Chapter 5

Filling in the layout

In the process of taking a brief tour of Ruby in Chapter 4, we learned about including the application stylesheet into the sample application (Section 4.1), but (as noted in Section 4.3.4) the stylesheet doesn't yet contain any CSS. In this chapter, we'll start filling in the custom stylesheet by incorporating a CSS framework into our application, and then we'll add some custom styles of our own.[1] We'll also start filling in the layout with links to the pages (such as Home and About) that we've created so far (Section 5.1). Along the way, we'll learn about partials, Rails routes, and the asset pipeline, including an introduction to Sass (Section 5.2). We'll end by taking a first important step toward letting users sign up to our site (Section 5.4).

Most of the changes in this chapter involve adding and editing markup in the sample application's site layout, which (based on the guidelines in Box 3.3) is exactly the kind of work that we wouldn't ordinarily test-drive, or even test at all. As a result, we'll spend most of our time in our text editor and browser, using TDD only to add a Contact page (Section 5.3.1). We will add an important new test, though, writing our first *integration test* to check that the links on the final layout are correct (Section 5.3.4).

[1] Thanks to reader Colm Tuite for his excellent work in helping to convert the sample application over to the Bootstrap CSS framework.

5.1 Adding some structure

The *Ruby on Rails Tutorial* is a book on web development, not web design, but it would be depressing to work on an application that looks like *complete* crap, so in this section we'll add some structure to the layout and give it some minimal styling with CSS. In addition to using some custom CSS rules, we'll make use of *Bootstrap*, an open-source web design framework from Twitter.[2] We'll also give our *code* some styling, so to speak, using *partials* to tidy up the layout once it gets a little cluttered.

When building web applications, it is often useful to get a high-level overview of the user interface as early as possible. Throughout the rest of this book, I will thus often include *mockups* (in a web context often called *wireframes*), which are rough sketches of what the eventual application will look like.[3] In this chapter, we will principally be developing the static pages introduced in Section 3.2, including a site logo, a navigation header, and a site footer. A mockup for the most important of these pages, the Home page, appears in Figure 5.1. You can see the final result in Figure 5.9. You'll note that it differs in some details—for example, we'll end up adding a Rails logo on the page—but that's fine, since a mockup need not be exact.

As usual, if you're using Git for version control, now would be a good time to make a new branch:

```
$ git checkout -b filling-in-layout
```

5.1.1 Site navigation

As a first step toward adding links and styles to the sample application, we'll update the site layout file **`application.html.erb`** (last seen in Listing 4.3) with additional HTML structure. This includes some additional divisions, some CSS

[2] Although more recent versions of Bootstrap are now available, this tutorial standardizes on Bootstrap 3 in order to retain compatibility with the design and HTML structure from previous editions.

[3] The mockups in the *Ruby on Rails Tutorial* are made with an excellent online mockup application called Mockingbird.

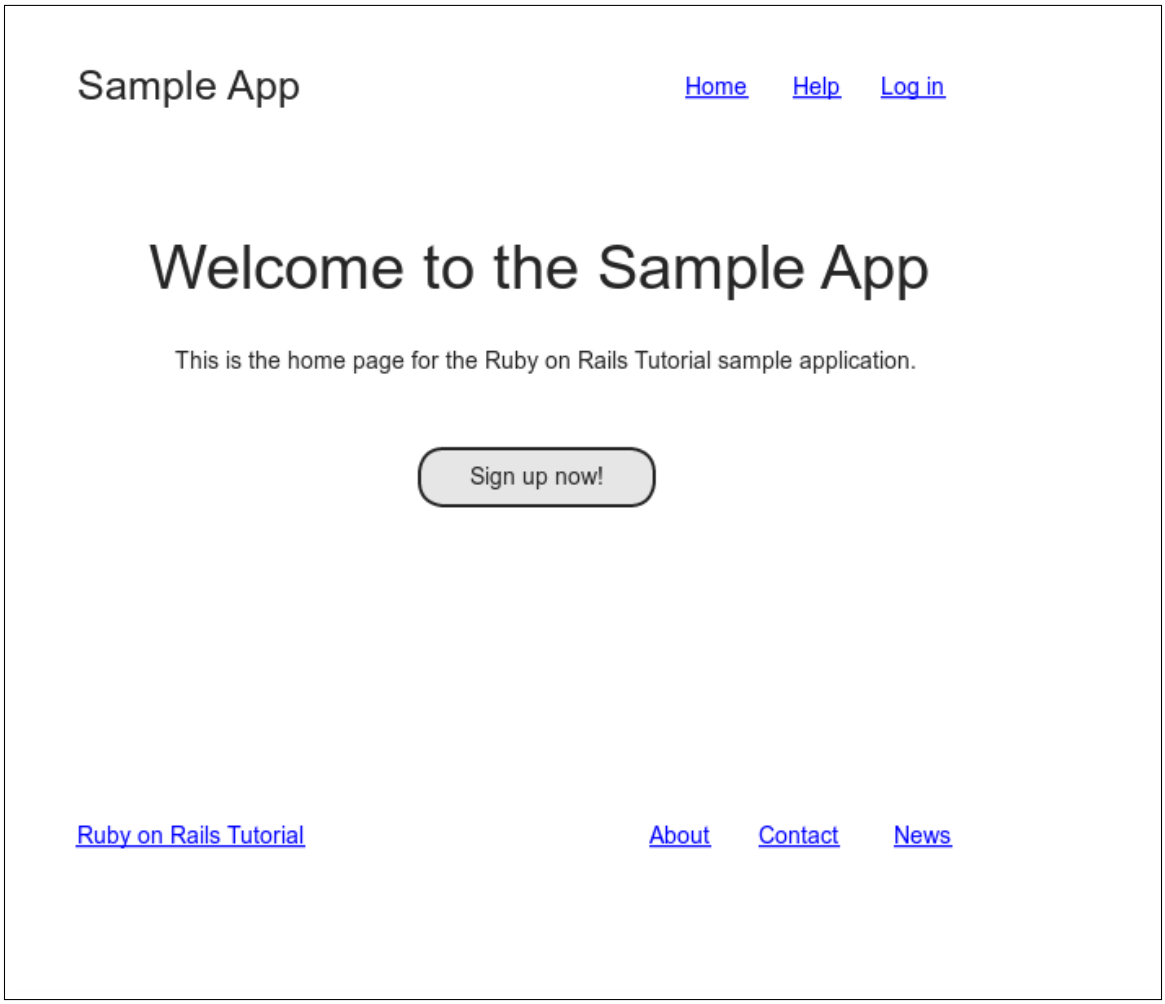

Figure 5.1: A mockup of the sample application's Home page.

classes, and the start of our site navigation. The full file is in Listing 5.1; explanations for the various pieces follow immediately thereafter. If you'd rather not delay gratification, you can see the results in Figure 5.2. (*Note*: it's not (yet) very gratifying.)

Listing 5.1: The site layout with added structure.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>

    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
    <!--[if lt IE 9]>
      <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
      </script>
    <![endif]-->
  </head>
  <body>
    <header class="navbar navbar-fixed-top navbar-inverse">
      <div class="container">
        <%= link_to "sample app", '#', id: "logo" %>
        <nav>
          <ul class="nav navbar-nav navbar-right">
            <li><%= link_to "Home",    '#' %></li>
            <li><%= link_to "Help",    '#' %></li>
            <li><%= link_to "Log in", '#' %></li>
          </ul>
        </nav>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

Let's look at the new elements in Listing 5.1 from top to bottom. As alluded to briefly in Section 3.4.1, Rails uses HTML5 by default (as indicated by the doctype **`<!DOCTYPE html>`**) which at this point most browsers support, but we

can make our site more accessible to older browsers by adding some JavaScript code, known as an "HTML5 shim (or shiv)":[4]

```
<!--[if lt IE 9]>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
  </script>
<![endif]-->
```

The somewhat odd syntax

```
<!--[if lt IE 9]>
```

includes the enclosed line only if the version of Microsoft Internet Explorer (IE) is less than 9 (`if lt IE 9`). The weird `[if lt IE 9]` syntax is *not* part of Rails; it's actually a conditional comment supported by Internet Explorer browsers for just this sort of situation. It's a good thing, too, because it means we can include the HTML5 shim *only* for IE browsers less than version 9, leaving other browsers such as Firefox, Chrome, and Safari unaffected.

The next section includes a `header` for the site's (plain-text) logo, a couple of divisions (using the `div` tag), and a list of elements with navigation links:

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", '#', id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home",    '#' %></li>
        <li><%= link_to "Help",    '#' %></li>
        <li><%= link_to "Log in", '#' %></li>
      </ul>
    </nav>
  </div>
</header>
```

[4] The words *shim* and *shiv* are used interchangably in this context; the former is the proper term, based on the English word whose meaning is "a washer or thin strip of material used to align parts, make them fit, or reduce wear", while the latter (meaning "a knife or razor used as a weapon") is apparently a play on the name of the shim's original author, Sjoerd Visscher.

Here the **header** tag indicates elements that should go at the top of the page. We've given the **header** tag three *CSS classes*,[5] called **navbar**, **navbar-fixed-top**, and **navbar-inverse**, separated by spaces:

```
<header class="navbar navbar-fixed-top navbar-inverse">
```

All HTML elements can be assigned both classes and *ids*;[6] these are merely labels, and are useful for styling with CSS (Section 5.1.2). The main difference between classes and ids is that classes can be used multiple times on a page, but ids can be used only once. In the present case, all the navbar classes have special meaning to the Bootstrap framework, which we'll install and use in Section 5.1.2.

Inside the **header** tag, we see a **div** tag:

```
<div class="container">
```

The **div** tag is a generic division; it doesn't do anything apart from divide the document into distinct parts. In older-style HTML, **div** tags are used for nearly all site divisions, but HTML5 adds the **header**, **nav**, and **section** elements for divisions common to many applications. In this case, the **div** has a CSS class as well (**container**). As with the **header** tag's classes, this class has special meaning to Bootstrap.

After the div, we encounter some embedded Ruby:

