Ruby on Rails Tutorial

Learn Web Development with Rails

Michael Hartl

Contents

Foreword

My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP (Google me to read about the drama). This book by Michael Hartl came so highly recommended that I had to try it, and the *Ruby on Rails Tutorial* is what I used to switch back to Rails again.

Though I've worked my way through many Rails books, this is the one that finally made me "get" it. Everything is done very much "the Rails way"—a way that felt very unnatural to me before, but now after doing this book finally feels natural. This is also the only Rails book that does test-driven development the entire time, an approach highly recommended by the experts but which has never been so clearly demonstrated before. Finally, by including Git, GitHub, and Heroku in the demo examples, the author really gives you a feel for what it's like to do a real-world project. The tutorial's code examples are not in isolation.

The linear narrative is such a great format. Personally, I powered through the *Rails Tutorial* in three long days, doing all the examples and challenges at the end of each chapter. Do it from start to finish, without jumping around, and you'll get the ultimate benefit.

Enjoy!

Derek Sivers (sivers.org)
Formerly: Founder, CD Baby
Currently: Founder, Thoughts Ltd.

Acknowledgments

The *Ruby on Rails Tutorial* owes a lot to my previous Rails book, *RailsSpace*, and hence to my coauthor Aurelius Prochazka. I'd like to thank Aure both for the work he did on that book and for his support of this one. I'd also like to thank Debra Williams Cauley, my editor on both *RailsSpace* and the *Ruby on Rails Tutorial*; as long as she keeps taking me to baseball games, I'll keep writing books for her.

I'd like to acknowledge a long list of Rubyists who have taught and inspired me over the years: David Heinemeier Hansson, Yehuda Katz, Carl Lerche, Jeremy Kemper, Xavier Noria, Ryan Bates, Geoffrey Grosenbach, Peter Cooper, Matt Aimonetti, Gregg Pollack, Wayne E. Seguin, Amy Hoy, Dave Chelimsky, Pat Maddox, Tom Preston-Werner, Chris Wanstrath, Chad Fowler, Josh Susser, Obie Fernandez, Ian McFarland, Steven Bristol, Wolfram Arnold, Alex Chaffee, Giles Bowkett, Evan Dorn, Long Nguyen, James Lindenbaum, Adam Wiggins, Tikhon Bernstam, Ron Evans, Wyatt Greene, Miles Forrest, the good people at Pivotal Labs, the Heroku gang, the thoughtbot guys, and the GitHub crew. Finally, many, many readers—far too many to list—have contributed a huge number of bug reports and suggestions during the writing of this book, and I gratefully acknowledge their help in making it as good as it can be.

About the author

Michael Hartl is the author of the *Ruby on Rails Tutorial*, the leading introduction to web development with Ruby on Rails. His prior experience includes writing and developing *RailsSpace*, an extremely obsolete Rails tutorial book, and developing Insoshi, a once-popular and now-obsolete social networking platform in Ruby on Rails. In 2011, Michael received a Ruby Hero Award for his contributions to the Ruby community. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Copyright and license

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—but, as noted in Section 4.3.4, this stylesheet is currently empty. In this chapter, we'll change this by incorporating the *Bootstrap* 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 also refactor the tests from Chapter 3 using the latest RSpec techniques. We'll end by taking a first important step toward letting users sign up to our site.

5.1 Adding some structure

The *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. 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.[2] In this chapter, we will principally be developing the static pages introduced in Section 3.1, 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 5.7. 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.

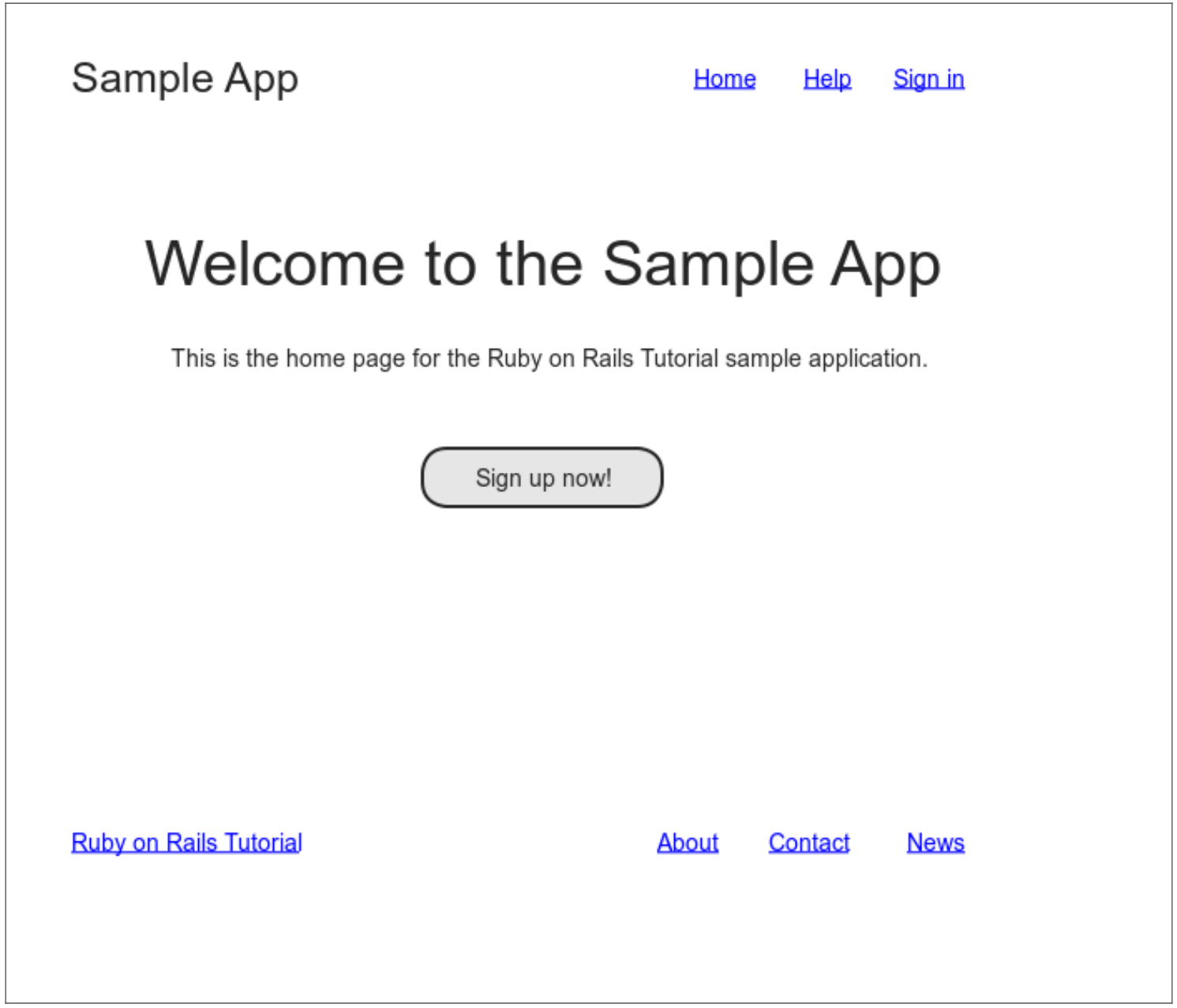

Figure 5.1: A mockup of the sample application's Home page. (full size)

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 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>
    <%= stylesheet_link_tag    "application", media: "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
    <!--[if lt IE 9]>
    <script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
    <![endif]-->
  </head>
  <body>
    <header class="navbar navbar-fixed-top">
      <div class="navbar-inner">
        <div class="container">
```

```
          <%= link_to "sample app", '#', id: "logo" %>
          <nav>
            <ul class="nav pull-right">
              <li><%= link_to "Home",    '#' %></li>
              <li><%= link_to "Help",    '#' %></li>
              <li><%= link_to "Sign in", '#' %></li>
            </ul>
          </nav>
        </div>
      </div>
    </header>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

One thing to note immediately is the switch from Ruby 1.8–style hashes to the new Ruby 1.9 style (Section 4.3.3). That is,

```
<%= stylesheet_link_tag "application", :media => "all" %>
```

has been replaced with

```
<%= stylesheet_link_tag "application", media: "all" %>
```

It's important to note the old hash syntax is deeply entrenched, so it's important to be able to recognize both.

Let's look at the other new elements in Listing 5.1 from top to bottom. As noted briefly in Section 3.1, Rails 3 uses HTML5 by default (as indicated by the doctype `<!DOCTYPE html>`); since the HTML5 standard is relatively new, some browsers (especially older versions Internet

Explorer) don't fully support it, so we include some JavaScript code (known as an "HTML5 shim") to work around the issue:

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.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">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", '#', id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home",    '#' %></li>
          <li><%= link_to "Help",    '#' %></li>
          <li><%= link_to "Sign in", '#' %></li>
        </ul>
      </nav>
```

```
    </div>
  </div>
</header>
```

Here the `header` tag indicates elements that should go at the top of the page. We've given the `header` tag two *CSS classes*,[3] called `navbar` and `navbar-fixed-top`, separated with a space:

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

All HTML elements can be assigned both classes and *ids*; 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, both of the `navbar` and `navbar-fixed-top` 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 couple of `div` tags:

```
<div class="navbar-inner">
  <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, each `div` has a CSS class as well. As with the `header` tag's classes, these classes have special meaning to Bootstrap.

