Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm that GitHub renders the Markdown for the README in Listing 3.3 as HTML (Figure 3.2).

2. By visiting the root route on the production server, verify that the deployment to Heroku succeeded.

3.2 Static pages

With all the preparation from Section 3.1 finished, we're ready to get started developing the sample application. In this section, we'll take a first step toward making dynamic pages by creating a set of Rails *actions* and *views* containing only static HTML.[6] Rails actions come bundled together inside *controllers* (the C in MVC from Section 1.2.3), which contain sets of actions related by a common purpose. We got a glimpse of controllers in Chapter 2, and will come to a deeper understanding once we explore the REST architecture more fully (starting in Chapter 6). In order to get our bearings, it's helpful to recall the Rails directory structure from Section 1.2 (Figure 1.11). In this section, we'll be working mainly in the `app/controllers` and `app/views` directories.

Recall from Section 1.3.4 that, when using Git, it's a good practice to do our work on a separate topic branch rather than the master branch. If you're using Git for version control, you should run the following command to checkout a topic branch for static pages:

[6] Our method for making static pages is probably the simplest, but it's not the only way. The optimal method really depends on your needs; if you expect a *large* number of static pages, using a Static Pages controller can get quite cumbersome, but in our sample app we'll only need a few. If you do need a lot of static pages, take a look at the `high_voltage` gem from thoughtbot.

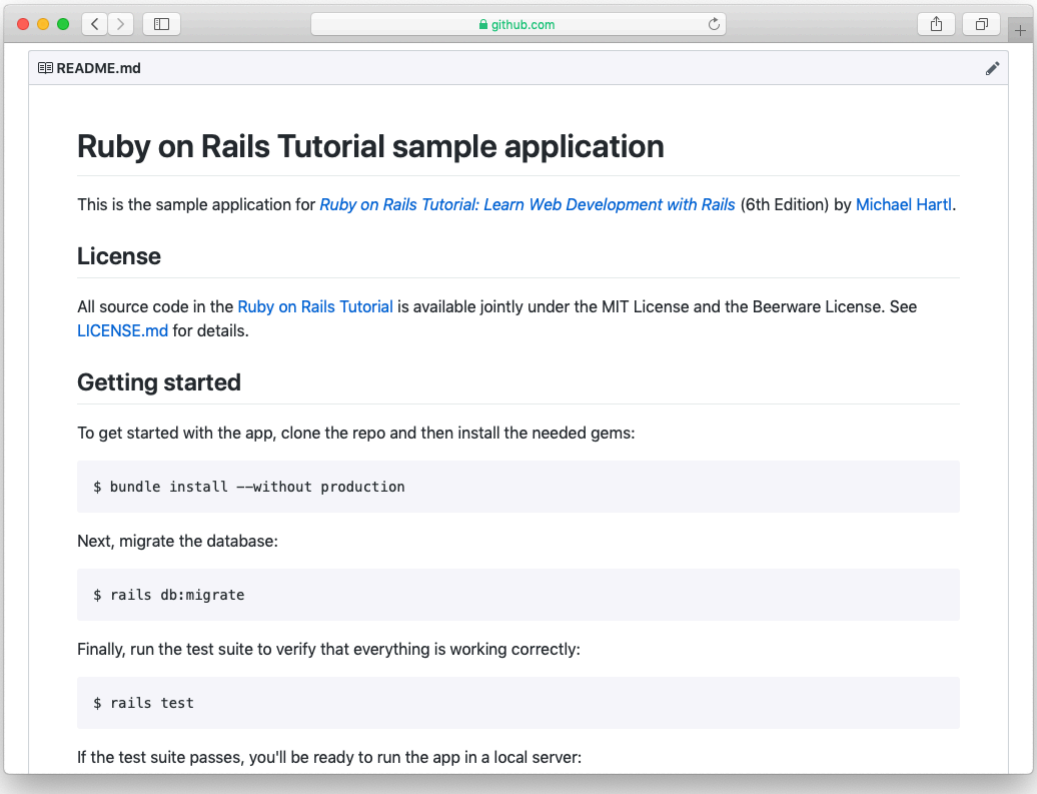

Figure 3.2: The sample app README at GitHub.

```
$ git checkout -b static-pages
```

3.2.1 Generated static pages

To get started with static pages, we'll first generate a controller using the same Rails **generate** script we used in Chapter 2 to generate scaffolding. Since we'll be making a controller to handle static pages, we'll call it the Static Pages controller, designated by the CamelCase name **StaticPages**. We'll also plan to make actions for a Home page, a Help page, and an About page, designated by the lower-case action names **home**, **help**, and **about**. The **generate** script takes an optional list of actions, so we'll include actions for the Home and Help pages directly on the command line, while intentionally leaving off the action for the About page so that we can see how to add it (Section 3.3). The resulting command to generate the Static Pages controller appears in Listing 3.7.