```
<%= link_to "sample app", '#', id: "logo" %>
<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><%= link_to "Home",   '#' %></li>
    <li><%= link_to "Help",   '#' %></li>
    <li><%= link_to "Log in", '#' %></li>
  </ul>
</nav>
```

[5] These are completely unrelated to Ruby classes.

[6] Short for "identification" and pronounced as the separate letters "I D". The usual convention in English is to use all-caps ("ID"), reserving "id" for a term in Freudian psychoanalysis. Because HTML is usually typed in all lower-case letters, though, it's more common in this context to write "id" instead.

This uses the Rails helper `link_to` to create links (which we created directly with the anchor tag `a` in Section 3.2.2); the first argument to `link_to` is the link text, while the second is the URL. We'll fill in the URLs with *named routes* in Section 5.3.3, but for now we use the stub URL `'#'` commonly used in web design (i.e., `'#'` is just a "stub", or placeholder, for the real URL). The third argument is an options hash, in this case adding the CSS id `logo` to the sample app link. (The other three links have no options hash, which is fine since it's optional.) Rails helpers often take options hashes in this way, giving us the flexibility to add arbitrary HTML options without ever leaving Rails.

The second element inside the divs is a list of navigation links, made using the *unordered list* tag `ul`, together with the *list item* tag `li`:

```
<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><%= link_to "Home",    '#' %></li>
    <li><%= link_to "Help",    '#' %></li>
    <li><%= link_to "Log in", '#' %></li>
  </ul>
</nav>
```

The `<nav>` tag, though formally unnecessary here, is used to more clearly communicate the purpose of the navigation links. Meanwhile, the `nav`, `navbar-nav`, and `navbar-right` classes on the `ul` tag have special meaning to Bootstrap and will be styled automatically when we include the Bootstrap CSS in Section 5.1.2. As you can verify by inspecting the navigation in your browser,[7] once Rails has processed the layout and evaluated the embedded Ruby the list looks like this:[8]

```
<nav>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Home</a></li>
    <li><a href="#">Help</a></li>
    <li><a href="#">Log in</a></li>
  </ul>
</nav>
```

[7] All modern browsers have the capability to inspect the HTML source of a page. If you've never used a web inspector before, do a web search for something like "web inspector <name of browser>" to learn more.

[8] The spacing might look slightly different, which is fine because (as noted in Section 3.4.1) HTML is insensitive to whitespace.

This is the text that will be returned to the browser.

The final part of the layout is a **div** for the main content:

```
<div class="container">
  <%= yield %>
</div>
```

As before, the **container** class has special meaning to Bootstrap. As we learned in Section 3.4.3, the **yield** method inserts the contents of each page into the site layout.

Apart from the site footer, which we'll add in Section 5.1.3, our layout is now complete, and we can look at the results by visiting the Home page. To take advantage of the upcoming style elements, we'll add some extra elements to the **home.html.erb** view (Listing 5.2).

Listing 5.2: The Home page with a link to the signup page.
app/views/static_pages/home.html.erb

```
<div class="center jumbotron">
  <h1>Welcome to the Sample App</h1>

  <h2>
    This is the home page for the
    <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </h2>

  <%= link_to "Sign up now!", '#', class: "btn btn-lg btn-primary" %>
</div>

<%= link_to image_tag("rails.svg", alt: "Rails logo", width: "200"),
                      "https://rubyonrails.org/" %>
```

In preparation for adding users to our site in Chapter 7, the first **link_to** creates a stub link of the form

```
<a href="#" class="btn btn-lg btn-primary">Sign up now!</a>
```

In the **div** tag, the **jumbotron** CSS class has a special meaning to Bootstrap, as do the **btn**, **btn-lg**, and **btn-primary** classes in the signup button.

The second **link_to** shows off the **image_tag** helper, which takes as arguments the path to an image and an optional options hash, in this case setting the **alt** and **width** attributes of the image tag using symbols. For this to work, there needs to be an image called **rails.svg**, which you should download from the Learn Enough website at https://cdn.learnenough.com/rails.svg and place in the **app/assets/images/** directory.

If you're using the cloud IDE or another Unix-like system, you can accomplish this with the **curl** utility, as shown in Listing 5.3.[9]

Listing 5.3: Downloading an image.

```
$ curl -o app/assets/images/rails.svg -OL https://cdn.learnenough.com/rails.svg
```

Because we used the **image_tag** helper in Listing 5.2, Rails will automatically find any images in the **app/assets/images/** directory using the asset pipeline (Section 5.2).

Now we're finally ready to see the fruits of our labors. You may have to restart the Rails server to see the changes (Box 1.2), and the results should appear as shown in Figure 5.2.

To make the effects of **image_tag** clearer, let's look at the HTML it produces by inspecting the image in our browser:[10]

```
<img alt="Rails logo" width="200px" src="/assets/rails-<long string>.svg">
```

Here the **<long string>** is a random value added by Rails to ensure that the filename is unique, which causes browsers to load images properly when they have been updated (instead of retrieving them from the browser cache). Note that the **src** attribute *doesn't* include **images**, instead using an **assets** directory common to all assets (images, JavaScript, CSS, etc.). On the server, Rails

[9] See *Learn Enough Command Line to Be Dangerous* for more information about **curl**.

[10] You might notice that the **img** tag, rather than looking like `<img>...</img>`, instead looks like `<img ... />`. Tags that follow this form are known as *self-closing* tags.

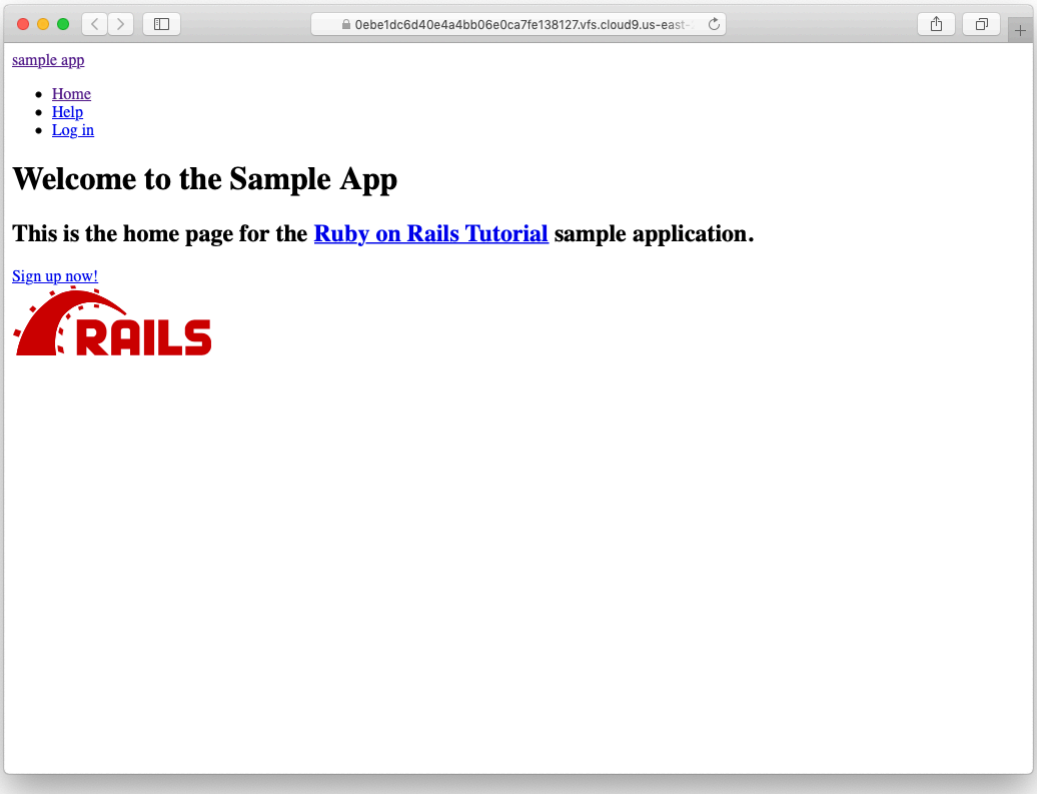

Figure 5.2: The Home page with no custom CSS.

associates images in the `assets` directory with the proper `app/assets/images` directory, but as far as the browser is concerned all the assets look like they are in the same directory, which allows them to be served faster. Meanwhile, the `alt` attribute is what will be displayed if the page is accessed by a program that can't display images (such as screen readers for the visually impaired).

As for the result shown in Figure 5.2, it might look a little underwhelming. Happily, though, we've done a good job of giving our HTML elements sensible classes, which puts us in a great position to add style to the site with CSS.

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. It's well-known that no web page is complete without a cat image. Using the command in Listing 5.4, arrange to download the kitten pic shown in Figure 5.3.[11]

2. Using the `mv` command, move `kitten.jpg` to the correct asset directory for images (Section 5.2.1).

3. Using `image_tag`, add `kitten.jpg` to the Home page, as shown in Figure 5.4.

Listing 5.4: Downloading a cat picture from the Internet.

