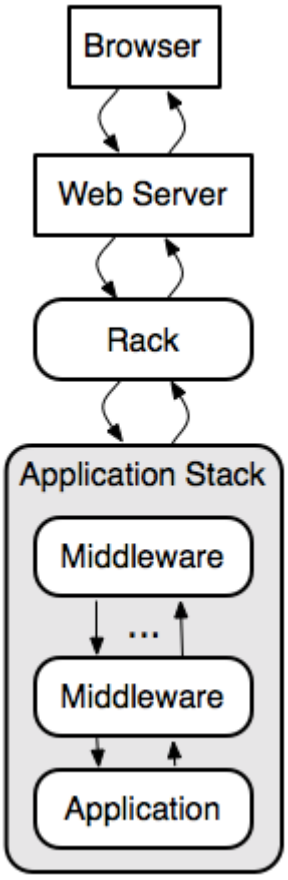

Figure 17.1 Application request through the stack

Rack then determines where to route this request, and in this case it has chosen to route to a specific application stack. The request passes through a series of pieces called *middleware* (covered in the final section of this chapter) before arriving at the application itself. The application will then generate a response and pass it back up through the stack to Rack, and then Rack will pass it back to the server, which will finally pass it back to the browser. All of this happens in a lightning quick fashion.

Separating Rack from Rails not only reduces bloat in the framework, but also provides a common interface that other frameworks can use. By standardizing the request/response cycle, applications that are built on top of Rack can interact with one another. In this chapter, you'll see how you can do this by making your Rails application work with applications built using Rack, but not Rails.

You'll build some Rack applications in this chapter that aren't Rails applications but will work just as seamlessly. You'll learn how the Rack provides the request/response cycle underneath Rails and other Ruby frameworks, and learn

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

how to build your own small, lightweight Rack-based applications.

With these lightweight applications crafted, you'll then create one more application that will re-implement the Tickets API functionality you first created in chapter 13, using another Rack-based web framework called Sinatra. You'll then mount this Sinatra application inside your Rails application using methods that Rails provides. This will provide an apt example of how you're able to interact with classes from your Rails application from within a mounted Rack application.

Finally, we'll take a look at Middleware within both the Rack and Rails stacks, and you'll learn how to use it to your advantage to manipulate requests coming into your application.

All Rack-based applications work the same way. You request a URL from the application and it sends back a response. But it's what goes on between that request and the response that's the most interesting part. Let's create a basic Rack application now so that you can understand the basics.

17.1 Building Rack applications

Rack standardizes the way an application receives requests across all the Ruby frameworks. With this standardization, you know that any application purporting to be a Rack application is going to have a standard way for you to send requests to it and a standard way of receiving responses.

You're going to build a basic Rack application so that you can learn about the underlying architecture for requests and responses found in Rails and other Ruby frameworks. With this knowledge, you'll be able to build lightweight Rack applications that you can hook into your Rails stack, or even Rack middleware.

When you're content with the first application, you'll create another and then make them work together as one big application. First things first, though.

17.1.1 A basic Rack application

To build a basic Rack application, you only need to have an object in Ruby that responds to the `call` method. That `call` method needs to take one argument (the request) and also needs to return a three-element `Array` object. This array represents the response that will be given back to Rack, and looks something like this:

```
[200, { "Content-Type": "text/plain"}, ["Hello World"]]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

The first element in this response array is the status code for your response. In this case, it's `200`, which represents a successful response. You had a bit of a play with status codes back in chapter 13 when you were building your API, so these should be no mystery at this point.

The second element in this array are the headers that will be sent back. These headers are used by the browser to determine how to deal with the response. In this case, the response will be rendered as-is to the page because the `Content-Type` header is `text/plain`, indicating normal text with no formatting applied. Usually your Rack application would set this to `text/html` to indicate an HTML response.

Finally, the third element represents the response body, which is sent back along with the status code and headers to Rack. Rack then compiles it all into an HTTP response, which is sent back to where the request came from.

Let's see this in action now. You're going to create a light Rack application that responds with "Hello World" whenever it receives a request. This kind of application is often used to check and see if a server is still up and responding to HTTP calls. You'll create a new file inside your Ticketee's application's lib called lib/heartbeat.ru (you're checking the "heartbeat" of the server) and fill it with this content:

```
run lambda { |env| [200, {'Content-Type': 'text/plain'}, ['OK']] }
```

The `.ru` extension for this file represents a Rack configuration file, also known as a "Rackup" file. In it, you call the `run` method, which needs an object that responds to `call`. When Rack receives a request to this application it will call the `call` method on the object passed to `run`, which will then generate and return a response back to the server. The object in this case is a `lambda` (or `Proc`) object, which automatically responds to `call`.

When the `call` method is called on this `lambda`, it will respond with the three-element array inside it, completely ignoring the `env` object that is passed through. Inside this array, you have the three elements Rack needs: the HTTP status, the headers for the response, and the body to return.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

NOTE **You already have a Rackup file**

Your Rails application also has one of these .ru files, called config.ru, which is used by Rack-based servers to run your application. You can see this in action by running the `rackup config.ru` command, which will start up your application using the config.ru file's configuration.

If you look in this file, you'll see these lines:

```
# This file is used by Rack-based servers to start the application.
require ::File.expand_path('../config/environment',  __FILE__)
run Ticketee::Application
```

The first line requires config/environment.rb of the application which is responsible for setting up the environment of the application. Then it uses the `run` method—just as you are—except it's passing `Ticketee::Application` which actually responds to `call`.

Cool stuff.

To see your lib/heartbeat.ru in action, you can launch a Rack server by using the command you saw in the above note:

```
rackup lib/heartbeat.ru
```

This is now running a server on 9292 (the standard port for Rack) using the built-in-to-Ruby WEBrick HTTP server, as indicated by the server output you'll see:

```
[timestamp] INFO  WEBrick 1.3.1
...
[timestamp] INFO  WEBrick::HTTPServer#start: pid=... port=9292
```

You can now go to your browser and open http://localhost:9292 to make a request to this application. You'll get back "Hello World", and that's okay with us.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You can also make a request to any path at the http://localhost:9292 application and it will respond in the same way, such as http://localhost:9292/status.

What you've done here is write one of the simplest Rack applications possible. This application receives a response to any path using any method, and always responds with `OK`. This application will respond very quickly because it hasn't loaded anything, but at the cost of being a one-trick pony.

You can make this little application respond differently in a number of ways. The easiest (and most fun!) would be to program it to change its response depending on the path it's given, like a Rails application does. For this, you'll use the `env` object. First up, let's see what this `env` object gives you by changing your little script to do this:

```
require 'yaml'
run lambda { |env| [200,
  {'Content-Type': 'text/plain'},
  ["#{env.to_yaml}"]]
}
```

The `to_yaml` method provided by the `yaml` standard library file will transform your `env` object (spoilers: it's a `Hash`) into a human-readable YAML output (like that found in config/database.yml in a Rails application).

To make this new change apply you can't refresh the page like you would in a Rails application, you have to stop the server and start it again. You can press Control+C to stop it and rerun `rackup lib/heartbeat.ru`. This time when you go to your server, you'll see output that looks like this:

```
---
GATEWAY_INTERFACE: CGI/1.1
PATH_INFO: /
QUERY_STRING: ""
REMOTE_ADDR: 127.0.0.1
REQUEST_METHOD: GET
REQUEST_URI: http://localhost:9292/
...
```

This output is the YAML-ized version of the `env` hash, which comes from Rack itself. Rack parses the incoming request and provides this `env` hash so that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you can determine how you'd like to respond to the request. You can alter the behavior of the request using any one of the keys in this hash[1], but in this case you'll keep it simple and use the `PATH_INFO` key.

Footnote 1 Yes, even the `HTTP_USER_AGENT` key to send users of a certain browser elsewhere.

A `lambda` is great for one-liners, but now your Rack application is going to become more complex, and so you've probably outgrown the usefulness of a `lambda`. You don't have to use a `lambda` though, you only need to pass `run` an object that has a `call` method that responds with that three-element array. Your new code will be a couple of lines long, and so it's probably best to define it as a method (called `call`) on an object, and what better object to define it on than a class?

A class object would allow you to define other methods, and can be used to abstract chunks of the `call` method as well. For good measure, let's call this class `Application` and put it inside a module called `Heartbeat`, replacing the content of lib/heartbeat.ru as shown in Listing 18.1.

Listing 17.1 A classy Rack application

```
module Heartbeat
  class Application
    def self.call(env)
      [200, {'Content-Type': 'text/plain'}, ["Classy!"]
    end
  end
end

run Heartbeat::Application
```

Here you've defined the `Heartbeat::Application` to have a `call` method, which once again returns "OK" for any request. On the final line, call `run` and pass in `Heartbeat::Application`, which will work like your first example because `Heartbeat::Application` has a `call` method defined on it. If this looks familiar, it's because there's a similar looking line in your application's config.ru file that you saw earlier:

```
run Ticketee::Application
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Your Rails application is actually a Rack-based application! Of course, there's a little bit more that goes on behind the scenes in your Rails application than in your Rack application at the moment, but the two are used identically. They both respond in nearly-identical ways with the three-element response array. Your Rack application is nearly the simplest form you can have. If you restart it and make a request to it, you'll see it output "Classy!"

Let's change your Heartbeat application now to respond differently to different request paths by referencing the `PATH_INFO` key within `env`. You'll now replace the code inside your `call` method with this:

```
def self.call(env)
  default_headers = { "Content-Type": "text/plain"}

  if env["PATH_INFO"] =~ /200/ <co id="ch18_90_1"/>
    body = "Success!"
    status = 200
  else
    body = "Failure!"
    status = 500 <co id="ch18_90_2"/>
  end

  [status, default_headers, ["#{env["PATH_INFO"]} == #{body}"]]
end
```

The `env["PATH_INFO"]` here returns the path that has been requested. If you made a request like http://localhost:9292/books to your Rack application, this variable would return `/books`. You compare this string to a regular expression using the `=~` operator and if it contains `200` you'll return "Success" in the body along with an HTTP status of `200`. For everything else, it's "Failure" with an HTTP status of 500.

Let's restart the server once again and then make a new request to http://localhost:9292. You'll see this output:

```
/ == Failure!
```

This is because for any request to this server that doesn't have 200 in it, you're

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

returning this message. If you make a request to http://localhost:9292/200 or even http://localhost:9292/the/200/page, you'll see the success message instead:

```
/the/200/page == Success!
```

Also, if you look in the console you can see a single line for each request that's been served:

```
127.0.0.1 - - [[timestamp]] "GET / HTTP/1.1" 500 - 0.0004
127.0.0.1 - - [[timestamp]] "GET /200 HTTP/1.1" 200 - 0.0004
127.0.0.1 - - [[timestamp]] "GET /the/200/page HTTP/1.1" 200 - 0.0004
```

This output shows the IP where the request came from, the local time the request happened, the request itself, the HTTP status contained within the response, and finally how long the page took to run. For the first request, it returned a 500 HTTP status and for the other two requests that contained "200" in their paths, it returned a 200 HTTP status.

What you've done here is implement a basic router for your Rack application. If the route for a request contains "200", then you give back a successful response. Otherwise, you give back a 500 status, indicating an error. Rails implements a much more complex routing system than this, extracting the complexity away and leaving us with methods such as `root` and `resources` that you use in config/routes.rb. The underlying theory is the same though.

You've learned the basics of how a Rack application works and gained an understanding that your Rails application is a bigger version of this little application you've written. There's much more to Rack than providing this abstraction for the underlying request/response cycle. For example, you can build more complex apps with logic for one part of the application in one class and additional logic in another.

One other feature of Rack is the ability to build applications by combining smaller applications into a larger one. You saw this with Rails when you used the `mount` method in your application's config/routes.rb to mount the engine you developed in the last chapter (chapter 17). Let's see how you can do this with Rack.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

17.2 Building bigger Rack applications

Your basic Rack application quickly outgrew the `lambda` shell you placed it in, and so you moved the logic in it into a class and added some more. With the class, you're able to define a `call` method on it, which then returns the response that Rack needs. The class allows you to cleanly write a more complex Rack application than a `lambda` would.

So what happens now if you outgrow a class? Well, you can abstract the function of your application into multiple classes and build a Rack application using those classes. The structure is not unlike the controller structure you have in a Rails application, because it will have separate classes that are responsible for different things.

In your new Rack application, you'll have two classes that perform separate tasks, but are still running on the same instance of the server. The first class is going to be your `Heartbeat::Application` class, and the second one will provide two forms, each with one button: one for success and one for failure. These forms will then submit to the actions provided within the `Heartbeat::Application` class, which will demonstrate how you can get your classes to talk to each other.