After the divs, we encounter some embedded Ruby:

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

This uses the Rails helper `link_to` to create links (which we created directly with the anchor tag `a` in Section 3.3.2); the first argument to `link_to` is the link text, while the second is the URI. We'll fill in the URIs with *named routes* in Section 5.3.3, but for now we use the stub URI `'#'` commonly used in web design. 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 pull-right">
    <li><%= link_to "Home",    '#' %></li>
    <li><%= link_to "Help",    '#' %></li>
    <li><%= link_to "Sign in", '#' %></li>
  </ul>
</nav>
```

The `nav` tag, though formally unnecessary here, communicates the purpose of the navigation links. The `nav` and `pull-right` classes on the `ul` tag have special meaning to Bootstrap. Once Rails has processed this layout and evaluated the embedded Ruby, the list looks like this:

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

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 3.3.4, 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 hero-unit">
  <h1>Welcome to the Sample App</h1>

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

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

<%= link_to image_tag("rails.png", alt: "Rails"), 'http://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-large btn-primary">Sign up now!</a>
```

In the `div` tag, the `hero-unit` CSS class has a special meaning to Bootstrap, as do the `btn`, `btn-large`, 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` attribute of the image tag using symbols. To make this clearer, let's look at the HTML this tag produces:[4]

```
<img alt="Rails" src="/assets/rails.png" />
```

The `alt` attribute is what will be displayed if there is no image, and it is also what will be displayed by screen readers for the visually impaired. Although people are sometimes sloppy about including the `alt` attribute for images, it is in fact required by the HTML standard. Luckily, Rails includes a default `alt` attribute; if you don't specify the attribute in the call to `image_tag`, Rails just uses the image filename (minus extension). In this case, though, we've set the `alt` text explicitly in order to capitalize "Rails".

Now we're finally ready to see the fruits of our labors (Figure 5.2). Pretty underwhelming, you say? Perhaps so. 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.

By the way, you might be surprised to discover that the `rails.png` image actually exists. Where did it come from? It's included for free with every new Rails application, and you will find it in `app/assets/images/rails.png`. Because we used the `image_tag` helper, Rails finds it automatically using the asset pipeline (Section 5.2).

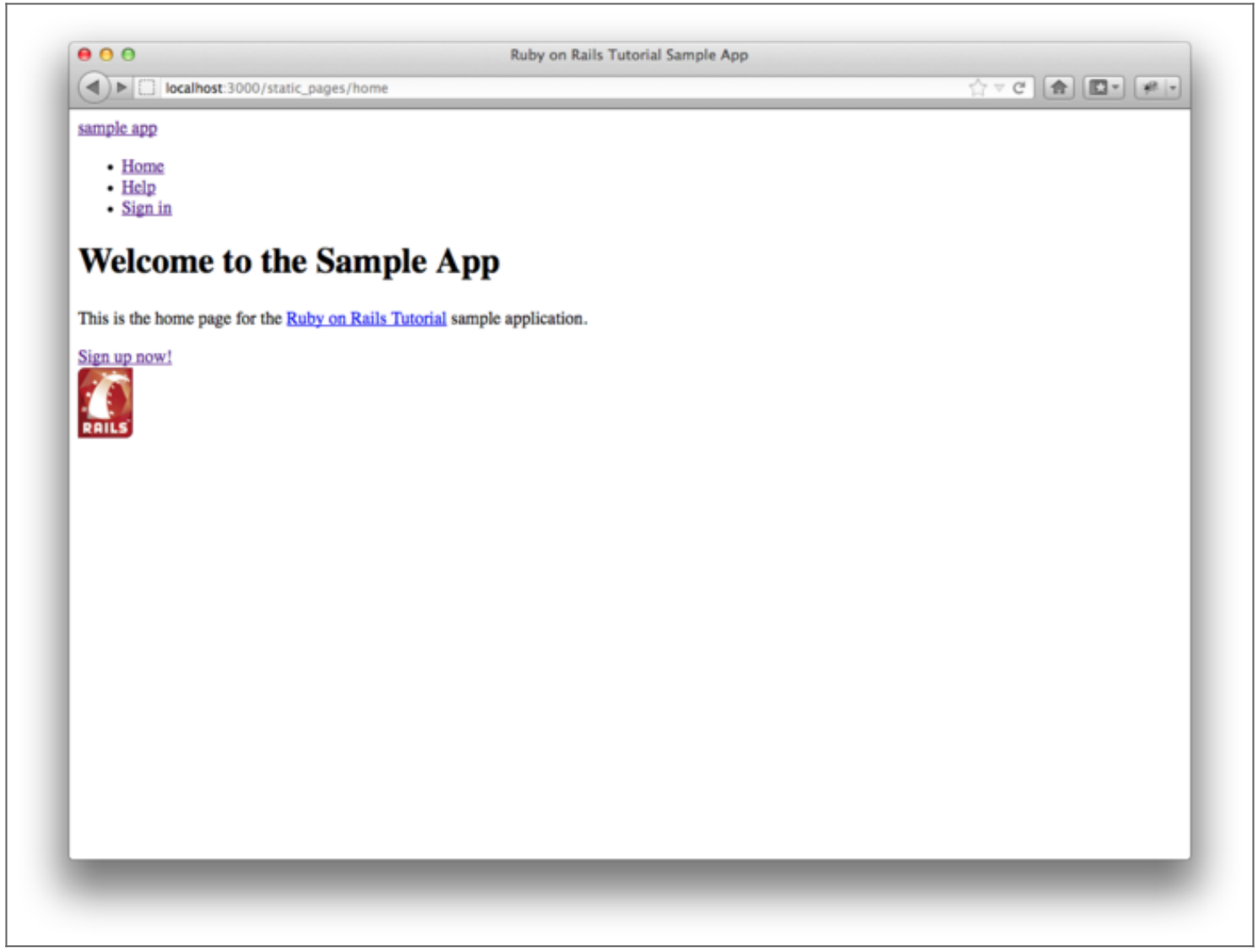

Figure 5.2: The Home page (/static_pages/home) with no custom CSS. (full size)

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 framework from Twitter 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.

Our first step is to add Bootstrap, which in Rails applications can be accomplished with the `bootstrap-sass` gem, as shown in Listing 5.3. 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.[5]

Listing 5.3. Adding the `bootstrap-sass` gem to the **`Gemfile`**.

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

gem 'rails', '3.2.8'
gem 'bootstrap-sass', '2.0.4'
.
.
.
```

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

```
$ bundle install
```

Then restart the web server to incorporate the changes into the development application.

The first step in adding custom CSS to our application is to create a file to contain it:

```
app/assets/stylesheets/custom.css.scss
```

(Use your text editor or IDE to create the new file.) Here both the directory name and filename 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.css.scss` includes the `.css` extension, which indicates a CSS file, and 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.)

After creating the file for custom CSS, we can use the `@import` function to include Bootstrap, as shown in Listing 5.4.

Listing 5.4. Adding Bootstrap CSS.
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";
```

This one line includes the entire Bootstrap CSS framework, with the result shown in in Figure 5.3. (You may have to restart the local web server.) The placement of the text isn't good and the logo doesn't have any style, but the colors and signup button look promising.