```
$ curl -OL https://cdn.learnenough.com/kitten.jpg
```

[11] Image retrieved from https://www.flickr.com/photos/deborah_s_perspective/14144861329 on 2016-01-09. Copyright © 2009 by Deborah and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 5.3: An obligatory kitten pic.

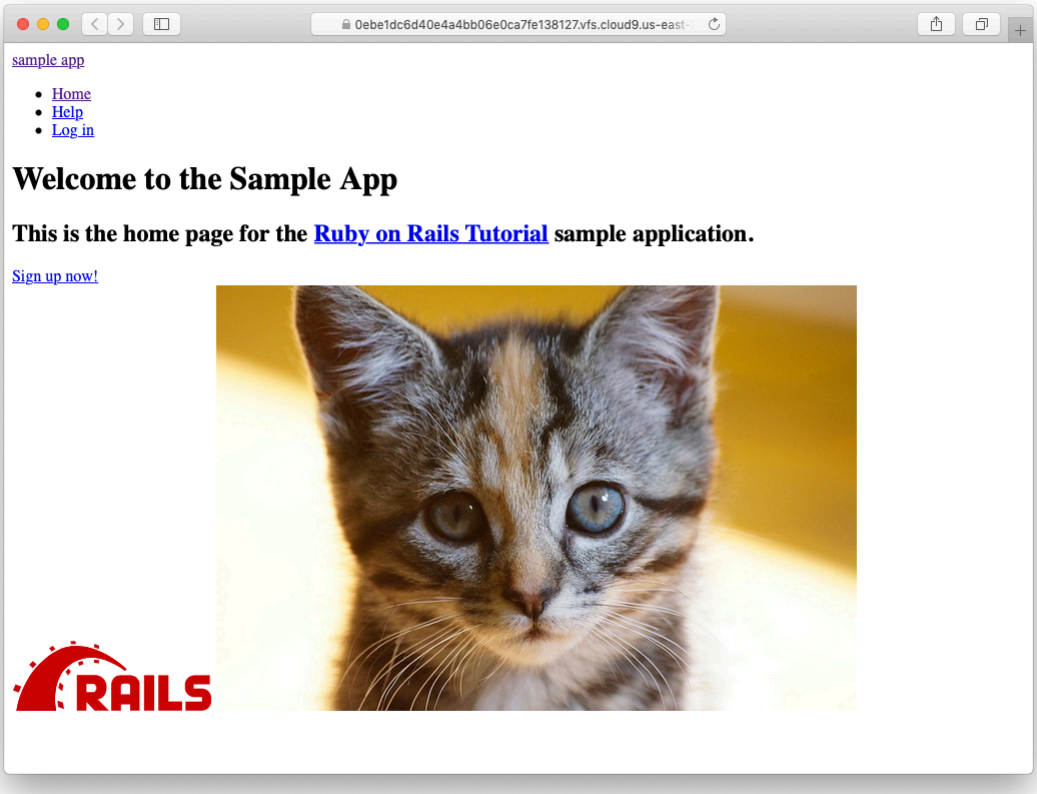

Figure 5.4: The result of adding a kitten image to the Home page.

5.1.2 Bootstrap and custom CSS

In Section 5.1.1, we associated many of the HTML elements with CSS classes, which gives us considerable flexibility in constructing a layout based on CSS. As noted in Section 5.1.1, many of these classes are specific to Bootstrap, a CSS framework that makes it easy to add nice web design and user interface elements to an HTML5 application. In this section, we'll combine Bootstrap with some custom CSS rules to start adding some style to the sample application. It's worth noting that using Bootstrap automatically makes our application's design *responsive*, ensuring that it looks sensible across a wide range of devices.

Our first step is to add Bootstrap, which in Rails applications can be accomplished with the `bootstrap-sass` gem, as shown in Listing 5.5.[12] The Bootstrap framework natively uses the Less CSS language for making dynamic stylesheets, but the Rails asset pipeline supports the (very similar) Sass language by default (Section 5.2), so `bootstrap-sass` converts Less to Sass and makes all the necessary Bootstrap files available to the current application.

Listing 5.5: Adding the `bootstrap-sass` gem to the **`Gemfile`**.

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

gem 'rails',          '6.0.1'
gem 'bootstrap-sass', '3.4.1'
gem 'puma',           '3.12.1'
.
.
.
```

To install Bootstrap, we run **`bundle install`** as usual:

```
$ bundle install
```

Although **`rails generate`** automatically creates a separate CSS file for each controller, it's surprisingly hard to include them all properly and in the

[12] As always, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed here.

right order, so for simplicity we'll put all of the CSS needed for this tutorial in a single file. The first step toward getting custom CSS to work is to create such a custom CSS file:

```
$ touch app/assets/stylesheets/custom.scss
```

(This uses the **touch** trick from Section 3.3.3 en route, but you can create the file however you like.) Here both the directory name and filename extension are important. The directory

```
app/assets/stylesheets/
```

is part of the asset pipeline (Section 5.2), and any stylesheets in this directory will automatically be included as part of the **application.css** file included in the site layout. Furthermore, the filename **custom.scss** includes the **.scss** extension, which indicates a "Sassy CSS" file and arranges for the asset pipeline to process the file using Sass. (We won't be using Sass until Section 5.2.2, but it's needed now for the `bootstrap-sass` gem to work its magic.)

Inside the file for the custom CSS, we can use the **@import** function to include Bootstrap (together with the associated Sprockets utility), as shown in Listing 5.6.[13]

Listing 5.6: Adding Bootstrap CSS.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";
```

The two lines in Listing 5.6 include the entire Bootstrap CSS framework. After restarting the webserver to incorporate the changes into the development application (by pressing Ctrl-C and then running **rails server** as in Section 1.2.2), the results appear as in Figure 5.5. The placement of the text isn't

[13] If these steps seem mysterious, take heart: I'm just following the instructions from the bootstrap-sass README file.

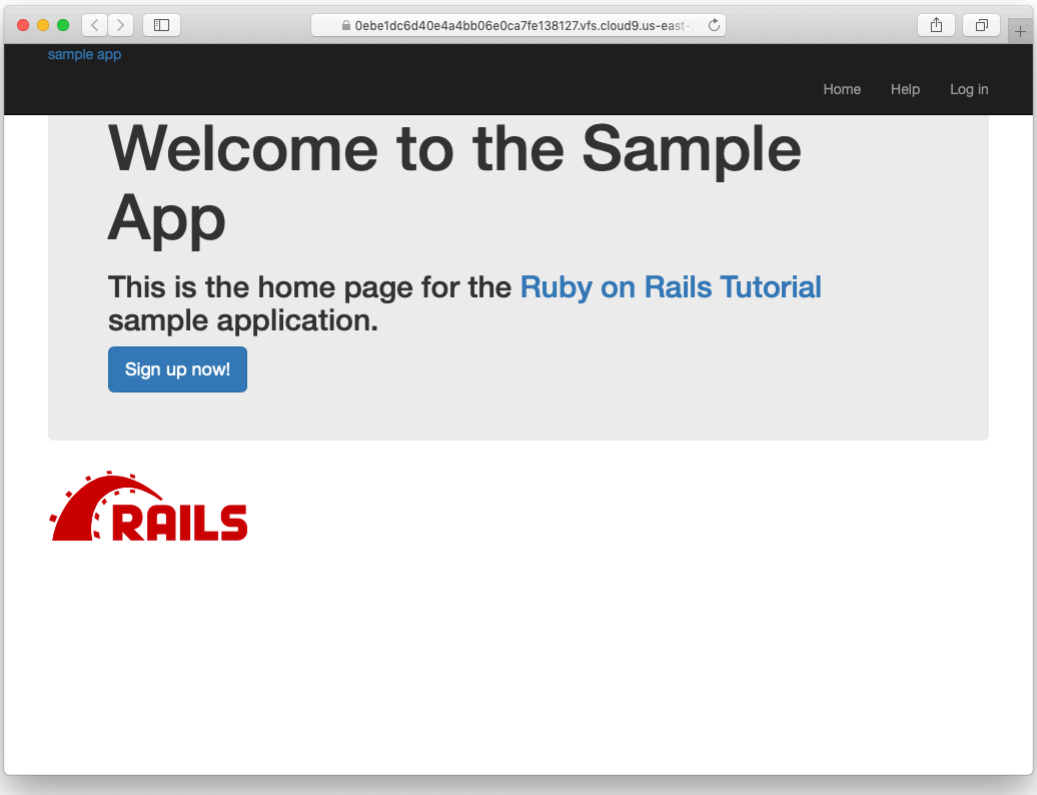

Figure 5.5: The sample application with Bootstrap CSS.

good and the logo doesn't have any style, but the colors and signup button look promising.

Next we'll add some CSS that will be used site-wide for styling the layout and each individual page, as shown in Listing 5.7. The result is shown in Figure 5.6. (There are quite a few rules in Listing 5.7; to get a sense of what a CSS rule does, it's often helpful to comment it out using CSS comments, i.e., by putting it inside `/* ... */`, and seeing what changes.)

Listing 5.7: Adding CSS for some universal styling applying to all pages.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* universal */

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

Note that the CSS in Listing 5.7 has a consistent form. In general, CSS rules refer either to a class, an id, an HTML tag, or some combination thereof, followed by a list of styling commands. For example,

```
body {
  padding-top: 60px;
}
```

puts 60 pixels of padding at the top of the page. Because of the **navbar-fixed-top** class in the **header** tag, Bootstrap fixes the navigation bar to the top of the page, so the padding serves to separate the main text from the navigation. (Because the default navbar color changed after Bootstrap 2.0, we need the **navbar-inverse** class to make it dark instead of light.) Meanwhile, the CSS in the rule

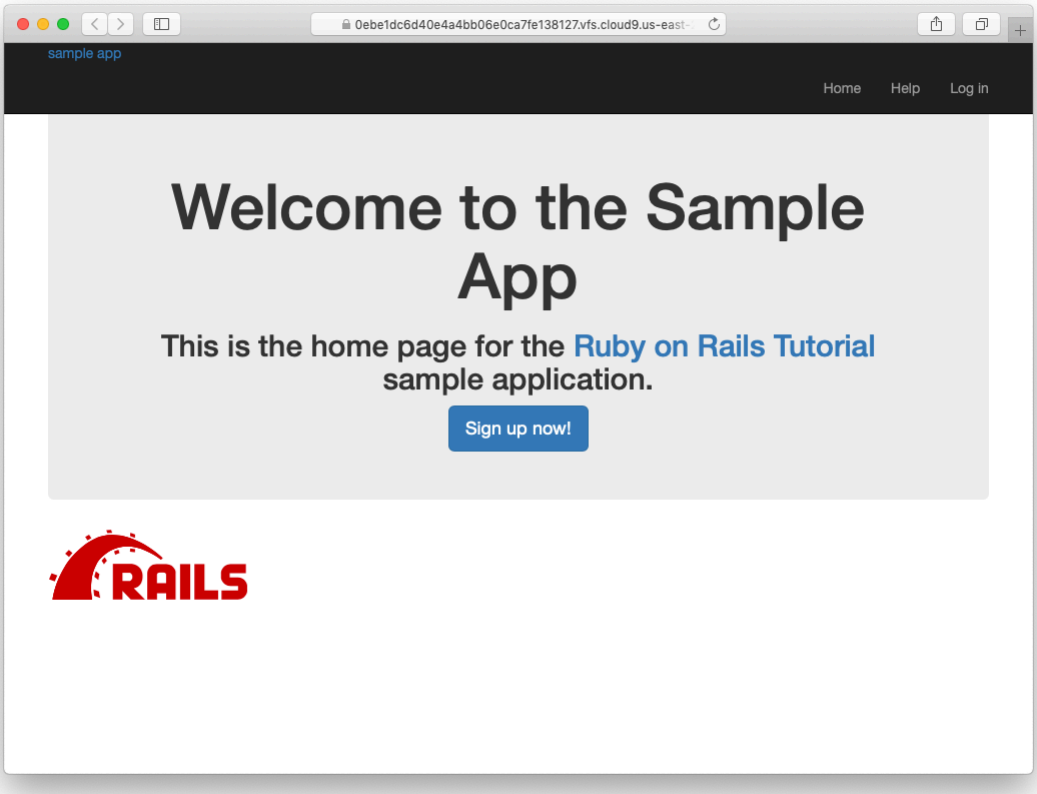

Figure 5.6: Adding some spacing and other universal styling.

