Matt Lambert, Bass Jobsen,
David Cochran, Ian Whitley

Complete Bootstrap: Responsive Web Development with Bootstrap 4

Learn all the new features and build a set of example applications for your portfolio with the latest version of Bootstrap





Packt

Introduction

Bootstrap, the most popular frontend framework built to design elegant, powerful, and responsive interfaces for professional-level web pages has undergone a major overhaul. Since its debut in August 2011, Twitter Bootstrap, now simply Bootstrap, has become by far the most popular framework for empowering and enhancing frontend web design. Millions of amazing sites across the web are being built with Bootstrap.

Bootstrap provides a palette of user-friendly, cross-browser, tested solutions for most standard UI conventions. Its ready-made, community-tested, combination of HTML markup, CSS styles, and JavaScript plugins greatly accelerates the task of developing a frontend web interface, and it yields a pleasing result out of the gate. With the fundamental elements quickly in place, we can customize the design on top of a solid foundation.

With version 4, Bootstrap reaches an exciting new milestone, a lean code base optimized for modern browsers. Bootstrap enables you to build applications and websites that look good and work well on all types of devices and screen sizes from a single code base. Bootstrap 4 introduces a wide range of new features that make frontend web design even more simple and exciting.

What's in it for me – Course Roadmap?

Maps are vital for your journey, especially when you're holidaying in another continent. When it comes to learning, a roadmap helps you in giving a definitive path for progressing towards the goal. So, here you're presented with a roadmap before you begin your journey.

This course is meticulously designed and developed in order to empower you with all the right and relevant information on Bootstrap. We've created this Learning Path for you that consists of three modules. Each of these modules is a mini-course in their own way, and as you complete each one, you'll have gained key skills and be ready for the material in the next module.

Let's take a look at your learning journey:

- **Bootstrap 4 Fundamentals**: In this module, you will be learning Bootstrap 4 from scratch that is from the installation to the browsers that support files and the source files of Bootstrap too. The advantages of Bootstrap 4 are also covered here. Incorporation of HTML5, CSS, and JavaScript are the main highlights of this module.
- **Bootstrap 4 Core Concepts**: In this module, you will dive into Bootstrap 4 features and the advanced concepts such as flexbox, layouts and JavaScript plugins. Working with interactive Bootstrap 4 components such as flexbox, layouts and content will give you a complete experience to explore Bootstrap in a wider way. JavaScript plugins to update your project is the cherry on the cake in this module. Adding Sass during development stage of the project will save your valuable time while creating the project.
- **Hands-on Projects with Bootstrap 4**: In the final module, you will explore Bootstrap 4 with hands-on projects that will give immense exposure to coding in Bootstrap wherein you will have complete programming knowledge and develop web apps and web pages like a pro. The projects which are covered in this comprehensive module includes Bootstrapping your portfolio, Bootstrapping business, Bootstrapping ecommerce, and Bootstrapping single-page marketing website, which will give you a complete package of knowledge to build a professional website.

How is this course different from other courses?

Packt courses are very carefully designed to make sure that they're delivering the best learning experience possible. This course is a blend of sections that form a sequential flow of concepts covering a focused learning journey presented in a modular manner. This helps you learn a range of topics at your own speed and also move towards your goal of learning the technology. We have prepared this course using extensive research and curation skills. Each section adds to the skilled learned and helps you to achieve mastery of Bootstrap. We hope you enjoy this and any other courses you might purchase from Packt.

What will I get from this course?

- Fire up Bootstrap and set up the required build tools to get started
- Understand how and when to use Flexbox with the Bootstrap layouts
- Learn responsive web design and discover how to build mobile-ready websites with ease

- Find out how to extend the capabilities of Bootstrap with a huge range of tools and plugins, including jQuery
- Play around with the huge variety of components that Bootstrap offers
- Customize your designs by working directly with Bootstrap's SASS files
- Explore the inner workings of Bootstrap 4 by building different websites

Prerequisites

This course is for web developers and designers who want to build enterprise-level and professional websites efficiently with Bootstrap 4. Some of the prerequisites that is required before you begin this course are:

- Knowledge on HTML, CSS, and JavaScript is a must
- Prior basic knowledge on Bootstrap would be beneficial; however not mandatory
- Familiarity with Sass

Credits

This course is a blend of text and quizzes, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Learning Bootstrap 4, Second Edition* by *Matt Lambert*
- *Bootstrap 4 Site Blueprints* by *Bass Jobsen*, *David Cochran*, and *Ian Whitley*

Meet your experts

We have the best works of the following esteemed author to ensure that your learning journey is smooth:

Matt Lambert is a designer and developer with more than 16 years of experience. He currently works full-time as a senior product designer for CA Technologies in Vancouver, BC, Canada.

Bass Jobsen has been programming the web since 1995, ranging from C to PHP. He has a special interest in the processes between designer and programmer. He works on the accessibility of Bootstrap and his JBST WordPress starters theme. With over 5 years of experience with Bootstrap, Bass has been actively contributing to the community with his blogs (`http://bassjobsen.weblogs.fm/`) and Git repos (`https://github.com/bassjobsen`).

David Cochran serves as an associate professor of communication at Oklahoma Wesleyan University. He has been teaching interactive design since 2005. When Twitter Bootstrap was first released in August 2011, he recognized it as a tool that would speed up development while supporting best practices. Thus, he began folding Bootstrap into his university courses, much to the benefit of his students. In 2012, David produced a Bootstrap 2.0 tutorial series for `https://webdesign.tutsplus.com/`.

Ian Whitley developed a passion for writing and literature at a young age. In 2010, he developed a deep interest in web development and decided to get involved in it. He was one of the early adopters of Twitter Bootstrap when it was first released in 2011. With the help of David Cochran, he quickly conquered the system and has used it for many different web projects. Currently, he uses Bootstrap in relation to WordPress, using both in conjunction to create custom and creative solutions for his client.

Table of Contents

Lesson 1: Setting up Our First Blog Project 1

Implementing framework files 1

Inserting the JavaScript files 2

The starter template 2

HTML5 DOCTYPE 3

Structuring the responsive meta tag 4

Normalizing and rebooting 4

Taking the starter template further 4

Using a static site generator 4

Converting the base template to a generator 5

Setting up the blog project 6

css 6

fonts 6

img 6

js 6

partial 7

EJS files 7

Setting up the JSON files 7

Creating the data JSON file 8

Setting up the layout 8

Setting up the header 10

Setting up the footer 11

Creating our first page template 12

Compiling your project 13

Running your project 13

Viewing your project 14

A note about Sass 15

Browser support 15

Vendor prefixes 16

Troubleshooting 17

Summary 17

Assessments 18

Lesson 2: Jumping into Flexbox 21

Flexbox basics and terminology 21

Ordering your Flexbox 22

Stretching your child sections to fit the parent container 24

Changing the direction of the boxes 24

Creating equal-height columns 28

Setting up the Bootstrap Flexbox layout grid 33

Updating the Sass variable 34

Setting up a Flexbox project 35

Adding a custom theme 35

Creating a basic three-column grid 36

Creating full-width layouts 38

Designing a single blog post 39

Summary 42

Assessments 43

Lesson 3: Working with Layouts 45

Working with containers 45

Creating a layout without a container 47

Using multiple containers on a single page 47

Inserting rows into your layout 48

Adding columns to your layout 49

Extra small 49

Small 49

Medium 49

Large 49

Extra large 50

Choosing a column class 50

Creating a simple three-column layout 50

Mixing column classes for different devices 52

What if I want to offset a column? 53

Coding the blog home page 54

Writing the index.ejs template 54

Using spacing CSS classes 55

Testing out the blog home page layout 57

Adding some content 57

What about mobile devices? 59

Using responsive utility classes **60**
Coding the additional blog project page grids 61
Updating _data.json for our new pages 61
Creating the new page templates 62
Coding the contact page template 62
Adding the contact page body 63
Coding the blog post template 65
Adding the blog post feature 66
Adding the blog post body 66
Converting the mailing list section to a partial 68
Summary **69**
Assessments **70**

Lesson 4: Working with Content **71**

Reboot defaults and basics **71**
Headings and paragraphs 72
Lists 72
Preformatted text 72
Tables 72
Forms 72
Learning to use typography **73**
Using display headings 73
Customizing headings **74**
Using the lead class 75
Working with lists 75
Coding an unstyled list 75
Creating inline lists 77
Using description lists 77
How to style images 78
Making images responsive **78**
Using image shapes 79
Aligning images with CSS 80
Coding tables **81**
Setting up the basic table 81
Inversing a table 82
Inversing the table header 83
Adding striped rows 84
Adding borders to a table 84
Adding a hover state to rows 85
Color-coating table rows 86

Making tables responsive 86
Summary 87
Assessments 87

Lesson 5: Playing with Components 89

Using the button component 89
Basic button examples 90
Creating outlined buttons 91
Checkbox and radio buttons 91
Creating a radio button group 92
Using button groups 94
Creating vertical button groups 94
Coding a button dropdown 95
Creating a pop-up menu 96
Creating different size drop-down buttons 97
Coding forms in Bootstrap 4 98
Setting up a form 98
Adding a select dropdown 99
Inserting a Textarea tag into your form 100
Adding a file input form field 101
Inserting radio buttons and checkboxes to a form 102
Adding a form to the blog contact page 104
Updating your project 104
Additional form fields 105
Creating an inline form 105
Hiding the labels in an inline form 106
Adding inline checkboxes and radio buttons 107
Changing the size of inputs 108
Controlling the width of form fields 108
Adding validation to inputs 109
Using the Jumbotron component 111
Adding the Label component 113
Using the Alerts component 114
Adding a dismiss button to alerts 116
Using Cards for layout 116
Moving the Card title 118
Changing text alignment in cards 119
Adding a header to a Card 120

Inverting the color scheme of a Card 122
Adding a location card to the Contact page 125
Updating the Blog index page 126
Adding the sidebar 129
Setting up the Blog post page 132
How to use the Navs component 134
Creating tabs with the Nav component 135
Creating a pill navigation 136
Using the Bootstrap Navbar component 137
Changing the color of the Navbar 139
Making the Navbar responsive 140
Adding Breadcrumbs to a page 142
Adding Breadcrumbs to the Blog post page 142
Using the Pagination component 143
Adding the Pager to the Blog post template 144
How to use the List Group component 144
Summary 145
Assessments 146

Lesson 6: Extending Bootstrap with JavaScript Plugins 150

Coding a Modal dialog 150
Coding the Modal dialog 151
Coding Tooltips 153
Updating the project layout 153
How to use Tooltips 154
How to position Tooltips 155
Adding Tooltips to buttons 156
Updating the layout for buttons 156
Avoiding collisions with our components 157
Using Popover components 158
Updating JavaScript 158
Positioning Popover components 159
Adding a Popover to a button 160
Adding our Popover button in JavaScript 160
Using the Collapse component 161
Coding an Accordion with the Collapse component 163
Coding a Bootstrap Carousel 165
Adding the Carousel bullet navigation 166

Including Carousel slides 167
Adding Carousel arrow navigation 167
Summary 169
Assessments 170

Lesson 7: Throwing in Some Sass 171

Learning the basics of Sass 171
Using Sass in the blog project 171
Updating the blog project 172
Using variables 173
Using the variables in CSS 174
Using other variables as variable values 175
Importing partials in Sass 176
Using mixins 177
How to use operators 179
Creating a collection of variables 180
Importing the variables to your custom style sheet 180
Adding a color palette 180
Adding some background colors 181
Setting up variables for typography 182
Coding the text color variables 182
Coding variables for links 183
Setting up border variables 184
Adding variables for margin and padding 184
Adding mixins to the variables file 185
Coding a border-radius mixin 185
Customizing components 187
Customizing the button component 187
Extending the button component to use our color palette 189
Writing a theme 192
Common components that need to be customized 192
Theming the drop-down component 193
Customizing the alerts component 195
Customizing the typography component 196
Summary 198
Assessments 198

Lesson 8: Bootstrapping Your Portfolio 200

What we'll build 200
Surveying the exercise files 203
Marking up the carousel 207
How does the carousel work? 210
Changing the carousel by adding new animations 211
JavaScript events of the Carousel plugin 212
Creating responsive columns 213
Turning links into buttons 215
Understanding the power of Sass 216
Customizing Bootstrap's Sass according to our needs 217
Customizing variables 218
Customizing the navbar 219
Adding the logo image 222
Adding icons 224
Styling the carousel 226
Adding top and bottom padding 226
Repositioning the carousel indicators 227
Styling the indicators 228
Tweaking the columns and their content 231
Styling the footer 234
Recommended next steps 238
Summary 239
Assessments 239

Lesson 9: Bootstrapping Business 241

Sizing up our beginning files 243
Setting up the basics of your design 244
Adding drop-down menus to our navbar 245
Setting the bottom border for the page header 247
Adding images with holder.js 247
Creating a complex banner area 248
Placing a logo above the navbar 249
Reviewing and checking navbar drop-down items 251
Adding utility navigation 254
Making responsive adjustments 257

Implementing the color scheme **260**
Styling the collapsed navbar **261**
Customizing the drop-down menus 263
Styling the horizontal navbar **264**
Enabling Flexbox support **265**
Designing a complex responsive layout **265**
Adjusting the large and extra-large layout 267
Adjusting the medium layout for tablet-width viewports 269
Adjusting headings, font sizes, and buttons 273
Enhancing the primary column 276
Adjusting the tertiary column 279
Fine touches for multiple viewports 282
Laying out a complex footer **282**
Setting up the markup **282**
Adjusting for tablet-width viewports 285
Adding a targeted responsive clearfix 287
Refining the details **288**
Summary **292**
Assessments **292**

Lesson 10: Bootstrapping E-Commerce **294**

Surveying the markup for our products page **296**
Styling the breadcrumbs, page title, and pagination **298**
Adjusting the products grid **301**
Don't forget the Card module 306
Cards with the CSS3 Flexbox layout module 308
Styling the options sidebar **312**
Setting up basic styles 313
Styling the Clearance Sale link 314
Styling the options list 317
Adding Font Awesome checkboxes to our option links 320
Using Sass mixins to arrange option links in columns 324
Adjusting the options list layout for tablets and phones 325
Collapsing the options panel for phone users 329
Adding a search form to your designing **332**
Using the Typeahead plugin 333
Summary **335**

Assessments **336**

Lesson 11: Bootstrapping a One-Page Marketing Website **338**

Overview **338**
Surveying the starter files **341**
Viewing the page content **343**
Adding Font Awesome to our project **343**
Adjusting the navbar **344**
Customizing the jumbotron **347**
Refining the jumbotron message design 350
Beautifying the features list **353**
Tackling customer reviews **357**
Positioning and styling captions 359
Refining the caption position 361
Adjusting for tiny screens 363
Creating attention-grabbing pricing tables **364**
Setting up the variables, files, and markup 364
Beautifying the table head 367
Styling the table body and foot 369
Differentiating the packages 370
Adjusting for small viewports 373
Providing a visual hierarchy to our tables 374
Adding the final touches **376**
Adding ScrollSpy to the navbar **378**
Animating the scroll 381
Summary **382**
Assessments **383**

Assessment Answers **385**

1
Setting up Our First Blog Project

Bootstrap is the most popular HTML, CSS, and JavaScript framework in the modern web development world today. Whether you are new to web development or an experienced professional, Bootstrap is a powerful tool for whatever type of web application you are building. With the release of version 4, Bootstrap is more relevant than ever and brings a complete set of components that are easy to learn and use. In this lesson, I'll show you how to set up our first blog project with all necessary files.

Implementing framework files

Before we get into building the basic template for a Bootstrap project, we should review the files that we need to include, so as to make the framework run properly. Minimal requirement for the purpose: one CSS file and two JavaScript files. These files can either be served from the Bootstrap **Content Delivery Network** (**CDN**) or downloaded and included directly in our project. When using CDN, simply include the following line of code in the head of your file:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/4.0.0-alpha.2/css/bootstrap.min.css" integrity="sha384-
y3tfxAZXuh4HwSYylfB+J125MxIs6mR5FOHamPBG064zB+AFeWH94NdvaCBm8qnd"
crossorigin="anonymous">
```



CDNs help to distribute bandwidth across multiple servers and allow users to download static content from a closer source.

Bootstrap can be loaded from `https://www.bootstrapcdn.com/`. BootstrapCDN is powered by MaxCDN which can be found at `https://www.maxcdn.com/`.

Inserting the JavaScript files

As I mentioned earlier, we need to include two JavaScript files to implement the framework properly. The files are the jQuery and Bootstrap JavaScript framework files. As with the CSS file, you can either do this through the use of a CDN or download and insert the files manually. The JavaScript files should be inserted at the bottom of your page right before the closing `</body>` tag. When using CDN, insert the following lines of code:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.2/
js/bootstrap.min.js" integrity="sha384-vZ2WRJMwsjRMW/8U7i6PWi6AlO1L79s
nBrmgiDpgIWJ82z8eA5lenwvxbMV1PAh7" crossorigin="anonymous"></script>
```

If you prefer to insert the files yourself, go back to the Bootstrap package you downloaded earlier and locate the `/js` directory. There will be a few files here but the one you want is `bootstrap.min.js`. You'll need to also head to `http://jquery.com` to download the jQuery framework file. Once you've done that, drop both files into the `/js` directory for your own project. Next, enter the following lines of code at the bottom of your page template. Make sure jQuery is loaded before `bootstrap.min.js`. This is critical; if you load them in the opposite order, the framework won't work properly:

```
<script src="/path/to/your/files/jquery.min.js"></script>
<script src="/path/to/your/files/bootstrap.min.js"></script>
```

That concludes the explanation of the key Bootstrap framework files you need to include to get your project started. The next step will be to set up the basic starter template so you can begin coding your project.

The starter template

The basic starter template is the bare bones of what you'll need to get a page going using Bootstrap. Let's start by reviewing the code for the entire template and then I'll break down each critical part:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags always come first -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
    <meta http-equiv="x-ua-compatible" content="ie=edge">

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/
bootstrap/4.0.0-alpha.2/css/bootstrap.min.css" integrity="sha384-
y3tfxAZXuh4HwSYylfB+J125MxIs6mR5FOHamPBG064zB+AFeWH94NdvaCBm8qnd"
crossorigin="anonymous">
  </head>
  <body>
    <h1>Hello, world!</h1>

    <!-- jQuery first, then Bootstrap JS. -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-
alpha.2/js/bootstrap.min.js" integrity="sha384-vZ2WRJMwsjRMW/8U7i6PWi6
AlO1L79snBrmgiDpgIWJ82z8eA5lenwvxbMV1PAh7" crossorigin="anonymous"></
script>
  </body>
</html>
```

HTML5 DOCTYPE

Like most projects nowadays, Bootstrap uses the HTML5 DOCTYPE for its template. That is represented by the following line of code:

```
<!DOCTYPE html>
```

Avoid using other DOCTYPES such as XHTML strict or transitional or unexpected issues will arise with your components and layouts.

Structuring the responsive meta tag

Bootstrap is a mobile-first framework so the following `meta` tag needs to be included to allow for responsive web design. To make sure your project renders properly on all types of devices, you must include this `meta` tag in the `<head>` of your project:

```
<meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
```

If you're interested in learning more about how responsive web design works in Bootstrap, you should check out the documentation at `http://v4-alpha.getbootstrap.com/layout/responsive-utilities/`.

That brings to a close the most important parts of the template that you need to be aware of. The remainder of the code in the starter template should be straightforward and easy to understand.

Normalizing and rebooting

As I mentioned earlier, Bootstrap uses `normalize.css` as the base CSS reset. With the addition of the Reboot reset, Bootstrap extends Normalize and allows for styling to only be done using CSS classes. This is a much safer pattern to follow, as it's much easier to deal with CSS specificity if you are NOT using CSS IDs for styling purposes. The CSS reset code is baked right into `bootstrap.min.css` so there is no need to include any further CSS files for the reset.

Taking the starter template further

Although we have our template set up, one of the main problems with static websites is when things change. If your project grew to 50, 100, or 500 pages and you wanted to possibly update to a new version of Bootstrap, you might be looking at having to update all of those files. This is extremely painful, to put it mildly. Now we enter static site generators.

Using a static site generator

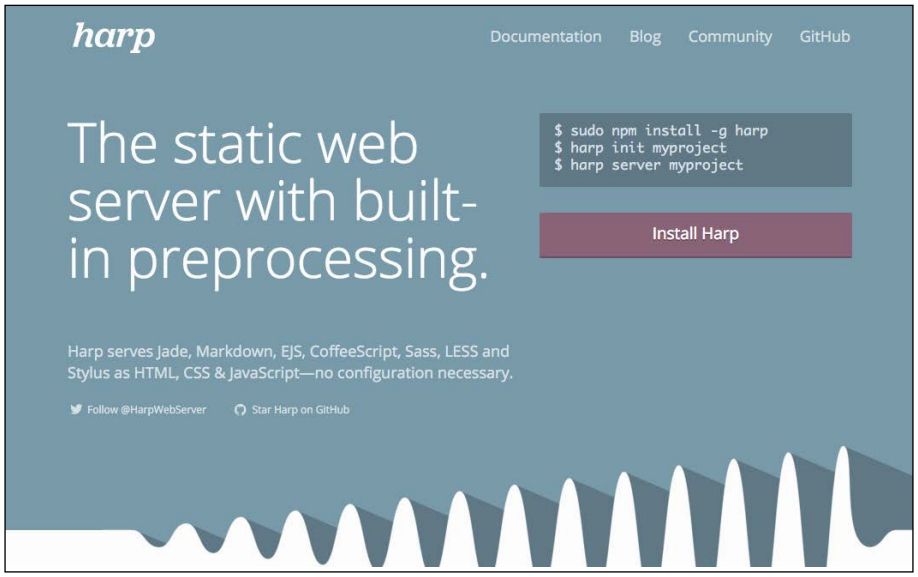
One of the hottest trends right now in web development is the use of static site generators. What exactly does that mean? Instead of having several static files that require updating every time something changes globally, you can use a series of base templates then load your body content into them. This is sometimes called **includes** or **partials**. This way, you only have one or two layout files that include the header and footer code.

Then, when something changes, you only have to update a few files instead of 500. Once your website is complete, you then generate a version that is plain HTML, CSS, and JavaScript, and deploy it to your server. This is what I would call creating your own frontend web development environment. This is also how most people work on larger projects nowadays to keep them manageable.

Converting the base template to a generator

Why don't we integrate the basic template into a generator so that I can show you what I'm talking about? My generator of choice is called Harp.js and you can install it by following the steps provided in the code folder (`Installation Manual`).

There are a number of great arguments for using a static site generator such as Harp.js: cleaner code, modern best practices, and more. However, the best reason is that it will just make your life simple. Instead of having to update a header on all 50 pages of the website, you can simply update the header partially and then have that compiled into all your templates. You can also take advantage of using variables to insert content and configuration.



Setting up the blog project

Let's start by creating a new directory and call it something like `Bootstrap Blog`. Open up that folder and create the following sub-directories:

- `css`
- `fonts`
- `img`
- `js`
- `partial`

css

The `css` directory will hold the Bootstrap framework's CSS file and a custom theme file which we'll build later on. Go to the Bootstrap source file directory and locate the `dist/css` folder. From there, copy `bootstrap.min.css` to our new blog project's `css` directory.

fonts

The `fonts` directory will hold either a font icon library such as Glyphicon or Font Awesome. Previously, Bootstrap shipped with Glyphicon but they have dropped it in version 4. If you wish to use it, you'll need to download the icon font set and then drop the files into this directory. You could also include a web font that you may want to use on your project in this directory. If you are looking for web fonts, a good place to start is Google Web Fonts.

img

The `img` directory will hold any images used in the blog.

js

The `js` or JavaScript directory will hold the Bootstrap framework JavaScript files. If you add any other third-party libraries, they should also be included in this directory. Go back to the Bootstrap source files one last time and locate the `dist/js` folder. From there, copy `bootstrap.min.js` to the `js` directory in the blog project.

partial

The `partial` directory will hold any reusable snippets of code that we want to use in multiple locations throughout our templates or web pages, for example, the header and footer for our project. It's important to note you can have as many partial files as you like or use none at all. Within this folder, create two new files and name them `_header.ejs` and `footer.ejs`. For now, you can leave them blank.

EJS files

EJS stands for **Embeddable JavaScript**. This is a type of template file that allows us to use things such as partials and variables in our templates. Harp also supports Jade if you prefer that language. However, I prefer to use EJS because it is very similar to HTML and therefore really easy to learn. If you've ever used WordPress, it is very similar to using template tags to insert bits of content or components into your design.

Setting up the JSON files

Each Harp project has at least two JSON files that are used for configuring a project. **JSON** stands for **JavaScript Object Notation** and it's a lightweight format for data interchange. If that sounds complicated, don't worry about it. The actual coding of a JSON file is actually really straightforward, as I will show you now.

The first is called `_harp.json` and it's used for configuring global settings and variables that will be used across the entire blog. In this case, we're going to set up a global variable for the name of our project that will be inserted into every page template. Start by creating a new file in the root of blog project and call it `_harp.json`. Within the file, insert the following code:

```
{
   "globals": {
      "siteTitle": "Learning Bootstrap 4"
   }
}
```

Here's what's happening in this code:

- We're using the `globals` keyword so any variables under this will be available across all of our templates
- I've created a new variable called `siteTitle` which will be the title of the project
- I've inserted `Learning Bootstrap 4` as the title for the project

That completes the setup of the global `_harp.json` file. In a little bit, I'll show you how to add the variable we set up to the main layout file.

Creating the data JSON file

The next thing we need to do is set up the `_data.json` file that can hold template-specific variables and settings. For our project, we'll set up one variable for each page template which will hold the name of the page. Create another file in the root of the blog project and name it `_data.json`. In that file, insert the following code:

```
{
   "index": {
      "pageTitle": "Home"
   }
}
```

Let me break down this code for you:

- `index` refers to a filename. In this case, it will be our home page. We haven't actually created this file yet but that is okay as we will in the next steps.
- I've created a variable called `pageTitle` which will refer to the title of each page template in our project
- Since this is the `index` template, I've assigned a value or name of `Home` to it

That completes the setup of the `_data.json` file for now. Later on, we'll need to update this file once we add more page templates. For now, this will give us the minimum resources that we need to get our project going.

Setting up the layout

Let's go ahead and set up the layout file for our project. The layout is a separate file that will be a wrapper for the content of all of our pages. It contains things such as the `<head>` of our page, a header partial, and a footer partial. This is one of the advantages to using a static site generator. We don't have to define this on every page so if we want to change something in our header, we only change it in the layout. On the next compile, all of the page templates' headers will be updated with the new code.

Create a new file in the root of the blog project called `_layout.ejs`. Since this is technically a type of layout file, we'll be creating it as an EJS template file. Once you've created the file, insert the following code into it:

```
<!DOCTYPE html>
<html lang="en">
```

```
  <head>
    <!-- Required meta tags always come first -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
    <meta http-equiv="x-ua-compatible" content="ie=edge">

    <title><%- pageTitle %> | <%- siteTitle %></title>

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="css/bootstrap.min.css">
  </head>
  <body>

    <%- partial("partial/_header") %>

    <%- yield %>

    <%- partial("partial/_footer") %>

    <!-- jQuery first, then Bootstrap JS. -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

There are a few things going on here, so let me explain everything that you need to know:

- The top is your standard `<head>` section that matches the basic Bootstrap template we covered in the first lesson, with, however, a few differences.
- Note the `<title>` tag and that it includes the two variables we set up previously. One for the `pageTitle` variable which will print out **Home** if we are on the index page. The second `siteTitle` variable will always print out **Learning Bootstrap 4** as that is what we set it to in `_harp.json`.
- Skip down to the `<body>` section and you'll see some new lines of code. The first partial is for our header. This line will include a snippet of code that we'll set up later that contains the markup for our header. Since this will be the same on all pages, we only need to include it here once instead of on every page.
- The second section in the `<body>` is the `<%- yield %>` tag. This is a Harp template tag and here is where the contents of our page template files will

load. In the case of our index page, any code that we enter into `index.ejs` (that we need to create still) will be loaded in at this place in the layout.

- The final line of code is a partial for the footer and works exactly the same as the header. At a minimum, you should have a header and footer partial in your projects. However, you are free to add as many partials as you like to make your project more modular.

That completes the setup of the layout. Next, let's move on to coding the header and footer partials.

Setting up the header

Let's set up our first partial by coding the header. We'll use the Bootstrap `navbar` component here for our global navigation for the blog. In the `partial` directory, open up the `_header.ejs` file that you created a little earlier and insert the following code:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="#">Learning Bootstrap 4</a>
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link" href="index.html">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="about.html">About</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="contact.html">Contact</a>
    </li>
  </ul>
  <form class="form-inline pull-xs-right">
    <input class="form-control" type="text" placeholder="Search">
    <button class="btn btn-primary" type="submit">Search</button>
  </form>
</nav>
```

If you're a Bootstrap 3 user, you'll likely notice that the code to render a `navbar` in version 4 is much cleaner. This will make the `navbar` much easier to use and explain. Let me break down the code for you:

- On the `<nav>` tag, we have a few classes we need to include. `.navbar` is the standard class need for this component. `.navbar-light` will render a light-colored navbar for us. There are some other color options you can check out in the Bootstrap documents. Finally, the `.bg-faded` class is optional but I like

to include it as it makes the background of the navbar a little more subtle.

- The `.navbar-brand` class is unchanged from Bootstrap 3 and I've inserted `Learning Bootstrap 4` for this tag. Feel free to name it whatever you want.
- Next, we have our navigation list of links. The `<ul>` tag needs to have the two required classes here: `.nav` and `.navbar-nav`.
- Within the list, you'll notice three pages: `Home`, `About`, and `Contact`. These are going to be the pages we'll build out through later lessons so please fill them in now.



Note the `.active` class on the index page link. This is optional and you may not want to include it in this manner as this is a global navigation..

- Finally, I've included a search form and used the `.pull-xs-right` to align it to the right of the navbar. If you're familiar with Bootstrap 3, this class used to simply be called `.pull-right`. In Bootstrap 4, you have more control of the alignment based on the viewport size of your device. If you always want the search bar to be aligned to the right then use the `-xs` value in the class.

Save the file and that will complete the setup of the header partial. Let's move on to setting up the footer.

Setting up the footer

The footer partial works exactly like the header. Open up the `_footer.ejs` file in the `partial` directory that we created earlier and paste in the following code:

```
<!-- footer //-->
<div class="container">
   <div class="row">
      <div class="col-lg-12">
         Learning Bootstrap 4 2016
      </div>
   </div>
</div>
```

The `footer` content is going to be quite basic for our blog. Here's a breakdown of the code:

- I'm using the `.container` class to wrap the entire footer, which will set a max width of 1140 px for the layout. The navbar wasn't placed into a container so it will stretch to the full width of the page. The `.container` class

will also set a left and right padding of .9375rem to the block. It's important to note that Bootstrap 4 uses REMs for the main unit of measure. EMs has been deprecated with the upgrade from version 3. If you're interested in learning more about REMs, you should read this blog post: `http://snook.ca/archives/html_and_css/font-size-with-rem`.

- It's also important to note that the column classes have NOT changed from Bootstrap 3 to 4. This is actually a good thing if you are porting over a project, as it will make the migration process much easier. I've set the width of the footer to be the full width of the container by using the `.col-lg-12` class.
- Finally I've entered some simple content for the footer: `Learning Bootstrap 4 2016`. Feel free to change this up to whatever you want.
- Save the file and the footer setup will be complete.

We're getting closer to having our Harp development environment set up. The last thing we need to do is set up our `index` page template and then we can compile and view our project.

Creating our first page template

For our first page template, we're going to create our `Home` or `index` page. In the root of the blog project, create a new file called `index.ejs`. Note this file is not prepended with an underscore like the previous files. With Harp, any file that has the underscore will be compiled into another and ignored when the files are copied into the production directory. For example, you don't want the compiler to spit out `layout.html` because it's fairly useless with the content of the `Home` page. You only want to get `index.html`, which you can deploy to your web server. The basic thing you need to remember is to not include an underscore at the beginning of your page template files. Once you've created the file, insert the following code:

```
<div class="container">
  <div class="row">
    <div class="col-lg-12">
      <h1>hello world!</h1>
    </div>
  </div>
</div>
```

To get us started, I'm going to keep this really simple. Here's a quick breakdown of what is happening:

- I've created another `.container` which will hold the content for the `Home` page

- Within the container, there is a full-width column. In that column, I've inserted an `<h1>` with a `hello world!` message

That will be it for now. Later on, we'll build this page out further. Save the file and close it. We've now completed setting up all the basic files for our Harp development environment. The last step is to compile the project and test it out.

Compiling your project

When we compile a project in Harp, it will find all the different partial, layout, and template files and combine them into regular HTML, CSS, and JavaScript files. We haven't used any Sass yet but, as with the template files, you can have multiple Sass files that are compiled into a single CSS file that can be used on a production web server. To compile your project, navigate to the root of the blog project in the Terminal. Once you are there, run the following command:

```
$ harp compile
```

If everything worked, a new blank line in the terminal will appear. This is good! If the compiler spits out an error, read what it has to say and make the appropriate changes to your template files. A couple of common errors that you might run into are the following:

- Syntax errors in `_harp.json` or `_data.json`
- Syntax errors for variable or partial names in `_layout.ejs`
- If you have created additional page templates in the root of your project, and not included them in `_data.json`, the compile will fail

Once your compile is successful, head back to the root of the blog project and notice that there is a new `www` directory. This directory holds all the compiled HTML, CSS, and JavaScript files for your project. When you are ready to deploy your project to a production web server, you would copy these files up with FTP or using another means of file transfer. Every time you run the `harp compile` command in your project, these files will be updated with any new or edited code.

Running your project

Harp has a built-in web server that is backed by Node.js. This means you don't need a web hosting account or web server to actually test your project. With a simple command, you can fire up the built-in server and view your project locally. This is also really great if you are working on a project somewhere with no Internet connection. It will allow you to continue building your projects Internet-free. To run the server, head back to the Terminal and make sure you are still in the root directory

of your blog project. From there, enter the following command:

```
$ harp server
```

In the Terminal, you should see a message that the server is running. You are now free to visit the project in a browser.

Viewing your project

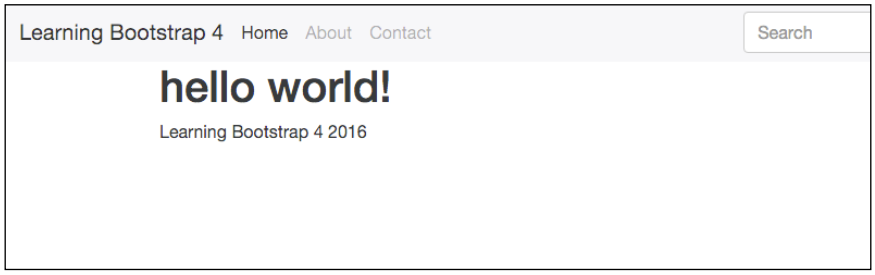
Now that the project is up and running on the web server, simply navigate to the following URL to view it: `http://localhost:9000`.

By default, Harp runs on port `9000` but you can specify a different port by modifying the last command. Go back to the terminal and quit the server by hitting *Ctrl* + *C*. Now enter the following command:

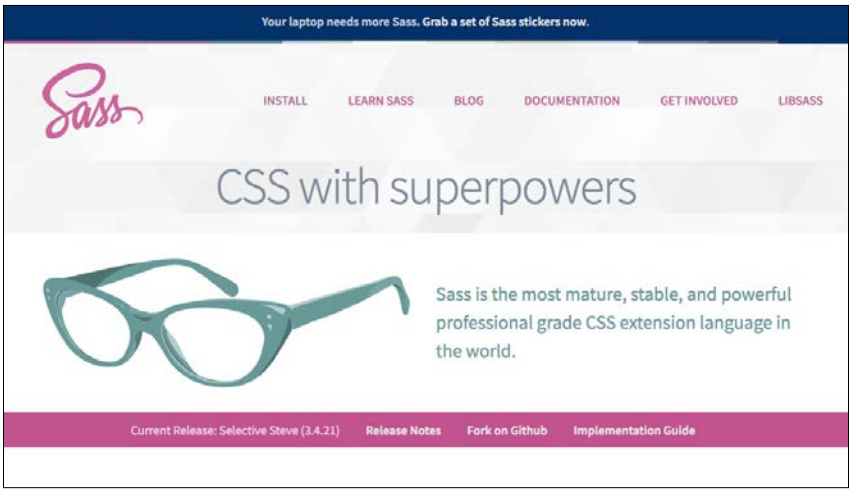
```
$ harp server  --port 9001
```

Using this command, you can invoke any port you would like to use. Head back to the web browser again and change the URL slightly to read `http://localhost:9001`.

Your project should load for you and look something like this:



It might not be much to look at right now but it works! Your project is successfully set up and running. In future lessons, we'll add to this page and build some more using additional Bootstrap 4 components.



A note about Sass

When building a project with Bootstrap 4, there are two ways you can work with Sass. The first would be by editing the actual Bootstrap framework files and then recompiling them using Grunt. This is useful if you'd like to use something such as Flexbox for your grid layout. I'll discuss this in greater depth in the next lesson. The other way you might want to use Sass is to craft a unique theme that applies a custom look and feel to Bootstrap. This is done in the actual Harp project. Within the `css` directory, you can include Sass files; when you compile your Harp project, they will be converted to regular CSS, as Harp has a built-in Sass compiler. Then it is just a simple exercise of including those additional files in your layout template. I'll also get into that a little later in the course but I wanted to point out the difference now.

Browser support

Bootstrap 4 does not support Internet Explorer version 8 and earlier. Bootstrap 4 also comes with optional Flexbox support. Only Internet Explorer versions 11 and higher support the CSS3 Flexible Box Layout Module. Besides Internet Explorer 8 and earlier, Bootstrap supports all major browsers, including many mobile browsers.

Vendor prefixes

CSS3 introduced vendor-specific rules, which offer you the possibility of writing some additional CSS, applicable for only one browser. At first sight, this seems the exact opposite of what we want. What we want is a set of standards and practicalities that work the same with every browser and a standard set of HTML and CSS which has the same effect and interpretation for every browser. These vendor-specific rules are intended to help us reach this utopia. Vendor-specific rules also provide us with early implementations of standard properties and alternative syntax. Last but not least, these rules allow browsers to implement proprietary CSS properties that would otherwise have no working standard (and may never actually become the standard).

For these reasons, vendor-specific rules play an important role in many new features of CSS3. For example, animation properties, `border-radius`, and `box-shadow`: all did depend on vendor-specific rules in past years. You can easily see that some properties may evolve from vendor prefixes to standard, because currently, most browsers support the `border-radius`, and `box-shadow` properties without any prefix.

Vendors use the following prefixes:

- **WebKit**: `-webkit`
- **Firefox**: `-moz`
- **Opera**: `-o`
- **Internet Explorer**: `-ms`

Consider the following CSS code:

```
transition: all .2s ease-in-out;
```

For full browser support, or to support at least the browser supported by Bootstrap, we'll have to write:

```
-webkit-transition: all .2s ease-in-out;
-o-transition: all .2s ease-in-out;
transition: all .2s ease-in-out;
```

More information about the transition property and browser support can also be found at the following URL: `http://caniuse.com/#feat=css-transitions`.

Because of different browsers and their different versions, browsers may use different vendor prefixes to support the same property in writing cross-browser CSS code which can become very complex.

Bootstrap's Sass code, which compiles into CSS code does not contain any prefixes. Instead of using prefixes, the PostCSS autoprefixer has been integrated into Bootstrap's build process. When you create your own build process you should also use the PostCSS autoprefixer.

Troubleshooting

If things are not running smoothly, you should ask yourself the following questions:

- Is your markup properly structured? Any unclosed, incomplete, or malformed tags, classes, and so on present?
- You might find it helpful to do the following:
- Work back through the preceding steps, double-checking things along the way.
- Validate your HTML to ensure it's well formed.
- Compare the completed version of the exercise files with your own.
- Refer to the Bootstrap documentation for new updates to the relevant tag structures and attributes.
- Place your code in a snippet at `https://jsfiddle.net/` or `https://codepen.io/`, and share it with the good folks at `http://stackoverflow.com/` for help.

When we have so many moving parts to work with, things are bound to happen and these are some of our best survival methods!

Bootply is a playground for Bootstrap, CSS, JavaScript, and jQuery; you can use it to test your HTML code. You can also add your compiled CSS code, but Bootply cannot compile your CSS code.



Bootply can be found online at `http://www.bootply.com/`.

Summary

That brings the first lesson to a close. In this lesson, we have set up our first blog project with all the necessary files.

Now that our environment is set up and ready to go, we'll start coding the blog in the next lesson. To get us started, we'll jump right into learning about how to use a Flexbox layout in Bootstrap.

Assessments

1. Which of the following does NOT use the Bootstrap framework?
    i. JavaScript
    ii. HTML
    iii. PHP
    iv. CSS
2. What is the full form of CDN?
    i. Cloud Deployment Network
    ii. Content Delivery Network
    iii. Context Delivery Network
    iv. Content Deployment Network
3. What does Bootstrap use as a base CSS reset?
    i. `normalize.css`
    ii. `reboot.css`
    iii. `bootstrap.css`
    iv. `bootstrap.min.css`
4. Which command is used to know the version number of Node.js installed in your system?
    i. `$ node –ver`
    ii. `$ node -vs`
    iii. `$ node –v`
    iv. `$ node –vr`
5. What is the command used for installing Harp in Mac?
    i. `$ sudo npm install -g harp`
    ii. `$ npm install -g harp`
    iii. `$ node install hard`
    iv. `$ harp install`

6. What does Bootstrap use for CSS and JavaScript build system?
    i. Selenium
    ii. Drone
    iii. Grunt
    iv. Ruby
7. Which of the following sets the basic styles for the navigation bar?
    i. `<navbar class="navbar navbar-light bg-faded" role="navigation"></navbar>`
    ii. `<nav classnav="navbar navbar-light bg-faded" rolenav="navigation"></nav>`
    iii. `<nav class="navbar navbar-light bg-faded" role="navigation"></nav>`
    iv. `<nav navclass="navbar navbar-light bg-faded" navrole="navigation"></nav>`
8. Which of the following is NOT a build tool used in Bootstrap?
    i. Node.js
    ii. Grunt.js
    iii. Surge.js
    iv. Harp.js
9. In order to install Grunt, which of the following commands will you run?
    i. `npm install -g grunt-cli`
    ii. `npm install  grunt-cli`
    iii. `npm install -g grunt-cli`
    iv. `npm install  grunt-cli`
10. What is used to run the documentation website and to compile the core Sass files into regular CSS?
    i. Grunt
    ii. Ruby
    iii. Perl
    iv. Bundlenr

11. Which of the following directories will hold any reusable snippets of code that you want to use in multiple locations throughout your templates or web pages?
    i. js
    ii. partial
    iii. JSON
    iv. CSS

2
Jumping into Flexbox

Alright, now that we have finished setting up our first blog, let's jump into an actual great new feature of Bootstrap 4. The latest version of the framework comes with CSS Flexbox support. The goal of the Flexbox layout module is to create a more effective way of designing a layout for a website or web application. The grid of boxes is aligned in a way that distributes them across their container even if their size is unknown. This is where the "Flex" in Flexbox comes from.

The motivation for a flexible box arose from a web design for mobiles. A way to have a section grow or shrink to best fill the available space was needed when building responsive web applications or websites. Flexbox is the opposite of block layouts that are either vertically or horizontally driven. It's important to note that Flexbox is generally best suited for use when designing web applications. The traditional grid method still works best for larger websites.

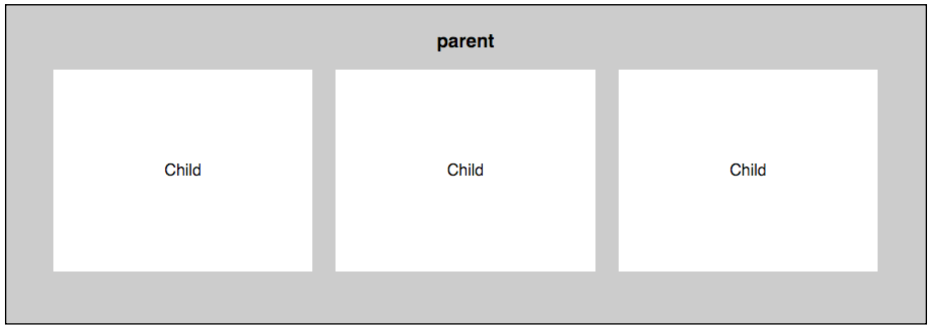
In our blog project, we're going to use Flexbox to create a homepage. There will be several rows of blocks, each being a post. I'll show you a few ways to lay the blocks out and different ways you can customize the contents of each block, all using the new Flexbox layout in Bootstrap.

Flexbox basics and terminology

Before we go too far, we should define a few Flexbox basics and some terminology that I'll use throughout the lesson. Every Flexbox layout is dependent on an outer container. As we move through the lesson, I'll refer to this container as the parent. Within the parent container there will always be a collection of boxes or blocks. I'll refer to these boxes as children or child elements of the parent. Why don't we start by talking a little bit more about why you would want to use Flexbox? The main purpose of Flexbox is to allow for the dynamic resizing of child boxes within their parent container.

This works for the resizing of both width and height properties on-the-fly. Many

designers and developers prefer this technique as it allows for easier layouts with less code:

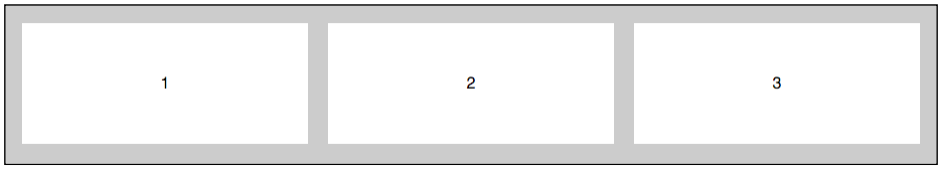


Ordering your Flexbox

Flexbox is a really powerful module as it comes with several properties that you can customize. Let's quickly go over some more basics before we fully take the plunge and use Flexbox in Bootstrap. Let's start by talking about the order of child boxes. By default, they will appear in the order that you insert them in the HTML file. Consider the following code:

```
<div class="parent">
   <div class="child">
   1
   </div>
   <div class="child">
   2
   </div>
   <div class="child">
   3
   </div>
</div>
```

A proper CSS will produce a layout that looks like this:



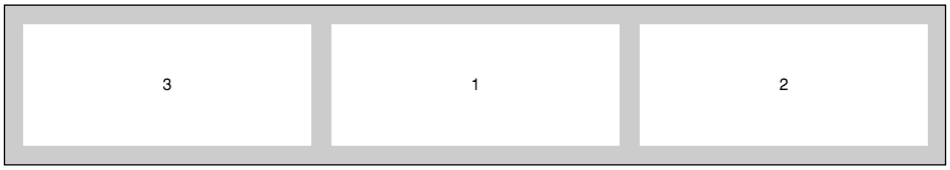
Here's the CSS to produce this layout if you are following along at home:

```
.parent {
  display: flex;
  background: #ccc;
  padding: 10px;
  font-family: helvetica;
}

.child {
  padding: 10px;
  margin: 10px;
  background: #fff;
  flex-grow: 1;
  text-align:center;
  height: 100px;
  line-height: 100px;
}
```

Now using an `order` property we can reorder the children using some CSS. Let's put the third box at the beginning. If you are reordering some blocks, you need to define the position for each one; you can't simply enter the value for a single block. Add the following CSS to your style sheet:

```
.child:nth-of-type(1) {
  order: 2;
}
.child:nth-of-type(2) {
  order: 3;
}
.child:nth-of-type(3) {
  order: 1;
}
```

I'm using the `nth-of-type` pseudo selector to target each of the three boxes. I've then used the `order` property to set the third box to the first position. I've also adjusted the other two boxes to move them over one space. Save the file and your boxes should now look like this:



As you can see, the third box has moved to the first position. It's as easy as that to rearrange blocks on boxes on a page. I think you'll likely see how this could be useful for coding up a web application dashboard.

Stretching your child sections to fit the parent container

Another important Flexbox feature is the ability to stretch the width of the child boxes to fit the full-width of the containing parent. If you look at the preceding CSS you'll notice a `flex-grow` property on the `.child` class. The property is set to 1 which means that the child boxes will stretch to equally fill their parent. You could also do something where one box is set to a different value, using the `nth-of-type` selector, and then it would be wider than the others. Here's the code to create equal-width columns as that is what you'll likely do in most cases:

```
.child {
   flex-grow: 1;
}
```

Changing the direction of the boxes

By default in Flexbox, the child boxes will be in a row going left to right. If you like, you can change the direction using the `flex-direction` property. Let's try out a few different directions. First let's review our base HTML code again:

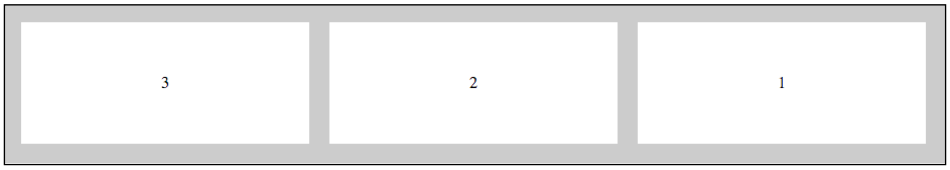
```
<div class="parent">
   <div class="child">
     1
   </div>
   <div class="child">
     2
   </div>
   <div class="child">
     3
   </div>
</div>
```

Here's the base CSS we wrote a little earlier. However, this time we'll add the `flex-direction` property (with a value of row-reverse) to the `.parent` class. This will reverse the order of the boxes:

```
.parent {
  display: flex;
  flex-direction: row-reverse;
```

```
  background: #ccc;
  padding: 10px;
}

.child {
  padding: 10px;
  margin: 10px;
  background: #fff;
  flex-grow: 1;
  text-align:center;
  height: 100px;
  line-height: 100px;
}
```

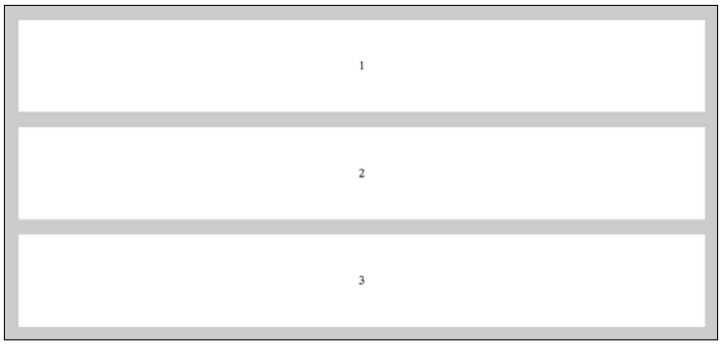
If you save the file and view it in a browser it should now look like this:



What if we wanted to order the boxes vertically so they were stacked on top of each other in descending order? We can do that by changing the `flex-direction` property to `column`:

```
.parent {
   ...
   flex-direction: column;
}
```

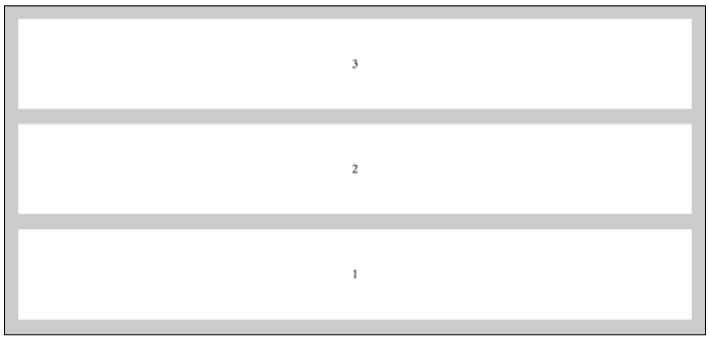
That configuration will produce a grid that looks like this:



Finally there is one more direction we can try. Let's do the same vertically stacked grid but this time we'll reverse it. We do that by switching the `flex-direction` property to `column-reverse`:

```
.parent {
   ...
   flex-direction: column-reverse;
}
```

That will produce a grid that looks like this:



Wrapping your Flexbox

By default all of your child boxes will try to fit onto one line. If you have a layout with several boxes, this may not be the look you want. If this is the case, you can use the `flex-wrap` property to wrap the child boxes as needed. Let's add more boxes to our original code with the following HTML:

```
<div class="parent">
   <div class="child">
   1
   </div>
   <div class="child">
   2
   </div>
   <div class="child">
   3
   </div>
   <div class="child">
   4
   </div>
   <div class="child">
```

```
      5
      </div>
      <div class="child">
      6
      </div>
      <div class="child">
      7
      </div>
      <div class="child">
      8
      </div>
      <div class="child">
      9
      </div>
   </div>
```

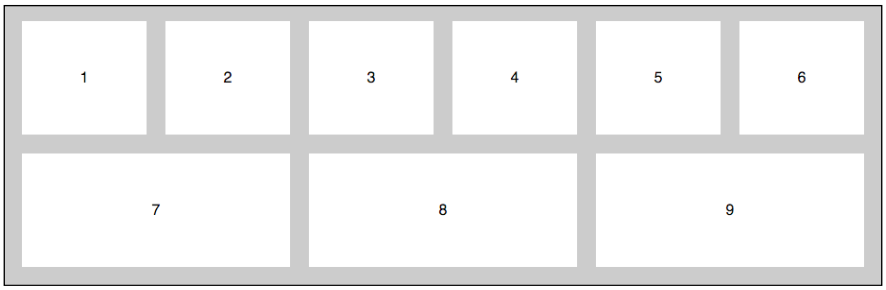
We now have nine boxes in our parent container. That should give us enough to work with to create a nice wrapping effect. Before we see what this looks like, we need to add some more CSS. Add the following properties to your CSS file:

```
.parent {
   ...
   flex-wrap: wrap;
}

.child {
   ...
   min-width: 100px;
}
```

I've added two new properties to achieve the layout we want. Let me break-down what is happening:

- I've added the `flex-wrap` property to the `.parent` class and set the value to `wrap`. This will wrap the boxes when it's appropriate.
- On the `.child` class I added a `min-width` of `100px`. I've done this so we can have some control on when the child boxes will break. If we don't add this, the width of the columns may get too thin.

Once you've added those properties to the existing code, save the file and test it. Your layout should now look something like this:



As you can see, we now have a two-row layout with six boxes on top and three below. Remember we added the `flex-grow property` previously, so the second row is stretching or growing to fit. If you want your boxes to always be equal you should use an even number, in this case 12. You could also remove the `flex-grow property`; then all the boxes would be the same width but they would not fill the layout the same way.

Creating equal-height columns

One of the best features of Flexbox is the ability to easily create equal height columns. In a regular horizontal layout, if your content is not the exact same length, each column will be a different height. This can be problematic for a web application layout because you usually want your boxes to be more uniform.

Let's check out some regular layout code and what it looks like in the browser:

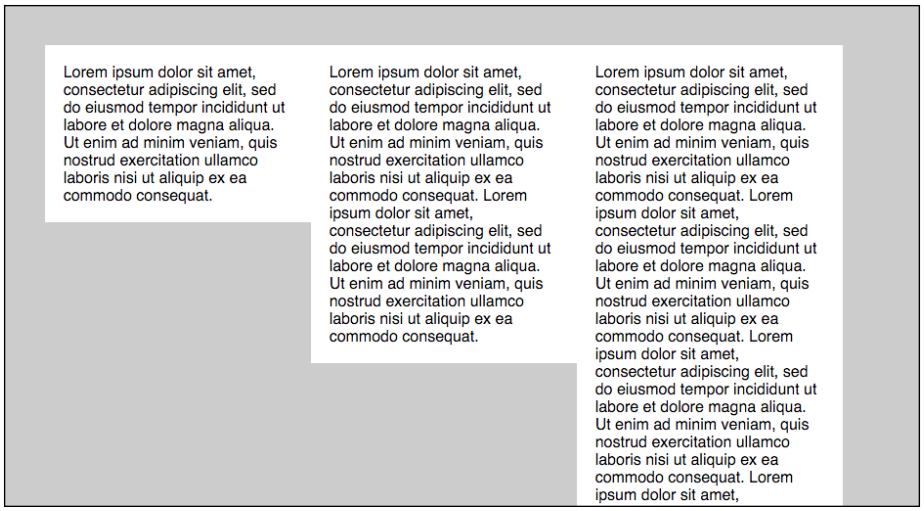
```
<div class="parent">
   <div class="child">
   Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat.
   </div>
   <div class="child">
   Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat. Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
   </div>
   <div class="child">
   Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed
do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
```

```
enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Lorem ipsum dolor sit
amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
consequat. Lorem ipsum dolor sit amet, consectetur adipiscing elit,
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
nisi ut aliquip ex ea commodo consequat. Lorem ipsum dolor sit amet,
consectetur adipiscing elit, sed do eiusmod tempor incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut
aliquip ex ea commodo consequat.
   </div>
</div>
```

I've created three columns with different amounts of text in each of them. Let's add some basic styling to these columns:

```
.parent {
  width: 100%;
  background: #ccc;
  font-family: helvetica;
  padding: 5%;
  float: left;
}

.child {
  padding: 2%;
  background: white;
  width: 25%;
  display: inline-block;
  float: left;
}
```

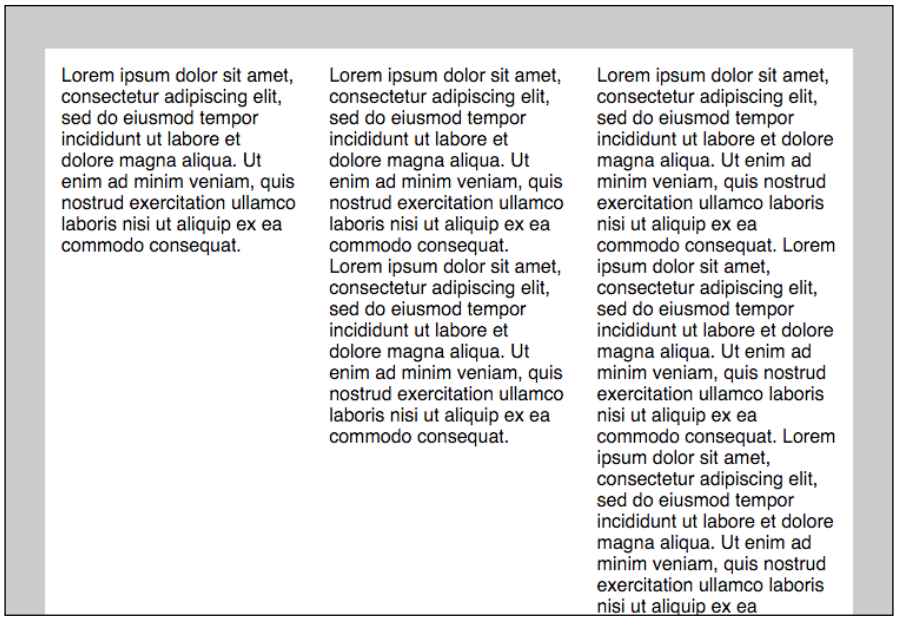
I've created a similar look and feel for this regular layout like our Flexbox. Let's see what this looks like if we view it in a browser:



That doesn't look very good does it? What would be better is if the two smaller columns stretched vertically to match the height of the longest column. The good news is this is really easy to do with Flexbox. Leave the HTML as it is but let's go and change our CSS to use a Flexbox approach:

```
.parent {
  display: flex;
  background: #ccc;
  font-family: helvetica;
  padding: 5%;
}

.child {
  padding: 2%;
  background: white;
  flex-grow: 1;
  min-width: 200px;
}
```

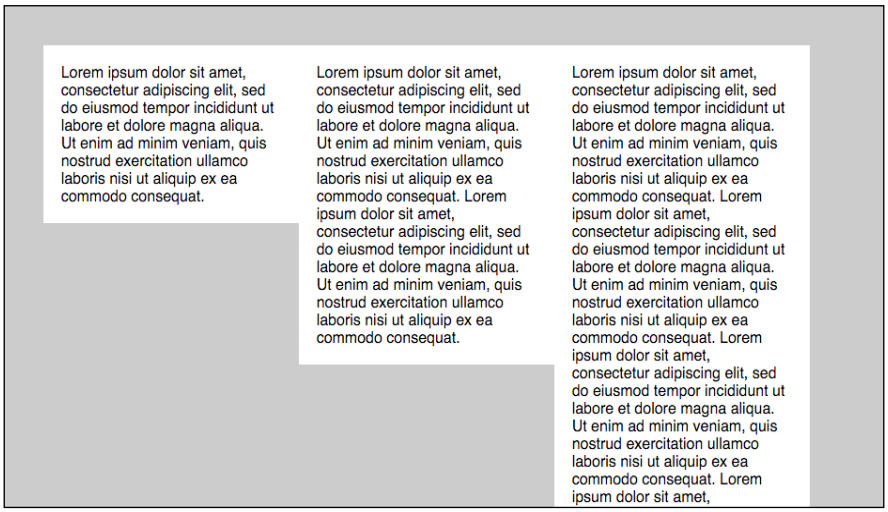
The preceding code is actually very similar to one of the first examples. Therefore, an equal height column comes standard right out of the Flexbox. I have added a `min-width` of `200px` to each column so that the text is readable. With the preceding CSS our layout will now look like this:



Perfect! Now the white background of each column has extended vertically to match the height of the tallest child. This looks much better and will allow for nicer horizontal alignment if you add additional rows of content. What's happening here is that the `align-items` property is defaulting to the stretch value. This value is what stretches the height of the columns to fit. There are some additional alignment values you can also try out. To continue, let's try out the `flex-start` value. Add the following CSS to the `.parent` class:

```
.parent {
   ...
   align-items: flex-start;
}
```

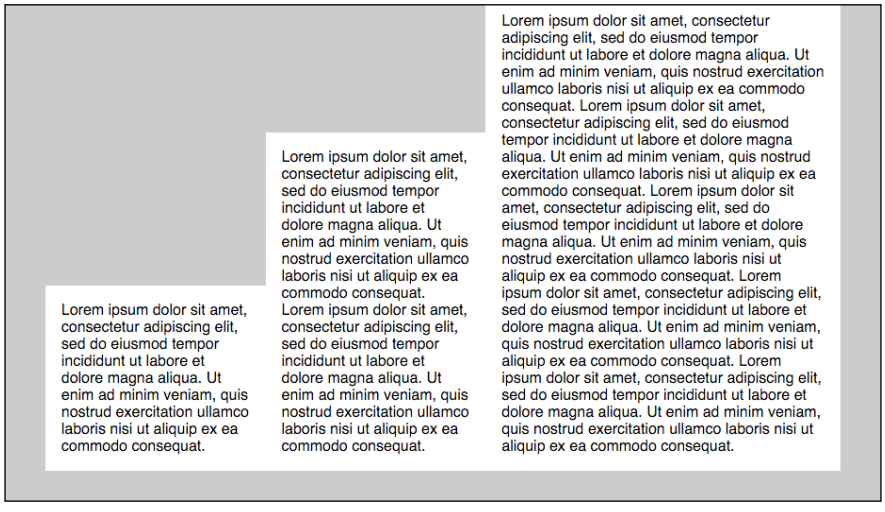
This configuration will actually undo the equal height columns and appear like a regular grid. Here's the image to refresh your memory:



A more useful value is the flex-end option, which will align the boxes to the bottom of the browser window. Change your CSS to:

```
.parent {
   ...
   align-items: flex-end;
}
```

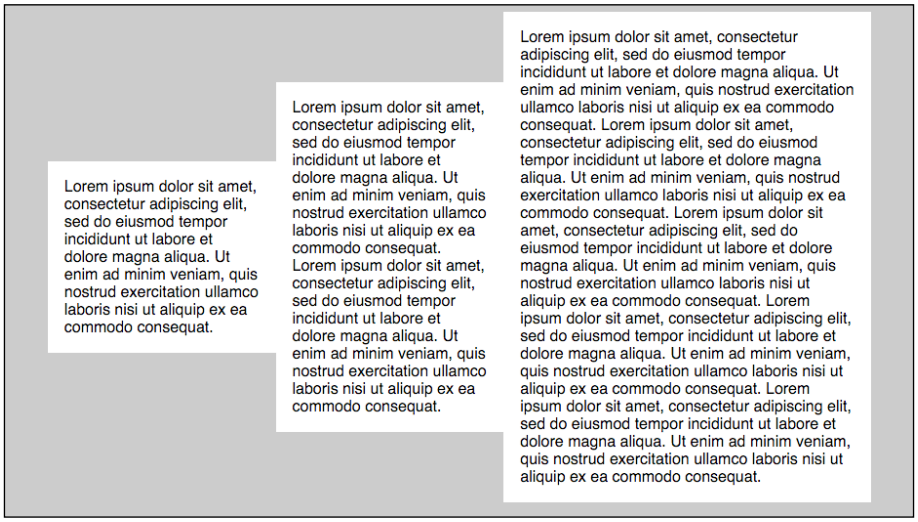
This setup will produce a grid that looks like this:



If you'd like to center your columns vertically in the layout, you can do that with the `center` value:

```
.parent {
   ...
   align-items: center;
}
```

If you go for this setup, your grid will look like this:



This is a just a taste of the properties you can use to customize the Flexbox grid. As I mentioned previously, I just wanted to give you a quick introduction to using Flexbox and some of the terminology that is needed. Let's take what we've learned and build on that by building a Flexbox grid in Bootstrap.

Setting up the Bootstrap Flexbox layout grid

Whether you are using Flexbox or not, the grid is based on Bootstrap's regular row and column classes. If you are familiar with the Bootstrap grid, this will work exactly as you expect it to. Before you start any Bootstrap project, you need to decide if you want to use a Flexbox or regular grid. Unfortunately, you can't use both at the same time in a Bootstrap project. Since the focus of this lesson is on Flexbox, we'll be using the appropriate grid configuration. By default Bootstrap is set up to use the regular grid. Therefore, we are going to need to edit the source files to activate the Flexbox

grid. Let's make a copy of the source files available in the code bundle.

Expand the ZIP file and rename it so you don't get confused. Call it something like `Flexbox Bootstrap`. Next we'll need to edit a file and recompile the source files to apply the changes.

Updating the Sass variable

To use the Flexbox grid, we need to edit a Sass variable in the `_variables.scss` file. The way Sass variables work is that you set a single value in the `_variables.scss` file. When you run the built-in compiler, that value is written into every component of the Bootstrap framework where it is needed. You can then grab the compiled `bootstrap.min.css` file and it will have all the required code you need to use the Flexbox grid:

1. In your new source file directory, using the Terminal, navigate to:

   ```
   $ scss/_variables.scss
   ```

2. Open the file in a text editor such as Sublime Text 2 or Notepad and find the following line of code:

   ```
   $enable-flex: false !default;
   ```

3. Change the false value to true. The line of code should now read:

   ```
   $enable-flex: true !default;
   ```

4. Save the file and close it. Before this change is applied, we need to recompile the source files. Since we downloaded a new version of the source files, we'll need to reinstall the project dependencies. Navigate to the root of the new Flexbox source files in the Terminal and run the following command:

   ```
   $ npm install
   ```

5. This will likely take a couple minutes and you can follow the progress in the Terminal. Once it's done we need to compile the project. To do this we use Grunt. To run the compiler, simply enter the following command into the Terminal:

   ```
   $ grunt
   ```

Again this will take a minute or two and you can follow the progress in the Terminal. Once it completes, the source files will have been compiled into the `/dist` directory. If it isn't clear, the production files that you want to use in your actual project will be compiled into the `/dist` directory.

Before we move onto our project, it would be a good idea to confirm that everything worked. Go back to your text editor and open the `dist/css/bootstrap.css` file from the root of your source files.

This is the un-minified version of the compiled Bootstrap CSS framework file. Once it's open do a quick find (*cmd* + *f* on *Mac* or *Ctrl* + *f* on Windows) and search for flex. If everything worked, it should quickly find an instance of flex in the file. This confirms that your compile worked.

Setting up a Flexbox project

A Flexbox project is structured exactly like a regular one. You just have to be sure to replace the `bootstrap.min.css` file in the `/css` directory with the new Flexbox version. Copy the project we made in the last lesson and paste it wherever you want on your computer. Rename the project to something like `Flexbox project`. Now open up that project and navigate to the `/css` directory. In a new window, open up the Flexbox sources files directory and navigate to the `/dist/css/` directory. Copy the `bootstrap.min.css` file from `/dist/css` into the `/css` directory in your new `Flexbox project`. You'll be prompted to overwrite the file and you should choose Yes. That's it, your new Flexbox project is ready to roll.

It would be a good idea to keep the Flexbox source files somewhere on your computer. In future projects, you can simply copy the compiled Flexbox version of the Bootstrap CSS over, saving you the trouble of having to recompile the source files each time you want a Flexbox layout.

Adding a custom theme

Before we code our first Flexbox grid, we need to add a custom CSS theme to our project. We're going to do this to add any custom look and feel styles on top of Bootstrap. In Bootstrap you never want to edit the actual framework CSS. You should use the cascading power of CSS to insert a theme for additional custom CSS or to overwrite existing Bootstrap styles. In a later lesson, I'll go into more depth on custom themes but for now let's set up a basic one that we can use for our Flexbox grid. First, let's start by creating a new file in the `/css` directory of our project called `theme.css`. For now, the file can be blank; just make sure you create it and save it.

Next we need to update our `_layout.ejs` file to include the theme file in our page. Open up `_layout.ejs` in a text editor and make sure it matches the following code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Required meta tags always come first -->
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1, shrink-to-fit=no">
```

```
    <meta http-equiv="x-ua-compatible" content="ie=edge">

    <title><%- pageTitle %> | <%- siteTitle %></title>

    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <link rel="stylesheet" href="css/theme.css">
  </head>
  <body>

    <%- partial("partial/_header") %>

    <%- yield %>

    <%- partial("partial/_footer") %>

    <!-- jQuery first, then Bootstrap JS. -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
    <script src="js/bootstrap.min.js"></script>
  </body>
</html>
```

I've added one line of code to the template that loads in `theme.css`:

```
<link rel="stylesheet" href="css/theme.css">
```



Note that this line of code is after `bootstrap.min.css`. This is important as our theme needs to be loaded last so that we can overwrite Bootstrap default styles if we want to. Our template is now up-to-date and we are ready to start with our first grid. Feel free to keep theme.css open as we'll be adding some styles to it in the next step.

Creating a basic three-column grid

Now that we've set up our project, let's go ahead and start doing some Bootstrap coding. The good news is that the Bootstrap column classes used with the Flexbox grid are exactly the same as the ones used in a regular grid. There is no need to learn any new class names. In your project folder, create a new file and name it `flexbox.ejs`.

Before you go any further, you need to add an instance for this page to `_data.json`. Otherwise your `harp compile` command will fail. Open up `_data.json` and add the

following code:

```
{
   "index": {
      "pageTitle": "Home"
   },
   "flexbox": {
      "pageTitle": "Flexbox"
   }
}
```

I've added a second entry for `flexbox.ejs` and given it this page title: Flexbox. Now we can safely start working on `flexbox.ejs` and the compile will work. Let's start with a simple three-column grid. Enter the following HTML code into `flexbox.ejs`:

```
<div class="container">
  <div class="row">
    <div class="col-md-4">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.</div>
    <div class="col-md-4">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.
Suspendisse tempus, justo sed posuere maximus, velit purus dictum
lacus, nec vulputate arcu neque et elit. Aliquam viverra vitae est eu
suscipit. Donec nec neque eu sapien blandit pretium et quis est.</div>
    <div class="col-md-4">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.
Suspendisse tempus, justo sed posuere maximus, velit purus dictum
lacus, nec vulputate arcu neque et elit. Aliquam viverra vitae est
eu suscipit. Donec nec neque eu sapien blandit pretium et quis est.
Sed malesuada sit amet mi eget pulvinar. Mauris posuere ac elit in
dapibus. Duis ut nunc at diam volutpat ultrices non sit amet nulla.
Aenean non diam tellus.</div>
  </div>
</div>
```

Let me breakdown what is happening here:

- Like in the earlier example, I've created three equal columns. Each one has a different amount of text in it.
- I'm using the `col-md-4` column class, as I want the three-column horizontal layout to be used for medium-size devices and upwards. Smaller devices will default to a single column width layout.
- I've also added a `.child` class to each of the column `<div>`s so that I can style

them.

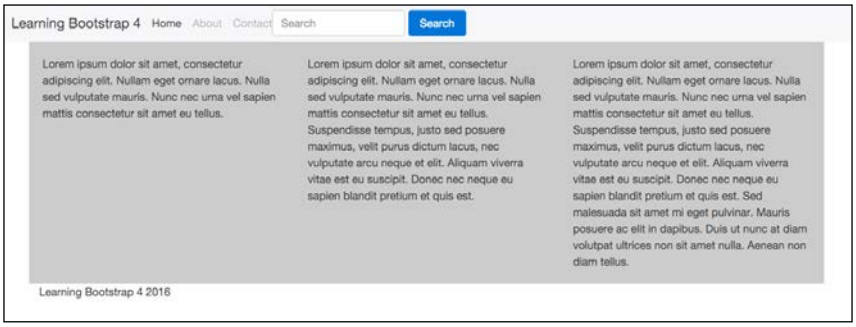
Now let's add a little CSS to theme.css so we can more easily see what is going on:

```
.child {

   background: #ccc;

   padding: 20px;

}
```

Here's what is happening with the `.child` class:

- I've added a light gray background color so we can easily see the child box.
- I've added some padding. Note that you can add padding to a Flexbox grid without worrying about breaking the grid. In a regular layout, this would break your box model and add extra width to the layout.

Here's what the finished layout should look like:



As you can see the light gray background has stretched to fit the height of the tallest column. An equal height column with almost no effort is awesome! You'll also notice that there is some padding on each column but our layout is not broken.

You may have noticed that I used the regular `.container` class to wrap this entire page layout. What if we want the layout to stretch the entire width of the browser?