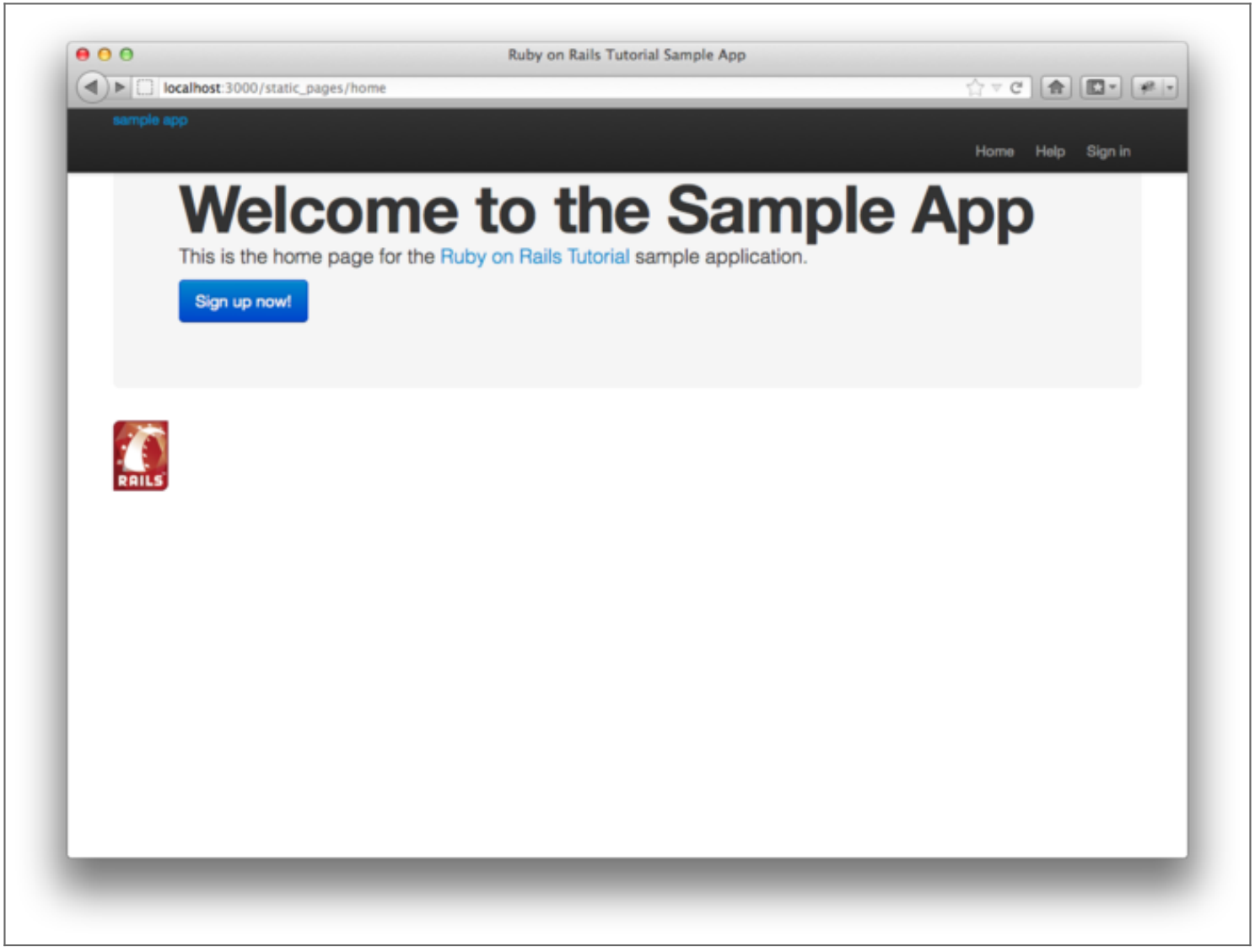

Figure 5.3: The sample application with Bootstrap CSS. (full size)

Next we'll add some CSS that will be used site-wide for styling the layout and each individual page, as shown in Listing 5.5. There are quite a few rules in Listing 5.5; 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. The result of the CSS in Listing 5.5 is shown in Figure 5.4.

Listing 5.5. Adding CSS for some universal styling applying to all pages.
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";

/* universal */

html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}

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

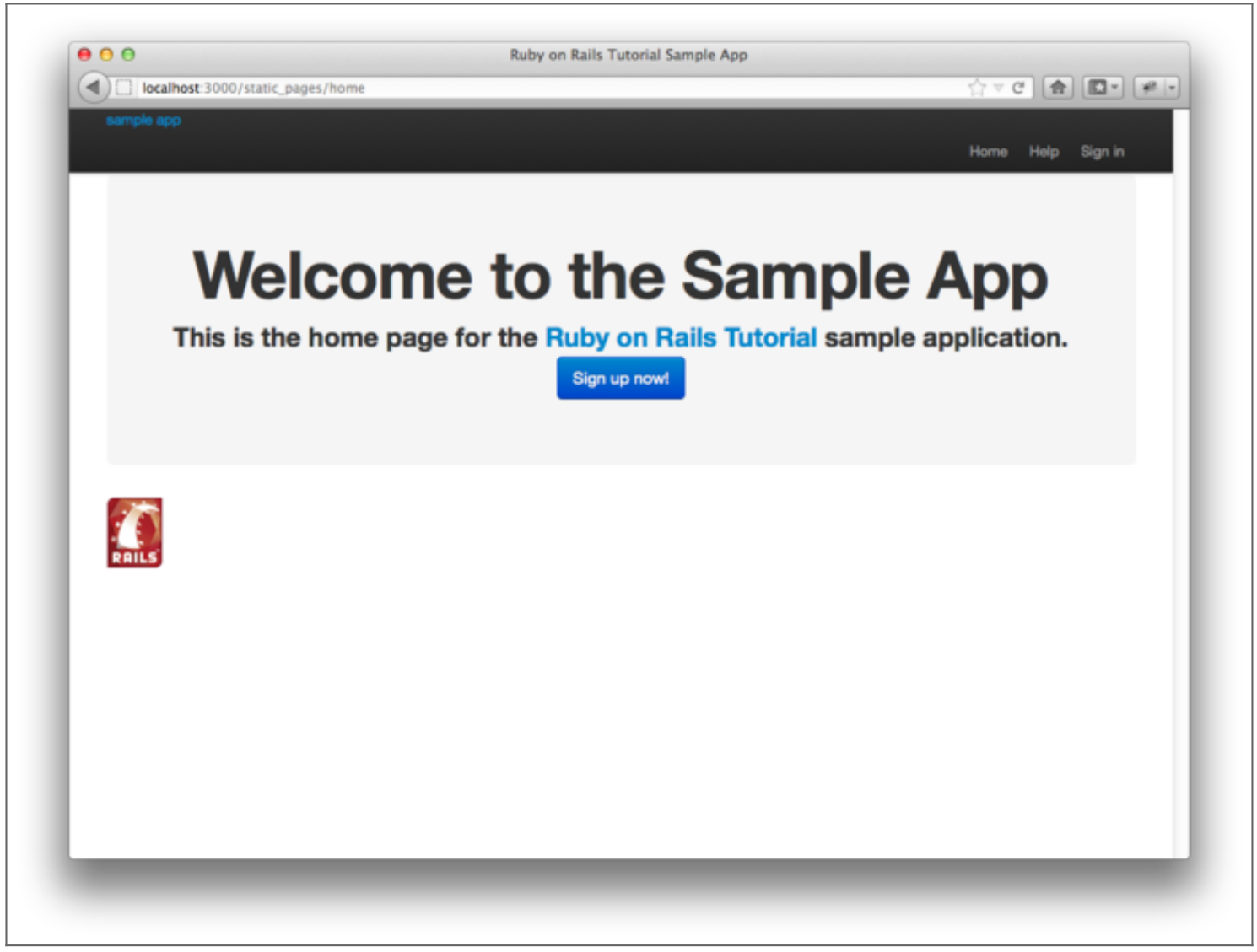

Figure 5.4: Adding some spacing and other universal styling. (full size)

Note that the CSS in Listing 5.5 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. Meanwhile, the CSS in the rule

```
.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 5.7, 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 5.6. (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 5.6 is shown in Figure 5.5.

Listing 5.6. Adding CSS for nice typography.
`app/assets/stylesheets/custom.css.scss`

```
@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: 1.7em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: #999;
}

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

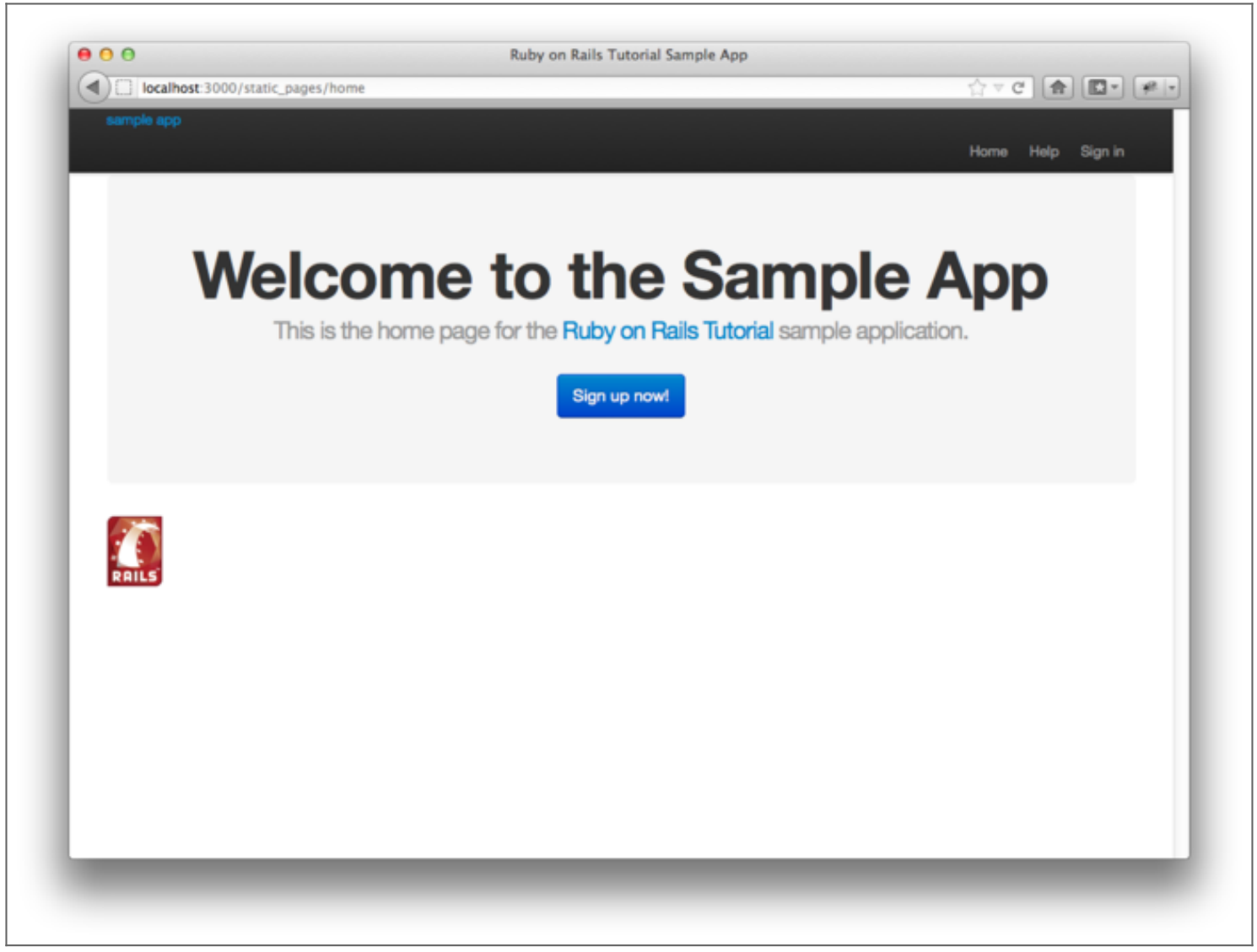

Figure 5.5: Adding some typographic styling. (full size)

Finally, we'll add some rules to style the site's logo, which simply consists of the text "sample app". The CSS in Listing 5.7 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 5.7. Adding CSS for the site logo.