```
.center {
  text-align: center;
}
```

associates the **center** class with the **text-align: center** property. In other words, the dot **.** in **.center** indicates that the rule styles a class. (As we'll see in Listing 5.9, the pound sign **#** identifies a rule to style a CSS *id*.) This means that elements inside any tag (such as a **div**) with class **center** will be centered on the page. (We saw an example of this class in Listing 5.2.)

Although Bootstrap comes with CSS rules for nice typography, we'll also add some custom rules for the appearance of the text on our site, as shown in Listing 5.8. (Not all of these rules apply to the Home page, but each rule here will be used at some point in the sample application.) The result of Listing 5.8 is shown in Figure 5.7.

Listing 5.8: Adding CSS for nice typography.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";
.
.
.
/* typography */

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: #777;
```

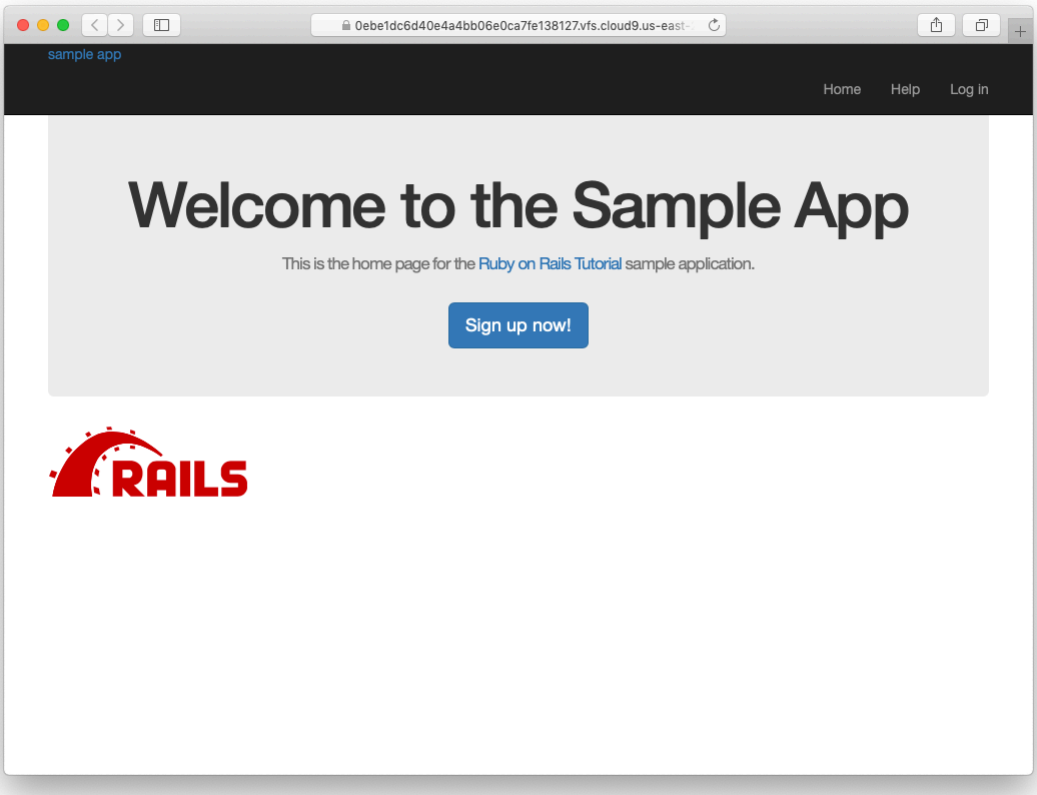

Figure 5.7: Adding some typographic styling.

```
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}
```

Finally, we'll add some rules to style the site's logo, which simply consists of the text "sample app". The CSS in Listing 5.9 converts the text to uppercase and modifies its size, color, and placement. (We've used a CSS id because we expect the site logo to appear on the page only once, but you could use a class instead.)

Listing 5.9: Adding CSS for the site logo.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";
.
.
.
/* header */

#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

Here `color: #fff` changes the color of the logo to white. HTML colors can be coded with three pairs of base-16 (hexadecimal) numbers, one each for the primary colors red, green, and blue (in that order). The code `#ffffff` maxes out all three colors, yielding pure white, and `#fff` is a shorthand for the full `#ffffff`. The CSS standard also defines a large number of synonyms for common HTML colors, including `white` for `#fff`. The result of the CSS in Listing 5.9 is shown in Figure 5.8.

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. Using code like that shown in Listing 5.10, comment out the cat image

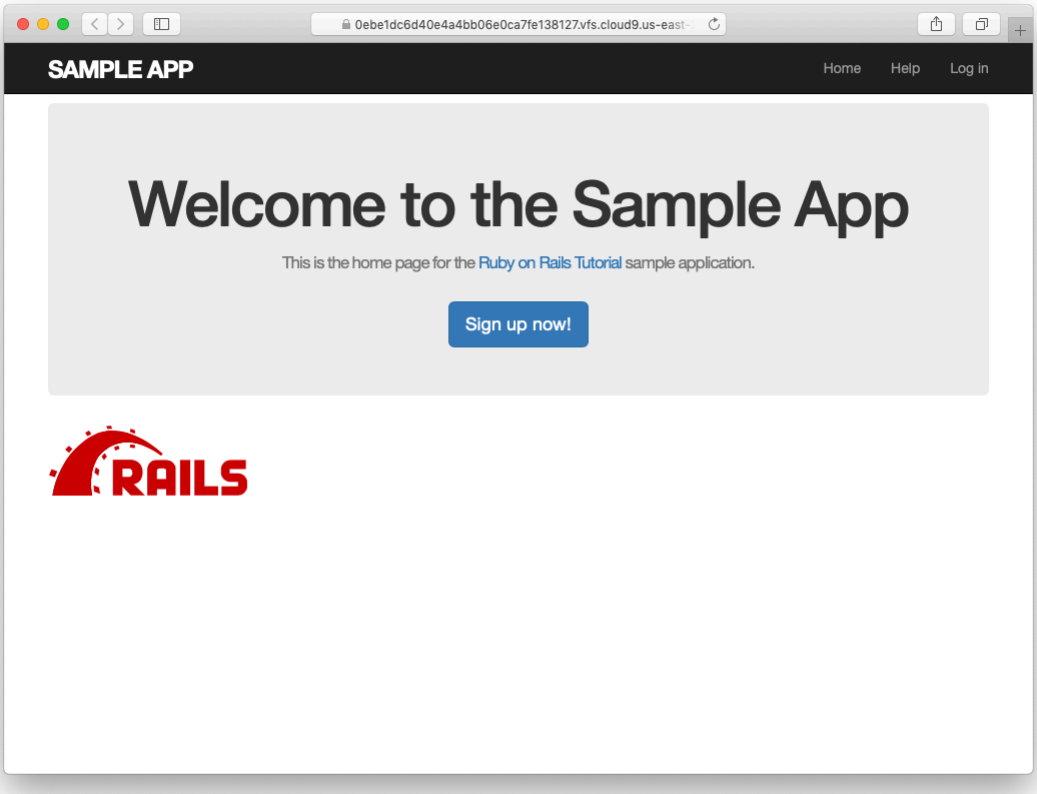

Figure 5.8: The sample app with nicely styled logo.

from Section 5.1.1. Verify using a web inspector that the HTML for the image no longer appears in the page source.

2. By adding the CSS in Listing 5.11 to **`custom.scss`**, hide all images in the application—currently just the Rails logo on the Home page). Verify with a web inspector that, although the image doesn't appear, the HTML source is still present.

Listing 5.10: Code to comment out embedded Ruby.

```
<%#= image_tag("kitten.jpg", alt: "Kitten") %>
```

Listing 5.11: CSS to hide all images.

```
img {
  display: none;
}
```

5.1.3 Partials

Although the layout in Listing 5.1 serves its purpose, it's getting a little cluttered. The HTML shim takes up three lines and uses weird IE-specific syntax, so it would be nice to tuck it away somewhere on its own. In addition, the header HTML forms a logical unit, so it should all be packaged up in one place. The way to achieve this in Rails is to use a facility called *partials*. Let's first take a look at what the layout looks like after the partials are defined (Listing 5.12).

Listing 5.12: The site layout with partials for the stylesheets and header.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
```

```
    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

In Listing 5.12, we've replaced the HTML shim stylesheet lines with a single call to a Rails helper called **render**:

```
<%= render 'layouts/shim' %>
```

The effect of this line is to look for a file called **app/views/layouts/_shim.html.erb**, evaluate its contents, and insert the results into the view.[14] (Recall that **<%= ... %>** is the embedded Ruby syntax needed to evaluate a Ruby expression and then insert the results into the template.) Note the leading underscore on the filename **_shim.html.erb**; this underscore is the universal convention for naming partials, and among other things makes it possible to identify all the partials in a directory at a glance.

To get the partial to work, we have to create the corresponding file and fill it with some content. In the case of the shim partial, this is just the three lines of shim code from Listing 5.1. The result appears in Listing 5.13.

Listing 5.13: A partial for the HTML shim.
app/views/layouts/_shim.html.erb

```
<!--[if lt IE 9]>
  <script src="//cdnjs.cloudflare.com/ajax/libs/html5shiv/r29/html5.min.js">
  </script>
<![endif]-->
```

[14] Many Rails developers use a **shared** directory for partials shared across different views. I prefer to use the **shared** folder for utility partials that are useful on multiple views, while putting partials that are literally on every page (as part of the site layout) in the **layouts** directory. (We'll create the **shared** directory starting in Chapter 7.) That seems to me a logical division, but putting them all in the **shared** folder certainly works fine, too.

Similarly, we can move the header material into the partial shown in Listing 5.14 and insert it into the layout with another call to **render**. (As usual with partials, you will have to create the file by hand using your text editor.)