17.2.1 You're breaking up

Now that your Rack application is getting more complex, you're going to break it out into three files. The first file will be the `Heartbeat::Application` class, the second will be a new class called `Heartbeat::TestApplication`, and the third will be the Rackup file that will be responsible for combining these two classes into one glorious application.

Let's begin by separating out your application and the Rackup file into two separate files. In a new directory at lib/heartbeat.rb, add the code shown in Listing 18.2 to lib/heartbeat/application.rb.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 17.2 lib/heartbeat/application.rb

```
module Heartbeat
  class Application
    def self.call(env)
      default_headers = { "Content-Type": "text/plain"}

      if env["PATH_INFO"] =~ /200/
        body = "Success!"
        status = 200
      else
        body = "Failure!"
        status = 500
      end

      [status, default_headers, ["#{env["PATH_INFO"]} == #{body}"]]
    end
  end
end
```

Next, in lib/heartbeat/config.ru, add the code shown in Listing 18.3.

Listing 17.3 lib/heartbeat/config.ru

```
heartbeat_root = File.expand_path(File.dirname(__FILE__))
require heartbeat_root + '/application'

run Heartbeat::Application
```

This new lib/heartbeat/config.ru sets up a `heartbeat_root` variable so that you can `require` files relative to the root of the heartbeat directory without having to specify direct paths to them[2]. At the moment, this file still contains the `run` line from the old heartbeat.ru, but you'll be changing this shortly.

Footnote 2 You could also use Ruby 1.9's `require_relative`

Before that change though, you're going to add your second application class, `Heartbeat::TestApplication` to a new file at lib/heartbeat/test_application.rb by using the content shown in Listing 18.4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Listing 17.4 lib/heartbeat/test_application.rb

```
module Heartbeat
  class TestApplication
    def self.call(env)
      default_headers = { "Content-Type": "text/html"}
      body = %Q{
        <h1>Success or FAILURE?!</h1>
        <form action='/test/200'>
          <input type='submit' value='Success!'>
        </form>

        <form action='/test/500'>
          <input type='submit' value='Failure!'>
        </form>
      }

      [200, default_headers, [body]]
    end
  end
end
```

This file follows the same style as the file that defines `Heartbeat::Application`, however in this class the body returned as part of the Rack response consists of two form tags each with their own submit button. The first form goes to /test/200 which should give you the response of "Success!" and /test/500 which should give you a "Failure!" response because the path doesn't include the number `200`.

A keen eye may have noticed that you've nested the paths to the heartbeat responses underneath a path called `test`. This is because when you build your combined class application, you'll make your `Heartbeat::Application` sit under the `/test` route. This is so that when you click the submit button on those two forms from `Heartbeat::TestApplication`, the request will be sent to `Heartbeat::Application`. When do you do this? Right now!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

17.2.2 Running a combined Rack application

You're now going to change the lib/heartbeat/config.ru file to now create a Rack application that combines both of your Rack application classes. For this, you're going to use the `Rack::Builder` class's `app` method, which lets you build Rack applications from different parts. Effectively providing something that's very similar to how the routing and controllers work within Rails. Let's fill lib/heartbeat/config.ru with the content shown in Listing 18.5

Listing 17.5 Combining two Rack applications

```
heartbeat_root = File.expand_path(File.dirname(__FILE__))
require heartbeat_root + '/application'
require heartbeat_root + '/test_application'

app = Rack::Builder.app do
  map '/test' do
    run Heartbeat::Application
  end

  map '/' do
    run Heartbeat::TestApplication
  end
end

run app
```

Rather than calling `run Heartbeat::Application` here, you're compiling a multi-faceted Rack application using `Rack::Builder.app`. The `run` method you've been using all this time is defined inside the `Rack::Builder` class, actually. A *.ru file is usually evaluated within the instance of a `Rack::Builder` object by the code the `rackup` command uses, and so you are able to use the `run` method without having to call `Rack::Builder.new` before it or wrapping or .ru code in a `Rack::Builder.app` block.

This time, you're being implicit and building a new `Rack::Builder` instance using `Rack::Builder.app`. Inside this instance, you'll declare two routes using the `map` method. Within a block given to each of your `map` calls you're calling the `run` method again, passing it one of your two "application" classes.

When a request comes into this application beginning with the path `/test` it

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

will be served by the `Heartbeat::Application` class. All other requests will be served by the `Heartbeat::TestApplication` class. This is not unlike certain requests in your Rails application beginning with `/tickets` are routed to the `TicketsController` and others beginning with `/projects` go to `ProjectsController`[3].

Footnote 3 In fact, the similarities are astounding.

Let's start this application and see what it can do by running this command:

```
rackup lib/heartbeat/config.ru
```

Now remember, to make requests to the `Heartbeat::Application` class you must prefix them with /test, otherwise they'll be served by `Heartbeat::TestApplication`. Keeping that in mind, let's make a request to http://localhost:9292/test/200. You'll see something unusual: the path displayed on the page isn't `/test/200` as you may expect, but rather it's `/200`. The `env["PATH_INFO"]` key doesn't need to contain the path where your application is mounted, as that's not important for routing requests within the application itself.

If you make a request to another path not beginning with the `/test` (such as http://localhost:9292/foo/bar) prefix, you'll see the two buttons in forms provided by the `Heartbeat::TestApplication`, as shown in Figure 18.2.

Figure 17.2 Success or FAILURE?!

When you click on the "Success!" button, you'll send a request to the /test/200 path, which will be served by the `Heartbeat::Application` class and will respond with a body that says `/200 == Success!`. When you press the back button in your browser and press the "Failure!" button, you'll see the `/500 == Failure!`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

This is the basic foundation for Rack applications and a lightweight demonstration of how routing in very basic Rack applications works. When you began, you were able to write `run Heartbeat::Application` to run a single class as your Rack application, but as it's grown more complex you've split different pieces of the functionality out into different classes. To combine these classes into one super-application you used the `Rack::Builder.app` method.

Now you should have a basic understanding of how you can build Rack applications to have a lightweight way of creating dynamic responses. So how does all of this apply to Rails? Well, in Rails you're able to mount a Rack application so that it can serve requests on a path (like you did with `Rack::Builder`), rather than having the request go through the entire Rails stack.

17.3 Mounting a Rack application with Rails

Sometimes, you'll want to serve requests in a lightning-fast fashion. Rails is great for serving super-dynamic requests quickly, but occasionally you'll want to forego the heaviness of the Rails controller stack and have a piece of code that receives a request and quickly responds.