Listing 3.7: Generating a Static Pages controller.

```
$ rails generate controller StaticPages home help
      create  app/controllers/static_pages_controller.rb
       route  get 'static_pages/home'
get 'static_pages/help'
      invoke  erb
      create    app/views/static_pages
      create    app/views/static_pages/home.html.erb
      create    app/views/static_pages/help.html.erb
      invoke  test_unit
      create    test/controllers/static_pages_controller_test.rb
      invoke  helper
      create    app/helpers/static_pages_helper.rb
      invoke    test_unit
      invoke  assets
      invoke    scss
      create      app/assets/stylesheets/static_pages.scss
```

Incidentally, it's worth noting that **rails g** is a shortcut for **rails generate**, which is only one of several shortcuts supported by Rails (Table 3.1). For clarity, this tutorial always uses the full command, but in real life most Rails

Full command	Shortcut
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ rails test`	`$ rails t`
`$ bundle install`	`$ bundle`

Table 3.1: Some Rails shortcuts.

developers use one or more of the shortcuts shown in Table 3.1.[7]

Before moving on, if you're using Git it's a good idea to add the files for the Static Pages controller to the remote repository:

```
$ git add -A
$ git commit -m "Add a Static Pages controller"
$ git push -u origin static-pages
```

The final command here arranges to push the **`static-pages`** topic branch up to GitHub. Subsequent pushes can omit the other arguments and write simply

```
$ git push
```

The commit and push sequence above represents the kind of pattern I would ordinarily follow in real-life development, but for simplicity I'll typically omit such intermediate commits from now on. (When following this tutorial, a good rule of thumb is to make a Git commit at the end of each section.)

In Listing 3.7, note that we have passed the controller name as CamelCase (so called because it resembles the humps of a Bactrian camel), which leads to the creation of a controller file written in snake case, so that a controller called StaticPages yields a file called **`static_pages_controller.rb`**. This is merely a convention, and in fact using snake case at the command line also works: the command

[7] In fact, many Rails developers also add an *alias* (as described in *Learn Enough Text Editor to Be Dangerous*) for the **`rails`** command, typically shortening it to just **`r`**. This allows us to run, e.g., a Rails server using the compact command **`r s`**.

```
$ rails generate controller static_pages ...
```

also generates a controller called `static_pages_controller.rb`. Because Ruby uses CamelCase for class names (Section 4.4), my preference is to refer to controllers using their CamelCase names, but this is a matter of taste. (Since Ruby filenames typically use snake case, the Rails generator converts CamelCase to snake case using the `underscore` method.)

By the way, if you ever make a mistake when generating code, it's useful to know how to reverse the process. See Box 3.1 for some techniques on how to undo things in Rails.

Box 3.1. Undoing things

Even when you're very careful, things can sometimes go wrong when developing Rails applications. Happily, Rails has some facilities to help you recover.

One common scenario is wanting to undo code generation—for example, when you change your mind on the name of a controller and want to eliminate the generated files. Because Rails creates a substantial number of auxiliary files along with the controller (as seen in Listing 3.7), this isn't as easy as removing the controller file itself; undoing the generation means removing not only the principal generated file, but all the ancillary files as well. (In fact, as we saw in Section 2.2 and Section 2.3, `rails generate` can make automatic edits to the `routes.rb` file, which we also want to undo automatically.) In Rails, this can be accomplished with `rails destroy` followed by the name of the generated element. In particular, these two commands cancel each other out:

```
$ rails generate controller StaticPages home help
$ rails destroy  controller StaticPages home help
```

Similarly, in Chapter 6 we'll generate a *model* as follows:

```
$ rails generate model User name:string email:string
```

This can be undone using

```
$ rails destroy model User
```

(In this case, it turns out we can omit the other command-line arguments. When you get to Chapter 6, see if you can figure out why.)