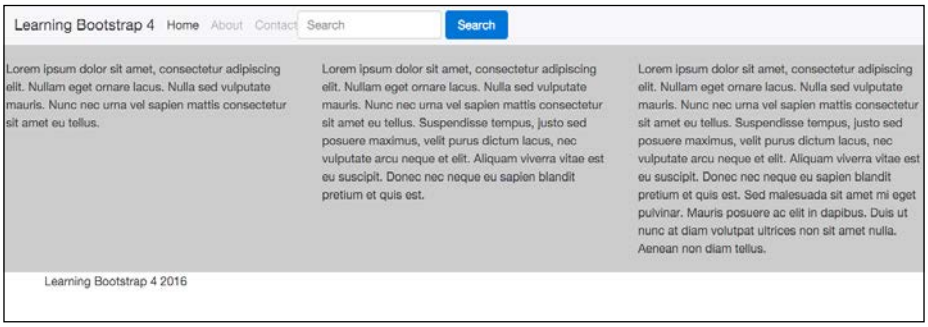
Creating full-width layouts

Creating a full-width layout with no horizontal padding is actually really easy. Just remove the container class. The HTML for that type of layout would look like this:

```
<div class="row">

  <div class="col-md-4 child">Lorem ipsum dolor sit amet, consectetur
```

```
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.</div>

  <div class="col-md-4 child">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.
Suspendisse tempus, justo sed posuere maximus, velit purus dictum
lacus, nec vulputate arcu neque et elit. Aliquam viverra vitae est eu
suscipit. Donec nec neque eu sapien blandit pretium et quis est.</div>

  <div class="col-md-4 child">Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nullam eget ornare lacus. Nulla sed vulputate mauris.
Nunc nec urna vel sapien mattis consectetur sit amet eu tellus.
Suspendisse tempus, justo sed posuere maximus, velit purus dictum
lacus, nec vulputate arcu neque et elit. Aliquam viverra vitae est
eu suscipit. Donec nec neque eu sapien blandit pretium et quis est.
Sed malesuada sit amet mi eget pulvinar. Mauris posuere ac elit in
dapibus. Duis ut nunc at diam volutpat ultrices non sit amet nulla.
Aenean non diam tellus.</div>

</div>
```

As you can see, I've simply removed the `<div>` with the `.container` class on it. Let's take a look at what the layout looks like now:



There we go, the columns are stretching right to the edges of the browser now. We've easily created a full-width layout that has equal height columns. Let's improve on this design by making each column an actual blog post and we'll also add more rows of posts.

Designing a single blog post

Let's start by designing the layout and content for a single blog post. At the very least, a blog post should have: a title, post-meta, description, and a read more link. Open up the `flexbox.ejs` file and replace the first column's code with this new

code:

```
<div class="col-md-4 child">
  <h3><a href="#">Blog Post Title</a></h3>
  <p><small>Posted by <a href="#">Admin</a> on January 1, 2016</
small></p>
  <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam
eget ornare lacus. Nulla sed vulputate mauris. Nunc nec urna vel
sapien mattis consectetur sit amet eu tellus.</p>
  <p><a href="#">Read More</a></p>
</div>
```

Let me breakdown what is happening here:

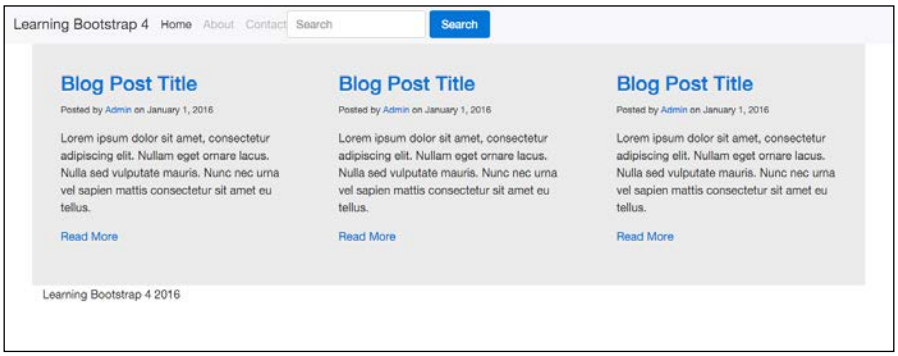
- I've added an `<h3>` tag with a link for the post title
- I've added some post-meta and wrapped it in a `<small>` tag so it is subtle
- I've left our description and added a read more link at the bottom

Now go ahead and copy and paste this code into the other two columns. If you want to play around with the length of the description text, feel free. For this example I'm going to keep it the same. When you're done, the entire page code should look like this. Note, I added the container `<div>`back in:

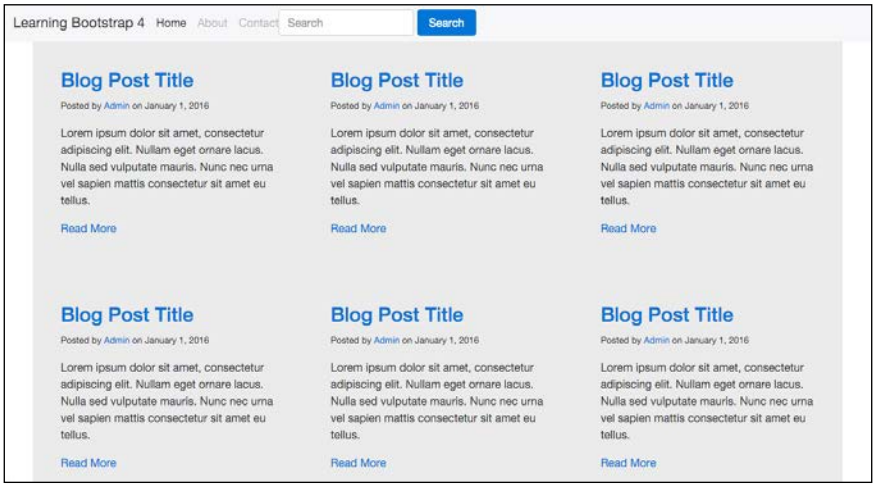
```
<div class="container">
  <div class="row">
    <div class="col-md-4 child">
      <h3><a href="#">Blog Post Title</a></h3>
      <p><small>Posted by <a href="#">Admin</a> on January 1, 2016</
small></p>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nullam eget ornare lacus. Nulla sed vulputate mauris. Nunc nec urna
vel sapien mattis consectetur sit amet eu tellus.</p>
      <p><a href="#">Read More</a></p>
    </div>
    <div class="col-md-4 child">
      <h3><a href="#">Blog Post Title</a></h3>
      <p><small>Posted by <a href="#">Admin</a> on January 1, 2016</
small></p>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nullam eget ornare lacus. Nulla sed vulputate mauris. Nunc nec urna
vel sapien mattis consectetur sit amet eu tellus.</p>
      <p><a href="#">Read More</a></p>
    </div>
    <div class="col-md-4 child">
      <h3><a href="#">Blog Post Title</a></h3>
      <p><small>Posted by <a href="#">Admin</a> on January 1, 2016</
```

```
small></p>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Nullam eget ornare lacus. Nulla sed vulputate mauris. Nunc nec urna
vel sapien mattis consectetur sit amet eu tellus.</p>
      <p><a href="#">Read More</a></p>
    </div>
  </div>
</div>
```

Save your file, do a `harp compile` if you haven't done so for in a while. Then do a harp server to launch the web server and head to `http://localhost:9000` to preview the page. It should look like this:



Great, now we have a decent-looking blog homepage. However, we need to add more posts to fill this out. Let's go ahead and add more `<div>` tag columns inside the same row. Since this is Flexbox, we don't need to start a new `<div>` with a row class for each row of posts. Let's add three more posts in then see what it looks like:



Perfect. Now our homepage is starting to take shape. Continue adding more posts until you have a number that you are happy with. At this point you should have a decent understanding of the Flexbox grid.

Summary

We started by reviewing the basic functionality of the Flexbox module and the terminology that goes along with it. Next, I showed you how to activate the Flexbox grid in Bootstrap by editing the Sass variable and recompiling the source files. Finally we got our hands dirty by learning how to build a blog homepage and feed using the Bootstrap Flexbox grid. In the next lesson, we'll move further into layouts and how you can set up your pages with Bootstrap.

Assessments

1. What is the goal of flexbox layout module?
    i. To shrink to best fill the available space when needed when building responsive
    web applications or websites
    ii. To create a more effective way of designing a layout for a website or web application
    iii. To create a homepage
    iv. All of the above

2. By default in Flexbox, the child boxes will be in a row going?
    i. Left to right
    ii. Right to left
    iii. Top to bottom
    iv. Diagonal

3. What does the following code outputs?

```
.parent {
...
flex-direction: column-reverse;
}
```

    i. Orders the boxes vertically so they were stacked on top of each other in descending order
    ii. Orders the boxes vertically so they were stacked on top of each other in ascending order
    iii. Reverses the order of boxes
    iv. There will be no output

5. What is the difference of output between the following code snippets?

```
.parent {
...
align-items:
flex-start;
}
```

```
.parent {
...
align-items:
flex-end;
}
```

```
.parent {
...
align-items:
center;
}
```

i. The first snippet will align the boxes to the bottom of the browser window, the second snippet will center your columns vertically in the layout and the third will undo the equal height columns and appear like a regular grid

ii. The first snippet will center your columns vertically in the layout, the second snippet will align the boxes to the bottom of the browser window and the third will align the boxes to the bottom of the browser window

iii. The first snippet will undo the equal height columns and appear like a regular grid, the second snippet will align the boxes to the bottom of the browser window and the third will center your columns vertically in the layout

iv. None of the above

5. Can you use both Flexbox and regular grid at the same time in a Bootstrap project?

    i. Yes

    ii. No

6. How to create a `full-width` layout with no horizontal padding?

    i. Use columns instead of rows

    ii. Rename the `.container` class

    iii. Remove all the `<div>` elements

    iv. Simply remove the `<div>` with the .container class on it

3
Working with Layouts

The core of any Bootstrap website is the layout or grid component. Without a grid, a website is pretty much useless. One of the biggest layout challenges we face as web developers nowadays is dealing with a large array of screen resolutions, from desktop to tablets, mobile phones, and even Apple watches. It is not easy to lay out a website and we rely on responsive web design and media queries to take a mobile-first approach. Perhaps the best feature of the Bootstrap layout grid is that it's mobile-first and built using media queries. This takes the hardest part out of constructing a grid and lets us concentrate on the actual design of our projects. Before we start to layout the next part of our blog project, let's review the lay out grid basics in Bootstrap 4.

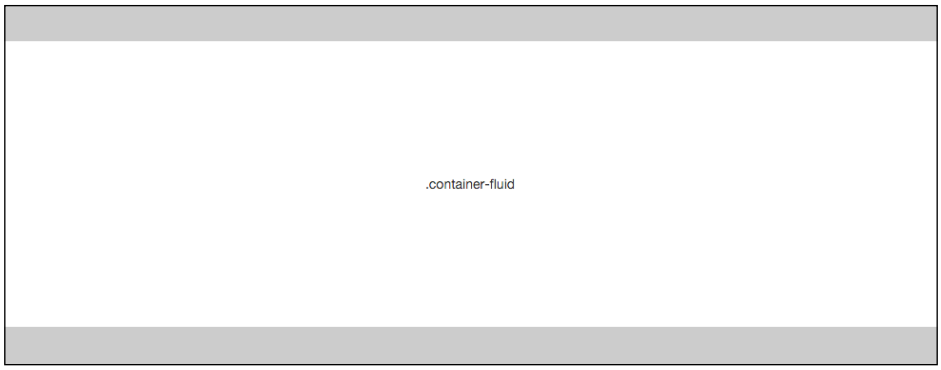
In this lesson, we are going to discuss the following listed topics briefly:

- Working with containers
- Adding columns to your layout
- Creating a simple three-column layout
- Coding the blog home page
- Using responsive utility classes

Working with containers

The base of any Bootstrap layout is a container class. There are two types of containers you can choose to use. The first is `.container-fluid`, which is a full-width box and will stretch the layout to fit the entire width of the browser window.

There is some left and right padding added so the content doesn't bump right up against the browser edge:



The second option is the basic `.container` class, which will have a fixed width based on the size of your device's viewport. There are five different sizes in Bootstrap, with the following width values:

- **Extra small**: <544px
- **Small**: >544px
- **Medium**: >768px
- **Large**: >992px
- **Extra-large**: >1140px



Let's take a look at the markup for both `container` types. I'll start with the basic `.container` class:

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

That's pretty easy. Now let's look at the code for the fluid container:

```
<div class="container-fluid">
  ...
</div>
```

Again, that is straightforward and is all that you need to know about using the container classes in Bootstrap 4.

Creating a layout without a container

Now, in some cases, you may not want to use a container and that is totally fine. An example of this would be if you want a full width layout but you don't want the default left and right padding. You may have an image banner that you want to stretch to the full width of the browser with no padding. In this case, just remove the `<div>` with the `container` class on it.

Using multiple containers on a single page

It is perfectly fine to use multiple containers on a single page template. The containers are CSS classes, so they are reusable. You may want to do this on longer page layouts, perhaps a landing page design, where you have multiple large regions. Another example is using a container to wrap your footer. If you are using a template system like Harp, you'll want to create a footer partial. You can make the footer component more self-contained by giving it its own container. Then you don't have to worry about closing a container `<div>` in the footer that was opened in a page template or even the header. I would recommend using multiple containers to make your designs more modular and easier to manage by multiple users. Let's take a quick look at how you would structure a basic page with multiple containers:

```
<div class="container">
   <!-- header code goes here, harp partial name would be _header.ejs
//-->
</div>

<div class="container">
   <!-- template code goes here, harp file name would be index.ejs //-
->
</div>

<div class="container">
   <!-- footer code goes here, harp partial name would be _footer.ejs
//-->
</div>
```

We have three separate files there and using a container class for each makes each section more modular. You also don't have to worry about opening a `<div>` in one file then closing it in another. This is a good way to avoid orphan closing `</div>` tags.

Inserting rows into your layout

The next step in creating a layout is to insert at least a single row of columns. Each container class can have one or more rows nested inside of it. A row defines a collection of horizontal columns that can be broken up to twelve times. The magic number when it comes to columns in Bootstrap is twelve, and you can sub-divide them any way you like. Before we get into columns though, let's review the markup for rows. First let's look at an example of a container with a single row:

```
<div class="container">
   <div class="row">
      <!-- insert column code here //-->
   </div>
</div>
```

As you can see, this is an easy next step in setting up your layout. Like I mentioned, you can have as many rows within a container as you like. Here's how you would code a five-row layout:

```
<div class="container">
   <div class="row">
      <!-- insert column code here //-->
   </div>
   <div class="row">
      <!-- insert column code here //-->
   </div>
   <div class="row">
      <!-- insert column code here //-->
   </div>
   <div class="row">
      <!-- insert column code here //-->
   </div>
   <div class="row">
      <!-- insert column code here //-->
   </div>
</div>
```

Like the container class, rows are also a CSS class, so they can be reused as many times as you like on a single page template.



You should never include actual contents inside a row `<div>`. All content should be included with column `<div>`s.

Adding columns to your layout

Before we jump into actually adding the columns, let's talk a little bit about the different column classes you have at your disposal with Bootstrap. In Bootstrap 3 there were four different column class widths to choose from: extra small, small, medium, and large. With Bootstrap 4, they have also introduced a new extra large column class. This is likely to allow for extra large monitors, like you would find on an iMac. Let's go over the fine points of each column class in Bootstrap 4.

Extra small

The smallest of the grid classes uses the naming pattern `.col-xs-#`, where `-#` is equal to a number from 1 to 12. Remember, in Bootstrap, your row must be divided into a number of columns that adds up to 12. A couple of examples of this would be `.col-xs-6` or `.col-xs-3`. The extra small column class is for targeting mobile devices that have a `max-width` of 544 pixels.

Small

The small column class uses the syntax pattern `.col-sm-#`, and some examples of that would be `.col-sm-4` or `.col-sm-6`. It is targeted for devices with a resolution greater than 544 pixels but smaller than 720 pixels.

Medium

The medium column class uses a similar naming pattern of `.col-md-#` and some examples would be `.col-md-3` or `.col-md-12`. This column class is for devices greater than 720 pixels and smaller than 940 pixels.

Large

The large column class again uses the naming pattern of `.col-lg-#` and some examples would be `.col-lg-6` or `.col-lg-2`. This column class targets devices that are larger than 940 pixels but smaller than 1140 pixels.

Extra large

The final and new column class is extra-large and the syntax for it is `.col-xl-#` with examples being `.col-xl-1` or `.col-xl-10`. This column class option is for all resolutions greater than or equal to 1140 pixels.

Choosing a column class

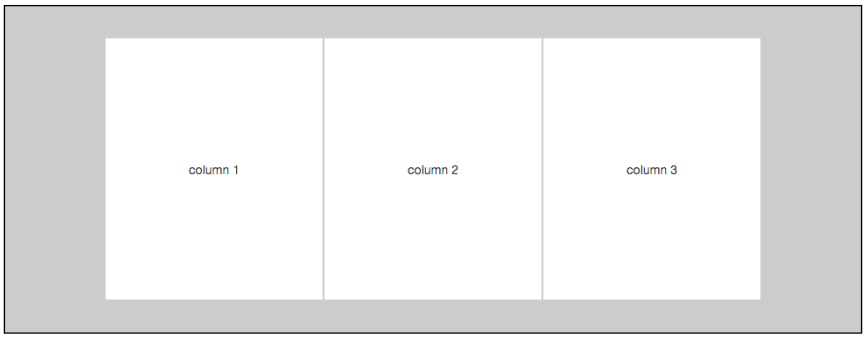
This is a good question. With all of the class options, it can be hard to decide which ones to use. If you are building a mobile app, then you would likely want to stick to the extra small or small column classes. For a tablet, you might want to use medium. If the primary user for your application will be on a desktop computer, then use the large or extra-large classes. But what if you are building a responsive website and you need to target multiple devices? If that is the case, I usually recommend using either the medium or large column classes. Then you can adjust to use larger or smaller classes where necessary if you have a component that needs some extra attention for specific resolutions.

Creating a simple three-column layout

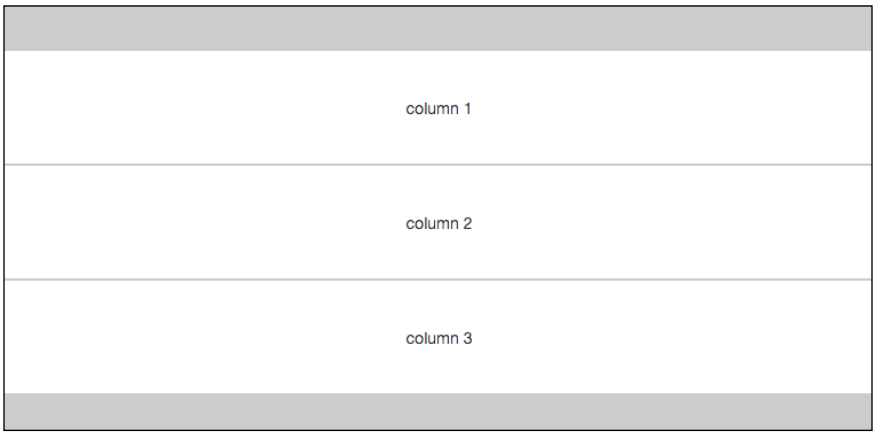
Let's assume that we are building a simple responsive website and we need a three-column layout for our template. Here's what your markup should look like:

```
<div class="container">
   <div class="row">
      <div class="col-md-4">
         <!-- column 1 //-->
      </div>
      <div class="col-md-4">
         <!-- column 2 //-->
      </div>
      <div class="col-md-4">
         <!-- column 3 //-->
      </div>
   </div>
</div>
```

As you can see, I've inserted three `<div>`s inside my row `<div>`, each with a class of `.col-md-4`. For devices that have a resolution of 768 pixels or greater, you'll see a three-column layout like this:



Now, if you were to view this same layout on a device with resolution smaller than 768 pixels, each column's width would change to 100% and the columns would be stacked on top of each other. That variation of the layout for smaller screens would look like this:



That's all well and good, but what if we wanted to have a different layout for the columns on smaller devices that didn't set them all to 100% width? That can be done by mixing column classes.

Mixing column classes for different devices

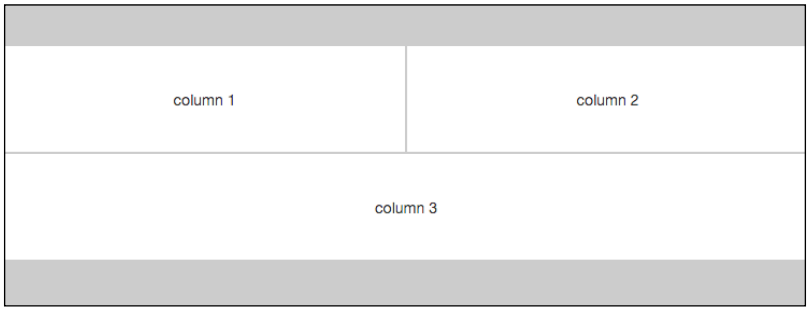
Adding additional classes to each of our column `<div>`s will allow us to target the grid layout for different devices. Let's consider our three-column layout from before, but this time, we want to lay it out like this:

- The first two columns should be 50% of the layout
- The third column should stretch 100% of the layout and be below the first two

To achieve this layout, we'll mix some different column classes. Here's what the markup should look like:

```
<div class="container">
   <div class="row">
      <div class="col-md-4 col-xs-6">
         <!-- column 1 //-->
      </div>
      <div class="col-md-4 col-xs-6">
         <!-- column 2 //-->
      </div>
      <div class="col-md-4 col-xs-12">
         <!-- column 3 //-->
      </div>
   </div>
</div>
```

I've added the `.col-xs-6` class to the first two column `<div>`s. Now, if our device resolution is less than 768 pixels, the first two columns will be set to a width of 50%. For the third column, I've used the `.col-xs-12` class, which will wrap the third column onto a new line and set it to 100% of the width of the layout. The resulting layout will look like this on smaller devices:



This will only apply to devices with a layout of less than 768 pixels. If you were to view this grid, using the latest code, on a larger device, it will still appear as three equal columns laid out horizontally.

What if I want to offset a column?

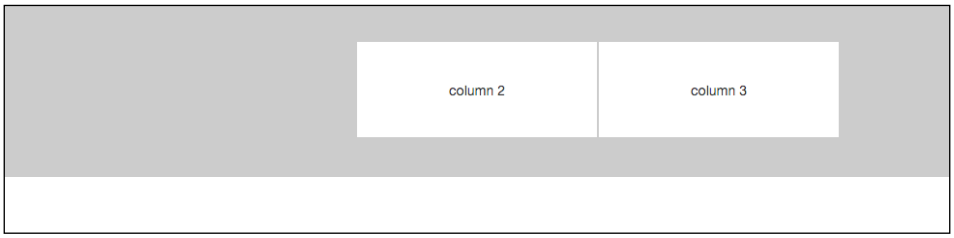
Perhaps your layout requires you to offset some columns and leave some horizontal blank space before your content starts. This can easily be used with Bootstrap's offset grid classes. The naming convention is similar to the regular column classes, but we need to add the offset keyword, like this: `.col-lg-offset-#`. A couple examples of this would be `.col-lg-offset-3` or `.col-md-offset-6`. Let's take our three-column grid from before but remove the first column. However, we want the second and third columns to remain where they are in the layout. This will require us to use the offset grid class. Here's what your markup should look like:

```
<div class="container">
   <div class="row">
      <div class="col-md-4 col-md-offset-4">
         <!-- column 2 //-->
      </div>
      <div class="col-md-4">
         <!-- column 3 //-->
      </div>
   </div>
</div>
```



Note how I removed the first column `<div>`. I also added the class `.col-md-offset-4` to the second column's `<div>`. Once you've done this, your layout should appear like this.

There you go; you've successfully offset your column by removing the first column and then sliding over the other two columns:



That concludes the basics of the Bootstrap grid that you'll need to know for the remainder of this lesson. Now that you have a good understanding of how the grid works, let's move onto coding up our blog home page layout using the grid classes.

Coding the blog home page

Now that you have a good grasp of how to use the Bootstrap 4 grid, we're going to code up our blog home page. This page will include a feed of posts, a sidebar, and a newsletter sign-up form section at the bottom of the page. Let's start by taking the code we wrote in *Lesson 1*, *Setting up Our First Blog Project*, for our `hello world!` template and duplicating the entire directory. Rename the folder `Lesson 3: Working with Layouts` or `Bootstrap Layout`.



Remember, we are using the regular grid moving forward, not the Flexbox grid. Make sure you are using the default build of `bootstrap.min.css`. If you carry out a simple duplication of the second lesson's code then you'll have the right file configuration.

Writing the index.ejs template

Good news! Since we set up our Harp project in the beginning, we can reuse a bunch of that code now for our blog home page. There's no need to make any updates to the JSON files and header or footer partials. The only file we need to make changes is `index.ejs`. Open the file up in a text editor and paste the following code to get started:

```
<div class="container">
  <!-- page title //-->
  <div class="row m-t-3">
    <div class="col-md-12">
      <h1>Blog</h1>
    </div>
  </div>
  <!-- page body //-->
  <div class="row m-t-3">
    <div class="col-md-8">
      <!-- blog posts //-->
    </div>
    <div class="col-md-4">
      <!-- sidebar //-->
    </div>
  </div>
```

```
    <!-- mailing list //-->
    <div class="row m-t-3">
      <div class="col-md-12">
        <!-- form //-->
      </div>
    </div>
  </div>
```

There are a few different things going on here so let me break them all down for you:

- I don't want a full width layout, so I've decided to use the `.container` class to wrap my templates layout.
- I've created three different rows, one for our page title, one for the page content (blog feed and sidebar), and one for the mailing list section.
- There are some classes on the row `<div>`s that we haven't seen before, like `m-t-3`. I'll cover what those do in the next section.
- Since I want my blog to be readable on devices of all sizes, I decided to use the medium-sized column classes.
- The page title column is set to `.col-md-12`, so it will stretch to 100% of the layout width.
- I've divided the second row, which holds most of our page content, into a two-column grid. The first column will take up 2/3 of the layout width with the `col-md-8` class. The second column, our sidebar, will take up 1/3 of the layout width with the `col-md-4`class.
- Finally, the third row will hold the mailing list and it is also using the `col-md-12` class and will stretch to fill the entire width of the layout.

The basic layout of the grid for our blog home page is now complete. However, let's revisit those new CSS classes from our layout that I added to the row `<div>`s.

Using spacing CSS classes

One of the new utilities that has been added in Bootstrap 4 is spacing classes. These are great as they add an easy, modular way to add extra vertical spacing to your layouts without having to write custom CSS classes for each region. Spacing classes can be applied to both the CSS `margin` and `padding` properties. The basic pattern for defining the class is as follows:

```
{property}-{sides}-{size}
```

Let's break down how this works in more detail:

- `property` is equal to either `margin` or `padding`.
- `sides` is equal to the side of a box you want to add either `margin` or `padding` to. This is written using a single letter: `t` for top, `b` for bottom, `l` for left, and `r` for right.
- `size` is equal to the amount of margin or padding you want to add. The scale is 0 to 3. Setting the size value to 0 will actually remove any existing margin or padding on an element.

To better understand this concept, let's construct a few spacer classes. Let's say that we want to add some top margin to a row with a size value of `1`. Our class would look like this:

```
.m-t-1
```

Applied to the actual row, `<div>`, the class would look like this:

```
<div class="row m-t-1">
```

For a second example, let's say we want to add some left padding to a div with a value of `2`. That combination would look like this when combined with a row `<div>`:

```
<div class="row p-l-2">
```

Are you starting to see how easy it is to add some spacing around your layout and components?



Spacing classes can be used on any type of element, not just the Bootstrap grid.

Now that you understand how these classes work, let's take a look at our blog home page template again. In that case, our `<div>`s looks like this:

```
<div class="row m-t-3">
```

On three sections of the template, I've decided to use these classes and they are all top margin with a size value of three. It's a good idea to try and keep these consistent as it will result in a visually appealing layout when you are done. It also makes it a little easier to do the math when you are setting up your page. Now that we've gone over the entire home page layout, we need to test it.

Testing out the blog home page layout

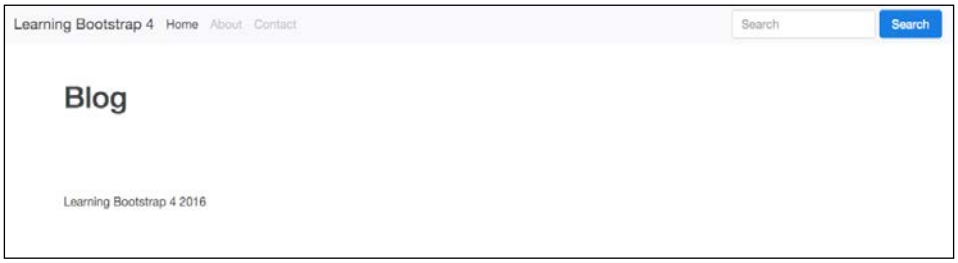
Let's test it out in the browser to make sure it's looking the way we want. Before we can do that we'll need to compile our code with Harp. Open the Terminal back up and navigate to the project directory for this lesson's code that we created. Run the `harp compile` command, here it is again in case you forgot:

```
$ harp compile
```

That should run without any errors; then, we can start-up the web server to view our page. Here's the command again to run the web server:

```
$ harp server
```

Now that the server has launched, head to a web browser and enter `http://localhost:9000` in the URL bar to bring up the blog home page. Here's what your page should look like:



Uh oh, that doesn't look quite right. You can see the page title but we can't see any of our columns. Oh yeah! We need to fill in some content so the columns are revealed. Let's add in some dummy text for demo purposes. In later lesson, I'll get into coding the actual components we want to see on this page. This lesson is just about setting up our layout.

Adding some content

Head back to `index.ejs` in your text editor and let's add some dummy text. Go to the first column of the main content area first and enter something like this:

```
<div class="col-md-8">
      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
```

```
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
</div>
```

If you're looking for a quick way to get filler text in HTML format, visit `http://html-ipsum.com/`.

Next, go to the sidebar column `<div>` and add the same paragraph of text, like so:

```
<div class="col-md-4">

      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
</div>
```

Finally, drop down to the mailing list `<div>` and add the same paragraph of content again. It should look like this:

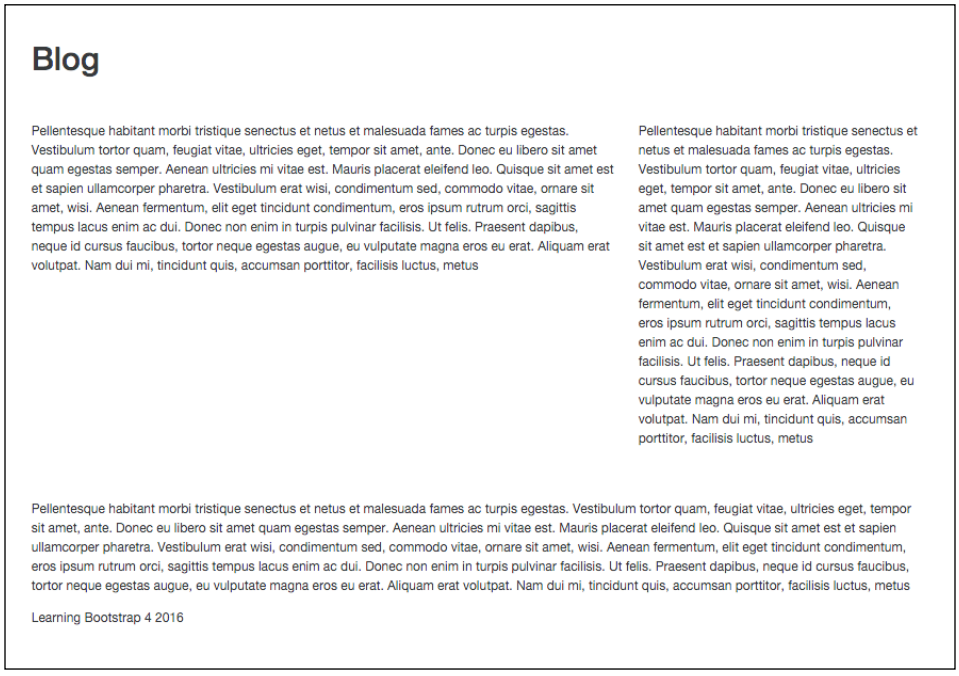
```
<div class="col-md-12">
      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
</div>
```

Now that we've added some actual content to our page body, let's recompile the project and launch the web server again:



With Harp, you don't actually have to recompile after every little change you make. You can also make changes to your files while the server is running and they will be picked up by the browser. It's a good habit to compile regularly in case you run into an error on compile. This will make it easier to troubleshoot potential problems.

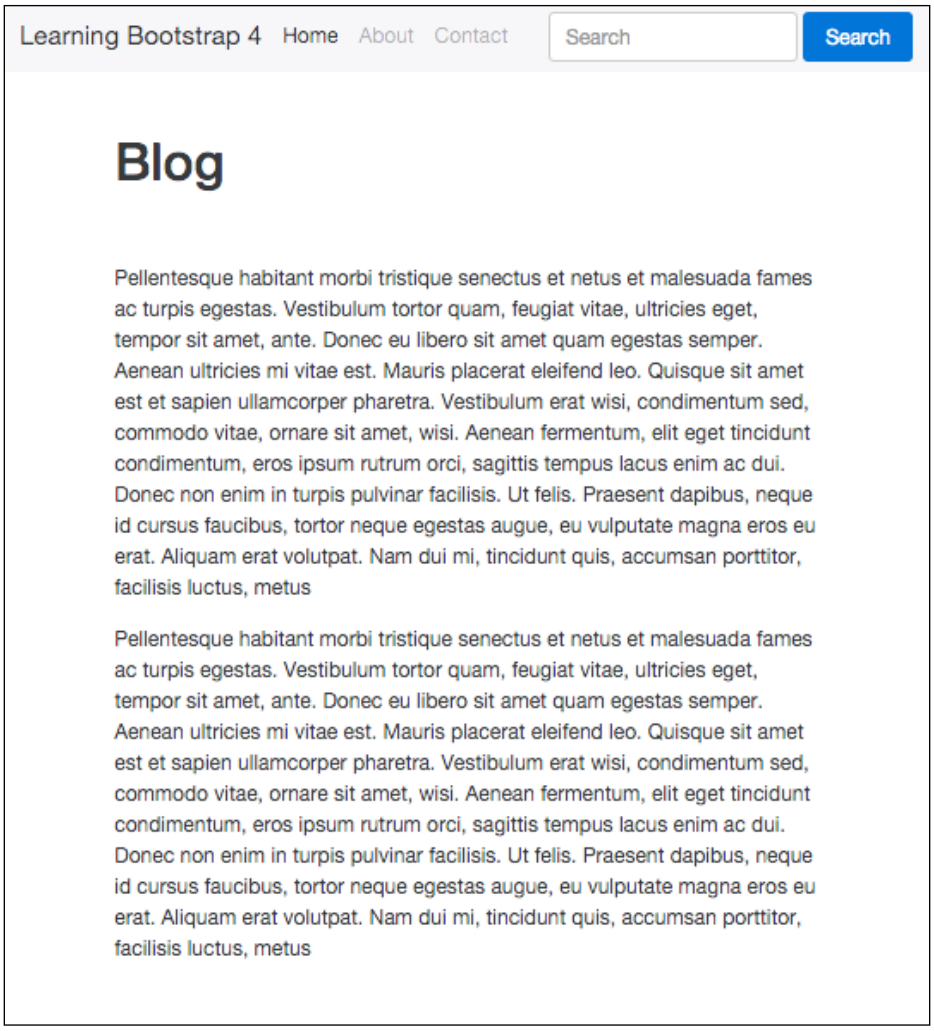
Once the server is up and running, return to your browser and refresh the page. Now your layout should look like this:



Yay! We can now see our columns and the dummy text that we just added. The page may not be much to look at right now, but what's important is to verify that your columns are laid out correctly.

What about mobile devices?

We need to consider what will happen to our layout on mobile devices and smaller screen resolutions. I used the medium grid layout class, so any device that is smaller than 720 pixels will have an adjusted layout. Resize your browser window, making it smaller to trigger the media query, and you'll see that all of the columns will be resized to 100% width of the container. Here's what it looks like:



I'm going to keep our blog layout pretty minimal so I'm okay with this layout. In this format, the sidebar will slide in under the main blog feed of posts. I'm actually not that crazy about this design, so I'm just going to hide the sidebar altogether when you view the blog on a smaller device.

Using responsive utility classes

Responsive utility classes will allow you to selectively hide `<div>` tags or components based on the screen resolution size. This is great for creating a mobile-first web application, because in many cases you'll want to hide some components

that don't work well on a phone or tablet. Mobile application design generally means a simpler, more minimal experience some using responsive utility classes will allow you to achieve this. Open up `index.ejs` in a text editor and go down to the sidebar `<div>`, then add the `.hidden-md-down` class to your code:

```
<div class="col-md-4 hidden-md-down">
```

Adding this class will hide the `<div>` from the browser when your screen resolution is smaller than 720 pixels. Make sure your column class, in this case `-md`, matches the hidden class. Now, if you shrink your web browser down again, you'll notice that the sidebar `<div>` will disappear.

There are a number of responsive utility classes you can use in your projects. There is a `-down` version for each of the column class names. You can also use a `-up` version if you wish to hide something when viewing at a larger resolution. Some examples of that class are `.hidden-lg-up` or `.hidden-xl-up`. To learn more about responsive utility classes, check out the page in the Bootstrap docs at `http://v4-alpha.getbootstrap.com/layout/responsive-utilities/`.

That completes the layout of the grid for our blog home page. Before we move onto learning about content classes in Bootstrap 4, let's set up the layout grid for the other pages we'll be building for the blog.

Coding the additional blog project page grids

Before we create new templates for our contact and single blog post pages, we need to update some of the Harp project files. Let's update those files, then we'll move onto the page templates.

Updating _data.json for our new pages

Remember a couple lesson back we learned how to code the `_data.json` file and we created a variable for the page title of each of our templates? We need to update this file for our two new pages by providing the `pageTitle` variable value for each of them. Open up `_data.json` in a text editor; you can find the file in the root of your blog project directory. Once you have the file open, insert the following code. The entire file should read as follows:

```
{
   "index": {
      "pageTitle": "Home"
   },
   "contact": {
      "pageTitle": "Contact"
```

```
    },
    "blog-post": {
       "pageTitle": "Blog Post"
    }
}
```

Originally, we only included the index file. I've added two more entries, one for the contact page and one for the blog-post page. I've entered a value for each page's `pageTitle` variable. It's as simple as that. Save the JSON file and then you can close it.

Creating the new page templates

Now that `_data.json` has been updated, we need to create the actual page template EJS files like we did with index. In your text editor, create two new files and save them as `contact.ejs` and `blog-post.ejs`. For now, just leave them blank and we'll start to fill them in the next steps. The templates are now set up and ready to be coded. For now, both of our new pages will use the same `_layout.ejs` file as the index file, so there is no need to create any more layouts. Let's start by coding the contact page template.

Coding the contact page template

Open up the `contact.ejs` file you just created in your text editor. Let's start the template by setting up our page title. Enter the following code into the file:

```
<div class="container">
  <!-- page title //-->
  <div class="row m-t-3">
    <div class="col-lg-12">
      <h1>Contact</h1>
    </div>
  </div>
</div>
```

Let's breakdown what's happening here in the code:

- I've opened up the file with a `<div>` with a `.container` class on it.
- Next I added `.row <div>` and I've added the same `m-t-3` spacing classes so it matches the blog home page.
- I've added a column `<div>` with a class of `.col-md-12`. Since this is our page title, we want it to stretch to the width of our layout.
- Finally, I've added an `<h1>` tag with our contact page title.

Adding the contact page body

Next let's insert our grid layout for the body of the contact page. Following the page title code, insert the following grid code:

```
<!-- page body //-->
  <div class="row m-t-3">
    <div class="col-md-12">
      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
    </div>
  </div>
```

Let's review the code for the page body:

- I've opened up another row `<div>` for the page body. It also has the same `m-t-3` spacing class on it for consistent vertical spacing.
- I've used the `col-md-12` column class again because the contact page layout will fill the whole width of our container.
- I've added some filler text for now so that we can verify that the page is laid out properly.

Before we finish, let's add one more row for our mailing list section. I'd like this to be available on every page of our blog. The grid code for this section will be identical to the markup we did for the page body. Here's what it looks like, for reference:

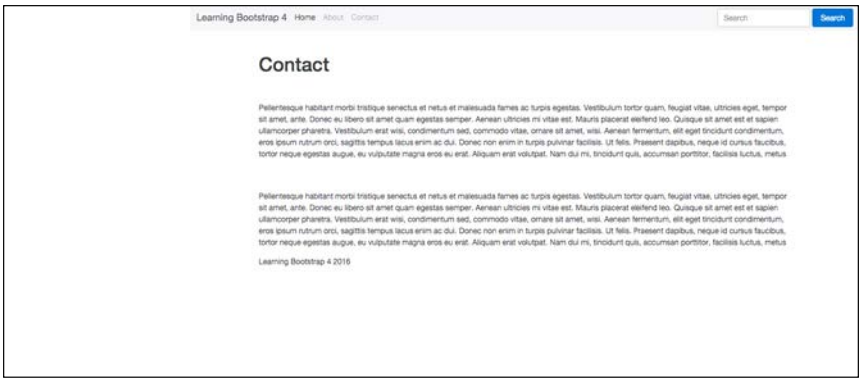
```
<!-- mailing list //-->
<div class="row m-t-3">
  <div class="col-md-12">
   <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
```

```
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
  </div>
</div>
```

Since this code is identical to the page body, I won't bother breaking it down again. Our layout for the contact page is now complete. Make sure you save the file and let's test it before we move onto the blog post page.

Open your Terminal back up and navigate to the root directory of the blog project. Once there, run the `harp compile` command and then the Harp server command to launch the local web server. Open your web browser and enter the following URL to preview your page: `http://localhost:9000/contact.html`.

Your contact page should load up and you should see a page title and two rows of filler text. Here's what it should look like:



Our contact page grid is now complete. Before we move onto creating the blog post template, let's take a look at all the code for the contact template:

```
<div class="container">
  <!-- page title //-->
  <div class="row m-t-3">
    <div class="col-md-12">
      <h1>Contact</h1>
    </div>
  </div>
  <!-- page body //-->
  <div class="row m-t-3">
    <div class="col-md-12">
```

```
      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
    </div>
  </div>
  <!-- mailing list //-->
  <div class="row m-t-3">
    <div class="col-md-12">
      <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
    </div>
  </div>
</div>
```

Coding the blog post template

Head back to your text editor and open the file `blog-post.ejs` that you previously created. Like our contact page template, let's start by first setting up the page title section of code. Enter the following code into the blog post template file:

```
<div class="container">
  <!-- page title //-->
  <div class="row m-t-2 m-b-2">
    <div class="col-lg-12">
      <h1>Post Title</h1>
    </div>
  </div>
</div>
```

As you can see, this code is almost identical to the contact page. There are two small differences that I will point out for you:

- I've changed up the spacing classes on the row `<div>`. In a future lesson, we are going to add some different components around the page title, so I've altered the vertical spacing to allow for them. I'm using the same margin top spacer but I've only set it to a value of `2`. I've added a second margin bottom spacer with a value of `2` with the `.m-b-2` class. Switching the `-t` to `a -b` will apply a bottom margin instead of top margin.
- I've changed the page title to `Post Title` in the `<h1>` tag.

Adding the blog post feature

The body of our blog post will have some different elements compared to the blog home template. After the page title, I'm going to insert a feature section that will be used for an image or carousel in a future lesson. For now, let's just lay in the grid column and some filler text for testing purposes. Enter the following code after the page title section:

```
<!-- feature //-->
<div class="row">
  <div class="col-md-12">
    <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
  </div>
</div>
```

This is a very simple section. Notice the `row <div>` doesn't have a spacer class on it, since we added the bottom margin to the page title section. I've inserted a `full-width col-md-12` column class so the feature can stretch to the width of the layout.

Adding the blog post body

Now that we've added the blog post feature section, let's add the actual body part of the template. This section will use the same layout as our blog home page. It will be a

two-column layout, the first being 2/3 wide, and the sidebar being 1/3 of the layout. Insert the following code after the feature section:

```
<!-- page body //-->
<div class="row m-t-2">
  <div class="col-md-8">
     <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
   </div>
   <!-- sidebar //-->
   <div class="col-md-4 hidden-md-down">
    <p>sidebar</p>
    <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
  </div>
</div>
```

Let's break down what's happening here in the code:

- The row `<div>` has a `m-t-2` spacer class added on to provide some vertical spacing
- I'm using the same `col-md-8` and `col-md-4` column classes to set up the layout
- I've also used the `hidden-md-4` class on the sidebar `<div>` so that this section will not be visible on smaller resolution devices
- Finally, I added some temporary filler text for testing purposes

Converting the mailing list section to a partial

As I mentioned earlier, I would like the mailing list section to appear on all pages of the blog. Since this is the case, it would make more sense to make this chunk of code a partial that can be included in each template. It saves us having to reinsert this snippet in every one of our page templates.

From your text editor, create a new file called `_mailing-list.ejs` and save it to the `partial` directory in the root of your blog project. Once the file is saved, insert the following code into it:

```
<div class="row m-t-3">
  <div class="col-md-12">
    <p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante. Donec eu libero sit amet
quam egestas semper. Aenean ultricies mi vitae est. Mauris placerat
eleifend leo. Quisque sit amet est et sapien ullamcorper pharetra.
Vestibulum erat wisi, condimentum sed, commodo vitae, ornare sit amet,
wisi. Aenean fermentum, elit eget tincidunt condimentum, eros ipsum
rutrum orci, sagittis tempus lacus enim ac dui. Donec non enim in
turpis pulvinar facilisis. Ut felis. Praesent dapibus, neque id cursus
faucibus, tortor neque egestas augue, eu vulputate magna eros eu erat.
Aliquam erat volutpat. Nam dui mi, tincidunt quis, accumsan porttitor,
facilisis luctus, metus</p>
  </div>
</div>
```

Now go back to the blog post template file and insert the following line of code where the mailing list section should appear:

```
<%- partial("partial/_mailing-list") %>
```

Remember to do the same thing for the index and contact template. Delete the hardcoded mailing list and replace it with the preceding partial line.

That concludes the setup of the blog post template. Let's test it out before we move onto the next lesson, to make sure the new mailing list partial is working properly. Return to the Terminal and compile your project from the root directory. Run the Harp server command, then visit the following URL: `http://localhost:9000/blog-post.html`.

If all went as planned, your blog post page should look like this:



Make sure you don't forget to test the index and contact page templates in your browser to make sure the mailing list partial is working properly. That concludes the design layout for the blog post template. All of our templates are now ready to go, but before we move onto the next lesson on content components, let's review what we've learned.

Summary

This lesson has been a detailed explanation of the Bootstrap layout grid, how to use it, and how to build a sample project. We started out by learning the basics of the Bootstrap container, container-fluid, and row classes. Next, we moved onto learning the differences between all the Bootstrap column classes. Following the columns, we covered some more advanced topics, like offsetting columns, spacing, and responsive utilities. Once you had a solid understanding of the grid, we coded up the remaining page layouts that we'll need for the rest of the course. Now that we have everything set up, we'll start to drop some real content into the blog using Bootstrap content classes.

Assessments

1. What are the two types of containers you can use for a Bootstrap layout?
    i. `.container-fluid`
    ii. `.container`
    iii. Both i & ii
    iv. None of the above
2. It is perfectly fine to use multiple containers on a single page template.
    i. True
    ii. False
3. `.col-xl-1` or `.col-xl-10`: this column class option is for all resolutions greater than or equal to _____ pixels.
    i. 940
    ii. 1140
    iii. 720
    iv. 544
4. What is the advantage of `spacing` classes in Bootstrap 4?
    i. Adds spaces to rows and columns
    ii. They add an easy, modular way to add extra vertical spacing to your layouts without having to write custom CSS classes for each region
    iii. Can be applied to both the CSS `margin` and `padding` properties
    iv. Both ii & iii
5. Which of the following is the use of utility classes?
    i. Uses the medium grid layout class, so any device that is smaller than *720 pixels* will have an adjusted layout
    ii. Multiple containers to make your designs more modular and easier to manage by multiple users
    iii. order the boxes vertically so they were stacked on top of each other in
    ascending order
    iv. Utility classes will allow you to selectively hide `<div>` tags or components based on
    the screen resolution size

4
Working with Content

Content components in Bootstrap 4 are reserved for the most commonly used HTML elements such as images, tables, typography, and more. In this lesson, I'll teach you how to use all the building blocks of HTML that you'll need to build any type of website or web application. We'll start with a quick overview of each component and then we'll build it into our blog project. Bootstrap 4 comes with a brand new CSS reset called **Reboot**. It builds on top of Normalize.css to make your site look even better out of the box. Before we jump in, let's review some Reboot basics when dealing with content components in Bootstrap 4.

Reboot defaults and basics

Let's start this lesson by reviewing the basics of Reboot when using content components in Bootstrap. One of the main changes for content components in Bootstrap 4 is the switch from em to rem units of measure. rem is short for root em and is a little bit different from a regular em unit. em is a relative unit to the font-size of the parent element it is contained within. This can cause a compounding issue in your code that can be difficult to deal with when you have a highly nested set of selectors. The rem unit is not relative to its parent, it is relative to the root or HTML element. This makes it much easier to determine the actual size of text or other content components that will appear on the screen.

The `box-sizing` property is globally set to `border-box` on every element. This is good because it ensures that the declared width of an element doesn't exceed its size due to excess margins or padding.

The base `font-size` for Bootstrap 4 is `16px` and it is declared on the `html` element. On the `body` element, the `font-size` is set to `1rem` for easy responsive type-scaling when using media queries.

There are also global `font-family` and `line-height` values set on the body tag for consistency through all components. By default, the `background-color` is set to `#fff` or white on the `body` selector.

Headings and paragraphs

There are no major changes to the styles for headings and paragraphs in Bootstrap 4. All heading elements have been reset to have their `top-margin` removed. Headings have a `margin-bottom` value of `0.5rem`, while paragraphs have a `margin-bottom` value of `1rem`.

Lists

The list component comes in three variations: `<ul>`, `<ol>`, and `<dl>`. Each list type has had its `top-margin` removed and has a `bottom-margin` of `1rem`. If you are nesting lists inside one another, there is no `bottom-margin` at all.

Preformatted text

This typography style is used for displaying blocks of code on a website using the `<pre>` tag. Like the previous components, its `top-margin` has been removed and it has a `bottom-margin` of `1rem`.

Tables

The table component has been adjusted slightly to ensure consistent text alignment in all cells. The styles for the `<caption>` tag have also been adjusted a bit for better legibility.

Forms

The form component is much simpler in Bootstrap 4. Much of the default styling has been removed to make the component easier to use and customize. Here are some of the highlights you should be aware of:

- Most of the styles have been removed on the `<fieldset>` tag. The borders, padding, and margin are no longer there.
- The `<legend>` tag has been simplified and is much more minimal in look now.
- The `<label>` tag is now set to `display: inline-block` to allow margins to be added.

- Default margins have been removed from the following tags: `<input>`, `<select>`, `<textarea>`, and `<button>`.
- `<textarea>` can now only be resized vertically. They can't be resized horizontally, which often breaks page layouts.

That covers the key elements you need to be aware of with Reboot. If you're interested in learning more, please check out the docs at `http://v4-alpha.getbootstrap.com/content/reboot/`.

Now that we've reviewed the Reboot CSS reset, it's time to actually start covering the content components and adding them to our blog project.



Content classes in Bootstrap 4 are not that different from version 3. If you are fluent in Bootstrap 3, you may want to jump ahead to the next lesson at this point.

Learning to use typography

In Bootstrap 4, there are no major changes with the core typographic HTML tags. Header tags and their supporting CSS classes still work as they always have. However, there are some new utility classes you can use with some type tags to provide further variations for things like headers and titles. Later on in the course we'll be using a number of type tags and styles in our blog project. A couple of quick examples would be header tags for page and post titles, and lists for a number of different components. Let's start by reviewing the new display heading classes in Bootstrap 4.

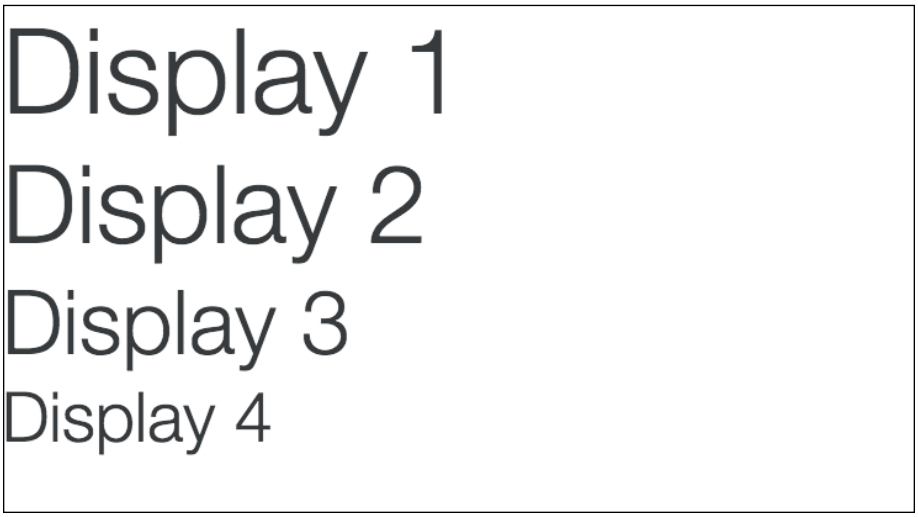
Using display headings

Regular header tags work great in the flow of a page and are key for setting up the hierarchy of an article. For a landing page or other display-type templates, you may require additional header styles. This is where you can use the new display heading classes to create slightly larger titles with some different styling. There are four different levels of display headings you can use and the markups to render them are as follows:

```
<h1 class="display-1">Display 1</h1>
<h1 class="display-2">Display 2</h1>
<h1 class="display-3">Display 3</h1>
<h1 class="display-4">Display 4</h1>
```

Keep in mind you can apply these classes to any header tag you like. `display-1` will be the largest and the headers will shrink as you increase their size. For example,

`display-4` would be the smallest of the options. Here's what the headers will look like when rendered in the browser:



Customizing headings

You may want to add some additional context to your headers and you can easily do this with some included Bootstrap 4 utility classes. By using a contextual text class, you can tag on a description to a heading like this:

```
<h3>
  This is the main title
  <small class="text-muted">this is a description</small>
</h3>
```

As you can see, I've added a class of `text-muted` to a `<small>` tag that is nested within my `header` tag. This will style the descriptive part of the text a bit differently, which creates a nice looking effect:

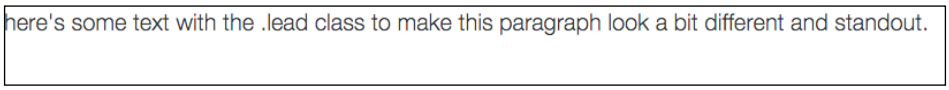


Using the lead class

Another utility text class that has been added to Bootstrap 4 is the `lead` class. This class is used if you want to make a paragraph of text stand out. It will increase the font size by 25% and set the font-weight of the text to light or 300. It's easy to add, as the following code shows:

```
<p class="lead">
here's some text with the .lead class to make this paragraph look a
bit different and standout.
</p>
```

The output of the preceding code will look like this:



As you can see, this gives the text a unique look. This would be good to use as the first paragraph in a blog post or perhaps to call out some text at the top of a landing page.

Working with lists

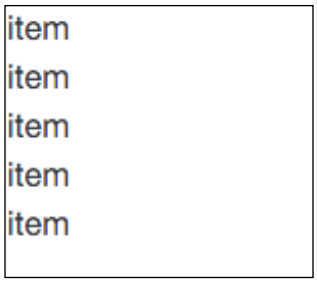
Bootstrap 4 comes with a number of list options out of the box. These CSS classes can be applied to the `<ul>`, `<ol>`, or `<dl>` tags to generate some styling. Let's start with the unstyled list.

Coding an unstyled list

In some cases, you may want to remove the default bullets or numbers that come with ordered or unordered lists. This can be useful when creating a navigation, or perhaps you just want to create a list of items without bullet points. You can do this by using the `list-unstyled` class on the wrapping list tag. Here's an example of a basic unstyled, unordered list:

```
<ul class="list-unstyled">
  <li>item</li>
  <li>item</li>
  <li>item</li>
  <li>item</li>
  <li>item</li>
</ul>
```

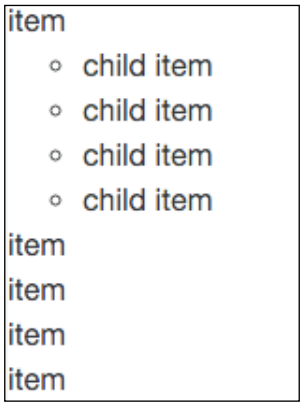
This will produce a list with no bullet points that will look like this:



We can also nest additional lists inside if we want to create multi-level, indented lists. However, keep in mind that the `list-unstyled` class will only work on the first level of your list. Any nested additional lists will have their bullets or numbers. The code for this variation would look something like this:

```
<ul class="list-unstyled">
  <li>item
    <ul>
      <li>child item</li>
      <li>child item</li>
      <li>child item</li>
      <li>child item</li>
     </ul>
  </li>
  <li>item</li>
  <li>item</li>
  <li>item</li>
  <li>item</li>
</ul>
```

The preceding variation will look like the following output:



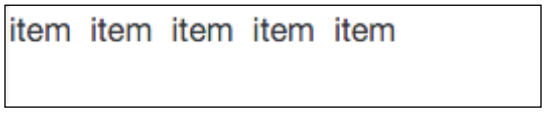
Now, if we check out this code sample in a browser, you'll notice you will see the bullet points for the child list that is nested within the parent.

Creating inline lists

The unstyled list is probably the one you will use the most. The next most useful class is `list-inline`, which will line up each `<li>` in a horizontal line. This is very useful for creating navigations or sub-navigations in a website or application. The code for this list is almost the same as the last, but we change the class name to `list-inline`. We also need to add a class of `list-inline-item` to each `<li>` tag. This is a new change for Bootstrap 4, so make sure you don't miss it in the following code:

```
<ul class="list-inline">
  <li class="list-inline-item">item</li>
  <li class="list-inline-item">item</li>
  <li class="list-inline-item">item</li>
  <li class="list-inline-item">item</li>
  <li class="list-inline-item">item</li>
</ul>
```

As I mentioned, the code is similar to the unstyled list, with a few changes. Here's what it will look like when rendered in the browser:



I think you can see how this would be a lightweight way to set up a horizontal navigation for your project. Let's move onto the last list type, which is a description list.

Using description lists

A description list allows you to create a horizontal display for terms and descriptions. Let's take a look at a basic list's code and then break it down:

```
<dl class="dl-horizontal">
  <dt class="col-sm-3">term 1</dt>
  <dd class="col-sm-9">this is a description</dd>

  <dt class="col-sm-3">term 2</dt>
  <dd class="col-sm-9">this is a different description</dd>

  <dt class="col-sm-3 text-truncate">this is a really long term name</
```

```
dt>
  <dd class="col-sm-9">this is one last description</dd>
</dl>
```

There are a few things going on here that you need to be aware of, so let me break them all down for you:

- First you start a description list using the `<dl>` tag. It requires a class of `dl-horizontal` to trigger the list component styles.
- Each row is made up of a `<dt>` and `<dd>` tag. `<dt>` stands for term, while `<dd>` stands for description. Each tag should take a column class and is flexible, depending on how you want to lay out your list.
- On the third row, you'll notice a class called `text-truncate`. This class will truncate really long terms or text so they don't run outside the width of the column. This is a good technique to use for long chunks of text.

Now that I've explained all the code for the description list, let's see what this sample should look like in the browser:

term 1	this is a description
term 2	this is a different description
this is a really l...	this is one last description

That completes the typography styles that you need to know about in Bootstrap 4. Next, let me teach you what you can do with images in Bootstrap.

How to style images

Bootstrap allows you to do a few useful things with images through the use of CSS classes. These things include: making images responsive, converting images to shapes, and aligning images. In the next section, I'll show you how to apply all these techniques to your images.

Making images responsive

Bootstrap 4 comes with a new responsive image class that is super-handy when developing websites or web-based applications. When applying the class `img-fluid` to an `<img>` tag, it will automatically set the `max-width` of the image to `100%` and the

`height` to `auto`. The result will be an image that scales with the size of the device viewport. Here's what the code looks like:

```
<img src="myimage.jpg" class="fluid-image" alt="Responsive Image">
```

It's as easy as adding that class to the image to trigger the responsive controls. A word of advice: I would recommend making your images a little bit bigger than the maximum size you think you will need. That way, the image will look good on all screen sizes.

Using image shapes

Bootstrap allows you to apply three different shape styles to images:

- `img-rounded`: It will add round corners to your image
- `img-circle`: It will crop your image into a circle
- `img-thumbnail`: It will add round corners and a border to make the image look like a thumbnail

As with the responsive images, all you need to do is add a single CSS class to the `<img>` tag to apply these styles. The reason you would want to use these classes is to avoid having to actually create these variations in an app such as Photoshop. It's much easier to apply this simple image formatting using code. Here's the code for each variation:

```
<img src="myimage.jpg" class="img-rounded" alt="Image Rounded">
<img src="myimage.jpg" class="img-circle" alt="Image Circle">
<img src="myimage.jpg" class="img-thumbnail" alt="Image Thumbnail">
```

Once you've coded that up, it should look like this in the browser:



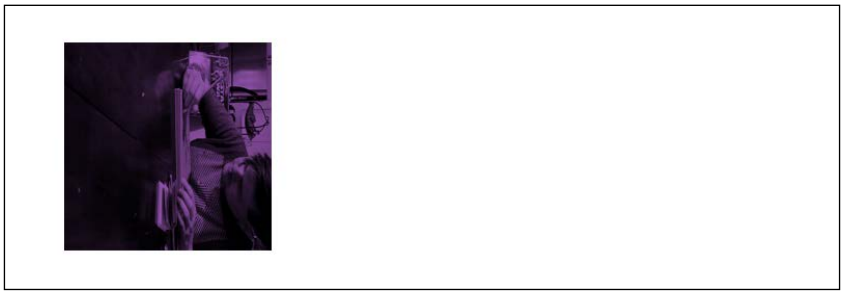


I'm using one of my own images here; you'll need to swap in an image in your code.

Aligning images with CSS

The final Bootstrap classes you can apply to images are the alignment classes. They will allow you to align your image to the left, right, or center of your layout. Like the previous examples, you only need to add a single CSS class to the `<img>` tag to apply the alignment you want. With left and right alignment, you can also provide a column size within the class name. The best policy would be to use the same size as the column the image is contained within. Therefore, if your image is displayed in a column with a class of `col-xs-4`, then use the `-xs` unit in the alignment class name. Here's what the left and right alignment code looks like using the extra small size:

```
<img src="myimage.jpg" class="pull-xs-left" alt="Left Aligned Image">
```



```
<img src="myimage.jpg" class="pull-xs-right" alt="Right Aligned
Image">
```



The final image alignment class you can use is to center the image in the layout. The class name for this is a little bit different, as you can see here:

```
<img src="myimage.jpg" class="center-block" alt="Center Aligned
Image">
```



That concludes the section on image classes that you can use in your Bootstrap layouts. Next we will look at writing and rendering tables using Bootstrap 4.

Coding tables

Tables in Bootstrap 4 are largely unchanged from the previous version of the framework. However, there are a few new things, like inverse color table options and responsive tables. Let's start with the basics and we will build in the new features as we go.

Setting up the basic table

The basic table structure in Bootstrap takes advantage of almost all the available HTML table tags. The header is wrapped in `<thead>` and the body `<tbody>` tags. This will allow additional styling as we get into the inverse table layout. For now, let's see how we put together a basic table in Bootstrap:

```
<table class="table">
<thead>
  <tr>
    <th>first name</th>
    <th>last name</th>
    <th>twitter</th>
  </tr>
</thead>
<tbody>
  <tr>
    <td>john</td>
    <td>smtih</td>
    <td>@jsmtih</td>
```

```
    </tr>
    <tr>
      <td>steve</td>
      <td>stevens</td>
      <td>@stevens</td>
    </tr>
    <tr>
      <td>mike</td>
      <td>michaels</td>
      <td>@mandm</td>
    </tr>
  </tbody>
  </table>
```

As you can see, the syntax is fairly straightforward. The only class being applied is the root `table` class on the `<table>` tag. This needs to be applied to any table variation you are using in Bootstrap. This will produce a table that looks like the following in the browser:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

Inversing a table

Let me quickly show you one of the new table classes in Bootstrap 4. If we add the class `table-inverse` to the `<table>` tag, the table colors will flip to be a dark background with light text. Here's the code you need to change:

```
<table class="table table-inverse">
  ...
</table>
```

This slight variation in code will produce a table that looks like this:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

That's a pretty handy trick to know if you need to get a quick variation of the basic table styles going.

Inversing the table header

Perhaps you don't want to inverse the entire table? If that is the case, you can use the `thead-inverse` class on the `<thead>` tag to only inverse that row:

```
<table class="table">
<thead class="thead-inverse">
  ...
</thead>
  ...
</table>
```

If this variation is applied, then your table will look like this:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

If you're looking for a more subtle design for your project, this approach may be more appealing to you.

Adding striped rows

Although not new to Bootstrap 4, the `table-striped` class is one that I use all the time. Applying this class to the `<table>` tag will add zebra striping to your table, starting with the first row in the body and applying a light grey background color on all the odd numbered rows:

```
<table class="table table-striped">
```

Using this class will produce a table that looks like this:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

Now our table is starting to come together. With a few classes, we can get an attractive-looking layout. Let's see what else we can do with tables.

Adding borders to a table

Another style that is regularly used is to add borders to your table. This can be done in the same way as stripes. Just change or add another class to the `<table>` tab called `table-bordered`. For this example, I'll remove the stripes and add the borders:

```
<table class="table table-bordered">
```

Now that we've added the borders and taken away the stripes, our table should look like this:

```
<table class="table table-bordered table-striped">
```

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

It's important to know that you can combine the table classes and use more than one. What if you wanted a table with stripes and borders? You can do that easily, by including both of the corresponding classes.

Adding a hover state to rows

It's possible and easy to add a hover state to each of your table rows. To do so, you just need to add the `table-hover` class to the `<table>` tag. When used, if you hover over a row in the table, its background color will change to indicate a state change:

```
<table class="table table-hover">
```

Here I've removed the other table classes to show you the basic hover table option. When viewed in the browser, the table should look like the following when a row is hovered over with the mouse:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

In some cases you may require a table with smaller text and compressed height. This can be done by adding the `table-sm` class to the `<table>` tag. This will make the look of the table more compact when viewing it:

```
<table class="table table-sm">
```

If you choose to use this class, your table should look like this:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm

Creating smaller tables

That concludes the core table variations that you can apply through a simple CSS class. Before we move on, there are a couple more important points on tables that we should go over.

Color-coating table rows

In some cases, you may want to color the background of a table row in a different color. This can easily be achieved through the use of some included contextual classes. There are five different color variations you can choose from:

- `table-active`: It is the hover color, light grey by default
- `table-success`: It is green for a positive action
- `table-info`: It is blue for an informational highlight
- `table-warning`: It is yellow to call out something that needs attention
- `table-danger`: It is red for a negative or error action

The preceding classes can be applied to either a `<tr>` or `<td>` tag. If I apply all of these color variations to a single table, they look like this:

first name	last name	twitter
john	smtih	@jsmtih
steve	stevens	@stevens
mike	michaels	@mandm
steve	stevens	@stevens
mike	michaels	@mandm

As you can see, these classes can be useful for validation or just highlighting a particular row that needs to stand out more.

Making tables responsive

Adding responsiveness to tables has never been very easy to do with CSS. Thankfully, Bootstrap 4 comes with some support built right in that you can easily take advantage of. To make a table responsive, you simply need to wrap a `<div>` around your `<table>` that has a class of `table-responsive` on it:

```
<div class="table-responsive">
  <table class="table">
    ...
  </table>
</div>
```

If you view the table on a viewport that is smaller than 768px, then the table cells will scroll horizontally, so they can all be viewed. If the viewport is larger, you will see no difference in the table compared to a regular one.

Summary

With tables finished off, that brings this lesson to a close. I hope this has been a good introduction to content components in Bootstrap, as well as a good review of what's new for these types of components in Bootstrap 4. To review, we learned about: Reboot, typography, images, and tables. In the next lesson, we'll start to jump into some more complicated components and build them into our blog project.

Assessments

1. What allowance does the description list provide the user?
    i. A description list allows the user to create a horizontal display for terms and descriptions
    ii. Allowance for creating navigations or sub-navigations in a website or application
    iii. When creating a navigation, or perhaps you just want to create a list of items without bullet points
    iv. None of the above
2. What are the changes that takes place in an image when the img-fluid class is applied to `<img>` tag?
    i. It will automatically set the width to 100%
    ii. It will set the height to auto
    iii. It will automatically set the max-width of the image to 100% and the height to auto
    iv. None of the above
3. Which of the following is the shape style of images?
    i. `img-border`
    ii. `img-thumbnail`
    iii. `img-pixel`
    iv. `img-round`

4. What will the below snippet output?

```
<table class="table table-inverse">
...
</table>
```

   i. It makes the table responsive

   ii. If you `hover` over a row in the table, its background color will change to indicate a state change

   iii. Will add zebra striping to your table, starting with the first row in the body and applying a light grey `background color` on all the odd numbered rows

   iv. The table colors will flip to be a dark background with light text

5. With left and right alignment, you can also provide a `column size` within the `class name`.

   i. True

   ii. False

5
Playing with Components

The real power of Bootstrap lies in the components contained within the framework. In this lesson, we'll go through a number of new and existing components. I'll show you how to use them and then we'll insert them into our sample blog project so you can see them in practice. Let's get right into it by covering one of the most commonly used components, which are buttons.

Using the button component

Buttons are one of the most commonly used components in Bootstrap. In version 4 of Bootstrap, some of the new options for the button component include an outlined variation, toggle states, and button groups with checkboxes and radios. Before we get into that, let's review the basic button options and configuration. Here's a few general points to keep in mind when using buttons:

- No matter what type of button you are creating, it will require the .btn CSS class to be included at a minimum
- The .btn class can be attached to a number of HTML tags, such as <button>, <a>, and <input>, to render a button
- There are different CSS classes for creating different size and color buttons

Basic button examples

Before we move on to more advanced configuration, let's cover the basics of creating Bootstrap buttons. If you aren't new to Bootstrap, you may want to skip this section. Bootstrap comes with six different button color options out of the box. Here's a breakdown of their names and when to use them:

- **Primary**: The main button used on your website. It is blue by default.
- **Secondary**: The alternate or secondary button used in your website. It is white by default.
- **Success**: Used for positive-based actions. It is green by default.
- **Info**: Used for informational buttons. It is a light blue by default.
- **Warning**: Used for warning-based actions. It is yellow by default.
- **Danger**: Used for error-based actions. It is red by default.

Now that I've explained all the button variations, let's check out the code for a button:

```
<button type="button" class="btn btn-primary">Primary</button>
```

As you can see, I'm using the `<button>` tag and I've added a couple of CSS classes to it. The first is the `.btn` class, which I mentioned you need to include on all buttons. The second is the `.btn-primary` class, which indicates that you want to use the Primary button variation. If you want to use a different button style, you simply change up that second class to use the corresponding keyword. Let's take a look at the code for all of the button variations:

```
<button type="button" class="btn btn-primary">Primary</button>
<button type="button" class="btn btn-secondary">Secondary</button>
<button type="button" class="btn btn-success">Success</button>
<button type="button" class="btn btn-info">Info</button>
<button type="button" class="btn btn-warning">Warning</button>
<button type="button" class="btn btn-danger">Danger</button>
<button type="button" class="btn btn-link">Link</button>
```

It's as easy as that. Note that the last line is a **Link** button option that I haven't talked about. This variation will appear as a text link in the browser, but will act as a button when you click or hover over it. I don't often use this variation so I left it out at first. If you view this code in your browser, you should see the following buttons:



Creating outlined buttons

Starting in Bootstrap 4, they've introduced a new button variation which will produce an outlined button instead of a filled one. To apply this look and feel, you need to change up one of the button classes. Let's take a look at the following code for all variations:

```
<button type="button" class="btn btn-primary-outline">Primary</button>
<button type="button" class="btn btn-secondary-outline">Secondary</
button>
<button type="button" class="btn btn-success-outline">Success</button>
<button type="button" class="btn btn-info-outline">Info</button>
<button type="button" class="btn btn-warning-outline">Warning</button>
<button type="button" class="btn btn-danger-outline">Danger</button>
```

As you can see, the class names have changed; here's how they map to each button variation:

- `btn-primary-outline`
- `btn-secondary-outline`
- `btn-success-outline`
- `btn-info-outline`
- `btn-warning-outline`
- `btn-danger-outline`

Basically, you just need to append `-outline` to the default button variation class name. Once you do, your buttons should look like this:



Checkbox and radio buttons

A new feature in Bootstrap 4 is the ability to convert checkboxes and radio buttons into regular buttons. This is really handy from a mobile standpoint because it is much easier to touch a button than it is to check a box or tap a radio button. If you are building a mobile app or responsive website, it would be a good idea to use this component. Let's start by taking a look at the code to generate a group of three checkboxes as a button group:

```
<div class="btn-group" data-toggle="buttons">
  <label class="btn btn-primary active">
    <input type="checkbox" checked autocomplete="off"> checkbox 1
```

```
    </label>
    <label class="btn btn-primary">
      <input type="checkbox" autocomplete="off"> checkbox 2
    </label>
    <label class="btn btn-primary">
      <input type="checkbox" autocomplete="off"> checkbox 3
    </label>
  </div>
```

Let me break down the code and explain what is going on here:

- To generate a button group with checkboxes, you need to wrap the boxes in a `<div>` with a class of `.btn-group`.
- To allow the buttons to toggle on and off, you also need to add the data attribute `data-toggle="buttons"` to the `<div>`.
- Next we need to use the button classes on the `<label>` tag to convert each checkbox into a button. Note that on the first button I'm using the `.active` class, which will make this checkbox toggled on by default. This class is totally optional.
- Your basic checkbox `<input>` tag is nested within the label.

Keep in mind since these are checkboxes, you can toggle multiple options on or off. Here's what the button group should look like when rendered in the browser:



As you can see, this renders a nice-looking button group that is optimized for mobile and desktop. Also, notice how the first checkbox has a different background color as it is currently toggled on because of the `.active` class applied to that label. In the same way that we've created a button group with checkboxes, we can do the same thing with radio buttons.