Previously, your Rack application had done just that. However, when you mount your Rack application inside of a Rails application, you're able to use the classes (i.e. models) from within the Rails application. With these models, you can do any number of things. For example, you can build a re-implementation of your tickets API, which will allow you to see an alternate way to craft the API you created in chapter 13. So let's do this.

This new API will be version 3 of your API (things move fast in this app!). It will be accessible at /api/v3/json/projects/:project_id/tickets[4] and—as with your original API—will require a token parameter to be passed through to your application. If the token matches to a user and that user has access to the requested project, you can send back a list of tickets in a JSON format. If the token sent through doesn't match to a user then you'll send back a helpful error message explaining that; if the project requested isn't accessible by the authenticated user you'll deny all knowledge of its existence by sending back a 404 response.

Footnote 4 This URL closely resembles the URL that GitHub uses for v2 of its API, but the similarities are purely coincidental.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

17.3.1 Mounting Heartbeat

Before you get into any of that though, you should probably look at how mounting works within Rails by using one of your basic applications first! Mounting a Rack application involves defining a route in your Rails application that basically says "I want to put this application at this path." Back when you were doing a pure Rack application, you did this in the lib/heartbeat/config.ru file like this:

```
map '/heartbeat' do
  run Heartbeat::Application
end
```

Rails has a better place than that for routes: config/routes.rb. This location provides you with some lovely helpers for mounting your Rack applications. In your Rails application, to do the same thing as you did in your Rack application, you'd need to first require the application by placing this line at the top of config/routes.rb:

```
require 'heartbeat/application'
```

Then inside the routes block of config/routes.rb put this line:

```
mount Heartbeat::Application, at: "/heartbeat"
```

The `mount` method accepts an object to mount and an options hash containing an `at` option to declare where this should be mounted. Alternatively, you could use the `match` method in routes:

```
match '/heartbeat' => Heartbeat::Application
```

Both lines are identical in function. So let's make these changes to your config/routes.rb file now and boot up your Rails server with this command:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
bin/rails s
```

You should now be able to go to http://localhost:3000/heartbeat/200 and see the friendly `/200 == Success!` message. This means that your `Heartbeat::Application` is responding as you'd like it to, nestled within the confines of your Rails application.

Rails has been told to forward requests that go to /heartbeat to this Rack application and it has done so diligently. Rather than initializing a new instance of a controller (which is what normally happens in a standard Rails request), a Rack class is much lighter and is perfect for serving high-intensity requests that don't require views, like the response from your `Heartbeat::Application` and the responses from your API.

So now that you've learned how you can mount your `Heartbeat::Application`, let's build this slightly more complex Rack application that will serve JSON API requests for tickets. To make sure everything works, you'll be writing tests using the same `Rack::Test::Methods` helpers that you used back in chapter 13. These helpers are designed for Rack applications, but they worked with your Rails application because... well, it's a Rack app too.

Rather than writing this application as a standard Rack app, let's mix things up a bit and use another Ruby web framework called Sinatra, which uses the Rack architecture underneath, just like Rails.

17.3.2 Introducing Sinatra

Sinatra is an exceptionally light-weight Ruby web framework that's perfect for building small applications, such as those that serve an API. Like Rails, it's built on top of Rack and so you'll have no worries about using them together. You'll use it here to create version 3 of your API. Building you app this way not only demonstrates the power of Sinatra, but also shows that there's more than one way to skin this particular cat[5].

Footnote 5 Although why anybody would skin a cat these days is unknown to the authors.

To install the `sinatra` gem, run this command:

```
gem install sinatra
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

You can make a small Sinatra script now by creating a file called sin.rb, as shown in Listing 18.6.

Listing 17.6 sin.rb

```
require 'sinatra'

get '/' do
  "Hello World"
end
```

This is the most basic Sinatra application that you can write. On the first line you require the sinatra file, which gives you some methods you can use to define your application, such as the `get` method you use on the next line. This `get` method is used to define a root route for your application, which returns the string "Hello World" for `GET` requests to `/`. You could also make it into a class, which is what you'll need to do for it to be mountable in your application:

```
require 'sinatra'

class Tickets < Sinatra::Base
  get '/' do
    "Hello World"
  end
end
```

By making it a class, you'll be able to mount it in your application using the `mount` method in config/routes.rb and specifying the class name. By mounting this Sinatra application inside your Rails application, it will have access to all the classes from your Rails application, such as your models, which is precisely what you're going to need for this new version of your API. You won't use this code example right now; it's handy to know that you can do this.

To use Sinatra with your application, you'll need to add it to the Gemfile with this line:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
gem 'sinatra'
```

Then you'll need to run `bundle install` to install it. So let's go ahead now and start building this API using Sinatra.[6]

Footnote 6 You can learn more about Sinatra at https://github.com/sinatra/sinatra/, and at http://sinatrarb.com/intro.

17.3.3 The API, by Sinatra

Let's create a new file to test your experimental new API at spec/api/v3/json/tickets_spec.rb. In this file you want to set up a project that has at least one ticket, as well as a user that you can use to make requests to your API. After that, you want to make a request to /api/v3/json/tickets and check that you get back a proper response of tickets. With this in mind, let's write a spec that looks like the code shown in Listing 18.7

Listing 17.7 spec/api/v3/json/tickets_spec.rb

```
require 'spec_helper'

describe Api::V3::JSON::Tickets, type: :api do
  let(:project) { FactoryGirl.create(:project) }
  let(:user) { FactoryGirl.create(:user) }
  let(:token) { user.authentication_token }

  before do
    FactoryGirl.create(:ticket, project: project)
    user.permissions.create!(thing: project, action: "view")
  end

  let(:url) { "/api/v3/json/projects/#{project.id}/tickets" }

  context "successful requests" do

    it "can get a list of tickets" do
      get url, token: token
      expect(last_response.body).to eql(project.tickets.to_json)
    end
  end
end
```

This test looks remarkably like the one in spec/api/v2/tickets_spec.rb, except this time you're only testing for JSON responses and you've changed the URL that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

you're requesting to `api/:version/:format/:path`. When you run this spec with `bin/rspec spec/api/v3/json/tickets_spec.rb` you'll see that it's giving you this error:

```
... uninitialized constant Api::V3
```

This is because you haven't yet defined the module for the `Api::V3` namespace yet. Let's create a new file at app/controllers/api/v3/json/tickets.rb that defines this module, as shown in Listing 18.8.

Listing 17.8 app/controllers/api/v3/json/tickets.rb