Listing 5.14: A partial for the site header.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", '#', id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home",    '#' %></li>
        <li><%= link_to "Help",    '#' %></li>
        <li><%= link_to "Log in", '#' %></li>
      </ul>
    </nav>
  </div>
</header>
```

Now that we know how to make partials, let's add a site footer to go along with the header. By now you can probably guess that we'll call it **_footer.html.erb** and put it in the layouts directory (Listing 5.15).[15]

Listing 5.15: A partial for the site footer.
app/views/layouts/_footer.html.erb

```
<footer class="footer">
  <small>
    The <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    by <a href="https://www.michaelhartl.com/">Michael Hartl</a>
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   '#' %></li>
      <li><%= link_to "Contact", '#' %></li>
      <li><a href="https://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

[15] You may wonder why we use both the **footer** tag and **.footer** class. The answer is that the tag has a clear meaning to human readers, and the class is used by Bootstrap. Using a **div** tag in place of **footer** would work as well.

As with the header, in the footer we've used `link_to` for the internal links to the About and Contact pages and stubbed out the URLs with `'#'` for now. (As with `header`, the `footer` tag is new in HTML5.)

We can render the footer partial in the layout by following the same pattern as the stylesheets and header partials (Listing 5.16).

Listing 5.16: The site layout with a footer partial.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                           'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
    </div>
  </body>
</html>
```

Next, we'll add some styling for the footer, as shown in Listing 5.17. The results appear in Figure 5.9.

Listing 5.17: Adding the CSS for the site footer.
app/assets/stylesheets/custom.scss

```
.
.
.
/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
```

```
  border-top: 1px solid #eaeaea;
  color: #777;
}

footer a {
  color: #555;
}

footer a:hover {
  color: #222;
}

footer small {
  float: left;
}

footer ul {
  float: right;
  list-style: none;
}

footer ul li {
  float: left;
  margin-left: 15px;
}
```

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. Replace the default Rails head with the call to **`render`** shown in Listing 5.18. *Hint*: For convenience, cut the default header rather than just deleting it.

2. Because we haven't yet created the partial needed by Listing 5.18, the tests should be RED. Confirm that this is the case.

3. Create the necessary partial in the **`layouts`** directory, paste in the contents, and verify that the tests are now GREEN again.

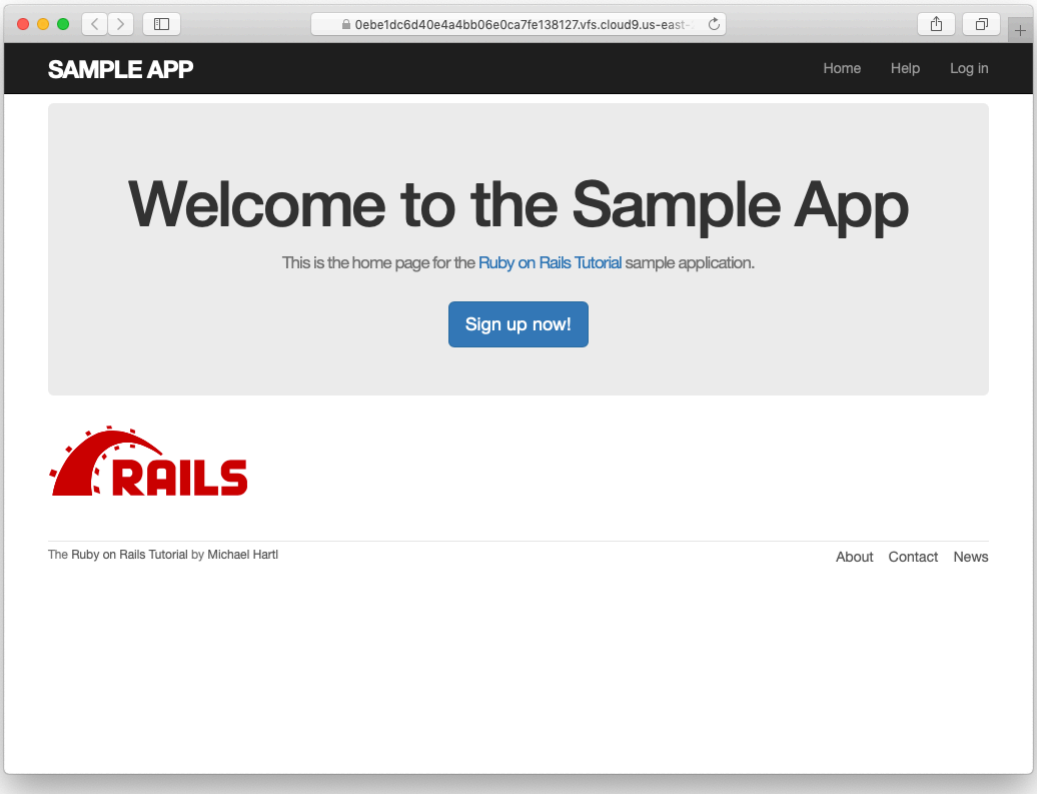

Figure 5.9: The Home page with an added footer.

Listing 5.18: Replacing the default Rails head with a call to `render`.
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= full_title(yield(:title)) %></title>
    <%= render 'layouts/rails_default' %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
    </div>
  </body>
</html>
```

5.2 Sass and the asset pipeline

One of the most useful features of Rails is the *asset pipeline*, which significantly simplifies the production and management of static assets such as CSS and images. The asset pipline also works well in parallel with Webpack (a JavaScript asset bundler) and Yarn (a dependency manager mentioned in Section 1.1.2), both of which are supported by default in Rails. This section first gives a high-level overview of the asset pipeline, and then shows how to use *Sass*, a powerful tool for writing CSS.

5.2.1 The asset pipeline

From the perspective of a typical Rails developer, there are three main features to understand about the asset pipeline: asset directories, manifest files, and preprocessor engines.[16] Let's consider each in turn.

[16]The original structure of this section was based on the excellent blog post "The Rails 3 Asset Pipeline in (about) 5 Minutes" by Michael Erasmus.

Asset directories

The Rails asset pipeline uses three standard directories for static assets, each with its own purpose:

- **`app/assets`**: assets specific to the present application
- **`lib/assets`**: assets for libraries written by your dev team
- **`vendor/assets`**: assets from third-party vendors (not present by default)

Each of these directories has a subdirectory for each of two asset classes—images and Cascading Style Sheets:

```
$ ls app/assets/
config  images  stylesheets
```

At this point, we're in a position to understand the motivation behind the location of the custom CSS file in Section 5.1.2: **`custom.scss`** is specific to the sample application, so it goes in **`app/assets/stylesheets`**.

Manifest files

Once you've placed your assets in their logical locations, you can use *manifest files* to tell Rails (via the Sprockets gem) how to combine them to form single files. (This applies to CSS and JavaScript but not to images.) As an example, let's take a look at the default manifest file for app stylesheets (Listing 5.19).

Listing 5.19: The manifest file for app-specific CSS.
`app/assets/stylesheets/application.css`

```
/*
 * This is a manifest file that'll be compiled into application.css, which will
 * include all the files listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets, or any
```

```
 * plugin's vendor/assets/stylesheets directory can be referenced here using a
 * relative path.
 *
 * You're free to add application-wide styles to this file and they'll appear at
 * the bottom of the compiled file so the styles you add here take precedence
 * over styles defined in any other CSS/SCSS files in this directory. Styles in
 * this file should be added after the last require_* statement.
 * It is generally better to create a new file per style scope.
 *
 *= require_tree .
 *= require_self
 */
```

The key lines here are actually CSS comments, but they are used by Sprockets to include the proper files:

```
/*
 .
 .
 .
 *= require_tree .
 *= require_self
*/
```

Here

```
*= require_tree .
```

ensures that all CSS files in the **app/assets/stylesheets** directory (including the tree subdirectories) are included into the application CSS. The line

```
*= require_self
```

specifies where in the loading sequence the CSS in **application.css** itself gets included.

Rails comes with sensible default manifest files, and in the *Rails Tutorial* we won't need to make any changes, but the Rails Guides entry on the asset pipeline has more detail if you need it.

Preprocessor engines

After you've assembled your assets, Rails prepares them for the site template by running them through several preprocessing engines and using the manifest files to combine them for delivery to the browser. We tell Rails which processor to use using filename extensions; the two most common cases are `.scss` for Sass and `.erb` for embedded Ruby (ERb). We first covered ERb in Section 3.4.3 and cover Sass in Section 5.2.2.

Efficiency in production

One of the best things about the asset pipeline is that it automatically results in assets that are optimized to be efficient in a production application. Traditional methods for organizing CSS involves splitting functionality into separate files and using nice formatting (with lots of indentation). While convenient for the programmer, this is inefficient in production. In particular, including multiple full-sized files can significantly slow page-load times, which is one of the most important factors affecting the quality of the user experience.

With the asset pipeline, we don't have to choose between speed and convenience: we can work with multiple nicely formatted files in development, and then use the asset pipeline to make efficient files in production. In particular, the asset pipeline combines all the application stylesheets into one CSS file (`application.css`) and then *minifies* it to remove the unnecessary spacing and indentation that bloats file size. The result is the best of both worlds: convenience in development and efficiency in production.

5.2.2 Syntactically awesome stylesheets

Sass is a language for writing stylesheets that improves on CSS in many ways. In this section, we cover two of the most important improvements, *nesting* and *variables*. (A third technique, *mixins*, is introduced in Section 7.1.1.)

As noted briefly in Section 5.1.2, Sass supports a format called SCSS (indicated with a `.scss` filename extension), which is a strict superset of CSS itself; that is, SCSS only *adds* features to CSS, rather than defining an entirely

new syntax.[17] This means that every valid CSS file is also a valid SCSS file, which is convenient for projects with existing style rules. In our case, we used SCSS from the start in order to take advantage of Bootstrap. Since the Rails asset pipeline automatically uses Sass to process files with the `.scss` extension, the `custom.scss` file will be run through the Sass preprocessor before being packaged up for delivery to the browser.

Nesting

A common pattern in stylesheets is having rules that apply to nested elements. For example, in Listing 5.7 we have rules both for `.center` and for `.center h1`:

```
.center {
  text-align: center;
}

.center h1 {
  margin-bottom: 10px;
}
```

We can replace this in Sass with

```
.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}
```

Here the nested `h1` rule automatically inherits the `.center` context.

There's a second candidate for nesting that requires a slightly different syntax. In Listing 5.9, we have the code

[17] Sass also supports an alternate syntax that does define a new language, which is less verbose (and has fewer curly braces) but is less convenient for existing projects and is harder to learn for those already familiar with CSS.