Another technique related to models involves undoing *migrations*, which we saw briefly in Chapter 2 and will see much more of starting in Chapter 6. Migrations change the state of the database using the command

```
$ rails db:migrate
```

We can undo a single migration step using

```
$ rails db:rollback
```

To go all the way back to the beginning, we can use

```
$ rails db:migrate VERSION=0
```

As you might guess, substituting any other number for `0` migrates to that version number, where the version numbers come from listing the migrations sequentially.

With these techniques in hand, we are well-equipped to recover from the inevitable development snafus.

The Static Pages controller generation in Listing 3.7 automatically updates the routes file (**`config/routes.rb`**), which we first saw in Section 1.2.4 when we edited the root route for the hello app (Listing 1.11), and which we most recently saw in Listing 3.6. The routes file is responsible for implementing the router (seen in Figure 2.11) that defines the correspondence between URLs and web pages. The routes file is located in the **`config`** directory, where Rails collects files needed for the application configuration (Figure 3.3).

Since we included the **`home`** and **`help`** actions in Listing 3.7, the routes file

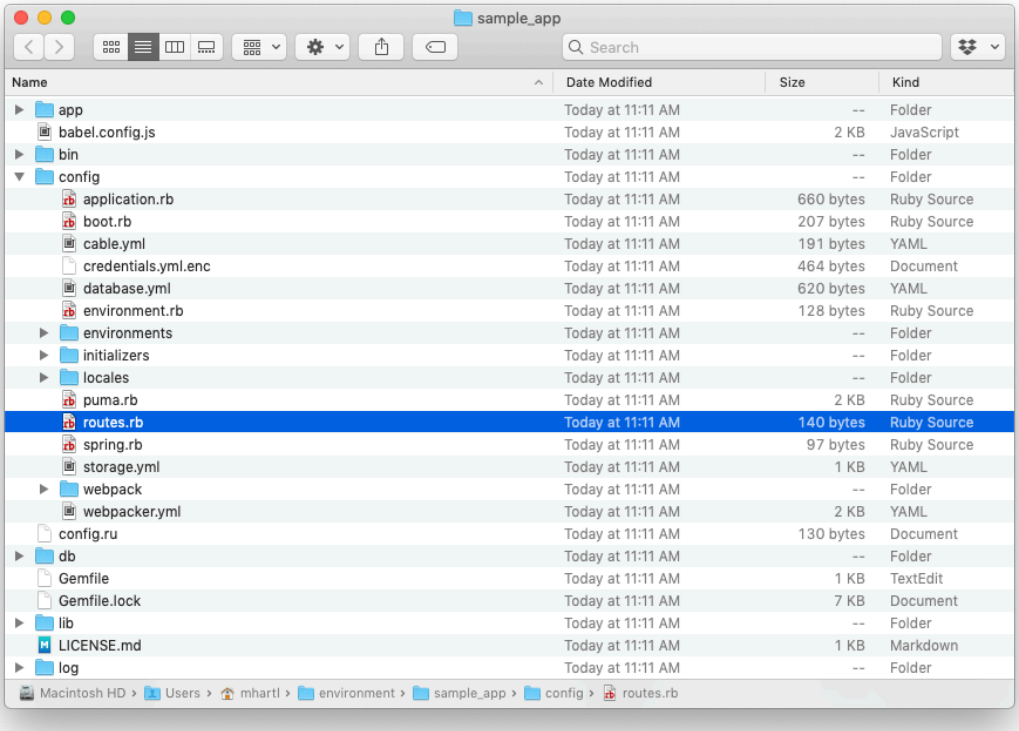

Figure 3.3: Contents of the sample app's **`config`** directory.

already has a rule for each one, as seen in Listing 3.8.