```
require 'sinatra'

module Api
  module V3
    module JSON
      class Tickets < Sinatra::Base
        before do
          headers "Content-Type": "text/json" <co id="ch18_12601_1"/>
        end
        get '/' do
          []
        end
      end
    end
  end
end
```

Within this file you define the `Api::V3::JSON::Tickets` class that is `describe`'d at the top of your spec, which will now make your spec run. This class inherits from `Sinatra::Base` so that you'll get the helpful methods that Sinatra provides, such as the `before` ❶ and `get` methods that you use here. You've already seen what `get` can do, but `before` is new. This method is similar to a `before_action` in Rails and will execute the block before each request. In this block, you set the headers for the request, using Sinatra's `headers` method, so that your API identifies as sending back a `text/json` response.

Why put this code inside app/controllers? Well, even though this "controller" is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

most definitely not a controller -- in the common Rails sense -- it's stilla class that's going to be handling requests and acting *like* a controller, and therefore app/controllers is still a perfectly good place for it.

Let's rerun the spec again using `bin/rspec spec/api/v3/json/tickets_spec.rb`:

```
Failure/Error: get url, token: token
ActionController::RoutingError:
  No route matches [GET] "/api/v3/json/projects/1/tickets"
```

This is a better start, now your test is running and failing as it should because you haven't defined the route for it yet. Your test is expecting to be able to do a `GET` request to `/api/v3/json/projects/1/tickets` but cannot.

This route can be interpreted as /api/v3/json/projects/:project_id/tickets and you can use the `api` namespace already in config/routes.rb to act as a 'home' for this route. Let's put some code for `v3` of your API inside this `namespace` now:

```
namespace :v3 do
  namespace :json do
    mount Api::V3::JSON::Tickets,
      at: "/projects/:project_id/tickets"
  end
end
```

By placing this `mount` call inside the namespaces, the Rack application will be mounted at /api/v3/json/projects/:project_id/tickets rather than the /tickets URI if you didn't have it nested. Additionally, you've specified a dynamic parameter in the form of `:project_id` inside the `at` option for the `mount` call, which means you'll be able to access the requested project id from inside your Rack application using a method very similar to how you'd usually access parameters in a controller.

If you attempted to run your spec again with `bin/rspec spec/api/v3/json/tickets_spec.rb` it would bomb out with another new error:

```
expected "[tickets array]"
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
got ""
```

This means that requests are able to get to your Rack app and that the response you've declared is being served successfully. Now you need to fill this response with meaningful data. To do this, find the project that's being referenced in the URL by using the parameters passed through found with the `params` method. Unfortunately, Sinatra doesn't load the parameters from your Rails application and so `params[:project_id]` is not going to be set. You can see this if you change your root route in your Sinatra application to this:

```
get '/' do
  p params
end
```

Then if you run your test, you'll see only the `token` parameter is available:

```
{"token"=>"6E06zoj01Pf5texLXVNb"}
```

Luckily, you can still get to this through one of the keys in the environment hash, which is accessible through the `env` method in your Sinatra actions, like it was available when you built your Rack applications. You saw this environment hash earlier when you were developing your first Rack application, but this time it's going to have a little more to it because it's gone through the Rails request stack. Let's change your root route to this:

```
get '/' do
  p env.keys
end
```

When you rerun your test, you'll see all the available keys output at the top, with one of the keys being `action_dispatch.request.path_parameters`. This key stores the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

parameters discovered by Rails routing, and your `project_id` parameter should fall neatly into this category. Let's find out by changing the `p env.keys` line in your root route to `p env["action_dispatch.request.path_parameters"]` and then re-running your test. You should see this:

```
{:project_id=>"3"}
```

Okay, so you can access two parameter hashes, but you'll need to merge them together if you are to do anything useful with them. You can merge them into a super `params` method by re-defining the `params` method as a private method in your app. Underneath the `get` you'll put this:

```
def params
  hash = env["action_dispatch.request.path_parameters"].merge!(super)
  HashWithIndifferentAccess.new(hash)
end
```

By calling the `super` method here, you'll reference the `params` method in the superclass, `Sinatra::Base`. You want to access the keys in this hash using either symbols or strings like you can do in your Rails application, so you create a new `HashWithIndifferentAccess` object, which is returned by this method. This lets you access your token with either `params[:token]` or `params["token"]`. This hash is quite indifferent to its access methods.

Let's switch your root route back to calling `p params`. When you run your test again, you should see that you finally have both parameters inside the one hash:

```
{:project_id=>"3", "token"=>"ZVSREe1aQjNZ2SrB9e8I"}
```

With these parameters you'll now be able to find the user based on their token, get a list of projects they have access to, and then attempt to find the project with

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the id specified. You can do this by putting two calls, a `find_user` and `find_project` method, in the `before` block you already have, using this code:

```
before do
  headers "Content-Type": "text/json"
  find_user
  find_project
end
```

The `find_user` and `find_project` methods can be defined underneath the `private` keyword using this code:

```
private

  def find_user
    @user = User.find_by_authentication_token(params[:token])
  end

  def find_project
    @project = Project.for(@user).find(params[:project_id])
  end
```

This code should look fairly familiar, it's basically identical to the code found in the `Api::V1::TicketsController` and `Api::V1::BaseController` classes inside your Rack application. First you find the user based on their token and then generate a scope for all projects that the user is able to view with the `Project.for` method. With this scope, you can then find the project matching the id passed in through `params[:project_id]`. You are referencing the models from your Rails application inside your Sinatra application and there's nothing special you have to configure to allow this.

Because you're not too concerned with what happens if an invalid `params[:project_id]` or user token is passed through at the moment, you'll fix those up after you've got this first test passing. With the project now found, you should be able to display a list of tickets in JSON form in your `call` method. Let's change your root route to return a list of JSON-ified tickets for this project:

```
get '/' do
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
  @project.tickets.to_json
end
```

Now your root route should respond with the list of tickets required to have your test pass. Let's see if this is the case by running `bin/rspec spec/api/v3/json/tickets_spec.rb`:

```
1 example, 0 failures
```

Great, this spec is now passing, which means that your Rack application is now serving a base for version 3 of your API. By making this a Rack application you can serve requests in a more lightweight fashion than you could within Rails.

But, you don't have basic error checking in place yet if a user isn't found matching a token or if a person can't find a project. So before you move on, let's quickly add tests for these two issues.

17.3.4 Basic error checking

You'll open spec/api/v3/json/tickets_spec.rb and add two tests inside the `describe` block in a new `context` block, as shown in Listing 18.9.

Listing 17.9 spec/api/v3/json/tickets_spec.rb

