Microposter Application: Style and Navigation
CSC340

These Microposter worksheets were developed from material in http://www.railstutorial.org/book by Michael Hartl.

We're going to switch applications at this point, exploring the application for making Microposts that we first considered when we were practicing working with scaffolds. We'll call this application "Microposter". So the first step is to copy that application (probably called scaffoldPractice) into a new folder called microposter:

```
cp –r scaffoldPractice microposter
```

Then `cd` into the microposter directory and work in there for this worksheet.

Based on what we've already done, the Microposter application currently allows you to create, read, update, and delete users and microposts. We'll now add on more functionality.

In this worksheet, we'll see examples of how to include some common functionality that we all expect in professional web pages:

I. Dynamic generation of page titles (displayed in the title bar of the web browser).
II. Integration of CSS, including a great CSS framework called Bootstrap.
III. Working navigation links on every page.

I. Dynamic Generation of Page Titles

1. Let's add some simple, static pages to play around with. Make a StaticPages controller with home, help, about, and contact actions.

2. Add some simple home, help, about, and contact messages to the appropriate views.

3. Open app/views/layouts/application.html.erb. Recall that this file defines the general layout of the application. It loads some stylesheets and javascript files as part of what is called the "asset pipeline." This is the mechanism through which Rails loads various resources, making them accessible to the rest of your application.

4. Note the content of the <title> element. This is the text that appears as the title of every page (in the title bar of the browser). Right now it's just plain text – the same text for every page.

5. Let's change the title element to
```
<title>Microposter | <%= yield(:title) %></title>
```

a. This calls the Rails method `yield`, with the result becoming part of the title element content.
b. The yield method takes one argument: a symbol representing the thing you want to get. There's nothing special about the symbol name `:title`. You can think of it as like a variable name. Of course right now, we haven't provided a value for `:title`, so this change does nothing yet.

6. In the home view, put the following at the top of your file: `<% provide(:title, 'Home') %>`
   a. As you may have guessed, the provide method call here associates the string "Home" with the `:title` symbol. So a call to `yield(:title)` will return "Home".
   b. This means that when the home page is visited, application.html.erb will call yield and get the string "Home" for inclusion in the title element.

7. Make similar changes to the help, about, and contact views.

II. Integration of CSS, Including Bootstrap

1. Look again at application.html.erb. Recall that the call to yield without any arguments, in the body element, will load the appropriate view when a page is visited. Note, then, that we can add whatever HTML we want around that yield call. This means that application.html.erb is the place to add headers and footers.

2. Replace the body element of application.html.erb with the following:

```
<body>
  <header class="navbar navbar-fixed-top navbar-inverse">
    <div class="navbar-inner">
      <div class="container">
        <%= link_to "Microposter", '#', id: "logo" %>
        <nav>
          <ul class="nav navbar-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 %>
    <footer class="footer">
      <nav>
        <ul>
          <li><%= link_to "About",   '#' %></li>
          <li><%= link_to "Contact", '#' %></li>
       </ul>
```

```
        </nav>
      </footer>
    </div>
  </body>
```

a. Note that `<%= yield %>` is still in the body element content, so the correct view will still be loaded. We've just added some standard code around it.
b. The header element contains the code we'll have at the top of every page of this application. The use of "class" and "id" in the tags applies some particular CSS to make things look nice. It doesn't affect any functionality – it's all just for looks and layout.
c. The div tags just divide up the HTML in convenient ways. Beyond that, they serve no purpose. The nav tag is similar – it's a signal that it contains navigation links, but this is really just a matter of HTML style. It isn't required.
d. Note that the header contains an unordered list, with links to a few common pages. The footer contains another list with links.
e. For now, the links do not have proper addresses supplied. The "#" is just a dummy address that won't work. We'll fix this soon.

3. Visit http://.../static_pages/home. You'll notice that the navigation links do appear, but things don't look nice at all. This is because the CSS we're referencing in application.html.erb isn't loaded in our application yet. So let's fix that now.