`app/assets/stylesheets/custom.css.scss`

```
@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;
  line-height: 1;
}

#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 5.7 is shown in Figure 5.6.

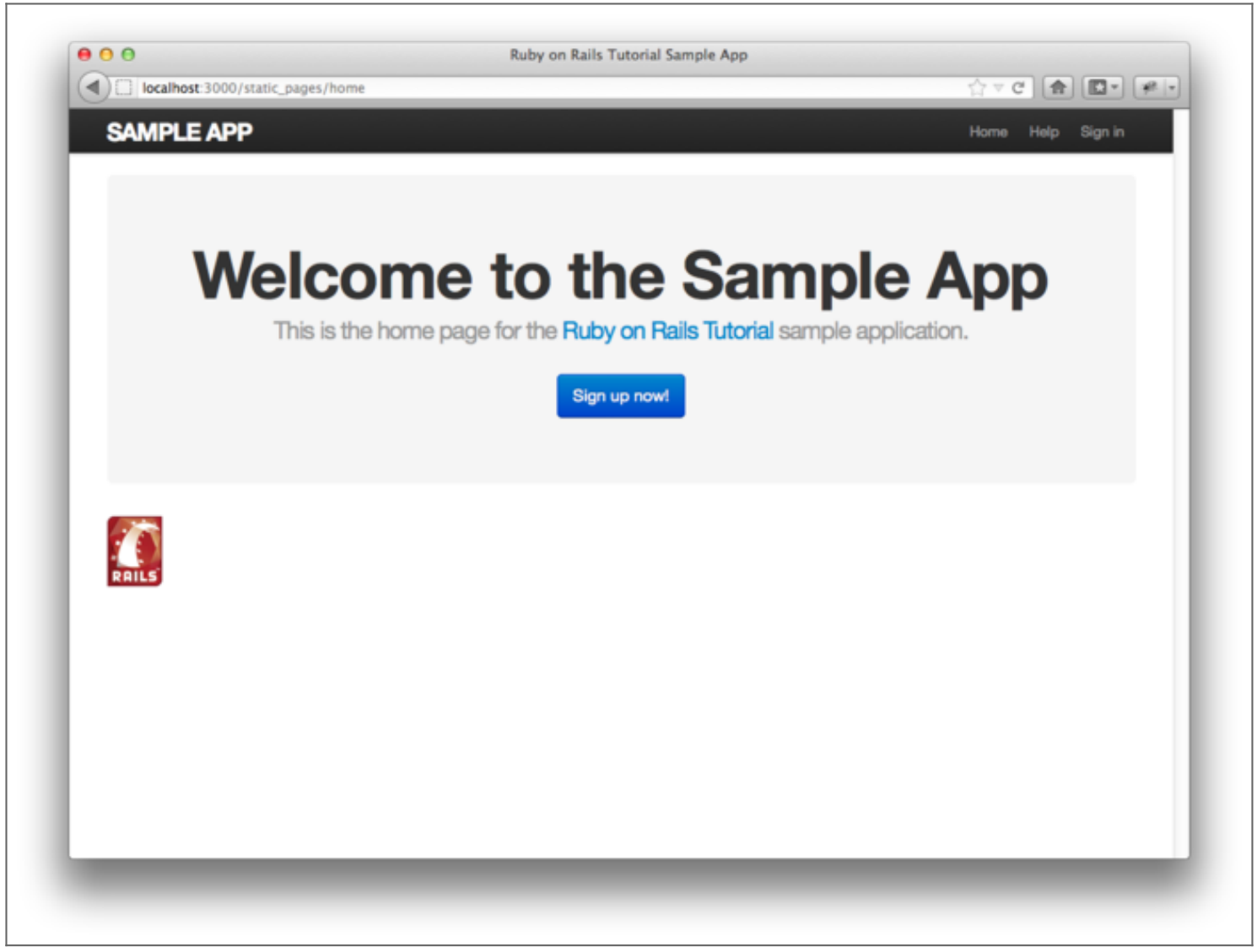

Figure 5.6: The sample app with nicely styled logo. (full size)

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 5.8).

Listing 5.8. 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>
    <%= stylesheet_link_tag    "application", media: "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
    </div>
  </body>
</html>
```

In Listing 5.8, 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.[6] (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.

Of course, to get the partial to work, we have to 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 5.9.

Listing 5.9. A partial for the HTML shim.

app/views/layouts/_shim.html.erb

```
<!--[if lt IE 9]>
<script src="http://html5shim.googlecode.com/svn/trunk/html5.js"></script>
<![endif]-->
```

Similarly, we can move the header material into the partial shown in Listing 5.10 and insert it into the layout with another call to `render`.

Listing 5.10. A partial for the site header.

app/views/layouts/_header.html.erb

```
<header class="navbar navbar-fixed-top">
  <div class="navbar-inner">
    <div class="container">
      <%= link_to "sample app", '#', id: "logo" %>
      <nav>
        <ul class="nav pull-right">
          <li><%= link_to "Home",    '#' %></li>
          <li><%= link_to "Help",    '#' %></li>
          <li><%= link_to "Sign in", '#' %></li>
        </ul>
      </nav>
    </div>
  </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 5.11).[7]

Listing 5.11. A partial for the site footer.

`app/views/layouts/_footer.html.erb`

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

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 URIs 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 5.12).

Listing 5.12. The site layout with a footer partial.

`app/views/layouts/application.html.erb`

```
<!DOCTYPE html>
<html>
  <head>
```

```
    <title><%= full_title(yield(:title)) %></title>
    <%= stylesheet_link_tag    "application", media: "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
    <%= render 'layouts/shim' %>
  </head>
  <body>
    <%= render 'layouts/header' %>
    <div class="container">
      <%= yield %>
      <%= render 'layouts/footer' %>
    </div>
  </body>
</html>
```

Of course, the footer will be ugly without some styling (Listing 5.13). The results appear in Figure 5.7.

Listing 5.13. Adding the CSS for the site footer.

`app/assets/stylesheets/custom.css.scss`

```
.
.
.