```
context "unsuccessful requests" do
  it "doesn't pass through a token" do
    get url
    expect(last_response.status).to eql(401)
    expect(last_response.body).to eql("Token is invalid.")
  end

  it "cannot access a project that they don't have permission to" do
    user.permissions.delete_all
    get url, token: token
    expect(last_response.status).to eql(404)
  end
end
```

In the first test you make a request without passing through a token, which should result in a `401` (unauthorized) status and a message telling you the "Token

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

is invalid." In the second test, you use the `delete_all` association method to remove all permissions for the user and then attempt to request tickets in a project that the user no longer has access to. This should result in the response being a 404 response, which means your API will deny all knowledge of that project and its tickets.

To make your first test pass you'll need to check that your `find_user` method actually returns a valid user, otherwise you'll return this 401 (Unauthorized) response. The best place to do this would be inside the `find_user` method itself, turning it into this:

```
def find_user
  @user = User.find_by_authentication_token(params[:token])
  halt 401, "Token is invalid." unless @user
end
```

The `halt` method here will stop a request dead in its tracks. In this case, it will return a 401 status code with the body being the string specified. When you run your tests again with `bin/rspec spec/api/v3/json/tickets_spec.rb` the first two should be passing, with the third one still failing:

```
3 examples, 1 failure
```

Alright, so now if an invalid token is passed, you're throwing exactly the same error as the last two iterations of your API did, good progress! This error tells the API client that the token used is invalid and returns a 401 (Unauthorized) status.

Finally, you'll need to send a 404 response when a project cannot be found within the scope for the current user. To do this, change the `find_project` method in your app to this:

```
def find_project
  @project = Project.for(@user).find(params[:project_id])
  rescue ActiveRecord::RecordNotFound
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

```
    halt 404, "The project you were looking for could not be found."
end
```

When you run your tests for a final time with `bundle exec rspec spec/api/v3/tickets_spec.rb`, they should all pass:

```
3 examples, 0 failures
```

Awesome! This should give you a clear idea of how you could implement an API similar to the one you created back in chapter 13 by using the lightweight framework of Sinatra. All of this is possible because Rails provides an easy way to mount Rack-based applications inside your Rails applications. You could go further with this API, but this is probably another exercise for you later on if you wish to undertake it.

You've learned how you can use Rack applications to serve as endpoints of requests, but you can also create pieces that hook into the middle of the request cycle called *middleware*. Rails has a few of these already, and you saw the effects of one of them when you were able to access the `env["action_dispatch.request.path_parameters"]` key inside your Sinatra application. Without the middleware of the Rails stack, this parameter would be unavailable. In the next section, we look at the middleware examples in the real world, including some found in the Rails stack, as well as how you can build and use your own.

17.4 Middleware

When a request comes into a Rack application, it doesn't go straight to a single place that serves the request. Instead, it goes through a series of pieces known as *middleware*, which may process the request before it gets to the end of the stack (your application) or modify it and pass it onward, as shown in Figure 18.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

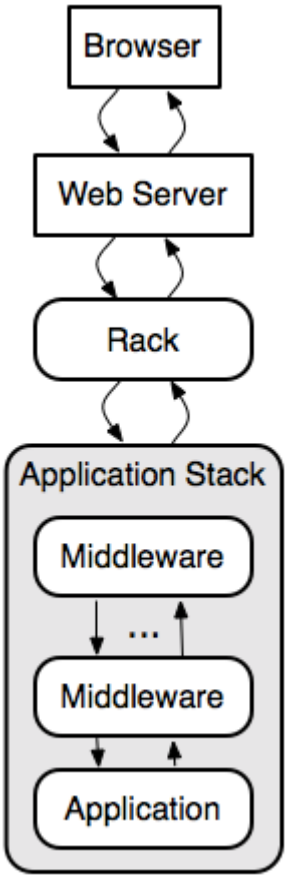

Figure 17.3 Full request stack, redux

You can run the `bin/rake middleware` within your Rails application's directory to see the list of middleware currently in use by your Rails application:

```
use ActionDispatch::Static
use Rack::Lock
use ActiveSupport::Cache::Strategy::LocalCache
use Rack::Runtime
...
use ActionDispatch::BestStandardsSupport
use Warden::Manager
run Ticketee::Application.routes
```

Each of these middleware pieces perform their own individual function. For instance, the first middleware `ActionDispatch::Static` intercepts requests for static files such as images, javascript files, or stylesheets found in public and serves them immediately, without the request to them falling through to the rest of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

the stack. It's important to note that this middleware is only active in the development environment, as in production your web server (such as nginx) is better suited for serving static assets.

Other middleware, such as `ActionDispatch::BestStandardsSupport`, set additional headers on your request. This particular piece of middleware sets the `X-UA-Compatible` header to `IE=Edge,chrome=1`, which tells Microsoft Internet Explorer to "display content in the highest mode available" that is "equivalent to IE9 mode", meaning your pages should render in a "best standards" fashion[7]. The `chrome=1` part of this header is for the Google Chrome Frame which again, will support "best standards" rendering on a page.

Footnote 7 For more information about the `IE=Edge` and `X-UA-Compatible` header: http://msdn.microsoft.com/en-us/library/cc288325(v=vs.85).aspx

Let's look at how `ActionDispatch::BestStandardsSupport` works.

17.4.1 Middleware in Rails

In the case of the `ActionDispatch::Static` middleware, a response is returned when it finds a file to serve and the request stops there. In the case of `ActionDispatch::BestStandardsSupport`, the request is modified and allowed to continued down the chain of middleware until it hits `Ticketee::Application.routes`, which will serve the request using the routes and code in your application. The process of `ActionDispatch::Static` can be seen in Figure 18.4

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders. http://www.manning-sandbox.com/forum.jspa?forumID=818