4. The classes and ids we use are defined in Bootstrap. Include the bootstrap-sass gem in your Gemfile and run `bundle install`.

5. Open up config/application.rb. We need to tell our application that we want Bootstrap image files to be included in the asset pipeline. In the application.rb file, inside the Application class itself, add the following line: `config.assets.precompile += %w(*.png *.jpg *.jpeg *.gif)`

6. Next we need to make sure the Bootstrap CSS declarations get loaded with our application. Create an app/assets/stylesheets/custom.css.scss file and put the following inside:
   `@import "bootstrap";`
   This is just a CSS command to import ("paste in", basically) CSS declarations from elsewhere. Since custom.css.scss will automatically be included in the asset pipeline, putting this import in here means that Bootstrap CSS declarations will now be part of the asset pipeline.

7. Check out the Bootstrap section of the Rails Quick Reference file. Note that it's a quick summary of what you need to do to be able to use Bootstrap.

8. While we're at it, let's throw in some more CSS of our own in the custom.css.scss file to really spruce things up. Paste in the following after the `@import "bootstrap"` line:

```
/* universal */
```

```
html {
  overflow-y: scroll;
}

body {
  padding-top: 60px;
}

section {
  overflow: auto;
}

textarea {
  resize: vertical;
}

.center {
  text-align: center;
}

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

div.hero-unit a {
  color:#ffffff;
}

/* 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: #999;
}

p {
  font-size: 1.1em;
```

```
  line-height: 1.7em;
}

/* 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;
}

/* footer */

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

footer a {
  color: #555;
}

footer a:hover {
  color: #ffffff;
}

footer small {
  float: left;
}

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

footer ul li {
  float: left;
  margin-left: 10px;
```

```
}
```

   a. Don't worry about what all the CSS means for now. We'll study CSS more soon. The point is, you now know how to integrate it into your Rails applications.

9. Let's change the home view to make use of some Bootstrap CSS. Replace everything currently in the home view with the following:

```
<% provide(:title, 'Home') %>

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

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

III. Finishing the Navigation Links

1. Restart the server, and visit static_pages/home. The links don't work yet, but the home page should look nice. You should also be able to manually type static_pages/about, static_pages/help, and static_pages/contact in the browser's address bar to visit those pages, and note the presence of the header and footer there as well.

2. Let's get those links working. We'll start by noticing a little design issue here. Internally, we put our static pages inside a controller called "StaticPages". That's a good idea, but it also means that to visit the help page, for example, we need to go to `static_pages/help`, rather than just `help`. So we're allowing our internal organization to affect our external user experience. That's a design problem. Let's fix this.

3. Open routes.rb. Delete or comment out the `get 'static_pages/about'` line. Replace it with:
   `match '/about', to: 'static_pages#about', via: 'get'`

   a. This means that when /about is entered as the URL, static_pages/about should be visited via a GET request.
   b. This also means we have access to about_path and about_url as named routes.
   c. Do the same idea for help and contact, but not for home.
   d. See the entry on the match command in the Quick Reference file, in the routes.rb section.

4. For home, we'll do something a little different. Replace the
   `get 'static_pages/home'`
   line with:
   `root 'static_pages#home'`

    a. This makes it so that the root page for your application is static_pages/home. Finally, no more default Rails page!
    b. We now have access to root_path and root_url.
    c. See the entry on the root command in the Quick Reference file, in the routes.rb section.

5. With these nice new named routes, replace the dummy "#" addresses in application.html.erb. We still don't have an address for signing in – that's coming soon! But you should be able to fill in all the other addresses in application.html.erb.

6. Finally, replace the dummy "#" address in home.html.erb with `new_user_path`
    a. You might recall that we got access to this named route when we create the User resource via scaffolding.

7. Test your application to make sure everything is working correctly.