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #999;
}

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: 10px;
}
```

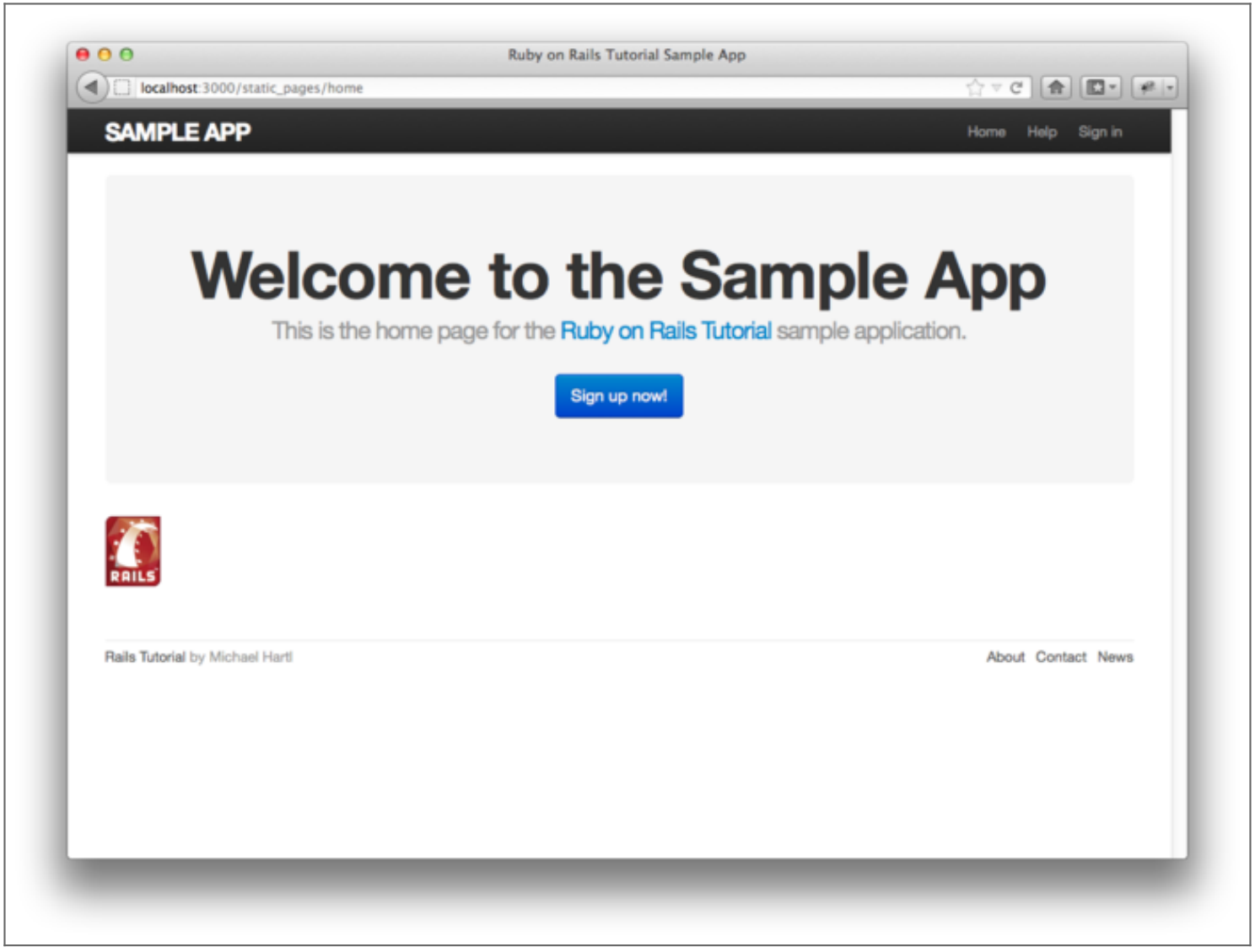

Figure 5.7: The Home page (/static_pages/home) with an added footer. (full size)

5.2 Sass and the asset pipeline

One of the most notable differences between Rails 3.0 and more recent versions is the asset pipeline, which significantly improves the production and management of static assets such as CSS,

JavaScript, and images. This section gives a high-level overview of the asset pipeline and then shows how to use a remarkable tool for making CSS called *Sass*, now included by default as part of the asset pipeline.

5.2.1 The asset pipeline

The asset pipeline involves lots of changes under Rails' hood, but from the perspective of a typical Rails developer there are three principal features to understand: asset directories, manifest files, and preprocessor engines.[8] Let's consider each in turn.

Asset directories

In versions of Rails before 3.0 (including 3.0 itself), static assets lived in the `public/` directory, as follows:

- `public/stylesheets`
- `public/javascripts`
- `public/images`

Files in these directories are (even post-3.0) automatically served up via requests to http://example.com/stylesheets, etc.

Starting in Rails 3.1, there are *three* canonical 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

As you might guess, each of these directories has a subdirectory for each asset class, e.g.,

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

At this point, we're in a position to understand the motivation behind the location of the `custom.css.scss` file in Section 5.1.2: `custom.css.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 5.14).

Listing 5.14. The manifest file for app-specific CSS.
`app/assets/stylesheets/application.css`

```
/*
 * This is a manifest file that'll automatically include all the stylesheets
 * available in this directory and any sub-directories. You're free to add
 * application-wide styles to this file and they'll appear at the top of the
 * compiled file, but it's generally better to create a new file per style
 * scope.
 *= require_self
 *= require_tree .
*/
```

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

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

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
```

ensures that CSS in `application.css` is also 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 three most common cases are `.scss` for Sass, `.coffee` for CoffeeScript, and `.erb` for embedded Ruby

(ERb). We first covered ERb in Section 3.3.3, and cover Sass in Section 5.2.2. We won't be needing CoffeeScript in this tutorial, but it's an elegant little language that compiles to JavaScript. (The RailsCast on CoffeeScript basics is a good place to start.)

The preprocessor engines can be chained, so that

```
foobar.js.coffee
```

gets run through the CoffeeScript processor, and

```
foobar.js.erb.coffee
```

gets run through both CoffeeScript and ERb (with the code running from right to left, i.e., CoffeeScript first).

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 and JavaScript involve splitting functionality into separate files and using nice formatting (with lots of indentation). While convenient for the programmer, this is inefficient in production; including multiple full-sized files can significantly slow page-load times (one of the most important factors affecting the quality of the user experience). With the asset pipeline, in production all the application stylesheets get rolled into one CSS file (`application.css`), all the application JavaScript code gets rolled into one JavaScript file (`javascripts.js`), and all such files (including those in `lib/assets` and `vendor/assets`) are *minified* to remove the unnecessary whitespace that bloats file size. As a result, we get the best of both worlds: multiple nicely formatted files for programmer convenience, with single optimized files 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.[9] 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.css.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 5.5 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 5.7, we have the code

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

#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;
  line-height: 1;
  &: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 5.13, which can be transformed into the following:

```
footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid #eaeaea;
  color: #999;
  a {
    color: #555;
    &:hover {
      color: #222;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
```

```
      margin-left: 10px;
    }
  }
}
```

Converting Listing 5.13 by hand is a good exercise, 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 5.6 and Listing 5.13, we see that there are repeated references to the same color:

```
h2 {
  .
  .
  .
  color: #999;
}
.
.
.
footer {
  .
  .
  .
  color: #999;
}
```

In this case, `#999` is a light gray, and we can give it a name by defining a variable as follows:

```
$lightGray: #999;
```

This allows us to rewrite our SCSS like this:

```
$lightGray: #999;
.
.
.
h2 {
  .
  .
  .
  color: $lightGray;
}
.
.
.
footer {
  .
  .
  .
  color: $lightGray;
}
```

Because variable names such as `$lightGray` are more descriptive than `#999`, it's often useful to define variables even for values that aren't repeated. Indeed, the Bootstrap framework defines a large number of variables for colors, available online on the Bootstrap page of LESS variables. That page defines variables using LESS, not Sass, but the `bootstrap-sass` gem provides the Sass equivalents. It is not difficult to guess the correspondence; where LESS uses an "at" sign `@`, Sass uses a dollar sign `$`. Looking the Bootstrap variable page, we see that there is a variable for light gray:

```
@grayLight: #999;
```

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

```
h2 {
  .
  .
  .
  color: $grayLight;
}
.
.
.
footer {
  .
  .
  .
  color: $grayLight;
}
```

Applying the Sass nesting and variable definition features to the full SCSS file gives the file in Listing 5.15. 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 5.15. The initial SCSS file converted to use nesting and variables.
`app/assets/stylesheets/custom.css.scss`

```
@import "bootstrap";

/* mixins, variables, etc. */

$grayMediumLight: #eaeaea;

/* universal */

html {
```

```
  overflow-y: scroll;
}

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: 1.7em;
  letter-spacing: -1px;
  margin-bottom: 30px;
  text-align: center;
  font-weight: normal;
  color: $grayLight;
}

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;
  line-height: 1;
  &:hover {
    color: white;
    text-decoration: none;
  }
}

/* footer */

footer {
  margin-top: 45px;
  padding-top: 5px;
  border-top: 1px solid $grayMediumLight;
  color: $grayLight;
  a {
    color: $gray;
    &:hover {
      color: $grayDarker;
    }
  }
  small {
    float: left;
  }
  ul {
    float: right;
    list-style: none;
    li {
      float: left;
      margin-left: 10px;
```

```
    }
  }
}
```

Sass gives us even more ways to simplify our stylesheets, but the code in Listing 5.15 uses the most important features and gives us a great start. See the Sass website for more details.

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 `'#'`. Of course, 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 URI 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 URI change everywhere **`about_path`** is used.

The full list of our planned links appears in Table 5.1, along with their mapping to URIs and routes. We'll implement all but the last one by the end of this chapter. (We'll make the last one in Chapter 8.)

Page	URI	Named route
Home	/	`root_path`
About	/about	`about_path`
Help	/help	`help_path`
Contact	/contact	`contact_path`
Sign up	/signup	`signup_path`
Sign in	/signin	`signin_path`

Table 5.1: Route and URI mapping for site links.

Before moving on, let's add a Contact page (left as an exercise in Chapter 3). The test appears as in Listing 5.16, which simply follows the model last seen in Listing 3.18. Note that, as in the application code, in Listing 5.16 we've switched to Ruby 1.9–style hashes.