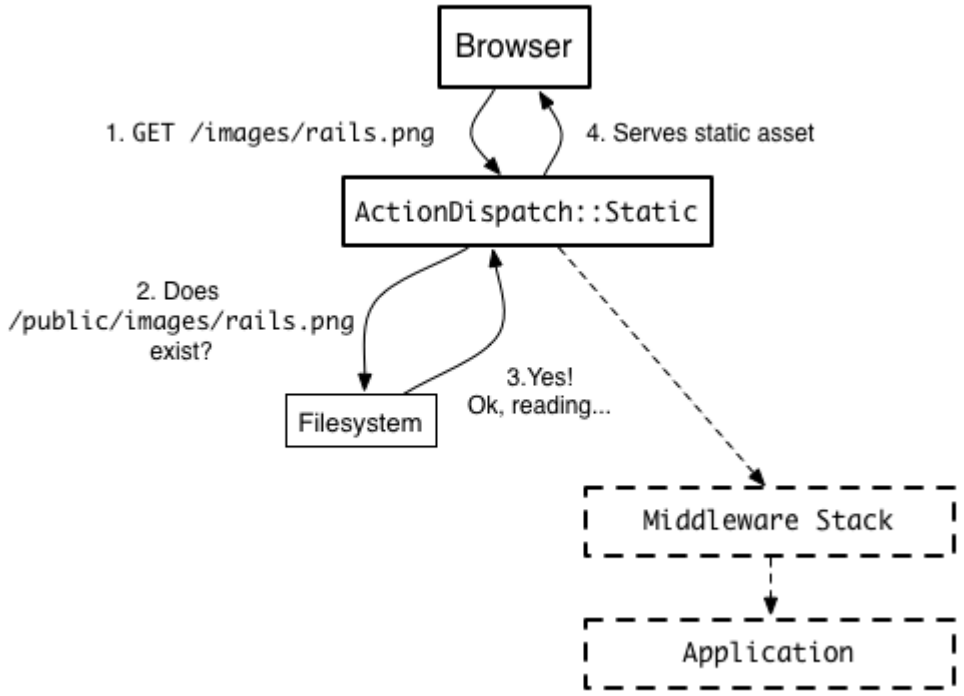

Figure 17.4 ActionDispatch::Static Request

When a request is made to /images/bin/rails.png, the middleware checks to see if the public/images/bin/rails.png file exists. If it does, then it is returned as the response of this request. This middleware will also check for cached pages. If you make a request to /projects, Rails (by default) will first check to see if a public/projects.html file exists before sending the request to the rest of the stack. This type of request is shown in Figure 18.5.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

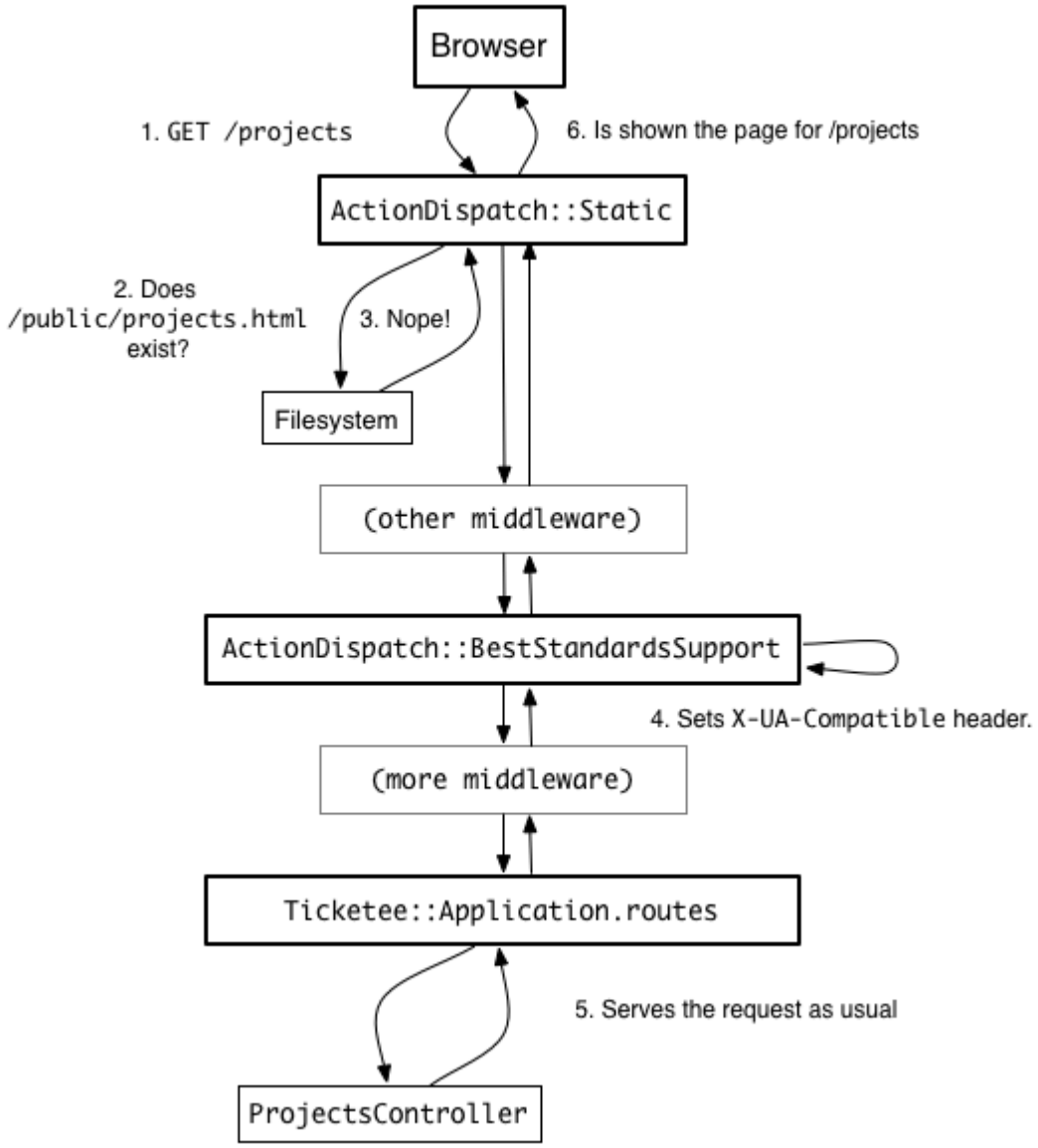

Figure 17.5 ActionDispatch::BestStandardsSupport

In this request, the `ActionDispatch::Static` middleware first checks for the presence of public/projects.html, which would be there if you had cached the page. Because it's not there, the request goes through the rest of the middleware stack being passed along. When it gets to `ActionDispatch::Best::StandardsSupport`, this middleware sets the `X-UA-Compatbile` header and passes along the request to the application, which then serves the request like normal.