Listing 3.8: The routes for the **home** and **help** actions in the Static Pages controller.
config/routes.rb

```
Rails.application.routes.draw do
  get  'static_pages/home'
  get  'static_pages/help'
  root 'application#hello'
end
```

Here the rule

```
get 'static_pages/home'
```

maps requests for the URL /static_pages/home to the **home** action in the Static Pages controller. Moreover, by using **get** we arrange for the route to respond to a `GET` request, which is one of the fundamental *HTTP verbs* supported by the Hypertext Transfer Protocol (Box 3.2). In our case, this means that when we generate a **home** action inside the Static Pages controller we automatically get a page at the address /static_pages/home. To see the result, start a Rails development server as described in Section 1.2.2:

```
$ rails server
```

Then navigate to /static_pages/home (Figure 3.4).

Box 3.2. `GET`, et cet.

The Hypertext Transfer Protocol (HTTP) defines the basic operations `GET`, `POST`, `PATCH`, and `DELETE`. These refer to operations between a *client* computer (typically running a web browser such as Chrome, Firefox, or Safari) and a

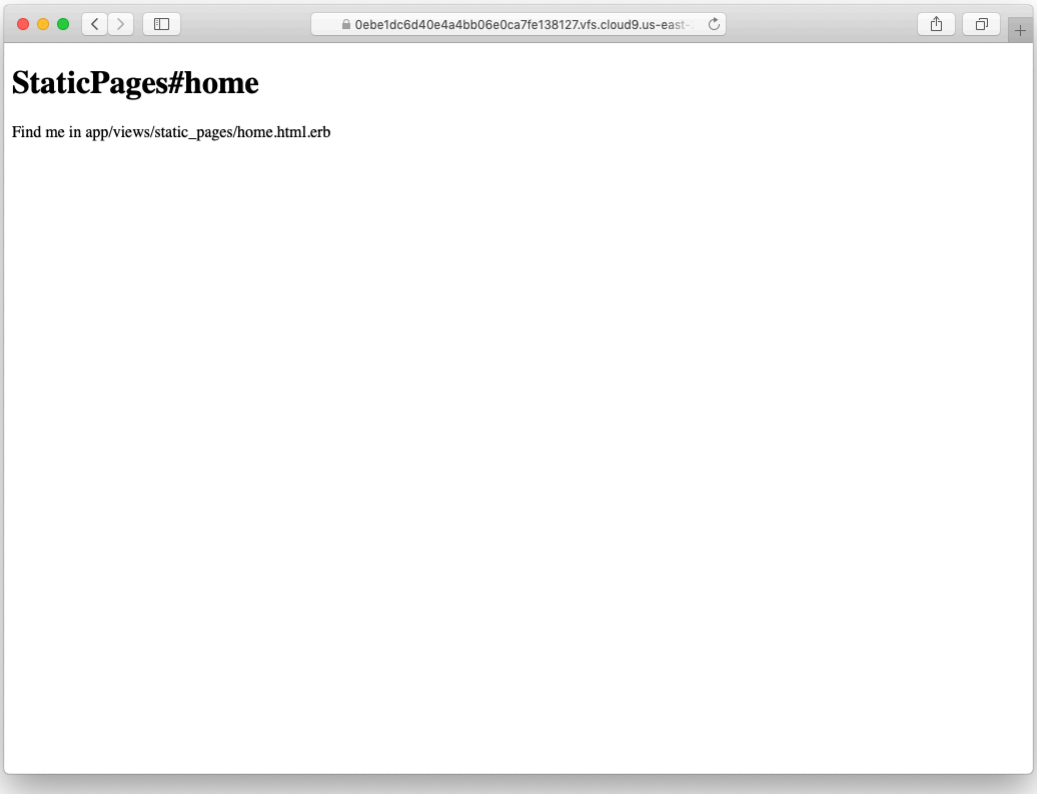

Figure 3.4: The raw home view (/static_pages/home).

server (typically running a webserver such as Apache or Nginx). (It's important to understand that, when developing Rails applications on a local computer, the client and server are the same physical machine, but in general they are different.) An emphasis on HTTP verbs is typical of web frameworks (including Rails) influenced by the *REST architecture*, which we saw briefly in Chapter 2 and will start learning more about in Chapter 7.