Creating a radio button group

Creating a radio button group is very similar to the checkboxes. Let's start by checking out the code to generate this different variation:

```
<div class="btn-group" data-toggle="buttons">
  <label class="btn btn-primary active">
    <input type="radio" name="options" id="option1" autocomplete="off"
checked> radio 1
  </label>
```

```
  <label class="btn btn-primary">
    <input type="radio" name="options" id="option2"
autocomplete="off"> radio 2
  </label>
  <label class="btn btn-primary">
    <input type="radio" name="options" id="option3"
autocomplete="off"> radio 3
  </label>
</div>
```

Let me explain what's happening here with this code:

- Like the checkboxes, you need to wrap your collection of radio buttons in a `<div>` with the same class and data attribute
- The `<label>` tag and button classes also work the same way
- The only difference is that we are swapping the checkbox `<input>` type for radio buttons

Keep in mind that with radio buttons, only one can be selected at a time. In this case, the first one is selected by default, but you could easily remove that. Here's what the buttons should look like in the browser:



As you can see, the button group is rendered the same way as the checkboxes, but in this case we are using radios. This should be the expected result to optimize your group of radio buttons for mobile and desktop. Next we'll build on what we've learned about button groups, but learn how to use them in other ways.



We'll circle back later in this lesson and actually add the components to our blog project.

Using button groups

If you're new to Bootstrap, button groups are exactly as they sound. They are a group of buttons that are connected horizontally or vertically to look like a single component. Let's take a look at the code to render the most basic version of the component:

```
<div class="btn-group" role="group" aria-label="Basic example">
  <button type="button" class="btn btn-secondary">Left</button>
  <button type="button" class="btn btn-secondary">Middle</button>
  <button type="button" class="btn btn-secondary">Right</button>
</div>
```

As you can see, we have a group of regular button tags surrounded by `<div>` with a class of `.btn-group` on it. At the very least, this is all you need to do to render a button group. There are a couple of other optional attributes on the `<div>` tag, which are `role` and `aria-label`. If you need to worry about accessibility, then you should include those attributes, otherwise they are optional. One other small change in this code is I've decided to use the `.btn-secondary` class to mix things up a bit with the button styles. Let's take a look at how this will appear in the browser:



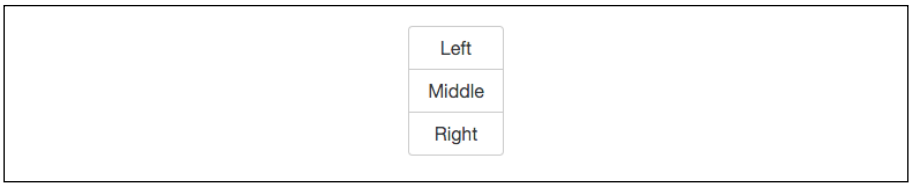
As you can see, we have a single component that is made up of three buttons. This component is commonly used for a secondary navigation, or in a form like I explained in the previous section. If you'd like to display the buttons vertically, that is also possible with a small change.

Creating vertical button groups

If you'd like to arrange the buttons in your group vertically, that is actually quite easy to do. There is no need to change any of the code on the `<button>` tags, you just need to update the CSS class name on the wrapping `<div>` tag. Here's the code you need to change:

```
<div class="btn-group-vertical">
  ...
</div>
```

If you make that alteration to your code, then the same button group will appear like this in the browser:



It would probably have made sense to change the left button label to the top and the right button label to the bottom. However, I left them as they are because I wanted to show you how you can simply shift the alignment of the group by changing one CSS class. That covers the basics of using the button groups component; in the next section, I'll show you how to create button drop-down menus.

Coding a button dropdown

The code to render a button as a dropdown is a little bit more complicated but still fairly easy to get up and running. You'll combine a button tag with `<div>` that has a nested collection of links inside it. Let's take a look at the code required to render a basic drop-down button:

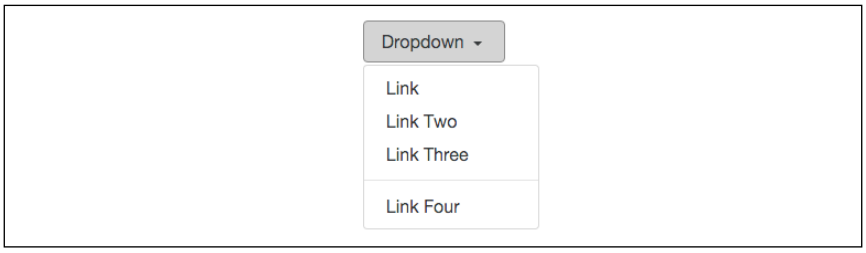
```
<div class="btn-group">
  <button type="button" class="btn btn-secondary dropdown-toggle"
data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
    Dropdown
  </button>
  <div class="dropdown-menu">
    <a class="dropdown-item" href="#">Link</a>
    <a class="dropdown-item" href="#">Link Two</a>
    <a class="dropdown-item" href="#">Link Three</a>
    <div class="dropdown-divider"></div>
    <a class="dropdown-item" href="#">Link Four</a>
  </div>
</div>
```

Okay, there are a few things going on here. Let's break them down one by one and explain how the dropdown works:

- The entire component needs to be wrapped in a `<div>` with a class of `.btn-group` on it.
- Next you insert a `<button>` tag with some button CSS classes on it. Like in the previous section, some of the other attributes are optional. However, it is a good idea to include this attribute: `aria-expanded`. This can either be set to `false` or `true` and controls whether the dropdown is open or closed on page load. In most cases, you will want to set this to `false`.

- After the `<button>` tag, insert another `<div>` tag which will hold all the links that appear in the drop-down menu list. Make sure you give this `<div>` a class of `.dropdown-menu`.
- Within the second `<div>` you insert a collection of `<a>` tags, one for each item in your list. Each `<a>` tag requires a class of `.dropdown-item` so that the proper CSS styling is applied.
- You may also want to insert a divider in your drop-down list if you have a large amount of links. This is done by inserting a third `<div>` with a class of `.dropdown-divider` on it.

As I mentioned, this component is a little more complex, but in Bootstrap 4 they have actually simplified it a bit to make it easier to use. Let's take a look at what it should look like in the browser. In the following screenshot, I've showed what the expanded version of the dropdown will look like so you can see the button and the list of links:



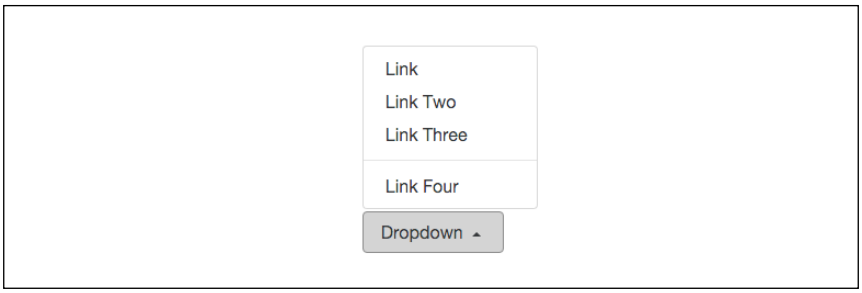
As you can see, we have a drop-down button with a list of links nested within it. Keep in mind that if you want to use this component, it does require that you include jQuery and `bootstrap.min.js` in your template. There are some other variations of this component you can easily implement, such as the pop-up menu.

Creating a pop-up menu

In some cases, you might want to have your menu pop up above the button instead of below it. You can achieve this by adding one class on the wrapping `<div>` for the component. Check out the code here:

```
<div class="btn-group dropup">
  ..
</div>
```

As you can see, I've added the class `.dropup` to the `<div>`. This will make the menu appear above the button, and it should look like this:



As you can see, the list appears above the button when it is expanded.

Creating different size drop-down buttons

By adding a single class to the `<button>` tag in the dropdown, you can make the trigger larger or smaller. Let's take a look at the code for the smaller and larger button variations:

```
<!-- large button //-->
<div class="btn-group">
  <button class="btn btn-secondary btn-lg dropdown-toggle"
type="button" data-toggle="dropdown" aria-haspopup="true" aria-
expanded="false">
    Large button
  </button>
  <div class="dropdown-menu">
    ...
  </div>
</div>

<!-- small button //-->
<div class="btn-group">
  <button class="btn btn-secondary btn-sm dropdown-toggle"
type="button" data-toggle="dropdown" aria-haspopup="true" aria-
expanded="false">
    Small button
  </button>
  <div class="dropdown-menu">
    ...
  </div>
</div>
```

If you find the button tag in the first example, you'll see I've added a class of `.btn-lg` to it. This class will increase the button size to be larger than the default. Take a look at the second chunk of code, find the `<button>` tag again, and you'll see a class of

`.btn-sm` on it. This works the same way except the button will now be smaller than the default. Let's see how these buttons will render in the browser.



The .btn-lg and .btn-sm classes are not exclusive to the button drop-down component. You can use them on any button component variation you like.



That concludes the basics of using the button drop-down component. In the next section, we'll cover a more complicated component, which is forms.

Coding forms in Bootstrap 4

If you are familiar with Bootstrap 3, then you'll notice the CSS form classes are pretty much the same in version 4. The biggest change I see in forms for the new version is that each form group uses a `<fieldset>` tag instead of `<div>`. If you are new to Bootstrap forms, a basic form group is made up of a label and an input. It can also include help text, but that is optional. Let's jump right in by creating a generic form that uses a number of core components.

Setting up a form

At the very least, a form needs to be made up of one input and one button. Let's start with the basics and create a form following those requirements. Here's the code to get you started:

```
<form>
  <fieldset class="form-group">
    <label>Text Label</label>
    <input type="text" class="form-control" placeholder="Enter Text">
    <small class="text-muted">This is some help text.</small>
  </fieldset>
  <button type="submit" class="btn btn-primary">Submit</button>
 </form>
```

Let me explain what is happening here in the code:

- Every form needs to start with a `<form>` tag. However, no special classes are

required on this tag.

- I've inserted a `<fieldset>` tag with a class of `.form-group` on it for our single input. This `<fieldset>` pattern will be repeated in the future when you add additional inputs.
- Within the `<fieldset>`, we have a `<label>`. Again, no special CSS classes need to be added to the `<label>`.
- After the label, you need to insert the form `<input>` tag. In this case, I'm using a text input. On this HTML tag, you need to add a class of `.form-control`. All input tags in Bootstrap will require this class. The placeholder text is optional but nice to add for usability.
- In the last line of the `<fieldset>`, I've included a `<small>` tag with a class of `.text-muted`, which will render the text small and light grey. This is the pattern you should use if you want to include some help text with your form input.
- Close the `<fieldset>` tag and then you need to add a `<button>` tag for the form submit button.
- Close the `<form>` and you are done.

After you've finished reviewing the code, fire up your web browser, and your form should look like this:



You've successfully coded your first Bootstrap 4 form. Let's continue and I'll explain how to implement other common form components using the latest version of Bootstrap.

Adding a select dropdown

Let's build on our form code by adding a select drop-down menu. Insert the following code after our text input:

```
<fieldset class="form-group">
  <label>Select dropdown</label>
  <select class="form-control">
```

```
      <option>one</option>
      <option>two</option>
      <option>three</option>
      <option>four</option>
      <option>five</option>
    </select>
  </fieldset>
```

Let's break down the parts of the code you need to be aware of:

- Note that the entire `<select>` is wrapped in a `<fieldset>` with a class of `.form-group`. This pattern should repeat for each type of form input you add.
- On the `<select>` tag, there is a class of `.form-control` that needs to be added.
- Aside from that, you should code the `<select>` as you normally would, following the best HTML syntax practices.

Once you're done, if you view the form in the browser, it should now look like this:



That completes the setup for <select> dropdowns. Next let's check out the <textarea> tag.

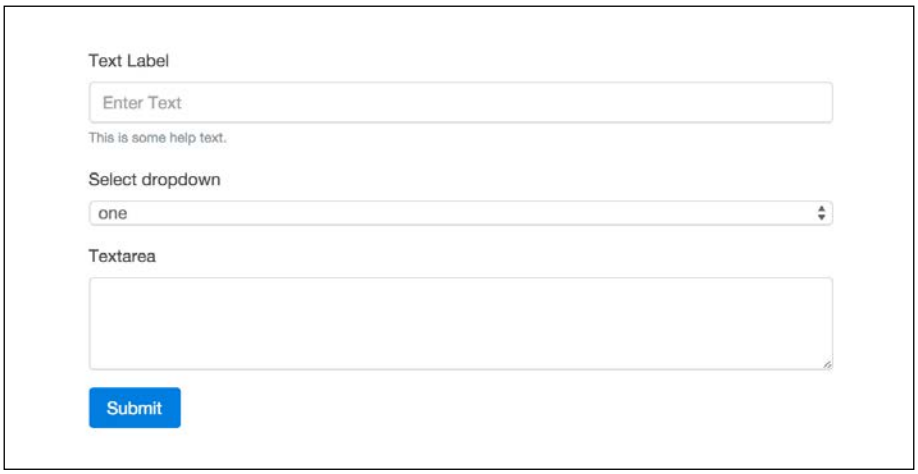
Inserting a Textarea tag into your form

Moving along to the next input type, let's insert a `<textarea>` tag into our form. After the `<select>` menu, add the following code:

```
<fieldset class="form-group">
  <label>Textarea</label>
  <textarea class="form-control" rows="3"></textarea>
</fieldset>
```

Using this input is fairly simple. Like our other examples, you need to use a `<fieldset>` tag with a CSS class of `.form-group` to wrap the entire thing. On the

actual `<textarea>` tag, you need to add the `.form-control` class. That's it; once you're done, your form should now look like this:



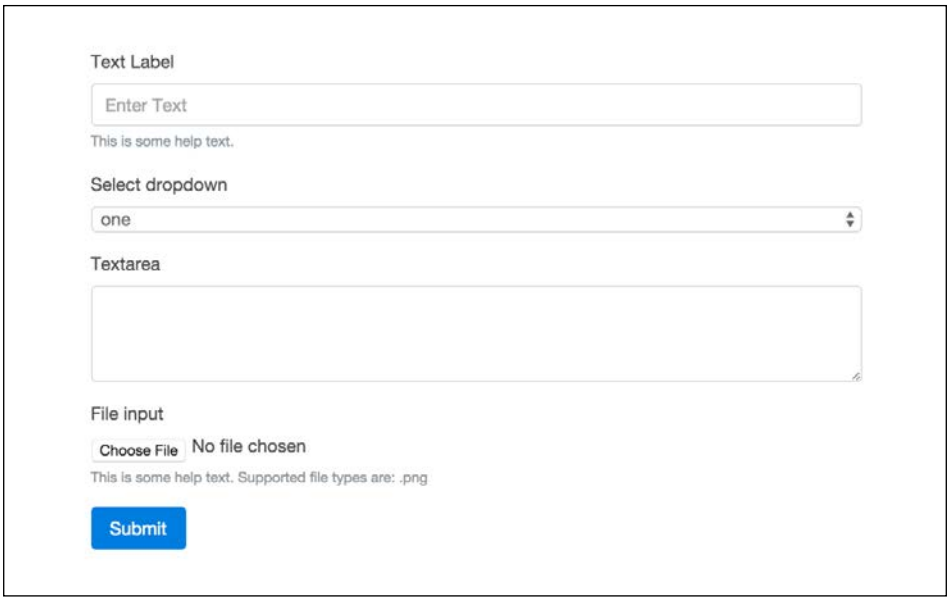
Now that the `<textarea>` is complete, let's move on to the file input form field.

Adding a file input form field

Historically, the file input form field has been a tricky one to style with CSS. I'm happy to say that in Bootstrap 4 they've created a new approach that's the best I've seen to date. Let's start by inserting the following code after the <textarea> in our form:

```
<fieldset class="form-group">
  <label>File input</label>
  <input type="file" class="form-control-file">
  <small class="text-muted">This is some help text. Supported file
types are: .png</small>
</fieldset>
```

Again, this form field is constructed in the same manner as the previous ones. However, there is one small change you need to be aware of with the File input field. On the `<input>` tag, you need to change the CSS class to `.form-control-file`. There are some specific styles being applied to clean up the look and feel of this form field. Once you're done, your form should look like this:



That completes the **File input** field which leaves us with two more basic form field inputs to go over. They are radio buttons and checkboxes. Let's learn how to add them next.

Inserting radio buttons and checkboxes to a form

These fields are pretty similar so I'm going to group them together in their own section. The code for these two fields differs a little bit from the other inputs, as I'll outline now. First, let's insert the following code after the File input field in our form:

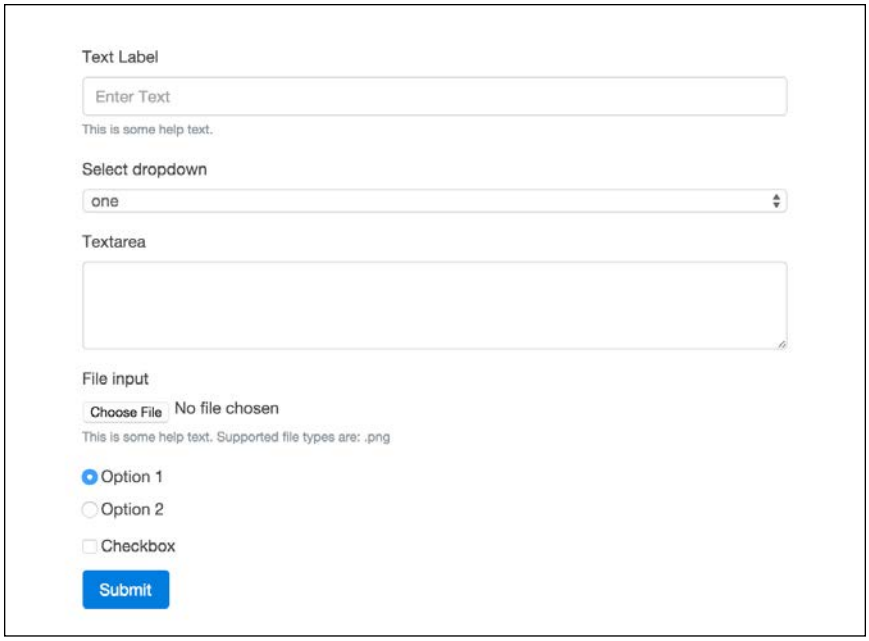
```
<div class="radio">
  <label>
    <input type="radio" name="optionsRadios" id="optionsRadios1"
value="option1" checked>
    Option 1
  </label>
</div>
<div class="radio">
  <label>
    <input type="radio" name="optionsRadios" id="optionsRadios2"
value="option2">
       Option 2
  </label>
</div>
```

```
<div class="checkbox">
  <label>
    <input type="checkbox"> Checkbox
  </label>
</div>
```

Let's start by going over the radio button code first, then we'll move on to the checkbox:

- The fields don't use the `<fieldset>` tag as the wrapper. In this case, you should use a `<div>` and give it a class of either `.radio` or `.checkbox`, depending on what type you want to use.
- For these fields, the `<label>` tag will also wrap around the `<input>` tag so that everything is displayed in a horizontal line. We don't want the text label to drop down below the radio button or checkbox.
- You don't need a special class on the `<input>` for either of these fields.

As you can see, the code for these fields is a bit different from what we've learned about the other form inputs. Not to worry, as they are pretty easy to use and there aren't a bunch of CSS classes you have to memorize. One of the nicest changes with forms in Bootstrap 4 is that they require less HTML markup, so are easier to write. Finally, if you view our form in the browser, it should look like this:



That completes the explanation of all the core form fields that you need to know how to use in Bootstrap 4. Before we move on to some more advanced form fields and variations, why don't we add a form to our blog project?

Adding a form to the blog contact page

I know, I know. I said we would wait till the end of the lesson to build components into the blog project. However, I'm thinking you might like a break from learning and actually add some of what you've learned to your project. Let's go ahead and do just that by filling in a form on the contact page.

Updating your project

Let's start by opening up our project directory and finding the file named `contact.ejs`. Open up that file in your text editor and we are going to add some new form code and remove some filler code. To start, find the body section of the page that is wrapped in the following column `<div>`:

```
<div class="col-md-12">
```

Within that `<div>` is currently some filler text. Remove that text and replace it with the following form code:

```
<form>
  <fieldset class="form-group">
    <label>Email</label>
    <input type="email" class="form-control" placeholder="Enter
email">
    <small class="text-muted">We'll never share your email with anyone
else.</small>
  </fieldset>
  <fieldset class="form-group">
    <label>Name</label>
    <input type="text" class="form-control" placeholder="Name">
  </fieldset>
  <fieldset class="form-group">
    <label>Message</label>
    <textarea class="form-control" rows="3"></textarea>
  </fieldset>
  <button type="submit" class="btn btn-primary">Submit</button>
</form>
```

I've coded up a basic contact form that you'll commonly see on a blog. It has e-mail, name, and message fields along with a **Submit** button. Save your file and then preview your project in a browser. The contact page should now look like this:



That concludes the updates to the contact page for now. Later on in the course, we'll add some additional components to this page. Let's jump back into learning about forms in Bootstrap 4 by reviewing some additional form controls.

Additional form fields

Now that we've learned how to build a basic form and added one to our project, let's circle back and talk about some more advanced form fields and variations you can apply with Bootstrap 4. I'm going to start by showing you how to lay out forms in a few different ways.

Creating an inline form

Let's start by learning how to create an inline form. This is a layout you might want to use in the header of a project or perhaps for a login page. In this case, we're going to align the fields and buttons of the form vertically across the page. For this example, let's create a simple login form with the following code:

```
<form class="form-inline">
  <div class="form-group">
    <label>Name</label>
    <input type="text" class="form-control" placeholder="Mike Smith">
  </div>
  <div class="form-group">
    <label>Email</label>
    <input type="email" class="form-control" placeholder="mike@gmail.
com">
  </div>
```

```
    <button type="submit" class="btn btn-primary">Login</button>
</form>
```

There are a few things going on in this form, so let me explain them for you:

- For inline forms, we need to add a CSS class named `.form-inline` to the `<form>` tag.
- You'll also notice the `<fieldset>` tags have been replaced with `<div>` tags. This is so they can be set to display as inline-block, which won't work with a fieldset.

Aside from those two differences, the form is coded up the same way as a regular one. Once you're done, your form should look like this in the browser:



If you're like me, you might find the labels next to the text inputs kind of ugly. The good news is there is an easy way to hide them.

Hiding the labels in an inline form

The reason those labels are there is for accessibility and screen readers. We don't want to remove them altogether from the code, but we can hide them by adding a CSS class named `.sr-only`. This class stands for screen reader only and will therefore only show the labels if they are viewed on an accessible screen reader. Here is an example of how to add the CSS class:

```
<label class="sr-only">Name</label>
```

After you apply that CSS class to all the labels in the form, it should now appear like this in the browser:



That concludes how to make a basic inline form. However, what if you want to include other fields in an inline manner? Let's see how we can add checkboxes and radios.

Adding inline checkboxes and radio buttons

If you'd like to include checkboxes and radio buttons to an inline form you need to make some changes to your code. Let's start by going over the checkbox code. Insert the following code after the last text input in the inline form:

```
<label class="checkbox-inline">
  <input type="checkbox" value="option1"> Remember me?
</label>
```

There are a couple of things here that you need to be aware of:

- First, there is no longer a `<div>` wrapped around the checkbox
- You need to add a class named `.checkbox-inline` to the checkbox's `<label>` tag

Once you do this, save your form and it should look like this in the browser:



Now that we've added the checkbox, let's check out an example using radio buttons. Add the following code to your form after the checkbox code:

```
<label class="radio-inline">
  <input type="radio" name="inlineRadioOptions" id="inlineRadio1"
value="option1"> Yes
</label>
<label class="radio-inline">
  <input type="radio" name="inlineRadioOptions" id="inlineRadio2"
value="option2"> No
</label>
```

As you can see, the pattern here is exactly the same. The `<div>` around each radio button has been removed. Instead, there is a CSS class named `.radio-inline` that needs to be added to each radio `<label>` tag. Once you've completed this step, your form should look like this:



That completes everything you need to know about inline forms. Let's now move on to some more utility-type actions that you can apply to your form fields.

Changing the size of inputs

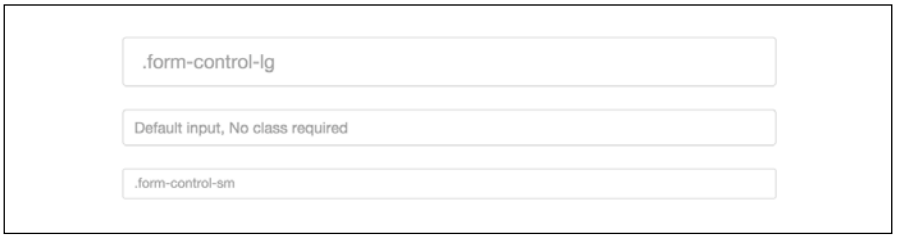
Bootstrap comes with a few handy utility CSS classes that you can use with form fields to have them appear at different sizes. Along with the default size, you can choose to display your fields in a larger or smaller size. Let's take a look at the code to render all three size variations:

```
<input class="form-control form-control-lg" type="text"
placeholder="form-control-lg">
<input class="form-control" type="text" placeholder="Default input, No
class required">
<input class="form-control form-control-sm" type="text"
placeholder="form-control-sm">
```

To use the different size inputs, you simply have to add an additional class to the tag:

- For a larger input, use the class `.form-control-lg`
- For a smaller input, use the class `.form-control-sm`
- The default input size requires no extra CSS class

Here's how each version looks in the browser:



As you can see, the larger input is taller and has some additional padding. The smaller input is shorter with reduced padding. These classes only cover the vertical size of an input. Now let's learn how to control the width of inputs.

Controlling the width of form fields

Since Bootstrap is a mobile-first framework, form fields are designed to stretch to fit the width of their column. Therefore, if you are using `.col-md-12` for your column class, the field is going to stretch to the width of the layout. This may not always be what you want, you may only want the input to stretch to half of the width of the layout.

If this is the case, you need to wrap your field in a `<div>` with a column class on it to control the width. Let's check out some example code to get the point across:

```
<div class="col-md-12">
    <input type="text" class="form-control" placeholder="full width">
</div>
<div class="col-md-6">
    <input type="text" class="form-control" placeholder="half width">
</div>
```

In the preceding code, I've removed some of the labels and other form code to make it easier to see what is going on. Here's a breakdown of what you need to know:

- You need to wrap your input in a `<div>` with a column class on it
- The first input will stretch to the width of the layout because of the `.col-md-12` class
- The second input will only stretch to fill 50% of the layout because of the `.col-md-6` class

Let's take a look at how this will look in the actual browser:



As you can see, the second input only stretches to half of the width. This is how you can control the width of inputs if you don't want them to fill the entire layout of your page. The last thing I'd like to cover when it comes to forms is validation of input fields.

Adding validation to inputs

Bootstrap 4 comes with some powerful yet easy to use validation styles for input fields. Validation styles are used to show things such as errors, warnings, and success states for form fields when you submit the actual form. Let's take a look at the code to render all three validation states:

```
<div class="form-group has-success">
  <label class="form-control-label">Input with success</label>
  <input type="text" class="form-control form-control-success">
</div>
<div class="form-group has-warning">
  <label class="form-control-label">Input with warning</label>
  <input type="text" class="form-control form-control-warning">
```

```
</div>
<div class="form-group has-danger">
  <label class="form-control-label">Input with danger</label>
  <input type="text" class="form-control form-control-danger">
</div>
```

The markup for each validation variation is very similar to a regular input with the addition of a few CSS classes to apply the proper state look and feel. Let's go over each change you need to be aware of:

- The first input is the success state. The wrapping `<div>` needs to have a class called `.has-success` added to it.
- Each `<label>` tag needs to have a class named .form-control-label added to it. This is required to color the label to match the state color.
- The success input requires a class named `.form-control-success`.
- The second input is the warning state. The wrapping `<div>` needs a class named `.has-warning` added to it.
- The warning input also needs a class named `.form-control-warning` added.
- Finally, the last input is the danger or error state. The wrapping `<div>` needs to have a class named `.has-danger` added.
- The danger input also needs a class named `.form-control-danger` added.

Let's take a look at how all these validation inputs should look in the browser:



As you can see, the inputs and labels are colored to match their state. You'll also notice each input has an icon to the right edge of it. These icons are automatically added when you include the required CSS files. There is no need to actually use any images here, which is great. That concludes everything that you need to know about forms in Bootstrap 4. In the next section, I'll teach you about the Jumbotron component.

Using the Jumbotron component

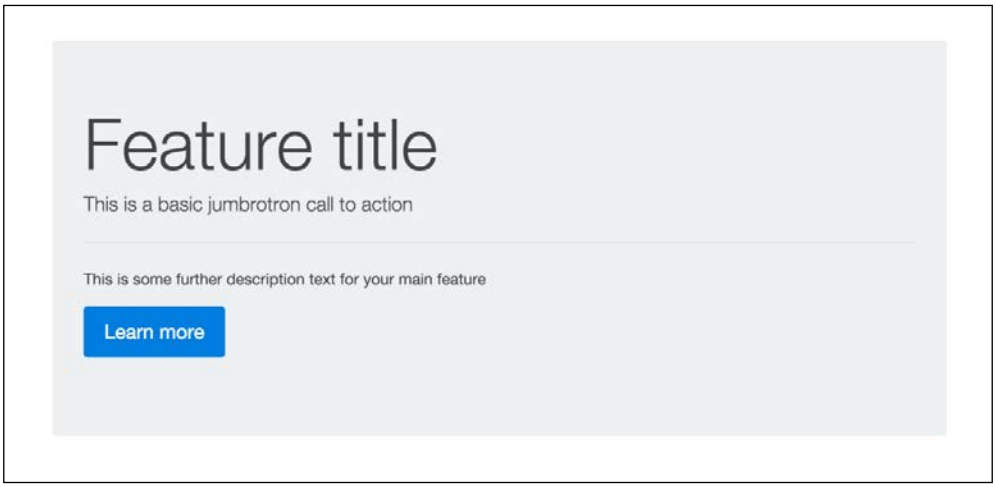
If you're new to Bootstrap, you may be asking yourself what the heck is a Jumbotron component. Jumbotron is used to feature a block of content, usually at the top of your page. This is your standard main feature block that you'll see on a number of websites. If you require something more sophisticated than a simple page title, Jumbotron is the component you'll want to use. Let's take a quick look at the code required to create this component:

```
<div class="jumbotron">
  <h1 class="display-3">Feature title</h1>
  <p class="lead">This is a basic jumbrotron call to action</p>
  <hr class="m-y-2">
  <p>This is some further description text for your main feature</p>
  <p class="lead">
    <a class="btn btn-primary btn-lg" href="#" role="button">Learn
more</a>
  </p>
</div>
```

There are some new CSS classes here that we need to review, as well as some existing ones we have already learned about. Let's break down what's happening in the code:

- The Jumbotron component is based off a `<div>` with a CSS class named `.jumbotron`. Within this `<div>`, you can pretty much use whatever text formatting tags you like. However, there are a few basics you should include to make it look good.
- The first tag you'll see is the `<h1>` with a class of .display-3 on it. Since the Jumbotron is more of a "display" component, you'll want to beef up the size of your `<h1>` by using the optional class we learned about earlier in the course.
- Next, you'll see a simple `<p>` tag for the feature's tagline. On that tag, there is a class named `.lead`. This class increases the base font size by 25% and sets the font-weight to 300 which is a lighter weight. Again, this gives the Jumbotron component more of a "feature" like look and feel.
- After the tagline text, you'll see an `<hr>` tag with a class of `.m-y-2` on it. If you remember, this is a utility spacing class. The -y in this case will add a margin above and below the `<hr>` tag.
- Next we have another `<p>` tag with some additional descriptive text in it.
- Finally, we have a `<button>` wrapped in a `<p>` tag so that there is a conclusion to the call to action in the Jumbotron block. Note that the user of the `.btn-lg` class will produce a larger-sized button.

After you've coded up your Jumbotron component, it should look like this in the browser:



By default, the Jumbotron component will stretch to fit the width of the column it is contained within. In most cases, you'll likely want it to span the entire width of your page. However, in some cases, you might want a Jumbotron to stretch from one edge of the browser to the other without any horizontal padding on it. If this is the case, you need to add the .jumbotron-fluid class to the main `<div>` and make sure it is outside of a Bootstrap .container. Let's take a look at the following code to see what I mean:

```
<div class="jumbotron jumbotron-fluid">
  <div class="container">
    <h1 class="display-3">Feature title</h1>
      <p class="lead">This is a basic jumbrotron call to action</p>
  </div>
</div>
```

As you can see, the .container `<div>` is now inside of the Jumbotron `<div>`. This is how you remove the horizontal padding on the section. Once completed, it should look like this in the browser:



That concludes the use of the Jumbotron component in Bootstrap 4. Next let's move on to learning how to use the Label component.

Adding the Label component

The Label component is used to add context to different types of content. A good example would be notifications on an application. You might use a label to indicate how many unread emails there are in an email app. Another would be to insert a warning tag next to an item in a table or list. Like buttons, labels are available in a number of color variations to meet your needs in your project. Let's take a look at the code to render the basic label options:

```
<span class="label label-default">Default</span>
<span class="label label-primary">Primary</span>
<span class="label label-success">Success</span>
<span class="label label-info">Info</span>
<span class="label label-warning">Warning</span>
<span class="label label-danger">Danger</span>
```

You'll likely notice some similarities here with the Button component CSS classes. When using a label, you should use the `<span>` tag as your base for the component. Here are some more important facts when using this component:

- Every variation of the Label component requires the use of the `.label` class on the `<span>` tag
- The Default label uses the `.label-default` class and is grey
- The Primary label uses the `.label-primary` class and is blue
- The Success label uses the `.label-success` class and is green
- The Info label uses the `.label-info` class and is light blue
- The Warning label uses the `.label-warning` class and is yellow

- Finally, the **Danger** label uses the `.label-danger` class and is red

Once you've coded that up, it should look like this in your browser:



By default, labels will be rectangular with slightly rounder corners. If you'd like to display them in pill format, you can do so by adding the `.label-pill` class to the `<span>` tag. Here's an example to see what I mean:

```
<span class="label label-pill label-default">Default</span>
<span class="label label-pill label-primary">Primary</span>
<span class="label label-pill label-success">Success</span>
<span class="label label-pill label-info">Info</span>
<span class="label label-pill label-warning">Warning</span>
<span class="label label-pill label-danger">Danger</span>
```

If you add that class to your labels, they should look like this in the browser:



That concludes the Label component in Bootstrap 4. Next, I'll teach you how to use the Alerts component.

Using the Alerts component

The Alerts component in Bootstrap provides contextual messages for typical uses, such as validation and general information that needs to stand out more. Like our previous components, it comes in a few different variations depending on your needs. Let's start by looking at the basic code required to render the different alert options:

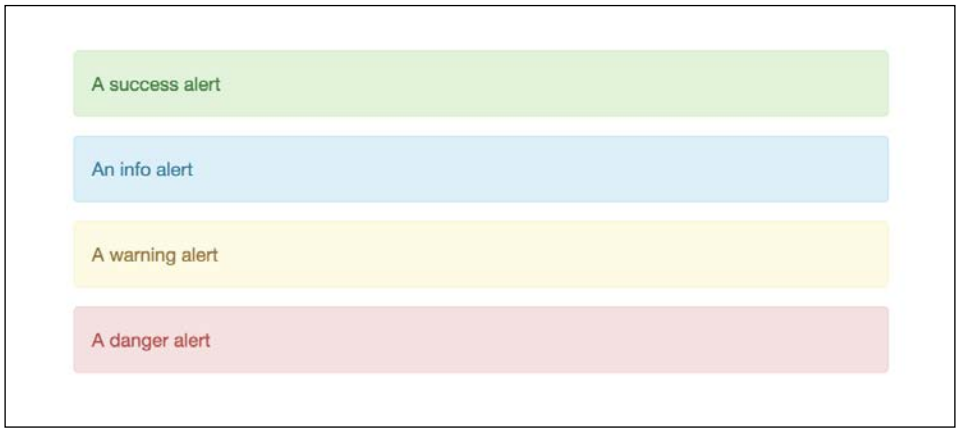
```
<div class="alert alert-success" role="alert">
  A success alert
</div>
```

```
<div class="alert alert-info" role="alert">
  An info alert
</div>
<div class="alert alert-warning" role="alert">
  A warning alert
</div>
<div class="alert alert-danger" role="alert">
  A danger alert
</div>
```

The classes used to create an alert can be added to any block element, but for demo purposes we'll implement them using `<div>` tags. Here are the key points you need to know:

- Any instance of the Alert component will require the use of the `.alert` CSS class on the `<div>` tag
- You also need a second CSS class to indicate which version of the alert you want to use
- The Success alert uses the class `.alert-success` and is green
- The Info alert uses the class `.alert-info` and is blue
- The Warning alert uses the class `.alert-warning` and is yellow
- The Danger alert uses the class `.alert-danger` and is red

Once you've set up the code for those alerts, they should look like this in the browser:



That was a basic example of using Alerts. There are some additional things you can do to extend this component such as adding a dismiss button.

Adding a dismiss button to alerts

If you want to make your alert bar dismissible, you can add a button to do this. To include the link, update the code for your bar, as follows:

```
<div class="alert alert-success" role="alert">
    <button type="button" class="close" data-dismiss="alert" aria-
label="Close">
    <span aria-hidden="true">&times;</span>
  </button>
  A success alert
</div>
```

The previous Alert bar code doesn't change, but you do need to add a button before the alert message:

- The `<button>` requires a class named `.close` to appear
- You'll also need the `data-dismiss` attribute to be included with a value of alert
- The `×` code will be rendered as an **X** in the browser

Once you've added the new code, your alert bar should look like this:



Now your alert bar has a dismissible **X** button that can be triggered to close when you implement the functionality of the component in your app or website. That completes the Alert component in Bootstrap 4. In the next section, I'll teach you about the best new component in version 4, which is Cards.

Using Cards for layout

In my opinion, the best new feature in Bootstrap 4 is the new Card component. If you're unfamiliar with Cards, they were made popular with the release of Google Material Design. They are a mobile first content container that works well for phones, tablets, and the desktop.

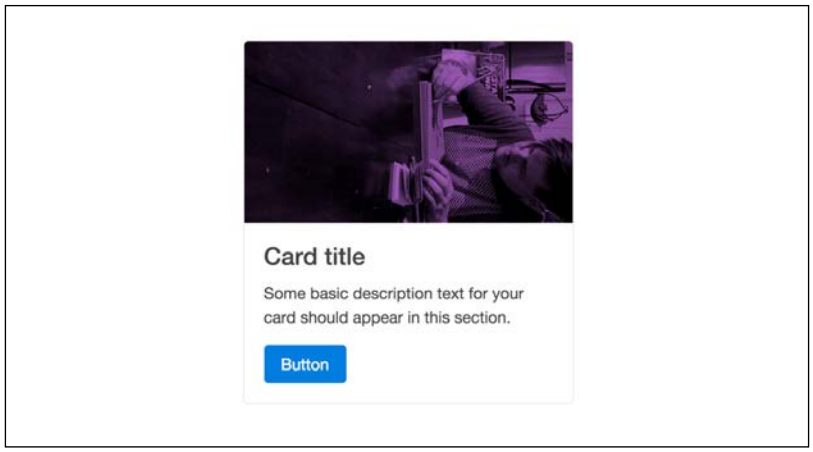
We'll be using the Card component heavily in our blog project so let's jump right in and start learning how to use them. Check out the following code to learn how to render a basic card:

```
<div class="card">
  <img class="card-img-top img-fluid"  src="path/to/your/image.jpg">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
```

There are a number of new CSS classes you need to be aware of here, so let's go through them one by one:

- Any instance of the Card component must use a `<div>` tag with a CSS class named `.card` on it.
- If you wish to include an image inside your card, it comes next. The image requires a class named `.card-img-top` to display the image at the top of the card. Although not required, I would also recommend adding the class `.img-fluid` to your image. This will make the image responsive so that it will automatically resize to match the width of your card.
- After the image, you need to start a new `<div>` with a CSS class named `.card-block`. This part of the Card will contain the actual textual content.
- The first thing your card should have is a title. Use an `<h4>` tag with a CSS class of .card-title for this section.
- Next, you can insert a paragraph of text with a `<p>` tag and a class of `.card-text`. If you choose to have multiple paragraphs, make sure each one uses that same class name.
- Finally, I've inserted a primary `<button>` so the user has something to click on to view the full piece of content.

After you've finished coding this up, it should appear like this in your browser. Note for demo purposes, I've included an image of my own so you can see how it works. You'll need to provide your own images for your projects:



As you can see, this will render a neat-looking little content component that you can use in many different ways. Let's move on by learning some other ways that you can customize the Card component.

Moving the Card title

Perhaps you want to move the title of your card above the image? This is actually really easy to do, you simply need to move the `<title>` tag before the image in the flow of the component, like this:

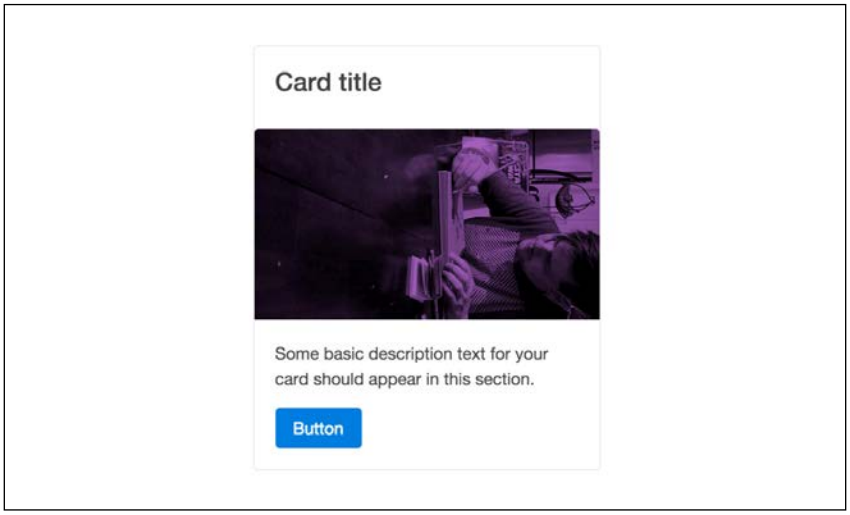
```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
  </div>
  <img
  class="card-img-top img-fluid"
  src="http://mattlambert.ca/img/matt-lambert.jpg">
  <div class="card-block">
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
```

There are a couple of things here that you need to know about:

- There are now two instances of `<div class="card-block">` in this card. It is perfectly fine to reuse this section within a single card. You'll notice that the header tag is wrapped inside of this `<div>`. This is required to apply the proper padding and margin around the title in the card.

- The second thing you need to note is that the header tag has been moved above the image in the Card layout.

After making this change, your card should look like this:



Hopefully this shows you how easy it is to work with different content in cards. Let's continue by showing some other things that you can do.

Changing text alignment in cards

By default, text and elements will always align left in a card. However, it is possible to change this quite easily. Let's create a second card and then we'll center one and right align the other. I'm going to remove the image so the code is easier to understand:

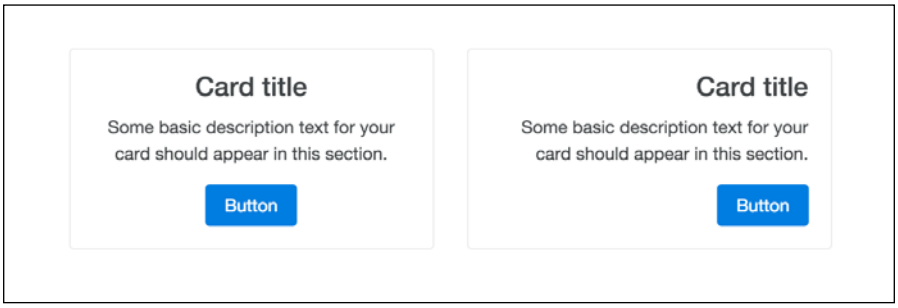
```
<div class="card">
  <div class="card-block text-xs-center">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
<div class="card">
  <div class="card-block text-xs-right">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
```

```
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
```

Not much has changed here, but let's go over what is different:

- First, as I mentioned, I removed the image to make the code simpler
- On the first card, I've added a class of `.text-xs-center`, which will center the text in the card
- On the second card, I added a class named `.text-xs-right`, which will right align everything

That's all you need to do. If you view this in the browser it should look like this:



So with one additional CSS class we can easily control the alignment of the text and elements in a card. Cards are a pretty powerful component, so let's continue to learn how you can customize them.

Adding a header to a Card

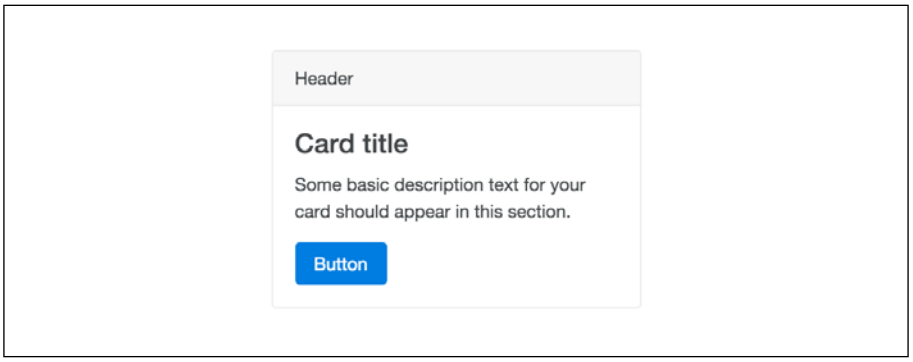
If you want to add a header to your Card, this is also pretty easy to do. Check out this code sample to see it in action:

```
<div class="card">
    <div class="card-header">
    Header
  </div>
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
```

With the addition of a new section of code, we can add a header:

- Before the `.card-block` section, insert a new `<div>` with a class named `.card-header`
- Within this new `<div>`, you can add the header title

Save your file and check it out in the browser, and it should look like this:



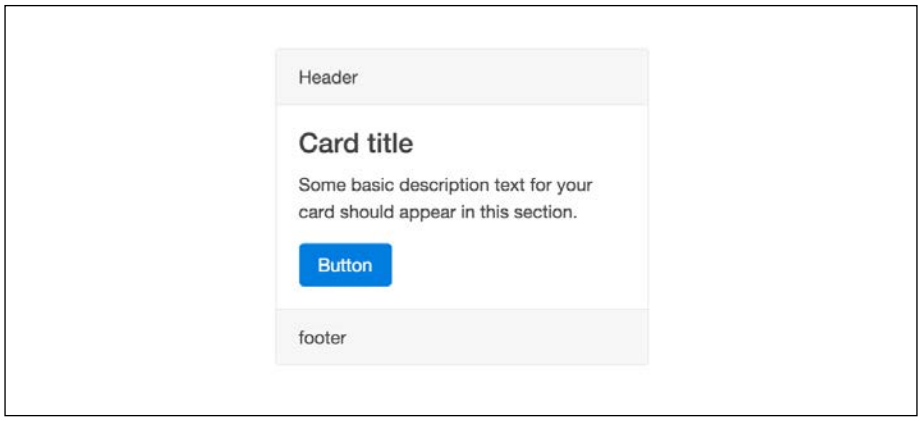
That's a super easy way to add a header section to your card. You can add a footer in the same manner. Let's add some additional code for the footer:

```
<div class="card">
    <div class="card-header">
    Header
  </div>
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
  <div class="card-footer">
      footer
  </div>
</div>
```

The setup for the footer is very similar to the header; let's break it down:

- This time, below the `.card-block` section, insert a new `<div>` with a class named `.card-footer`
- Inside this new `<div>`, insert your footer text

Save the file again and view it in the browser, and it should look like this:



Easy as that, we've now also included a footer with our Card. Next, let's learn a way to apply a different look and feel to our Card.

Inverting the color scheme of a Card

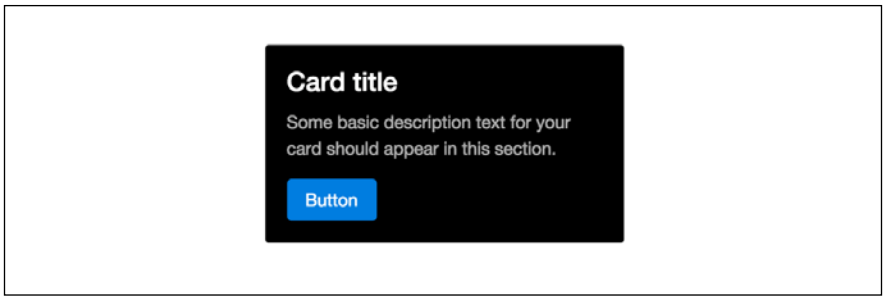
In some cases, you may want a different look and feel for your Card to make it stand out more. There are some CSS classes included with Bootstrap that will allow you to inverse the color scheme. Let's take a look at the code to apply this style:

```
<div class="card card-inverse" style="background:#000;">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    <a href="#" class="btn btn-primary">Button</a>
  </div>
</div>
```

Again, this variation is pretty easy to apply with a couple of small changes:

- On the `<div>` with the .card class, add a second class named `.card-inverse`.
- This will only inverse the text in the card. You need to set the background color yourself. For speed, I just did an inline CSS style in the demo code. I'd recommend actually creating a CSS class in your stylesheet for your own project, which is a nicer way to do things.

That's all you need to do. Once you're done, your card should look like this:



In this case, you do need to specify the custom background color. However, Bootstrap does have some background color variations that you can use if you want to add an additional CSS class. The naming convention for these options is just like buttons and labels. Let's take a look at what the code will look like:

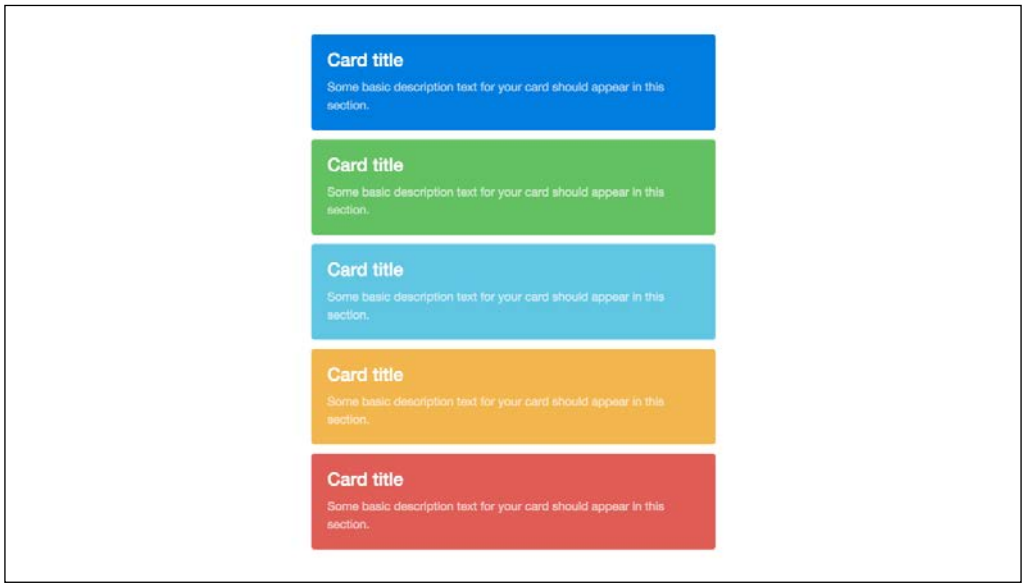
```
<div class="card card-inverse card-primary">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
  </div>
</div>
<div class="card card-inverse card-success">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
  </div>
</div>
<div class="card card-inverse card-info">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
  </div>
</div>
<div class="card card-inverse card-warning">
  <div class="card-block">
    <h4 class="card-title">Card title</h4>
    <p class="card-text">Some basic description text for your card
should appear in this section.</p>
  </div>
</div>
<div class="card card-inverse card-danger">
  <div class="card-block">
```

```
      <h4 class="card-title">Card title</h4>
      <p class="card-text">Some basic description text for your card
should appear in this section.</p>
    </div>
</div>
```

This is a bunch of code, but there are only a couple of things that change from our previous card example:

- All I've done is add an additional CSS class to the `<div>` with our base `.card class` on it. Let's review each one in the following points.
- The Primary card uses the `.card-primary` class and is blue.
- The Success card uses the `.card-success` class and is green.
- The Info card uses the `.card-info class` and is light blue.
- The Warning card uses the `.card-warning` class and is yellow.
- The Danger card uses the `.card-danger` class and is red.

Once you've set up the above code, your cards should look like this in the browser:



That concludes the basic and advanced styling you can do with the Card component. Why don't we take a break from learning for a bit and actually build some Cards in our blog project.

Adding a location card to the Contact page

Let's jump back into our project by adding a simple Card component to the Contact page. Reopen `contact.ejs` in your text editor and head down to the main body that we recently updated with a contact form. Find the following column code for that section:

```
<div class="col-md-12">
```

We're going to split this full width column into two separate columns. Change the class on the previous snippet of code to `.col-md-8` and add a new `<div>` with a class of `.col-md-4` on it. When you're done, the body of the page code should now look like this:

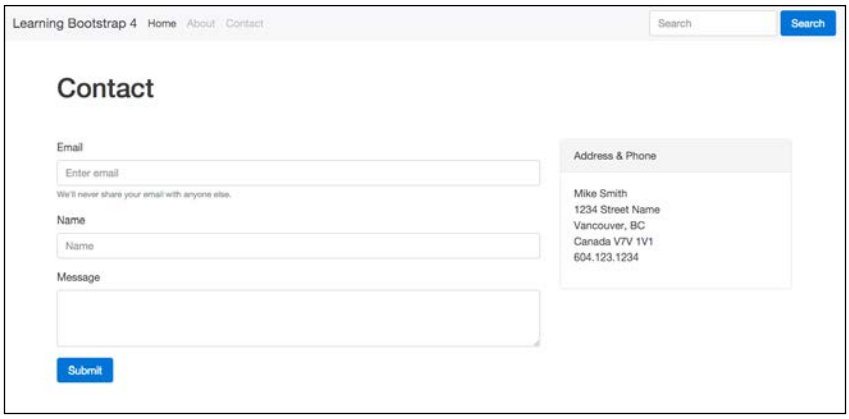
```
<div class="col-md-8">
      <form>
        <fieldset class="form-group">
          <label>Email</label>
          <input type="email" class="form-control" placeholder="Enter
email">
          <small class="text-muted">We'll never share your email with
anyone else.</small>
        </fieldset>
        <fieldset class="form-group">
          <label>Name</label>
          <input type="text" class="form-control" placeholder="Name">
        </fieldset>
        <fieldset class="form-group">
          <label>Message</label>
          <textarea class="form-control" rows="3"></textarea>
        </fieldset>
        <button type="submit" class="btn btn-primary">Submit</button>
      </form>
    </div>
    <div class="col-md-4">
    </div>
```

Now that the column is set up, let's insert a Card component into our new column. Enter the following code into the second column in the layout:

```
<div class="card">
  <div class="card-header">
    Address &amp;amp; Phone
  </div>
  <div class="card-block">
    <ul class="list-unstyled">
      <li>Mike Smith</li>
```

```
        <li>1234 Street Name</li>
        <li>Vancouver, BC</li>
        <li>Canada V7V 1V1</li>
        <li>604.123.1234</li>
      </ul>
    </div>
  </div>
```

Once you've inserted the Card component code, save your file and check it out in a browser. It should look like this:



Now the Contact page is starting to take more shape. Let's add the Card component to a few other pages before we move on to our next Content component.

Updating the Blog index page

Now that we've covered the card component, it's time to set up the main layout for our Blog index page. The design is going to rely heavily on the Card component, so let's get to it. First of all, open up `index.ejs` in your text editor and find the body of the page section. The left column will look like this:

```
<div class="col-md-8">
```

Within this `<div>` currently is some filler text. Delete the filler text and insert the following Card component, which will be our first Blog post:

```
<div class="card">
  <img class="card-img-top img-fluid" src="img/image.jpg" alt="Card
image cap">
  <div class="card-block">
```

```
    <h4 class="card-title">Post title</h4>
    <p><small>Posted by <a href="#">Admin</a> on January 1, 2016 in <a
href="#">Category</a></small></p>
    <p class="card-text">Some quick example text to build on the card
title and make up the bulk of the card's content.</p>
    <a href="#" class="btn btn-primary">Read More</a>
  </div>
</div>
```

Now that we've added our first card to the Blog roll, let's break down what's happening:

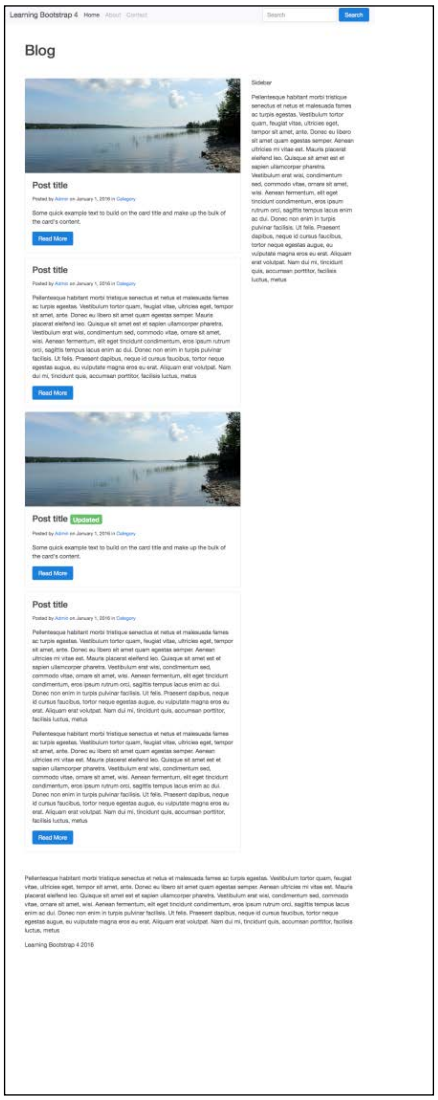
- I've started by including a photo I took in Nova Scotia a few summers ago. I've given it a class of `.img-fluid` so it stretches the width of the card.
- From there, I've set up my card exactly like I taught you previously, but in this case, I've added some real content for a blog.

Let's go ahead and add the rest of the Card component code for the blog roll. Insert the following code after the first Card in the left column:

```
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Post title</h4>
    <p><small>Posted by <a href="#">Admin</a> on January 1, 2016 in <a
href="#">Category</a></small></p>
    <p>Pellentesque habitant morbi tristique...</p>
    <a href="#" class="btn btn-primary">Read More</a>
  </div>
</div>
<div class="card">
  <img class="card-img-top img-fluid" src="img/image.jpg" alt="Card
image cap">
  <div class="card-block">
    <h4 class="card-title">Post title <span class="label label-
success">Updated</span></h4>
    <p><small>Posted by <a href="#">Admin</a> on January 1, 2016 in <a
href="#">Category</a></small></p>
    <p class="card-text">Some quick example text to build on the card
title and make up the bulk of the card's content.</p>
    <a href="#" class="btn btn-primary">Read More</a>
  </div>
</div>
<div class="card">
  <div class="card-block">
    <h4 class="card-title">Post title</h4>
    <p><small>Posted by <a href="#">Admin</a> on January 1, 2016 in <a
```

```
href="#">Category</a></small></p>
    <p>Pellentesque habitant morbi tristique senectus...</p>
    <a href="#" class="btn btn-primary">Read More</a>
  </div>
</div>
```

That's a long chunk of code. The filler text is just in there to give you an idea. Feel free to remove that or replace it with actual text. Now that we've filled out the left column with a good amount of content, your page should look like this:



Now that the main blog roll content is complete, let's also add the right column content.

Adding the sidebar

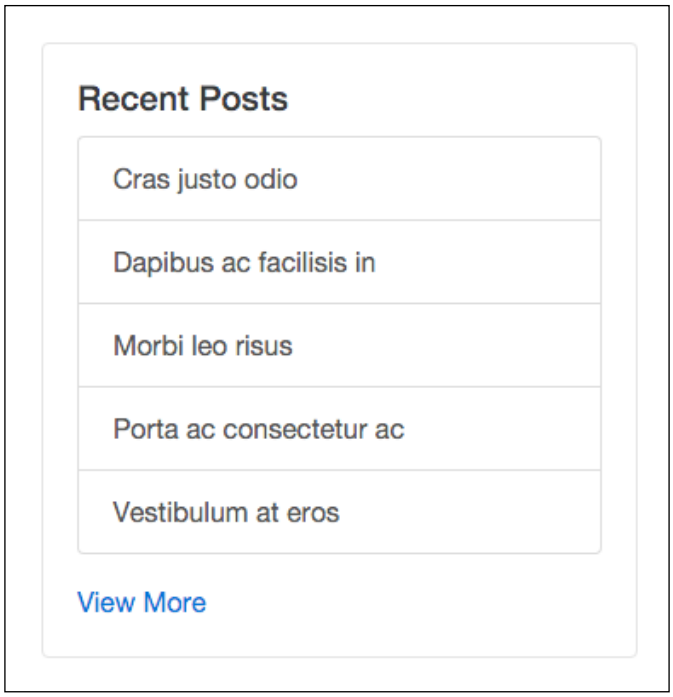
Let's add some more content to the index page by adding the sidebar. We'll also use the Card component here, but in this case, some different variations of it. Go back to index.ejs and remove the filler text from the second column. Instead, insert the following Card code:

```
<div class="card card-block">
  <h5 class="card-title">Recent Posts</h5>
  <div class="list-group">
    <button type="button" class="list-group-item">Cras justo odio</
button>
    <button type="button" class="list-group-item">Dapibus ac facilisis
in</button>
    <button type="button" class="list-group-item">Morbi leo risus</
button>
    <button type="button" class="list-group-item">Porta ac consectetur
ac</button>
    <button type="button" class="list-group-item">Vestibulum at eros</
button>
  </div>
  <div class="m-t-1"><a href="#">View More</a></div>
</div>
```

You'll notice in this Card I'm using a different variation, which is the List Group option. To do this, follow these steps:

- Create a new `<div>` with a class of .list-group inside your card.
- Inside, insert a `<button>` with a class of `.list-group-item` on it for every item of your list. I've added five different options.

Once you're done, save your file and it should look like this in the browser:

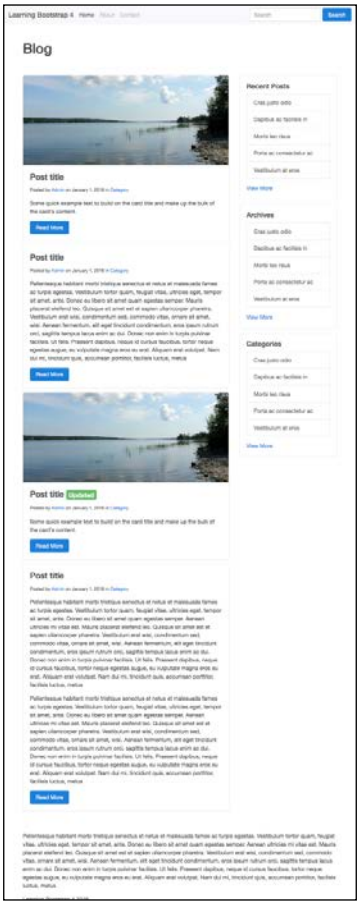


As you can see, that will draw a nice-looking sidebar list component. Let's fill out the rest of the sidebar by inserting the following code after the first Card component:

```
<div class="card card-block">
  <h5 class="card-title">Archives</h5>
  <div class="list-group">
    <button type="button" class="list-group-item">Cras justo odio</
button>
    <button type="button" class="list-group-item">Dapibus ac facilisis
in</button>
    <button type="button" class="list-group-item">Morbi leo risus</
button>
    <button type="button" class="list-group-item">Porta ac consectetur
ac</button>
    <button type="button" class="list-group-item">Vestibulum at eros</
button>
  </div>
  <div class="m-t-1"><a href="#">View More</a></div>
</div>
<div class="card card-block">
  <h5 class="card-title">Categories</h5>
  <div class="list-group">
```

```
    <button type="button" class="list-group-item">Cras justo odio</
button>
    <button type="button" class="list-group-item">Dapibus ac facilisis
in</button>
    <button type="button" class="list-group-item">Morbi leo risus</
button>
    <button type="button" class="list-group-item">Porta ac consectetur
ac</button>
    <button type="button" class="list-group-item">Vestibulum at eros</
button>
  </div>
  <div class="m-t-1"><a href="#">View More</a></div>
</div>
```

This will produce two more List Group Card components for the sidebar of your blog project. Once it's all done, the entire page should now look like this:



That concludes the user of the Card component on the index page. The last page we need to set up with the Card component is our Blog post page.

Setting up the Blog post page

The index page is a list of all the Blog posts in our project. The last page we need to setup is the Blog post page, which is just a single post in our project. Open up the `blog-post.ejs` template you created earlier in the course and let's start updating some code. Head down to the page body section and find this line of code:

```
<div class="col-md-8">
Currently, you'll see some filler text in that <div>; let's replace it
with the following code:
<div class="card">
  <div class="card-block">
    <p><small>Posted by <a href="#">Admin</a> on January 1, 2016 in <a
href="#">Category</a></small></p>
    <p>Pellentesque habitant morbi tristique senectus et...</p>
    <p><code>&lt;p&gt;this is what a code sample looks like&lt;/
p&gt;</code></p>
    <p>Pellentesque habitant morbi tristique senectus et netus...</p>
    <!-- pre sample start //-->
    <h4>pre sample code</h4>
    <pre>This is what code will look like</pre>
    <!-- pre sample end //-->
    <!-- image //-->
    <h4>responive image</h4>
    <p><img src="img/image.jpg" class="img-fluid" alt="Responsive
image"></p>
    <!-- table //-->
    <h4>table</h4>
    <table class="table">
      <thead>
        <tr>
          <th>#</th>
          <th>First Name</th>
          <th>Last Name</th>
          <th>Username</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th scope="row">1</th>
          <td>john</td>
```

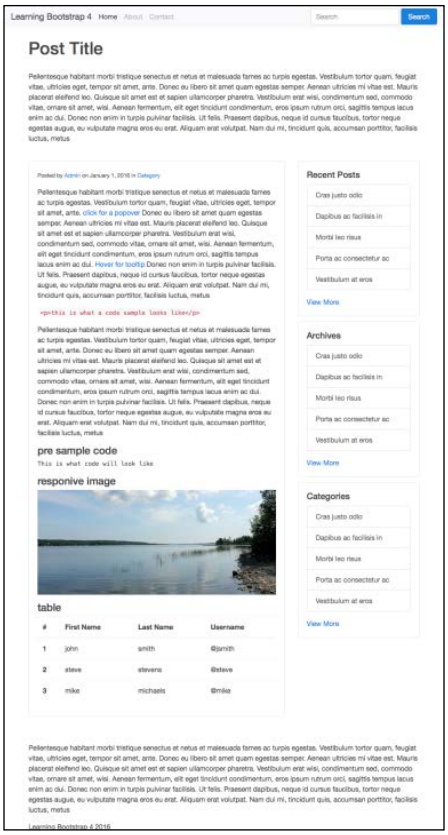
```
            <td>smith</td>
            <td>@jsmith</td>
          </tr>
          <tr>
            <th scope="row">2</th>
            <td>steve</td>
            <td>stevens</td>
            <td>@steve</td>
          </tr>
          <tr>
            <th scope="row">3</th>
            <td>mike</td>
            <td>michaels</td>
            <td>@mike</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
```

There's a good chunk of things going on in this code. I've thrown in a few other components we've already learned about so you can see them in action. The Card component has the following things included inside it:

- Text, `<code>` and `<pre>` tags
- Tables
- Images

Let's also update this template to use the same sidebar code as the index page. Copy the right column code from the index template and paste it into the same location in the blog post template.

When you are done, the page should now look like this:



As you can see, we're using a single Card component to hold all of the content for the body of the page. We're also using the same Card components for the sidebar that we copied over from the index page. Now that we've added the Cards to all of our page templates, let's get back to learning about some other Content components in Bootstrap 4.

How to use the Navs component

The Navs component in Bootstrap can be displayed in a couple of different ways. The default view for the component is just a simple unstyled list of links. This list can also be transformed into tabs or pills for ways of organizing your content and navigation. Let's start by learning how to create a default Nav component:

```
<ul class="nav">
  <li class="nav-item">
    <a class="nav-link" href="#">Link 1</a>
  </li>
```

```
    <li class="nav-item">
      <a class="nav-link" href="#">Link 2</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="#">Link 3</a>
    </li>
  </ul>
```

The most basic version of the Nav component is built using the preceding code:

- The component is based on an unordered list with a class of `.nav`
- Each `<li>` tag in the list requires a class of `.nav`-item
- Nested inside the `<li>` tag must be an `<a>` tag with a class of `.nav-link`

Once you've completed adding that code it should look like this in the browser:



As I mentioned, this is just a basic unstyled list of links. One easy change you can make is to display the list of links inline horizontally. To achieve this, you just need to add a class named `.nav-inline` to the `<ul>` tag, like this:

```
<ul class="nav nav-inline">
```

This will display all the links in a horizontal line. Why don't we move on to something a little more exciting, such as converting this list into tabs.

Creating tabs with the Nav component

Converting the basic list to tabs is easy to do by adding a couple of things to our code. Take a look at this sample:

```
<ul class="nav nav-tabs">
  <li class="nav-item">
    <a class="nav-link active" href="#">Link 1</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link 2</a>
  </li>
  <li class="nav-item">
```

```
      <a class="nav-link" href="#">Link 3</a>
    </li>
  </ul>
```

I've made two changes to the code, let's review them now:

- On the `<ul>` tag, I removed the `.nav-inline class` and added `.nav-tabs`. This will render the list as tabs.
- I then added a class of `.active` to the first link so that it is the selected tab when the page loads.

After you've coded that up, it should look like this in the browser:



Just like that you can render the list as a set of tabs. The next variations you'll want to try are pills.

Creating a pill navigation

Changing the style of the Nav component to Pills is actually really easy. Take a look at the following sample code:

```
<ul class="nav nav-pills">
  <li class="nav-item">
    <a class="nav-link active" href="#">Link 1</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link 2</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#">Link 3</a>
  </li>
</ul>
```

Let's breakdown what is new here. I've only made one change to the code. I've removed the `.nav-tabs` class from the `<ul>` tag and replaced it with a `.nav-pills` class. This is the only change you need to make.

Save your file with the changes and it should look like this in the browser:



The preceding example is the default display for Nav pills. There is another variation you can try though, which are stacked pills. This pattern is commonly used in sidebar navigations. To create this version, update the following line of code:

```
<ul class="nav nav-pills nav-stacked">
```

Here I've simply added a class of `.nav-stacked` to the `<ul>` tag to stack the pills. Here's how it will look in the browser:



That concludes the Nav component in Bootstrap 4. As you learned, it's pretty easy to create a few different styles of navigation with a simple list of unordered links. In the next section, we'll review the more complicated navigation component, which is the Navbar.

Using the Bootstrap Navbar component

The Navbar component is a staple of Bootstrap that gets used all the time. In the past, this component has required a decent amount of markup to get it working. I'm glad to report that in Bootstrap 4 they have simplified this component and made it easier to use. Let's start by going over a basic example of the Navbar:

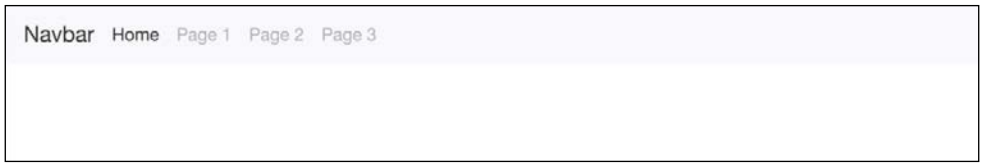
```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="#">Navbar</a>
  <ul class="nav navbar-nav">
    <li class="nav-item active">
      <a class="nav-link" href="#">Home</a>
    </li>
    <li class="nav-item">
```

```
        <a class="nav-link" href="#">Page 1</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Page 2</a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Page 3</a>
      </li>
    </ul>
  </nav>
```

You may notice some similarities here with the Nav component. The Navbar uses some of the same code, but you can extend it further and combine additional components into it. Let's start by breaking down this basic example:

- A Navbar component can be used outside or inside of a `<div>` with a `.container` class on it. If you want the Navbar to be flush with the edges of the browser, you should not include it inside a .container `<div>`. However, if you do want the default padding and margins applied, put it inside the `<div>` . For this example, I'm going to build it outside of a container.
- The Navbar component starts with an HTML5 `<nav>` tag that has the following CSS classes added to it.
- `.navbar` is the default class that always needs to appear on the component.
- `.navbar-light` is the color of component you want to use. There are some other variations you can pick from.
- `.bg-faded` is a utility class that you can use to make the background lighter. This is an optional class.
- The first element inside of a Navbar is the Brand. The Brand should be the title for your project. To render the element, create an `<a>` tag and give it a class of `.navbar-brand`. The anchor text for this link should be the name of your project or website. Keep in mind, using the Brand is optional.
- The core part of the Navbar is the list of navigation links. This is created with an unordered list, similar to the Nav component. In this case, your `<ul>` tag should have classes of `.nav` and `.navbar-nav` included.
- The nested `<li>` and `<a>` tags should use the same `.nav-item` and `.nav-link` classes from the Nav component.

This will create a basic Navbar component for you. This is how it should look in the browser:



Now that you've learned how to build a basic Navbar, let's learn how to extend the component further.

Changing the color of the Navbar

In Bootstrap 3, you could invert the color scheme of the Navbar. However, in Bootstrap 4 you have multiple options for coloring the Navbar component. All that is needed to edit is some of the classes on the `<nav>` tag that wrap the component. Let's take a look at the code for some of the different color options:

```
<nav class="navbar navbar-inverse">
  ...
</nav>
<nav class="navbar navbar-primary">
  ...
</nav>
<nav class="navbar navbar-success">
  ...
</nav>
<nav class="navbar navbar-warning">
  ...
</nav>
<nav class="navbar navbar-info">
  ...
</nav>
<nav class="navbar navbar-danger">
  ...
</nav>
```

As you can see, we're reusing the keywords for color variations that we've used in other components. Let's break down each variation of the Navbar component:

- `.navbar-inverse` will color the component black and grey
- `.navbar-primary` will color the component blue
- `.navbar-success` will color the component green
- `.navbar-warning` will color the component yellow
- `.navbar-info` will color the component light blue

- `.navbar-danger` will color the component red

Once you're done coding that up, the navbars should look like this in the browser:



As you can see, we now have the Navbar in a whole range of colors you can choose from. Let's learn what else we can add to this component.