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
}

#logo:hover {
  color: #fff;
  text-decoration: none;
}
```

Here the logo id `#logo` appears twice, once by itself and once with the `hover` attribute (which controls its appearance when the mouse pointer hovers over the element in question). In order to nest the second rule, we need to reference the parent element `#logo`; in SCSS, this is accomplished with the ampersand character `&` as follows:

```
#logo {
  float: left;
  margin-right: 10px;
  font-size: 1.7em;
  color: #fff;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  &:hover {
    color: #fff;
    text-decoration: none;
  }
}
```

Sass changes `&:hover` into `#logo:hover` as part of converting from SCSS to CSS.

Both of these nesting techniques apply to the footer CSS in Listing 5.17, which can be transformed into the following:

```
footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #777;
  a {
    color: #555;
    &:hover {
      color: #222;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 15px;
    }
  }
}
```

Converting Listing 5.17 by hand is a good exercise (Section 5.2.2), and you should verify that the CSS still works properly after the conversion.

Variables

Sass allows us to define *variables* to eliminate duplication and write more expressive code. For example, looking at Listing 5.8 and Listing 5.17, we see that there are repeated references to the same color:

```
h2 {
  .
  .
  .
  color: #777;
}
.
.
.
footer {
  .
  .
```

```
  .
  color: #777;
}
```

In this case, `#777` is a light gray, and we can give it a name by defining a variable as follows:

```
$light-gray: #777;
```

This allows us to rewrite our SCSS like this:

```
$light-gray: #777;
.
.
.
h2 {
  .
  .
  .
  color: $light-gray;
}
.
.
.
footer {
  .
  .
  .
  color: $light-gray;
}
```

Because variable names such as `$light-gray` are more descriptive than `#777`, it's often useful to define variables even for values that aren't repeated. Indeed, the Bootstrap framework defines a large number of variables for colors, available online on the Bootstrap page of Less variables. That page defines variables using Less, not Sass, but the `bootstrap-sass` gem provides the Sass equivalents. It is not difficult to guess the correspondence; where Less uses an "at" sign `@`, Sass uses a dollar sign `$`. For example, looking at the Bootstrap variable page, we see that there is a variable for light gray:

```
@gray-light: #777;
```

This means that, via the `bootstrap-sass` gem, there should be a corresponding SCSS variable **`$gray-light`**. We can use this to replace our custom variable, **`$light-gray`**, which gives

```
h2 {
  .
  .
  .
  color: $gray-light;
}
.
.
.
footer {
  .
  .
  .
  color: $gray-light;
}
```

Applying the Sass nesting and variable definition features to the full SCSS file gives the file in Listing 5.20. This uses both Sass variables (as inferred from the Bootstrap Less variable page) and built-in named colors (i.e., **`white`** for **`#fff`**). Note in particular the dramatic improvement in the rules for the **`footer`** tag.

Listing 5.20: The initial SCSS file converted to use nesting and variables.
app/assets/stylesheets/custom.scss

```
@import "bootstrap-sprockets";
@import "bootstrap";

/* mixins, variables, etc. */

$gray-medium-light: #eaeaea;

/* universal */

body {
```

```
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
  h1 {
    margin-bottom: 10px;
  }
}

/* typography */

h1, h2, h3, h4, h5, h6 {
  line-height: 1;
}

h1 {
  font-size: 3em;
  letter-spacing: -2px;
  margin-bottom: 30px;
  text-align: center;
}

h2 {
  font-size: 1.2em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: $gray-light;
}

p {
  font-size: 1.1em;
  line-height: 1.7em;
}


/* header */

#logo {
  float: left;
  margin-right: 10px;
```

```
  font-size: 1.7em;
  color: white;
  text-transform: uppercase;
  letter-spacing: -1px;
  padding-top: 9px;
  font-weight: bold;
  &:hover {
    color: white;
    text-decoration: none;
  }
}

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid $gray-medium-light;
  color: $gray-light;
  a {
    color: $gray;
    &:hover {
      color: $gray-darker;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 15px;
    }
  }
}
```

Sass gives us even more ways to simplify our stylesheets, but the code in Listing 5.20 uses the most important features and gives us a great start. See the Sass website for more details.

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. As suggested in Section 5.2.2, go through the steps to convert the footer CSS from Listing 5.17 to Listing 5.20 to SCSS by hand.

5.3 Layout links

Now that we've finished a site layout with decent styling, it's time to start filling in the links we've stubbed out with `'#'`. Because plain HTML is valid in Rails ERb templates, we could hard-code links like

```
<a href="/static_pages/about">About</a>
```

but that isn't the Rails Way™. For one, it would be nice if the URL for the about page were /about rather than /static_pages/about. Moreover, Rails conventionally uses *named routes*, which involves code like

```
<%= link_to "About", about_path %>
```

This way the code has a more transparent meaning, and it's also more flexible since we can change the definition of **`about_path`** and have the URL change everywhere **`about_path`** is used.

The full list of our planned links appears in Table 5.1, along with their mapping to URLs and routes. We took care of the first route in Section 3.4.4, and we'll have implemented all but the last one by the end of this chapter. (We'll make the last one in Chapter 8.)

5.3.1 Contact page

For completeness, we'll add the Contact page, which was left as an exercise in Chapter 3. The test appears as in Listing 5.21, which simply follows the model last seen in Listing 3.26.

Page	URL	Named route
Home	/	`root_path`
About	/about	`about_path`
Help	/help	`help_path`
Contact	/contact	`contact_path`
Sign up	/signup	`signup_path`
Log in	/login	`login_path`

Table 5.1: Route and URL mapping for site links.

Listing 5.21: A test for the Contact page. RED
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get static_pages_home_url
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end

  test "should get contact" do
    get static_pages_contact_url
    assert_response :success
    assert_select "title", "Contact | Ruby on Rails Tutorial Sample App"
  end
end
```

At this point, the tests in Listing 5.21 should be RED:

Listing 5.22: RED

```
$ rails test
```

The application code parallels the addition of the About page in Section 3.3: first we update the routes (Listing 5.23), then we add a **`contact`** action to the Static Pages controller (Listing 5.24), and finally we create a Contact view (Listing 5.25).

Listing 5.23: Adding a route for the Contact page. RED
config/routes.rb

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

Listing 5.24: Adding an action for the Contact page. RED
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController
  .
  .
  .
  def contact
  end
end
```

Listing 5.25: The view for the Contact page. GREEN
app/views/static_pages/contact.html.erb

```
<% provide(:title, 'Contact') %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="https://www.railstutorial.org/contact">contact page</a>.
</p>
```

Now make sure that the tests are GREEN:

Listing 5.26: GREEN

```
$ rails test
```

5.3.2 Rails routes

To add the named routes for the sample app's static pages, we'll edit the routes file, **`config/routes.rb`**, that Rails uses to define URL mappings. We'll begin by reviewing the route for the Home page (defined in Section 3.4.4), which is a special case, and then define a set of routes for the remaining static pages.

So far, we've seen three examples of how to define a root route, starting with the code

```
root 'application#hello'
```

in the hello app (Listing 1.11), the code

```
root 'users#index'
```

in the toy app (Listing 2.7), and the code

```
root 'static_pages#home'
```

in the sample app (Listing 3.43). In each case, the **`root`** method arranges for the root path / to be routed to a controller and action of our choice. Defining the root route in this way has a second important effect, which is to create named routes that allow us to refer to routes by a name rather than by the raw URL. In this case, these routes are **`root_path`** and **`root_url`**, with the only difference being that the latter includes the full URL:

```
root_path -> '/'
root_url  -> 'http://www.example.com/'
```

In the *Rails Tutorial*, we'll follow the common convention of using the **_path** form except when doing redirects, where we'll use the **_url** form. (This is because the HTTP standard technically requires a full URL after redirects, though in most browsers it will work either way.)

Because the default routes used in, e.g., Listing 5.21 are rather verbose, we'll also take this opportunity to define shorter named routes for the Help, About, and Contact pages. To do this, we need to make changes to the **get** rules from Listing 5.23, transforming lines like

```
get 'static_pages/help'
```

to

```
get  '/help', to: 'static_pages#help'
```

This new pattern routes a `GET` request for the URL /help to the **help** action in the Static Pages controller. As with the rule for the root route, this creates two named routes, **help_path** and **help_url**:

```
help_path -> '/help'
help_url  -> 'http://www.example.com/help'
```

Applying this rule change to the remaining static page routes from Listing 5.23 gives Listing 5.27.

Listing 5.27: Routes for static pages. RED

config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  '/help',    to: 'static_pages#help'
  get  '/about',   to: 'static_pages#about'
  get  '/contact', to: 'static_pages#contact'
end
```

Note that Listing 5.27 also removes the route for `'static_pages/home'`, as we'll always use `root_path` or `root_url` instead.

Because the tests in Listing 5.21 used the old routes, they are now RED. To get them GREEN again, we need to update the routes as shown in Listing 5.28. Note that we've taken this opportunity to update to the (optional) convention of using the `*_path` form of each named route.

Listing 5.28: The static pages tests with the new named routes. GREEN
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get root_path
    assert_response :success
    assert_select "title", "Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get help_path
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get about_path
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end

  test "should get contact" do
    get contact_path
    assert_response :success
    assert_select "title", "Contact | Ruby on Rails Tutorial Sample App"
  end
end
```

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. It's possible to use a named route other than the default using the **as:** option. Drawing inspiration from this famous *Far Side* comic strip, change the route for the Help page to use **helf** (Listing 5.29).
2. Confirm that the tests are now RED. Get them to GREEN by updating the route in Listing 5.28.
3. Revert the changes from these exercises using Undo.

Listing 5.29: Changing 'help' to 'helf'.

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  '/help',    to: 'static_pages#help', as: 'helf'
  get  '/about',   to: 'static_pages#about'
  get  '/contact', to: 'static_pages#contact'
end
```

5.3.3 Using named routes

With the routes defined in Listing 5.27, we're now in a position to use the resulting named routes in the site layout. This simply involves filling in the second arguments of the **link_to** functions with the proper named routes. For example, we'll convert

```
<%= link_to "About", '#' %>
```

to

```
<%= link_to "About", about_path %>
```

and so on.

We'll start in the header partial, **_header.html.erb** (Listing 5.30), which has links to the Home and Help pages. While we're at it, we'll follow a common web convention and link the logo to the Home page as well.