Listing 5.16. Tests for a Contact page.
spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  describe "Contact page" do

    it "should have the h1 'Contact'" do
      visit '/static_pages/contact'
      page.should have_selector('h1', text: 'Contact')
```

```
    end

    it "should have the title 'Contact'" do
      visit '/static_pages/contact'
      page.should have_selector('title',
                    text: "Ruby on Rails Tutorial Sample App | Contact")
    end
  end
end
```

You should verify that these tests fail:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

The application code parallels the addition of the About page in Section 3.2.2: first we update the routes (Listing 5.17), then we add a `contact` action to the StaticPages controller (Listing 5.18), and finally we create a Contact view (Listing 5.19).

Listing 5.17. Adding a route for the Contact page.
config/routes.rb

```
SampleApp::Application.routes.draw do
  get "static_pages/home"
  get "static_pages/help"
  get "static_pages/about"
  get "static_pages/contact"
  .
  .
  .
end
```

Listing 5.18. Adding an action for the Contact page.
app/controllers/static_pages_controller.rb

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

Listing 5.19. The view for the Contact page.
app/views/static_pages/contact.html.erb

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

Now make sure that the tests pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

5.3.1 Route tests

With the work we've done writing integration test for the static pages, writing tests for the routes is simple: we just replace each occurrence of a hard-coded address with the desired named route from Table 5.1. In other words, we change

```
visit '/static_pages/about'
```

to

```
visit about_path
```

and so on for the other pages. The result appears in Listing 5.20.

Listing 5.20. Tests for the named routes.

spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  describe "Home page" do

    it "should have the h1 'Sample App'" do
      visit root_path
      page.should have_selector('h1', text: 'Sample App')
    end

    it "should have the base title" do
      visit root_path
      page.should have_selector('title',
                        text: "Ruby on Rails Tutorial Sample App")
    end

    it "should not have a custom page title" do
      visit root_path
      page.should_not have_selector('title', text: '| Home')
    end
  end

  describe "Help page" do

    it "should have the h1 'Help'" do
      visit help_path
      page.should have_selector('h1', text: 'Help')
    end
```

```
    it "should have the title 'Help'" do
      visit help_path
      page.should have_selector('title',
                        text: "Ruby on Rails Tutorial Sample App | Help")
    end
  end

  describe "About page" do

    it "should have the h1 'About'" do
      visit about_path
      page.should have_selector('h1', text: 'About Us')
    end

    it "should have the title 'About Us'" do
      visit about_path
      page.should have_selector('title',
                    text: "Ruby on Rails Tutorial Sample App | About Us")
    end
  end

  describe "Contact page" do

    it "should have the h1 'Contact'" do
      visit contact_path
      page.should have_selector('h1', text: 'Contact')
    end

    it "should have the title 'Contact'" do
      visit contact_path
      page.should have_selector('title',
                    text: "Ruby on Rails Tutorial Sample App | Contact")
    end
  end
end
```

As usual, you should check that the tests are now red:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

By the way, if the code in Listing 5.20 strikes you as repetitive and verbose, you're not alone. We'll refactor this mess into a beautiful jewel in Section 5.3.4.

5.3.2 Rails routes

Now that we have tests for the URIs we want, it's time to get them to work. As noted in Section 3.1.2, the file Rails uses for URI mappings is **`config/routes.rb`**. If you take a look at the default routes file, you'll see that it's quite a mess, but it's a useful mess—full of commented-out example route mappings. I suggest reading through it at some point, and I also suggest taking a look at the Rails Guides article "Rails Routing from the outside in" for a much more in-depth treatment of routes.

To define the named routes, we need to replace rules such as

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

with

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

This arranges both for a valid page at **`/help`** and a named route called **`help_path`** that returns the path to that page. (Actually, using **`get`** in place of **`match`** gives the same named routes, but using **`match`** is more conventional.)

Applying this pattern to the other static pages gives Listing 5.21. The only exception is the Home page, which we'll take care of in Listing 5.23.

Listing 5.21. Routes for static pages.

config/routes.rb

```
SampleApp::Application.routes.draw do
  match '/help',    to: 'static_pages#help'
  match '/about',   to: 'static_pages#about'
  match '/contact', to: 'static_pages#contact'
  .
  .
  .
end
```

If you read the code in Listing 5.21 carefully, you can probably figure out what it does; for example, you can see that

```
match '/about', to: 'static_pages#about'
```

matches **'/about'** and routes it to the **about** action in the StaticPages controller. Before, this was more explicit: we used

```
get 'static_pages/about'
```

to get to the same place, but **/about** is more succinct. In addition, as mentioned above, the code **match '/about'** also automatically creates *named routes* for use in the controllers and views:

```
about_path => '/about'
about_url  => 'http://localhost:3000/about'
```

Note that `about_url` is the *full* URI http://localhost:3000/about (with `localhost:3000` being replaced with the domain name, such as `example.com`, for a fully deployed site). As discussed in Section 5.3, to get just /about, you use `about_path`. 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 after redirects the HTTP standard technically requires a full URI, although in most browsers it will work either way.

With these routes now defined, the tests for the Help, About, and Contact pages should pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

This leaves the test for the Home page as the last one to fail.

To establish the route mapping for the Home page, we *could* use code like this:

```
match '/', to: 'static_pages#home'
```

This is unnecessary, though; Rails has special instructions for the root URI / ("slash") located lower down in the file (Listing 5.22).

Listing 5.22. The commented-out hint for defining the root route.
`config/routes.rb`

```
SampleApp::Application.routes.draw do
  .
  .
  .
  # You can have the root of your site routed with "root"
  # just remember to delete public/index.html.
  # root :to => "welcome#index"
```

```
  .
  .
  .
end
```

Using Listing 5.22 as a model, we arrive at Listing 5.23 to route the root URI / to the Home page.

Listing 5.23. Adding a mapping for the root route.

`config/routes.rb`

```
SampleApp::Application.routes.draw do
  root to: 'static_pages#home'

  match '/help',    to: 'static_pages#help'
  match '/about',   to: 'static_pages#about'
  match '/contact', to: 'static_pages#contact'
  .
  .
  .
end
```

This code maps the root URI / to /static_pages/home, and also gives URI helpers as follows:

```
root_path => '/'
root_url  => 'http://localhost:3000/'
```

We should also heed the comment in Listing 5.22 and delete `public/index.html` to prevent Rails from rendering the default page (Figure 1.3) when we visit /. You can of course simply remove the file by trashing it, but if you're using Git for version control there's a way to tell Git about the removal at the same time using `git rm`:

```
$ git rm public/index.html
```

You may recall from Section 1.3.5 that we used the Git command `git commit -a -m "Message"`, with flags for "all changes" (`-a`) and a message (`-m`). As shown above, Git also lets us roll the two flags into one using `git commit -am "Message"`.

With that, all of the routes for static pages are working, and the tests should pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

Now we just have to fill in the links in the layout.

5.3.3 Named routes

Let's put the named routes created in Section 5.3.2 to work in our layout. This will entail 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 5.24), 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 5.24. Header partial with links.

`app/views/layouts/_header.html.erb`

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

We won't have a named route for the "Sign 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 5.25).

Listing 5.25. Footer partial with links.

`app/views/layouts/_footer.html.erb`

```
<footer class="footer">
  <small>
    <a href="http://railstutorial.org/">Rails Tutorial</a>
    by Michael Hartl
  </small>
```

```
    <nav>
      <ul>
        <li><%= link_to "About",   about_path %></li>
        <li><%= link_to "Contact", contact_path %></li>
        <li><a href="http://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 5.8).

By the way, it's worth noting that, although we haven't actually tested for the presence of the links on the layout, our tests will fail if the routes aren't defined. You can check this by commenting out the routes in Listing 5.21 and running your test suite. For a testing method that actually makes sure the links go to the right places, see Section 5.6.

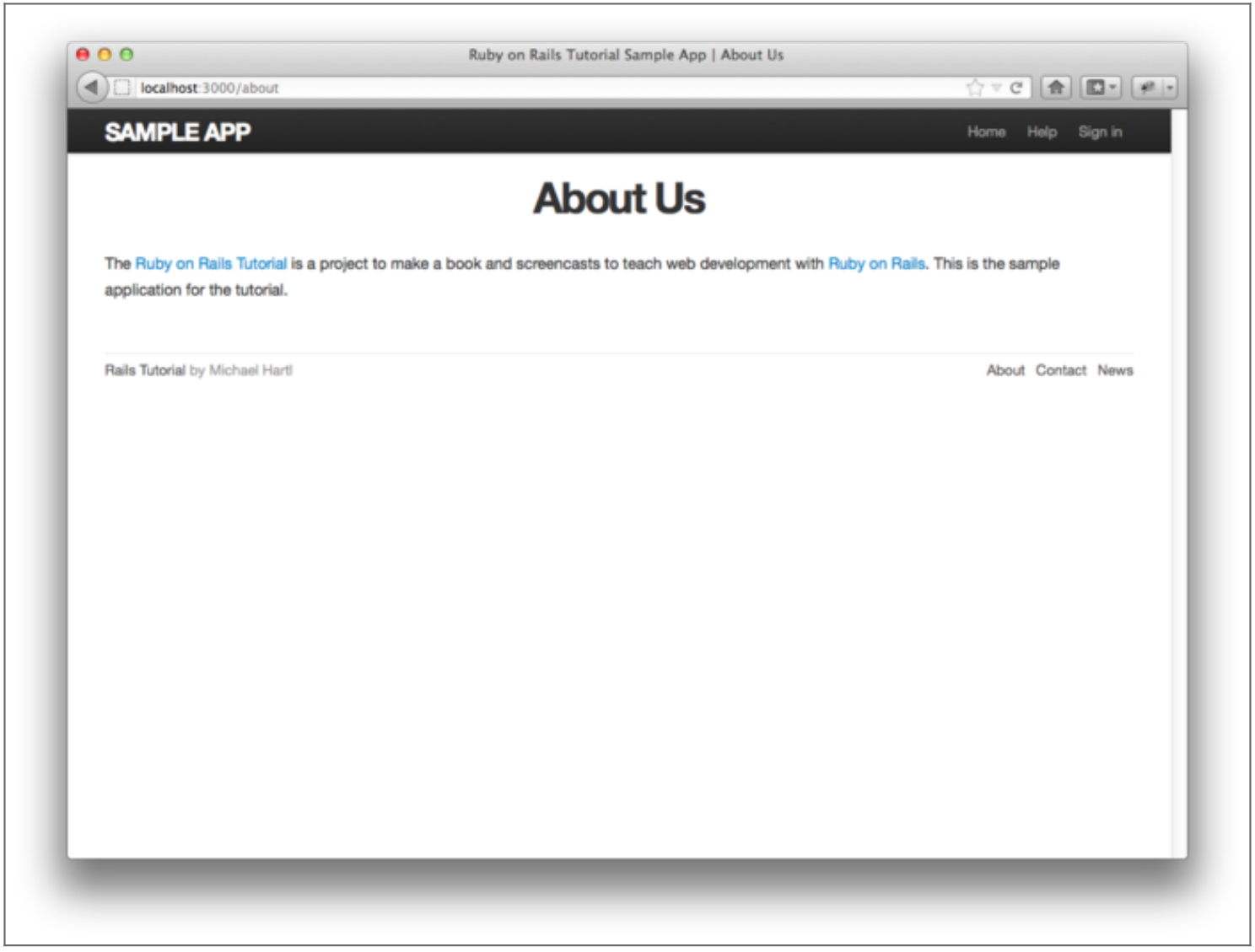

Figure 5.8: The About page at /about. (full size)

5.3.4 Pretty RSpec

We noted in Section 5.3.1 that the tests for the static pages are getting a little verbose and repetitive (Listing 5.20). In this section we'll make use of the latest features of RSpec to make our

tests more compact and elegant.

Let's take a look at a couple of the examples to see how they can be improved:

```
describe "Home page" do

  it "should have the h1 'Sample App'" do
    visit root_path
    page.should have_selector('h1', text: 'Sample App')
  end

  it "should have the base title" do
    visit root_path
    page.should have_selector('title',
                      text: "Ruby on Rails Tutorial Sample App")
  end

  it "should not have a custom page title" do
    visit root_path
    page.should_not have_selector('title', text: '| Home')
  end
end
```

One thing we notice is that all three examples include a visit to the root path. We can eliminate this duplication with a `before` block:

```
describe "Home page" do
  before { visit root_path }

  it "should have the h1 'Sample App'" do
    page.should have_selector('h1', text: 'Sample App')
  end

  it "should have the base title" do
    page.should have_selector('title',
                      text: "Ruby on Rails Tutorial Sample App")
  end
```

```
  it "should not have a custom page title" do
    page.should_not have_selector('title', text: '| Home')
  end
end
```

This uses the line

```
before { visit root_path }
```

to visit the root path before each example. (The **`before`** method can also be invoked with **`before(:each)`**, which is a synonym.)

Another source of duplication appears in each example; we have both

```
it "should have the h1 'Sample App'" do
```

and

```
page.should have_selector('h1', text: 'Sample App')
```

which say essentially the same thing. In addition, both examples reference the **`page`** variable. We can eliminate these sources of duplication by telling RSpec that **`page`** is the *subject* of the tests using

```
subject { page }
```

and then using a variant of the `it` method to collapse the code and description into one line:

```
it { should have_selector('h1', text: 'Sample App') }
```

Because of `subject { page }`, the call to `should` automatically uses the `page` variable supplied by Capybara (Section 3.2.1).

Applying these changes gives much more compact tests for the Home page:

```
  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_selector('h1', text: 'Sample App') }
    it { should have_selector 'title',
                         text: "Ruby on Rails Tutorial Sample App" }
    it { should_not have_selector 'title', text: '| Home' }
  end
```

This code looks nicer, but the title test is still a bit long. Indeed, most of the title tests in Listing 5.20 have long title text of the form

```
"Ruby on Rails Tutorial Sample App | About"
```

An exercise in Section 3.5 proposes eliminating some of this duplication by defining a `base_title` variable and using string interpolation (Listing 3.30). We can do even better by defining a `full_title`, which parallels the `full_title` helper from Listing 4.2. We do this by creating both a `spec/support` directory and a `utilities.rb` file for RSpec utilities (Listing 5.26).

Listing 5.26. A file for RSpec utilities with a `full_title` function.

`spec/support/utilities.rb`

```
def full_title(page_title)
  base_title = "Ruby on Rails Tutorial Sample App"
  if page_title.empty?
    base_title
  else
    "#{base_title} | #{page_title}"
  end
end
```

Of course, this is essentially a duplicate of the helper in Listing 4.2, but having two independent methods allows us to catch any typos in the base title. This is dubious design, though, and a better (slightly more advanced) approach, which tests the original `full_title` helper directly, appears in the exercises (Section 5.6).

Files in the `spec/support` directory are automatically included by RSpec, which means that we can write the Home tests as follows:

```
  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_selector('h1',    text: 'Sample App') }
    it { should have_selector('title', text: full_title('')) }
  end
```

We can now simplify the tests for the Help, About, and Contact pages using the same methods used for the Home page. The results appear in Listing 5.27.

Listing 5.27. Prettier tests for the static pages.

spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do

  subject { page }

  describe "Home page" do
    before { visit root_path }

    it { should have_selector('h1',    text: 'Sample App') }
    it { should have_selector('title', text: full_title('')) }
    it { should_not have_selector 'title', text: '| Home' }
  end

  describe "Help page" do
    before { visit help_path }

    it { should have_selector('h1',    text: 'Help') }
    it { should have_selector('title', text: full_title('Help')) }
  end

  describe "About page" do
    before { visit about_path }

    it { should have_selector('h1',    text: 'About') }
    it { should have_selector('title', text: full_title('About Us')) }
  end

  describe "Contact page" do
    before { visit contact_path }

    it { should have_selector('h1',    text: 'Contact') }
    it { should have_selector('title', text: full_title('Contact')) }
  end
end
```

You should now verify that the tests still pass:

```
$ bundle exec rspec spec/requests/static_pages_spec.rb
```

This RSpec style in Listing 5.27 is much pithier than the style in Listing 5.20—indeed, it can be made even pithier (Section 5.6). We will use this more compact style whenever possible when developing the rest of the sample application.

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

It's been a while since we created our first controller, the StaticPages controller, way back in Section 3.1.2. 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` and pass it as an argument to `generate controller` to create it automatically (Listing 5.28).

Listing 5.28. Generating a Users controller (with a `new` action).

```
$ rails generate controller Users new --no-test-framework
      create  app/controllers/users_controller.rb
       route  get "users/new"
      invoke  erb
      create    app/views/users
      create    app/views/users/new.html.erb
      invoke  helper
```

```
      create    app/helpers/users_helper.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.js.coffee
      invoke    scss
      create      app/assets/stylesheets/users.css.scss
```

This creates a Users controller with a `new` action (Listing 5.29) and a stub user view (Listing 5.30).

Listing 5.29. The initial Users controller, with a `new` action.

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def new
  end

end
```

Listing 5.30. The initial `new` action for Users.

app/views/users/new.html.erb

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

5.4.2 Signup URI

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 URI to be /signup instead. As in Section 5.3, we'll first write some integration tests, which we'll now generate:

```
$ rails generate integration_test user_pages
```

Then, following the model of the static pages spec in Listing 5.27, we'll fill in the user pages test with code to test for the contents of the `h1` and `title` tags, as seen in Listing 5.31.

Listing 5.31. The initial spec for users, with a test for the signup page.

`spec/requests/user_pages_spec.rb`

```
require 'spec_helper'

describe "User pages" do

  subject { page }

  describe "signup page" do
    before { visit signup_path }

    it { should have_selector('h1',    text: 'Sign up') }
    it { should have_selector('title', text: full_title('Sign up')) }
  end
end
```

We can run these tests using the `rspec` command as usual:

```
$ bundle exec rspec spec/requests/user_pages_spec.rb
```

It's worth noting that we can also run all the request specs by passing the whole directory instead of just one file:

```
$ bundle exec rspec spec/requests/
```

Based on this pattern, you may be able to guess how to run *all* the specs:

```
$ bundle exec rspec spec/
```

For completeness, we'll usually use this method to run the tests through the rest of the tutorial. By the way, it's worth noting (since you may see other people use it) that you can also run the test suite using the **spec** Rake task:

```
$ bundle exec rake spec
```

(In fact, you can just type **rake** by itself; the default behavior of **rake** is to run the test suite.)

By construction, the Users controller already has a **new** action, so to get the test to pass all we need is the right route and the right view content. We'll follow the examples from Listing 5.21 and add a **match '/signup'** rule for the signup URI (Listing 5.32).

Listing 5.32. A route for the signup page.
config/routes.rb

```
SampleApp::Application.routes.draw do
  get "users/new"

  root to: 'static_pages#home'

  match '/signup',  to: 'users#new'

  match '/help',    to: 'static_pages#help'
  match '/about',   to: 'static_pages#about'
  match '/contact', to: 'static_pages#contact'
  .
  .
  .
end
```

Note that we have kept the rule `get "users/new"`, which was generated automatically by the Users controller generation in Listing 5.28. Currently, this rule is necessary for the `'users/new'` routing to work, but it doesn't follow the proper REST conventions (Table 2.2), and we will eliminate it in Section 7.1.2.

To get the tests to pass, all we need now is a view with the title and heading "Sign up" (Listing 5.33).

Listing 5.33. The initial (stub) signup page.
app/views/users/new.html.erb

```
<% provide(:title, 'Sign up') %>
<h1>Sign up</h1>
<p>Find me in app/views/users/new.html.erb</p>
```

At this point, the signup test in Listing 5.31 should pass. All that's left is to add the proper link to the button on the Home page. As with the other routes, `match '/signup'` gives us the named route `signup_path`, which we put to use in Listing 5.34.

Listing 5.34. Linking the button to the Signup page.
app/views/static_pages/home.html.erb

```
<div class="center hero-unit">
  <h1>Welcome to the Sample App</h1>

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

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

```
<%= link_to image_tag("rails.png", alt: "Rails"), 'http://rubyonrails.org/' %>
```

With that, we're done with the links and named routes, at least until we add a route for signing in (Chapter 8). The resulting new user page (at the URI /signup) appears in Figure 5.9.