Making the Navbar responsive

Being that Bootstrap is a mobile-first framework, it would only make sense that you need the ability to make the Navbar component responsive. Let's check out the basic code for this:

```
<nav class="navbar navbar-light bg-faded">
   <button class="navbar-toggler hidden-sm-up" type="button" data-
toggle="collapse" data-target="#responsive-nav">

  </button>
  <div class="collapse navbar-toggleable-xs" id="responsive-nav">
     <a class="navbar-brand" href="#">Navbar</a>
     <ul class="nav navbar-nav">
       <li class="nav-item active">
         <a class="nav-link" href="#">Home</a>
       </li>
       <li class="nav-item">
```

```
          <a class="nav-link" href="#">Page 1</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Page 2</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Page 3</a>
        </li>
      </ul>
    </div>
</nav>
```

There's a few different things in the code here that you need to be aware of:

- After the opening `<nav>` class, you need to insert a `<button>` with the CSS classes `.navbar-toggle` and `.hidden-sm-up`. The first class says this button will toggle the navigation. The second class says only show the responsive navigation for sizes above small. You also need to include the data attribute data-toggle="collapse" to all the Nav to collapse. Finally, you need to add a data-target, which will point to the area you want to be collapsible. I've given that an ID of #responsive-nav.
- Next, head down to your list of links and wrap a `<div>` around them. This section needs CSS classes named .collapse and `.navbar-toggleable-xs`. You also need to give it an ID of responsive-nav to tie it to the button from the previous step.

That's it; once you code this up shrink your browser window to a small size and your bar should switch to look like this. Oh, and don't forget that the code in the button will render a hamburger menu icon in the responsive Navbar:



That concludes the Navbar component in Bootstrap 4. I know this has been a long lesson, but we only have a few more components to go over.

Adding Breadcrumbs to a page

The Breadcrumbs component is a pretty easy one to use in Bootstrap. Let's check out the code for how to render one:

```
<ol class="breadcrumb">
  <li><a href="#">Home</a></li>
  <li><a href="#">Page 1</a></li>
  <li class="active">Page 2</li>
</ol>
```

As you can see, the code for this component is pretty basic, let's review it:

- The Breadcrumb component uses an ordered list or `<ol>` tag as its base.
- Within the ordered list, you simply just need to create a list of links. The last item in the list should have a class of `.active` on it.

Adding Breadcrumbs to the Blog post page

For this example, let's actually add some Breadcrumbs to our Blog post page template. Open up blog-`post.ejs` and add the following code after the container `<div>` at the top:

```
<div class="row m-t-1">
    <ol class="breadcrumb">
      <li><a href="#">Home</a></li>
      <li><a href="#">Blog</a></li>
      <li class="active">Post Title</li>
    </ol>
  </div>
```

This code should come before the page title and once you make the update, your page should now look like this at the top:



There, now we've added a nice Breadcrumb to our blog post template. Let's move on to adding Pagination to our page templates.

Using the Pagination component

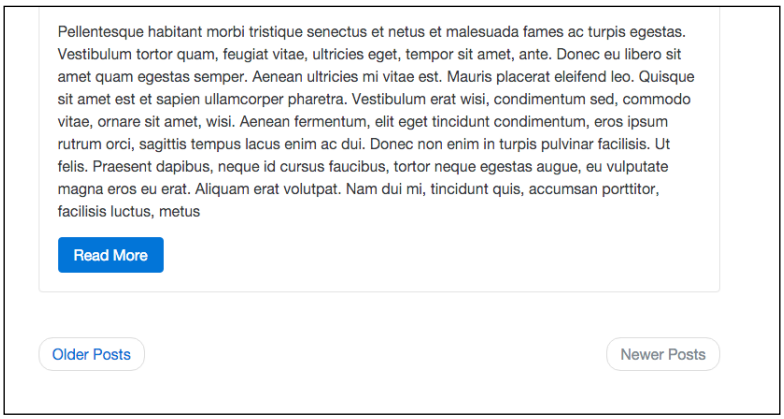
Let's continue adding some more components to our templates by learning how to use the Pagination component. For our blog project, we want to use the Pager version of the component. Open up `index.ejs` and insert the following code after the last Card component in our blog feed:

```
<nav>
  <ul class="pager m-t-3">
    <li class="pager-prev"><a href="#">Older Posts</a></li>
    <li class="pager-next disabled"><a href="#">Newer Posts</a></li>
  </ul>
</nav>
```

The Pager is wrapped in an HTML5 `<nav>` tag and uses an unordered list as its base:

- The `<ul>` tag should have a class of `.pager` added to it.
- The first list item in the group should have a class of `.pager-prev` on it.
- The second list item should have a class of `.pager`-next on it. In this case, I've also added the class `.disabled` which means there are no more posts to go to.

After you've added this code to your index template, it should look like this in the browser:



Let's also add this component to the Blog post page template.

Adding the Pager to the Blog post template

Open up `blog-post.ejs` and paste the same snippet of code from previously at the bottom of the left column, right after the end of the Card component. I won't bother posting another screenshot, as it should look the same as the previous example. Let's continue by learning how to use another component.

How to use the List Group component

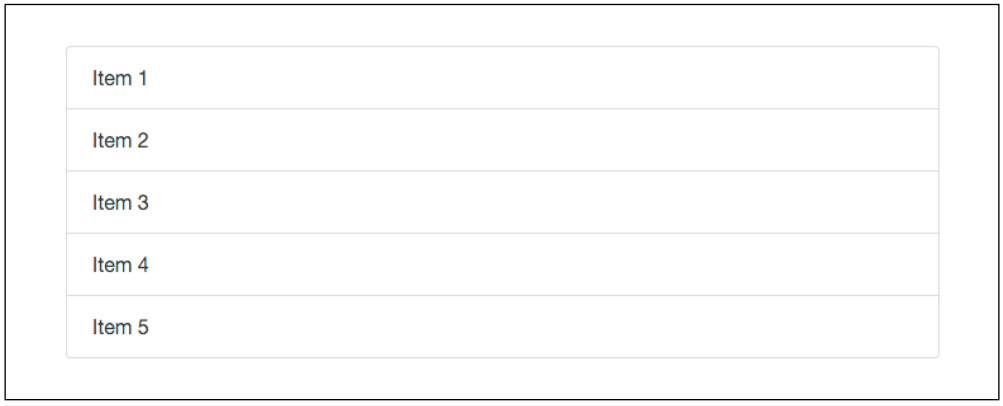
This is the last main content component we need to go over for this lesson. Let's get right into it by reviewing the code needed to render a List Group:

```
<ul class="list-group">
  <li class="list-group-item">Item 1</li>
  <li class="list-group-item">Item 2</li>
  <li class="list-group-item">Item 3</li>
  <li class="list-group-item">Item 4</li>
</ul>
```

Like the components before it, this one is based off of an unordered list:

- The `<ul>` tag needs a class of .list-group on it to start
- Each `<li>` needs a class of .list-group-item on it

Once you're done, your List Group should look like this in the browser:



As you can see, with some minimal coding you can render a decent looking component. You may have missed it, but we actually already used this component when we were building our sidebar on the index and blog post page templates. Open up one of them in a text editor and you'll see the following code, which is a List

Group:

```
<div class="card card-block">
  <h5 class="card-title">Recent Posts</h5>
  <div class="list-group">
    <button type="button" class="list-group-item">Cras justo odio</
button>
    <button type="button" class="list-group-item">Dapibus ac facilisis
in</button>
    <button type="button" class="list-group-item">Morbi leo risus</
button>
    <button type="button" class="list-group-item">Porta ac consectetur
ac</button>
    <button type="button" class="list-group-item">Vestibulum at eros</
button>
  </div>
  <div class="m-t-1"><a href="#">View More</a></div>
</div>
```

That concludes the use of the List Group component. That also concludes the Content components lesson.

Summary

This has been a really long lesson but I hope you have learned a lot. We have covered Bootstrap components including buttons, button groups, button dropdown, forms, input groups, dropdowns, Jumbotron, Label, Alerts, Cards, Navs, Navbar, Breadcrumb, Pagination, and List Group. Our blog project is really starting to take shape now, too. In the next lesson, we'll dive into some JavaScript components in Bootstrap 4 that will include Modal, Tooltips, Popovers, Collapse, and Carousel.

Assessments

1. The .btn class cannot be attached to which among the following HTML tags?
    i. `<button>`
    ii. `<a>`
    iii. `<p>`
    iv. `<input>`
2. Which among the following button is blue by default?
    i. `Primary`
    ii. `Secondary`
    iii. `tsInfo`
    iv. `Success`
3. Which among the following button is yellow by default?
    i. `Primary`
    ii. `Warning`
    iii. `Danger`
    iv. `Secondary`
4. Which among the following is the correct command for displaying a primary button?
    i. `<button type="Button" class="btn btn-primary">Primary</button>`
    ii. `<button type="Button" class="btn-primary">Primary</button>`
    iii. `<button type="button" class="btn-primary">Primary</button>`
    iv. `<button type="button" class="btn btn-primary">Primary</button>`
5. What will be the output if a `<button>` tag has class `.btn-success-outline`?
    i. Entire button will be filled with the color of class `.btn-success`
    ii. The border of the button will get the color of the button with class `.btn-success`
    iii. The text inside the button will get the color of the button with `.btn-success`
    iv. Entire button with the text inside will be filled with the color of class `.btn-success`

6. Which among the following classes groups buttons together on a single line?
    i. `.btn-group`
    ii. `.btn-group-justified`
    iii. `.btn-group-lg`
    iv. `.btn-group-vertical`

7. Which of the following attribute needs to be added to allow the buttons to `toggle on and off`?
    i. `data-target`
    ii. `data-toggle`
    iii. `class`
    iv. `href`

8. Which among the following tags need to be added to convert each checkbox into a button?
    i. `<label>`
    ii. `<div>`
    iii. `<input>`
    iv. `<button>`

9. Which among the factors are required to wrap your collection of radio buttons in a `<div>`?
    i. All the `<div>` should have same class and different data attribute
    ii. All the `<div>` should have same class and same data attribute
    iii. All the `<div>` should have different class and same data attribute
    iv. All the `<div>` should have different class and different data attribute

10. What is the output of class `.dropup` if you apply it on a `<div>`?
    i. This makes the menu disappear
    ii. This makes the menu appear below the button
    iii. This makes the menu appear above the button
    iv. This displays a popup

11. Consider the code below:

```
 <button class="btn btn-secondary btn-sm dropdown-
toggle" type="button"data-toggle="dropdown" aria-
haspopup="true" aria-expanded="false">
```

What is the output of this code?

i. Dropdown button appears large

ii. Dropdown button appears small

iii. Dropdown button appears to be toggled

iv. Dropdown button is having justified size

12. Consider the code below:

```
<form>
  <fieldset class="form-group">
    <label>Text Label</label>
    <input type="text" class="form-control"
placeholder="Enter Text">
    <small class="text-muted">This is some help
text.</small>
  </fieldset>
  <button type="submit" class="btn btn-
primary">Submit</button>
 </form>
```

What does the `<small>` tag with a class of `.text-muted` display?

i. It renders the text small and light grey

ii. It renders the text small and dark grey

iii. It renders the text small

iv. It renders text in dark grey color

13. For which of the following we use the `<select>` tag?

i. Displaying a dropdown

ii. Displaying radio buttons

iii. Displaying checkboxes

iv. None of the above

14. Which among the following tags is used for displaying large text?
    i. `<fieldset>`
    ii. `<textarea>`
    iii. `<input>`
    iv. `<form>`

15. What is the output if the `.form-inline` class is applied on a `<form>` tag with elements inside it?
    i. All of the elements are inlined
    ii. All of the elements are inlined and right-aligned
    iii. All of the elements are inlined and left-aligned
    iv. All of the elements are inlined and center-aligned

16. Which among the following classes is used to hide labels?
    i. `.sr-only`
    ii. `disabled`
    iii. `readonly`
    iv. `.form-control`

17. Which among the following classes will make an image responsive so that it will automatically resize to match the width of your card?
    i. `.img-responsive`
    ii. `.card-img-top`
    iii. `.img-thumbnail`
    iv. `.img-fluid`

18. Which among the following nav classes will color the component black and grey?
    i. `.navbar-warning`
    ii. `.navbar-primary`
    iii. `.navbar-inverse`
    iv. `.navbar-info`

6
Extending Bootstrap with JavaScript Plugins

In this lesson, we're going to dive deeper into Bootstrap components by learning how to extend the framework using JavaScript plugins. You may remember that back in the first lesson we included bootstrap.min.js in our template. This file contains a number of JavaScript components that come with Bootstrap. In this lesson, we'll go over how to use some of these components, including: Modals, Tooltips, Popovers, Collapse, and Carousel. Let's get right to it by learning how to create a Modal in Bootstrap 4.

Coding a Modal dialog

Modals go by a number of different names; you may also know them as dialogs, popups, overlays, or alerts. In the case of Bootstrap, this component is referred to as a Modal and that is how I'll be referring to it throughout the course. A Modal is made up of two required pieces of code. The first is a button, and here's the basic code required to render it:

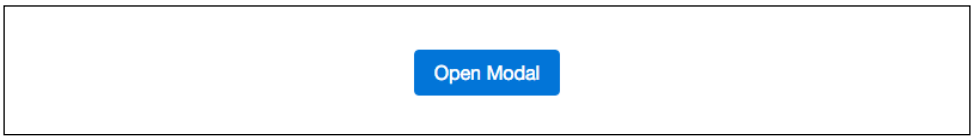
```
<button type="button" class="btn btn-primary" data-toggle="modal"
data-target="#firstModal">
  Open Modal
</button>
```

As you can see, this is a basic Button component with a few attributes added to it:

1. The first is the data-toggle data attribute, which needs to be set to modal. This tells the browser that this <button> is attached to a Modal component.
2. The second is the data-target attribute, which should be an ID. It doesn't really matter what you name this, I've called it #firstModal. It's important to note this ID name as it will be tied in later. Also make sure that the ID name

is unique.

Once you've coded this up, it should look like a regular button in the browser:



Coding the Modal dialog

The second part of the Modal component is the dialog. This is the part that will pop up in the browser once you click the button. Let's take a look at some basic code for creating the dialog:

```
<div class="modal fade" id="firstModal" tabindex="-1" role="dialog"
aria-hidden="true">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <button type="button" class="close" data-dismiss="modal" aria-
label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
        <h4 class="modal-title">Dialog Title</h4>
      </div>
      <div class="modal-body">
        Some copy for your modal.
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-
dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save</button>
      </div>
    </div>
  </div>
</div>
```

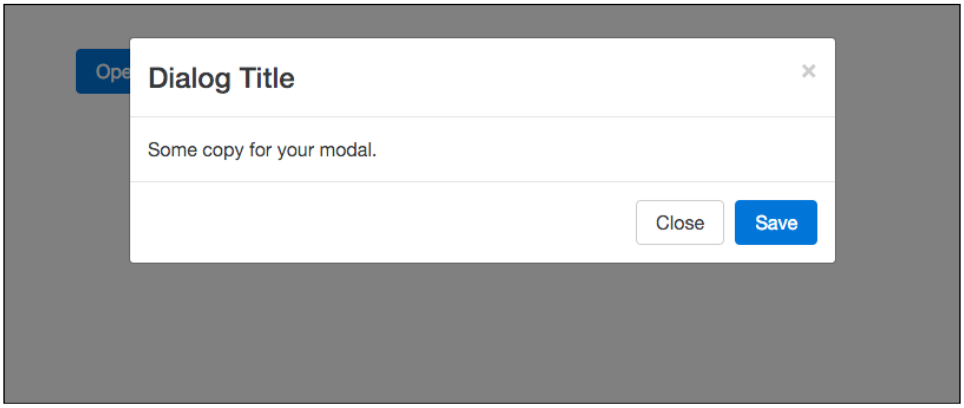
This is a bigger piece of code and there are a few things going on here that I need to explain to you:

- The entire dialog is wrapped in a `<div>` with a required class of `.modal`. There's also an optional .fade class there, which will fade the dialog in. Note the ID on this `<div>` because it's important. The ID value needs to match the

`data-target` attribute you set on the button. This is how we tell the browser to link that button with this dialog. Finally, there are a couple of other attributes that are required by Bootstrap including `tabindex`, `role`, and `aria-hidden`. Make sure you include those with their corresponding values.

- Inside the first `<div>` we have a second one with a class of `.modal-dialog` on it; make sure you include that.
- Next, the interior of the Modal is split into three parts: header, body, and footer.
- Inside our .modal-dialog, add another `<div>` with a class of .modal-header on it. Within this section you'll notice another button. This button is the Close or X icon for the Modal; although not required, it's a good idea to include this.
- After the button you need to include a header tag, in this case a `<h4>`, with a CSS class of .modal-title on it. Here you should enter the title for your Modal.
- The next section is another `<div>` for our body and it has a class of .modal-body on it. Within this section you should enter the body copy for your Modal.
- Finally, we have the footer section, which is another `<div>` with a class of `.modal-footer` on it. Inside this section you'll find two buttons that you need to include. The first is the white button labeled Close which when clicked will close the Modal. Note that the `<button>` tag has a data attribute called data-dismiss on it and its value is modal. This will close the Modal. The second button is a primary button that would be used as a Save button if you were hooking in the actual functionality.

After coding all that up, go to the browser and click on your button. You should then see a Modal that looks like this:



As you can see, our Modal has popped up over the button. You can read the Modal title and body and see the footer buttons as well as the Close or X button in the top-right corner. You may have noticed that you didn't actually have to write any JavaScript to make this Modal work. That is the power of the Bootstrap framework; all of the JavaScript is already written for you and you can simply call the Modal functionality by using the HTML data attributes, which makes things much easier. That concludes the lesson on Modals; next let's move on to learning how to use Tooltips.

Coding Tooltips

A Tooltip is a marker that will appear over a link when you hover over it in the browser. They are pretty easy to add with data attributes in Bootstrap, but we do need to make some updates to get them working. In Bootstrap 4 they have started using a third-party JavaScript library for Tooltips called Tether. Before we go any further, head over to the Tether website below and download the library:

`http://github.hubspot.com/tether/`

Once you've downloaded the library, unzip it and open the main directory where you'll see a number of files. Navigate to the `/dist/js` directory and find the file named `tether.min.js`:

Now `copy tether.min.js` into the `/js` directory of our blog project. This is the only file you need from Tether's directory, so you can keep the rest of the files or delete them. Once the file is in our project directory we need to update our template.

Updating the project layout

Now that we have the Tether file in our project directory we need to update our `_layout.ejs` template to include it when the page is compiled. From the root of our project directory, open up `_layout.ejs` and insert the following line of code near the bottom after jQuery. It's critical that the Tether file is loaded after jQuery, but before `bootstrap.min.js`:

```
<script src="js/tether.min.js"></script>
```

Save the file and make sure you recompile your project so that this is imported into all of your HTML files. Once that's done, you will now be able to use Tooltips on any page that is included in our project.

How to use Tooltips

Now that we've included the Tether library, we can learn how to actually use Tooltips in Bootstrap. Let's try them out on one of our project files. Open up `index.ejs` in your text editor and find a section of code that is just text, like this:

```
<p>Pellentesque habitant morbi tristique senectus et netus et
malesuada fames ac turpis egestas...</p>
```

Once you've found that section of code, let's wrap an `<a>` tag around the first three words with the following attributes on it:

```
<p><a href="#" data-toggle="tooltip" >Pellentesque habitant morbi</a>
tristique senectus et netus et malesuada fames ac turpis egestas.</p>
```

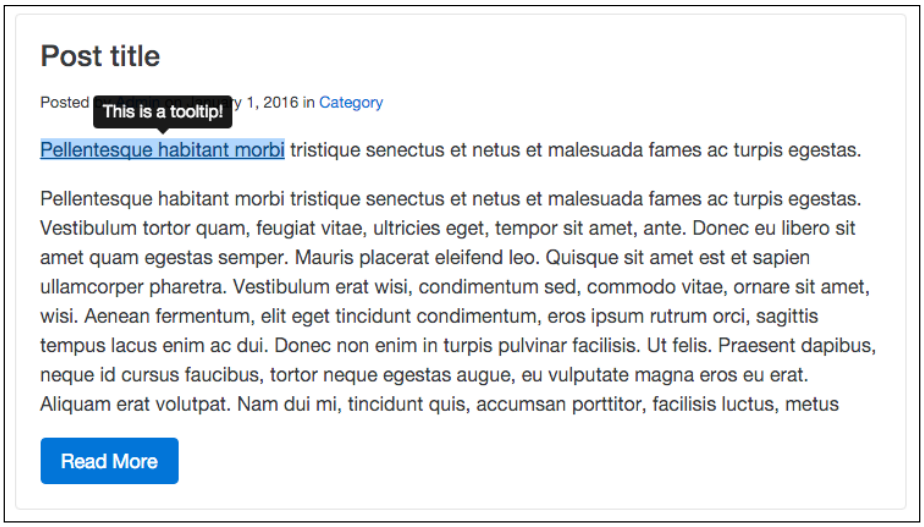
This is the basic markup needed to render a Tooltip. Let's breakdown what is happening here:

- The data-toggle attribute is required to tell the browser that this is a Tooltip. The value should be set to tooltip.
- The title attribute is also required and the value will be the text that appears in your Tooltip. In this case, I have set it to This is a tooltip!.

Before we can test this out in the browser, we need to add something else to our `_layout.ejs` template. Open that file in your text editor and insert the following code after the Tether library:

```
<script src="js/bootstrap.min.js"></script>
<script>
  $("a").tooltip();
</script>
```

In Bootstrap 4, Tooltips need to be initialized before you can use them. Therefore, I'm using a little jQuery here to say that all `a` tags should be initialized to use the Tooltip method, which will activate all link tags for use with a Tooltip. This is a little trick you can use so you don't have to use an ID to indicate every Tooltip you want to initialize. Once you've completed this step, save all your files, recompile them, and then view your project in the browser; it should look like this when you rollover the link anchor text:



How to position Tooltips

By default, in Bootstrap the position for Tooltips is above the anchor text. However, using the data-placement attribute will allow you to place the tip above, below, left, or right of the anchor text. Let's take a look at the code required to render the different versions:

```
<p><a href="#" data-toggle="tooltip"  data-
placement="top">Pellentesque habitant morbi</a> tristique senectus et
netus et malesuada fames ac turpis egestas.</p>
<p><a href="#" data-toggle="tooltip"  data-
placement="bottom">Pellentesque habitant morbi</a> tristique senectus
et netus et malesuada fames ac turpis egestas.</p>
<p><a href="#" data-toggle="tooltip"  data-
placement="right">Pellentesque habitant morbi</a> tristique senectus
et netus et malesuada fames ac turpis egestas.</p>
<p><a href="#" data-toggle="tooltip"  data-
placement="left">Pellentesque habitant morbi</a> tristique senectus et
netus et malesuada fames ac turpis egestas.</p>
```

As you can see, I've added the `data-placement` attribute to each link tag. The following values will control the position of the Tooltip when you hover over it:

- Top: `data-placement="top"`
- Bottom: `data-placement="bottom"`
- Right: `data-placement="right"`
- Left: `data-placement="left"`

Adding Tooltips to buttons

It's also quite easy to add a Tooltip to a button by using the same data attributes as links. Let's take a look at how to code a simple button with a Tooltip above it:

```
<button type="button" class="btn btn-primary" data-toggle="tooltip"
data-placement="top" data-original->This is a button tooltip!</button>
```

Here you'll see a basic button component, but with the Tooltip data attributes:

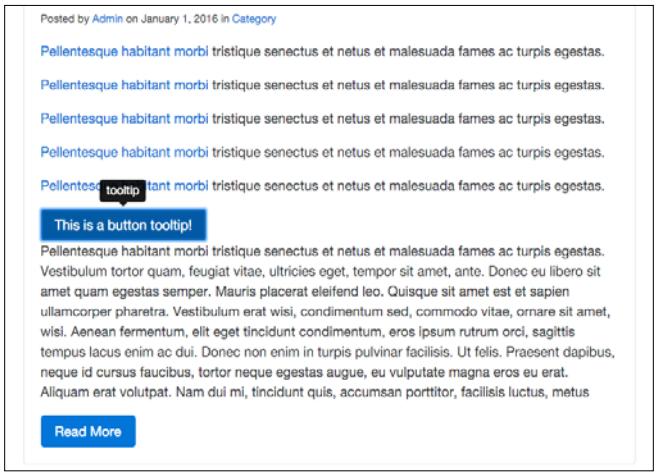
- I've added the `data-toggle` attribute with a value of tooltip
- You can optionally include the `data-placement` attribute; if you leave it out it will default to top
- You need to include the `data-original-title` attribute and the value will be the Tooltip message

Updating the layout for buttons

To get Tooltips on buttons working, you need to initialize them the same way you did the links in the previous section. Open up `_layout.ejs` again in your text editor and include the following line of code. The entire section of JavaScript should now look like this:

```
<script>
  $("a").tooltip();
  $("button").tooltip();
</script>
```

Like we did with the link tags, we'll initialize all button tags to use the Tooltip component if called in the HTML template. Let's take a look at how our Tooltip on a button should look in the browser when it's done correctly:



Avoiding collisions with our components

Until now we've only used the Tooltip JavaScript component so our code is solid. However, in the next section, we will introduce a different component called Popovers. We need to do some clean-up of our JavaScript code so that the two don't collide with each other and give us unwanted results.

Since this is the case, we should go back to `_layout.ejs` and edit the code by providing a specific ID for each Tooltip that you want to use in your project. Our script should now look like this:

```
<script>
  $("#tooltip-link").tooltip();
  $("#tooltip-button").tooltip();
</script>
```



Note that I removed the a and button selectors and replaced them with IDs named #tooltip-link and #tooltip-button. Now we also need to update our link and button code on the index template to include these IDs.

```
<p><a id="tooltip-link" data-toggle="tooltip" >Pellentesque habitant
morbi</a> tristique senectus et netus et malesuada fames ac turpis
egestas.</p>

<button type="button" id="tooltip-button" class="btn btn-primary"
data-toggle="tooltip" data-placement="top" data-original->This is a
button tooltip!</button>
```

As you can see, I've included the ID for each element in the preceding code. Now we are safe to start introducing new components without any worry of collisions occurring in the JavaScript. Let's move on to the component in question; Popovers.

Using Popover components

Popover components are similar to Tooltips but allow for more content to be included. Popovers are also revealed on a click action, not a hover action like Tooltips. Let's take a look at the basic code to render a Popover. First, let's make sure we add this Popover to our project, so open up `index.ejs` again and find another filler line of code to add this new component. When you do, enter the following code into the template:

```
<p><a id="popover-link" data-toggle="popover"  data-content="This is
the content of my popover which can be longer than a tooltip">This is
a popover</a>. Pellentesque habitant morbi tristique senectus et netus
et malesuada fames ac turpis egestas. Vestibulum tortor quam, feugiat
vitae, ultricies eget, tempor sit amet, ante.</p>
```

As you can see, there are a few new things we need to go over here:

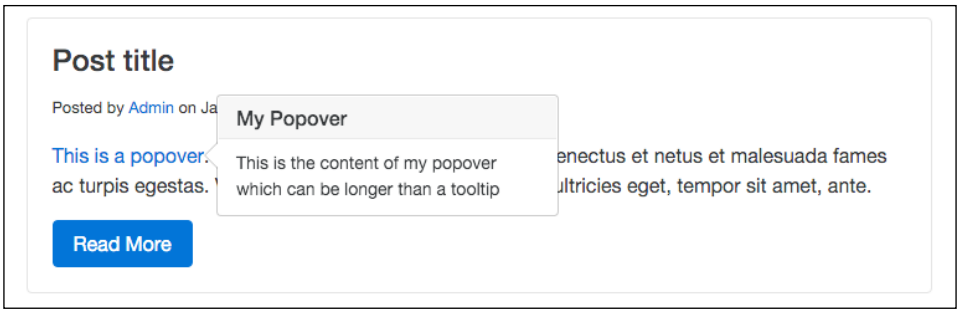
- First of all, you'll notice I've given the link tag this ID; `popover-link`.
- In this case, `data-toggle` is set to `popover`.
- The `title` attribute is required and will be the title for your Popover.
- Finally, we have a new attribute named `data-content`. The value for this should be the copy you want to appear on the Popover.

Updating JavaScript

Like we did with Tooltips, we also need to update the JavaScript for this new component. Open up `_layout.ejs` again and insert the following line of code after the Tooltip JavaScript:

```
$("#popover-link").popover();
```

This code will initialize a Popover component on the element with the `#popover-link` ID on it. Once you've completed that, save both files and go to your browser. Find the link you created for the Popover and click it. This is what you should see in the browser:



As you can see, the Popover component has more to it than the Tooltip. It includes a title and content. You should use this component if you need to give more context than can be achieved through the use of a regular Tooltip.

Positioning Popover components

Again, like Tooltips, it is possible to control the position of a Popover component. This is done in the same way by using the `data-placement` attribute on the link tag. Here's the code for each variation:

```
<p><a id="popover-link" data-placement="top" data-toggle="popover"
data-content="This is the content of my popover which can be longer
than a tooltip">This is a popover</a>. Pellentesque habitant
morbi...</p>

<p><a id="popover-link" data-placement="bottom" data-toggle="popover"
data-content="This is the content of my popover which can be
longer than a tooltip">This is a popover</a>. Pellentesque habitant
morbi...</p>

<p><a id="popover-link" data-placement="right" data-toggle="popover"
data-content="This is the content of my popover which can be longer
than a tooltip">This is a popover</a>. Pellentesque habitant
morbi...</p>

<p><a id="popover-link" data-placement="left" data-toggle="popover"
data-content="This is the content of my popover which can be longer
than a tooltip">This is a popover</a>. Pellentesque habitant
morbi...</p>
```

Since this works in exactly the same way as for Tooltips, I won't bother breaking it down any further. Simply include the `data-placement` attribute and give it one of the four positioning values to control where the Popover appears when clicked.

Adding a Popover to a button

A Popover component can also be easily added to a button. Open up the index template again and insert the following button code:

```
<p><button type="button" id="popover-button" class="btn btn-primary"
data-toggle="popover"  data-content="This is a button popover
example">Popover Button</button></p>
```

As you can see, this markup is very similar to the Tooltip button. Let's break it down again:

- The button tag needs an ID of popover-button to be added
- As with the link, set the data-toggle attribute to popover
- Include a value for title and the data-content attribute

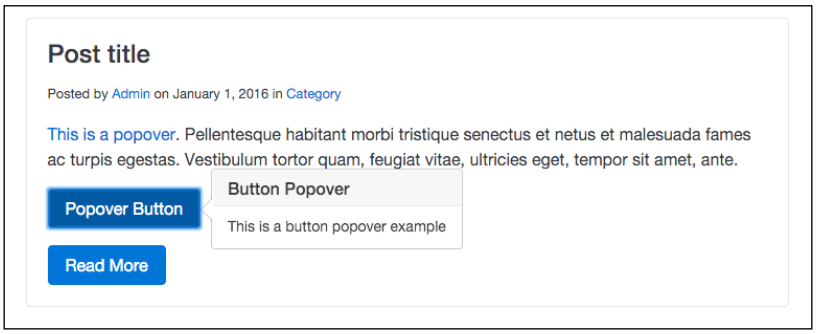
As with the previous examples, don't forget to update the JavaScript!

Adding our Popover button in JavaScript

The last thing we need to do is update the JavaScript to initialize our new Popover button. Open up `_layout.ejs` and insert the following line of code after the Popover link JavaScript:

```
$("#popover-button").popover();
```

Once that is complete, save both files and open up the index page in your browser. Locate the button you inserted and click it. Your Popover should look like this:



As you can see, you now have a button with a Popover component attached to it. This can be useful for calling out something important with a button, and then once it has been clicked it reveals a message to your users. I have a couple more JavaScript components I would like to review with you; the next one is the Collapse component.

Using the Collapse component

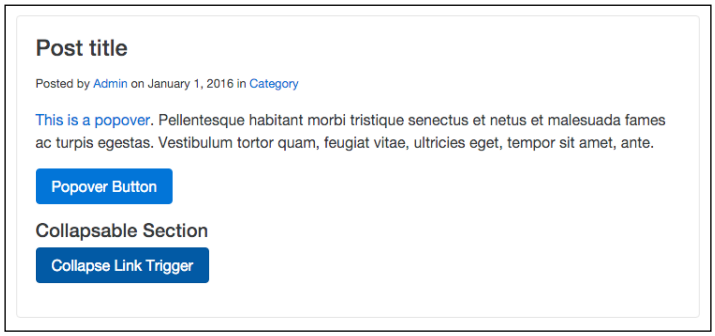
I find that the Collapse component's name is a bit confusing. What it really means is a collapsable section that can be shown or hidden on a click action. Let's start by creating a simple collapsable section of text on the `index.ejs` template. Open that template and insert the following code wherever you like:

```
<p><a class="btn btn-primary" data-toggle="collapse" href="#collapse-
link" aria-expanded="false">Collapse Link Trigger</a></p>
```

The Collapse component is broken into two parts. The first is the trigger to show or hide the collapsable content. The second is the actual content you want to show or hide. Let's review it in more detail to show how to code this up:

- The first part is the trigger for the collapsable content, and I have chosen to use a link that has some button classes on it
- The link requires the `data-toggle` attribute with a value of `collapse` on it
- The `href` for the link needs to be a unique ID name, in this case, `#collapse-link`
- Finally, we set the `aria-expanded` value to `false` because we want the collapsable content to be hidden on page load

On page load, your new component should just appear like a regular button:



Coding the collapsible content container

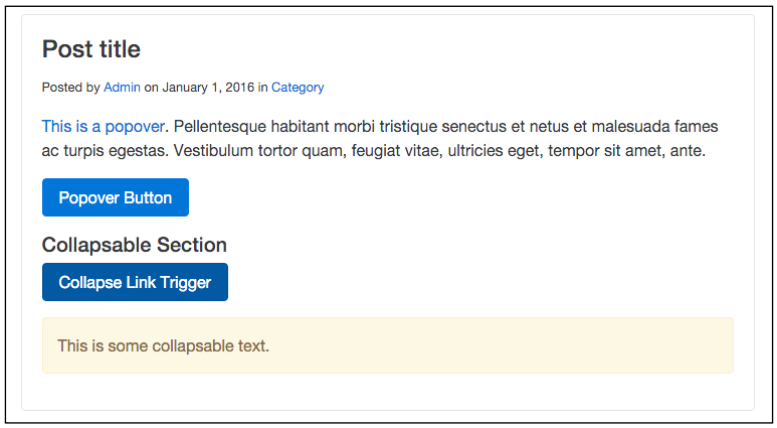
Now that the trigger for the Collapse is set up, we need to code the content container. After the link tag, insert the following code:

```
<div class="collapse" id="collapse-link">
  <p class="alert alert-warning">This is some collapsable text.</p>
</div>
```

Here's how to assemble this section of code:

- We start with a `<div>` that needs to have a CSS class of `collapse` on it. You also need to include an ID here. This should match the ID you set as the `href` in the trigger link; in this case, `#collapse-link`.
- Within the `<div>` you can include any content you want. This content will be the hidden, collapsable content that you will show or hide when the trigger is clicked. To make the example really obvious, I've wrapped a warning Alert around some text to make it stand out.

After you've coded this up and saved your file, head to the browser, find the button, and click it. You should see the following in your window once you click the trigger link:



This is only a simple example of how you can code up the Collapse component. Using additional code and setup, you can use this component to create an Accordion.

Coding an Accordion with the Collapse component

In the previous section, I taught you a pretty simple way to use the Collapse component. The same component can be used to create a more complex version, which is the Accordion. Let's take a look at the basic code to create an Accordion:

```
<div id="accordion">
  <div class="panel panel-default">
    <div class="panel-heading" role="tab" id="headerOne">
      <h4 class="panel-title">
        <a data-toggle="collapse" data-parent="#accordion"
href="#sectionOne" aria-expanded="true" aria-controls="sectionOne">
          Section #1
        </a>
      </h4>
    </div>
    <div id="sectionOne" class="panel-collapse collapse in"
role="tabpanel" aria-labelledby="headerOne">
      This is the first section.
    </div>
  </div>
  <div class="panel panel-default">
    <div class="panel-heading" role="tab" id="headerTwo">
      <h4 class="panel-title">
        <a class="collapsed" data-toggle="collapse" data-
parent="#accordion" href="#sectionTwo" aria-expanded="false" aria-
controls="sectionTwo">
          Section #2
        </a>
      </h4>
    </div>
    <div id="sectionTwo" class="panel-collapse collapse"
role="tabpanel" aria-labelledby="headerTwo">
      This is the second section.
    </div>
  </div>
  <div class="panel panel-default">
    <div class="panel-heading" role="tab" id="headerThree">
      <h4 class="panel-title">
        <a class="collapsed" data-toggle="collapse" data-
parent="#accordion" href="#sectionThree" aria-expanded="false" aria-
controls="sectionThree">
          Section #3
```

```
          </a>
        </h4>
      </div>
      <div id="sectionThree" class="panel-collapse collapse"
role="tabpanel" aria-labelledby="sectionThree">
       This is the third section.
      </div>
    </div>
</div>
```

Now that might look like a ton of code, but it's actually a repeating pattern that is pretty easy to put together once you understand it. Let me breakdown everything that is happening here:

- The entire component is wrapped in a `<div>` with an ID on it. In this case, I'm using `#accordion`.
- Each section of the Accordion is a `<div>` with a class of `.panel` on it. I've also included the `.panel-default` class to just do the most basic styling.
- Each panel is made up of a heading and a body or section. Let's cover the header first. Create another `<div>` with a class of `.panel-heading` on it. Also include the role attribute with a value of tab and you need to give your header a unique ID, in this case, `#headerOne`.
- Inside the header include a header tag, in this case, a `<h4>` , with a class of `.panel-title`.
- Finally, nested inside the header tag, code a link that has a few attributes that you need to include:
    - `.collapsed` is required for the Accordion component.
    - `data-toggle` is also required.
    - `data-parent` should be the same ID that you set on the first `<div>` for the accordion.
    - `href` will be a link to the body of the section that will be collapsable. In this case, it is called `sectionOne`.
    - `aria-expanded` should be set to true because we want this section to be open on page load. The other links should be set to false, unless you want them to be open on page load.
    - `aria-controls` should also match the ID name of the corresponding section.
    - Now that the header has been broken down, let's cover the body of the panel.
- After the header, insert another `<div>` with an ID of `#sectionOne` on it. It should also have a class of `.panel-collapse` and `.collapse` on it. Include

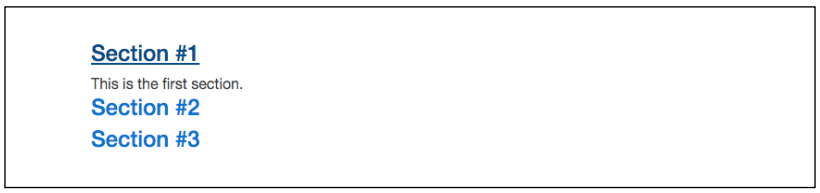
the attribute role with a value of `tabpanel` on it. Finally, include `aria-labelled` by attribute with the value of `sectionOne`.

- Inside this `<div>` include the content of the section that you want to display.

For the next sections, you need to repeat what you did for the first panel. Simply copy and paste and then you need to change a few things:

- Change `headerOne` to `headerTwo`
- Change `sectionOne` to sectionTwo
- Change up the header title and content of the body for the second section

Do the same for the third section, and then the Accordion component is done. Once you're done, this is what it should look like in the browser:



That completes the Collapse and Accordion components. We have one more to go, which is the Carousel component.

Coding a Bootstrap Carousel

Carousel is a popular component used on many different types of websites. We're going to build a Carousel in the Blog Post template of our project. Let's start by opening up `blog-post.ejs` from the project directory in your text editor. After the page title block of code, insert the following markup:

```
<div id="carousel-example-generic" class="carousel slide" data-
ride="carousel">
  <ol class="carousel-indicators">
    <li data-target="#carousel-example-generic" data-slide-to="0"
class="active"></li>
    <li data-target="#carousel-example-generic" data-slide-to="1"></
li>
    <li data-target="#carousel-example-generic" data-slide-to="2"></
li>
  </ol>
  <div class="carousel-inner" role="listbox">
    <div class="carousel-item active">
      <img src="..." alt="First slide">
```

```
      </div>
      <div class="carousel-item">
        <img src="..." alt="Second slide">
      </div>
      <div class="carousel-item">
        <img src="..." alt="Third slide">
      </div>
    </div>
    <a class="left carousel-control" href="#carousel-example-generic"
role="button" data-slide="prev">
      <span class="icon-prev" aria-hidden="true"></span>
      <span class="sr-only">Previous</span>
    </a>
    <a class="right carousel-control" href="#carousel-example-generic"
role="button" data-slide="next">
      <span class="icon-next" aria-hidden="true"></span>
      <span class="sr-only">Next</span>
    </a>
  </div>
```

This is a larger component like the Accordion so let's go through it section by section:

The Carousel component starts with a `<div>` and it needs a unique ID. In this case, `#carouselOne`. Also include the following classes: `.carousel` and `.slide`. Finally, you need to add the attribute `data-ride` with a value of `carousel`.

Adding the Carousel bullet navigation

The first thing we need to add to the Carousel is the bullet or indicator navigation. It's made up of an ordered list. Here's the code, then we'll break it down:

```
<ol class="carousel-indicators">
  <li data-target="#carouselOne" data-slide-to="0" class="active"></
li>
  <li data-target="#carouselOne" data-slide-to="1"></li>
  <li data-target="#carouselOne" data-slide-to="2"></li>
</ol>
```

Here's how the Carousel navigation works:

- On the `<ol>` tag allocate a class of `.carousel-indicators`.
- Each `<li>` in the list needs to have a few things:
    - The `data-target` needs to be the same ID that you gave to your root Carousel , in this case, `#carouselOne`.

- Include the `data-slide-to` attribute and the first value should be 0. Increase it by one for each list item after the first.

Including Carousel slides

The next step is to include the actual Carousel slides. I'm not going to include images in the code, that will be up to you to insert, but don't worry, I'll show you where to put them. Here's the code for the section that wraps the slides:

```
<div class="carousel-inner" role="listbox">
  ..
</div>
```

Give that a class of .carousel-inner and add the role attribute with a value of listbox. Inside this you're going to add another section for each image slide in the Carousel. Here's the code for one slide in the Carousel:

```
<div class="carousel-item active">
  <img src="..." alt="First slide">
</div>
```

Let's breakdown what's happening here in the code:

- In this case, insert a `<div>` tag with the classes `.carousel-item` and `.active`



Note you should only include the `.active` class on the first slide. This is where the Carousel will start on page load.

- Inside the `<div>`, insert an `img` tag with the following attributes:
    - Insert the `src` attribute and the value should be the path to the image file for the slide
    - Optionally, include an `alt` attribute with a value for the image

Adding Carousel arrow navigation

The last thing we need to add to the Carousel is the arrow navigation. Here's the code for rendering the arrows:

```
<a class="left carousel-control" href="#carouselOne" role="button"
data-slide="prev">
  <span class="icon-prev" aria-hidden="true"></span>
  <span class="sr-only">Previous</span>
```

```
</a>
<a class="right carousel-control" href="#carouselOne" role="button"
data-slide="next">
  <span class="icon-next" aria-hidden="true"></span>
  <span class="sr-only">Next</span>
</a>
```

Let me explain how the arrow navigation works:

- The left and right arrow navigation is based on `href` tags.
- The first will be the left arrow; code a link with the following classes on it: `.left` and `.carousel-control`.
- The `href` for the link should be set to the main ID for the Carousel, in this case, `#carouselOne`.
- Set the `role` attribute to `button`.
- Finally, set the `data-slide` attribute to `prev`.
- Within the link, add a `<span>` with a class of `.icon-prev` on it. This will render the arrow icon. Include the `aria-hidden` attribute and set it to true.
- Lastly, you can include another optional `<span>` for accessibility reasons. If you want to include it, give it a class of `.sr-only`. Within the `<span>` include the text `Previous`.
- Now let's go over the differences for the right arrow:
    - Code another link tag and switch the `.left` class to `.right`.
    - Change the `data-slide` attribute value to `next`.
    - In the first `<span>` tag change the class value to `.icon-next`.
    - If you included the accessibility `<span>` tag change the text to Next.

That completes the setup of the Carousel component. Fire up the project server and view the Blog Post page in the browser, and it should look like this:



That concludes the lesson on JavaScript components in Bootstrap. In this lesson, I taught you how to code up the following components: Modals, Tooltips, Popovers, Collapse, Accordion, and the Carousel. In the next lesson, I'll teach you how to use Sass in Bootstrap.

Summary

In this lesson, we have covered all components in Bootstrap that rely on JavaScript. This included: Modals, Tooltips, Popovers, Collapse, and Carousel.

In the next lesson, we will see how in Bootstrap 4 the framework has moved from Less to Sass as its CSS preprocessor. We will cover the basics of using Sass in a Bootstrap theme. I'll also explain how you can customize or use existing variables, or write your own.

Assessments

1. Which among the following can also be called as modals?
    i. Dialogs
    ii. Popups
    iii. Overlays
    iv. All of the above
2. Which among the following data attributes needs to be set to modal?
    i. `data-target`
    ii. `data-toggle`
    iii. `data-type`
    iv. `data-value`
3. Which among the following values is assigned to `data-target` attribute of a button for displaying a modal?
    i. Modal ID
    ii. Modal type
    iii. Modal class
    iv. Modal name
4. Which among the following is a required class for modal?
    i. `.modal-header`
    ii. `.modal-footer`
    iii. `.modal-body`
    iv. `.modal`
5. Which among the following data attributes is used to position a Tooltip?
    i. `data-target`
    ii. `data-placement`
    iii. `data-value`
    iv. `data-toggle`

7
Throwing in Some Sass

Up until now we've covered a bunch of different Bootstrap components and how to use them. In this lesson, we're going to change gears and learn about Sass, which will allow you to customize the look and feel of your components. I'll start by introducing some Sass basics that you need to know, move on to writing some basic code, and then show you the power of using variables in your components to save yourself valuable time when creating your web app or project.

Learning the basics of Sass

Sass stands for **Syntactically Awesome Style Sheets**. If you've never used or heard of Sass before, it's a CSS preprocessor. A preprocessor extends regular CSS by allowing the use of things such as variables, operators, and mixins in CSS. Sass is written during the development stage of your project and it needs to be compiled into regular CSS before you deploy your project into production. I'll cover that in more detail in the next section but don't worry because Harp.js makes this really easy to do.

Up until version 4 of Bootstrap, the CSS preprocessor used was actually Less. For a good while both Sass and Less were popular in frontend design circles. However, over the last few years, while Sass has emerged as the best choice for developers, the Bootstrap team decided to make the change in version 4. If you are familiar with Less but have never used Sass, don't worry as they are pretty similar to use so it won't take much to get you up-to-speed.

Using Sass in the blog project

As I mentioned in the previous section, Sass is part of the development process and the browser cannot read it in its native format. Before you can deploy the project, you need to convert or compile the Sass files into regular CSS files. Normally this

would require you to install a Ruby gem and you would have to manually compile your code before you can test it. Luckily for us, Harp.js actually has a Sass compiler built into it. So when you run the `harp compile` command to build your templates, it will also build your Sass files into regular CSS. I bet you're starting to like Harp even more after learning that.

Updating the blog project

Before we go any further, we need to make a few updates to our blog project to set it up for Sass. Head to your project directory and navigate to the CSS directory. In this directory, create a new file called `custom.scss`.



The file extension used for Sass files is `.scss`.

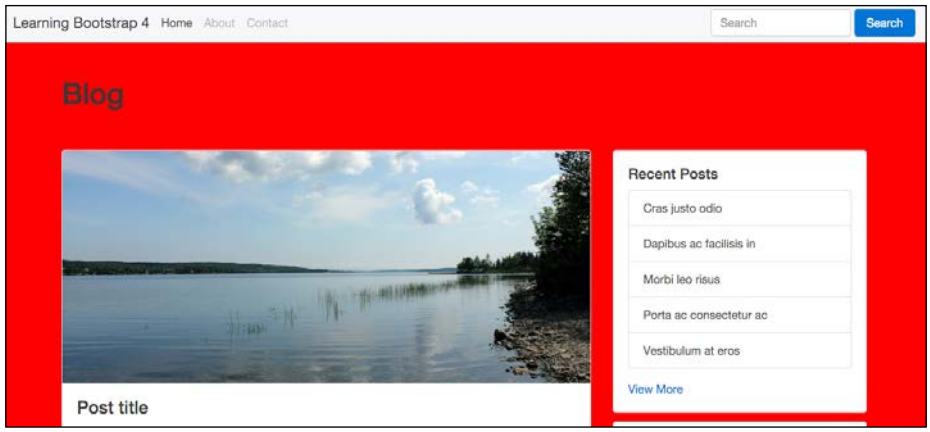
What we're doing here is creating a custom style sheet that we are going to use to overwrite some of the default Bootstrap look-and-feel CSS. To do this, we need to load this custom file after the Bootstrap framework CSS file in our layout file. Open up `_layout.ejs` in the root of the project directory and insert the following line of code after `bootstrap.min.css`. Both lines together should look like this:

```
<link rel="stylesheet" href="css/bootstrap.min.css">
<link rel="stylesheet" type="text/css" href="css/custom.css">
```

Note here that I'm using the `.css` file extension for `custom.css`. This is because, after the files are compiled, the template will be looking for the actual CSS file, not the Sass file. The critical part is just that the actual filenames match and that you use `.css` in the layout file. Before we go any further, let's test out our Sass file to make sure it is set up properly. Open up `custom.scss` in your text editor and add the following code:

```
body {
   background: red;
}
```

This is just a simple way to make sure that Sass is compiling to CSS and is being inserted into our layout. Compile your project and launch the server. If you've done everything correctly the background for your homepage should be red and look like this:



Hopefully this is what you're seeing and you can confirm you've set up your file correctly. Once you've successfully done this, delete the CSS we entered in the Sass file.



It's perfectly acceptable to write regular CSS in Sass files. Ideally, you want to combine regular CSS code with Sass syntax to take full advantage of the preprocessor.

Now that you've finished setting up your files, let's start to learn a little bit more about using Sass in your project.

Using variables

In Sass, variables are called by using the `$` sign character. If you're familiar with Less, the `@` symbol is used for variables. So in that case, all you would need to do is use `$` instead of `@`. To write a variable, start with the `$` sign and then insert a descriptive keyword that can be anything you like. Here are a few examples of generic variable names:

```
$background-color
$text-size
$font-face
$margin
```

I've named these pretty generically and they actually match some CSS property names. This is a good idea and they are easy to reuse and make sense of if multiple developers are working on the same project. However, like I said, you can name your variables whatever you want. If you'd like to get more creative, you could name

variables like this:

```
$matts-best-color
$awesome-background-color
$fantastic-font-face
```

These are extreme examples and it is advisable not to name your variables in this way. To you `$awesome-background-color` might mean red but to another person it could mean anything. It's always a good idea to name your variables in a descriptive manner that makes sense.

I've shown you how to write the variable name but the other side of the equation is the actual value for the variable. Let's add in some sample values for our first set of variable names:

```
$background-color: #fff;
$text-size: 16px;
$font-face: helvetica, sans-serif;
$margin: 1em;
```

You write Sass variables the same way that you would write CSS properties. It's also worth noting that you should enter your variables at the very top of your style sheet so that they can be used in all of the CSS you write after them.

Using the variables in CSS

Now that we've written some variables, let's actually insert them into some CSS. After the variables in `custom.scss`, enter the following code:

```
body {
   background: $background-color;
   font-size: $text-size;
   font-family: $font-face;
   margin: $margin;
}
```

So instead of using actual values for our CSS properties, we're using the variable names that we set up. This starts to get more powerful as we add more CSS. Let's reuse some of these variables:

```
body {
   background: $background-color;
   font-size: $text-size;
   font-family: $font-face;
   margin: $margin;
}
```

```
h1 {
   font-size: 36px;
   font-family: georgia, serif;
}

h2 {
   font-size: $text-size;
   font-family: $font-face;
}
```

In this example, you can see a few things going on that I should explain:

- For the `<h1>` tag, I'm not using any variables. I'm using regular CSS property values.
- For the `<h2>` tag, I'm reusing the same variables to insert the `font-size` and `font-family` values.

As your style sheet grows longer, I'm sure you'll see the value in this strategy. For example, if I decide I want to change my `font-size` to `24px`, all I need to do is change the value for the `$text-size` variable to `24px`. I don't have to go through my entire style sheet and change all the values individually. These are just the basics of what you can do with variables. Let's look at a more advanced use case.

Using other variables as variable values

That might sound like a bit of a mouthful, but you can actually use a variable as the default value for another variable. A good example of where you might want to do this is when you are defining a color palette. You can switch the hex values to readable names and then use them for your other variables. This is much easier to scan and understand when you are debugging your code. Here's an example of what I mean:

```
$black: #000;
$white: #fff;
$red: #c00;

$background-color: $white;
$text-color: $black;
$link-color: $red;
```

Let me break down what is happening here for you:

- First I've created three color variables for `black`, `white`, and `red`

- Next I've created three CSS property variables for `background-color`, `text-color`, and `link-color`; the values for these CSS property variables are the color variables

Instead of using hex number values for the CSS property variables, I used a color keyword variable which is much easier to read and understand. That concludes the introduction to variables in Sass. Next we'll learn about importing different files into `custom.css` and using partials.

Importing partials in Sass

Just as you can do in Harp.js, you can use partials in Sass. If you've forgotten what a partial is, it's a little snippet of code that is saved into a different file and then imported into the main CSS theme or layout, in the case of Harp. This can be handy for making your CSS modular and easier to manage. For example, it would make a ton of sense to break every Bootstrap component into its own CSS file and then use the `@import` directive to bring them all into a single master theme, which is then included in your project. Let's go over an example of how you could do this for a single component. In your project, go to the `/css` directory and create a new sub-folder called `/components`. The full path should be:

```
/css/components
```

In the `/components` directory, create a new Sass file and name it `_buttons.scss`. Make sure you always insert an underscore at the start of the filename of a partial. The compiler will then ignore these files as the underscore means it is being inserted into another file. Enter the following at the top of the file as a marker:

```
/* buttons */
```

Save the buttons file and then open up `custom.scss` and add the following line of code to the file:

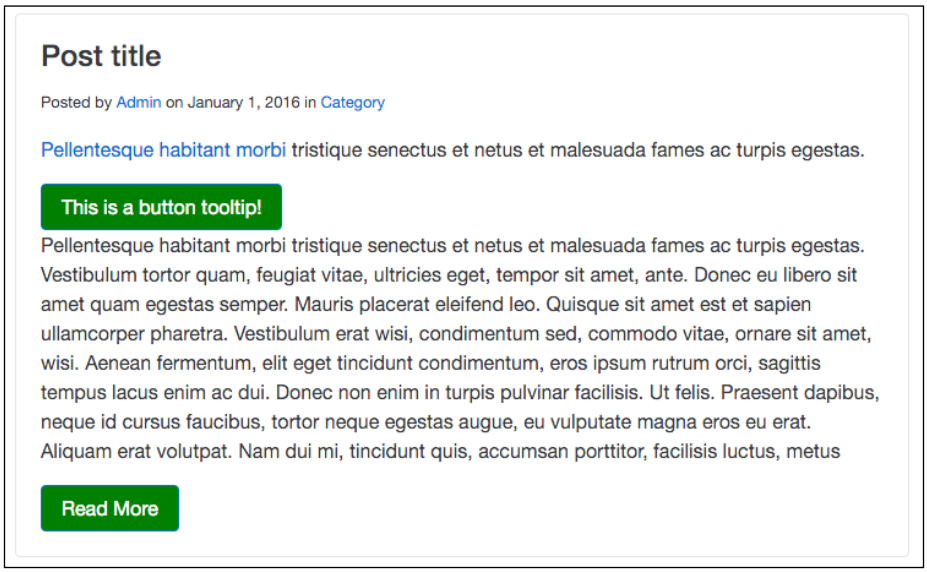
```
@import "components/_buttons.scss";
```

That line of code uses the `@import` rule, which will allow us to import the `_buttons.scss` file into our main theme file that we are calling `custom.scss`. As I've mentioned, the reason you need to do this is for maintainability. This makes the code much easier to read and to add/remove components, which is just another way of saying it makes it more modular.

Before we can test this out to make sure it works, we need to add some code to our `_buttons.scss` file. Let's add some simple CSS to change the primary button as an example:

```
.btn-primary {
  background-color: green;
}
```

After adding this code, save the file and do a `harp compile`. Then launch the server and check out the home page; the buttons will be green like this:



After testing that out, you may want to take that custom code out unless you want the buttons to remain green. That's just a simple example of how you can use partials to make your Bootstrap components more modular. I'll get into that topic in greater depth in a future lesson but for now we are going to focus on using Sass mixins.

Using mixins

Writing something in CSS, such as, for example, browser vendor prefixes, can be really tedious. Mixins allow you to group CSS declarations together so that you can reuse them through your project. This is a great because you can include the code for, say, a `border-radius`, using one line of code instead of multiple lines for each browser. To start, open up `custom.scss` and insert the following code at the top of the file:

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}
```

Let's go over a few things that are happening here:

- A mixin is always started in Sass with the `@mixin` keyword
- Following that, you want to include the property name to target as well as set a variable, in this case `$radius`
- We then apply the `$radius` variable to each browser prefix instance

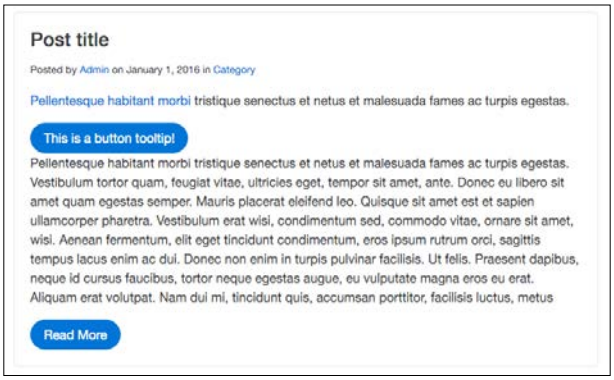
We've set up the mixin to handle the `border-radius` property but we still need to add the corner value to an element. Let's change the `border-radius` value for the default Bootstrap button. Open up `_buttons.scss` and insert the following code:

```
.btn {
  @include border-radius(20px);
 }
```

Let me explain what is happening here:

- I'm targeting all Bootstrap buttons by inserting the `.btn` class
- Inserting the `@include` keyword will grab the `border-radius` mixin
- Lastly, I've provided a value of 20px, which will make our buttons look really rounded on each end

Save your file, run the `harp compile` command, and then, when you view the project in the browser, it should look like this:



That concludes a fairly simple example of how to use mixins in Bootstrap 4. You can use them for many other reasons but replacing CSS3 vendor prefixes is one of the most common and useful. Next we'll cover a slightly more complicated topic in Sass, which is the use of operators.

How to use operators

Sass allows you to perform basic math operations in CSS, which is useful for a number of reasons. First of all, you can use the following operators `+`, `-`, `*`, `/`, and `%`. To give you an understanding of how you can use operators in CSS, let's learn how to convert a pixel-based grid into percentages. We'll create two columns in pixels and then use some Sass to convert them to percentages. Open up `custom.scss` and insert the following code:

```
.left-column {
  width: 700px / 1000px * 100%;
}

.right-column {
  width: 300px / 1000px * 100%;
}
```

Now, I've created two columns here. The `.left-column` class will have a width of 70% after we compile this Sass operator. The `.right-column` class will have a width of 30%. So if we add those together we'll get roughly a three-quarter layout with a larger column on the left and a smaller column on the right. Run a `harp compile` command to build this code and then open up `custom.css` in the `/www/css` folder. There you should find the following code:

```
.left-column {
  width:70%;
}

.right-column {
  width:30%;
}
```

As you can see, our Sass operators have been converted into regular percentage values. That's just one way you can use operators in Sass; I'd encourage you to play around more with them. Next we're going to learn how to set up a library of Sass variables that you can use to create a Bootstrap theme.

Creating a collection of variables

One of the main things you'll want to do when using Sass in Bootstrap is to create a library of global variables that can be used throughout your theme. Think of things such as colors, backgrounds, typography, links, borders, margins, and padding. It's best to only define these common properties once and then you can reuse them through different components. Before we go too far, we need to create a new `.scss` file. Open up your text editor, create a new file, and call it `_variables.scss`. Save that file to the `/css/components` directory. For now, you can just leave it blank.

Importing the variables to your custom style sheet

Now that we've created the variables Sass file, we need to import it into our custom style sheet. Open up `custom.css` in your text editor and paste the following line of code at the top of the file:

```
@import "components/_variables.scss";
```

It's important to note that this file must be at the top of your custom style sheet file. The variables will cascade through all the code that follows them so they must load first. Let's start filling out our variables file with a color palette.

Adding a color palette

Save the custom style sheet and then go back to the variables file. Let's start by inserting a color palette into the variables file like this:

```
$red: #e74c3c;
$red2: #c0392b;
$blue: #3498db;
$blue2: #2980b9;
$green: #2ecc71;
$green2: #27ae60;
$yellow: #f1c40f;
$yellow2: #f39c12;
$purple: #9b59b6;
$purple2: #8e44ad;
$white: #fff;
$off-white: #f5f5f5;
$grey: #ccc;
$dark-grey: #333;
$black: #000;
```

As you can see, I've set up a palette of several colors that I'll use through my components and later my theme. Here are a few key points to keep in mind:

- It's good to have two variations for your key colors. This comes in handy for a component such as a button where `$red` would be the static color and `$red2` would be the hover or active color for the button.
- I'm guessing you can already see how using variable names such as `$purple` is much more readable than hex values in a long style sheet.

Adding some background colors

The next thing you should add to your collection of variables is background colors. As we move through this variables file, we're going to create a variable for all properties that get used over and over again in our style sheet.

Add the following background color variables to the file:

```
$primary-background: $white;
$secondary-background: $off-white;
$inverse-background: $black;
```

Let me explain, as best practice, how I have set this up:

- First of all, I'm using the color variables we just set up as the values for our new background color variables. This keeps things simple and it also allows you to change the color and have it cascade through all your other variables. This is a great time-saving tip.
- At the very least, it's a good idea to define a `primary`, `secondary`, and `inverse` background color variable. Note how I'm reusing the same language here that Bootstrap uses. This is a good practice to follow. Feel free to define additional background colors if you think you'll need them in your project.

Setting up the background color variables is pretty simple. Next let's set up our base typography variables.

Setting up variables for typography

The next section of variables we are going to set up is for the base typography styles. Insert the following code after the background colors:

```
$body-copy: helvetica, arial, verdana, sans-serif;
$heading-copy: helvetica, arial, verdana, sans-serif;
$base-font-size: 16px;
$font-size: 1em;
$base-line-height: 1.75;
```

Let me explain why I'm setting the following variables for the typography:

- For consistency, it's good to have a body and heading typeface. In this case, I'm using the same font stack for both but you could easily change the heading variable to something else. As you are coding your CSS, it's really easy to think of the `font-family` in either the body or heading version, compared with trying to remember the entire font stack for each, which also involves much more typing.
- For the `$base-font-size` variable, we are going to use a pixel value. This is one of the only places you'll see pixels and its set to the base em size that everything else will work off. Remember that ems are a relative sizing unit, so if you ever want to make all your components a little bigger or smaller, you can just tweak this one pixel value.
- We also need a `$font-size` variable, which will be set to `1em`. This is a base unit and it can easily be changed in other selectors by using Sass operators. The reason we set it to 1em is because it simply makes the math easy to do.
- Finally, I set the `$base-line-height` to `1.75` because I like a little extra line spacing in my copy. You could choose to leave this out if you are fine with the Bootstrap default, which is closer to 1.5.

Now that we've set up our typography variables, let's move on to coding our text colors.

Coding the text color variables

As with the background colors, we need to set up some common color styles for text, as well as defining some colors for base HTML tags such as `<pre>` and `<code>`. Insert the following markup after the typography variables in the file:

```
$primary-text: $black;
$light-text: $grey;
$loud-text: $black;
$inverse-text: $white;
$code-text: $red;
$pre-text: $blue;
```

Let me break down how each variable is set up:

- As in the background color variables, we are using a variable name for the value of our text color variables. I've included a variable called `$primary-text` and set it to black, following the same naming convention that was previously established.
- I've added `$light-text` and `$loud-text` variables so we can easily apply lighter or darker text throughout our components.
- I've also included an `$inverse-text` variable to be used with the corresponding background color.
- Finally, I've set up default colors for the `<pre>` and `<code>` tags, which we will use to overwrite the default colors so they match our theme and color palette.

That finishes off the color variables that I recommend setting up. Feel free to add more if you have other uses you want to cover. Next we'll continue with some text colors by adding links.

Coding variables for links

An extension of basic text colors will be colors for links in our project. Go ahead and add the following code after the text colors in the file:

```
$primary-link-color: $purple;
$primary-link-color-hover: $purple2;
$primary-link-color-active: $purple2;
```

In this case, I've decided to only define a primary link color to keep things simple. In your own projects, you will likely want to come up with a couple more variations.

- For the static link color, I'm using the `$purple` color variable.
- For the hover and active states of the primary link, I'm using `$purple2`. As I previously mentioned, this is an example of why it's a good idea to have two variations of each color in your palette.

Like I said, I've kept the `link` variables simple. It's nice to try and keep your set of variables as compact as possible. If you have too many then it starts to defeat the purpose of using them as it will be harder to remember them in your code. Next let's cover the variables we should set up for borders.

Setting up border variables

Another CSS property that gets used often is borders. That makes it a great candidate for Sass variables. Insert the following code after the link colors in the file:

```
$border-color: $grey;
$border-size: 1px;
$border-type: solid;
$border-focus: $purple;
```

Let me explain why I've set up the border variables in this manner:

- When you are deciding on a value for `$border-color`, you should pick a color that you think will get used the most often in your components. Something like `$grey` is always a safe bet in most designs.
- As with the color value, you should set the `$border-size` to the most common border size you anticipate using. It's also a good idea to set this to 1px because you can easily do the math to apply a Sass operator if you want a thinner or thicker border.
- Again for the `$border-type`, set it to the value you will use the most, which is probably going to be solid.
- Finally, set up a common `$border-focus` color. This is primarily used in form inputs once they are active. It's a good idea to pick a contrasting color for this variable so it really stands out when the input is in focus.

That concludes all the border variables I would recommend including. Next let's include some basic layout variables.

Adding variables for margin and padding

For consistent spacing throughout your designs, it's a good idea to use variables for `margin` and `padding` so that you can standardize on size. These properties are also used often so it's smart to make them variables that can be reused. Add the following code after the border markup:

```
$margin: 1em;
$padding: 1em;
```

All I'm doing here is setting a base size (for both `padding` and `margin`) 1em. Again, it's a good idea to set both of these to 1em because it is easy to do the math if you want to use Sass operators to increase or decrease the values of specific components. Those are the last variables that I would recommend adding to your variables file. However, we should add at least one mixin to the file before we are finished.

Adding mixins to the variables file

Since mixins will also be used through a number of your components, you should define them in this variables file. Then they will be available to all the CSS code that follows them in the custom theme file. At the very least, I would recommend setting up a mixin for `border-radius`, which I will show you how to do next. You may also want to include additional mixins for other CSS3 features.

Coding a border-radius mixin

We talked a little bit about mixins earlier but let's review them again now that we are actually applying them to our project. Insert the following code after the layout variables in your file:

```
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}
```

In Less, it is possible to set a global value for all your border-radius in a mixin. However, with Sass you have to set up the above formula but then on the actual selectors that follow you have to set the actual border-radius value. An example of that would look like this:

```
.my-component {
  @include border-radius(5px);
 }
```

In this example, I've added the `border-radius` mixin to a CSS class called `.my-component`. The component will have a `border-radius` of 5px applied to it. You will need to repeat this step on any CSS class or component where you want to apply the `border-radius` mixin. That concludes our variables Sass file. We went over a bunch of code there, so let's see what it all looks like together. I've also included some CSS comments in the following code to help remind you what each section does:

```
/* variables */

/* color palette */
$red: #e74c3c;
$red2: #c0392b;
$blue: #3498db;
$blue2: #2980b9;
$green: #2ecc71;
$green2: #27ae60;
$yellow: #f1c40f;
$yellow2: #f39c12;
$purple: #9b59b6;
$purple2: #8e44ad;
$white: #fff;
$off-white: #f5f5f5;
$grey: #ccc;
$dark-grey: #333;
$black: #000;

/* background colors */
$primary-background: $white;
$secondary-background: $off-white;
$inverse-background: $black;

/* typography */
$body-copy: helvetica, arial, verdana, sans-serif;
$heading-copy: helvetica, arial, verdana, sans-serif;
$base-font-size: 16px;
$font-size: 1em;
$base-line-height: 1.75;

/* text colors */
$primary-text: $black;
$light-text: $grey;
$loud-text: $black;
$inverse-text: $white;
$code-text: $red;
$pre-text: $blue;

/* links */
$primary-link-color: $purple;
$primary-link-color-hover: $purple2;
$primary-link-color-active: $purple2;
```

```
/* border */
$border-color: $grey;
$border-size: 1px;
$border-type: solid;
$border-focus: $purple;

/* layout */
$margin: 1em;
$padding: 1em;

/* border-radius mixin */
@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
     -moz-border-radius: $radius;
      -ms-border-radius: $radius;
          border-radius: $radius;
}
```

Now that we have all our variables and mixins set up, let's go ahead and start to learn how to apply them. We'll continue to build on the button example we started earlier by extending it into a custom look and feel.

Customizing components

Let's first start by customizing a single component; later on I'll talk about creating a theme where you customize all the components in Bootstrap. To get started, we'll build on the button component we started to work on earlier. In this next step we are going to expand on the CSS we have added to fully customize the component. What you want to do is overwrite all the CSS classes and properties that you want to change. In some cases, this might only be a few things but in other scenarios you may want to change quite a bit.

Customizing the button component

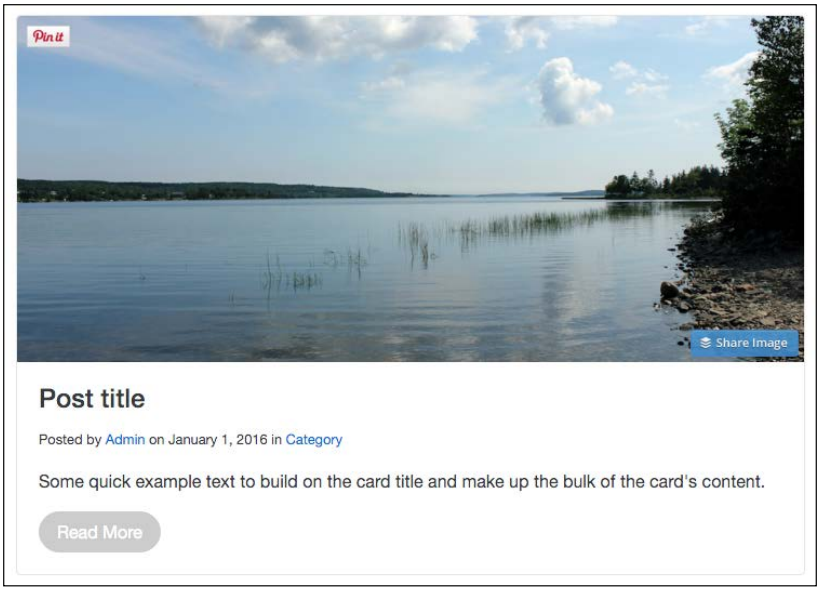
To start, open up `_buttons.scss` located in `/css/components` in our project directory. The first thing we need to customize is the base `.btn` CSS class. Once we have applied some changes there, we'll add more CSS to control the look and feel of the different button variations. Insert the following CSS at the top of the file for the base button class:

```
.btn {
  background-color: $grey;
  border-color: $grey;

  @include border-radius(20px);
}
```

To keep things simple, I'm only going to overwrite a few properties. You're totally free to get more creative and change additional properties to make your buttons look different from the Bootstrap default link. Let's break down what I've done:

- First I've set the `background-color` and `border-color` to use the `$grey` from our color palette. This is a good time to point out that if you want to do a full theme you need to overwrite all the Bootstrap default colors on all components to match your color palette.
- Next I've inserted the `border-radius` mixin and given it a value of `20px`. This will make the buttons really rounded. I'm going for this look so you can clearly see that the button has been customized.

Once you have saved these changes, go to the terminal and run the harp compile command from the root of the project directory. Then fire up the server and open the home page of the project that has a bunch of buttons on it. Your buttons should now look like this:



Now that might not look too useful, but it's important that we customize the base `.btn` class first; now we'll continue building the component out by applying our color palette to all of the different button variations.

Extending the button component to use our color palette

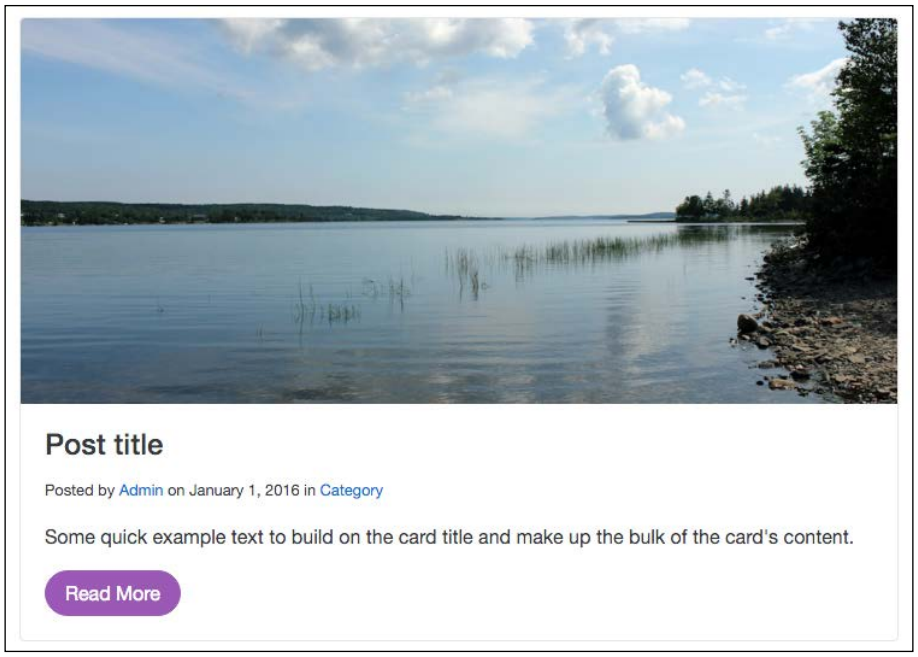
In this next section, we will extend the button component further by applying our color palette to all the different Bootstrap button variations. Before we get to all the different button types, let's start by customizing the `.btn-primary` variation. Enter the following code in the `_buttons.scss` file after the base `.btn` styles:

```
.btn-primary {
   background-color: $purple;
   border-color: $purple;
}

.btn-primary:hover,
.btn-primary:active {
   background-color: $purple2;
   border-color: $purple2;
}
```

There are a few different things going on so let's review them all:

- There are two sections of CSS for each button variation. The first is the static state of the button. The second is the hover and active states of the button.
- For the static state we use the `.btn-primary` class and insert the `background-color` and `border-color` properties. I want to make my primary button purple so I've inserted the `$purple` Sass variable to overwrite the Bootstrap default color.
- For the other states, we have `.btn-primary:hover` and `.btn-primary:active`. In this case, I'm using the second purple color variable which is `$purple2`. On the hover or active button there will be a slightly darker shade of purple.

Save the file, run a `harp compile` in the terminal, and then open up the home page in your browser. If everything was coded correctly, your buttons should now look like this:

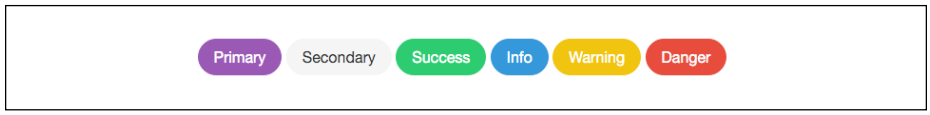


As you can see, the primary button is now purple! It's as simple as that; you can start to apply a custom look and feel to the button component. Let's build out the rest of the button color variations by entering the following code into the `_buttons.scss` file:

```
.btn-secondary {
   background-color: $off-white;
   border-color: $off-white;
}

.btn-secondary:hover,
.btn-secondary:active {
   background-color: $grey;
   border-color: $grey;
}

.btn-success {
   background-color: $green;
   border-color: $green;
}

.btn-success:hover,
.btn-success:active {
```

```
    background-color: $green2;
    border-color: $green2;
}

.btn-info {
    background-color: $blue;
    border-color: $blue;
}

.btn-info:hover,
.btn-info:active {
    background-color: $blue2;
    border-color: $blue2;
}

.btn-warning {
    background-color: $yellow;
    border-color: $yellow;
}

.btn-warning:hover,
.btn-warning:active {
    background-color: $yellow2;
    border-color: $yellow2;
}

.btn-danger {
    background-color: $red;
    border-color: $red;
}

.btn-danger:hover,
.btn-danger:active {
    background-color: $red2;
    border-color: $red2;
}
```

That's a bunch of code but it should be fairly easy to understand. I've simply followed the same steps I completed for the primary button for every other button variation. Along the way, I've replaced the default Bootstrap color values with our custom color palette. Once you're done, all of your buttons should now look like this:



We've now successfully customized the entire button component. As I mentioned earlier, there may be additional things you might want to do to the buttons. However, at the very least, we've done enough to show how you can make the component your own. The next step in this process is to go through every Bootstrap component one by one and apply the same customization process. We call this writing your own Bootstrap theme.

Writing a theme

Creating your own Bootstrap theme is a bit of an undertaking. The good news is that once you've done it you can reuse a ton of the code for future themes. That's where the real power in making your code modular comes into play. Instead of starting over from scratch each time, you can reuse old code and just extend it. In the last section, we learned how to customize the button component that was the start of our own theme. Let's first start by looking at some common Bootstrap components that you'll want to customize for your own themes.

Common components that need to be customized

There are many ways that you can theme Bootstrap. In some cases, you may only need to customize a few components to get a unique look and feel going. However, you may want to do a more thorough theming process so that your theme doesn't resemble the default Bootstrap look at all. In this section, let's start by listing some of the common components you will most likely want to customize.

Next we'll go through the process of writing the code to customize a few so you get an idea as to how it works. Here's a list of components that I would recommend customizing:

- Buttons
- Drop-downs
- Alerts
- Navbar
- Typography
- Tables

This list is just a starting place. If you want to create a unique theme, you should really try to customize all Bootstrap components. At the very least, you should change them to use your custom color palette, typography, and layout styles. We've already covered buttons so let's jump into customizing the drop-down component, which is an extension of the button.

Theming the drop-down component

The drop-down component requires a medium-sized amount of customization so it's a good starting place to get an idea of what is involved in this process. It also builds on the code we wrote for the button so it's a natural second step. It's important to note that some components will require a good amount of CSS to customize them, while others will only need a little bit. Let's start by creating a new Sass file for drop-downs. From your project folder, create a new file called `_dropdown.scss` in the `css/components` directory. You can leave the file blank for now, just save it.

Once you've created the new Sass file for the drop-down component, we need to import it into our main theme is called `custom.scss`. Open up the custom style sheet in your text editor and insert the following line of code after the `@import` for the button component:

```
@import "components/_dropdown.scss";
```

Now we are ready to start coding our custom drop-down styles. Open up `_dropdown.scss` in your text editor and let's insert this first section of CSS:

```
.dropdown-menu {
   color: $primary-text;
}
```

As with the buttons in the previous section, I'm only going to change the most basic properties to demonstrate how you can customize the component. Feel free to customize additional properties to get a more unique look and feel.

Let's break down what is happening here. The drop-down component is made up of the base `.dropdown-menu` CSS class. This controls how the menu will look. Here I've simply changed the text color to use for the `$primary-text` variable.

We also need to do some work on the list of links that appear in our drop-down menu. Insert the following CSS after the first section you just entered:

```
.dropdown-item:focus,
.dropdown-item:hover {
   color: $primary-text;
   background-color: $secondary-background;
}
```

Let me break down what is happening here:

- These CSS classes control the hover and focus states for each list item in our drop-down menu. Again, I've set it to use our `$primary-text` font color.
- When you hover on a list item, the background color changes. I've changed that background color to use our `$secondary-background` color variable. In this case you should use the background color variable, not a customized color variable. The reason for this is it's easier to keep track of what background colors you are using as you progress through the writing of your code.

The last thing we need to do is update the actual drop-down button trigger with some additional code. Enter the last part of CSS into the file:

```
.open > .btn-primary.dropdown-toggle:focus {
   background-color: $purple2;
   border-color: $purple2;
}
```

When the drop-down button trigger is clicked the `.open` CSS class will dynamically be inserted into the HTML code. This initiates a unique variation on the button class, a drop-down toggle focus. That may sound complicated but what you need to know is that you need to set this selector to our `$purple2` color so it matches the rest of the button.

I've overwritten the `background-color` and `border-color` properties to use `$purple2` from our color palette.

That's it, the drop-down component has now been themed to match our look and feel. If you preview it in the browser it should look like this when the menu is open:



Now that we've finished with the drop-down component let's move on to learning how to theme the alerts component.

Customizing the alerts component

The alerts component in Bootstrap is fairly easy to theme. As with the button component, it comes in a few variations. Let's start by coding up the CSS for the default color method. Create a new file called `_alerts.scss` and save it to the `css/components` directory. Don't forget to import it into `custom.scss` with the following line of code:

```
@import "components/_alerts.scss";
```

Once you've set up the file, let's get started with the code for the success alert component:

```
.alert-success {
   color: $green;
   background-color: lighten( $green, 30% );
   border-color: lighten( $green, 30% );
}
```

What you're now seeing should start to look familiar. However, I have introduced something new that I need to explain:

- This is the success alert so it should be green in color. The first thing I've done is change the text color to use the green from our palette with the `$green` variable.
- For the `background-color` and `border-color` properties, I'm using something new, a Sass function. In this case, I want a green color that is slightly lighter than my text. Instead of introducing another green color variable, I can use a Sass function to lighten the base `$green` variable color.
- To create the function, you use the `lighten` keyword. Inside the brackets you need to include the variable name you want to target, in this case `$green`, and finally include a percentage value for how much to lighten it by. This is a nice little trick to save you having to create more variables.

Once you code this up it should look like this in the browser:



As you can see, we are using the green color values from our color palette. Let's continue and customize the colors for the rest of the alert bar variations. Enter the following code into the `_alerts.scss` file:

```
.alert-info {
   color: $blue;
   background-color: lighten( $blue, 30% );
   border-color: lighten( $blue, 30% );
}

.alert-warning {
   color: $yellow;
   background-color: lighten( $yellow, 30% );
   border-color: lighten( $yellow, 30% );
}

.alert-danger {
   color: $red;
   background-color: lighten( $red, 30% );
   border-color: lighten( $red, 30% );
}
```

The other alerts follow the same pattern as the success version. They should look like this in the browser when you are done:



As you can see, the alerts are now using our color palette. Let's move on to the last component that I will show you how to customize, which is typography.

Customizing the typography component

The typography component isn't difficult to customize. We'll build off the base variables we set up to apply them to the appropriate HTML tags. As we did with our other components, start with creating a new file called `_typography.scss` and save it to the `css/components` directory. Once you do this, import the file into `custom.scss` with the following line of code:

```
@import "components/_typography.scss";
```

Let's start customizing the type by applying some styles to the base header tags:

```
h1, h2, h3, h4, h5, h6 {
   font-family: $heading-copy;
   color: $primary-text;
}
```

Here I've simply used the `$heading-copy` variable and applied it to all the HTML heading tags. This will allow our custom heading typeface to be used for all headers. I've also added the `$primary-text` variable so that our headers are using the correct text color. Next let's take a look at a few miscellaneous text styles that you will likely want to overwrite:

```
small {
   color: $light-text;
}

pre {
   color: $pre-text;
}

code {
   color: $code-text;
}
```

As we did with our base variables, I'm now applying some of them on actual selectors. Let's break it down:

- For the `<small>` HTML tag, I want it to look more subtle so I've set the text color to use the `$light-text` variable.
- I purposely set up color text variables for the HTML `<pre>` and `<code>` tags. I've now applied the `$pre-text` and `$code-text` variables to these tags.

That covers some of the basic typography styles you're going to want to customize. There are more you could add but I will let you explore these on your own. That also goes for all the Bootstrap components. We have only scratched the surface of the level of customizing you can do for your Bootstrap theme. However, I think I've given you a good introduction to what you need to do for coding your own Bootstrap themes.

Summary

That brings this lesson to a close. We've covered a ton of new content in this lesson including: the basics of Sass, how to use Sass in Bootstrap, how to create a library of Sass variables, how to apply those variables to customize Bootstrap components, and, finally, how to start writing your own Bootstrap theme.

Now it's time to try some hands-on projects. Before jumping to the next lesson, I recommend you to go through the steps mentioned in the `Build Process Manual` (code bundle).

Assessments

1. Which of the following is the right CSS code?
    i.
    ```
    .test {
       background: "red";
    }
    ```
    ii.
    ```
    class test {
       background: red;
    }
    ```
    iii.
    ```
    body {
       background: red;
    }
    ```
    iv.
    ```
    .test {
      "background": "red";
    }
    ```
2. Which among the following is the correct symbol to declare variable in CSS
    i. `var`
    ii. `$`
    iii. `%`
    iv. `.`
3. What is the correct syntax to import a `test.scss` file in Sass?
    i. `@import "components/_test.scss";`
    ii. `#import "components/test.scss";`
    iii. `#import "components/_test.scss";`
    iv. `@import "components/test.scss";`

4. Which of the following code adds `background color` variables to the file?
    i. `$primary-background: $white;`
    ii. `background-color: $white;`
    iii. `$background-color: white;`
    iv. `background-color: white;`
5. Which among the following selectors selects links on `mouse over`?
    i. `:link`
    ii. `:hover`
    iii. `:focus`
    iv. `:selection`

8
Bootstrapping Your Portfolio

From this chapter onwards, the learning experience would be dynamic as you will go through interactive real world, hands-on project which will provide you complete exposure to Bootstrap 4 and Sass and thereby raising your confidence in Bootstrap to a tremendous level.

Let's imagine we're ready for a fresh design of our online portfolio. As always, time is scarce. We need to be efficient, but the portfolio has to look great. And of course, it has to be responsive. It should work across devices of various form factors, since this is a key selling point for our prospective clients. This project will enable us to leverage a number of Bootstrap's built-in features, even as we customize Bootstrap to suit our needs.

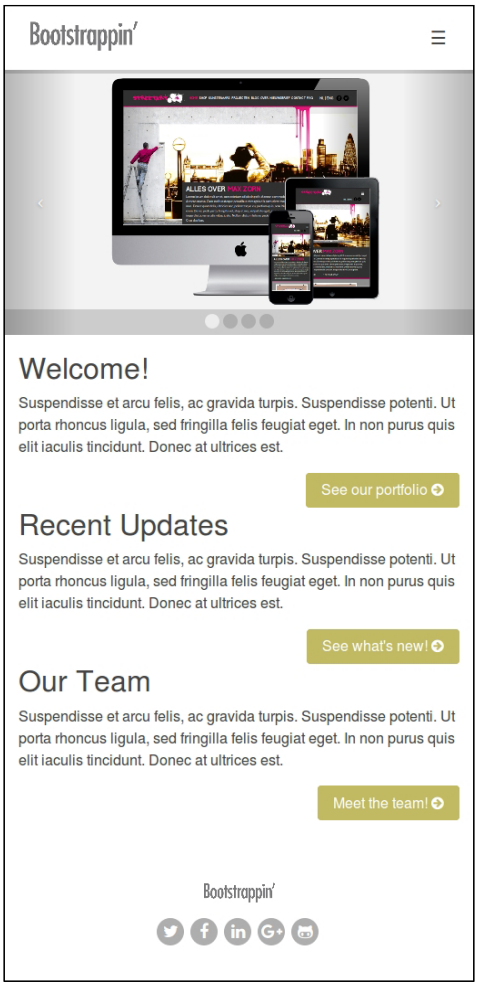
What we'll build

We've thrown together a couple of home page mock-ups. Though we have in mind what we want for large screens, we've begun with a handheld screen size to force ourselves to focus on the essentials.

You'll notice the following features:

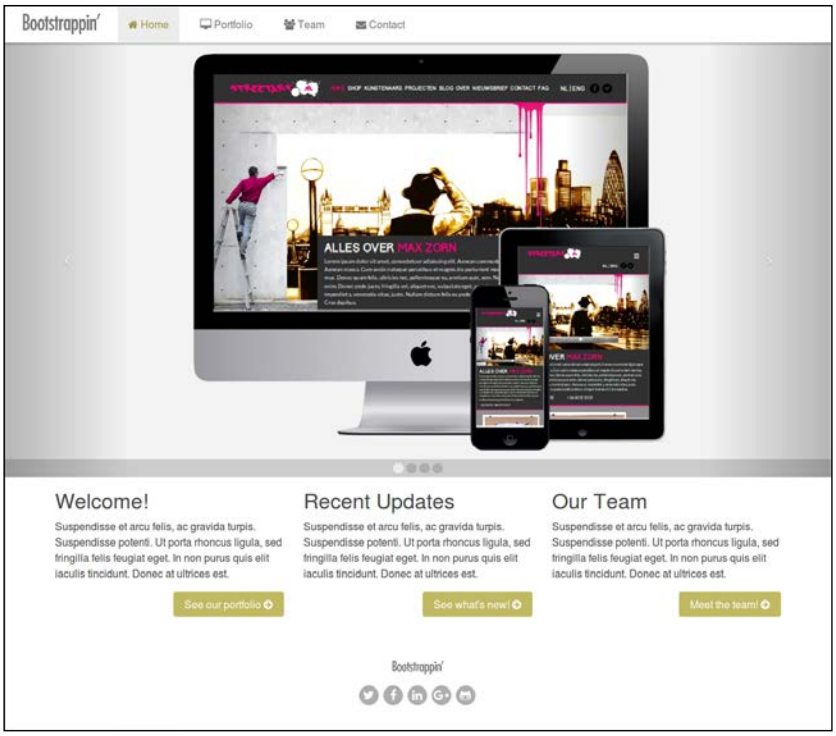
- A collapsed responsive navbar with logo
- A sliding carousel with four images of featured portfolio items
- A single-column layout with three blocks of content, each with a heading, a short paragraph, and a nice big button with an invitation to read further
- A footer with social media links

Here is the design mock-up as shown in the following screenshot:



Altogether, this should provide a good introduction to our work. The carousel is tall enough to give a good amount of visual space to our portfolio images. It is not difficult to navigate quickly to the content below, where a user can efficiently scan key options for taking the next step inside. By presenting key links as nice big buttons, we will establish a helpful visual hierarchy for the key action items, and we will ensure that visitors do not have problems because of fat fingers.

For ease of maintenance, we've elected to have only two major breakpoints in this design. We'll use the single-column layout for screen sizes narrower than 768 px. Then, we'll shift to a three-column layout:



You'll note the following features in the mock-up for tablets and higher versions:

- A navigation bar at the top, which is enhanced with icons
- A widescreen version of the home page carousel, with images stretching to fill the full width of the browser
- A three-column layout for our textual content blocks
- A footer with content at the center

The color scheme is fairly simple: Shades of gray, plus a golden-green color for links and highlights.

With these design goals in mind, we'll can move on and get our content in place.

Surveying the exercise files



Hope you've gone through the steps mentioned in the Build Process Manual available in the code bundle.

Let's look at the first few files for this exercise. Create a new project by using Bootstrap CLI.

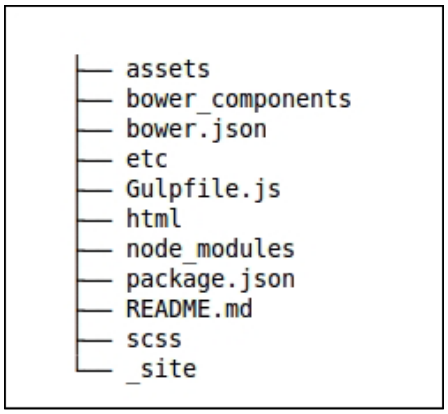
You can install Bootstrap CLI by running the following command in your console:

```
npm install -g bootstrap-cli
```

Then you can set up your project by running the following command:

```
bootstrap new
```

Again, choose a new empty Bootstrap project. When prompted, select Panini, Sass, and Gulp option.



There are a few additions you will have to make now:

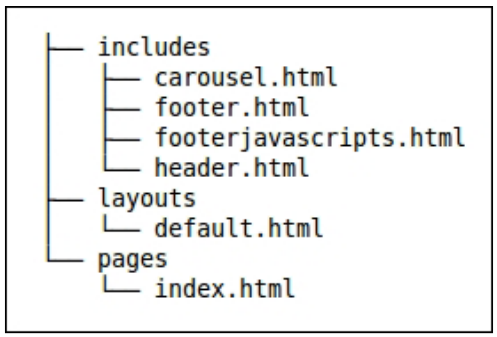
1. Create a new `assets/images` folder.
2. Copy the files of the `img` folder to the new `assets/images` folder. It contains five images:
    - One logo image, named `logo.png`
    - Four portfolio item images
3. Add a new task to the `Gulpfile.js` file:

```
// Copy assets
gulp.task('copy', function() {
    gulp.src(['assets/**/*']).pipe(gulp.dest('_site'));
 });
```

4. And finally, add the preceding task to the default task at the end of the file:

```
gulp.task('build', ['clean','copy','compile-js','compile
sass','compile-html']);
```

The `html` folder that contains your Panini HTML templates should have the following file and folder structure:



You can read more about Panini at `https://github.com/zurb/panini`.

Here are some of the details of the files shown in the preceding screenshot:

- The `html/pages/index.html` file contains the following HTML and template code:
    - The `{{> carousel}}` snippet which includes the carousel (`includes/carousel.html`)
    - Content blocks, like that shown beneath:

```
<h2>Welcome!</h2>
<p>Suspendisse et a.....</p>
<p><a href="#">See our portfolio</a></p>
```

- The `includes/header.html` file, included in `layouts/default.html`, contains our navbar and has the following new touches:
    - Navbar items have been updated to reflect our new site architecture:

```
<header role="banner">
  <nav class="navbar navbar-light bg-faded"
role="navigation">
```

```
          <a class="navbar-brand" href="index.
html">Bootstrapping</a>
           <button class="navbar-toggler hidden-md-up pull-xs-
right"
         type="button" data-toggle="collapse"
        data-target="#collapsiblecontent">
            ☰
          </button>
          <ul class="nav navbar-nav navbar-toggleable-sm collapse"
         id="collapsiblecontent">
            <li class="nav-item">
              <a class="nav-link active" href="#">Home <span
class="sr-only">
         (current)</span></a>
             </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Portfolio</a>
            </li>
             <li class="nav-item">
              <a class="nav-link" href="#">Team</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="#">Contact</a>
            </li>
          </ul>
          </nav>
        </header>
```

- The `includes/footer.html` file, included in `layouts/default.html`, contains the following items:
    - A logo in the footer
    - Social links:

```
<footer role="contentinfo">

    <p><a href="{{root}}index.html"><img
        src="{{root}}images/logo.png"
        width="80" alt="Bootstrapping"></a></p>

    <ul class="social">
      <li><a href="#" >Twitter</a></li>
      <li><a href="#" >Facebook</a></li>
      <li><a href="#" >LinkedIn</a></li>
      <li><a href="#" >Google+</a></li>
```

```
                <li><a href="#" >GitHub</a></li>
            </ul>

        </footer>
```

Other than the navbar, no Bootstrap classes have been added to style the carousel, columns, or icons yet.

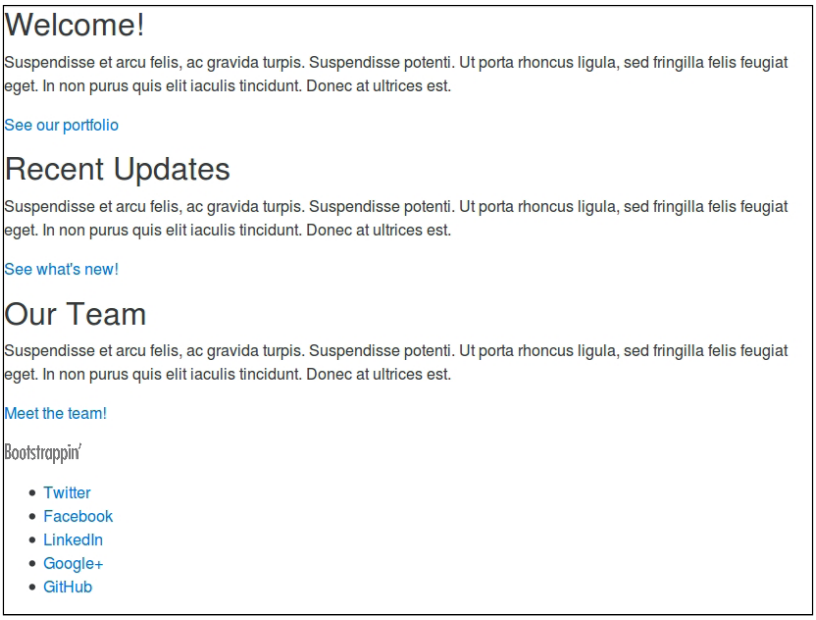
Further on, we'll discuss how to use Sass to customize your project. Now you can see that the `app.scss` file imports the `includes/_navbar.scss` file. The SCSS in the file removes the floats for the navigation links for smaller screens.

Instead of the preceding modifications, you can also start with the files found in the `Lesson 8/start` folder. In this folder, run the `npm install` and `bower install` commands first. After running npm and bower commands, you can run the bootstrap watch or gulp command to view the results in your browser.

You'll see the navbar, followed by the portfolio images:



The blocks of text and the footer, with a list of social links, follow after the images:



It's not much to speak of yet. Let the transformation begin.

We'll start by applying the Bootstrap classes, allowing us to quickly and efficiently establish the fundamentals for our interface elements using Bootstrap's default CSS styles and JavaScript behaviors.

Marking up the carousel

Let's get started with our carousel, which will rotate between four featured images from our portfolio.

Bootstrap's carousel markup structure can be found in its documentation pages at `http://getbootstrap.com/components/carousel/`.

Following the pattern used in the example, we'll begin with this structure to set up the fundamental element. This will contain all parts of the carousel, followed by the progress indicators:

```
<div id="carousel-feature" class="carousel slide" data-
ride="carousel">
  <ol class="carousel-indicators">
    <li data-target="#carousel-feature" data-slide-to="0"
class="active"></li>
    <li data-target="#carousel-feature" data-slide-to="1"></li>
```

```
      <li data-target="#carousel-feature" data-slide-to="2"></li>
    </ol>
  </div>
```

Note that I've used a `div` tag with an ID (`id="carousel-feature"`) to establish the fundamental context of `carousel`. The `carousel` class applies the Bootstrap's carousel CSS to the carousel elements, adding appropriate styles to the carousel indicators, the carousel items, and the next and previous controls.

The `carousel-feature` ID must be used in the `data-target` attributes of the progress indicators. This signals the JavaScript plugin to update the indicator for the active carousel item with the `active` class. We've provided that class for the first indicator to get things started. From there, the JavaScript takes over. It removes the class, and adds it to the appropriate indicator as the carousel cycles.

Also, note that the `data-slide-to` values begin counting from 0. This is the standard behavior for JavaScript and other programming languages. Just remember: Start counting at zero, not one.

After the indicators, the element of the class `carousel-inner` follows. This serves as the wrapper to contain all of the carousel items—in this case, our images.

The carousel items come within `carousel-inner`. They are a group of `div` tags, each with `class="item"`. Modify the first `item` to have both the classes item and `active`, to make it visible from the outset.

Thus, the markup structure works as follows:

```
<!-- Wrapper for slides -->

<div class="carousel-inner" role="listbox">
    <div class="carousel-item active">
        <img src="{{root}}images/project1.png" alt="Streetart.com
Homepage">
    </div>
    <div class="carousel-item">
        <img src="{{root}}images/project2.png" alt="Your bussiness">
    </div>
    <div class="carousel-item">
        <img src="{{root}}images/project3.png" alt="Your blog">
    </div>
    <div class="carousel-item">
        <img src="{{root}}images/project4.png" alt="Menstrualcups.eu
Homepage">
    </div>
</div><!-- /.carousel-inner -->
```

After the carousel items, we need to add the carousel controls. These will provide the next and previous buttons at the left and right edges of the carousel. After the controls, we'll close up our entire markup structure with the closing `div` tag:

```
<!-- Controls -->
  <a class="left carousel-control" href="#carousel-feature"
role="button" data-slide="prev">
    <span class="icon-prev" aria-hidden="true"></span>
    <span class="sr-only">Previous</span>
  </a>
  <a class="right carousel-control" href="#carousel-feature"
role="button" data-slide="next">
    <span class="icon-next" aria-hidden="true"></span>
    <span class="sr-only">Next</span>
  </a>
</div><!-- /#homepage-feature.carousel -->
```



The carousel controls need to have the ID of the fundamental carousel element (`#carousel-feature`) for their `href` value. The code, then, looks like this: `<a class="left carousel-control" href="#carousel-feature" role="button" data-slide="prev">`

Now you can write down the complete code for the carousel in the file. Once this code is in place, run the `gulp` command if you didn't run the bootstrap watch already. Bootstrap's styles and JavaScript should start working. Your images should now work as a sliding carousel!

Notice that the carousel component requires both jQuery and the JavaScript plugin. The Gulp build process merges jQuery and all plugin code into a single `app.js` file.

By default, the carousel will slide every 5 seconds. Let's set the interval to 2 seconds to give our users time to appreciate the full beauty of our work:

1. Create a new file called `js/main.js`.
2. Add the following lines. We'll begin with the jQuery method of checking to ensure page elements are ready, and then initialize the carousel with an interval of 2,000 milliseconds:

```
$( document ).ready(function() {
  $('.carousel').carousel({
    interval: 2000
  });
});
```

3. Notice that you should automatically copy the `js/main.js` file from the assets folder and link it in the file, or add it to the `compile-js` task in your `Gulpfile.js` file:

```
gulp.task('compile-js', function() {
  return gulp.src([bowerpath+
  'jquery/dist/jquery.min.js', bowerpath+
  'tether/dist/js/tether.min.js', bowerpath+
  'bootstrap/dist/js/bootstrap.min.js','js/main.js'])
 .pipe(concat('app.js'))
 .pipe(gulp.dest('./_site/js/'));
});
```



You should also consider adding the `js/main.js` file to the Gulp `watch` task.

4. Save and restart your application. You will see that the interval has increased to 2 seconds.

Instead of passing the options via JavaScript as we did just now, you can also pass them via data attributes. The interval of the carousel can be set via the `data-interval` attribute:

```
<div id="carousel-feature" class="carousel slide" data-ride="carousel"
data-interval="2000">
```

For this and other options, see the Bootstrap carousel documentation at `http://getbootstrap.com/javascript/#carousel`.

We'll return to customize the styling of the carousel, its indicators, and its icons later in the lesson. In the next section, you will learn how to use both JavaScript and CSS (SCSS) to modify the working of the carousel.

How does the carousel work?

The jQuery plugin changes the CSS classes of the items of the carousel. When the page loads, the first item already has the active class; when the interval has passed, the plugin moves the `active` class to the next item, and so on. The plugin not only changes the position of the `active` class, but also temporally adds the next and left classes. Together with the CSS3 animations on these classes, the sliding effect is created. You can read more about CSS3 animations at `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations`.

The `transition` is set on the `carousel-inner` class as follows:

```
transition: transform .6s ease-in-out;
```

In this declaration, the ease-in-out value sets the transition-timing-function of the animation (transition effect); for more information, see `https://developer.mozilla.org/en/docs/Web/CSS/transition-timing-function`. In essence, it lets you establish an acceleration curve, so that the speed of the transition can vary over its duration. Later on, we'll see that you can also use `keyframes` to describe the different states of the transition.

The transformations performed are translate3ds. The `translate3d()` CSS function moves the position of the element in the 3D space. More information can be found at `https://developer.mozilla.org/en-US/docs/Web/CSS/transform-function/translate3d`. The carousel moves the carousel items over the X-axis as follows:

```
&.next,
&.active.right {
  left: 0;
  transform: translate3d(100%, 0, 0);
}
```

Changing the carousel by adding new animations

When we replace the CSS animation as described in the previous section with another animation, the carousel's slider effect changes.

The `Animate.css` project by Daniel Ede contains a lot of CSS animations for you to use in your projects. You can use these animations for our carousel too. You can find the library at `http://daneden.github.io/animate.css/`.

We can create the new animation with SCSS code. Because our build process already runs the `autoprefixer`, you don't need to take the vendor prefixes into account. In our example, we'll use the `flipInX` animation of the `Animate.css` library, which rotates the images over the x axis.

Now add the SCSS code present in the file named `Animation_Carousel` at the end of the `scss/includes/_carousel.scss` file:

If ran the `bootstrap watch` or `gulp` command already, you can inspect the results in the browser. You'll find that the images rotate over the X-axis now.

JavaScript events of the Carousel plugin

Bootstrap provides custom events for most plugins' unique actions. The Carousel plugin fires the `slide.bs.carousel` (at the beginning of the slide transition) and `slid.bs.carousel` (at the end of the slide transition) events. You can use these events to add custom JavaScript code. You can, for instance, change the background color of the body on the events by adding the following JavaScript into the `js/main.js` file:

```
$('.carousel').on('slide.bs.carousel', function () {
  $('body').css('background-color','#'+(Math.random()*0xFFFFFF<<0).
toString(16));
});
```

Note that the `gulp watch` task is not set for the `js/main.js` file, so you have to run the `gulp` or `bootstrap watch` command manually after you are done with the changes.

For more advanced changes to the plugin's behavior, you can overwrite its methods by using, for instance, the following JavaScript code:

```
!function($) {
var number = 0;
    var tmp = $.fn.carousel.Constructor.prototype.cycle;
    $.fn.carousel.Constructor.prototype.cycle = function
(relatedTarget) {
        // custom JavaScript code here
        number = (number % 4) + 1;
        $('body').css('transform','rotate('+ number * 90 +'deg)');
        tmp.call(this); // call the original function
    };

}(jQuery);
```

This code sets the `transform` CSS property without vendor prefixes. The autoprefixer only prefixes your static CSS code. For full browser compatibility, you should add the vendor prefixes in the JavaScript code yourself.



Bootstrap exclusively uses CSS3 for its animations, but Internet Explorer 9 doesn't support the necessary CSS properties.

Let's continue leveraging the power of Bootstrap's default styles and set up a responsive grid for the content below the carousel.

Creating responsive columns

We have three blocks of text, each with a heading, a short paragraph, and a link. In screen sizes of approximately tablet width or more, we would like this content to be laid out in three columns. In narrower screen widths, the content will organize itself in one full-width column.

Take a moment to visit and read the documentation for Bootstrap's mobile-first responsive grid. You can find it at `http://getbootstrap.com/css/#grid`.

In short, the grid is based on a 12-column system. The basic class structure allows us to use a class of col-12 for full width, `col-6` for half width, `col-4` for one-third width, and so on.

Thanks to the creative use of media queries, Bootstrap's grid can be very adept at responding to different screen sizes. Recall that we want our welcome message to have a single-column layout up for screens to tablet-size, and then adapt a three-column layout at approximately 768 px for larger screens. Conveniently, Bootstrap has a built-in breakpoint at 768 px, which is the default value defined in the `$grid-breakpoints` Sass variable. Above 768 px is the large range, beginning at 992 px, also defined in the `$grid-breakpoints` Sass variable, then the extra-large screen, measured at 1,200 px and higher. I'll refer to these as Bootstrap's extra-small, small, medium, large, and extra-large breakpoints.

With the medium breakpoint, there is a special column class that uses the formulation col-md-. Because we want three columns after the small breakpoint, we'll use `class="col-md-4"`. Below the medium breakpoint, the elements will remain full-width. Above it, they will shift to one-third width and line up side by side. Notice that the navbar also collapses at 768 px. The full structure is given here, with paragraph contents abbreviated for clarity:

```
<div class="container">
  <div class="row">
    <div class="col-sm-4">
    <h2>Welcome!</h2>
    <p>Suspendisse et arcu felis ...</p>
    <p><a href="#">See our portfolio</a></p>
  </div>
  <div class="col-sm-4">
    <h2>Recent Updates</h2>
    <p>Suspendisse et arcu felis ...</p>
    <p><a href="#">See what's new!</a></p>
  </div>
  <div class="col-sm-4">
    <h2>Our Team</h2>
```

```
      <p>Suspendisse et arcu felis ...</p>
      <p><a href="#">Meet the team!</a></p>
    </div>
    </div><!-- /.row -->
  </div><!-- /.container →
```

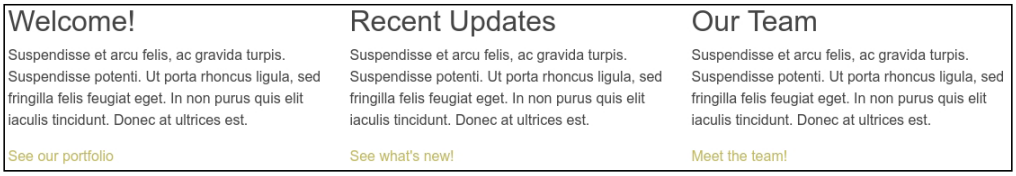


You should edit the preceding code in the `html/pages/index.html` file.

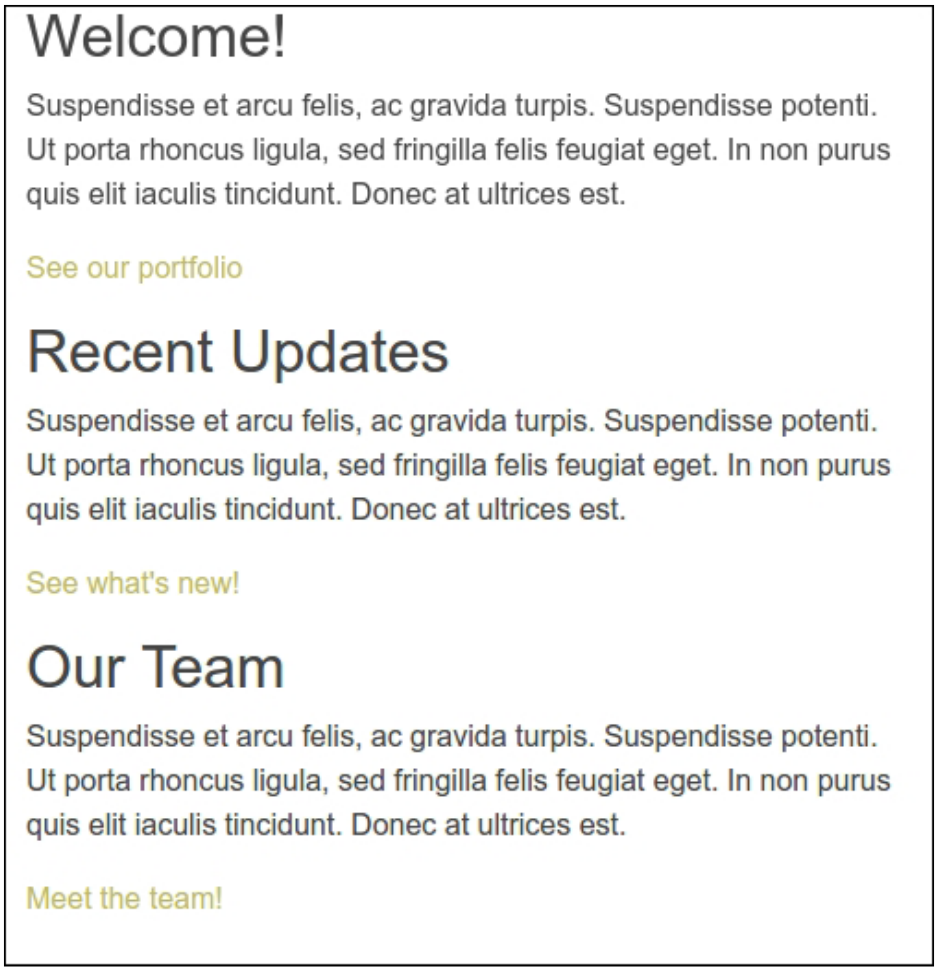
If you're unfamiliar with the `container` and `row` classes, here is what they do:

- The `container` class constrains the width of the content and keeps it centered within the page
- The `row` class provides the wrapper for our columns, allowing extra left and right margins for the column gutters
- Both the `container` class and the `row` class are `clearfixed` so that they contain floating elements and clear any previous floating elements

Now, save the file and run the `bootstrap watch` or `gulp` command if you have not already done so. With your browser width above 768 px, you should see the following three-column layout take shape:



Resize your browser window below 768 px, and you'll see it revert to a single column:



With our responsive grid in place, let's turn those links into clearly visible calls to action by utilizing Bootstrap's button styles.

Turning links into buttons

Turning our key content links into visually effective buttons is straightforward. The key classes we'll employ are as follows:

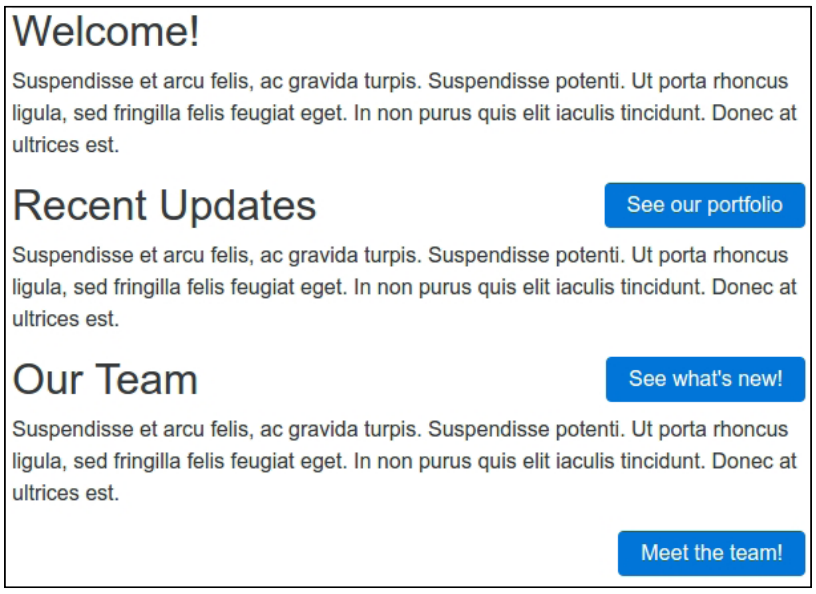
- The `btn` class will style a link as a button.
- The `btn-primary` class will assign a button the color of our primary brand color.
- The `pull-xs-right` class will float the link to the right, moving it into wider

space to make it a more appealing target. The `xs` part of the class name means that it should be applied on every viewport wider than the extra-small breakpoint of 0 pixels. The `pull-md-right` class only floats the elements on viewports wider than 768 px.

Add these classes to the link at the end of each of our three content blocks:

```
<p><a class="btn btn-primary pull-xs-right" href="#">See our
portfolio</a></p>
```

Save. You should see the following result:



We've made great progress. Our key elements are taking shape.

With our fundamental markup structure in place, we can start working on the finer details. Getting there will require some custom CSS. We're going to approach this by leveraging the power of Bootstrap's Sass files. If you're new to Sass, no worries! I'll walk you through it step by step.

Understanding the power of Sass

In the following sections, we will be organizing, editing, customizing, and creating SCSS files in order to generate the desired CSS for our designs.



If you are unfamiliar with Sass and would like to learn more about it, I would recommend reading the documentation at `https://www.sass-lang.com/`.

In a nutshell, we may say that generating CSS using the Sass preprocessor is an exciting and freeing experience. The key benefits of working with Sass are discussed in the following sections.

Customizing Bootstrap's Sass according to our needs

As we work with Bootstrap's Sass files, we'll exert considerable control over them by doing the following:

- Organizing our `scss` folder to give us the flexibility and freedom to accomplish what we need while making future maintenance easier
- Customizing Bootstrap's Sass variables
- Creating a few custom Sass files of our own
- Incorporating a set of font-based icons in our site assets, providing the icons that we need for our social media links

In other words, we'll be doing more than merely learning and applying Bootstrap's conventions. We'll be bending them to our will.

In this lesson's exercise files, open the `scss` directory. Inside, you should see the following structure:

```
├── app.scss
└── includes
    ├── _backgrounds.scss
    ├── _bootstrap.scss
    ├── _carousel.scss
    ├── _footer.scss
    ├── mixins
    │   └── _size.scss
    ├── _mixins.scss
    ├── _navbar.scss
    ├── _page-contents.scss
    └── _variables.scss
```

To prepare for what's ahead, I've given you a head start by explaining the new layer of organization. All of Bootstrap's Sass files are saved in the `bower_components/bootstrap/scss/` folder. You should not modify these files; you can (re)use them as described in the next sections. Keeping the original files untouched enables you to update Bootstrap without undoing your modifications.

First, the `app.scss` file imports two partial files:

```
@import "includes/variables";
@import "includes/bootstrap";
```

The file includes/_bootstrap.scss is a modified copy of the original `bootstrap.scss` file. This file imports all other original Bootstrap files, and it is used in the compiling process to create one unified style sheet from all of our imported Sass files. The `includes/_variables.scss` file is a modified copy of the original `_variables.scss` file. This file contains the declaration of all Bootstrap's Sass variables. Because the `includes/_variables.scss` file is imported before the original `_variables.scss` file, the variables in it can be used to overwrite Bootstrap's default settings.

Why go through this trouble? Because we'll soon be modifying Bootstrap's default settings and creating custom Sass files of our own. When we do that, we can leave the `Bootstrap` folder and its files as they are while making adjustments in the custom files that we will create.

Let's begin the customization! We'll start by customizing Bootstrap's variables and adding a few new variables of our own.

Customizing variables

Next, we'll create a copy of Bootstrap's variables file and customize it to our needs:

1. Find the `includes/_variables.scss` file in the `scss` folder and open it in your editor.
2. Scanning through the lines of this file, you'll see the variables used to set the CSS values for everything from basic colors to the body background, font-families, navbar height and background, and so on. It's beautiful to behold. It's even more fun to meddle with. Before we meddle, let's create our own copy of this file, allowing us to leave Bootstrap's default variables intact in case we ever want to revert back to them. Next, let's implement our new color scheme:

3. In the top-most section of our new `includes/_variables.scss` file, you'll see the default Bootstrap variables for grays and brand colors:

```
$gray-dark:                 #373a3c;
$gray:                      #55595c;
$gray-light:                #818a91;
$gray-lighter:              #eceeef;
$gray-lightest:             #f7f7f9;
```

4. We have the specific values that we're after. So, let's simply substitute our desired values (feel free to do the math if you prefer!). Then, we'll add an additional two variables to encompass the full range that we need. The result is as follows:

```
$gray-dark:               #454545;
$gray:                    #777;
$gray-light:              #aeaeae;
$gray-lighter:            #ccc;
$gray-lightest:           #ededed;
```

5. Next, we'll update the `$brand-primary` variable under `Brand colors`. We'll adjust this to our golden hue:

```
// Brand colors
// -------------------------
$brand-primary:         #c1ba62;
```

6. When you run the bootstrap watch (or gulp) command already, your browser automatically reloads after saving the `includes/_variables.scss file`.

If this is successful, the most noticeable changes will be in the link color and buttons with the `btn-primary` class, which will both take the new `$brand-primary` color.

Customizing the navbar

Now, let's edit the variables that set the navbar height, colors, and hover effects.

Let's start by changing the height. The navbar has a padding of `$spacer / 2` by default, and the total height is set by the font size and the vertical padding.

In the local `includes/_variables.scss` file, search for the `$navbar-padding-vertical` variable and update it as follows. This will expand the `navbar height`:

```
$navbar-padding-vertical: $spacer;
```

Now we can set the background color of the navbar. The original HTML code in the `html/includes/headers.html` file had a `bg-faded` class. Because we want a white

background color for our navbar, we can simply remove this class; the body already has a white background, and not setting the background property for the navbar will color the navbar white too.

You can also use Sass and Bootstrap's mixins to generate a new `bg-white class`. Create a new file `scss/includes/_backgrounds.scss` and edit the following SCSS into it:

```
@include bg-variant('.bg-white', #fff);
```

Note that the `bg-variant` mixin declares the `background-color` and `color` properties with `!important`. Because the `color` property is set to `#fff` (white) by default, using the `bg-variant` mixin does not seem to be the most flexible solution. Setting the `background-color` property for the navbar selector in the `scss/includes/_navbar.scss` file is a better solution. You can set the `background-color` property as follows:

```
.navbar {
 background-color: #fff;
}
```

The color of the navbar links is set by the `.navbar-light` or `.navbar-dark` classes. You should use the `.navbar-dark` class for navbars with a dark background and `.navbar-light` for light background. Our navbar in the `html/includes/header.html` file has got the `.navbar-light` class, so in order to change the link colors, you'll have to modify the `$navbar-light-*` variables in the `includes_variables.scss` file as follows:

```
$navbar-light-color:            $gray;
$navbar-light-hover-color:      $link-hover-color;
$navbar-light-active-color:     $link-hover-color;
$navbar-light-disabled-color:   $gray-lighter;
```

Note that we've already changed the `$gray` and `$gray-lighter` variables before, and the $link-hover-color variable has got the same value as the `$brand-primary` variable which we have set to the `#c1ba62`; color value.

If you like having a different background color for the hovered and active links, you can open the `scss/includes/_navbar.scss` file with your favorite text editor and perform the following steps:

1. First, remove the vertical padding with the following SCSS code:

```
.navbar {
  padding-top: 0;
  padding-bottom: 0;
}
```

2. Now apply the padding on the `.nav-link` selectors and set the background color for the hover and active states:

```
.navbar {
  .nav-link
  {
    padding: $spacer;
    &:hover,
    &.active {
      background-color: $gray-lightest;
    }
  }
}
```

3. Because the `.navbar-brand` has a larger font size, you will have to correct the padding for this selector:

```
.navbar {
  .navbar-brand {
    padding: ($spacer - (($font-size-lg - $font-
      size-base) * $line-height) / 2));
  }
}
```

If you did not run the `bootstrap watch` command already, run it now and you should see the following new features in your navbar:

- It should grow about 16px (2 x 1em) taller
- Its background color should turn white
- The nav item backgrounds change on hover and active states
- Link text should activate our brand-primary color on hover and when active, as shown in the following screenshot:



Now, let's put our logo image in place.

Adding the logo image

Find the `logo.png` file in the `assets/images` folder. You may notice that its dimensions are large, 900 px wide. In our final design, it will be only 120 px wide. Because the pixels will be compressed into a smaller space, this is a relatively easy way to ensure that the image will look good on all devices, including retina displays. Meanwhile, the file size of the image, which has already been optimized for the Web, is only 19 KB.

So, let's put it in place and constrain its width:

1. Open the `html/includes/header.html` file in your text editor.
2. Search for this line within the navbar markup:

```
<a class="navbar-brand"
href="index.html">Bootstrapping</a>
```

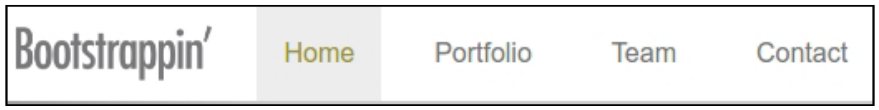
3. Replace the HTML from the previous step with the following image tag, including its `alt` and `width` attributes:

```
<a class="navbar-brand" href="index.html"><img src="
{{root}}/images/logo.png" alt="Bootstrapping"
width="120"></a>
```



Be sure to include the `width` attribute, setting its width to 120 px. Otherwise, it will appear very large on the page.

If you didn't run the bootstrap watch command already, run it now. You should see the logo in place:



You may notice that the navbar height has expanded, and that its bottom edge no longer lines up with the bottom edge of the active nav item. This is due to the padding placed around the `bar-brand` class earlier. We need to adjust the appropriate padding values. We can do that in a few quick steps:

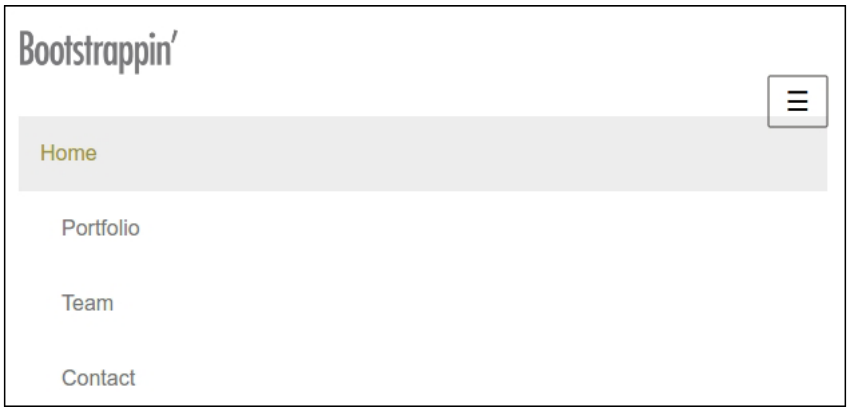
1. Use your text editor to open the `scss/includes/_navbar.scss` file again. Change the padding of the `bar-brand` class as follows:

```
padding: ($spacer - ((2.16rem - ($font-size-base * $line-height)) /
2));
```

2. When resizing the width of the image to 120 px, its height becomes around 34.51 pixels, 34.51 / 16 = 2.16 rem.

The powers of Sass continue to impress. Of course, we should also take care of the collapsed responsive navigation, so resize your browser to a viewport smaller than 768 px.

The navigation should now look like that shown in the following screenshot:



You will see that there is not enough padding around the logo and that the toggle button is not in line with the logo. We will use Sass again to correct these issues. Again, open the `scss/includes/_navbar.scss` file. Remember that we set the vertical padding of the navbar to 0 before. Now wrap this declaration into a CSS media query as follows to only apply it for the larger viewports:

```
@include media-breakpoint-up(md) {
  padding-top: 0;
  padding-bottom: 0;
}
```

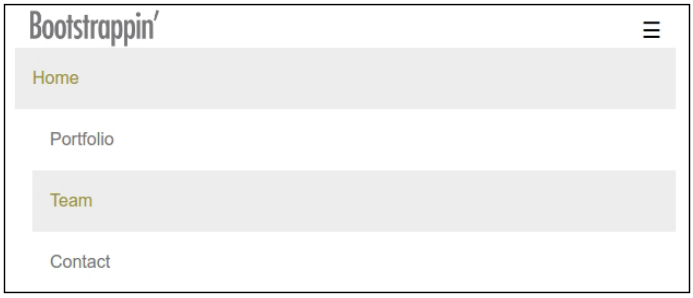
As already explained, the `media-breakpoint-up mixin` is part of Bootstrap's Sass mixin and can be used to hide or show elements according to Bootstrap's media query ranges. The SCSS preceding code compiles into CSS as follows:

```
@media (min-width: 768px) {
  .navbar {
    padding-top: 0;
    padding-bottom: 0;
  }
}
```

To get the logo in line with the toggle button, you will have to change the display property of the logo image from block to inline-block. You can establish that by editing the following SCSS code in the `scss/includes/_navbar.scss file`:

```
.navbar {
  @include media-breakpoint-down(sm) {
    .navbar-brand,
    .nav-item {
      float: none;
      > img {
        display: inline-block;
      }
    }
  }
}
```

Finally, inspect the latest version of your `scss/includes/_navbar.scss` file. In your browser, the results should look like the following screenshot:



Now, let's add icon powers.

Adding icons

It's time to add icons to our navigation. Glyphicons that come with Bootstrap 3 are dropped in Bootstrap 4. Here, we'll use the large library of icons offered by Font Awesome. Other icon font sets can be found around the Web.

Font Awesome is a font icon set that offers 628 icons at the time of writing this course. Font Awesome icons are free, open source, and built to play nice with Bootstrap. You can see the Font Awesome home page at `http://fortawesome.github.io/Font-Awesome/`.

Let's fold Font Awesome into our workflow.

Here, we compile the CSS code of Font Awesome into our main `app.css` file:

1. First, install Font Awesome in your project folder by running the following command in your console:

```
bower install font-awesome --save
```

2. After that, you can import Font Awesome's main SCSS file into your `scss/app.scss` file:

```
@import "includes/variables";
@import "font-awesome/scss/font-awesome.scss";
@import "includes/bootstrap";
@import "includes/navbar";
```

3. Last but not least, copy the font files to your assets folder:

```
cp bower_components/font-awesome/fonts/*
 assets/fonts/
```

4. The Font Awesome `scss` files include a variable specifying the path to the Font Awesome web fonts. We need to check to make sure that this variable matches our folder structure. Ensure that the `$fa-font-path` variable is set to `../fonts` as follows in our `scss/includes/_variables.scss file`:

```
$fa-font-path:    "../fonts";
```



This path is relative to the compiled CSS file, which is in our `css` directory

5. Now, in the `html/includes/header.html` file, let's update the icon for the `Team` navbar item to use the Font Awesome icon named `fa-group`. We also need the standalone `fa` class: `<i class="fa fa-group"></i> Team`
6. Save this change to the `html/includes/header.html` file, and refresh your browser.

If all works as it should, you should see the following result:





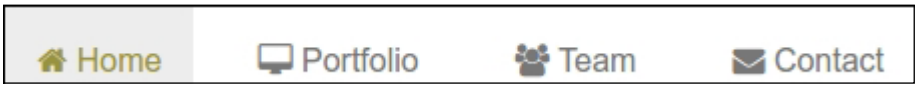
If you see a strange symbol or nothing, that's a sign that the web fonts are not coming through. Double-check that your icon classes are correct (including the `fa` class), your Font Awesome web font files are in your `fonts` directory, and the path is set correctly in the `scss/includes/_variables.scss` file.

Now update your icon markup in the `html/includes/header.html` file to make use of the desired Font Awesome icons.

The Font Awesome icon page `http://fortawesome.github.io/Font-Awesome/icons/` allows you to scan your options. Our mock-up calls for these icons in the navbar:

```
<i class="fa fa-home"></i> Home
<i class="fa fa-desktop"></i> Portfolio
<i class="fa fa-group"></i> Team
<i class="fa fa-envelope"></i> Contact
```

This has the following result:



This completes our nav, or almost completes it. We've inadvertently created a small problem that we need to fix before moving on.

It's time to move on to the carousel.

Styling the carousel

We're going to take Bootstrap's default carousel styles and apply some significant customization. Let's create a new `_scss/includes/carousel.scss` file and import it into our `scss/app.scss` file.

Now to begin customizing and making aesthetic enhancements.

Adding top and bottom padding

Let's add some top and bottom padding to the `.carousel` element itself and add our `@gray-lighter` color for a background color:

```
.carousel {
  padding-top: 4px; // added
```

```
  padding-bottom: 28px; // added
  background-color: @gray-lighter; // added
}
```

After saving and reloading (run the bootstrap watch or gulp command), you'll see the light gray background appears in our newly created space above and below the carousel images. This provides a bit of framing to set them off from the other elements above and below. In a bit, we'll take advantage of the extra bottom padding to position our carousel indicators in a way that allows them to stand out much more clearly.

Now to style the carousel indicators.

Repositioning the carousel indicators

The carousel indicators serve to inform the user how many slides are in our carousel, and highlight the current spot in the rotation. At present, these indicators are barely visible, languishing near the bottom center edge of our portfolio images:



Note that I have temporarily set the border color to white to make the preceding picture as follows:

```
.carousel-indicators li {
  border: 1px solid white;
}
```

Let's move these indicators into their own space, just below the image:

1. We want to move the indicators down even closer to the bottom edge, into our light gray area created by the padding we added previously. So, let's adjust the bottom positioning. In addition, we need to remove the default bottom margin by zeroing it out. Write down the following SCSS in the `_scss/includes/carousel.scss` file:

```
.carousel-indicators {
 position: absolute;
  bottom: 0;
  margin-bottom: 0;
}
```

2. Save the file; if you have run the bootstrap watch command already, your browser automatically reloads.

This brings our desired result. The indicators now stay positioned in the desired space across all screen dimensions:



Now let's update their appearance to make them larger and easier to see.

Styling the indicators

We'll make our carousel indicators more visible by using our gray variables. We'll also increase their size a bit. We can get a start in our `scss/includes/_variables.scss` file:

1. In `scss/includes/_variables.scss`, just after the `$carousel-control` variables, you'll find two variables beginning with `$carousel-indicator:`

```
$carousel-indicator-active-bg:          #fff;
$carousel-indicator-border-color:       #fff;
```

   These are used to provide a white border around the default indicators, and then fill the active indicator with the background color.

2. Let's add a default background color variable here, so that we may fill the default indicators with our `$gray-light` value:

```
$carousel-indicator-bg:          $gray-light;
```

3. Then, we'll update the active background color:

```
$carousel-indicator-active-bg:    $gray-lightest;
```

4. Finally, we'll make the border color transparent:

```
@carousel-indicator-border-color: transparent;
```

5. Save, compile, and refresh.

At present, this has the effect of making all but the active item invisible:



Now for some work in the `_scss/includes/_carousel.scss` file:

1. In the `_scss/includes/_carousel.scss` file, move to the first set of rules for `.carousel-indicator` where we were previously working:

```
.carousel-indicators {
  position: absolute;
```

2. Look for the `li` selector nested under it. Here, let's edit several values. Specifically, we'll perform the following actions:
    1. Increase the width and height to 16px
    2. Remove the margin
    3. Add background color using our newly created variable `$carousel-indicator-bg`
    4. Remove the border line altogether (the transparent value we set for the border variable is now merely a failsafe)
    5. I've implemented these changes in the following code snippet:

```
.carousel-indicators {

position: absolute;

bottom: 0;

margin-bottom: 0;

li {

background-color: $carousel-indicator-bg;

&,

      &.active {
      border: 0;
      height: 16px;
      width: 16px;
      margin: 0;
     }
    }
    }
```

3. In Bootstrap's default CSS, the active indicators are a little larger (12 px) than normal indicators (10 px); because of that, you have to set the new size (16 px) for both the normal and active indicators. You can accomplish that by using the Sass and parent reference as in the preceding code snippet. Consider the following snippet of SCSS code:

```
.selector {
 &,
 &.active,
 &.otherstate {
   property: equal-for-all-states;
 }
}
```

4. This SCSS code compiles in CSS code like the one shown here:

```
.selector, .selector.active, .selector.otherstate {
  property: equal-for-all-states;
}
```

Save, and check out the result!



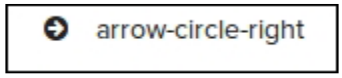
Carousel adjustments accomplished! We've learned a lot in the process—a lot about Bootstrap and perhaps a little about Sass as well.

Let's move on to the next section. What remains is considerably simpler.

Tweaking the columns and their content

Let's fine-tune the blocks of content under the three headings **Welcome!**, **Recent Updates**, and **Our Team**:

1. First, let's add the arrow-circle icon to the button in each of these three blocks. Recall that we're using Font Awesome for our icon selection.
2. Visit the Font Awesome documentation at `http://fortawesome.github.io/Font-Awesome/icons/`. You'll find the icon that we're after:

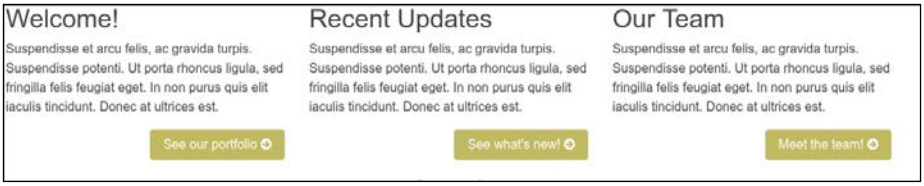


3. In the `html/pages/index.html` file, add an `i` tag with the appropriate classes inside each link. Here is the first one, which I've spaced out by adding an extra carriage return between elements:

```
<p>
  <a class="btn btn-primary pull-right" href="#">
    See our portfolio  <i class="fa fa-arrow-circle-
      right"></i>
  </a>
</p>
```

4. Repeat for each link.

You should now have the desired icon in each of the three buttons:



While we're at it, let's add a bit of vertical padding between the carousel and this section of text. Right now, it's pretty tight.

The question that comes up at this point is where best to compose the styles that we'll need for this; adding extra padding around page content sections will likely be a pretty normal practice for us now and in the future. Let's create a Sass file to hold these and other tweaks to the ordinary contents of pages (as it happens, we'll need this file for an additional and more important responsive adjustment, so it seems well justified):

1. Create a file named `scss/includes/_page-contents.scss`.
2. Save it in your `scss` folder alongside your other custom Sass files:

```
<image of the directory scss>
```

3. Comment the file:

```
//
// Page Contents
// -------------------------
```

4. Now, let's create a sensible class for this purpose and add our desired padding, including some padding for the bottom:

```
.page-contents {
  padding-top: 20px;
  padding-bottom: 40px;
}
```

5. Save the file.
6. Add the `scss/includes/_page-contents.scss` file to the imports in the `scss/main.scss` file. I'll add mine in a new section near the bottom of the file and include a helpful comment for orientation purposes:

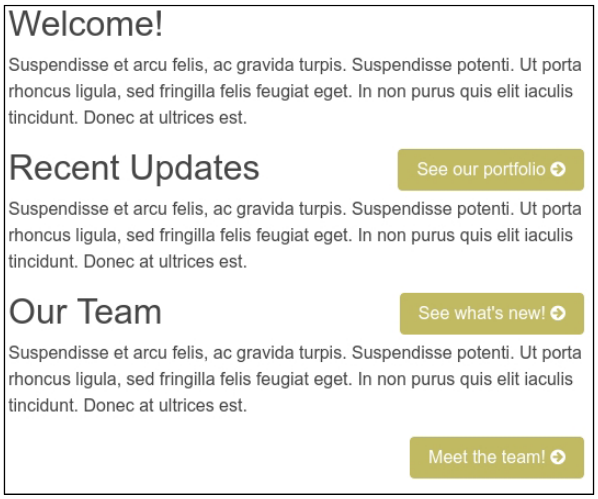
```
// Other custom files
@import "includes/page-contents";
```

7. Now let's add the necessary class to our markup. Open the `html/pages/index.html` file and add the class `page-contents` to the `div` with the class container, which follows just after the carousel including:

```
{{> carousel}}<!-- /#homepage-feature.carousel -->
<div class="page-contents container">
  <div class="row">
```

Save, and you should see the padding added.

Next, we need to tidy up the narrow-screen view of these blocks. Note that, when viewed in single-column layout, the headings do not clear the floated buttons:



Fixing this is just a little tricky. We might want to add a clear fix to the div containing each of these three blocks. However, that won't work because we need these blocks to float side by side once the viewport width is 768 px or higher.

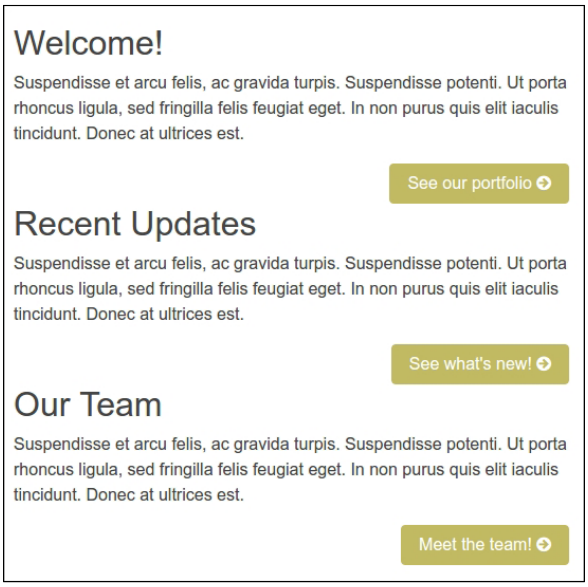
This calls for a media query. Knowing that our three-column view begins at the medium breakpoint, or 768 px, let's set a rule to clear floats when the window is 1 pixel below this breakpoint. As seen before, we can use Bootstrap's Sass media query mixin to do this.

While we're at it, let's also add some bottom padding to our columns so that they have a bit of extra vertical breathing room when stacked.

Inside the media query mixin, we'll add a CSS2 attribute selector to select all elements with a class that contains col-, so that the same rules will apply to a column of any size:

```
.page-contents {
  padding-top: 20px;
  padding-bottom: 40px;

  @include media-breakpoint-down(sm) {
    [class*="col-"] {
      clear: both;
      padding-bottom: 40px;
    }
  }
}
```

Save. The result is much improved!



Much better! Now let's move on to the footer!

Styling the footer

The biggest feature of the footer is our social icons. Font Awesome to the rescue!

Consulting the Font Awesome documentation, we find a slew of available icons under the category of Brand Icons. Here is the direct link: `http://fortawesome.github.io/Font-Awesome/icons/#brand`

Now we only need to replace the text for each social link in our footer, in the `html/includes/footer.html` file, with `i` elements, using the appropriate classes:

```
<ul class="social">
  <li><a href="#" ><i class="fa      fa-twitter"></i></a></li>
  <li><a href="#" ><i class="fa      fa-facebook"></i></a></li>
  <li><a href="#" ><i class="fa      fa-linkedin"></i></a></li>
  <li><a href="#" ><i class="fa fa-     google-plus"></i></a></li>
  <li><a href="#" ><i class="fa fa-     github-alt"></i></a></li>
</ul>
```

This updated markup puts our icons in place:



Now, perform the following steps to lay them out horizontally and align them to the center:

1. Create a new file, `scss/includes/_footer.scss`, to manage these styles.
2. Save the file to the `scss` directory.
3. Add an import variable for this file in `__main.less`:

```
// Other custom files
@import "includes/navbar";
@import "includes/carousel";
@import "includes/page-contents";
@import "includes/footer";
```

Now we'll write the styles we need. Let me simply lay them out, and then list what they do:

```
//
// Footer
// --------------------------

ul.social {
  margin: 0;
  padding: 0;
  width: 100%;
  text-align: center;
  > li {
    display: inline-block;
    > a {
      display: inline-block;
      font-size: 18px;
      line-height: 30px;
      @include square(30px); // see includes/mixins/_size.scss
      border-radius: 36px;
```

```
        background-color: $gray-light;
        color: #fff;
        margin: 0 3px 3px 0;
        &:hover,
      &:focus    {
          text-decoration: none;
          background-color: $link-hover-color;
        }
      }
    }
  }
```

Because Bootstrap tries to avoid both element and child selectors in Sass, you can consider rewriting the preceding SCSS code as follows:

```
.social {
  margin: 0;
  padding: 0;
  width: 100%;
  text-align: center;
}

.social-item {
 display: inline-block;
}

.social-link {
  display: inline-block;
  font-size: 18px;
  line-height: 30px;
  @include square(30px);
  // see includes/mixins/_size.scss
  border-radius: 36px;
  background-color: $gray-light;
  color: #fff;
  margin: 0 3px 3px 0;
  @include hover-focus {
    // bootstrap/scss/mixins/_hover.scss
    text-decoration: none;
    background-color: $link-hover-color;
    color: #fff;
  }
}
```

When you're using the latest SCSS code to compile your CSS code, you also have to

modify your HTML code according to it. Modify the HTML of the footer links in the `html/includes/footer.html` file so that it looks like the snippet shown here:

```
            <ul class="social">
              <li class="social-item"><a href="#" class="social-link"
><i class="fa fa-twitter"></i></a></li>
              <li class="social-item"><a href="#" class="social-link"
><i class="fa fa-facebook"></i></a></li>
              <li class="social-item"><a href="#" class="social-link"
><i class="fa fa-linkedin"></i></a></li>
              <li class="social-item"><a href="#" class="social-link"
><i class="fa fa-google-plus"></i></a></li>
              <li class="social-item"><a href="#" class="social-link"
><i class="fa fa-github-alt"></i></a></li>
            </ul>
```

Here's what's happening:

- The normal margin and padding is stripped away from the `ul`
- It is stretched to 100 percent width
- Its content is center aligned
- The list items are displayed inline to the block, thereby centering them
- The links are displayed inline to block, so that they fill up their available space
- The font size and line height are increased
- The width and height are set to 30px square, using a custom mixin (note that this mixin is copied from Bootstrap 3)
- To see this mixin, open `includes/mixins/_size.scss`, and you'll find the following relevant lines:

```
// Sizing shortcuts

@mixin size($width, $height) {
  width: $width;
  height: $height;
}

@mixin square($size) {
  @include size($size, $size);
}
```

- The `border-radius` property is set large enough to make the icons and their backgrounds appear circular

- The `background-color`, `color`, and `margin` properties are set
- The underline is removed from the `hover` and `focus` states, and the background color is altered to a lighter gray

With these steps done, let's polish off the footer by adding a healthy bit of top and bottom padding, and then center aligning the content in order to move our logo to the center, above the social icons:

```
footer[role="contentinfo"] {
  padding-top: 24px;
  padding-bottom: 36px;
  text-align: center;
}
```

The result is as follows:



Recommended next steps

Let me strongly recommend at least one additional step you'll need to take before taking a project like this to production. It's imperative that you take time to optimize your images, CSS, and JavaScript. These steps are not difficult:

- Compressing images takes just a bit of time, and it addresses the single largest cause for large page footprints. I've already used the save to web process option of Photoshop, but chances are you can squeeze a few more bytes out. In the code folder (`Build Process Manual`), you can see how to add an image compress task to your gulp build process..
- In addition, we badly need to remove unneeded Bootstrap Sass files from the import sequence in the `scss/includes/_bootstrap.scss file`, and then compress the resulting `main.css` file.
- Finally, we need to slim down our `plugins.js` file by replacing Bootstrap's all-inclusive `bootstrap.min.js` file with compressed versions of only the three plugins that we're actually using: `carousel.js, collapse.js`, and `transitions.js`. We then compress the final `plugins.js` file.

Combined, these steps can cut the footprint of this website by roughly half. In an age where speed matters—both for user retention and for SEO ranking—that's a big deal.

In addition, there is one other very sensible step you may want to take: We know that users of touch-enabled devices appreciate the ability to swipe their way forward and back through a carousel.

But, for the present moment, let's stop and celebrate.

Summary

Let's take stock of what we've accomplished in this lesson. We started a new Bootstrap project, powered by Panini, Sass, and Gulp, by using Bootstrap CLI. After that, we leveraged Bootstrap's responsive navbar, carousel, and grid system, and customized several of Bootstrap's Sass codes and mixins. You also learned how to create your own Sass files and folded them seamlessly into the project. Last but not least, you folded the Font Awesome icons into our workflow. At the end, you improved future maintenance of the site by implementing a thoughtful file organization scheme—all without creating code bloat.

With this experience under your belt, you're equipped to bend Bootstrap to your will, using its power to speed website development, and then customizing the design to your heart's content. In future lessons, we'll expand your experience further. First, however, let's take this design to create a complex business home page.

Assessments

1. Which of the following classes will assign the color of our primary brand color to a button?

    i. `btn-primary`

    ii. `button-primary`

    iii. `btn`

    iv. `bttn-primary`

2. Which of the following is the right code for a `media breakup` point?

    i.
    ```
    .include media-breakpoint-up(md) {
        padding-top: 0,
        padding-down: 0;
        }
    ```

    ii.
    ```
    @include media-breakpoint-up(md) {
        padding-top: 0;
    ```

```
        padding-bottom: 0;
        }
```

iii.
```
$include media-breakpoint-up(md) {
        padding-up: 0,
        padding-bottom: 0;
        }
```

iv.
```
&includes media-breakpoint-up(md) {
        padding-top: 0,
        padding-bottom: 0,
        }
```

3. Which class will float the link to the right, moving it into wider space to make it a more appealing target?

   i. `pull-xs-right`
   ii. `pulls-xs-right`
   iii. `pull-right`
   iv. `pulls-xl-right`

4. Which of the following statement is right for using the grey color to style elements across bootstrap?

