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.