`GET` is the most common HTTP operation, used for *reading* data on the web; it just means "get a page", and every time you visit a site like `https://www.google.com/` or `https://www.wikipedia.org/` your browser is submitting a `GET` request. `POST` is the next most common operation; it is the request sent by your browser when you submit a form. In Rails applications, `POST` requests are typically used for *creating* things (although HTTP also allows `POST` to perform updates). For example, the `POST` request sent when you submit a registration form creates a new user on the remote site. The other two verbs, `PATCH` and `DELETE`, are designed for *updating* and *destroying* things on the remote server. These requests are less common than `GET` and `POST` since browsers are incapable of sending them natively, but some web frameworks (including Ruby on Rails) have clever ways of making it *seem* like browsers are issuing such requests. As a result, Rails supports all four of the request types `GET`, `POST`, `PATCH`, and `DELETE`.

To understand where this page comes from, let's start by taking a look at the Static Pages controller in a text editor, which should look something like Listing 3.9. You may note that, unlike the demo Users and Microposts controllers from Chapter 2, the Static Pages controller does not use the standard REST actions. This is normal for a collection of static pages: the REST architecture isn't the best solution to every problem.

Listing 3.9: The Static Pages controller made by Listing 3.7.
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController
  def home
```

```
  end

  def help
  end
end
```

We see from the `class` keyword in Listing 3.9 that `static_pages_controller.rb` defines a *class*, in this case called `StaticPagesController`. Classes are simply a convenient way to organize *functions* (also called *methods*) like the `home` and `help` actions, which are defined using the `def` keyword. As discussed in Section 2.3.4, the angle bracket `<` indicates that `StaticPagesController` *inherits* from the Rails class `ApplicationController`; as we'll see in a moment, this means that our pages come equipped with a large amount of Rails-specific functionality. (We'll learn more about both classes and inheritance in Section 4.4.)

In the case of the Static Pages controller, both of its methods are initially empty:

```
def home
end

def help
end
```

In plain Ruby, these methods would simply do nothing. In Rails, the situation is different—`StaticPagesController` is a Ruby class, but because it inherits from `ApplicationController` the behavior of its methods is specific to Rails: when visiting the URL /static_pages/home, Rails looks in the Static Pages controller and executes the code in the `home` action, and then renders the *view* (the V in MVC from Section 1.2.3) corresponding to the action. In the present case, the `home` action is empty, so all visiting /static_pages/home does is render the view. So, what does a view look like, and how do we find it?

If you take another look at the output in Listing 3.7, you might be able to guess the correspondence between actions and views: an action like `home` has a corresponding view called `home.html.erb`. We'll learn in Section 3.4 what the `.erb` part means; from the `.html` part you probably won't be surprised that it basically looks like HTML (Listing 3.10).

Listing 3.10: The generated view for the Home page.
app/views/static_pages/home.html.erb

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

The view for the **help** action is analogous (Listing 3.11).

Listing 3.11: The generated view for the Help page.
app/views/static_pages/help.html.erb

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

Both of these views are just placeholders: they have a top-level heading (inside the **h1** tag) and a paragraph (**p** tag) with the full path to the corresponding file.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Generate a controller called **Foo** with actions **bar** and **baz**.
2. By applying the techniques described in Box 3.1, destroy the **Foo** controller and its associated actions.

3.2.2 Custom static pages

We'll add some (very slightly) dynamic content starting in Section 3.4, but as they stand the files shown in Listing 3.10 and Listing 3.11 underscore an important point: Rails views can simply contain static HTML. This means we can begin customizing the Home and Help pages even with no knowledge of Rails, as shown in Listing 3.12 and Listing 3.13.

Listing 3.12: Custom HTML for the Home page.
app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listing 3.13: Custom HTML for the Help page.
app/views/static_pages/help.html.erb

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="https://www.railstutorial.org/help">Rails Tutorial Help page</a>.
  To get help on this sample app, see the
  <a href="https://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

The results of Listing 3.12 and Listing 3.13 are shown in Figure 3.5 and Figure 3.6.

3.3 Getting started with testing

Having created and filled in the Home and Help pages for our sample app (Section 3.2.2), now we're going to add an About page as well. When making a change of this nature, it's a good practice to write an *automated test* to verify that the feature is implemented correctly. Developed over the course of building an application, the resulting *test suite* serves as a safety net and as executable documentation of the application source code. When done right, writing tests also allows us to develop *faster* despite requiring extra code, because we'll end up wasting less time trying to track down bugs. This is true only once we get good at writing tests, though, which is one reason it's important to start practicing as early as possible.