```
a. $gray-dark= @373a3c;
b. .gray: #55595c;
c. $gray-light= #818a91;
d. @gray-lighter: #eceeef;
```

   i. Both a and c
   ii. Only c
   iii. Only d
   iv. Only a

5. What should be in the place of `'?'` in the following syntax:

```
?import "font-awesome/scss/font-awesome.
scss";
```

   i. .
   ii. #
   iii. @
   iv. $

9
Bootstrapping Business

We've built our portfolio site. Now, it's time to flesh out our portfolio with some projects that demonstrate the range of our powers. Let's now turn to designing a complex business home page.

Take a moment to survey the home pages of successful businesses, such as these:

- Zappos (http://www.zappos.com)
- Amazon (http://www.amazon.com)
- Adobe (http://www.adobe.com/)
- HP (http://www.hp.com)

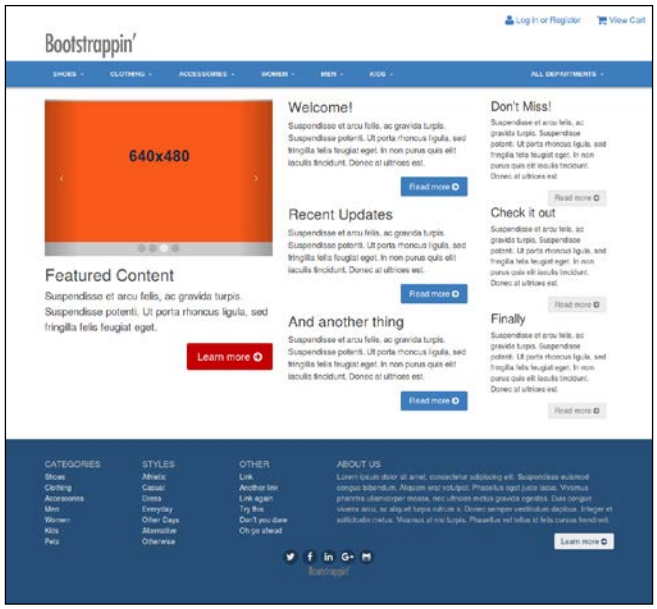
While each has its own approach, what these sites have in common is that they manage considerable complexity.

We can get a grasp of some common features by breaking the website down into three categories, as follows, based on regions of the page:

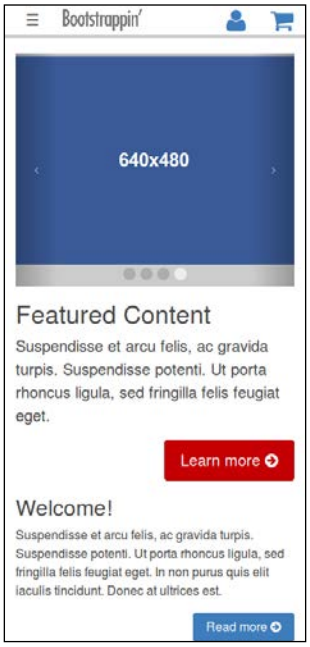
- **Banner/masthead**: This part contains the logo, main navigation with drop-down menus, a secondary or utility navigation, and a login or register option
- **Main content area**: This features a complex layout with at least three columns, if not more
- **Footer**: This is filled with multiple columns of links and information

Let's demonstrate our ability to manage this degree of complexity. To do so, we will take full advantage of Bootstrap's responsive 12-column grid system.

Here is the design we'll create, when viewed in medium and wide viewports:



In narrow viewports, it will adapt considerably, as shown in the following screenshot:



After that, we will perform the following steps:

1. Begin with a set of starter files based on the Portfolio project from *Lesson 8*, *Bootstrapping Your Portfolio*.
2. Create a complex banner with the logo above the navbar and a utility navigation in the top-right corner, in desktop viewports.
3. For smaller viewports, we'll enable our utility options to appear only as icons atop the collapsed responsive navbar.
4. Implement a business-style color scheme.
5. Make adjustments to both the responsive and desktop versions of the navbar.
6. Set up complex multicolumn grids for the main content and footer areas.

First things first: let's size up our project starter files.

Sizing up our beginning files

As with all the projects in this course, the beginning files for this project can be downloaded from the Packt Publishing website at `http://www.packtpub.com/support`. You'll find the files for this project in the `Lesson 9/start` folder.

These files are a copy of our results from *Lesson 8*, *Bootstrapping Your Portfolio*. Thus, we have the benefit of the following key components:

- A complete build process, including the Sass compiler and the Panini template engine
- Bootstrap SCSS and JavaScript files
- The Panini HTML templates

In addition to these key assets, we have some of the custom Sass touches we created during the project in *Lesson 8*, *Bootstrapping Your Portfolio*. They can be found in the following files, which are present in the `scss` and `scss/includes` directories:

- `_main.scss`: This is customized to import Bootstrap's Sass files from the `bower_components/bootstrap/scss` directory, as well as Font Awesome font icons and our custom Sass files
- `_carousel.scss`: This file has custom touches on the carousel padding, background, and indicators
- `_footer.scss`: This file contains styles for the layout and design of the logo and social icons
- `_navbar.scss`: This has adjusted padding in the `.navbar-brand` class to enable the navbar logo to fit

- `_page-contents.scss`: This contains styles to ensure that columns with floated buttons clear one another in narrow single-column layouts
- `_variables.scss`: This has custom versions of gray and some adjustments to variables for the navbar and carousel

Font Awesome font icons include the following:

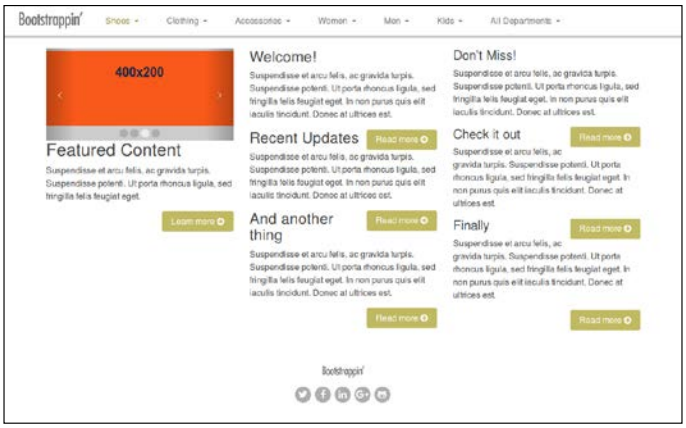
- Icon fonts in the `fonts` directory
- Sass files in the `bower_components/font-awesome` directory
- In order to start using these files, you should run the following commands in your console:

```
bower installing
npm install
```

After that, you can run the `npm start` or `bootstrap watch` command to compile your project and start watching your changes in the browser.

Setting up the basics of your design

We start modifying the result of *Lesson 8*, *Bootstrapping Your Portfolio*, and should end up with the basics, similar to that shown in the following screenshot:



Note the following features:

- A complex navbar that has seven main `nav` items, each with a drop-down menu
- The first of the three columns is equipped with a carousel, followed by a heading, paragraph, and button

- The second and third columns, which have headings, paragraphs, and Read more -> buttons
- A footer that has the logo and social icons

You'll recognize elements we've already worked with in *Lesson 8*, *Bootstrapping Your Portfolio*. The carousel is now smaller—constrained by its containing column. Otherwise, the markup is the same.

Adding drop-down menus to our navbar

Bootstrap's JavaScript Dropdown Plugin enables you to create drop-down menus with ease. You can also add these drop-down menus to your navbar.

Open the `html/includes/header.html` file in your text editor. Notice that the Gulp build process uses the Panini HTML compiler to compile our HTML templates into HTML pages. Panini is powered by the Handlebars template language. In your templates, you can use helpers, iterations, and custom data. In this example, you'll use the power of Panini to build the navbar items with drop-down menus.

First, create an `html/data/productgroups.yml` file that contains the titles of the navbar items:

```
- Shoes
- Clothing
- Accessories
- Women
- Men
- Kids
- All Departments
```

The preceding code is written in the YAML format. YAML is a human-readable data serialization language that takes concepts from programming languages and ideas from XML; you can read more about it at the following URL: `http://yaml.org/`.

Using the preceding data, you can use the following HTML and template code to build the navbar items:

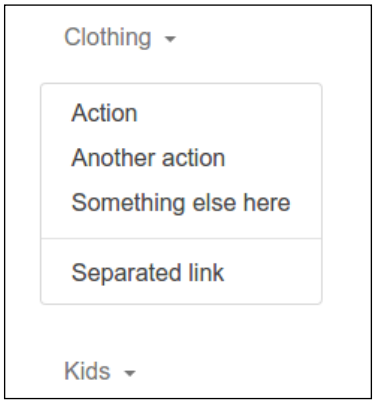
```
  <ul class="nav navbar-nav navbar-toggleable-sm collapse"
  id="collapsiblecontent">
  {{#each productgroups}}
    <li class="nav-item dropdown {{#ifCond this 'Shoes'}}active{{/
ifCond}}">
      <a class="nav-link dropdown-toggle" data-toggle="dropdown"
href="#"
      role="button" aria-haspopup="true" aria-expanded="false">
```

```
        {{ this }}
        </a>
          <div class="dropdown-menu">
            <a class="dropdown-item" href="#">Action</a>
            <a class="dropdown-item" href="#">Another action</a>
            <a class="dropdown-item" href="#">Something else here</a>
            <div class="dropdown-divider"></div>
            <a class="dropdown-item" href="#">Separated link</a>
          </div>
      </li>
    {{/each}}
    </ul>
```

The preceding code uses a `(for) each` loop to build the seven navbar items, and each item gets the same drop-down menu. The Shoes menu gets the active class. Handlebars, and therefore Panini, does not support conditional comparisons by default. The `if` statement can only handle a single value, but you can add a custom helper to enable conditional comparisons. The custom helper, which enables us to use the `ifCond` statement, can be found in the `html/helpers/ifCond.js` file. Read my How to set up Panini for different environments blog post, at `http://bassjobsen.weblogs.fm/set-panini-different-environments/`, to learn more about Panini and custom helpers.

The HTML code for the drop-down menu is in accordance with the code for drop-down menus described for the Dropdown plugin at `http://getbootstrap.com/components/dropdowns/`.

The navbar collapses for smaller screen sizes. By default, the drop-down menus look the same on all grids:



Setting the bottom border for the page header

Create a new Sass partial file and add the following SCSS code to it to give our page header a clear boundary:

```
header[role="banner"] {
  border-bottom: 4px solid $gray-lighter;
}
```

Adding images with holder.js

One other wrinkle is that I've used the excellent `holder.js` JavaScript plugin to dynamically generate placeholder images for our carousel.

You can install the `holder.js` plugin with Bower by running the following command in your console:

```
bower install holderjs --save-dev
```

After installing the plugin, you can use the `compile-js` Gulp task in `Gulpfile.js` to link it together with your other JavaScript code into the `app.js` file as follows:

```
gulp.task('compile-js', function() {
  return gulp.src([
        bowerpath+ 'jquery/dist/jquery.min.js',
        bowerpath+ 'tether/dist/js/tether.min.js',
        bowerpath+ 'bootstrap/dist/js/bootstrap.min.js',
        bowerpath+ 'holderjs/holder.min.js', // Holder.js for project
        development only
        'js/main.js'])
    .pipe(concat('app.js'))
    .pipe(gulp.dest('./_site/js/'));
});
```

If you examine the markup, you'll see near the bottom of the page that I've included the `holder.js` script right before `plugins.js`, as follows:

```
<!-- Holder.js for project development only -->
<script src="js/vendor/holder.js"></script>
```

We won't be using placeholder images in our final production site, so it makes sense to link it separately with a prominent comment.

With `holder.js` in place, we can conveniently build image tags that reference `holder.js` as their source. The remainder of the pseudo-URL specifies dimensions, color, and filler text, as follows:

```
<img src="holder.js/600x480/auto/vine/textmode:literal"
    alt="Holder Image">
```



For more information about `holder.js`, consult the documentation at `https://github.com/imsky/holder`.

With these elements in place—and thanks in particular to Bootstrap's ready repertoire of styles and behaviors—we're starting out in good shape. Let's get to the details.

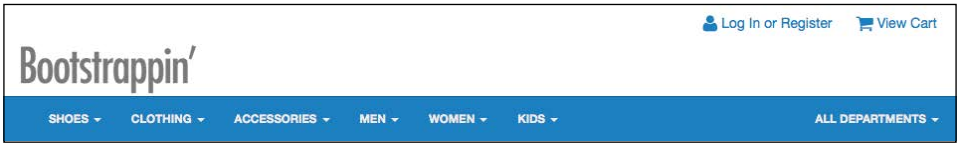
First, we'll reposition our navbar within a more complex banner design.

Creating a complex banner area

Let's start from the top and create our complex banner area with the following features:

- A site logo positioned above the navbar for desktops and larger viewports
- A navbar with many menu items, including drop-down menus
- A utility navigation area
- A login form with username and password
- An option to register

Here is the mockup of our desired end goal on a desktop-width viewport:



On a narrow viewport, it will adjust to this:



We'll start by working on a new arrangement for our top logo.

Placing a logo above the navbar

In this new design, we need a logo in two spots, for two contexts:

- For desktop and widescreen viewports, we want the logo to display above the navbar
- For tablet and phone viewports, we want the logo to display within the responsive navbar

Thanks to Bootstrap's responsive utility classes, we can do both! Here's how:

1. Open the `html/includes/header.html` file in your editor.
2. Move the logo and toggle button outside the `nav` element and wrap them with `<div class="container">...</div>` to constrain it within Bootstrap's centered grid space.
3. `Remove the width attribute from the ig element of the logo.`
4. Then wrap the `<ul class="nav navbar-nav"></ul>` with `<div class="container">...</div>` too.
5. Move `navbar-toggleable-sm` and collapse the classes to the second container's div.
6. At the end, your HTML code in the `html/includes/header.html` file should look as follows:

```
        <header role="banner">
        <div class="container">
        <button class="navbar-toggler hidden-md-up" type="button"
        data-toggle="collapse" data-target="#collapsiblecontent">
        ☰
        </button>
        <a class="navbar-brand" href="/"><img src="{{root}}/
images/logo.png"
        alt="Bootstrapping"></a>
        </div>
        <nav class="navbar navbar-full" role="navigation">
        <div class=  "container navbar-toggleable-sm collapse"
        id="collapsiblecontent">
        <ul class="nav navbar-nav">
        ...
        </ul>
        </div>
        </nav>
        </header>
```

7. After your HTML changes, you can use the power of Sass again to style the

logo depending on the width of the viewport. Edit the following SCSS code in the `scss/includes/_header.scss` file:

```
header[role="banner"] {
  .navbar-brand {
  > img {
      width: 120px;
      padding-left: $spacer-x;
      @include media-breakpoint-up(md) {
        padding-left: 0;
        width: 180px;
      }
    }
  }
}
```

In the preceding steps, a couple of predefined Bootstrap classes are used. The `hidden-md-up` helper class hides content for the medium viewport and up and so ensures that the `toggle button` is only visible on the small viewports. On the other hand, the `navbar-toggleable-sm` class only affects small and extra-small viewports.

On narrow viewports, the logo gets a width of 120 pixels. For medium grids and larger, the left padding of the logo is removed and the width is set to 180 pixels.



Recall that our original logo image is large, about 900 pixels wide. We've resized it to 120 pixels wide via the width attribute (we could alternatively use CSS rules) in order to pack its pixels tighter for retina screens.

Save the changes and refresh the page, and you should see these results! In medium and large viewports, the larger logo will appear:



In small and extra-small viewports, a small version of the logo will appear:



Ah, the beauty of Bootstrap!

Now, let's make some adjustments to our navbar.

Reviewing and checking navbar drop-down items

The navbar, with its seven items and submenus, reflects the needs of a large complex website.

The markup for the drop-down menus is taken directly from the Bootstrap drop-down documentation at `http://getbootstrap.com/components/dropdowns/`.

If you look at our resulting markup, you'll notice these special classes and attributes:

- `class="dropdown"` on the parent li
- `class="dropdown-toggle"` on the link
- `attribute="data-toggle"` also on the link
- `class="dropdown-menu"` on the submenu div element
- `class="dropdown-item"` on each drop-down menu item

Here is the resulting markup:

```
<li class="nav-item dropdown">
  <a class="nav-link dropdown-toggle" data-toggle="dropdown" href="#"
  role="button" aria-haspopup="true" aria-expanded="false">
    Shoes
  </a>
  <div class="dropdown-menu">
    <a class="dropdown-item" href="#">Action</a>
    <a class="dropdown-item" href="#">Another action</a>
    <a class="dropdown-item" href="#">Something else here</a>
    <div class="dropdown-divider"></div>
    <a class="dropdown-item" href="#">Separated link</a>
  </div>
</li>
```

Also note that the small drop-down indicator is a CSS triangle. Bootstrap uses the following SCSS code in the `bower_components/bootstrap/scss/_` to create these triangles:

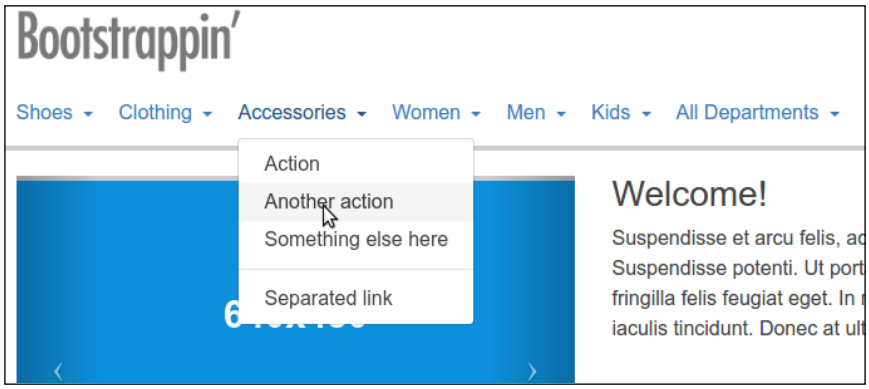
```
.dropdown-toggle {
  // Generate the caret automatically
  &::after {
```

```
    display: inline-block;
    width: 0;
    height: 0;
    margin-right: .25rem;
    margin-left: .25rem;
    vertical-align: middle;
    content: "";
    border-top: $caret-width solid;
    border-right: $caret-width solid transparent;
    border-left: $caret-width solid transparent;
  }

  // Prevent the focus on the dropdown toggle when closing dropdowns
  &:focus {
    outline: 0;
  }
 }
}
```



When working with plugins and components, make sure that the required SCSS and JavaScript code is included in your project. Sass partials are imported into `main.scss` file. The import `scss/includes/_bootstrap.scss` in its turn imports the SCSS code for Bootstrap's components. The JavaScript plugins are included in the file in the Gulp task.

With the Sass, JavaScript, and HTML markup in place, our navbar and its dropdowns should presently look and work as shown in the following screenshot (note that Bootstrap dropdowns responds on click):



Now that we're familiar with the HTML markup structure and have ensured everything's working as it should, let's move the **All Departments** menu to the right-hand end of the navbar, setting it apart from the others.

To do this, we need to nest this list item within its own unordered list, as follows:

1. Before the All Departments list item, close the `ul` tag for `ul class="nav"`, which surrounds all previous menu items.
2. Start a new `ul` tag with the `nav` and `navbar-nav` classes before the All Departments menu item. Once this opening tag is added, it will nest this list item in the standard structure for navigation menus.
3. In addition to the `nav` and `navbar-nav` classes, add a third class, `pull-right`, which is a convenient Bootstrap utility class, to float an element to the right.

The newly added lines are highlighted in the following snippet—after which I'll include the original list item and link in context:

```
  <ul class="nav navbar-nav">
  {{#each productgroups}}
    {{#ifCond this 'All Departments'}}</ul><ul class="nav navbar-nav
    pull-md-right">{{/ifCond}}
    <li class="nav-item dropdown {{#ifCond this 'Shoes'}}active{{/
ifCond}}">
      <a class="nav-link dropdown-toggle" data-toggle="dropdown"
href="#"
      role="button" aria-haspopup="true" aria-expanded="false">
      {{ this }}
      </a>
        <div class="dropdown-menu">
          <a class="dropdown-item" href="#">Action</a>
          <a class="dropdown-item" href="#">Another action</a>
          <a class="dropdown-item" href="#">Something else here</a>
          <div class="dropdown-divider"></div>
          <a class="dropdown-item" href="#">Separated link</a>
        </div>
    </li>
  {{/each}}
  </ul>
```

Notice that we've used the `IfCond` helper again to make sure that the `</ul><ul class="nav navbar-nav pull-md-right">` snippet is only inserted before the beginning of the All Departments category.

Save the changes and inspect the results in your browser. If you have not run the

`bootstrap watch` or `gulp` command already, you should run it now. You should see the All Departments drop-down menu item float to the right-hand end of the navbar, as follows:



Instead of modifying your HTML code, you can also use the following SCSS code to accomplish the same thing:

```
.nav-item:last-child {
  float:right;
}
```

So far so good! Now, let's add our utility navigation.

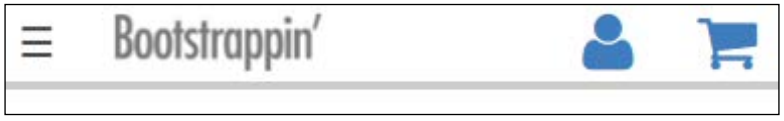
Adding utility navigation

This project requires utility navigation to allow users to log in or register and to view their carts.

On medium, large, and extra-large viewports, we'll place this utility navigation in the very top-right corner of our banner area, as follows:



On smaller screens, we'll display icons at the far right of the collapsed navbar:



Notice that the collapsed navbar has a different color scheme; we'll discuss how to do this later on.

Now let's set the navbar changes up.

First, to give the logo a little more space in the the `scss/includes/_header.scss` file, set top-padding for only the larger viewports, as follows:

```
header[role="banner"] {
  .navbar-brand {
    > img {
      width: 120px;
      padding-left: $spacer-x;
      @include media-breakpoint-up(md) {
        padding-top: $spacer-y * 3;
        padding-left: 0;
        width: 180px;
      }
    }
  }
```

Then, still working in the `html/includes/header.html` file, we need to add the markup for our utility navigation within the banner, just after the `navbar-brand` attribute. Here is the full markup, beginning with the opening `header` tag for our banner area. I've highlighted the new `utility-nav` markup in the following code snippet:

```
<header role="banner">
  <div class="container">
   <button class="navbar-toggler hidden-md-up" type="button"
   data-toggle="collapse" data-target="#collapsiblecontent">
    ☰
   </button>
   <a class="navbar-brand" href="/"><img src="{{root}}/images/logo.
png"
   alt="Bootstrapping"></a>
   <div class="utility-nav">
      <ul>
        <li><a href="#" ><i class="icon fa
        fa-user fa-lg"></i> Log In or Register</a></li>
        <li><a href="#" ><i class="icon fa fa-shopping-cart
        fa-lg"></i> View Cart</a></li>
      </ul>
   </div>
  </div>
...
</header>
```

Note a few things about this markup:

- The `utility-nav` class is simply created for our use. It is not a Bootstrap-specific class and has no specific styles attached.
- I've included Font Awesome's user and shopping cart icons and added the class of `fa-lg` to increase their size by 33 percent. See Font Awesome's documentation on this at `http://fontawesome.io/examples/#larger`.

Save the changes and inspect the results in your browser, and you should see our new `utility-nav` class appear on the right-hand side of the `logo`, as follows:



Now, to complete the layout and related adjustments, we need to apply some custom styles. We need a new file to manage styles for our banner area.

We need to set the position of `.utility-nav` to absolute, at the top right. We'll specify `header[role="banner"]` as the context for these styles. Add the following SCSS code to `the scss/includes/_header.scss` file:

```
header[role="banner"] {
 // Banner Area Styles
 .utility-nav {
   position: absolute;
   top: $spacer-y;
   right: 0;
  }
}
```

1. Remove bullets from the unordered list.
2. Float the list items on the left.
3. Add padding to the link.
4. Remove underlines from the hover effect.

```
.utility-nav {
  ul {
    list-style: none;
    li {
      float: left;
```

```
            a {
              padding: 0 $spacer-x;
              @include hover {
                text-decoration: none;
              }
            }
          }
        }
      }
```

Save the changes and ensure that it compiles. In the preceding code, we've set the padding for the anchors to `padding: 0 $spacer-x;`. We could accomplish this by adding Bootstrap's utility classes for spacing into the HTML code.

The following HTML code also sets a padding of `$spacer-x` on the left- and right-hand sides of the `<a>` element:

```
<a href="#"  class="p-1-x">
```

Then make sure your browser window is at desktop width. You should see your `utility-nav` class take its place at the top right of the banner:



That takes care of medium viewports and larger. Now, let's address the needs of the collapsed responsive navbar.

Making responsive adjustments

On small screens, the elements of our page header may overlap:



We need to move the toggle to the left-hand side of our navbar. This can be done as follows:

1. Open the `scss/includes/_header.scss` file in your editor and add the following SCSS code to it:

```
header[role="banner"] {
.navbar-toggler {
float: left;
}
}
```

2. Save and compile these changes, and you'll see the navbar toggle shift to the left end of the collapsed navbar, as shown in the following screenshot:



So far so good.

Now to address the problem of crowding by hiding the text for all devices except for screen readers on the collapsed navbar. In an uncluttered collapsed navbar, the icons will be enough to communicate the point, especially if we make the icons larger. Let's do that:

1. In the `html/includes/_header.html` file, place `span` tags around the text within each link of our `utility-nav` class, as follows:

```
        <li><a href="#" ><i class="icon fa
        fa-user fa-lg"></i> <span>Log In or
    Register</span></a></li>
        <li><a href="#" ><i class="icon fa
        fa-shopping-cart fa-lg"></i> <span>

   View Cart</span></a></li>
```

2. This will give us a handle for our upcoming style adjustment.
3. Now, in `_headers.scss`, we'll add a media query to target these `span` tags. Thanks to the power of Sass, we can nest the media query precisely where we want it to do its work. We'll use Bootstrap's `@media-breakpoint-down(sm)` mixin, setting a `max-width` query to the small breakpoint value minus one, since at this point our navbar makes the transition from collapsed to expanded. Within this media query, we'll use the `sr-only` utility class

as a mixin to hide text from all devices except screen readers. (See the documentation on this class at `http://getbootstrap.com/components/utilities/#screen-readers-and-keyboard-users`.)

4. Instead of the `sr-only` mixin, you can also add the `sr-only` class to your HTML. The `sr-only-focusable` mixin and `sr-only-focusable` CSS class are available to make content hidden by the `sr-only` class visible on focus, which is useful for keyboard-only users.
5. Here is the code snippet:

```
header[role="banner"] {
  @include media-breakpoint-down(sm) {
    top: 0;
    span {
      @include sr-only;
    }
  }
}
```

6. This will hide the text between our `span` tags, leaving us only with the icons!
7. Now, we will increase the size of the icons and add some line height to position them vertically. We'll do this within the same media query:

```
header[role="banner"] {
  @include media-breakpoint-down(sm) {
    top: 0;
    span {
      @include sr-only;
    }
    .icon {
      font-size: 2em;
      line-height: 1.2;
    }
  }
}
```

Save your changes, run the `bootstrap watch` command if you haven't already done so, and you should see the following result:



Take a minute to resize your browser window back and forth across the breakpoint.

You should see the entire banner and navbar adjust seamlessly across the breakpoint.

If you're like me, it's hard not to be pleased with a framework that enables us to be this efficient at building such an adept and responsive interface.

Next up, we need to begin implementing the color scheme.

Implementing the color scheme

We've been provided with a business-friendly palette of blue, red, and gray. Let's work these colors into our color variables:

1. Open `scss/includes_variables.scss` in your editor. We'll be working at the beginning of the file, in the color variables.
2. Let's review the range of grays we have available. If you've begun with the `Lesson 8/finish` files, you'll see we've carried these variables over from *Lesson 8*, *Bootstrapping Your Portfolio*. They served us well there, and we'll make use of them again here:

```
x`
// -------------------------


@gray-darker:            #222; // edited
@gray-dark:              #454545; // edited
@gray:                   #777; // edited
@gray-light:             #aeaeae; // edited
@gray-lighter:           #ccc; // edited
@gray-lightest:          #ededed; // edited
@off-white:              #fafafa; // edited
```

3. Now, below the grays, let's fold in our new brand colors. We'll modify the value for `@brand-primary` and create a `@brand-feature`variable for red:

```
@brand-primary:          #3e7dbd; // edited blue
@brand-feature:          #c60004; // added new red
```

4. Now, let's adjust our link-hover color so that it will lighten (rather than darken) the `@brand-primary` color, which is already dark:

```
// Links
// -------------------------
@link-color:            @brand-primary;
@link-color-hover:      lighten(@link-color, 15%);
```

5. Finally, let's define the colors for our navbar. We'll create two sets of variables the `-xs-` variables for the small screens and the `-md-` variables for the larger screens:

```
        // Navbar
        $navbar-xs-color:             $body-color;
        $navbar-xs-bg:                #fff;
        $navbar-md-color:             $gray-lightest;
        $navbar-md-bg:                $brand-primary;

        // Navbar links
        $navbar-xs-color:                $navbar-xs-color;
        $navbar-xs-hover-color:          $navbar-xs-color;
        $navbar-xs-hover-bg:             darken($navbar-xs-bg,
5%);
        $navbar-xs-active-color:         $navbar-xs-color;
        $navbar-xs-disabled-color:       $navbar-xs-hover-bg;

        $navbar-md-color:                $navbar-md-color;
        $navbar-md-hover-color:          $navbar-md-color;
        $navbar-md-hover-bg:             darken($navbar-md-bg,
5%);
        $navbar-md-active-color:         $navbar-md-color;
        $navbar-md-disabled-color:       $navbar-md-hover-bg;
```

Having set up these fundamental color variables, we're ready to work on our navbar.

Styling the collapsed navbar

While still in `_navbar.less`, search for `// Navbar`, which will take you to the navbar variables. Note that most of the standard values specified here will affect both the collapsed responsive navbar for small viewports and the expanded navbar for wider viewports.

We want the background, text, and link colors for the collapsed responsive navbar to remain largely consistent with the default values but then change to our blue background and a light text color for medium and larger viewports.

We'll develop a responsive color scheme to accomplish the preceding color changes based on the viewport.

Open the `scss/includes/_navbar.scss` file and edit the default values for the small viewport as follows:

```
// responsive color scheme
.navbar {
  background-color: $navbar-xs-bg;
  color: $navbar-xs-color;

  .nav-link
  {
    @include hover-focus-active {
      background-color: $navbar-xs-hover-bg;
    }
  }
}
```

As you can see, we'll use the `-xs-` variables now. For medium and large viewports—where our navbar stretches out horizontally below the logo—we want our navbar to take on the blue color. Using Bootstrap's media query mixins again, we can change the colors for the larger viewports, as follows:

```
// responsive color scheme
.navbar {
  background-color: $navbar-xs-bg;
  color: $navbar-xs-color;

  .nav-link
  {
    @include hover-focus-active {
      background-color: $navbar-xs-hover-bg;
    }
  }

  @include media-breakpoint-up(md) {
    background-color: $navbar-md-bg;
    color: $navbar-md-color;

    .nav-link
    {
      color: $navbar-md-color;
      @include hover-focus-active {
        background-color: $navbar-md-hover-bg;
      }
    }
  }
}
```

Customizing the drop-down menus

We will use the power of Sass again to customize the drop-down menus for smaller screens. Edit the following code in the `scss/includes/_navbar.scss` file:

```
@include media-breakpoint-down(sm) {
  .navbar {

    .nav-item + .nav-item {
      margin-left: 0;
    }

    .dropdown {
      position: initial;
    }

    .dropdown-menu {
      position: initial;
      z-index: initial;
      float: initial;
      border: initial;
      border-radius: initial;
    }
  }
}
```

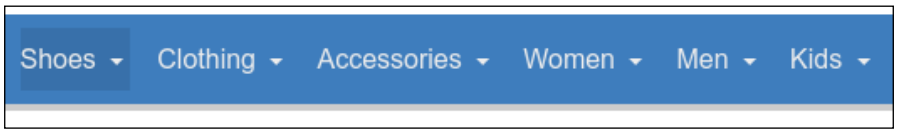
In the preceding code, we've used Bootstrap's media query mixins to reset the styles for screens smaller than 768 pixels. The initial value the initial value of a property to an element. So the preceding code sets the position to again and removes the float, border, and border-radius, and the drop-down menu will look like a "normal" list now:



Fantastic! Now we can address the horizontal navbar.

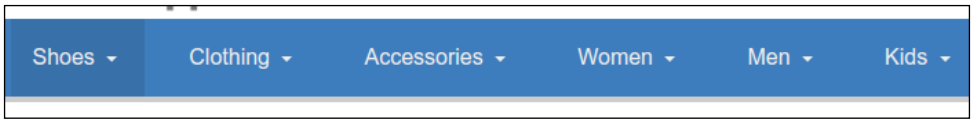
Styling the horizontal navbar

When you move your mouse over the navbar links, you'll find that the hover background color of the link is smaller than the height of the navbar:



You can solve this issue by applying padding on the links instead of the navbar. Use the following SCSS code in the file to change the padding:

```
.navbar {
  @include media-breakpoint-up(md) {
    padding-top: 0;
    padding-bottom: 0;
  }
  .nav-link
  {
    padding: $spacer;
    @include media-breakpoint-only(md) {
      padding: $spacer-y ($spacer-x / 2);
    }
  }
}
```



And finally, let's transform the text to upper case, reduce its size a bit, and make it bold. In `_navbar.scss`, add these highlighted lines:

```
.nav-link
{
  padding: $spacer;
  @include media-breakpoint-up(md) {
   text-transform: uppercase;
   font-size: 82%;
   font-weight: bold;
  }
}
```



Our banner and navbar are complete!

Enabling Flexbox support

Bootstrap 4 comes with optional flexbox support. You can simply enable flexbox support by declaring `$enable-flex: true;` in the `scss/includes/_variables` file. If you do so, you'll have clear the floats of the container of the navbar because of we've implement flexbox support for it. You can clear the floats by adding the following SCSS code to the file:

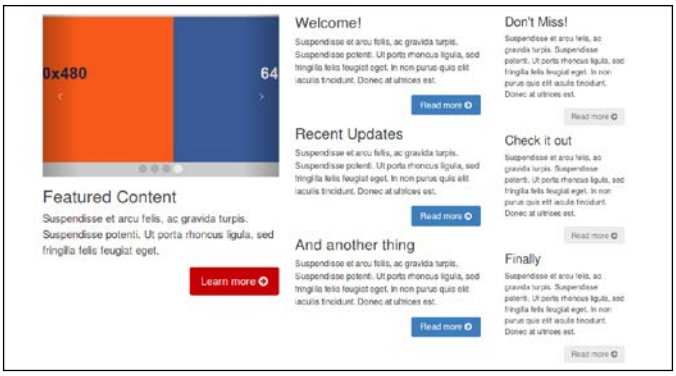
```
header[role="banner"] {
  // header container do not use the flexbox layout, so floats have to
be cleared
  @if $enable-flex {
    .container {
      @include clearfix();
    }
  }
}
```

Now it's time to move on to the main content of our page.

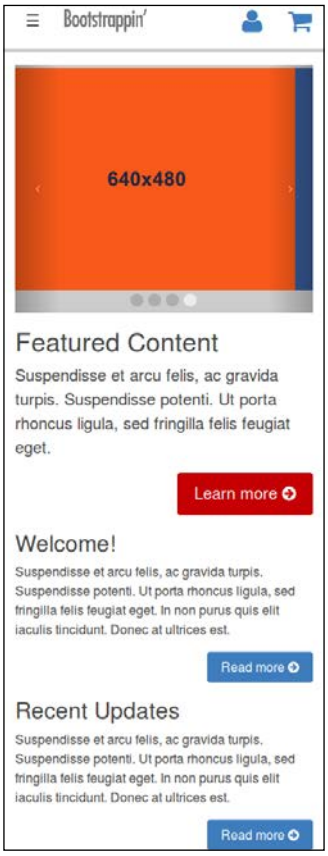
Designing a complex responsive layout

Let's imagine we've emerged from client meetings with a plan to organize the home page content in three tiers, ranked by importance.

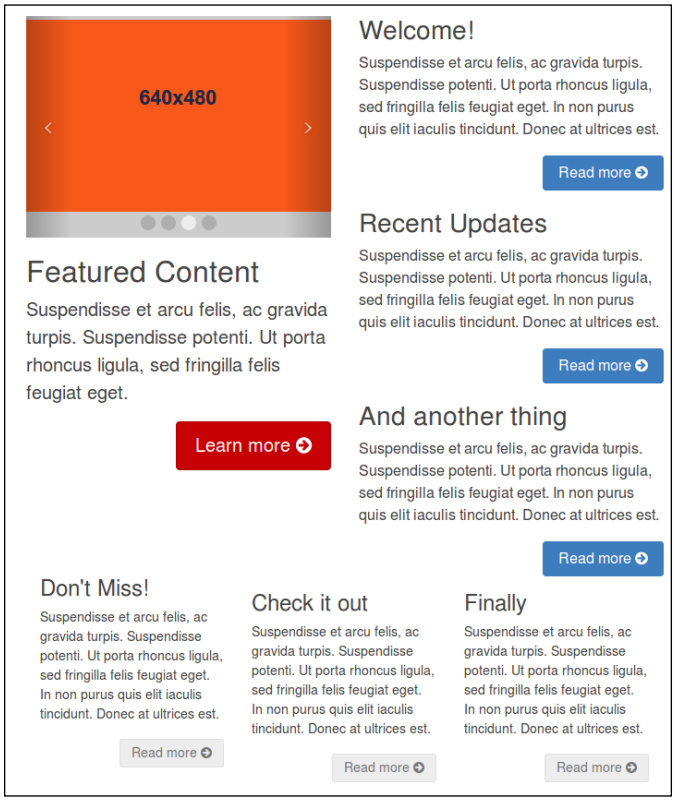
In medium and wide viewports, this content will be laid out in three columns, as shown in the following screenshot:



In a narrow viewport, these will be laid out one after another, in a single vertical column:



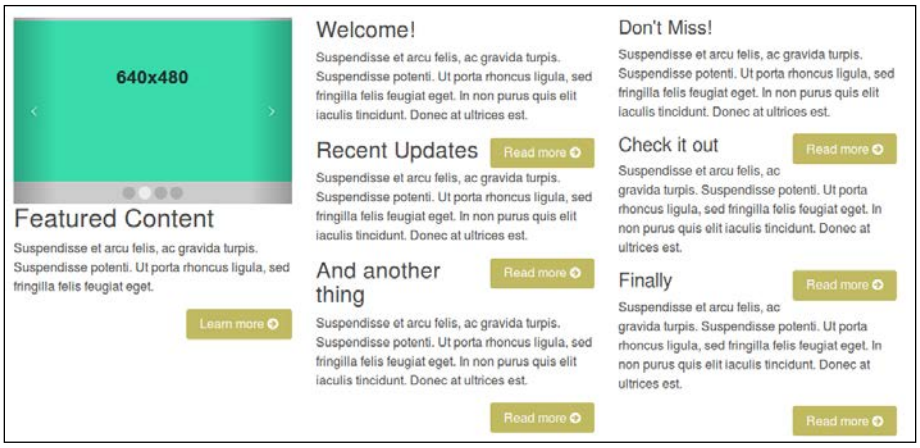
And in a small and medium `tablet-width` viewport, we'll arrange the content in two side-by-side columns, with the third tier of content laid out beneath it as a horizontal row, as shown in the following screenshot:



To get us started, I've provided the basic markup for three equal columns. Let's review what we have and then adapt it to the needs of this design. We'll begin with the three-column layout for medium and wide viewports.

Adjusting the large and extra-large layout

Currently, in large and extra-large viewports, our three columns are equal in width, font size, button size, and color. As a result, the presentation lacks visual hierarchy, as seen in the following screenshot:



We can take significant strides by adjusting column width, font size, button size, and color to establish a clearer hierarchy between these tiers of content. Let's do that. We'll start by adjusting column widths:

1. In `html/pages/index.html`, search for the `section` tag for the primary content:

   ```
   <section class="content-primary col-md-4">
   ```

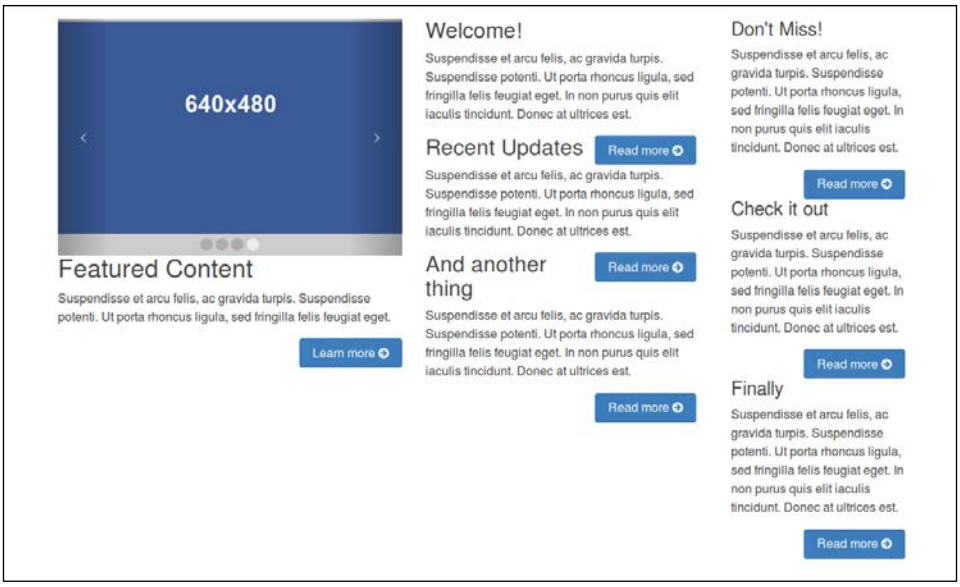
2. Note that the `col-md-4` class sets the width of this column to one-third of the width of the parent element, beginning at the small viewport width (768 pixels and up).
3. We want to save the three-column layout for the large and extra-large viewports (992 pixels and up), and we want this first column to be wider than the others.
4. Edit the `col-md-4` class to read `col-lg-5`, as follows:

   ```
   <section class="content-primary col-lg-5">
   ```

5. This will set this column to 5/12 width with the medium viewport and larger.
6. Now search and find the opening `section` tags for the next two columns and adjust the column classes to `col-lg-4` and `col-lg-3` respectively:

   ```
   <section class="content-secondary col-lg-4">
   ...
   <section class="content-tertiary col-lg-3">
   ```

Save, refresh, and you'll see the desired visual hierarchy in the width of our columns:



Adjusting the medium layout for tablet-width viewports




```
.navbar {
   @include media-breakpoint-only(md) {
     .nav-item + .nav-item {

        margin-left: 0;

}
    }
    .nav-link
    {
      padding: $spacer;
      @include media-breakpoint-only(md) {
        padding: $spacer-y ($spacer-x / 2);
      }
```

```
    }
}

@include media-breakpoint-only(md) {
   padding: $spacer-y ($spacer-x / 2);
}

@media (min-width: 768px) and (max-width: 991px) {
   .navbar .nav-link {
   padding: 1rem 0.5rem;
   }
}
```



You may have noticed that your file contains a lot of media queries now. Sass does not merge media queries, so the compiled CSS code contains a lot of media queries too. Merging the same CSS media query rules into one media query rule may be a performance improvement for your CSS code. The `css-mqpacker` node package can process your CSS code pack's same CSS media query rules into one media query rule.

You can run the package with the `gulp-postcss` package just like the autoprefixer plugin. More information about `css-mqpacker` and how to integrate it in your Gulp build process can be found at `https://www.npmjs.com/package/css-mqpacker`.

A `media-breakpoint-between()` mixin is available too. The `media-breakpoint-between()` mixin enables you to target a range of grids between two breakpoints. The `@include media-breakpoint-only(sm,md){}` call targets both the small and medium grids.

```
<button class="navbar-toggler hidden-lg-up" type="button"
data-toggle="collapse" data-target="#collapsiblecontent">

<div class="container navbar-toggleable-md collapse"
id="collapsiblecontent">

.navbar {
   @include media-breakpoint-down(md) {
   .navbar-brand,
   .nav-item {
   float: none;
   > img {
     display: inline-block;
     }
    }
    // dropdown menus
```

```
      .nav-item + .nav-item {
        margin-left: 0;
      }
      .dropdown {
        position: initial;
        }
        .dropdown-menu {
          position: initial;
    z-index: initial;
    float: initial;
    border: initial;
    border-radius: initial;
    }
        }
  }
```

```
    .navbar-brand {
      > img {
        width: 120px;
        padding-left: $spacer-x;
        @include media-breakpoint-up(lg) {
          padding-top: $spacer-y * 3;
          padding-left: 0;
          width: 180px;
        }
      }
    }
```

```
{{#ifCond this 'All Departments'}}</ul><ul class="nav navbar-nav pull-
lg-right">{{/ifCond}}
```

```
<section class="content-primary col-md-6 col-lg-5">
  ...
</section>
<section class="content-secondary col-md-6 col-lg-4">
  ...
</section>
<section class="content-tertiary col-md-12 col-lg-3">
  ...
</section>
```

The `col-md-*` classes in the preceding code only affect the medium, because the `col-lg-*` classes overwrites them for the larger grids, according to mobile first coding.

```
<article class="col-md-4 col-lg-12">...</article>

<section class="content-tertiary col-md-12 col-lg-3">
<div class="row">
<article class="col-md-4 col-lg-12">
<h4>Don't Miss!</h4>
<p>Suspendisse et arcu felis, ac gravida turpis. Suspendisse potenti.
    Ut porta rhoncus ligula, sed fringilla felis feugiat eget. In non
purus
    quis elit iaculis tincidunt. Donec at ultrices est.</p>
  <p><a class="btn btn-primary pull-right" href="#">Read more
    <span class="icon fa fa-arrow-circle-right">
  </span></a></p>
  </article>
  <article class="col-md-4 col-lg-12">                    <h4>Check it
out</h4>
  <p>Suspendisse et arcu felis, ac gravida turpis. Suspendisse
potenti.
    Ut porta rhoncus ligula, sed fringilla felis feugiat eget. In non
purus
    quis elit iaculis tincidunt. Donec at ultrices est.</p>
  <p><a class="btn btn-primary pull-right" href="#">Read more
    <span class="icon fa fa-arrow-circle-right">
  </span></a></p>
  </article>
  <article class="col-md-4 col-lg-12">                    <h4>Finally</
h4><p>Suspendisse et arcu felis, ac gravida turpis. Suspendisse
potenti.
    Ut porta rhoncus ligula, sed fringilla felis feugiat eget. In non
purus
    quis elit iaculis tincidunt. Donec at ultrices est.</p><p><a
class="btn btn-primary pull-right" href="#">Read more
    <span class="icon fa fa-arrow-circle-right"></span></a></p></
article></div>  </section>
```



Adjusting headings, font sizes, and buttons

Let's begin by adjusting our headings so that they consistently clear the buttons above them, which have been floated to the right. For this purpose, we'll use the file we previously created to manage the details of the page contents, `_page-contents.scss`.

Here's how to do it:

1. In `_page-contents.scss`, let's write a selector to select headings `h1` through `h4` when they're nested inside a Bootstrap column class. We'll use the CSS2 attribute selector and cover our bases by targeting any element whose classes include the col-string.

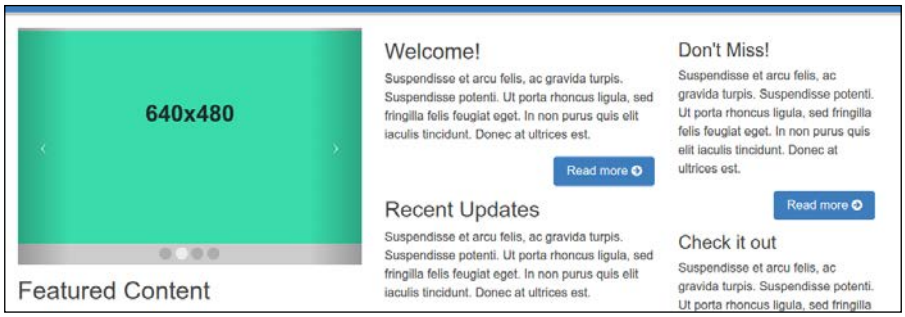


Later in this lesson, we will equip our footer with its own set of responsive columns. Thus, we need to make sure we nest these rules within the selector for the main element.

2. Within this context, we'll select all heading tags we might potentially use and set them to clear floated elements, with some added padding for separation:

```
[class*="col-"] {
  h1, h2, h3, h4 {
    clear: both;
    padding-top: $spacer-y;
  }
}
```

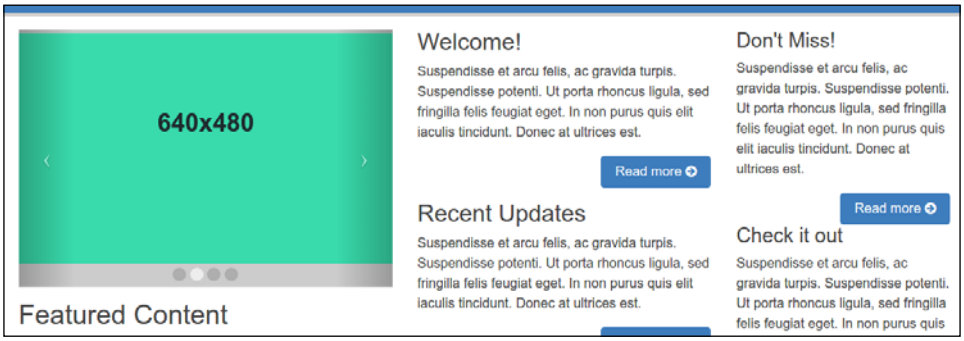
3. This gives the necessary separation between our headings and floated buttons. But it also creates unneeded padding at the top of the secondary and tertiary columns.
4. In the following screenshot, the lower arrows highlight the improvement accomplished now that our headings clear the floated buttons. Also, the ragged top edge of our columns, where padding causes a problem:



5. Let's remove the margin and padding from the uppermost heading in each column. We'll use the `:first-child` selector for these headings, nesting these lines within our heading selectors. We'll use the `&` combinator, which in this formulation allows us to select any first-child instance of these headings:

```
&:first-child {
  margin-top: 0;
  padding-top: 0;
}
```

6. This removes the extra margin and padding and evens up the top edge of the second:



7. But now there is a problem with the third column. The `:first-child` selector matches each `h4` element in the third column due to the nesting we added before. You can solve this issue by creating a new h4 selector in your Sass code, as follows:

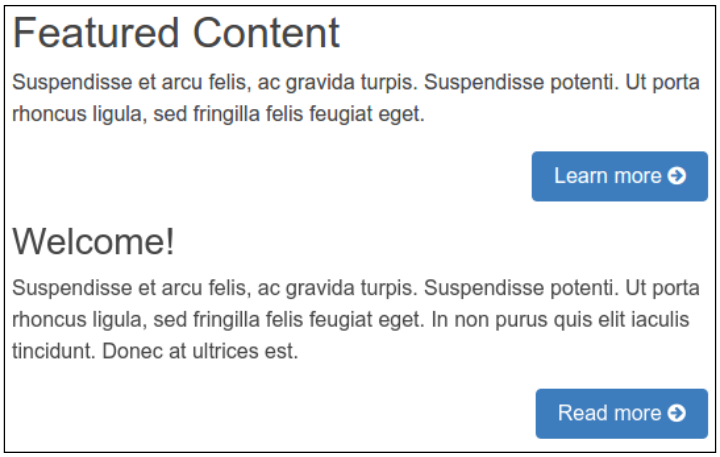
```
h4 {
  clear: both;
  padding-top: $spacer-y;
}
> article:first-child h4 {
```

```
    margin-top: 0;
    padding-top: 0;
  }
```

8. However, we only want to remove this top margin and padding in medium, large, or extra-large viewports, which accommodate multiple columns. Thus, we need to nest this rule within a media query corresponding with the breakpoint at which our layout expands from a narrow single-column layout to a wider multi-column layout.
9. Thus, we need to nest what we've just done within a media query for medium viewports and larger:

```
[class*="col-"] {
  h1, h2, h3, h4 {
    clear: both;
    padding-top: $spacer-y;
    @include media-breakpoint-up(md) {
      &:first-child {
        margin-top: 0;
        padding-top: 0;
      }
    }
  }
  h4 {
    clear: both;
    padding-top: $spacer-y;
  }
  @include media-breakpoint-up(md) {
    > article:first-child h4 {
      margin-top: 0;
      padding-top: 0;
    }
  }
}
```

With the preceding media query, we've retained the padding we need between elements in the single-column layout for narrow viewports, as seen in the following screenshot:



With this accomplished, we can move on to adjust buttons and font sizes to reflect the informational hierarchy of our content. Let's begin by enlarging the font size, button size, and color in our primary content area.

Enhancing the primary column

First, let's increase the font size of our primary column content:

- In Bootstrap's `_variables.scss` file, the `$font-size-large` variable is set to the following by default:

```
$font-size-lg: 1.25rem !default;
```

- Now, in the `scss/includes/_page-contents.scss` file, add these lines to use this font size for our primary content:

```
.content-primary {
  font-size: $font-size-lg;
}
```

Save these changes, compile the file, and refresh your browser. You should see the font size increase accordingly!

Now, let's adjust the color of our button to utilize the red `$brand-feature` color. We'll utilize the `$brand-feature` variable we set up in the local `scss/includes/_variables.scss` file.

```
$brand-feature:        #c60004;
```

We'll also utilize an excellent `button-variant()` mixin provided in the Bootstrap `mixins/_buttons.scss` file. You may want to take a moment to check it out. Open

the `mixins/_buttons.scss` in the Bootstrap's source code in the `bower_components` directory and search for `//` Button variants. You'll find a mixin that begins as follows:

```
@mixin button-variant($color, $background, $border) {
```

The mixin does the following:

- Specifies the button font, background, and border colors (in other words, the three parameters that the mixin accepts)
- Generates hover, focus, active, and disabled states for the button, adjusting font color, background color, and border

If you'd like to, you can see how Bootstrap uses this mixin in `bootstrap/_buttons.scss` under the `// Alternate` buttons comment. Here are the lines generating styles for the default and primary buttons:

```
//
// Alternate buttons
//
.btn-primary {
  @include button-variant($btn-primary-color, $btn-primary-bg, $btn-
primary-border);
}
.btn-secondary {
  @include button-variant($btn-secondary-color, $btn-secondary-bg,
$btn-secondary-border);
}
.btn-info {
  @include button-variant($btn-info-color, $btn-info-bg, $btn-info-
border);
}
```



You will find the variables beginning with `$btn-primary-` and `$btn-secondary-` in the `bower_components/bootstrap/scss/_variables.scss` file.

Following this pattern, we can generate our custom feature button in four simple steps:

1. First, we'll set up a new set of button variables. In the `_scss/includes/_variables.scss` file, under `// Buttons`, make a copy of the three `$btn-primary- variables`, and customize them, replacing `-primary-` with `-feature-` and using `$brand-feature` as the background color:

```
$btn-feature-color:              #fff;
$btn-feature-bg:                 $brand-feature;
$btn-feature-border:             darken($btn-feature-bg, 5%);
```

2. Next, we can make a file to keep our custom buttons. Create `scss/includes/_buttons.scss` and write a mixin based on the `.btn-primary` mixin from `bootstrap/_buttons.scss`, as follows:

```
.btn-feature {
  @include button-variant($btn-feature-color, $btn-feature-bg,
  $btn-feature-border);
```

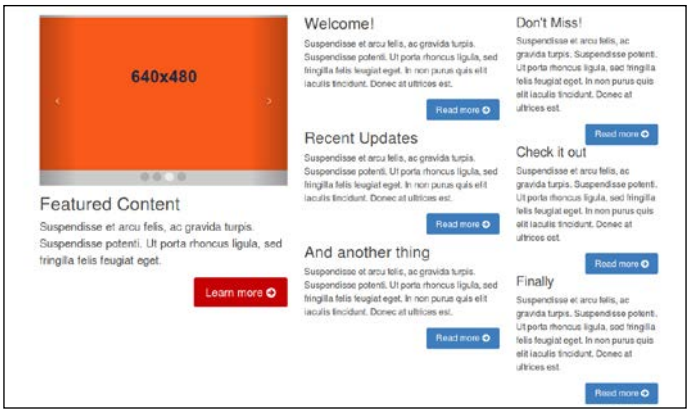
3. Save this file and add it to the import sequence in `scss/app.scss` as follows:

```
@import "includes/carousel"; @import "includes/buttons"; // added
```

4. Now, in the `html/pages/index.html` file, change the button class from `btn-primary` to `btn-feature`. While we're at it, we want to make the button large, so add the `btn-lg` class:

```
<a class="btn btn-feature btn-lg pull-right" href="#">Learn more
```

5. Save your work. Run the `bootstrap watch` command, and you should see the following result. The primary column to the left now has a larger font size and a large button with our `brand-feature` color:



Meanwhile, the font size and button colors of the secondary (center) column are exactly what we want. What needs to happen next is this: we need to de-emphasize the tertiary column content so that it takes its appropriate place in the informational hierarchy.

Adjusting the tertiary column

Our task for the tertiary content is fairly straightforward. We have to reduce the font size and de-emphasize the buttons. This can be accomplished as follows:

1. First, we'll adjust the font-size. In Bootstrap's `_variables.scss` file, the `$font-size-sm` variable is set as follows:

```
$font-size-sm:  .875rem !default;
```

2. Now we need only add these lines to the `_scss/includes/_page-contents.scss` file:

```
.content-tertiary {
  font-size: $font-size-sm;
}
```

3. If you have run the bootstrap watch command already, you should see the font size reduce, after saving your changes.
4. Next, in the `html/pages/index.html` file, we need to edit our button classes. We'll change them from btn-primary to `btn-secondary`, and we'll reduce their size using the `btn-sm` class:

```
<a class="btn btn-secondary btn-sm pull-right"
href="#">Read more ...
```

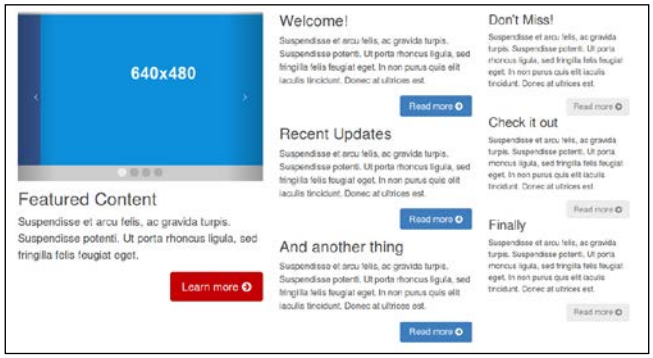
5. This will reduce the button size and turn the button's background white.

6. Let's adjust the background to a light gray and adjust the font color and border as well. In the `_variables.scss` file, adjust the values for the three `$btn-secondary-` variables as follows:

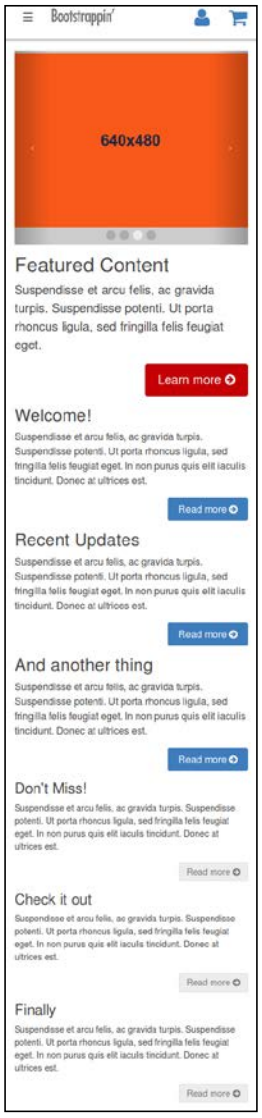
```
$btn-secondary-color:              $gray;
$btn-secondary-bg:                 $gray-lightest;
$btn-secondary-border:             darken($btn-secondary-
bg, 5%);
```

Save the changes, compile the file, and refresh your browser.

We now have a clear visual hierarchy, from the primary content (on the left), to the secondary (center), and tertiary (right):



Now, take a moment to notice that our adjustments work reasonably well in the narrow single-column layout as well:



In narrow viewports, our three columns stretch out vertically, one after the other, with primary content first, followed by secondary and tertiary.

All that remains is some fine-tuning to make our content even more user friendly across devices and viewports.

Fine touches for multiple viewports

It's always good to give our content—and our viewers' eyes—some room to breathe. Visual indicators of section boundaries are good as well. Let's fold these in:

First, we'll add padding above and below our content. Add a bit of top padding to the `main` element itself. This padding will serve us well in all viewports, so we won't need a media query:

```
main {
  padding-top: $spacer-y;
  padding-bottom: $spacer-y * 2;
}
```

That's it. Our main content layout is ready. Now for the complex footer area.

Laying out a complex footer

In the following steps, we'll create a complex footer built to manage multiple goals, including three lists of links to key sections of our website, a bit of About Us text, social icons, and our logo.

Setting up the markup

We will start by creating the footer markup. We want this footer to be as functional and useful for the user as possible. We'll build the markup as follows:

1. Start with the footer of the Portfolio project. You'll find the HTML Markup code of the footer in the `html/includes/footer.html` file.
2. Move HTML code for the logo and add it directly under the social links and create a new include for the additional footer content as follows:

```
<footer role="contentinfo">
{{> footercolumns}}

<div class="container social-logo">
<ul class="social">
  <li class="social-item"><a href="#" class="social-link"
  ><i class="fa fa-twitter"></i></a></li>
  <li class="social-item"><a href="#" class="social-link"
  ><i class="fa fa-facebook"></i></a></li>
  <li class="social-item"><a href="#" class="social-link"
  ><i class="fa fa-linkedin"></i></a></li>
  <li class="social-item"><a href="#" class="social-link"
```

```
            ><i class="fa fa-google-plus"></i></a></li>
            <li class="social-item"><a href="#" class="social-link"
            ><i class="fa fa-github-alt"></i></a></li>
          </ul>

          <p><a href="{{root}}index.html"><img src="{{root}}images/
logo.png"
           width="80" alt="Bootstrapping"></a></p>
          </div>
          </footer>
```

3. Now create a new HTML partial called `html/includes/footercolumns.html`. You can pass the additional footer content into this file.
4. Before pasting the content, let's prepare to utilize the Bootstrap grid system. To do this, we'll wrap the area within `div class="row"`, as follows:

```
<div class="container">
  <div class="row">
    ...
  </div><!-- /.row -->
</div><!-- /.container -->
```

5. Now, paste the new content in place.
6. Next, we'll wrap each of the three lists of links, along with their headings, in a `col-lg-2` class `div`. This way, each list will take one-sixth of the available width in medium and larger viewports. Together, these three lists will take half the available viewport width.
7. Now, to complete our row, wrap the About Us heading and its paragraph in `col-lg-6` class div so that it takes up the remaining half of the available width:

```
<div class="about col-lg-6">
  <h3>About Us</h3>
```



Be sure to add the necessary closing tags for each new div element.

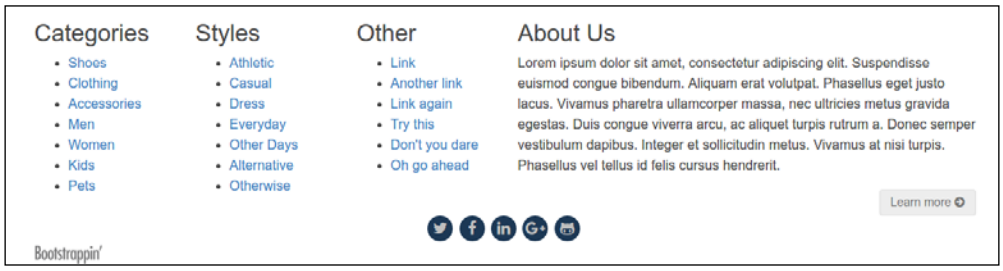
8. Save, run the bootstrap watch or `gulp` command, and check your results.
9. After performing the preceding steps, you should end up with the following HTML code:

```
<div class="container">
  <div class="row">
    <div class=col-lg-2">
```

```
            <h3>Categories</h3>
            <ul>
              <li><a href="#">Shoes</a></li>
              <li><a href="#">Clothing</a></li>
              <li><a href="#">Accessories</a></li>
              <li><a href="#">Men</a></li>
              <li><a href="#">Women</a></li>
              <li><a href="#">Kids</a></li>
              <li><a href="#">Pets</a></li>
            </ul>
          </div>
          <div class="col-lg-2">
            <h3>Styles</h3>
            <ul>
              <li><a href="#">Athletic</a>  </li>
              <li><a href="#">Casual</a></li>
              <li><a href="#">Dress</a></li>
              <li><a href="#">Everyday</a></li>
              <li><a href="#">Other Days</a></li>
              <li><a href="#">Alternative</a></li>
              <li><a href="#">Otherwise</a></li>
            </ul>
          </div>
          <div class="col-lg-2">
            <h3>Other</h3>
            <ul>
              <li><a href="#">Link</a></li>
              <li><a href="#">Another link</a></li>
              <li><a href="#">Link again</a></li>
              <li><a href="#">Try this</a></li>
              <li><a href="#">Don't you dare</a></li>
              <li><a href="#">Oh go ahead</a></li>
            </ul>
          </div>

          <!-- Add the extra clearfix for only the required viewport -->
          <div class="clearfix hidden-sm-down hidden-lg-up"></div>

          <div class="about col-lg-6">
            <h3>About Us</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
            Suspendisse euismod congue bibendum. Aliquam erat volutpat.
            Phasellus eget justo lacus. Vivamus pharetra ullamcorper
massa, nec
```

```
      ultricies metus gravida egestas. Duis congue viverra arcu,
ac aliquet
      turpis rutrum a. Donec semper vestibulum dapibus.
      Integer et sollicitudin
      metus. Vivamus at nisi turpis. Phasellus vel tellus id felis
cursus
      hendrerit.</p>
      <p><a class="btn btn-secondary btn-sm pull-right"
href="#">Learn more
      <span class="fa fa-arrow-circle-right"></span></a></p>
    </div>
  </div><!-- /.row -->
</div><!-- /.container -->
```

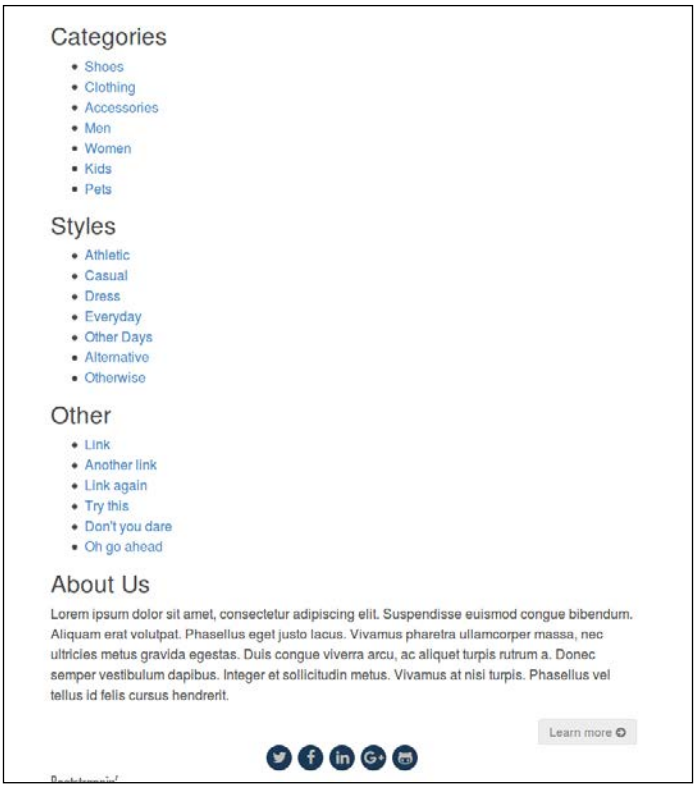
Due to the Bootstrap grid classes we've added in our HTML in a viewport of 980 pixels and larger, our columns should organize themselves as follows:



This is the layout we want in large and extra-large viewports. Extra-small screen sizes are served just fine by the single-column layout. However, for tablet-width screen sizes that fall within the range of 768 to 980 pixels, our layout can benefit from some adjustments. Let's address that.

Adjusting for tablet-width viewports

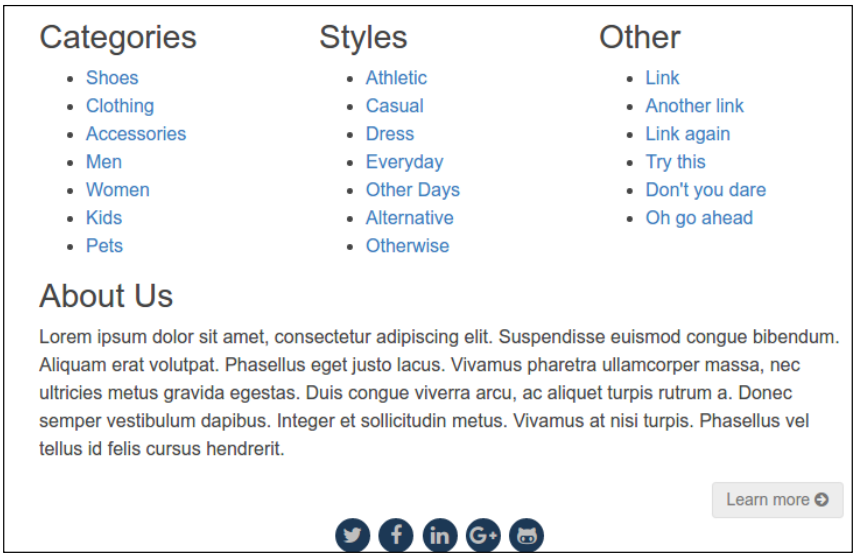
Test the layout in a viewport that falls between 768 and 980 pixels. Bootstrap refers to this as the medium breakpoint and `col-md- grid` classes. At this width, the single-column layout leaves unnecessary white space. Here is what you'll see:



We can improve this layout by allowing our three lists of links to float next to each other. Using the Bootstrap `col-md-*` column classes for Bootstrap's grid, let's set the three lists of links to be one-third width, or `col-sm-4`, and the About Us column to be full width, or `col-sm-12`:

```
<div class="col-md-4 col-lg-2">
...
<div class="col-md-4 col-lg-2">
...
<div class="col-md-4 col-lg-2">
...
<div class="about col-xs-12 col-lg-6">
```

Save this and try it out in the medium viewport range. You will see the following result:



Much improved! But we're not quite finished. Try clicking on the links in the upper three columns. Chances are that you won't be able to. Inspect the element and you'll find that the fourth `div` element contains the code for the About Us column. This code does not clear the floated columns above it. Though the About Us heading and its paragraph will appear below the three floating columns, the `div` element itself will overlap them.

Adding a targeted responsive clearfix

In a standard Bootstrap layout situation, we would use a `div` element with the row class to clear the floating columns. Here, we need a different solution, as we want this block of content to clear floats only within this specific breakpoint.

To accomplish this, we could write custom styles in our Sass files. But we can also use a Bootstrap responsive utility class to provide a targeted clearfix directly in the markup. Since we've already specified grid classes in our markup, let's use the second option in this context.

You can find the approach we'll use mentioned in Bootstrap's documentation at `http://getbootstrap.com/layout/grid/#example-responsive-column-resets`. Following that method, we'll create a div element with the `clearfix` class, and add a Bootstrap responsive utility class to make it visible only on small screens. We'll place this new div element immediately prior to the About Us column:

```
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-sm-down hidden-lg-up"></div>
```

```
<div class="about col-xs-12 col-lg-6">
```

The `clearfix` class will force this element to clear the floats above it. The `hidden-sm-down` and `hidden-lg-up` classes will allow this `div` to display only within our targeted breakpoint. At other breakpoints, it will be as if this div does not exist.

Save this, and you should find that the About Us column now clears the floats above it and that the links are clickable.

Task complete. Now for a few finishing touches.

Refining the details

We have a few final touches we want to implement as we finish our footer. These include the following:

- Refining the presentation of our three lists of links
- Adjusting margins and padding
- Reversing the color scheme to match our navbar colors

To accomplish these refinements, we'll write some custom styles. Let's tackle this in cascading fashion, starting with general rules for the footer and moving to the specific rules:

1. Open `_footer.scss`, the file for custom footer styles, in your editor.
2. Here, you'll find some initial rules that includes some initial padding for the footer, as well as styles for the social icons and the footer version of the logo.
3. Now to add the refinements we need for our new complex footer. Let's start by reducing the footer font size and inverting the color scheme to correspond with the inverted navbar—a blue background with light text. I'll begin with those colors and then darken them slightly. To do this, I'll make use of appropriate variables from Bootstrap's `_variables.scss` and the local `scss/includes/_variables.scss` files, including `$font-size-sm`, `$navbar-md-bg`, and `$navbar-md-color`:

```
footer[role="contentinfo"] {
  padding-top: 24px;
  padding-bottom: 36px;

  font-size: $font-size-sm;
  background-color: darken($navbar-md-bg, 18%);
  color: darken($navbar-md-color, 18%);
}
```



In this and all that follows, we need to nest our new rules within `footer[role="contentinfo"]`.

4. Next, we need to adjust our links and buttons to fit the new color scheme. Still nesting rules within `footer[role="contentinfo"]`, I've done this as follows:

```
footer[role="contentinfo"] {
  a {
      color: $navbar-md-color;
      @include hover-focus-active {
        color: $navbar-md-hover-color;
      }
  }
  .btn-secondary {
  color: darken($navbar-md-bg, 18%) !important;
  }
}
```

5. Now to address the four `h3` headings. I'll adjust font size, trim the bottom margin, and convert the text to uppercase:

```
footer[role="contentinfo"] {
  h3 {
    font-size: 120%;
    margin-top: $spacer-y;
    margin-bottom: 4px;
    text-transform: uppercase;
  }
}
```

6. Having done this, we can next remove bullets from our list of links, and adjust their padding and margin:

```
ul {
  list-style: none;
  padding: 0;
  margin: 0;
}
```

7. And we can center the logo and social icons:

```
footer[role="contentinfo"] {
  .social-logo {
    text-align: center;
  }
}
```

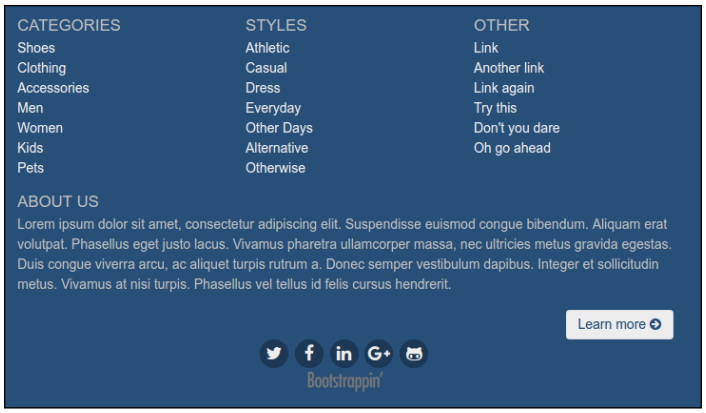
8. Finally, let's adjust our social icons. We'll add a bit of top padding and then adjust their colors to work better with the new color scheme. Since these are Font Awesome icons, we can do this simply by adjusting the color and background-color values, as follows:

```
        .social-link {
          display: inline-block;
          font-size: 18px;
          line-height: 30px;
          @include square(30px); // see includes/mixins/_size.scss
          border-radius: 36px;
          background-color: darken($navbar-md-bg, 27%);
          color: darken($navbar-md-color, 18%);
          margin: 0 3px 3px 0;
          @include hover-focus { // bootstrap/scss/mixins/_hover.
scss
          text-decoration: none;
            background-color: darken($navbar-md-bg, 32%);
            color: $navbar-md-hover-color;
          }
        }
```

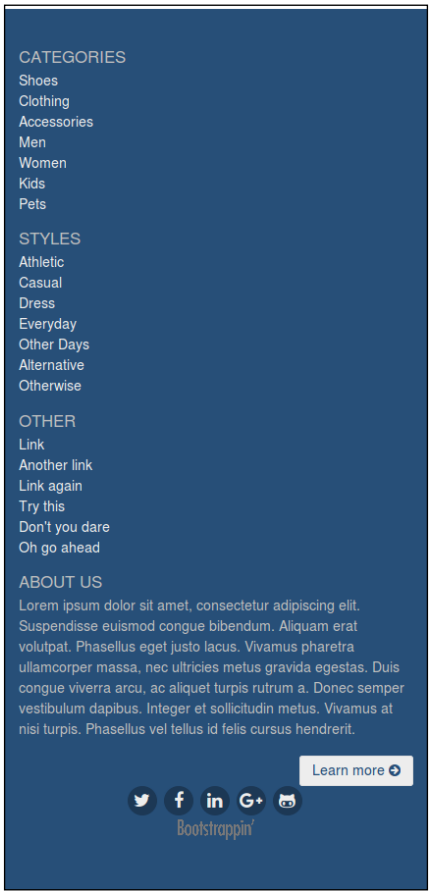
9. That's it. Save, run the `bootstrap watch` command, and enjoy! Here is our result in the large and extra-large viewports:



Here is the result for the medium viewport:



And this is for extra-small and small viewports:



Not bad! We have built a footer capable of managing a complex array of content across the full spectrum of extra-small, small, medium, large, and extra-large viewports.

Summary

This project has enabled us to level up our Bootstrapping skills in a number of ways. We've designed a responsive layout for the main content of our page, providing an appropriate visual hierarchy for three tiers of information. At the top of our page we've built a complex responsive navbar, so that it appears below the logo and banner area in medium, large, and extra-large viewports and yet collapses into a mobile-friendly navbar on smaller screens. The footer part of the project effectively manages multiple blocks of links and text across viewports.

Congratulations! In the next lesson, we'll build on these skills by designing a products page suitable for an e-commerce section for this website.

Assessments

1. In which of the three categories is the website broken down into?
    i. Banner/ masthead
    ii. Footer
    iii. Header
    iv. Both i & ii
2. Which of the following code is correct to `right align` the `last-child`?
    i.
    ```
    .nav-item:last-child {
    float:right;
    }
    ```
    ii.
    ```
    .nav-item:last-child {
    align:right;
    }
    ```
    iii.
    ```
    .nav-item:child-last {
    float:right;
    }
    ```
    iv. None of the above

3. What is the main purpose of adding utility navigation?
    i. To display icons at the far right of the collapsed navbar
    ii. To allow users to log in or register and to view their carts
    iii. Both i & ii
    iv. None of the above

4. What is the output of the following snippet?