Listing 5.30: Header partial with links.
app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home",    root_path %></li>
        <li><%= link_to "Help",    help_path %></li>
        <li><%= link_to "Log in", '#' %></li>
      </ul>
    </nav>
  </div>
</header>
```

We won't have a named route for the "Log in" link until Chapter 8, so we've left it as `'#'` for now.

The other place with links is the footer partial, **`_footer.html.erb`**, which has links for the About and Contact pages (Listing 5.31).

Listing 5.31: Footer partial with links.
app/views/layouts/_footer.html.erb

```
<footer class="footer">
  <small>
    The <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    by <a href="https://www.michaelhartl.com/">Michael Hartl</a>
  </small>
  <nav>
    <ul>
      <li><%= link_to "About",   about_path %></li>
      <li><%= link_to "Contact", contact_path %></li>
      <li><a href="https://news.railstutorial.org/">News</a></li>
    </ul>
  </nav>
</footer>
```

With that, our layout has links to all the static pages created in Chapter 3, so that, for example, /about goes to the About page (Figure 5.10).

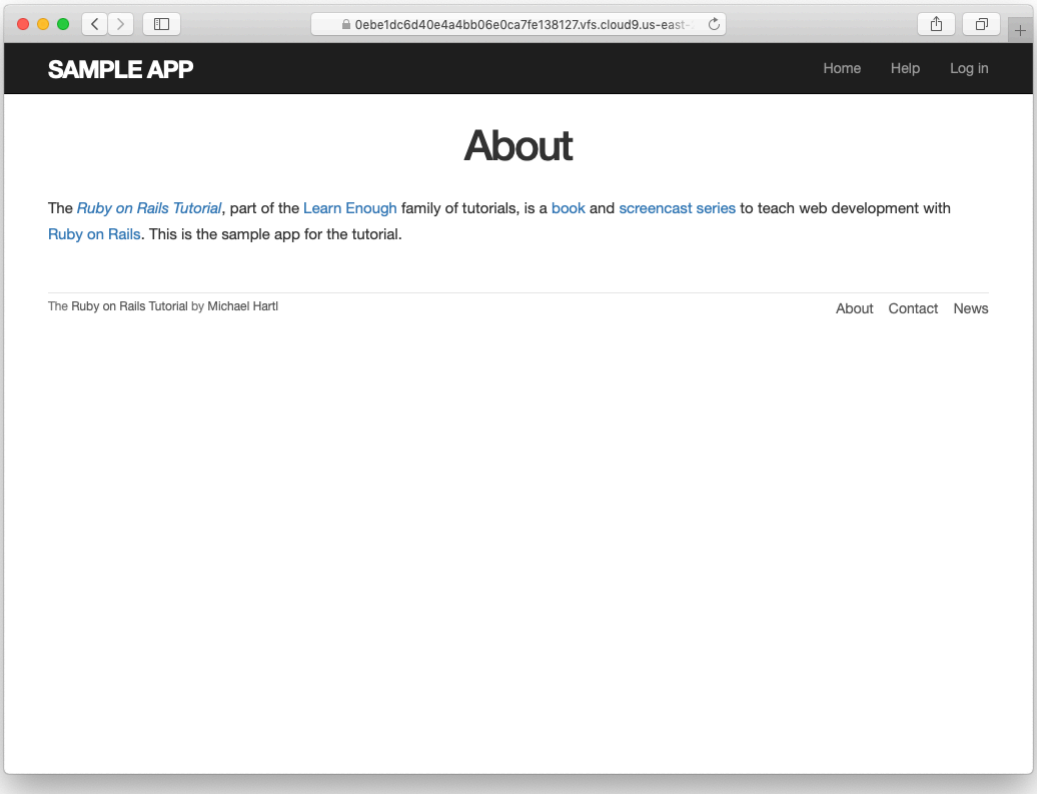

Figure 5.10: The About page at /about.

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. Update the layout links to use the `helf` route from Listing 5.29.
2. Revert the changes using Undo.

5.3.4 Layout link tests

Now that we've filled in several of the layout links, it's a good idea to test them to make sure they're working correctly. We could do this by hand with a browser, first visiting the root path and then checking the links by hand, but this quickly becomes cumbersome. Instead, we'll simulate the same series of steps using an *integration test*, which allows us to write an end-to-end test of our application's behavior. We can get started by generating a template test, which we'll call `site_layout`:

```
$ rails generate integration_test site_layout
      invoke  test_unit
      create    test/integration/site_layout_test.rb
```

Note that the Rails generator automatically appends `_test` to the name of the test file.

Our plan for testing the layout links involves checking the HTML structure of our site:

1. Get the root path (Home page).
2. Verify that the right page template is rendered.
3. Check for the correct links to the Home, Help, About, and Contact pages.

Listing 5.32 shows how we can use Rails integration tests to translate these steps into code, beginning with the **assert_template** method to verify that the Home page is rendered using the correct view.[18]

Listing 5.32: A test for the links on the layout. GREEN
test/integration/site_layout_test.rb

```
require 'test_helper'

class SiteLayoutTest < ActionDispatch::IntegrationTest

  test "layout links" do
    get root_path
    assert_template 'static_pages/home'
    assert_select "a[href=?]", root_path, count: 2
    assert_select "a[href=?]", help_path
    assert_select "a[href=?]", about_path
    assert_select "a[href=?]", contact_path
  end
end
```

Listing 5.32 uses some of the more advanced options of the **assert_select** method, seen before in Listing 3.26 and Listing 5.21. In this case, we use a syntax that allows us to test for the presence of a particular link–URL combination by specifying the tag name **a** and attribute **href**, as in

```
assert_select "a[href=?]", about_path
```

Here Rails automatically inserts the value of **about_path** in place of the question mark (escaping any special characters if necessary), thereby checking for an HTML tag of the form

[18] Some developers insist that a single test shouldn't contain multiple assertions. I find this practice to be unnecessarily complicated, while also incurring an extra overhead if there are common setup tasks needed before each test. In addition, a well-written test tells a coherent story, and breaking it up into individual pieces disrupts the narrative. I thus have a strong preference for including multiple assertions in a test, relying on Ruby (via minitest) to tell me the exact lines of any failed assertions.

Code	Matching HTML
`assert_select "div"`	`<div>foobar</div>`
`assert_select "div", "foobar"`	`<div>foobar</div>`
`assert_select "div.nav"`	`<div class="nav">foobar</div>`
`assert_select "div#profile"`	`<div id="profile">foobar</div>`
`assert_select "div[name=yo]"`	`<div name="yo">hey</div>`
`assert_select "a[href=?]", '/', count: 1`	`<a href="/">foo</a>`
`assert_select "a[href=?]", '/', text: "foo"`	`<a href="/">foo</a>`

Table 5.2: Some uses of `assert_select`.

```
<a href="/about">...</a>
```

Note that the assertion for the root path verifies that there are *two* such links (one each for the logo and navigation menu element):

```
assert_select "a[href=?]", root_path, count: 2
```

This ensures that both links to the Home page defined in Listing 5.30 are present.

Some more uses of `assert_select` appear in Table 5.2. While `assert_select` is flexible and powerful (having many more options than the ones shown here), experience shows that it's wise to take a lightweight approach by testing only HTML elements (such as site layout links) that are unlikely to change much over time.

To check that the new test in Listing 5.32 passes, we can run just the integration tests using the following Rake task:

Listing 5.33: GREEN

```
$ rails test:integration
```

If all went well, you should run the full test suite to verify that all the tests are GREEN:

Listing 5.34: GREEN

```
$ rails test
```

With the added integration test for layout links, we are now in a good position to catch regressions quickly using our test suite.

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. In the footer partial, change **`about_path`** to **`contact_path`** and verify that the tests catch the error.

2. It's convenient to use the **`full_title`** helper in the tests by including the Application helper into the test helper, as shown in Listing 5.35. We can then test for the right title using code like Listing 5.36. This is brittle, though, because now any typo in the base title (such as "Ruby on Rails Tutoial") won't be caught by the test suite. Fix this problem by writing a direct test of the **`full_title`** helper, which involves creating a file to test the application helper and then filling in the code indicated with **`FILL_IN`** in Listing 5.37. (Listing 5.37 uses **`assert_equal <expected>, <actual>`**, which verifies that the expected result matches the actual value when compared with the **`==`** operator.)

Listing 5.35: Including the Application helper in tests.
test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
.
.
.
class ActiveSupport::TestCase
  fixtures :all
```

```
  include ApplicationHelper
  .
  .
  .
end
```

Listing 5.36: Using the `full_title` helper in a test. GREEN
test/integration/site_layout_test.rb

```
require 'test_helper'

class SiteLayoutTest < ActionDispatch::IntegrationTest

  test "layout links" do
    get root_path
    assert_template 'static_pages/home'
    assert_select "a[href=?]", root_path, count: 2
    assert_select "a[href=?]", help_path
    assert_select "a[href=?]", about_path
    assert_select "a[href=?]", contact_path
    get contact_path
    assert_select "title", full_title("Contact")
  end
end
```

Listing 5.37: A direct test of the `full_title` helper.
test/helpers/application_helper_test.rb

```
require 'test_helper'

class ApplicationHelperTest < ActionView::TestCase
  test "full title helper" do
    assert_equal full_title,         FILL_IN
    assert_equal full_title("Help"), FILL_IN
  end
end
```

5.4 User signup: A first step

As a capstone to our work on the layout and routing, in this section we'll make a route for the signup page, which will mean creating a second controller along

the way. This is a first important step toward allowing users to register for our site; we'll take the next step, modeling users, in Chapter 6, and we'll finish the job in Chapter 7.

5.4.1 Users controller

We created our first controller, the Static Pages controller, in Section 3.2. It's time to create a second one, the Users controller. As before, we'll use **generate** to make the simplest controller that meets our present needs, namely, one with a stub signup page for new users. Following the conventional REST architecture favored by Rails, we'll call the action for new users **new**, which we can arrange to create automatically by passing **new** as an argument to **generate**. The result is shown in Listing 5.38.

Listing 5.38: Generating a Users controller (with a **new** action).

```
$ rails generate controller Users new
      create  app/controllers/users_controller.rb
       route  get 'users/new'
      invoke  erb
      create    app/views/users
      create    app/views/users/new.html.erb
      invoke  test_unit
      create    test/controllers/users_controller_test.rb
      invoke  helper
      create    app/helpers/users_helper.rb
      invoke    test_unit
      invoke  assets
      invoke    scss
      create      app/assets/stylesheets/users.scss
```

As required, Listing 5.38 creates a Users controller with a **new** action (Listing 5.39) and a stub user view (Listing 5.40). It also creates a minimal test for the new user page (Listing 5.41).