Let's dive into exactly how `ActionDispatch::BestStandardsSupport` works.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

17.4.2 Investigating ActionDispatch::BestStandardsSupport

The `ActionDispatch::BestStandardsSupport` is the simplest piece of middleware within the Rails stack. Here's the entirety of this class's code, from actionpack/lib/action_dispatch/best_standards_support.rb within Rails itself:

Listing 17.10 ActionDispatch::BestStandardsSupport class definition

```
module ActionDispatch
  class BestStandardsSupport
    def initialize(app, type = true)
      @app = app

      @header = case type
      when true
        "IE=Edge,chrome=1"
      when :builtin
        "IE=Edge"
      when false
        nil
      end
    end

    def call(env)
      status, headers, body = @app.call(env)
      headers["X-UA-Compatible"] = @header
      [status, headers, body]
    end
  end
end
```

The first method defined in this class is the initialize method, which takes two arguments: an `app` object and a `type` object, which defaults to `true`. The `app` object is the next piece of middleware in the stack. This is made available so that you can choose to call it and delegate the job of serving the request to that piece of middleware instead. That piece may then choose to serve an actual three-part Rack response, or pass it on to another piece of middleware. At any time during the stack, a piece of middleware can choose to send back a response and then all future middleware objects will not be parsed. We'll get to what this piece of middleware is doing in just a moment.

The other code inside this method will define a `@header` variable, checking the value of `type`. If that value is true, then it will set `@header` to `IE=Edge,chrome=1` which will tell Internet Explorer to use the edge mode, and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

will enable Google Chrome Frame on a user's browser, if they have it. If it's set to `:builtin`, it will just use the `IE=Edge` header. If it's set to `false`, then there will be no header. These two parts of the header are explained in much greater detail here: http://stackoverflow.com/a/6771584/15245.

The other method inside this class, the `call` method, takes one argument called `env` which is the Rack environment hash. This will pass the request down the chain of middleware, returning that three-part Rack response you've come to know and love. Once it's got that, it adds the `X-UA-Compatible` header to the headers Hash and then returns all three parts.

Now that you've got a nice grasp of how one piece of middleware works, let's build your own!

17.4.3 Crafting middleware

Soon you'll have your own piece of middleware that you can put into the middleware stack of a Rails or Rack application. This middleware will allow the request to run all the way down the chain to the application and then will modify the body, replacing specific letters in the text for links with other, equally specific letters. Create a new file for your middleware at lib/link_jumbler.rb and fill it with the content shown in Listing 18.11.

Listing 17.11 lib/link_jumbler.rb

```
require 'nokogiri'
class LinkJumbler

  def initialize(app, letters)
    @app = app
    @letters = letters
  end

  def call(env)
    status, headers, response = @app.call(env)
    body = Nokogiri::HTML(response.body)
    body.css("a").each do |a|
      @letters.each do |find, replace|
        a.content = a.content.gsub(find.to_s, replace.to_s)
      end
    end
    [status, headers, body.to_s]
  end
end
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

In this file you've defined the `LinkJumbler` class, which contains an `initialize` and a `call` method. The `initialize` method sets the stage, setting up the `@app` and `@letters` variables you'll use in your `call` method.

In the `call` method, you make a call down the middleware stack in order to setup your `status`, `headers`, and `body` values. You can do this because the `@app.call(env)` call will always return a three-element array. Each element of this array will be assigned to its respective variable. In a Rails application's middleware stack, the third element isn't an array but rather an instance of `ActionDispatch::Response`. To get to the good part of this response you can use the `body` method, like you do on the second line of your `call` method.

With this body you use the `Nokogiri::HTML` method (provided by the `require 'nokogiri'` line at the top of this file) to parse the body returned by the application into a `Nokogiri::HTML::Document` object. This will allow you to parse the page more easily than if you used regular expressions. With this object, you call the `css` method and pass it the `"a"` argument, which finds all `a` tags in the response body. You then iterate through each of these tags and go through all of your letters from `@letters`, using the keys of the hash as the `find` argument and the values as the `replace` argument. You then set the content of each of the `a` tags to be the substituted result.

Finally, you return a three-element array using your new body, resulting in links being jumbled. To see this middleware in action, you'll need to add it to the middleware stack in your application. To do that, put these two lines inside the `Ticketee::Application` class definition in config/application.rb:

```
require 'link_jumbler'
config.middleware.use LinkJumbler, { "e": "a" }
```

The `config.middleware.use` method will add your Middleware to the end of the middleware stack, making it the last piece of middleware to be processed before a request hits your application[8]. Any additional arguments passed to the `use` method will be passed as arguments to the `initialize` method for this middleware, and so this hash you've passed here will be the `letters` argument in your middleware. This means your `LinkJumbler` middleware will replace the letter "e" with "a" anytime it finds it in an `a` tag.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

Footnote 8 For more methods for `config.middleware` look at the "Configuring Middleware" section of the Configuring official guide: http://guides.rubyonbin/rails.org/configuring.html#configuring-middleware

To see this middleware in action, let's fire up a server by running `bin/rails s` in a terminal. When you go to http://localhost:3000 you should notice something's changed, as shown in Figure 18.6

Figure 17.6 What's a Tickataa?!

As you can see in this figure your links have had their "e's" replaced with "a's" and any other occurrence, such as the user's email address, has been left untouched.

This is one example of how you can use middleware to affect the outcome of a request within Rails; you could have modified anything or even sent a response back from the middleware itself. The opportunities are endless. This time though, you've made a piece of middleware which finds all the `a` tags and jumbles up the letters based on what you tell it to.

17.5 Summary

You've now seen a lot of what Rack, one of the core components of the Rails stack can offer us. In the beginning of this chapter you built a small Rack application that responded with "OK". You then fleshed this application out to respond differently based on the provided request. Then you built another Rack application that called this first Rack application, running both of these within the same instance by using the `Rack::Builder` class.

Then you saw how you could use these applications within the Rails stack by first mounting your initial Rack application and then branching out into something a little more complex, with a Sinatra-based application serving what could possibly be the beginnings of version 3 of Ticketee's API. Sinatra is a lightweight framework offering the same basic features as Rails.

Finally, you saw two pieces of middleware, the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818

`ActionDispatch::Static` piece and the `ActionDispatch::BestStandardsSupport`. You dissected the first of these, figuring out how it worked so that you could use that knowledge to build your own middleware, a neat little piece that jumbles up the text on the link based on the options passed to it.

Index Terms

ActionDispatch::BestStandardsSupport
ActionDispatch::Static
config.middleware
Middleware
PATH_INFO
Rack environment object
rake middleware
Sinatra, halt method
Using middleware

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
http://www.manning-sandbox.com/forum.jspa?forumID=818