```
<a href="#" class="p-1-x">
```

    i. Sets a padding of `$spacer-x` on the right-hand side of the `<a>` element
    ii. Sets a padding of `$spacer-x` on the left hand side of the `<a>` element
    iii. Sets a padding of `$spacer-x` on the left- and right-hand sides of the `<a>` element
    iv. None of the above

5. After completion of footer, which of the following final touches need to be implemented?
    i. Refining the presentation of our three lists of links
    ii. Adjusting margins and padding
    iii. Reversing the color scheme to match our navbar colors
    iv. All of the above

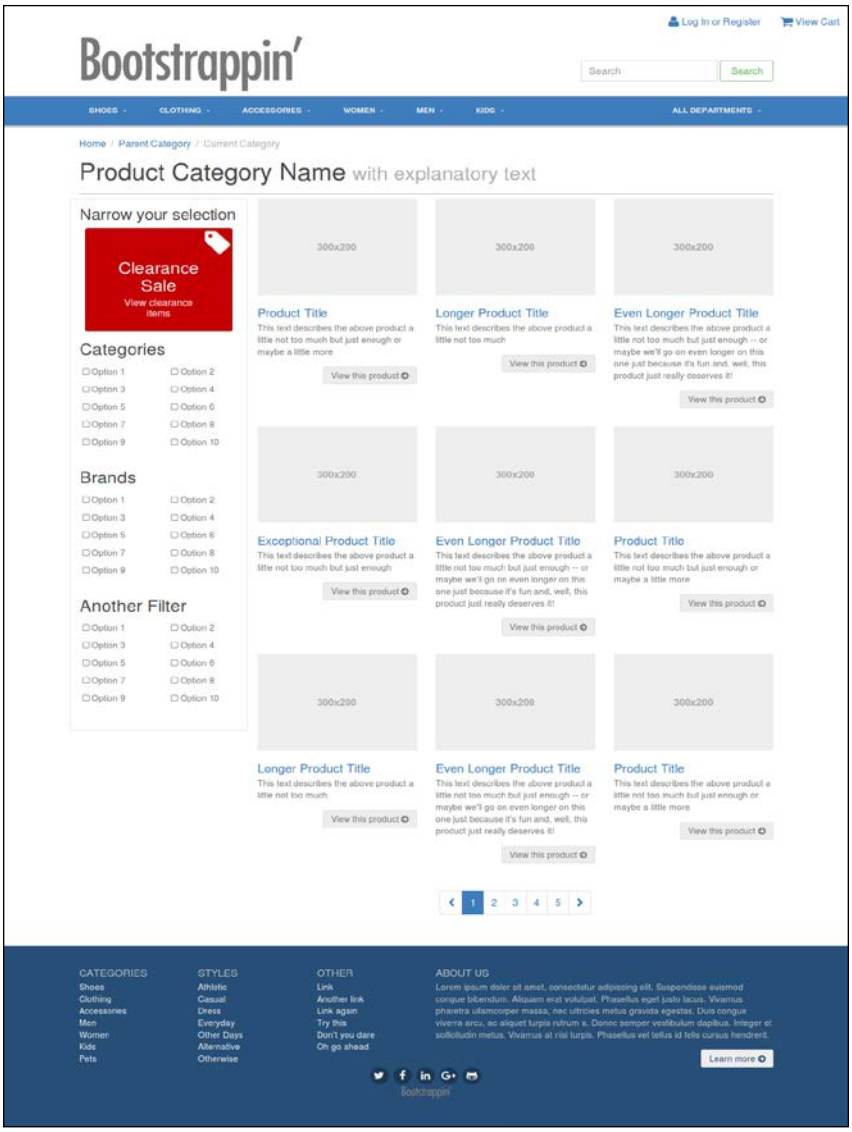
10
Bootstrapping E-Commerce

Having built our business home page, it's time to design our online store.

We'll build on the design from the previous lesson, adding a new page with the following elements:

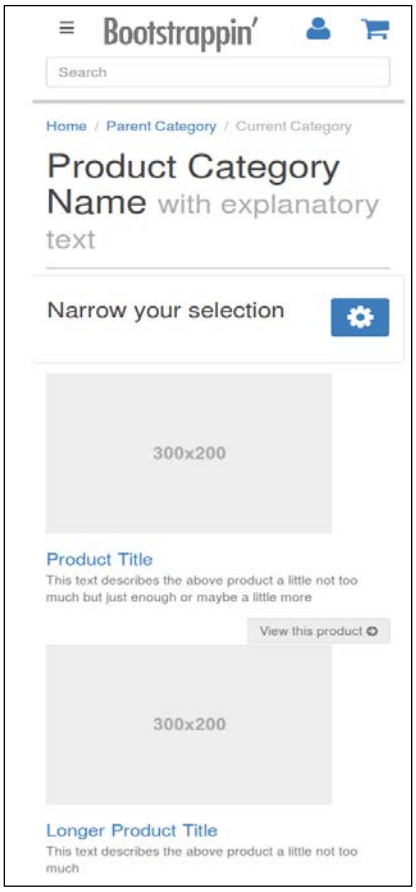
- A grid of product thumbnails, titles, and descriptions
- A left-hand sidebar with options to filter our products by category, brand, and so on
- Breadcrumbs and pagination to ease navigation through our inventory
- Take a few moments to visit websites such as Zappos (http://www.zappos.com) and Amazon (http://www.amazon.com). Search or browse for products and you will see product grids with features similar to what we will be creating in this lesson.

When complete, we want our products page to look like the following screenshot on medium, large, and extra-large screens:



Layout for medium large and extra-large screens

On extra-small screens, we want our products page to adjust to the following single column layout:

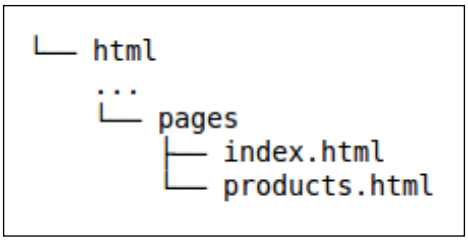


Bootstrap gives us a big head start in accomplishing this design-after which we can use the power of Sass to refine things to completion.

Surveying the markup for our products page

You'll find this lesson's files prepared and ready in the folder `Lesson 10/start`. This project builds directly on the completed design from *Lesson 9*, *Bootstrapping Business*. If anything in these files seems strange, you may want to review *Lesson 9*, *Bootstrapping Business*, before proceeding.

Run the `bower install` and `npm install` commands in your console before going on with the next steps! For this lesson, there is one new file in the `html/pages` directory, `products.html`.



Open `products.html` in your editor to view the markup. Let's survey its contents.

Inside the `main role="main"` element is where we'll find what's new. Here, you'll find the following elements in the same order as they appear:

- Breadcrumb links marked up as an ordered list
- A page title within a `h1` heading
- A series of options for filtering products
- Nine products with thumbnails, titles, descriptions, and a button
- An unordered list of pagination links just below the products and before

You can run the `bootstrap watch` command and point your browser to `http://localhost:8080/products.html` to view the file in your browser. You'll see that much remains to be done. Breadcrumbs do not yet look like breadcrumbs, the filtering options look like a long series of bulleted lists, the layout of our product items is uneven (and in places broken), and so on.

Don't let these current imperfections worry you. These are the things that we'll be addressing in the following steps. Here is what's coming:

- We will apply Bootstrap's built-in styles to the breadcrumbs, page title, and pagination, and then customize them further
- We will improve the layout of the nine product items, innovating the Bootstrap grid system to maintain a visually well-organized grid across breakpoints
- We will style the filtering options by enhancing the layout and then using the Font Awesome icons to provide checkboxes

Run the `bootstrap watch` or `gulp` command in your console and point your browser to `http://localhost:8080/products.html`. Your browser will automatically reload now after saving the Sass or HTML templates.

Now that we have a plan, let's get started!

Styling the breadcrumbs, page title, and pagination

In the following steps, we'll apply Bootstrap styles to our breadcrumbs, page title, and pagination, and then customize them to fit our design:

1. Open `products.html` in your editor.
2. Find the ordered list just above the `h1` page title, add the `"breadcrumb"` class to the `ol` tag, and then add the `"active"` class to the last list item, as follows:

```
<ol class="breadcrumb">
  <li class="breadcrumb-item"><a href="/">Home</a></li>
  <li class="breadcrumb-item"><a href="#">Parent Category</
a></li>
  <li class="breadcrumb-item active">Current Category</li>
</ol>
```

   These classes correspond with Bootstrap breadcrumb styles, which you will find documented at `http://v4-alpha.getbootstrap.com/components/breadcrumb/`.

   Save and refresh your browser. You should see the result shown in the following screenshot:



3. To customize the breadcrumbs for this design, let's remove the light gray background and the extra padding.
4. Let's set the `padding` to `0` and remove the `background-color` entirely
5. Create a new Sass partial in the `scss/_includes` directory called `_breadcrumb.scss` and add the following SCSS to it:

```
.breadcrumb {
padding: 0;
background-color: initial;
}
```

6. Do not forget to import the new `_breadcrumb.scss` partial file into your `app.scss` file as follows:

```
// Components
@import "includes/breadcrumb";
```

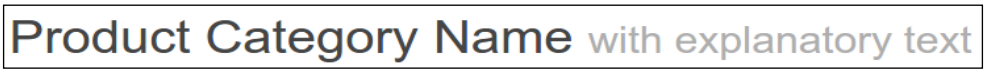
7. Now for the page title. The page title works by nesting the top-level page heading within a `div` tag of the `page-header` class. The SCSS code for the `page-header` class can be edited in a new `scss/includes/_page-header.scss` partial and may look as shown here:

```
// Page header
// -------------------------
.page-header {
  padding-bottom: ($spacer / 2);
  margin: $spacer 0 ($spacer / 2);
  border-bottom: 1px solid $page-header-border-color;
}
```

8. Notice that you'll have to declare the `$page-header-border-color;` variable in the `scss/includes/_varaibels.scss` file and of course also have to import the `scss/includes/_page-header.scss` partial in the main `app.scss` file.
9. Let's adjust our markup accordingly. For the title, an `h1` tag with Bootstrap's `display-*` classes will be used. Let's also add some text within a `small` tag having Bootstrap's `text-muted` class to take advantage of the Bootstrap style for adding the explanatory notes to our headings:

```
    <div class="page-header">
      <h1 class="display-5">Product Category Name <small
class="text-muted">with explanatory text</small></h1>
    </div>
```

That will produce the following result:



You can read more about Bootstrap's typography and heading classes at `v4-alpha.getbootstrap.com/content/typography/#headings`.

10. Finally, the pagination. Our markup for this is found just a few lines above the closing main tag (`</main>`). Above that closing tag, you'll see commented closing div tags for the `.container`, `.row`, and `.products-grid`:

```
        </div><!-- /.products-grid -->
      </div><!-- /.row -->
    </div><!-- /.container -->
  </main>
```

Bootstrap's documentation for pagination styles is found at `v4-alpha.getbootstrap.com/components/pagination`.

11. To apply these styles here, we only need to add `class="pagination"` to the `ul` tag that you will find a few lines above the closing `.products-grid` tag:

```
  <ul class="pagination">
    <li class="page-item">
      <a class="page-link" href="#" aria-label="Previous">
        <span aria-hidden="true" class="fa fa-chevron-left"></
span>
        <span class="sr-only">Previous</span>
      </a>
    </li>
    <li class="page-item active">
      <a class="page-link" href="#">1 <span class="sr-
only">(current)</span></a>
    </li>
    <li class="page-item"><a class="page-link" href="#">2</a></li>
    <li class="page-item"><a class="page-link" href="#">3</a></li>
    <li class="page-item"><a class="page-link" href="#">4</a></li>
    <li class="page-item"><a class="page-link" href="#">5</a></li>
    <li class="page-item">
      <a class="page-link" href="#" aria-label="Next">
        <span aria-hidden="true" class="fa fa-chevron-right"></
span>
        <span class="sr-only">Next</span>
      </a>
    </li>
  </ul>
```

The markup for the navigation links may contain different classes to set the state of a link. The active CSS class makes clear that the link has an active state, whilst the disabled CSS class enables you to give some links a disabled state.

The HTML code for a disabled item may look like the following:

```
<li class="page-item disabled">
   <a class="page-link" href="#" tabindex="-1" aria-
label="Previous">
     <span aria-hidden="true">&laquo;</span>
     <span class="sr-only">Previous</span>
   </a>
 </li>
```

Neither disabled nor active items are clickable. You can add the add `pagination-lg` or `pagination-sm` CSS classes for larger or smaller pagination as follows:

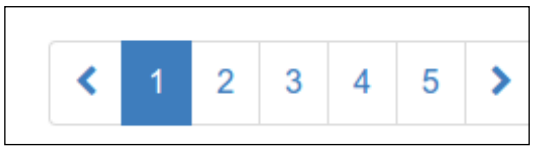
```
<ul class="pagination pagination-lg">
```

```
  ...
</ul>
```

Also notice that Bootstrap takes accessibility into account. The navigation contains various `aria-*` attributes. **Accessible Rich Internet Applications** (**ARIA**) is a set of special accessibility attributes that can be added to any markup, but is especially suited to HTML. You can read more about ARIA in HTML at the website of the **World Wide Web Consortium** (**W3C**), see `https://www.w3.org/TR/html-aria/`. Elements with Bootstrap's `sr-only` class provided additional information for only screen readers.



For the `Next` and `Prev` items, I've already provided the `span` tags for the Font Awesome icons, `fa-chevron-left` and `-right`. This gives us the result shown in the following screenshot:



12. Let's center align the pagination below our grid. First, wrap it in a parent `div` tag. We'll place the `row` class on this to ensure it clears the content above it, and then we'll add an appropriately named Bootstrap class `text-xs-center`. The `xs` in the naming means for the extra small grid and up:

```
<nav class="text-xs-center">
<ul class="pagination">
<li> ...
</ul>
</nav>
```

Adjusting the products grid

Before you start you should notice that the product images, provided by the `holder.js` image placeholders as described in *Lesson 9*, *Bootstrapping Business*, are not responsive. Let's make all images responsive by default by adding the following lines of SCSS code to our `app.scss` file:

```
// make images responsive by default
img {
  @extend .img-fluid;
}
```

The process of making images responsive by default. It influence all images. The logo in the also becomes responsive now and ignores the width we've set in the `scss/includes/_header.scss` file before. You can solve that by putting the SCSS code for the responsive image before the import of the `_header.scss` file in the `app.scss` file.

You should also inspect the footer logo after making the images responsive by default. You'll find that the logo does not center anymore. The `img-fluid` class changes your images into block level element. Block level elements cannot be centered by the text-align: center; declaration. You can use the following CSS code in the file to center the logo again:

```
.social-logo {
  img {
    margin 0 auto;
  }
}
```

Now let's make our products grid look as it should. Before we start, we move the product grid to its own HTML template file. Create a new HTML partial called `html/includes/products-grid.html`.

In the `html/includes/products.html` file, use the following code to include the product grid:

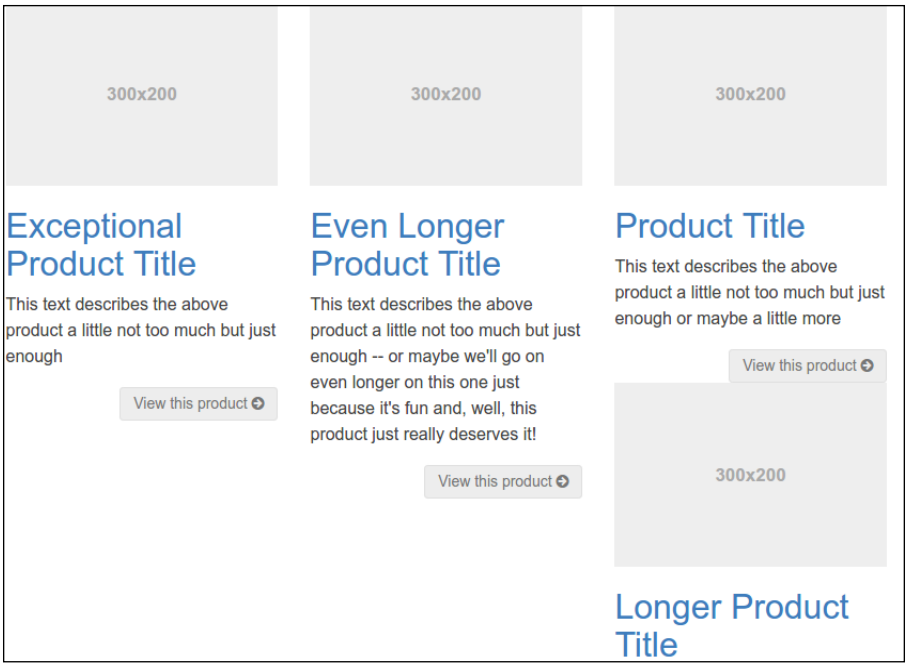
```
<div class="products-grid col-md-9">
  {{> products-grid }}
</div>
```

If you inspect the markup for our product items, you'll see that each item has been given a class of `col-sm-4`:

```
<div class="product-item col-sm-4">
```

While this constrains the width of each of our product items and the grid automatically wraps the columns into the row, it has failed to produce an effective grid.

The primary problem here is that our items have varying heights. Thus, when trying to float left, as Bootstrap grid components do, these items bump into one another. This results in a broken, uneven layout, as shown in the following screenshot:



Currently, in a medium, large, or extra-large viewport, product items 4 to 7 refuse to float neatly due to their uneven heights. The problem of your columns don't clear quite right as one is taller than the other can be solved by using an additional div element with a combination of the `clearfix` class and the responsive utility classes. You can read more about the clearfix and the responsive utility classes at the following links:

- `http://getbootstrap.com/docs/4.0/utilities/clearfix/`
- `https://v4-alpha.getbootstrap.com/layout/responsive-utilities/`

Now, let's fix our layout problem. We should `clearfix` the layout after each third item. You can do this by adding the following HTML snippet after each third item in the `html/pages/products.html` file:

```
<!-- Add the extra clearfix for only the required viewport -->

<div class="clearfix hidden-sm-down"></div>
```

Then we want our grid to reduce to two products per row for the medium screens, while large and extra-large viewports will have three items per row. To accomplish this, we need to find and replace the classes in each of our product items so that they are as follows:

```
<div class="product-item col-md-6 col-lg-4">
```

These classes will set each product item to half width within extra-small and small viewports, and then transition to one-third width for medium and large viewports.

The preceding change also means that we have to replace and extend our responsive column resets as follows:

1. After each third item the HTML code should look as follows:

```
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-md-down"></div>
```

2. The `hidden-sm-down` class in the preceding code has been replaced with the class `hidden-sm-down`. And then add the following lines of HTML after each second item:

```
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-sm-down hidden-lg-up"></div>
```

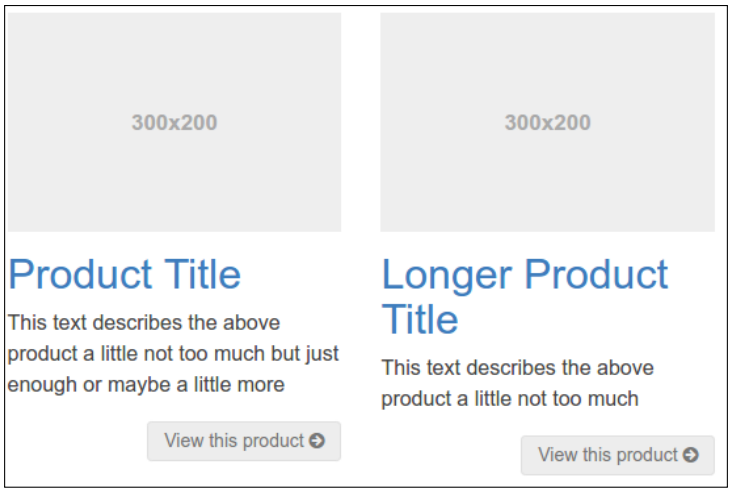
3. After the sixth item you will get the following HTML code now:

```
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-md-down"></div>
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-sm-down hidden-lg-up"></div>
```

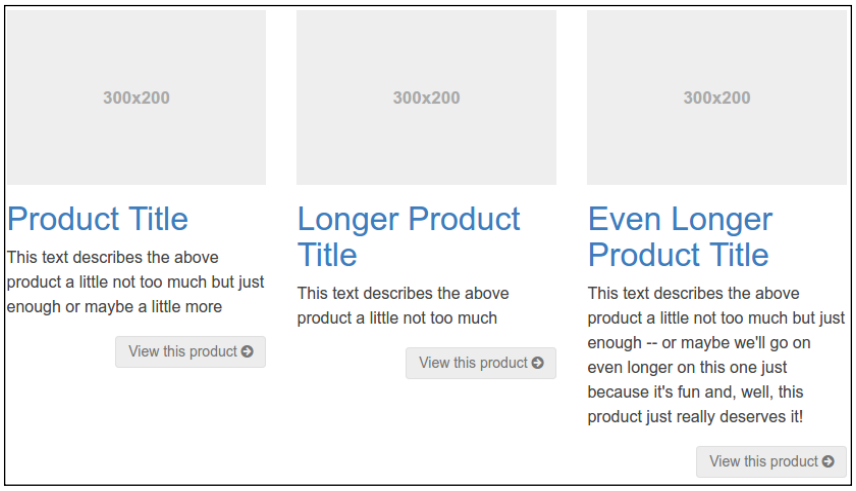
4. The HTML code after the sixth item above should be replaced with this:

```
<!-- Add the extra clearfix for only the required viewport -->
<div class="clearfix hidden-sm-down">
```

5. Product items will now be laid out in two columns on a medium viewport:



6. Then, our grid will transform to a three-column layout in large and extra-large viewports:



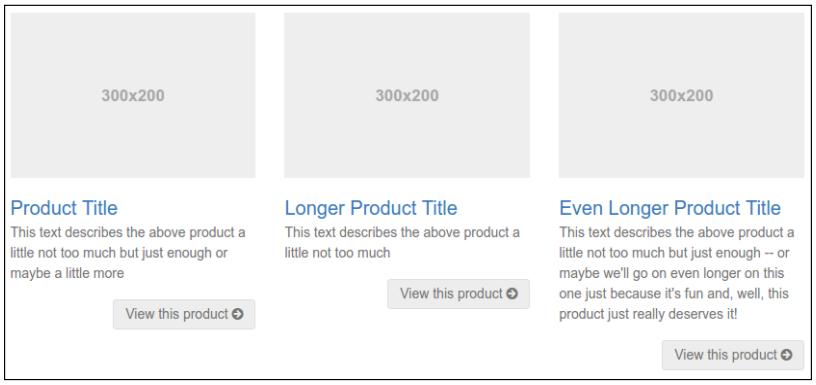
7. Let's adjust the styles of our grid items to enhance their visual presentation. Having done that, we can fix this layout problem.
8. As we'll be writing custom styles, create and have the `sccs/includes/_products-grid.scss` file open in your editor and import it into the main `app.scss` file.
9. Let's write styles to adjust image width, font size, padding, and margins, as shown in the following lines of code:

```
.product-item {
  padding-bottom: ($spacer-y * 2);

  h2 {
    font-size: $font-size-lg;
    line-height: $line-height-lg;
    padding: 0;
    margin-top: ($spacer-y / 7);
    margin-bottom: ($spacer-y / 8);
  }
  p {
    font-size: $font-size-sm;
    line-height: $line-height-sm;
    color: $gray;
  }
}
```

10. These styles will accomplish the following:

- Add bottom padding to each product item.
- Reduce the `h2` heading font size to the size of our `$font-size-lg`.
- Reduce the `p` font size to our `$font-size-sm` value.
- Reduce `h2` padding by adding `!important` to override any conflicting rules that we've written to apply in the standard pages.
- Set the `p` font color to `$gray`.

Save these new styles, and run the `bootstrap watch` or `s` command. Though the layout will still be broken in places, you should see significant improvement in the styling of the product items, as shown in the following screenshot:



It's a beautiful thing to behold.

Don't forget the Card module

In the preceding section you used Bootstrap Grid to build to the product grid. You can also use Bootstrap's new card module to build the product grid. Cards include header, footers, top and bottom image caps.

First, create a new HTML partial called `html/includes/product-grid-cards.html` to rebuild the product grid using the Cards module. The HTML code for each card should look as follows:

```
<div class="card">
  <a href="#"><img data-src="holder.js/300x200?auto=yes"
  alt="sample product" /></a>
  <div class="card-block">
    <h4 class="card-title"><a href="#">Product Title</a></h4>
```

```
          <p class="card-text">This text describes the above product a
          little not too much but just enough or maybe a little more</p>
          <a class="btn btn-secondary btn-sm pull-sm-right"
href="#">View
          this product <i class="fa fa-arrow-circle-right"></i></a>
        </div>
      </div>
```

Bootstrap enables you to organize your cards in Groups or Decks. In this example, you will use decks. Decks contain a set of equal width and height cards that aren't attached to one another. The HTML structure of a Deck of Cards will look as follows:

```
<div class="card-deck-wrapper">
  <div class="card-deck">
    <div class="card product-item">
    ---
    </div>
    <div class="card product-item">
    ---
```

You'll have to wrap each block of three cards into its own `card-deck` wrapper.

The Card and Deck groups got a single breakpoint at between the extra small and small grid at 576 pixels. Below the breakpoint, the cards will stack. For the small grid between 576 and 768 pixels there are three cards in a row too. This cards are very small, so you'll have to reduce the size of the button for the small grid by using the following SCSS code:

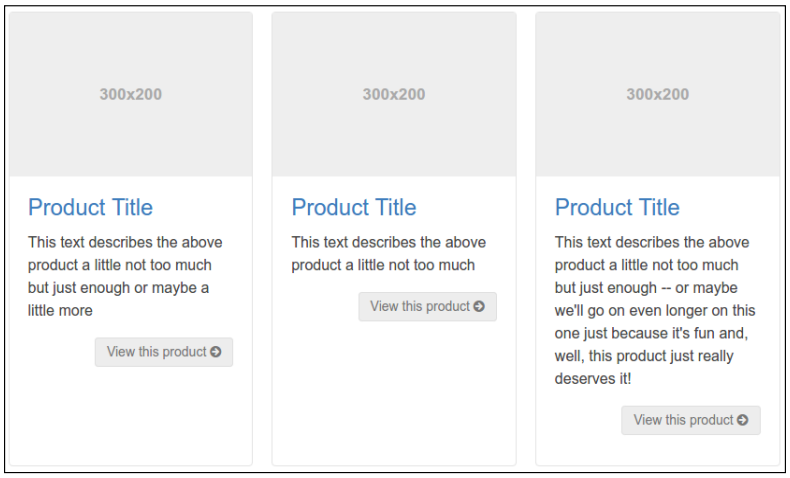
```
.product-item {
  .btn-sm {
    @include media-breakpoint-only(sm) {
       font-size: $font-size-sm * 0.8;
    }
  }
}
```

In the preceding code, we've reduced the size of the button in the small viewport. Now let's add some space between the card for the larger viewports.

In `scss/includes/_product-grid.scss` you can add the following SCSS code to create some space between the cards:

```
@include media-breakpoint-up(sm) {
  .card-deck {
      padding-bottom: ($spacer-y * 2);
  }
}
```

With the Card Deck, your product grid may look like the following screenshot:



Cards with the CSS3 Flexbox layout module

Bootstrap has option flexbox support built in. You can enable flexbox support by setting the `$enable-flex` Sass variable to `true`.

Create a new HTML partial called `html/includes/product-grid-cards-flexbox.html` to test the flexbox layout. Don't forget to replace the include statement in the `html/product.html` file as follows:

```
{{> products-grid-cards-flexbox}}
```

In the `scss/includes/_variables.scss` file, add the following line of SCSS code:

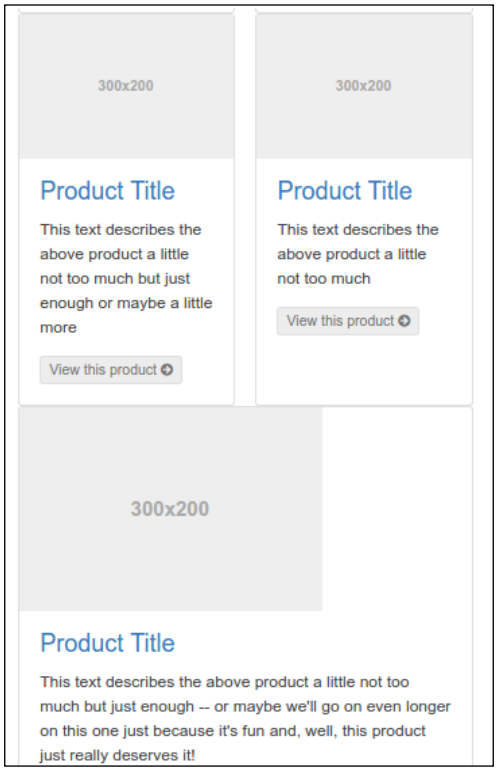
```
// Options
//
// Quickly modify global styling by enabling or disabling optional
features.
$enable-flex:           true;
```

The `html/includes/product-grid-cards-flexbox.html` file may contain HTML code similar to the HTML code used in the `html/includes/product-grid-cards-flexbox.html` file. The card-deck-wrapper wrapper is not required when the flexbox support is enabled. You can wrap all Cards into a single card-deck wrapper. The breakpoint is still set at 576 pixels. For viewports wider than the breakpoint the flexbox is responsive by default. The more space there is, the more Cards there are on each row. On large and extra-large viewports there are four cards on each

row by default. Use the `flex-basis` property to get three cards on each row. The `flex-basis` property specifies the initial length of a flexible item. You can use the following SCSS code to set the flex-basis property:

```
.card-deck .card {
  flex-basis: 30%;
}
```

On a medium grid you will got two cards in each row. The last row has only one card, because of we have an odd number of cards. The last card takes 100% of the available space and will look like that shown in the following screenshot:



You can try to fix this with Sass by setting the `max-width` for each card as follows:

```
@include media-breakpoint-up(sm) {
  .card-deck .card {
    max-width: 46%;
  }
}
```

Or alternatively, add an empty card that's only visible on the medium grid by using Bootstrap's responsive utilities classes:

```
<div class="card hidden-xs-down hidden-lg-up">
   <!-- empty card -->
</div>
```

If you've not removed the borders and rounded corners of the cards, you should remove them for the empty card. You can remove the borders and/or rounded corners with Sass. Use the following SCSS code to remove the borders from the empty card:

```
@include media-breakpoint-up(sm) {
  .card-deck .card {
    &:last-child {
      border: initial; // 0
    }
  }
}
```

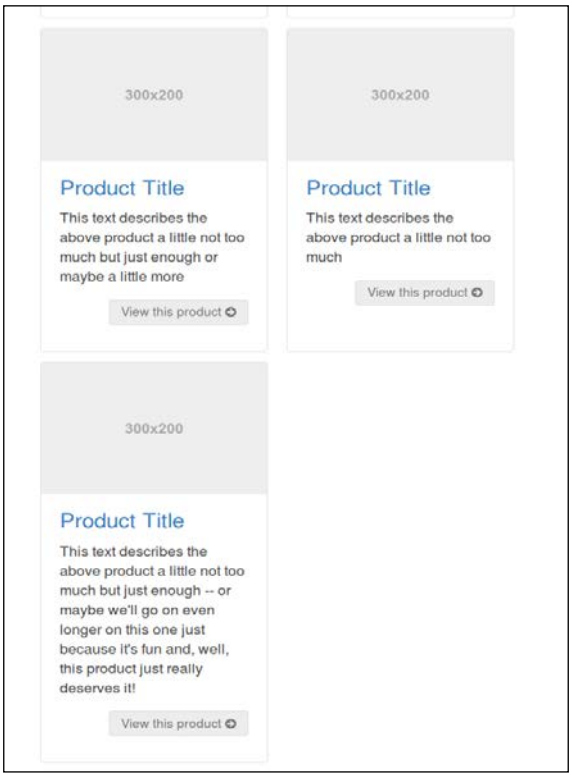
In the preceding code, `:last-child` is a CSS pseudo-class. CSS pseudo-classes can be added to selectors that specifies a special state of the element to be selected. The `last-child` pseudo-class selects any element that is the last child element of its parent. You can read more about the `last-child` pseudo-class at the following URL: `https://developer.mozilla.org/nl/docs/Web/CSS/:last-child`.



Notice that the SCSS code with the `&` parent reference in front of the `last-child` pseudo-class compiles into CSS code as follows:

```
.card-deck .card:last-child order: initial; }
```

Now the last cards on the medium grid should look as follows:



Of course, you can you can also remove just the rounded corners by using the following SCSS code:

```
.card-deck .card {
    border-radius: initial;
}
```



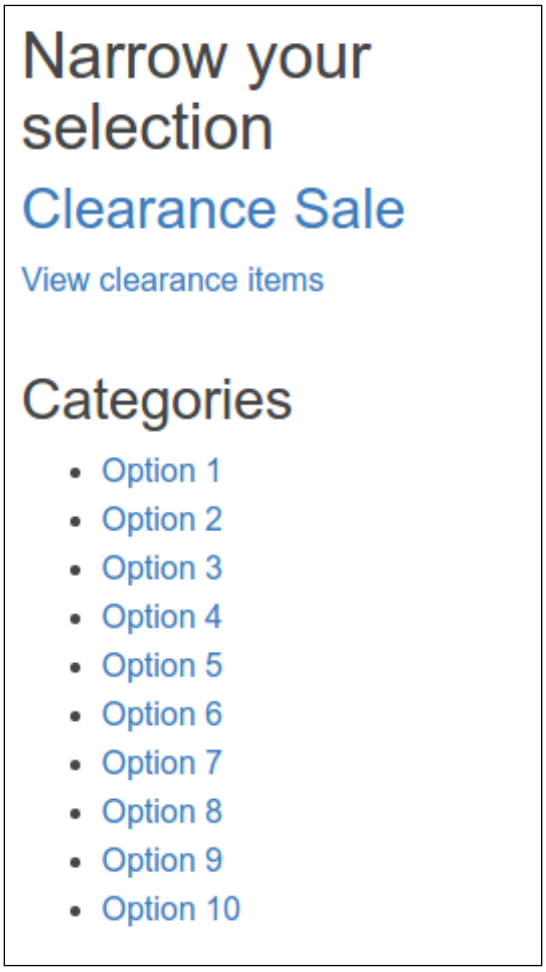
You can read more about the CSS3 flexbox layout module at `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Flexible_Box_Layout/Using_CSS_flexible_boxes`. Internet Explorer 9 and earlier do not support flexbox.

Next, we'll style the filtering options sidebar.

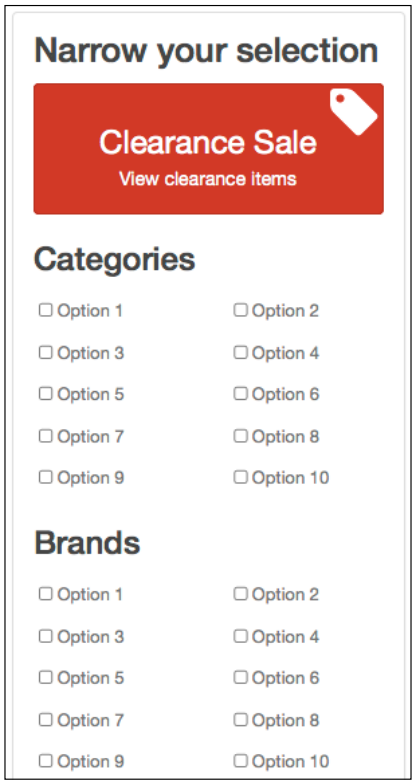
Styling the options sidebar

Now, let's style our filtering options. These appear just before the markup for our product items. In small, medium, and large viewports, they appear as a left-hand sidebar.

At the moment, they appear like the following screenshot:



For our final design, we want to transform the **Clearance Sale** link into an attractive extra-large button and arrange the filtering options into two columns with checkboxes rather than bullets, as shown in the following screenshot:



Let's begin by setting up some basic styles to lay a basic groundwork.

Setting up basic styles

We'll start by adjusting fonts, colors, margins, and padding.

Let's add these rules to a new Sass partial called `_grid-options.scss`:

```
.grid-options {
  @extend .card;
  padding-top: 12px;
  padding-bottom: 24px;
  > h2 {
    margin-top: 0;
    font-size: 1.3 * ($font-size-lg);
    line-height: 1.2;
    color: $gray-dark;
  }
}
```

The preceding code does the following:

- Adds Bootstrap Card styles to our sidebar (see the relevant Bootstrap documentation at `v4-alpha.getbootstrap.com/components/card/`)
- Adds top and bottom padding to the sidebar so that our new background extends past the sidebar content
- Adjusts font size, line-height, and color for the `h2` heading

Notice that you should not forget to import the `_grid-options.scss` file into your `app.scss` file.

Next, we will style the **Clearance Sale** link.

Styling the Clearance Sale link

We want to transform our **Clearance Sale** link into an extra-large attractive button.

Let's adjust the markup to do the following:

- Turn the linked heading and paragraph into a button.
- Add the custom button `btn-feature` class, which we created in *Lesson 9*, *Bootstrapping Business*, to give the button our special featured color—red.
- Add a Font Awesome icon for a sale tag. We'll make it three times the normal size by using Font Awesome's built-in `icon-3x` class.

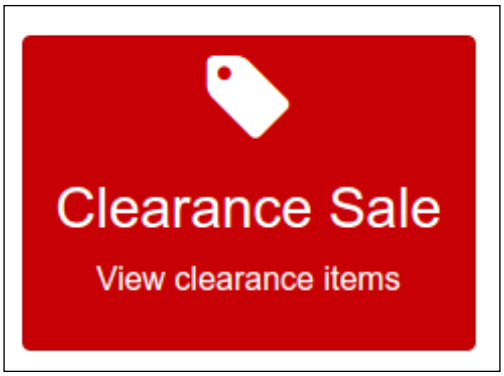


For more information about Font Awesome's special sizing classes, see the documentation at `http://fontawesome.io/examples/#larger`.

The resulting HTML markup is as follows:

```
<a class="btn btn-feature choose-clearance" href="#">
  <span class="icon fa fa-tag fa-3x"></span>
  <h3>Clearance Sale</h3>
  <p>View clearance items</p>
</a>
```

This immediately gives us a good start towards our desired result as shown in the following screenshot:



Now to polish it up, perform the following steps:

1. Display the **Clearance Sale** button as a block-level element by setting the display property to block and center it by extending Bootstrap's `m-x-auto` class. The `m-x-auto` class is part of Bootstrap's Utility classes and centers fixed-width block level content by setting the horizontal margins to auto.
2. Force its width to fill 92.5 percent of its containing column.
3. Add top and bottom padding.
4. Override Bootstrap's `white-space:nowrap` rule for buttons, so that our text can wrap as it should (See Bootstrap's white-space rule in `/bootstrap/scss/_buttons.scss`. You can learn more about the `white-space` property at `http://css-tricks.com/almanac/properties/w/whitespace/`.)
5. Position it relatively so that we can apply absolute positioning to the tag icon.
6. Adjust font, color, font-size, and margins on our heading and paragraph.
7. Position the tag icon at the top right.

We can accomplish these goals by adding the following style rules:

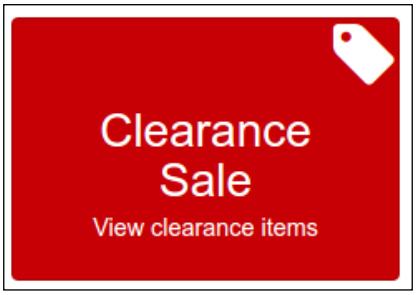
```
.choose-clearance {
  @extend .m-x-auto;
  display: block;
  width: 92.5%;
  padding-top: $spacer-y * 2;
  padding-bottom: $spacer-y;
  font-size: 90%;
  white-space: normal;
  position: relative;
  h3 {
    font-weight: normal;
    color: #fff;
```

```
      padding-top: $spacer-y / 2;
      margin: $spacer / 3;
    }
    p {
      margin: $spacer / 3 $spacer * 2;
      line-height: 1.2;
    }
    .icon {
      position: absolute;
      top: 0;
      right: 2px;
    }
  }
```

Notice that the background-color of the **Clearance Sale** button is set by the class in the HTML code in the `html/pages/products.html` file. The `btn-feature` class is generated via the `scss/includes/_buttons.scss` partial file with the following SCSS code:

```
.btn-feature {
  @include button-variant($btn-feature-color, $btn-feature-bg,
  $btn-feature-border);
}
```

At the end, this gives us a pleasing result:



As a bonus, these styles work well across viewport sizes. Take a few moments to test it. Then of course, as always, feel free to take what we've begun and beautify it further.

Meanwhile, let's move down to the options for filtering our products.

Styling the options list

In this section, we will transform our lists of product filtering options.

If you take a moment to examine the markup of product filtering options in a store such as Amazon (`http://www.amazon.com`) or Zappos (`http://www.zappos.com`), you'll find that they are composed lists of links that have been specially styled to appear like checkboxes. We will style our links to look like checkboxes, which will appear as checked once selected, and we'll adjust them to work nicely across devices, such as tablet and phone devices.



On e-commerce websites such as Amazon and Zappos, the filter options are connected to a content management system, which dynamically updates the grid of shown products in response to the options selected. Bootstrap is a frontend design framework, and not a content management system. Thus, we will not be dynamically filtering our products as a part of this project. Instead, we will prepare a design that is ready to be used in the context of a complete content management system.

In the coming section, we'll use the HTML code from the `html/pages/products.html` file. The HTML code of an option list may look as follows:

```
<h3>Brands</h3>
  <ul class="options-list options-brands">
    <li><a href="#">Option 1</a></li>
    <li><a href="#">Option 2</a></li>
    <li><a href="#">Option 3</a></li>
    <li><a href="#">Option 4</a></li>
    <li><a href="#">Option 5</a></li>
    <li><a href="#">Option 6</a></li>
    <li><a href="#">Option 7</a></li>
    <li><a href="#">Option 8</a></li>
    <li><a href="#">Option 9</a></li>
    <li><a href="#">Option 10</a></li>
  </ul>
```

Edit your SCSS code in the `scss/includes/_grid-options.scss` partial file. We'll start with the `h3` headings for the lists, adjusting their size, line-height, margin, and color:

```
.grid-options {
  > h3 {
    font-size: $font-size-lg;
    line-height: 1.2;
```

```
      margin-top: $spacer-y / 2;
      color: $gray-dark;
    }
  }sss
```



We need to use the `>h3` child selector since we don't want these rules to apply to other `h3` tags, especially the one within our **Clearance Sale** button.

Now, let's turn our attention to the unordered lists. These have a special class of `options-list`, which we'll use as our selector to ensure we're targeting only these special lists.

First, let's remove bullets and padding:

```
.options-list {
  list-style-type: none;
  padding-left: 0;
}
```

Now we'll style the links. Shortly, we'll also style the list items, so we'll include them in the sequence of nested selectors.

```
.options-list {
  list-style-type: none;
  padding-left: 0;
  li {
    a {
       @extend .btn;
       @extend .btn-sm;
       padding-left: 0;
       padding-right: 0;
       color: $gray;
       @include hover-focus-active {
         color: $link-color;
       }
     }
   }
}
```

The rules we just set accomplish the following:

- We'll use the power of Sass' extend feature to pull in the fundamental button styles associated with the `btn` class that includes displaying the `inline-block` link and the addition of padding.

- Since we added no other button class, there is no background color
- What we gain from these basic button styles is a convenient way to make our links user-friendly click targets-including fingers on touch devices
- We then extend the styles associated with the `btn-sm` class to reduce padding and for the font-size to be a bit smaller than the standard button (for a refresher on Bootstrap button classes, go to `v4-alpha.getbootstrap.com/components/buttons/`)
- We then remove unneeded left and right padding
- We change the color of our link text to `$gray`
- Finally, we set the color of hovered, focused, and active links to our `$link-color` value

You may want to save, compile, and test the results. The following screenshot depicts the result we get:



Our option links have gained improved padding and font size and taken our desired colors.



You may be wondering why I've chosen to extend the button styles by using the `.btn` and `.btn-sm` classes in our Sass files rather than adding the classes directly to the markup. We could do the latter, but given the number of option links, I think you will agree that it is far more efficient to apply the styles via CSS as we've done. In the section that follows, I will continue this pattern and extend it by bringing in Font Awesome icons via Sass rather than by adding markup.

Now we'll add checkboxes to our option links.

Adding Font Awesome checkboxes to our option links

In this section, we'll use Font Awesome icons to add an empty checkbox to the left of each option link. Rather than adding icons in the markup, we will do it here via Sass as it will be far more efficient. Then we'll push a step further, adding styles to pull in an alternate Font Awesome icon-for a checked checkbox - to the hovered, focused, and active option links.

Adding icons via Sass requires extending Font Awesome styles. First, we will take these fundamental styles from the fa base class, which can be found in the `_core.scss` file in the `bower_components/font-awesome` folder. In this file, you'll find the following key styles:

```
.#{$fa-css-prefix} {
  display: inline-block;
  font: normal normal normal #{$fa-font-size-base}/#{$fa-line-height-
base}
  FontAwesome; // shortening font declaration
  font-size: inherit; // can't have font-size inherit on line above,
  so need to override
  text-rendering: auto; // optimizelegibility throws things off #1094
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```



In the preceding code we've used the `.#{$fa-css-prefix}` selector, which is based on Sass' variable interpolation. The Sass compiler uses the `#{}` interpolation syntax to compile variables into selectors and property names. Read more about variable interpolation in Sass at `http://sass-lang.com/documentation/file.SASS_REFERENCE.html#interpolation_`.

These styles establish the fundamental rules for all Font Awesome icons, including the Font Awesome icon for the font family and then refine the details of its presentation.

For our present purposes, we do not need the selector or the braces but only the rules. We will take these and apply them to our links. Primarily, we'll use the `:before` pseudo-element as it ensures the best results.

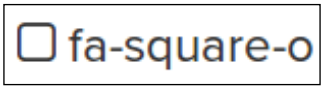


For more information about the CSS 2.1 `:before` pseudo-element, go to `http://coding.smashingmagazine.com/2011/07/13/learning-to-use-the-before-and-after-pseudo-elements-in-css/`.

So edit the following rules in the `_grid-options.scss` file, nested as follows:

```
.options-list {
  li {
    a {
      &:before {
        @extend .#{$fa-css-prefix};
      }
    }
  }
}
```

These rules establish the fundamentals. Next, we need to specify which Font Awesome icon to use. Browsing the options at http://fontawesome.io/icons/, we find the following open checkbox icon:



The Sass rules for this icon are found in the `_icons.scss` file inside the `font-awesome` folder. By opening that file and searching for the `}-square-o` string (including the closing curly brace before `-s` to narrow the results), we can find the following relevant line:

```
.#{$fa-css-prefix}-square-o:before { content: $fa-var-square-o; }
```

From the previous line, we only need `content: $fa-var-square-o`, which we can copy and paste in the `_grid-options.scss` file directly after the preceding rules, which are applied to our `a:before` selector or alternatively extend the `.fa-square-o:before` selector:

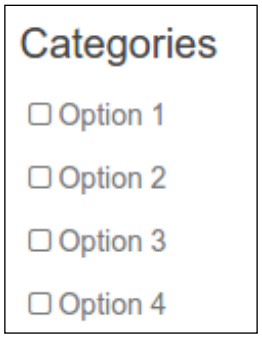
```
.options-list {
  li {
    a {
      &:before {
        @extend .#{$fa-css-prefix};
            @extend .#{$fa-css-prefix}-square-o:before;
      }
```

```
      }
    }
  }
```

Finally, we want to grab Font Awesome styles to give our icons a fixed width and to avoid any shifting when the icon changes to the checked version. These styles are found in the `_fixed-width.scss` file inside the `ss` folder. Extend the `.fa-fw` class as follows:

```
.options-list {
  li {
    a {
      &:before {
        @extend .#{$fa-css-prefix};
        @extend .#{$fa-css-prefix}-square-o:before;
        @extend .#{$fa-css-prefix}-fw;
      }
    }
  }
}
```

After adding these rules, run the bootstrap watch command and inspect the results in your browser. You should see the checkboxes appear as shown in the following screenshot:



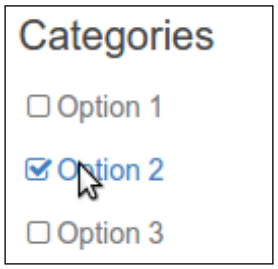
Now, following the same approach, we'll add the following selectors and rules to apply the checked version of the Font Awesome icon to the hovered, focused, and active states of our links:

```
.options-list {
  li {
    a {
      &:before {
```

```
            @extend .#{$fa-css-prefix};
            @extend .#{$fa-css-prefix}-square-o:before;
            @extend .#{$fa-css-prefix}-fw;
          }
          @include hover-focus-active {
            color: $link-color;
            &:before {
              content: $fa-var-check-square-o;
            }
          }
        }
      }
    }
  }
```

Bootstrap's `hover-focus-active` mixin can be found in the `bower_components/bootstrap/scss/mixins/_hover.sccs` partial. You can use this mixin to set the active, hover, and focus states once.

Save the file, and inspect the results in your browser. You'll find that the checked version of the square icon appears when you hover on one of the links, as shown in the following screenshot:





As a reminder, it is not currently possible to force one of these links to stay in the active state as we have no content management system in place. What we do have is a set of styles ready and waiting to go to work in the context of such a content management system.

That's it! We've successfully given our links the appearance of checkboxes to provide desired user feedback.

Next, let's make more efficient use of our space by floating our options side by side.

Using Sass mixins to arrange option links in columns

In the previous section, we used custom Sass rules to accomplish things that might have been accomplished by adding markup. Given the number of option links we need to manage, this has proven significantly more efficient. The same dynamic applies when we want to arrange our option links into columns.

We might accomplish our desired result by using Bootstrap row and column classes, adjusting our markup with the following pattern:

```
<ul class="options-list options-categories row">
     <li class="col-xs-6"><a href="#">Option 1</a></li>
     <li class="col-xs-6"><a href="#">Option 2</a></li>
     ...
```



In *Lesson 9*, *Bootstrapping Business*, you saw that the Panini template engine does support loops and iterations. Using loops in your template is also an alternative for DRY coding and preventing duplicate code.

An example can be found in the `html/pages/products.html` file. The code of the first list looks as follows:

```
{{#each numbers-10}}
<li><a href="#">Option {{this}}</a></li>
{{/each}}
```

The `numbers-10` variable is read from the file, which contains the number 1 through 10 in YAML format. Using the index of the iteration seem to make more sense, but unfortunately, Panini does not support this feature of Handlebars. See also `https://github.com/zurb/panini/issues/67`.

Thanks to the power of Bootstrap's mixins, we can accomplish the same result with a few lines of Sass, as shown in the following steps:

1. First, we'll apply the `make-row()` mixin to the `options-list` selector, as follows:

```
.options-list {
  @include make-row();
}
```

2. This mixin applies the same styles to our options list that we would have gained by applying the `row` class in the markup. In this case, it's simply more

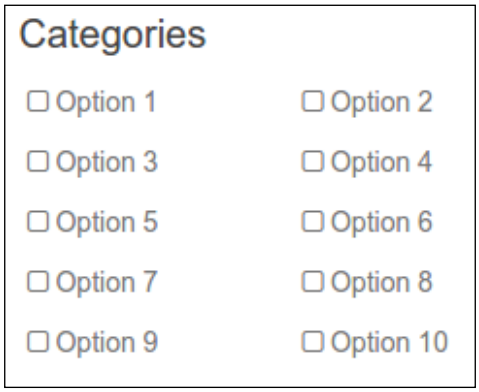
efficient to do it here.

3. Next, we can use a `make-col(6)` mixin and set the number of column to six to apply column rules to our list items as follows:

```
.grid-options {
  @include make-row();
  li {
    @include make-col-ready();
    @include make-col(6);
  }
}
```

4. This will apply the same styles to our list items as would be applied if we had added the `col-xs-6` class to each of the relevant `li` tags. Later on, you will read how to make the columns responsive.

After adding the preceding lines, save the file, compile to CSS, and refresh your browser. You should see the option links line up in two columns:



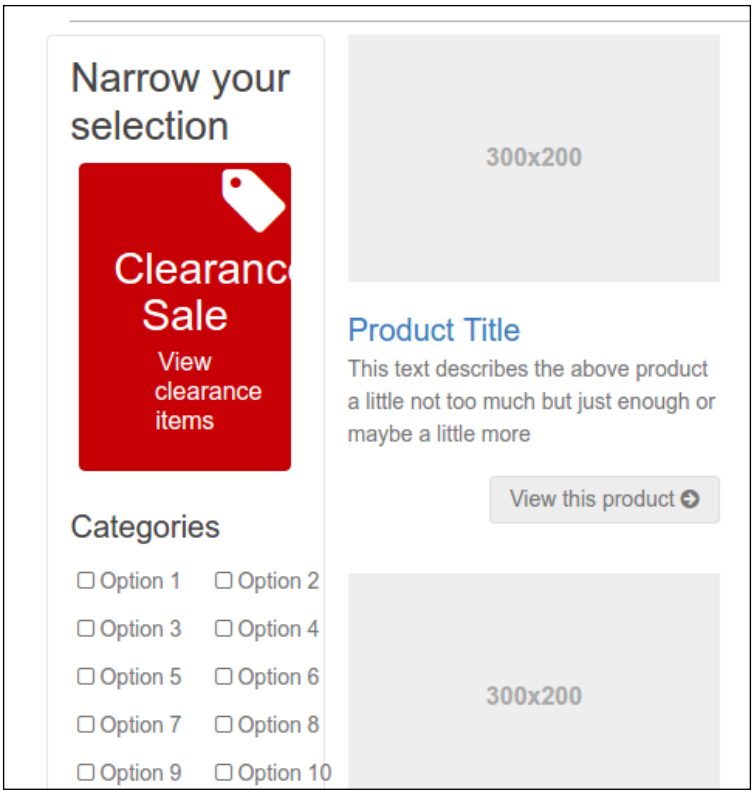
Not bad!

We will now make some adjustments for smaller viewports.

Adjusting the options list layout for tablets and phones

We need to constrain the width of our options panel so that it does not range too widely in tablet-width devices.

On the medium grid for tablets, between 768 and 992 pixels, neither the **Clearance Sale** button nor the options fit the column, as shown in the following screenshot:



We can fix this issues with the options by using Sass and force the options into a single column again for the medium grid as follows:

```
.grid-options {
  @include make-row();
  li {
    @include make-col-ready();
    @include make-col(6);
    @include media-breakpoint-only(md) {
       @include make-col(12);
    }
  }
}
```

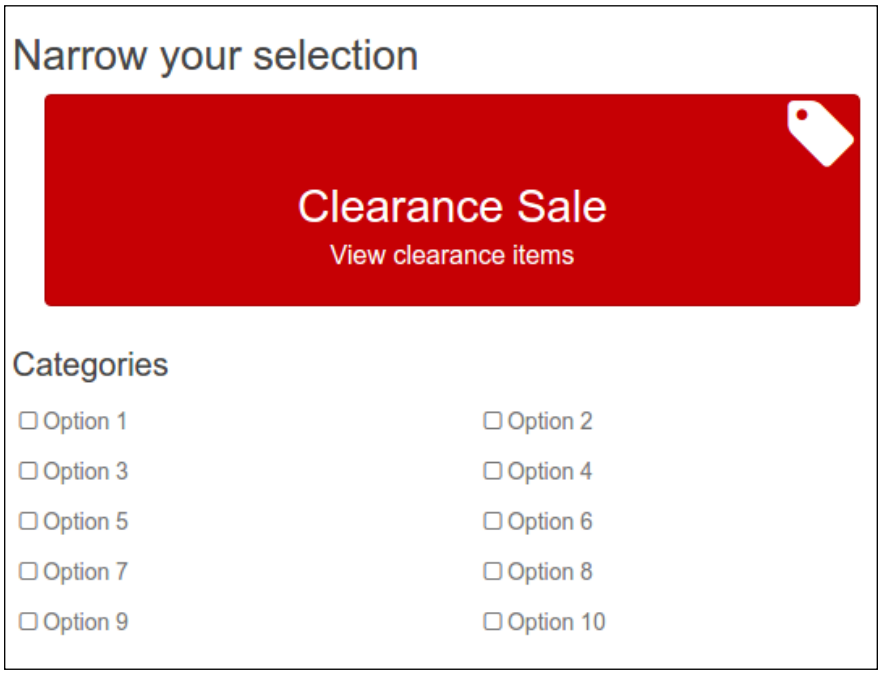
The preceding SCSS code does not fix the **Clearance Sale** button for the medium grid. You may try to reduce the font size to fix it.

Alternatively, you can adopt the main grid to solve the issues on the medium grid. In the `html/pages/products.html` file change the grid classes as shown here:

```
<div class="grid-options col-md-4 col-lg-3">
 ...
</div>
<div class="products-grid col-md-8 col-lg-9">
 ...
</div>
```

The grid-options area will span four columns on the medium grid now. The problems for the medium grid are fixed. Let's take on the small grid.

Right now, our **Clearance Sale** button stretches too wide, and our options list items spread too far apart on viewports between 480 pixels and 768 pixels wide. Thus, they can end up appearing like the following screenshot:



This can be easily fixed by setting a `max-width` property with a value of 480 pixels for the entire options panel:

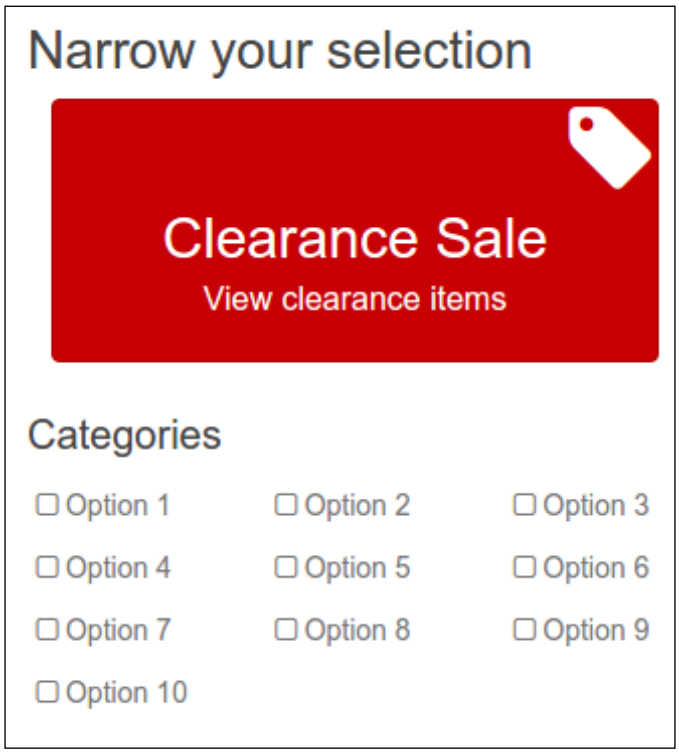
```
.grid-options {
  max-width: 480px;
}
```

Now let's adjust our option list items so that they organize themselves in three columns in small viewports. Using Sass, we can nest a media query within the

appropriate selector and add an adjusted `make-col(4)` mixin, as shown in the following code snippet:

```
.grid-options {
  @include make-row();
  li {
    @include make-col();
    @include make-col(6);
    @include media-breakpoint-down(sm) {
       @include make-col(4);
    }
  }
}
```

After making these adjustments, save the file and test in a narrow viewport. You should see the result shown in the following screenshot:



Now let's address the next problem facing our single-column layout: we need to hide our options away until they're needed.

Collapsing the options panel for phone users

At present, our options take up a considerable amount of vertical space. This creates a problem in narrow viewports. The single-column layout winds up pushing our grid of products far down the page.

This is a great deal of vertical space for options that are not needed. The products themselves are priority items. We need to allow users of phones to find the products more quickly while still allowing them to access the filtering options when desired.

We'll use Bootstrap's collapse plugin for this. In the following steps, we'll apply the collapse plugin to the options panel, add a button to expand the panel when desired, and restrict the behavior to narrow viewports only:

1. Open your editor with `products.html`.
2. Add a new `div` tag to wrap our **Clearance Sale** button and three options lists. We need to give this new `div` a special class of collapse as well as a distinctive ID so that we can target it with our JavaScript plugin. For good measure, we'll give it a matching special class as well:

```
<h2>Narrow your selection</h2>
<div id="options-panel" class="options-panel collapse">
...
</div>
```

3. Notice that the `collapse` class in the previous step hides the content for all viewports. You can add the `navbar-toggleable-sm` class to ensure the content is always visible on larger viewports:

```
<h2>Narrow your selection</h2>
<div id="options-panel" class="options-panel collapse
navbar-toggleable-sm">
...
</div>
```



Bootstrap's collapse JavaScript plugin is what powers the collapsible responsive navbar. It may also be put to other uses, such as the one shown in the Bootstrap documentation at `v4-alpha.getbootstrap.com/components/collapse/`.

4. Save the file and refresh it in your browser. You should see that the **Clearance Sale** button and options lists will now be hidden from view. All that remains of the options panel content will be the `h2` heading **Narrow your selection**, as shown in the following screenshot:



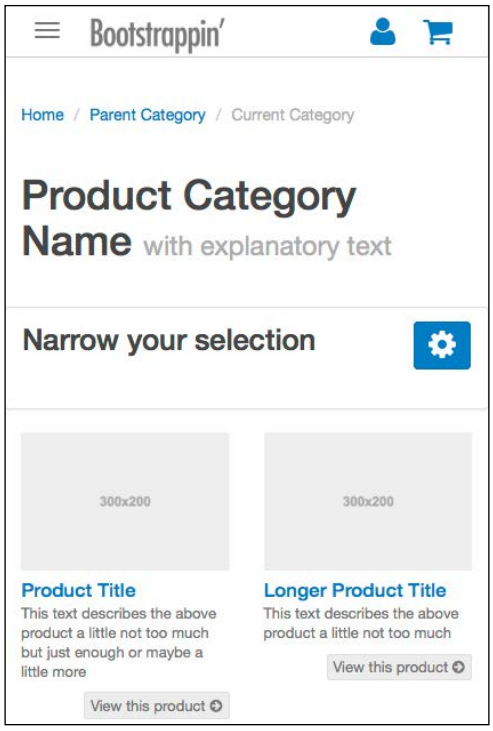
5. Now we need a toggle button to expand our filter options when clicked.
6. Within the still visible `h2` heading that reads **Narrow your selection**, add a button element with the following attribute structure:

```
        <h2 class="clearfix">Narrow your selection
          <button type="button"
            class="options-panel-toggle btn btn-primary pull-right
hidden-md-up"
            data-toggle="collapse" data-target="#options-panel">
            <span class="icon fa fa-cog fa-2x"></span>
          </button>
        </h2>
```

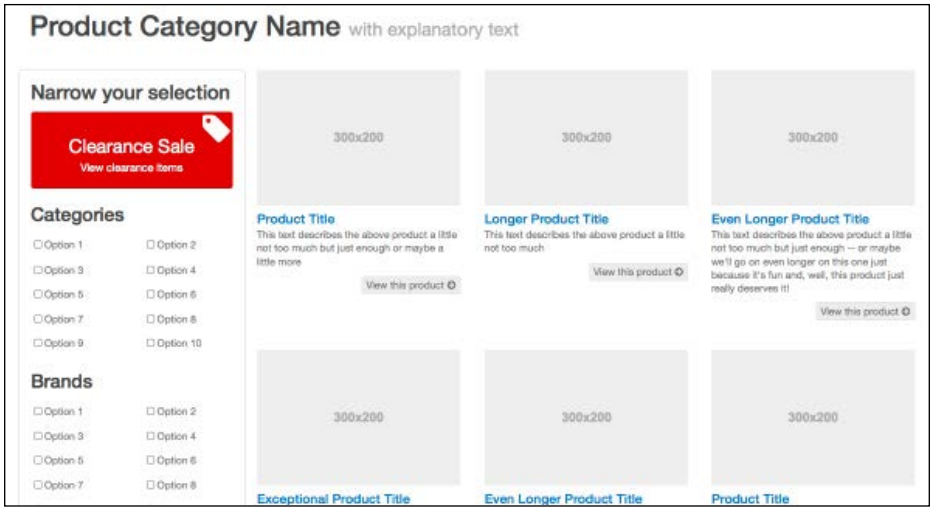
7. The following points explain what the preceding markup will do:
   - The `clearfix` class will ensure that the h2 heading will contain the toggle button, which will float to the right due to the pull-right class.
   - The `btn` and `btn-primary` classes will style our new button element with the Bootstrap's `btn` styles, which includes our background color of `$brand-primary`
   - The `hidden-md-up` class hides the button on larger viewports.
   - Within the button element, we've placed a Font Awesome icon using the `fa-2x` class to double its size.
   - Save this and find the following result in your browser:



8. In narrow viewports, the options list is collapsed and the toggle button is visible:



9. In small, medium, and large viewports, the toggle button is hidden, and the options list is visible:



Adding a search form to your designing

In the preceding sections we've build a navigation structure. About fifty percent of your visitors will use this navigation, the other half will prefer to search your content. So a good ability to search the content and your products must always be represented on your pages.

We can add a search form in the header of our page, which should look as follows:



Edit the following HTML code in the `html/includes/header.html` file:

```
<div class="utility-nav">
   <ul>
     <li><a href="#" ><i class="icon fa fa-user
     fa-lg"></i><span> Log In or Register</span></a></li>
     <li><a href="#" ><i class="icon fa fa-shopping-cart
     fa-lg"></i><span> View Cart</span></a></li>
   </ul>
</div>
<form class="search-form form-inline pull-md-right">
  <input class="form-control" type="text" placeholder="Search">
  <button class="btn btn-outline-success hidden-sm-down"
  type="submit">Search</button>
</form>
</div>
```

The following points explain what the preceding markup will do:

- The form-inline and form-control classes are Bootstrap classes for inline forms. You can read more about Bootstrap's inline forms at `v4-alpha.getbootstrap.com/components/forms/#inline-forms`.
- The `pull-md-right` class will ensure that the form float on the right side of the header for
- The `hidden-sm-down` class hides the search button on the small viewports; only the search input is visible in these viewports.

The preceding code overlaps the icons; you can fix this by setting a padding-top using the following SCSS code in the `scss/includes/_header.scss` Sass partial:

```
header[role="banner"] {
```

```
    .search-form {
      @include media-breakpoint-up(md) {
        padding-top: $spacer-y * 6;
      }
    }
  }
```

The `media-breakpoint-up(md)` mixin call ensures that the padding is only added for medium and larger viewports.

Using the Typeahead plugin

Adding an autocomplete function to your search form may improve the usability of search function. The `Typeahead` plugin from Bootstrap 2 can be used to build an autocomplete function. More information about this plugin can be found at `https://github.com/bassjobsen/Bootstrap-3-Typeahead`. The plugin is ready to use with Bootstrap 4.

The following steps describe how to integrate the `Typeahead` plugin in your project:

1. First, add the plugin to your bower project dependencies to the `bower.json` file of your project, as follows:

```
   "dependencies": {
    "bootstrap": "4",
    "tether": "^1.1.2",
    "font-awesome": "^4.6.1",
    "bootstrap3-typeahead": "git://github.com/bassjobsen/
Bootstrap-3-
    Typeahead.git#master"
  }
```

2. Then run the bootstrap update or bower update command in your console.
3. Then edit the `compile-js` task in Gruntfile.js to ensure that the plugin is included in your project:

```
gulp.task('compile-js', function() {
  return gulp.src([
        bowerpath+ 'jquery/dist/jquery.min.js',
        bowerpath+ 'tether/dist/js/tether.min.js',
        bowerpath+ 'bootstrap/dist/js/bootstrap.min.js',
        bowerpath+ 'holderjs/holder.min.js', // Holder.js for
project
        development only
        bowerpath+ 'bootstrap3-typeahead/bootstrap3-typeahead.min.
js',
```

```
        'js/main.js'])
    .pipe(concat('app.js'))
    .pipe(gulp.dest('./_site/js/'));
});
```

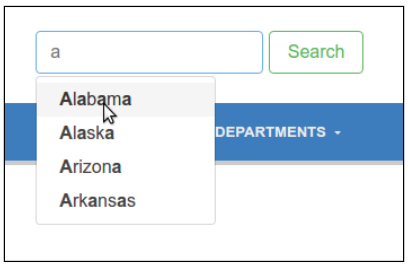
4. Then initiate the plugin and attached it to the search form. Open the `js/main.js` and edit the following JavaScript code into it:

```
$('.search-form .form-control').typeahead( { items: 4, source:
["Alabama","Alaska","Arizona","Arkansas","California","Colorado"
,"Connecticut","Delaware","Florida","Georgia","Hawaii","Idaho","
Illinois","Indiana","Iowa","Kansas","Kentucky","Louisiana","Main
e","Maryland","Massachusetts","Michigan","Minnesota","Mississipp
i","Missouri","Montana","Nebraska","Nevada","New Hampshire","New
Jersey","New Mexico","New York","North Dakota","North Carolina",
"Ohio","Oklahoma","Oregon","Pennsylvania","Rhode Island","South
Carolina","South Dakota","Tennessee","Texas","Utah","Vermont","Vir
ginia","Washington","West Virginia","Wisconsin","Wyoming"] });
```

5. And lastly, set the CSS `z-index` value of the suggestions menu to prevent the navbar overlapping it. You can set the `scss/includes/_header.scss` partial by using the following SCSS code:

```
header[role="banner"] {
  .search-form {
    @include media-breakpoint-up(md) {
      padding-top: $spacer-y * 6;
    }
    .typeahead.dropdown-menu {
      z-index: 2000;
    }
  }
}
```

Now your search form with autocomplete is ready. Run the Bootstrap watch command and expect the result in your browser. Type the capital `A` in the search form and you will find that a drop-down list with suggestions appears:





The Bootstrap team dropped the `Typeahead` plugin in version 3 in favor of using typeahead.js; see `https://github.com/twitter/typeahead.js`. To use `typeahead.js` with Bootstrap 4, some additional CSS code is required. The required CSS code, include the generation SCSS code, can be found at `https://github.com/bassjobsen/typeahead.js-bootstrap4-css/`.