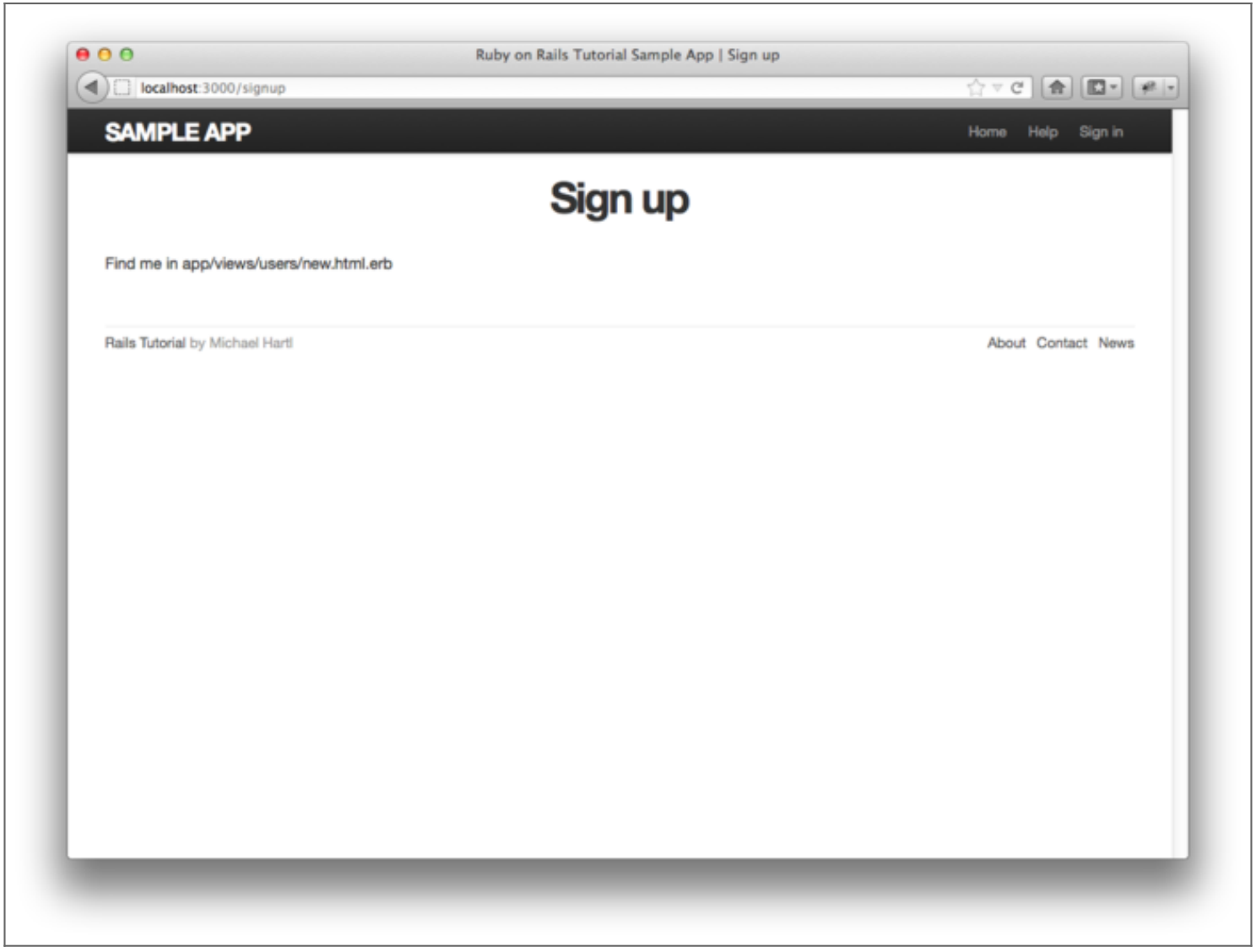

Figure 5.9: The new signup page at /signup. (full size)

At this point the tests should pass:

```
$ bundle exec rspec spec/
```

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, sign in, and sign 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 the changes back into the master branch:

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

You can also push up to GitHub:

```
$ git push
```

Finally, you can deploy to Heroku:

```
$ git push heroku
```

The result should be a working sample application on the production server:

```
$ heroku open
```

If you run into trouble, try running

```
$ heroku logs
```

to debug the error using the Heroku logfile.

5.6 Exercises

1. The code in Listing 5.27 for testing static pages is compact but is still a bit repetitive. RSpec supports a facility called *shared examples* to eliminate the kind of duplication. By following the example in Listing 5.35, fill in the missing tests for the Help, About, and Contact pages. Note that the `let` command, introduced briefly in Listing 3.30, creates a local variable with the given value on demand (i.e., when the variable is used), in contrast to an instance variable, which is created upon assignment.

2. You may have noticed that our tests for the layout links test the routing but don't actually check that the links on the layout go to the right pages. One way to implement these tests is to use `visit` and `click_link` inside the RSpec integration test. Fill in the code in Listing 5.36 to verify that all the layout links are properly defined.

3. Eliminate the need for the `full_title` test helper in Listing 5.26 by writing tests for the original helper method, as shown in Listing 5.37. (You will have to create both the

`spec/helpers` directory and the `application_helper_spec.rb` file.) Then `include` it into the test using the code in Listing 5.38. Verify by running the test suite that the new code is still valid. *Note*: Listing 5.37 uses *regular expressions*, which we'll learn more about in Section 6.2.4. (Thanks to Alex Chaffee for the suggestion and code used in this exercise.)

Listing 5.35. Using an RSpec shared example to eliminate test duplication.
`spec/requests/static_pages_spec.rb`

```
require 'spec_helper'

describe "Static pages" do

  subject { page }

  shared_examples_for "all static pages" do
    it { should have_selector('h1',    text: heading) }
    it { should have_selector('title', text: full_title(page_title)) }
  end

  describe "Home page" do
    before { visit root_path }
    let(:heading)    { 'Sample App' }
    let(:page_title) { '' }

    it_should_behave_like "all static pages"
    it { should_not have_selector 'title', text: '| Home' }
  end

  describe "Help page" do
    .
    .
    .
  end

  describe "About page" do
    .
    .
    .
  end
```

```
  describe "Contact page" do
    .
    .
    .
  end
end
```

Listing 5.36. A test for the links on the layout.

spec/requests/static_pages_spec.rb

```
require 'spec_helper'

describe "Static pages" do
  .
  .
  .
  it "should have the right links on the layout" do
    visit root_path
    click_link "About"
    page.should have_selector 'title', text: full_title('About Us')
    click_link "Help"
    page.should # fill in
    click_link "Contact"
    page.should # fill in
    click_link "Home"
    click_link "Sign up now!"
    page.should # fill in
    click_link "sample app"
    page.should # fill in
  end
end
```

Listing 5.37. Tests for the `full_title` helper.

spec/helpers/application_helper_spec.rb

```
require 'spec_helper'
```

```
describe ApplicationHelper do

  describe "full_title" do
    it "should include the page title" do
      full_title("foo").should =~ /foo/
    end

    it "should include the base title" do
      full_title("foo").should =~ /^Ruby on Rails Tutorial Sample App/
    end

    it "should not include a bar for the home page" do
      full_title("").should_not =~ /\|/
    end
  end
end
```

Listing 5.38. Replacing the `full_title` test helper with a simple `include`.

spec/support/utilities.rb

```
include ApplicationHelper
```

« Chapter 4 Rails-flavored Ruby

Chapter 6 Modeling users »

---

1. Thanks to reader Colm Tuite for his excellent work in helping to convert the sample application over to Bootstrap. ↑
2. The mockups in the *Ruby on Rails Tutorial* are made with an excellent online mockup application called Mockingbird. ↑
3. These are completely unrelated to Ruby classes. ↑
4. 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. ↑

5. It is also possible to use LESS with the asset pipeline; see the `less-rails-bootstrap` gem for details. ↑

6. 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. ↑

7. 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. ↑

8. The structure of this section is based on the excellent blog post The Rails 3 Asset Pipeline in (about) 5 Minutes by Michael Erasmus. For more details, see the Rails Guide on the Asset Pipeline. ↑

9. The older `.sass` format, also supported by Sass, defines 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. ↑