Listing 5.39: The initial Users controller, with a **new** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

  def new
  end
end
```

Listing 5.40: The initial `new` view for Users.
app/views/users/new.html.erb

```
<h1>Users#new</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

Listing 5.41: The generated test for the new user page. GREEN
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  test "should get new" do
    get users_new_url
    assert_response :success
  end
end
```

At this point, the tests should be GREEN:

Listing 5.42: GREEN

```
$ rails test
```

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. Per Table 5.1, change the route in Listing 5.41 to use **`signup_path`** instead of **`users_new_url`**.
2. The route in the previous exercise doesn't yet exist, so confirm that the tests are now RED. (This is intended to help us get comfortable with the RED/GREEN flow of Test Driven Development (TDD, Box 3.3); we'll get the tests back to GREEN in Section 5.4.2.)

5.4.2 Signup URL

With the code from Section 5.4.1, we already have a working page for new users at /users/new, but recall from Table 5.1 that we want the URL to be /signup instead. We'll follow the examples from Listing 5.27 and add a **`get '/signup'`** rule for the signup URL, as shown in Listing 5.43.

Listing 5.43: A route for the signup page. RED
config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  '/help',    to: 'static_pages#help'
  get  '/about',   to: 'static_pages#about'
  get  '/contact', to: 'static_pages#contact'
  get  '/signup',  to: 'users#new'
end
```

With the routes in Listing 5.43, we also need to update the test generated in Listing 5.38 with the new signup route, as shown in Listing 5.44.

Listing 5.44: Updating the Users controller test to use the signup route. GREEN
test/controllers/users_controller_test.rb

```
require 'test_helper'

class UsersControllerTest < ActionDispatch::IntegrationTest

  test "should get new" do
    get signup_path
    assert_response :success
  end
end
```

Next, we'll use the newly defined named route to add the proper link to the button on the Home page. As with the other routes, **`get 'signup'`** automatically gives us the named route **`signup_path`**, which we put to use in Listing 5.45. Adding a test for the signup page is left as an exercise (Section 5.3.2.)

Listing 5.45: Linking the button to the signup page.
app/views/static_pages/home.html.erb

```
<div class="center jumbotron">
  <h1>Welcome to the Sample App</h1>

  <h2>
    This is the home page for the
    <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
    sample application.
  </h2>

  <%= link_to "Sign up now!", signup_path, class: "btn btn-lg btn-primary" %>
</div>

<%= link_to image_tag("rails.svg", alt: "Rails logo", width: "200"),
                      "https://rubyonrails.org/" %>
```

Finally, we'll add a custom stub view for the signup page (Listing 5.46).

Listing 5.46: The initial (stub) signup page.
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
<p>This will be a signup page for new users.</p>
```

With that, we're done with the links and named routes, at least until we add a route for logging in (Chapter 8). The resulting new user page (at the URL /signup) appears in Figure 5.11.

Exercises

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

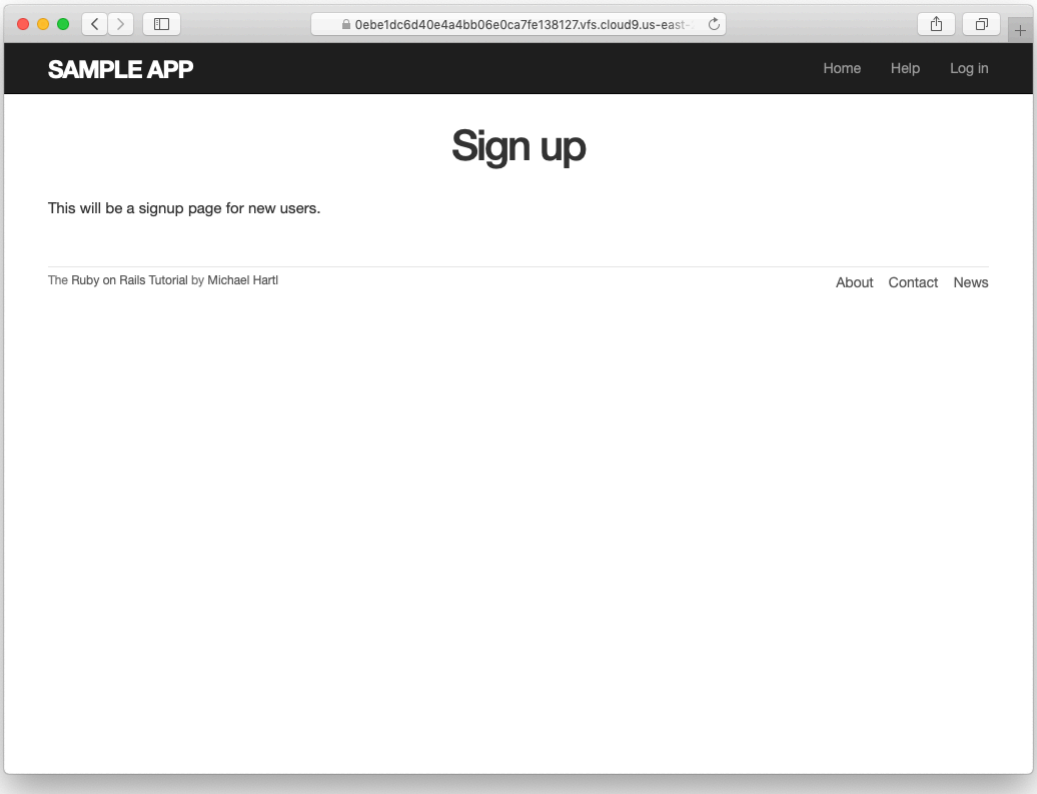

Figure 5.11: The new signup page at /signup.

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. If you didn't solve the exercise in Section 5.4.1, change the test in Listing 5.41 to use the named route **`signup_path`**. Because of the route defined in Listing 5.43, this test should initially be GREEN.

2. In order to verify the correctness of the test in the previous exercise, comment out the **`signup`** route to get to RED, then uncomment to get to GREEN.

3. In the integration test from Listing 5.32, add code to visit the signup page using the **`get`** method and verify that the resulting page title is correct. *Hint*: Use the **`full_title`** helper as in Listing 5.36.

5.5 Conclusion

In this chapter, we've hammered our application layout into shape and polished up the routes. The rest of the book is dedicated to fleshing out the sample application: first, by adding users who can sign up, log in, and log out; next, by adding user microposts; and, finally, by adding the ability to follow other users.

At this point, if you are using Git, you should merge your changes back into the master branch:

```
$ git add -A
$ git commit -m "Finish layout and routes"
$ git checkout master
$ git merge filling-in-layout
```

Then push up to GitHub (running the test suite first for safety):

```
$ rails test
$ git push
```

Finally, deploy to Heroku:

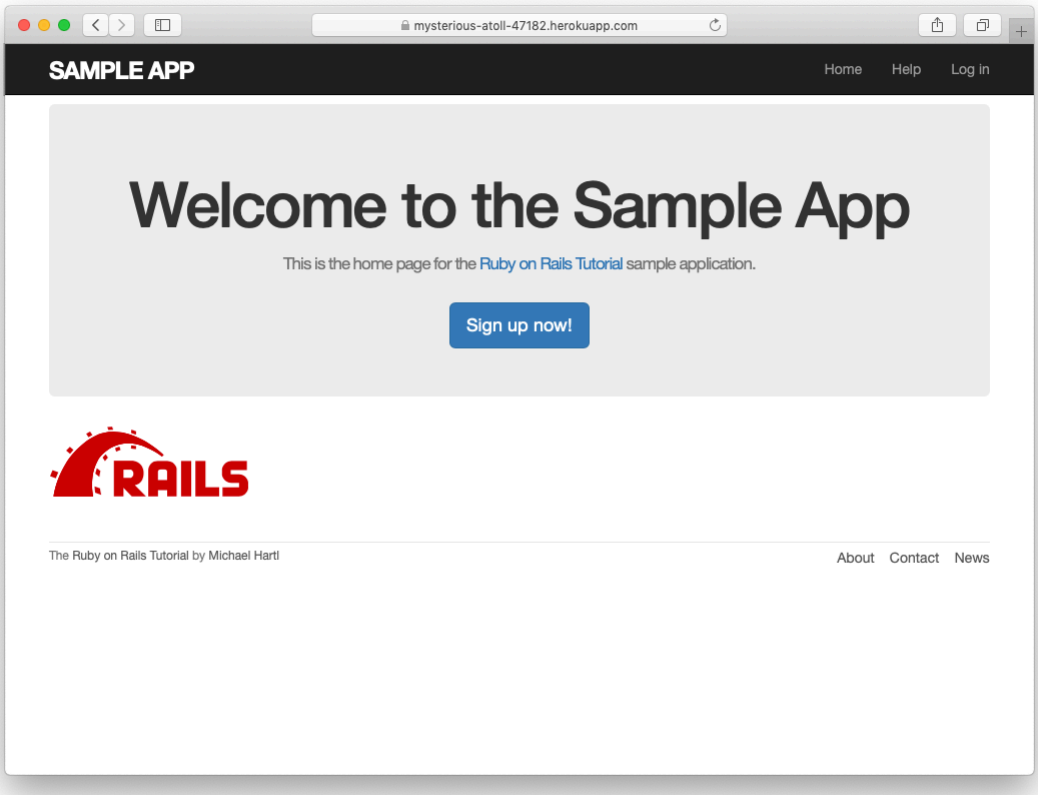

Figure 5.12: The sample application in production.

```
$ git push heroku
```

The result of the deployment should be a working sample application on the production server (Figure 5.12).

5.5.1 What we learned in this chapter

- Using HTML5, we can define a site layout with logo, header, footer, and main body content.

- Rails partials are used to place markup in a separate file for convenience.
- CSS allows us to style the site layout based on CSS classes and ids.
- The Bootstrap framework makes it easy to make a nicely designed site quickly.
- Sass and the asset pipeline allow us to eliminate duplication in our CSS while packaging up the results efficiently for production.
- Rails allows us to define custom routing rules, thereby providing named routes.
- Integration tests effectively simulate a browser clicking from page to page.