Congratulations! With the search form, we have accomplished our design.

Summary

In this lesson, we have employed Bootstrap styles to quickly set up breadcrumbs, a page title, and pagination customized according to our needs. Then we created a visually pleasing grid of product items, all of the same height so as to ensure a regular grid by using Bootstrap's mobile first and responsive grid styles.

We finished our design by adding a complex **Clearance Sale** button with a `$brand-feature` red background color and a list of filter options that are easily clickable. We also used Bootstrap column classes with responsive adjustments to arrange our options list items optimally for multiple viewport widths.

At the end, we also added a search form with an autocomplete function.

Congratulations! We now have an attractive business website with a well-crafted e-commerce section.

Assessments

1. Which of the following snippet will set `max-width` for cards?
    i.
    ```
    $include media-breakpoint-up(md) {
    .card-deck .card {
    width: 46%;
    }
    }
    ```
    ii.
    ```
    @include media-breakpoint-up(sm) {
    .card-deck .card {
    max-width: 46%;
    }
    }
    ```
    iii. Both i & iib
    iv. None of the above
2. What does the following snippet do?

```
.grid-options {
@extend .card;
padding-top: 12px;
padding-bottom: 24px;
> h2 {
margin-top: 0;
font-size: 1.3 * ($font-size-lg);
line-height: 1.2;
color: $gray-dark;
}
}
```

    i. Adds Bootstrap Card styles to our sidebar
    ii. Adds top and bottom padding to the sidebar so that our new background extends
    past the sidebar content
    iii. Adjusts font size, line-height, and color for the h2 heading
    iv. All of the above

3. What is the purpose of setting the CSS `z-index` value of the suggestions menu?
    i. To prevent the navbar overlapping the menu
    ii. To add plugin to your bower project dependencies
    iii. To edit the `compile-js` task in `Gruntfile.js` to ensure that the plugin is included in your project
    iv. All of the above

4. What is the advantage of adding an autocomplete function to your search form?
    i. To read more about Bootstrap's inline forms
    ii. To acquire more information about this plugin
    iii. To improve the usability of search function
    iv. None of the above

5. Which of the following is the use of the `clearfix` class?
    i. It ensures that the heading will contain the `toggle button`, which will float to the right due to the `pull-right` class
    ii. It allows users of phones to find the products more quickly while still allowing them to access the filtering options when desired.
    iii. It uses Sass, to nest a media query within the appropriate selector and adds an adjusted `make-col(4)` mixin
    iv. All of the above

11
Bootstrapping a One-Page Marketing Website

We've developed some significant skills with Bootstrap. Now it's time to bring an extra touch of beauty and creativity to helping our clients achieve their full online marketing potential. So, let's create a beautiful, one-page, upscale marketing site.

We'll cover the following topics in this lesson:

- A large introductory carousel with a customized responsive welcome message
- A section for customer reviews with images and captions laid out in the masonry format
- A features list with large Font Awesome icons
- A signup section with custom-designed pricing tables
- A ScrollSpy navbar with animated scrolling behavior

Overview

We've been approached by a new prospective client. She is stricken by the beauty of one-pager websites that scroll vertically, providing a visually stimulating presentation of a product or message with a clear call to action at the end. She wants one of these.

This client is knowledgeable and discerning. She frequents `http://onepagelove.com` and has a list of her current favorites to hand. Her desired features include:

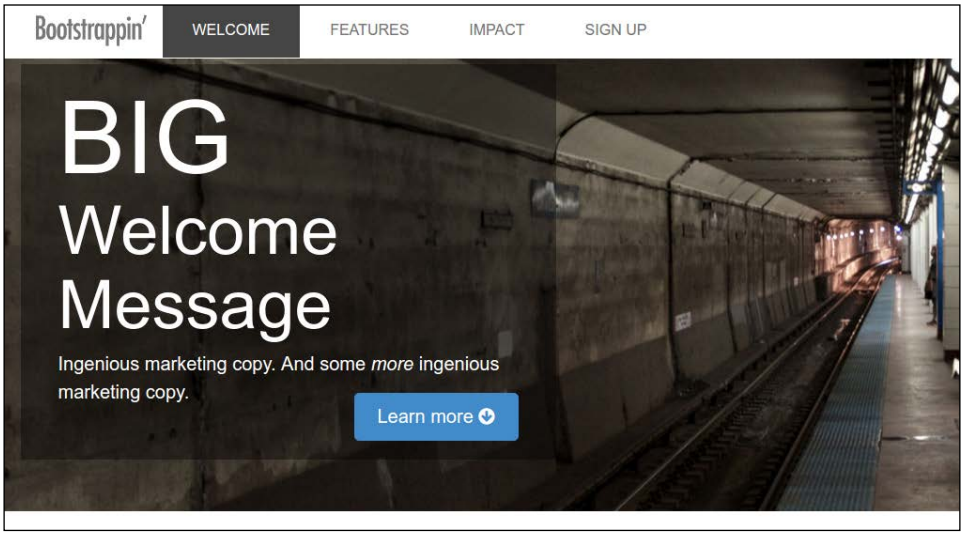
- A clean, modern, aesthetic website.
- An introductory welcome message with a visually intriguing background

image.

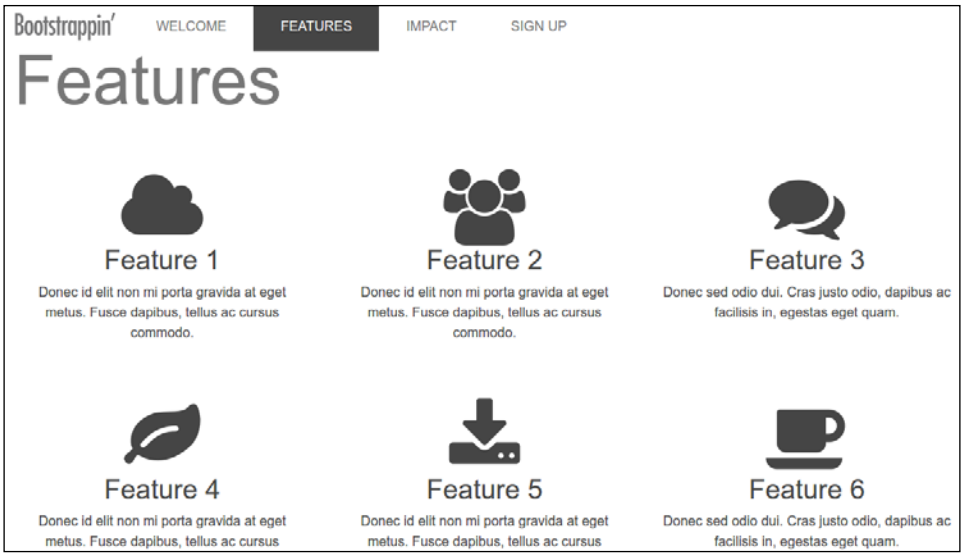
- An efficient presentation of the main features of her product, accentuated with visually appealing icons.
- Customer testimony presented in a visually stimulating way.
- An easy-to-understand overview of three basic packages that a customer can choose from. These need to be presented clearly in a way that makes it easy to choose the right fit and then sign up!
- Conversions! Everything should draw the user down the page, making it nearly impossible to avoid clicking on the Signup button at the end.

To protect the secrecy of her upcoming product launch, our client has chosen not to reveal the exact nature of her product or service to us. Rather, she has provided mock-ups of the design she would like us to create by using a dummy copy for placeholders.

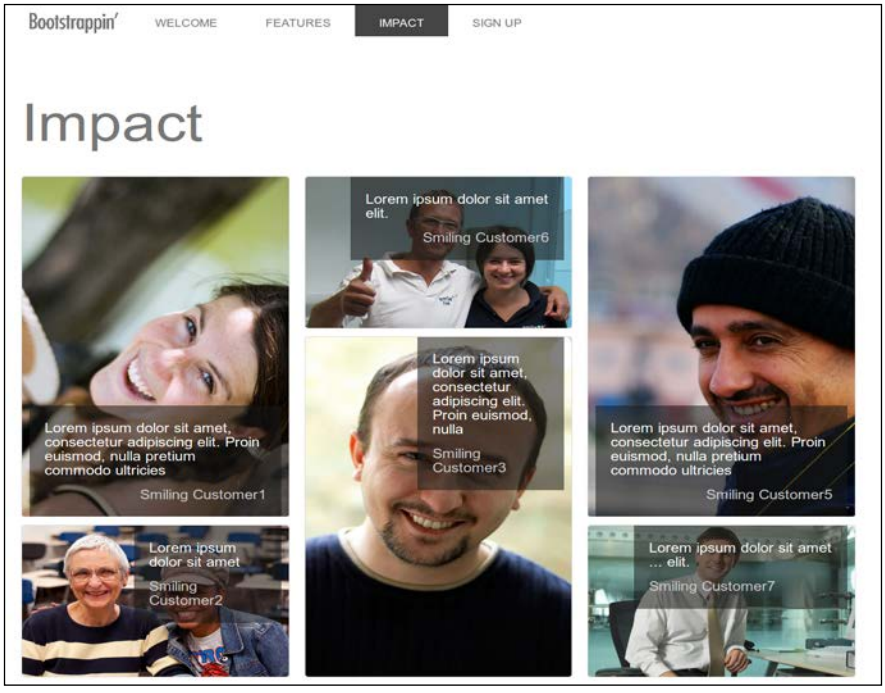
The first section will open with an interesting full-width image, a large welcome message, and an invitation to scroll down the page to learn more, as shown in the following screenshot:



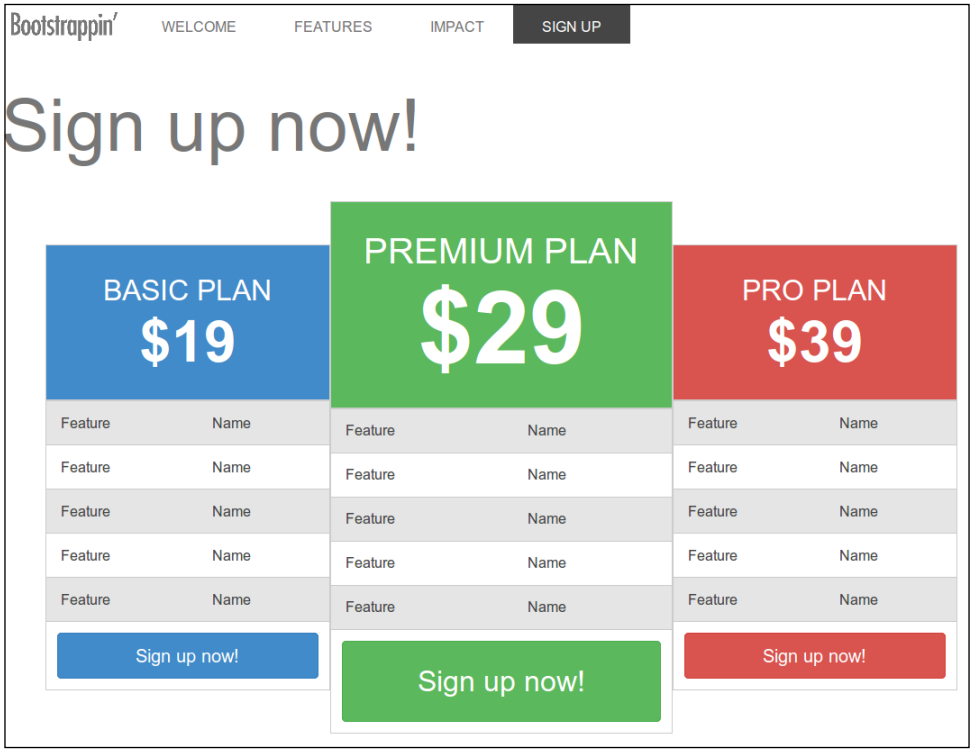
The second section will list six key features of the product, which are laid out in a three-column grid, and illustrated by appropriate icons as shown in the following screenshot:



The third section will feature client testimonies with photos and quotations laid out in the masonry style:



The fourth and final section will feature three available plans, each with a pricing table, and will have a visual emphasis on the center of the three tables, as shown in the following screenshot:



Surveying the starter files

Let's survey the initial files for this exercise. Create a new project by using Bootstrap CLI.

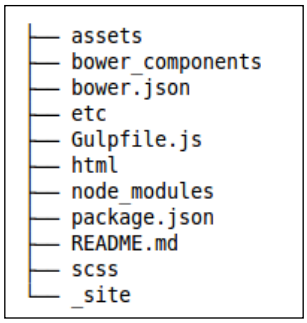
You can install Bootstrap CLI by running the following command in your console:

```
npm install -g bootstrap-cli
```

Then you can set up your project by running the following command:

```
bootstrap new
```

Again, choose the **An empty new Bootstrap project. Powered by Panini, Sass and Gulp** option when prompted.

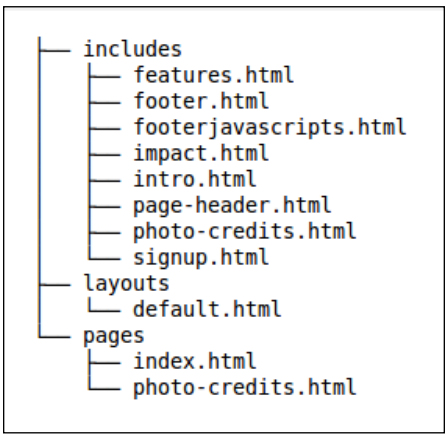


There are a few additions you will have to make now:

1. Create a new `assets/images` folder.
2. Copy the files in the images folder to the new `assets/images` folder. It contains five images:
    - One logo image, named logo.png.
    - Two background images for the intro section.
    - Seven images of happy people for the Impact section.
3. The images are automatically copied to the _site folder by the copy task in the `Gulpfile.js` file:

```
// Copy assets
gulp.task('copy', function() {
  gulp.src(['assets/**/*']).pipe(gulp.dest('_site'));
 });
```

The `html` folder, which contains your Panini HTML templates, should have the file and folder structure shown as follows:



You can read more about Panini at `https://github.com/zurb/panini`.

Instead of the modifications described above, you can also start with the files found in the `Lesson 11/start` folder. In this folder, run the `npm install` and `bower install` commands first. After running the `npm` and `bower` commands, you can run the `bootstrap watch` or `gulp` command to view the results in your browser.

Viewing the page content

Run the bootstrap watch command and watch the page in your browser at `http://localhost:8080/`. You'll see the following major components in place. Each component got its own HTML partial. Of course, at present, they will be displayed with default Bootstrap styles, awaiting the customization that needs to be done:

- A fixed top navbar
- A jumbotron with a big welcome message
- A features section with icons, headings, and text organized in three columns
- The **Impact** section with photos of six happy customers and placeholder content for their positive testimony
- A **Sign up now!** section with three tables laying out the **Basic Plan**, **Premium Plan**, and **Pro Plan** packages, with a Sign up now! button under each
- A footer logo
- Photo credits (images are attribution-licensed)

To view the markup, open the corresponding Panini HTML partial in your editor. We will get very familiar with the markup in the steps that follow!

Adding Font Awesome to our project

Font Awesome gives you scalable vector icons that can instantly be customized: size, color, drop shadow, and anything that can be done with the power of CSS.

Here, we simply load Font Awesome's CSS code from CDN by linking it in the `html/layouts/default.html` HTML template as follows:

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.6.1/css/font-awesome.min.css">
```

Adjusting the navbar

This design calls for a fixed top navbar with a significant color shift for hovered and active links. I've already applied some of these styles by setting appropriate variables. Let me point those out, and then we'll move on to make some necessary adjustments to the markup.

The `scss/_variables.scss` file is based on Bootstrap's `variables.scss` file. I've customized the shades of gray in line with previous projects. You'll see these in the topmost section of the file.

I've further adjusted the following navbar variables, adjusting its height, margin, colors, and hover colors specifically for this design:

```
// Navbar
$navbar-bg:                   #fff;

// Navbar links
$navbar-link-color:               $gray;
$navbar-link-bg:                  #fff;
$navbar-link-hover-color:         #fff;
$navbar-link-hover-bg:            $gray;
$navbar-link-active-color:        #fff;
$navbar-link-active-bg:           $gray-dark;
```

We'll use the responsive navbar variant. The HTML code for the navbar can be found in the `html/includes/page-header.html` file and looks like the following:

```
<nav class="navbar navbar-fixed-top">
  <div class="container">
  <button class="navbar-toggler hidden-sm-up" type="button"
data-toggle="collapse" data-target="#exCollapsingNavbar2" aria-
controls="exCollapsingNavbar2" aria-expanded="false" aria-
label="Toggle navigation">
    ☰
  </button>
  <div class="collapse navbar-toggleable-xs" id="exCollapsingNavbar2">
    <a class="navbar-brand" href="index.html"><img src="{{root}}
images/logo.png" alt="Bootstrapping"></a>
    <ul class="nav navbar-nav">
      <li class="nav-item active">
        <a class="nav-link" href="#">Welcome <span class="sr-
only">(current)</span></a>
      </li>
      <li class="nav-item">
        <a class="nav-link" href="#">Features</a>
      </li>
```

```
        <li class="nav-item">
          <a class="nav-link" href="#">Impact</a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Sign up</a>
        </li>
      </ul>
    </div>
    </div>
  </nav>
```

As you can see in the preceding code, the navbar also gets the `navbar-fixed-top` class, which fixes the `navbar to the top` of our page. The `navbar-fixed-top` class also sets the `border-radius property` to `0`. The `navbar-fixed-top` class is one of Bootstrap's navbar classes determining the placement of the navbar; classes are available for static or fixed navbars.

Along with the custom variables, I've made a few adjustments to the `_navbar.scss` file.

I've customized the list items in the expanded navbar, adding padding, removing the spacing between the links, and transforming the text to uppercase:

```
.navbar {
  background-color: $navbar-bg;
  color: $navbar-link-color;
  padding: 0 1rem;

  .nav-item + .nav-item {
    margin-left: 0;
  }
  .nav-link, .navbar-brand {
    padding: $spacer-y * .75 $spacer-x * 2;
  }
}
.navbar-brand img {
  width: $brand-image-width;
}
.nav-link {
  color: $navbar-link-color;
  line-height: $brand-image-height;
  text-transform: uppercase;

  .active & {
    background-color: $navbar-link-active-bg;
```

```
      color: $navbar-link-active-color;
    }
    @include hover {
      background-color: $navbar-link-hover-bg;
      color: $navbar-link-hover-color;
    }
  }
```



Bootstrap's predefined CSS classes also contain some classes for text transformations. More information about these text capitalization classes can be found at `http://getbootstrap.com/components/utilities/#text-transform`.

The original logo image file had these settings: width `900` pixels and height `259` pixels. We can use these values to calculate the height in Sass when we resize its width to `120` pixels as follows:

```
$brand-image-width: 120px;
$brand-image-height: (259 * $brand-image-width / 900);
```

I use the `$brand-image-height` variable to set the line-height of the navbar links to ensure that the brand image and links are in line.

Now the total height of the navbar becomes `$brand-image-height + 2 * ($spacer-y * 0.75)`. We'll use this value to set the `padding-top` of the HTML body element, because the fixed navbar will overlap the body.

The `$brand-image-height` variable got pixel units, whilst the `$spacer-y` got rem units. Sass can't add up these values with different dimensions. You can remove the rem units by dividing with 1rem. Now the unitless value times `$font-size-root` will give you the value in pixels.

First, create a new `_page-contents.scss` in the main `scss/includes` folder.

Import it into `main.scss` just as shown in the following line:

```
@import "_page-contents";
```

Then calculating the padding-top value for the HTML body element in `scss/app.scss` will look like this:

```
body {
  padding-top: (2 * ($spacer-y * .75) / 1rem * $font-size-root) +
$brand-image-height;
}
```

When combined, the adjusted variables and navbar customizations yield these visual results:



Let's move on to the jumbotron with its big welcome message.

Customizing the jumbotron

The jumbotron is a Bootstrap component highlighting the key message of your website. More information about the jumbotron and its HTML markup can be found at the following URL: `http://v4-alpha.getbootstrap.com/components/jumbotron/`.

In this section, we'll customize the jumbotron to display our client's big welcome message with stylistic touches in line with her mockup. This will include adding a large background image, enlarging the welcome message text, and then adjusting its presentation for multiple viewports.

In `index.html`, find the following markup:

```
    <!-- INTRO SECTION -->
    <section class="jumbotron" id="welcome">
      <div class="container">
      <h1 class="display-3"><strong>Big</strong> Welcome Message</h1>
      <p class="lead">
        Ingenious marketing copy. And some <em>more</em> ingenious
marketing copy.<a href="#features" class="btn btn-lg btn-primary pull-
xs-right">Learn more <span class="icon fa fa-arrow-circle-down"></
span></a>
      </p>
      </div>
    </section>
```

Let's start by expanding the height of our jumbotron and putting our desired background image in place:

1. Open a new custom `Sass partial` file, `scss/includes/_jumbotron.scss`, in your editor. Don't forget to import it in the `scss/app.scss` file too.
2. Now, let's set the height, background color, and font color for the `#welcome` section. While we're at it, we'll add some top margin to the button:

```
.jumbotron {
  height: 300px;
```

```
        background-color: $jumbotron-bg;
        color: $jumbotron-color;
        .btn {
          margin-top: $spacer-y;
    }
}
```

3. The background and font color of the jumbotron are set in the `scss/includes/_variable.scss` file as follows:

```
// Jumbotron
$jumbotron-bg: #191919;
$jumbotron-color: contrast($jumbotron-bg);
```

4. The `contrast()` Sass function can be found in the `scss/functions_contrast.scss` file. The `contrast()` function uses the built-in lightness function of Sass to return a light (white) or dark (black) color depending on the lightness of the input color.



Using color contrasts in your design may improve the accessibility of your projects. When your font colors depend on the background color and change automatically when you change the base colors of your design, the changes do not influence readability and accessibility. In this lesson, we'll use a simple `contrast()` function. Sass libraries such as `Compass` have their own contrast function. Also read **Design Accessibly, See Different: Color Contrast Tips And Tools by Cathy O' Connor** at `https://www.smashingmagazine.com/2014/10/color-contrast-tips-and-tools-for-accessibility/`.

5. Next, let's use a media query to place our background image for large screens and up (991px, according to the current default Bootstrap media query breakpoint values).
6. If you like, take a few minutes to open and read the documentation about Bootstrap's responsive breakpoints again. The information can be found at `getbootstrap.com/layout/overview/#responsive-breakpoints`. All media queries are available via Sass mixins.
7. We can use the power of Sass to nest a media query within the context of the jumbotron selector. Within this media query, we'll specify the `subway-906x600.jpg` image for the background. This image is scaled to be large enough for this breakpoint while still loading relatively quickly:

```
.jumbotron {
  @include media-breakpoint-down(md) {
```

```
          background: url('#{$images-path}subway-906x600.jpg')
center center no-repeat;
        }
      }
```

Remember that the preceding SCSS code compiles into CSS code as follows:

```
@media (max-width: 991px) {
.jumbotron {
background: url("../images/subway-906x600.jpg") center center no-
repeat;
}
}
```

8. Save the file, run the `bootstrap watch` command, and inspect the results in your browser. You should see the new background image appear - but only within a window width of `991px` or less.
9. Next, let's expand the height of the jumbotron for tablet-sized viewports. We'll write a media query only the medium grid, which increases the `jumbotron` element's height to `480px` within this breakpoint:

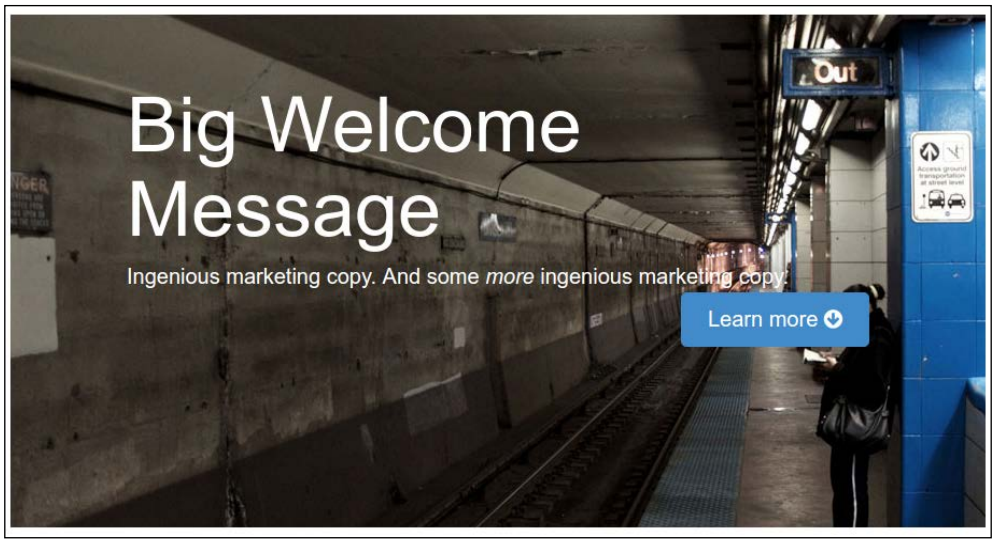
```
@include media-breakpoint-only(md) {
 height: 480px;
}
```

10. Save the file, run the `bootstrap watch` command, and watch the results in your browser. You should see the jumbotron grow to 480px in height for viewports between 768 px and 991pxin width.
11. Now, for medium and larger (greater than 992px in width) viewports, we'll increase the height of the jumbotron to 540px. At this width, we'll use the larger version of the *subway-1600x1060.jpg* background image. While we're at it, we'll set the background size to cover:

```
@include media-breakpoint-up(lg) {
  height: 540px;
  background: #191919 url('#{$images-path}subway
  -1600x1060.jpg') center center no-repeat;
  background-size: cover;
}
```

12. With these style rules in place, large viewports will have a 1600px-wide background image.
13. Save the file, and test it in your browser. You should find that we have our major breakpoints nicely covered.

After these steps, the jumbotron should look as shown in the following screenshot:



Notice that we've set a max-width for the small background image by using the `@include media-breakpoint-down(md)` mixin call. The preceding may break the mobile-first approach of Bootstrap. Media queries can be used to conditional-load background images based on screen size and so reduce load time and bandwidth on mobile phones and tablets. Read *Media Query & Asset Downloading Results* by *Tim Kadlec* for more information about testing a browser's file requests and media queries. You can find the test result at `https://timkadlec.com/2012/04/media-query-asset-downloading-results/`.

Next, we can style our big marketing message for maximum impact.

Refining the jumbotron message design

Our client wants the welcome message in the jumbotron to be extra big. Bootstrap's display-3 styles in the jumbotron increase the font size by 350 % globally. We want to enhance the results further. We also want to constrain the width of the message on wide screens and put a dark translucent box behind it.

In our current results, we should reduce the font size for small and extra-small screens. We can, however, improve the contrast of our text by placing a translucent dark overlay behind the text. Let's do that here by performing the following steps:

1. In `index.html`, add a new `div` tag inside the jumbotron `container` class and above the `h1` heading and paragraph. Give this new div tag a class of `welcome-message`:

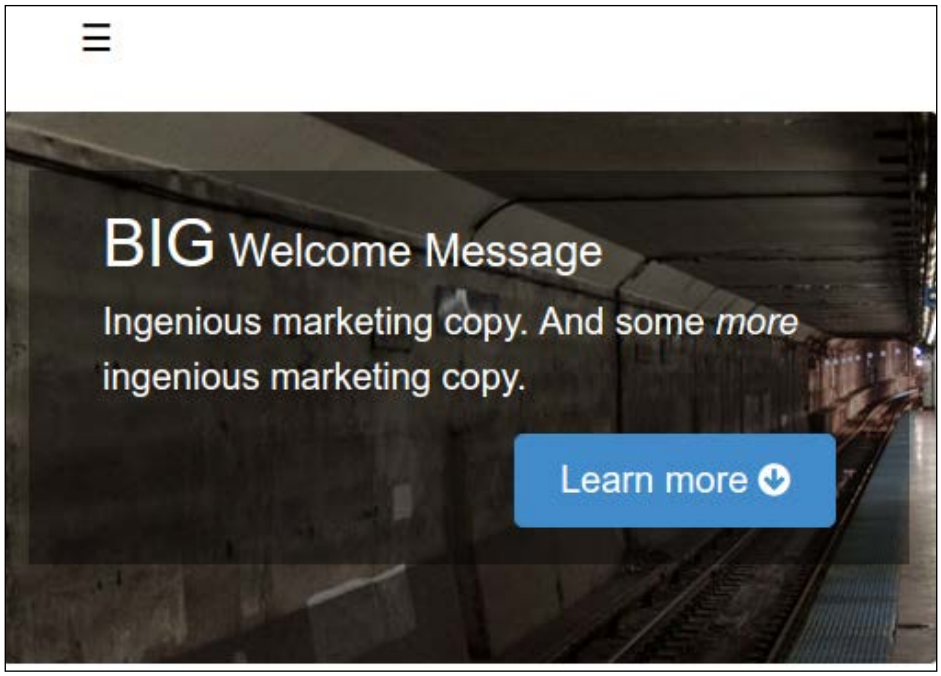
```
<section class="jumbotron">
```

```
    <div class="container">
      <div class="welcome-message">
        <h1 class="display-3"><strong>Big</strong>
        Welcome Message</h1>
        <p class="lead">
           Ingenious marketing copy. And some
           <em>more</em> ingenious marketing copy.<a
           href="#features" class="btn btn-lg btn-primary
           pull-right">Learn more <span class="icon fa
           fa-arrow-circle-down"></span></a>
        </p>
      </div>
    </div>
  </section>
```

2. Now, to create some styles for this new `div`, in the `scss/includes/_jumbotron.sccs` file we will perform the following steps:
    - Give it a translucent dark background using HSLA
    - Stretch it to fill the full width and height of our jumbotron by positioning it as `absolute` and setting its top, bottom, left, and right values to `0`
    - Position the jumbotron itself as `relative` using the `container` selector inside the jumbotron so that it will anchor our absolute-positioned welcome message
    - Add internal padding to the welcome message
    - Use the provided `strong` tag to transform the word `Big` to uppercase and increase its font size:

```
.jumbotron {  .container {    position: relative;    height:
100%;    .welcome-message {      background-color:
hsla(0,0,1%,0.4); // translucent overlay      position:
absolute;       top: 0;      left: 0;      right: 0;      @
include media-breakpoint-up(lg) {        right: 50%;
}      bottom: auto;      padding: 20px 40px;      strong {
font-size: 1.5em;        text-transform: uppercase;      }
@include media-breakpoint-down(sm) {        .display-3 {
font-size: 1.5em;        }       }      }  }}
```

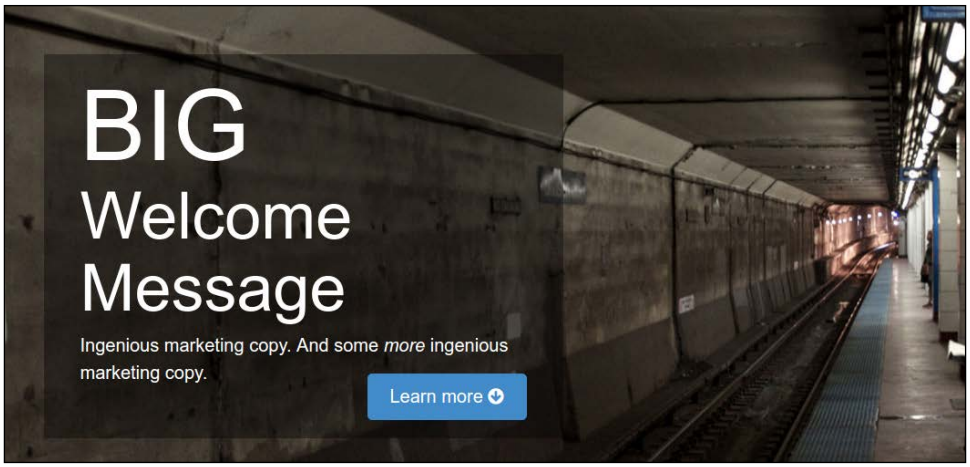
3. Save the file, run the `bootstrap watch` command, and inspect the results in your browser. You should see the background darken and the text stand out more clearly against it, as shown in the following screenshot:



4. Finally, let's address the medium and large viewports. We'll constrain the width a bit more. This can all be done under `with Sass` media query mixins again:

```
.jumbotron {
  .container {
   .welcome-message {
    right: 0;
     @include media-breakpoint-up(lg) {
       right: 50%;
    }
   }
  }
}
```

5. Again, save the file, and take a look in your browser. You should see the following result in a large viewport:



Mission accomplished!

Our customized jumbotron is finished, providing the large welcome message our client has asked for, including the ability to adapt to tablet- and phone-sized viewports, which we've accomplished efficiently with a mobile-first approach.

Now we're ready to move on to the features list.

Beautifying the features list

We need to enlarge the icons, align the text at the center, and iron out the grid layout. Let's review the markup structure for the features list:

```
<section id="features">
  <div class="container">
    <h1>Features</h1>
    <div class="row">
      <div class="features-item col-md-4">
        <span class="icon fa fa-cloud"></span>
        <h2>Feature 1</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce
dapibus, tellus ac cursus commodo. </p>
      </div>
    ...
```

Each feature with its icon, heading, and paragraph is wrapped in a `div` tag with two classes: `features-item` and `col-md-4`.

With this in mind, let's write the styles we need:

1. Create `scss/includes/_features.scss`, a new Sass partial, and do not forget to import it in the scss/app.scss file:

```
@import "includes/navbar";
@import "includes/jumbotron";
@import "includes/features";
```

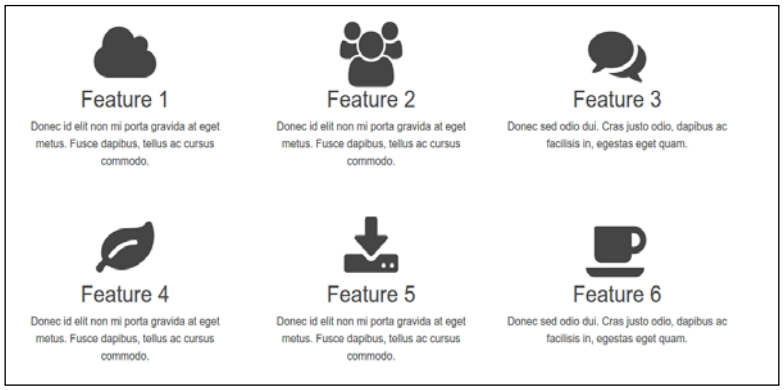
2. With `scss/includes/_features.scss` opened in your editor, add a new section with a comment for our #features section:

```
// Features Section
#features {

 }
```

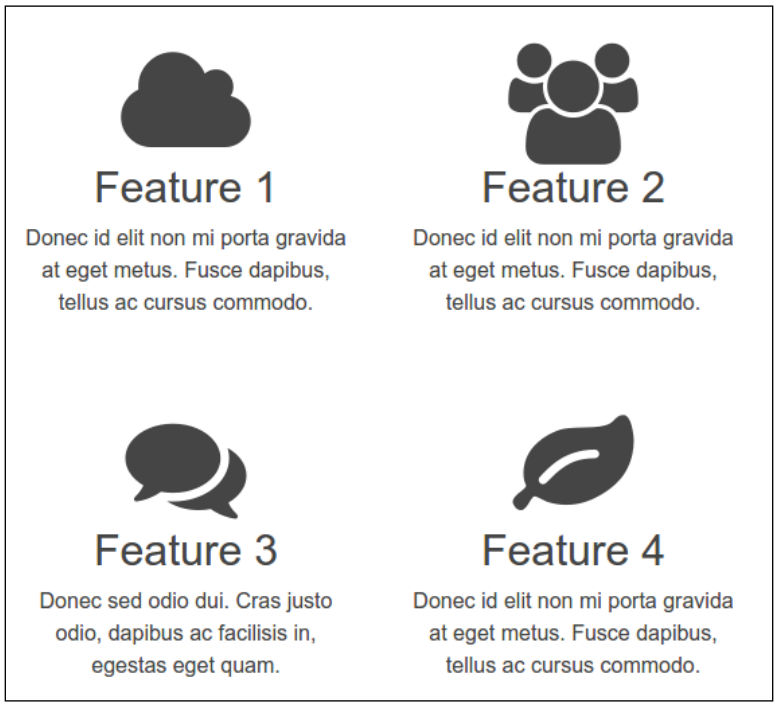
3. Now let's focus on the `.features-item` section by aligning the text at the center, adding padding, providing a set height to keep the floating items from interfering with each other, and increasing the `.icon` font size to `90px`:

```
#features {
  .features-item {
   text-align: center;
   padding: 20px;
   height: 270px;
   .icon {
      font-size: 90px;
   }
    }
}
```

4. Save the file, and test the results in your browser. Run the `bootstrap watch` command first, if you haven't already done so. You should see the following result in a medium viewport:



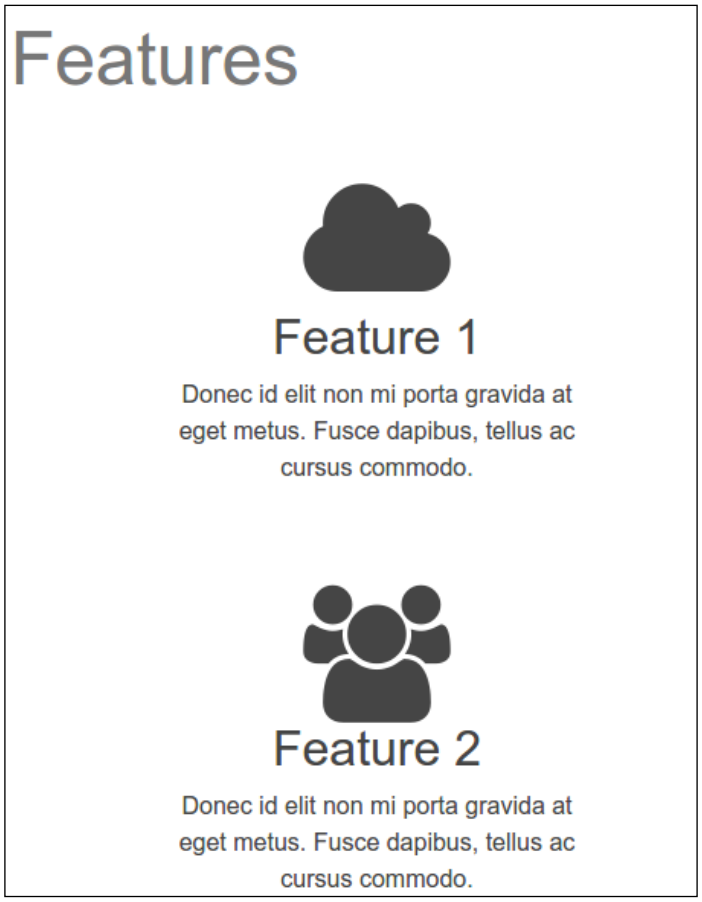
5. That's a great start! Now let's adapt our features section for small screens. Currently, our `.features-item` section includes a class of `col-md-4`. We can shift our small-screen layout to two columns, as shown in the following screenshot, by adding a class of `col-sm-6`:



6. And then, of course, they'll arrange themselves in a single column for extra-small screens.
7. Unfortunately, at the upper range of extra-small screens, 500px to 767px, the full-width layout allows the descriptive text to range too wide.
8. We can fix this by adding a media query within which we set a maximum width on the `.features-item` section and center the content by setting the horizontal margins to auto:

```
// Features Section
#features {
   @include media-breakpoint-only(xs) {
    margin: 0 auto;
    max-width: 320px;
    }
  }
}
```

9. Bootstrap also includes a `m-x-auto` class for horizontally centering fixed-width block level content. The `m-x-auto` class replaces the center-block class and mixin from Bootstrap 3.
10. With these lines in place, our `.features-item` elements retain their desired dimensions across all viewports! On small viewports, our features will look like this:



11. At this point, we have satisfied our client's demands for this section of her website! We're ready to move on to the customer reviews.

Tackling customer reviews

Our next section, named Impact, presents reviews from happy customers. In this section, we see smiling faces of happy customers with excerpts from their commentary about our client's product.

We'll use the **Card** module for this section again. The Card module is a flexible and extensible content container which replaces the panels, thumbnails, and wells used in earlier versions of Bootstrap.

The Card module is used to create the masonry grid layout. A masonry grid layout works by placing elements in optimal position based on available vertical space, sort of like a mason fitting stones in a wall. You'll create a Masonry grid layout for the Impact section again. The Bootstrap masonry solution uses CSS only. If you need a JavaScript solution which works in older browsers too, you can use a JavaScript masonry plugin available at `http://masonry.desandro.com`.

The Card columns use the CSS multi-column layout; you can read more about it at the following URL: `https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Columns/Using_multi-column_layouts`.

The masonry grid layout is not available in IE9 and below as they have no support for the column-* CSS properties.

So the initial markup starts as follows:

```
<!-- IMPACT SECTION -->
   <section id="impact">
   <div class="container">
         <h1>Impact</h1>
   <div class="reviews card-columns">
```

Each review is marked up as follows using the `hreview` microformat:



Microformats are an extension of HTML to mark up things such as people, organizations, products, and reviews. Sites using microformats publish a standard API, which can be consumed by search engines, browsers, and other tools. h-review is a simple, open format for publishing reviews on the Web. More information can be found at the following URL: `http://microformats.org/`.

```
            <div class="hreview review-item-1 card">
             <img class="card-img img-fluid" src="{{root}}images/
smiling1-by-RomainGuy-600x900.jpg" alt="Customer Photo1">
              <div class="caption card-img-overlay">
                <blockquote class="description card-img-overlayquote">
```

```
                    <p>Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Proin euismod, nulla pretium commodo ultricies</p>
                    <footer>Smiling Customer1</footer>
                  </blockquote>
                </div>
              </div>
```

Each card image gets the `img-fluid` class to make the image responsive and fit the cards.

The class turns an image into a card background and overlays the card's text by setting the position property of the image to absolute and the card's position property to relative.

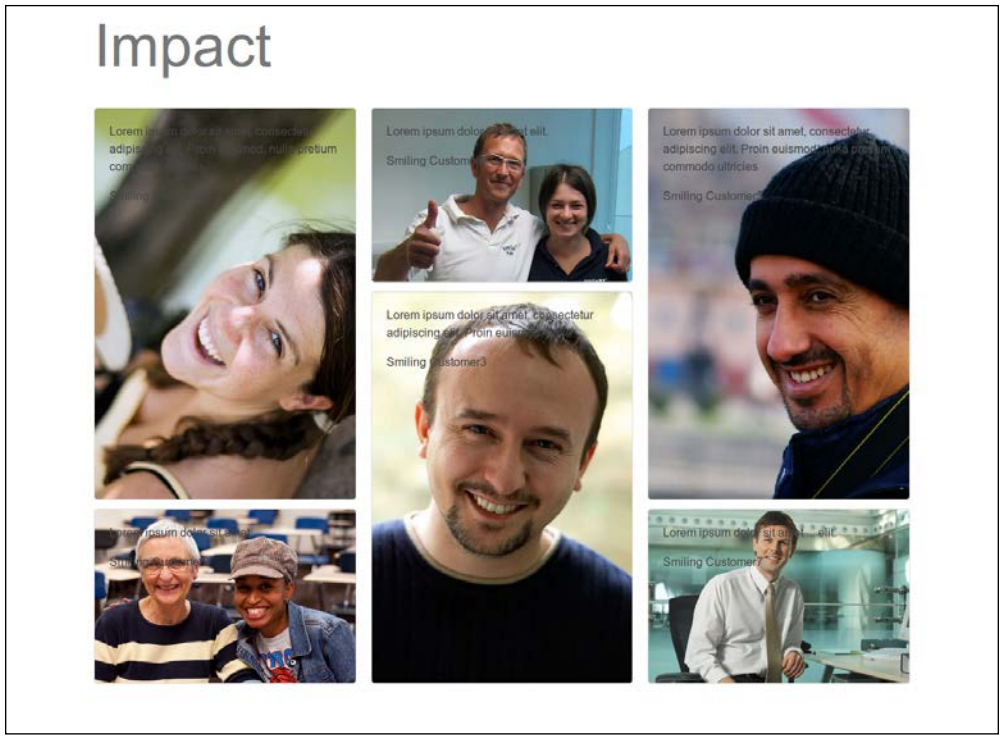
Each card (having the card class) will be automatically arranged in the grid due to the card-columns class of the selector.

The card-columns class creates CSS columns for the small grid and up by default. On the extra-small grid, the grid item will stack. Use the following SCSS code in the `scss/includes/_impact.scss` file for two columns on the small grid:

```
.card-columns {
  column-gap: $card-columns-sm-up-column-gap;
  @include media-breakpoint-up(sm) {
    column-count: 2;
  }
  @include media-breakpoint-up(md) {
    column-count: 3;
  }
  > .card {
    // see: https://github.com/twbs/bootstrap/pull/18255#issuecomme
nt-237034763
    display: block;
  }
}
```

You may read about the `hreview` microformat at `http://microformats.org/wiki/hreview-examples`.

Save your modifications and run the `bootstrap watch` command. You'll find that the **Impact** section should now look like the following screenshot:



Both in terms of semantics and presentational starting points, we're off to a good start.

Because we wanted to arrive at a masonry layout, our images are a mixture of portrait and landscape aspect ratios. We've made them all of equal width in order to provide enough room for bright faces and textual overlays with short laudatory statements.

Before addressing the layout for larger viewports, let's start by tackling captions.

Positioning and styling captions

Let's begin by positioning our captions as overlays on top of their respective customer photos:

1. While editing the `scss/includes/_impact.scss` file, add a new comment and selector for the `#impact` section:

```
// Impact Section
#impact {
```

```
}
```

2. Now, we can work on the captions. We'll add a translucent background and position them as absolute at the bottom of each image:

```
.hreview {
  .caption {
    position: absolute;
     top: auto;
     left: 10px;
     right: 10px;
    bottom: 0;
    line-height: 1.1;
    background: hsla(0,0,10%,0.55);
  }
```

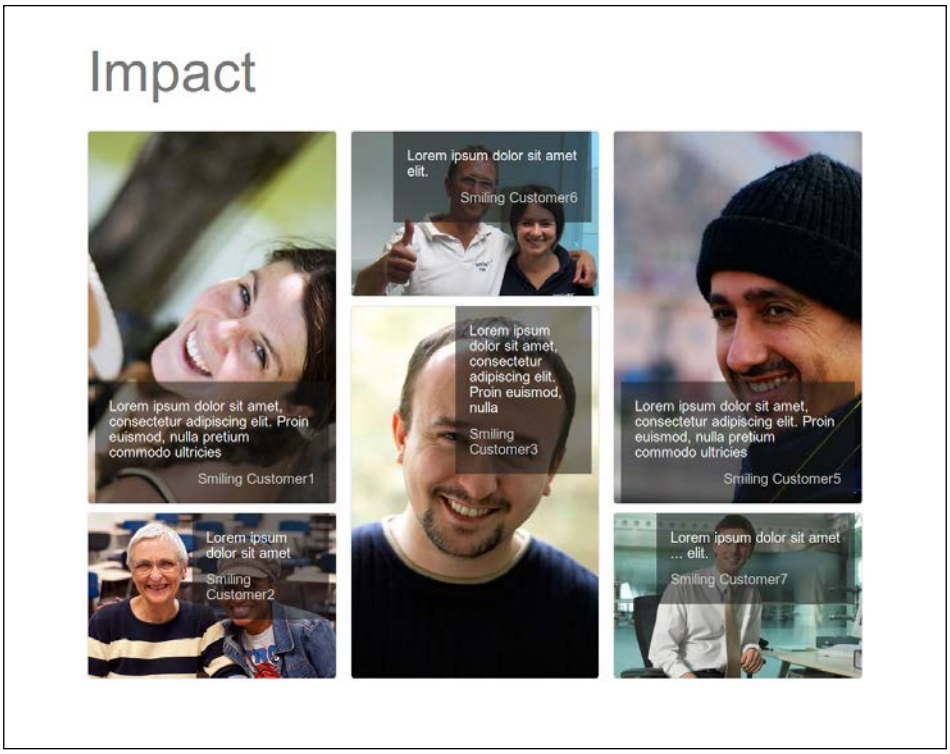
3. Now we can focus on the review text and specify the margin, border, font family, font size, and color:

```
blockquote {
  margin-top: 4px;
  border: none;
   font-family: @font-family-serif;
   font-size: @font-size-large;
     color: #fff;
}
```

4. Next, specify styles for the reviewer's name, which appears below the review text:

```
.reviewer {
   margin-top: 2px;
   margin-bottom: 4px;
   text-align: right;
  color: $gray-lighter;
 }
```

5. Save the file, run the `s` command, and check your progress.
6. You should end up with an **Impact** section like that shown in the following screenshot:



Not bad! However, we can do it one step better.

Refining the caption position

Looking carefully at the available open space in the preceding screenshots and examining the overlap variations at various viewport widths in your responsive grid, you may want to position each caption in a way that works best for each customer's photo.

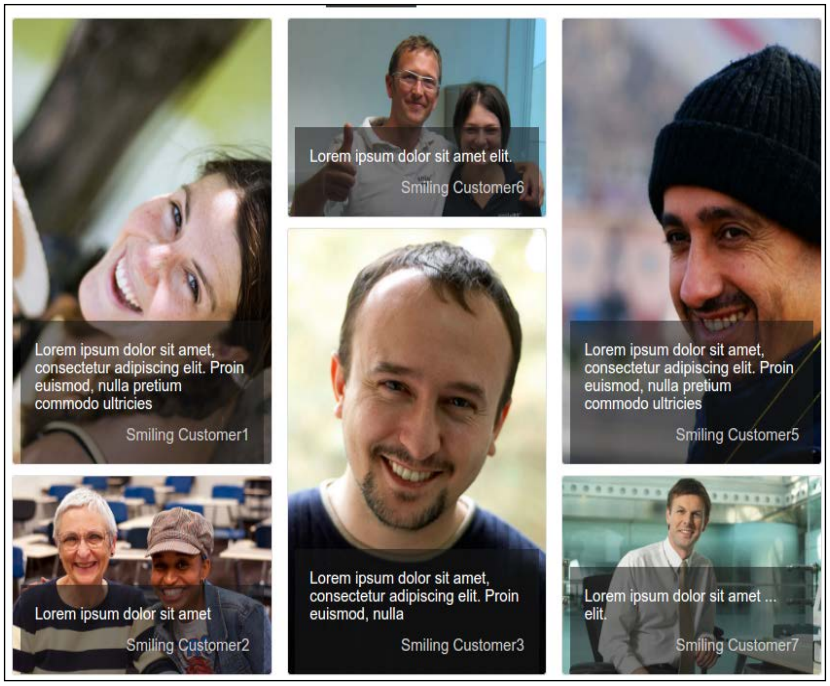
This is where `the review-item-1`, `review-item-2`, and so on classes become relevant and helpful, as we may use these specific classes to position each caption in a way that fits best with its image.

To demonstrate the positions of the caption, I've added the following lines in the `scss/includes/_impact.scss` file:

```
.hreview:nth-child(2n) .caption {
  top: 0;
  left: 62%;
  right: 10px;
```

```
    bottom: auto;
    .reviewer {
      margin-top: 6px;
      text-align: left;
    }
  }
  .hreview:nth-child(3n) .caption {
    top: 0;
    left: 17%;
    right: 10px;
    bottom: auto;
  }
```

The preceding markup adjusts the absolute positioning of each second and third caption, which yields the results shown in the following screenshot:



Instead of using the `:nth-child()` selectors in the above, you can also write your own SCSS to position each specific caption.

Adjusting for tiny screens

On the extra-small grid, the reviews are stacked, and on the small grid we'll arrange them in two columns.

First, we'll reduce the font size of the captions for the small grid. To reduce the font size, insert the following SCSS code into the `scss/includes/_impact.scss` file:

```
#impact {
  .caption {
    blockquote {
      font-size: $font-size-sm;
      @include media-breakpoint-only(sm) {
        font-size: $font-size-lg;
      }
    }
  }
}
```

On the small and extra-small grid, we'll only show the first four reviews. Hide the last reviews by default by using the following SCSS code:

```
// Impact Section
#impact {
 .hreview:nth-child(5), .hreview:nth-child(6) {
   display: none;
   @include media-breakpoint-only(md) {
     display: block;
   }
 }
}
```

Save the file and then test the results in your browser.

Voilà! The customer reviews are now performing entirely in line with our client's desires.

Now to take care of the last major item in our client's desired home page design: the pricing tables.

Creating attention-grabbing pricing tables

Let's revisit the mockup of how our client would like the pricing tables to look on desktop-sized screens:



Let's see how close we can get to the desired result, and what we can work out for other viewport sizes.

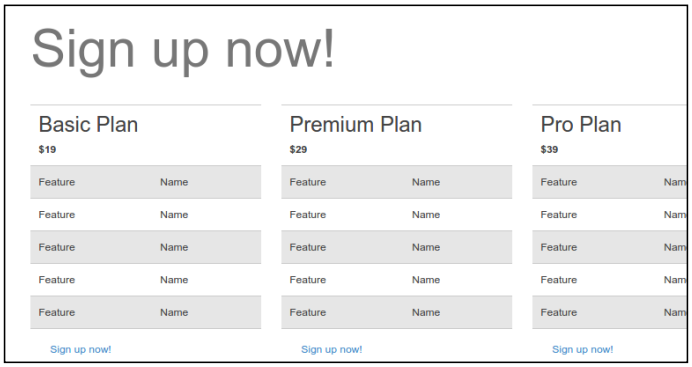
Setting up the variables, files, and markup

As shown in the preceding screenshot, there are a few tables in this design. We can begin by adjusting a few fundamental variables for all tables. These are found in Bootstrap's `_variables.scss` file. Search for the tables section and adjust the variables for background, accented rows, and borders as desired. I've made these adjustments as shown in the following lines of code, and saved them in the local `scss/includes/_variables.scss` file:

```
// Tables
//
// Customizes the `.table` component with basic values, each used
across all table variations.

$table-cell-padding:            .75rem;
$table-sm-cell-padding:         .3rem;

$table-bg:                      transparent;
$table-bg-accent:              hsla(0,0,1%,.1); // for striping
$table-bg-hover:                hsla(0,0,1%,.2);
$table-bg-active:               $table-bg-hover;

$table-border-width:           1px;
```

```
$table-border-color:            $gray-lighter;
```

Save the file, and run the `bootstrap watch` command to see the result as shown in the following screenshot:



That's a start. Now we need to write more specific styles.

To carry the custom styles, let's create a new Sass file for these pricing tables:

1. Create `_pricing-tables.scss` in the main `scss/includes` folder.
2. Import it into `main.scss` just after `the import of the _impact.scss` file as shown in the following line:

   ```
   @import "_pricing-tables";
   ```
3. Open `_pricing-tables.less` in your editor and begin writing your new styles.

But before we begin writing styles, let's review the markup that we'll be working with.

We have the following special classes already provided in the markup on the parent element of each respective table:

- package package-basic
- package package-premium
- package package-pro

Thus, for the first table, you'll see the following markup on its parent `div`:

```
<div class="package package-basic col-lg-4">
   <table class="table table-striped">
...
```

Notice that the `table` and `table-striped` classes in the above HTML code are part

of Bootstrap's styles to display content. You can use these styles by just adding the table base class to any `<table>` element, extend the base class with custom styles or by including a modifier class such as the `table-striped` class. Read more about tables in Bootstrap at `http://getbootstrap.com/content/tables/http://v4-alpha.getbootstrap.com/content/tables/`.

Similarly, we'll use `package package-premium` and `package package-pro` for the second and third tables, respectively.

These parent containers obviously also provide basic layout instructions using `the col-md-4` class to set up a three-column layout in medium viewports and up.

Next, we will observe the markup for each table. We see that the basic `table` and `table-striped` classes have been applied:

```
<table class="table table-striped">
```

The table uses the `<thead>` element for its topmost block. Within this, there is `<th>` spanning two columns, with an `<h2>` heading for the package name and `<div class="price">` to mark up the dollar amount:

```
<thead>
  <tr>
    <th colspan="2">
      <h2>Basic Plan</h2>
      <div class="price">$19</div>
    </th>
  </tr>
</thead>
```

Next is the `tfoot` tag with the **Sign up now!** button:

```
<tfoot>
  <tr><td colspan="2"><a href="#" class="btn">Sign up        now!</
a></td></tr>
  </tfoot>
```

Then there is the `tbody` tag with the list of features laid out in a straightforward manner in rows with two columns:

```
<tbody>
  <tr><td>Feature</td><td>Name</td></tr>
  <tr><td>Feature</td><td>Name</td></tr>
  <tr><td>Feature</td><td>Name</td></tr>
  <tr><td>Feature</td><td>Name</td></tr>
  <tr><td>Feature</td><td>Name</td></tr>
</tbody>
```

And finally, of course, the closing tags for the `table` and parent `s` tags:

```
    </table>
  </div><!-- /.package .package-basic -->
```

Each table repeats this basic structure. This gives us what we need to start work!

Beautifying the table head

To beautify the `thead` element of all of our tables, we'll do the following:

- Align the text at the center
- Add a background color; for now, add a gray color that is approximately a midtone similar to the colors we'll apply to the final version
- Turn the font color white
- Convert the `h2` heading to uppercase
- Increase the size of the price table
- Add the necessary padding all around the tables

We can apply many of these touches with the following lines of SCSS code. We'll specify the `#signup` section as the context for these special table styles:

```
#signup {
  table {
    border: 1px solid $table-border-color;
    thead th {
      text-align: center;
      background-color: $gray-light;
      color: #fff;
      padding: 2 * $spacer-y 0;

      h2 {
        text-transform: uppercase;
        font-size: 2em;
      }
    }
  }
}
```

In short, we've accomplished everything except increasing the size of the price tables. We can get started on this by adding the following lines of code, which are still nested within our `#signup table` selector:

```
.price {
      font-size: 4em;
      line-height: 1;
}
```

This yields the following result:



This is close to our desired result, but we need to decrease the size of the dollar sign. We can nest the first letter within our styles for `.price`:

```
.price {
  font-size: 4em;
  line-height: 1;
  &::first-letter {
    font-size: .5em;
    vertical-align: super;
  }
}
```

`::first-letter` is a pseudo element, which allows you to style the first letter in an element without needing to stick a `<span>` tag around that first letter in your HTML. You can read more about this pseudo element at `https://css-tricks.com/almanac/selectors/f/first-letter/`.

These lines reduce the dollar sign to half its size and align it at the top. The following screenshot shows the result:



Styling the table body and foot

Continuing to focus on the styles that apply to all three pricing tables, let's make the following adjustments:

- Add left and right padding to the list of features
- Stretch the button to full width
- Increase the button size

We can accomplish this by adding the following rules:

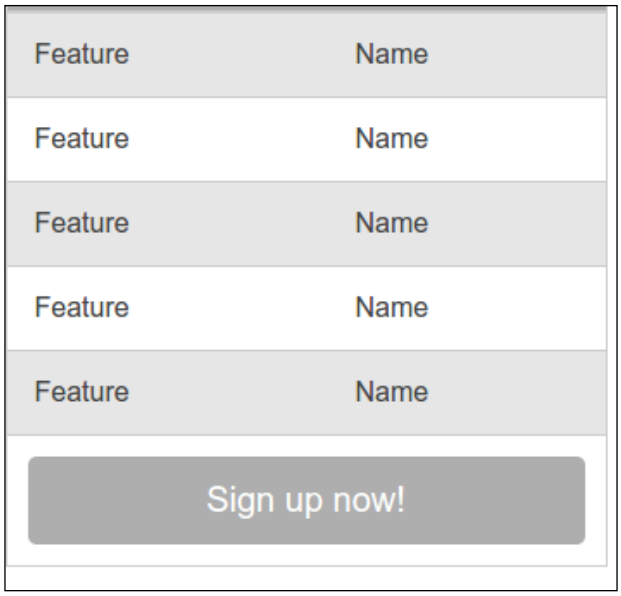
```
#signup {
  table {
    tbody {
      td {
        padding-left: $spacer-x;
        padding-right: $spacer-x;
      }
    }
    a.btn {
      @extend .btn-lg;
      font-size: 1.25em;
      display: block;
      width: 100%;
      background-color: $gray-light;
      color: #fff;
    }
  }
}
```

In the preceding SCSS code, the `@extend` feature of Sass has been used to extend the button with Bootstrap's styles for large buttons. Bootstrap itself avoids the `@extend` feature, but you can use it.



Alternatively, you can use Bootsstrap's `button-size()` mixin to set the large button styles. Notice that I have set font-size: 1.25em; afterward. The button mixin sets the font size in rem units and we want the font size to scale with its parent.

Save the file, run the `bootstrap watch` command, and you should see the following result:



We're now ready to add styles to differentiate our three packages.

Differentiating the packages

Let's begin by giving each package the desired color for the table head and the **Sign up now!** button. Our provided mockup uses blue for the **Basic**, green for the **Premium**, and red for the **Pro** packages. Let's prepare our color scheme by using the chosen color values in new variables for primary, secondary, and tertiary brand colors, as shown in the following lines of code:

```
$brand-primary:             #428bca;
$brand-secondary:           #5cb85c;
$brand-tertiary:            #d9534f;
```

Having set up these colors, we can efficiently apply them to the appropriate `thead` and `button` elements. We'll use the distinctive class that we applied earlier to each table's parent element, that is, `package-basic`, `package-premium`, and `package-pro`:

1. In the `scss/includes/_pricing-tables.scss` file, begin a new section with a comment:

```
// Pricing Table Colors
```

2. We'll apply the primary brand color to the `.package-basic` table using the `$brand-primary` variable; we'll try it first on the `thead th` element:

```
#signup .package-basic table {
 thead th {
    background-color: $brand-primary;
 }
```

3. Then, apply the primary brand color to the `thead th` element's button. Here, we'll use the `.button-variant()` mixin from the `bootstrap/mixins.less` file to efficiently apply styles to the `:hover` and `:active` states. The mixin takes three parameters: color, background color, and border color. We'll define them as follows:

```
...

.btn {

  @include button-variant(#fff, $brand-primary,
  darken($brand-primary, 5%));

 }

}
```

4. When compiled, this concise mixin will generate styles for the button and its hover and active states!



For a reminder of how the `button-variant()` mixin works, consult the `bootstrap/scss/mixins/_buttons.scss` file, where the mixin is defined, and then `bootstrap/scss/_buttons.scss`, where it is used to define the default Bootstrap button classes.

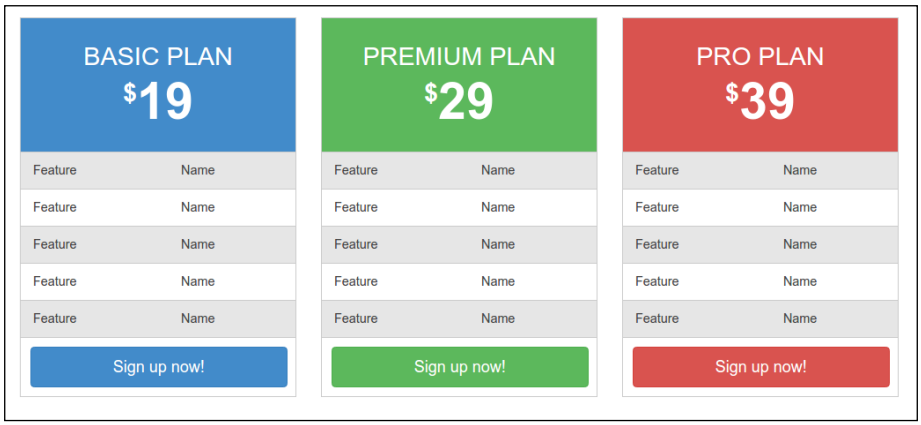
5. Now we need to repeat this for our `.package-premium` table; this time, however, use the `$brand-secondary` variable:

```
#signup .package-premium table {

 thead th {

   background-color: $brand-secondary;

  }

 .btn {

    @include button-variant(#fff, $brand-secondary,
darken($brand-secondary,

    5%));

  }

}
```

6. Finally, we'll apply the tertiary brand color to the `.package-pro` table using `the $brand-tertiary` variable:

```
#signup .package-pro table {
 thead th {
    background-color: $brand-tertiary;
  }
 .btn {
   @include button-variant(#fff, $brand-tertiary,
darken($brand-tertiary,
   5%));
 }
}
```

7. You might have noticed that the preceding steps and code are very repetitive. Sass can help you to code your CSS code **DRY** (**Do not Repeat Yourself**). By wrapping the names in a Sass map and using an `@each` loop, you'll have to write the code only once.
8. Save the file, and run the `bootstrap watch` command if you have not already run it. You should see the new colors we applied to our tables:



Nice!

Now let's check how our tables respond to various viewport widths.

Adjusting for small viewports

Thanks to the attention Bootstrap gives to responsive design, our tables perform quite well across viewport breakpoints. We've already seen how our tables fare in the medium breakpoint range. In large screens, the tables expand wider. In narrow viewports, the tables stack up vertically, quite nicely.

However, there is an awkward range of width approximately between 480px and 768px, where the tables expand to fill the full width of the screen. Clearly, they become too wide.

Because we have three tables, there is no benefit involved in having a two-column layout at this dimension. Instead, let's constrain the width of our tables and align them at the center with `auto` left and right margins. We'll use the `media-breakpoint-down()` media query mixin to set `400px` as our maximum width, and set the horizontal margins to auto to keep our tables at the center of the window:

```
//
// Constrain width for small screens and under
// -------------------------------------------

@include media-breakpoint-down(sm) {
  #signup .package {
      max-width: 400px;
      margin: 0 auto;
    }
}
```

You cannot center the tables by using the `@extend` feature of Sass to extend the `.m-x-auto` class, because you may not `@extend` an outer selector from within `@media`.

On the medium grid, the tables are too wide for the grid columns, so we reduce the total font size for only the medium grid by using the following SCSS code:

```
#signup {
  font-size: 100%;
  @include media-breakpoint-only(md) {
    font-size: 70%;
  }
}
```

Save the file, and test the results in your browser. You should see nicely constrained tables aligned at the center within the window!

At this point, our tables are differentiated by color and are responsive. However, one last step remains. In the medium and large viewport widths, we want the premium plan to stand out.

Providing a visual hierarchy to our tables

If we look back at the mockup, we see that the design - at least for desktop-sized viewports-calls for visual emphasis on the central Premium plan by increasing its size and bringing it visually into the foreground.

This can be accomplished with some adjustments to padding, margins, and font sizes. We'll do this within a media query for medium viewports and up:

```
//
// Visually enhance the premium plan
// ----------------------------------------
@include media-breakpoint-up(md) {

}
```

Our first aim is to bring the tables closer to one another. This can be done by removing the padding (the gutter of the grid) between the grid columns:

```
#signup {
  // Squeeze tables together
  .col-md-4 {
    padding: 0;
  }
}
```

Then we can enlarge the font size for the price information in the premium block as follows:

```
#signup {
  .package-premium .price {
    font-size: 7em;
  }
}
```

Nested within this media query, we can first reduce the widths of our basic and pro tables (the first and third) and add a little margin to the top to push them down a bit:

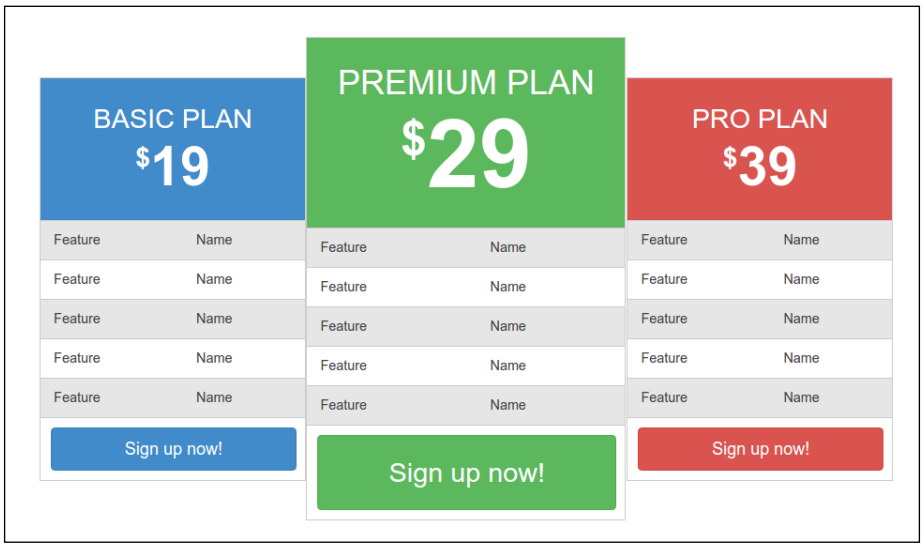
```
  // Size down the basic and pro
  #signup .package-basic {
    padding-left: 4 * $spacer-y;
  }
  #signup .package-pro {
    padding-right: 4 * $spacer-y;
```

```
}
#signup .package-basic table,
#signup .package-pro table {
  margin-top: 3 * $spacer-x;
}
```

Next, let's enhance the font size of our premium table and add padding to its button:

```
// Size up the premium
#signup .package-premium table {
  thead th {
    h2 {
      font-size: 2.5em;
    }
  }
  a.btn {
    font-size: 2em;
    padding-top: 1.5 * $spacer-x;
    padding-bottom: 1.5 * $spacer-x;
  }
}
```

Save the file, and inspect the results in the browser. You should see the following result in large viewports of 1,200px and above:



That's it! We've accomplished the last major challenge in our client's design. Now let's tidy things up by applying those little touches that hold it all together.

Adding the final touches

In this section, we will enhance the details that hold our design together. First, we'll enhance the `h1` headings for each of our major sections and add some needed top and bottom padding to each section. Then, we'll enhance the navigation experience by adding ScrollSpy to the navbar and using jQuery to animate the scrolling action when triggered by a click on the navbar item.

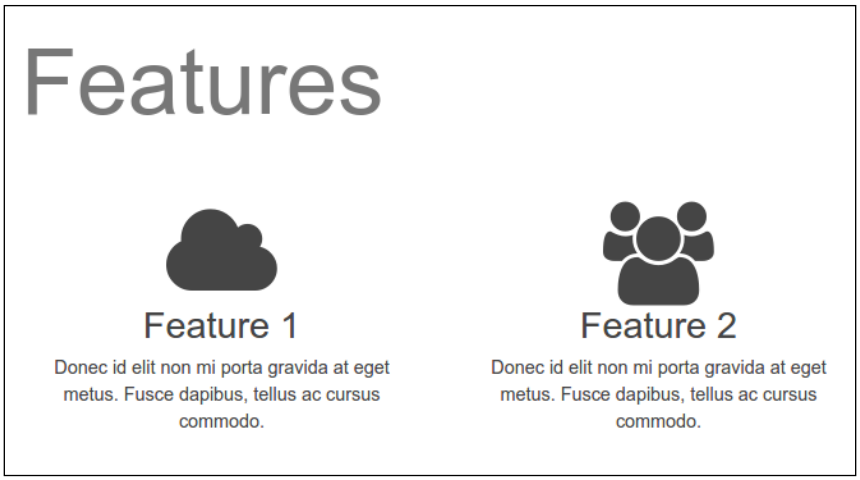
Let's begin by enhancing the size and contrast of our major `h1` headings for each section and increasing the top and bottom padding. If you pause to look at these `h1` headings, you may note that they are rather lackluster.

Enlarging these headings, bringing the contrast down a little, and providing extra padding will make a big difference. We only want these rules to apply to the **Features**, **Impact**, and **Sign up** sections. We will select these by ID:

1. Open the `scss/includes/_page-contents.scss` file again in your editor.
2. At the top of the file, after the rule applying top padding to the body, add the following lines:

```
#features, #impact, #signup {
  padding-top: $spacer-y * 2.5;
  padding-bottom: $spacer-y * 3;
  h1 {
    font-size: 5em;
    color: $gray;
    line-height: 1.3;
    padding-bottom: $spacer-y * 1.5;
 }
}
```

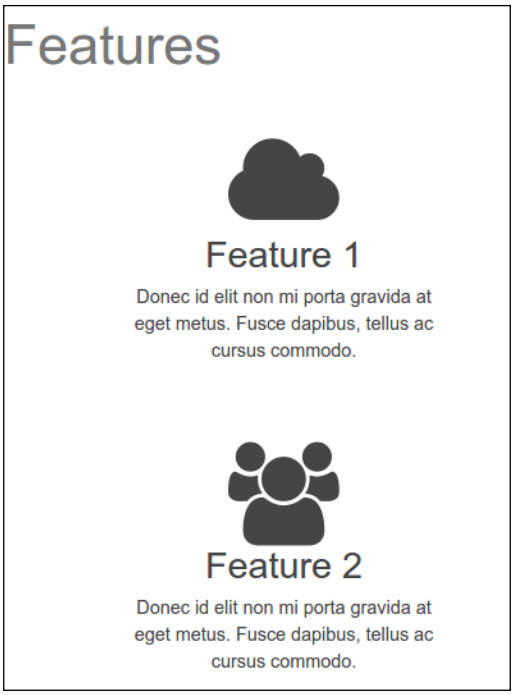
3. Here, we've done the following:
    - Added top and bottom padding to these sections
    - Significantly increased the size of the h1 heading
    - Reduced the heavy contrast of that heading
    - Ensured that the heading has room to breathe by setting the line height and bottom padding
4. Save your work and notice the difference in your browser:



This yields a nice result across almost all viewport sizes. For small viewports, the `h1` font size is now a bit large. So, let's adjust this. As we do not want these styles to flow up to larger viewports, we'll wrap the styles we have already written in a query by limiting them to larger viewports. Finally, the refactored and mobile-first SCSS code should look like this:

```
#features, #impact, #signup {
  padding-top: $spacer-y * 1.5;
  padding-bottom: $spacer-y * 1;
  h1 {
    font-size: 3em;
    color: $gray;
    line-height: 1.3;
    padding-bottom: $spacer-y;
  }

  @include media-breakpoint-up(md) {
    padding-top: $spacer-y * 2.5;
    padding-bottom: $spacer-y * 3;
    h1 {
      font-size: 5em;
      padding-bottom: $spacer-y * 1.5;
    }
  }
}
```

The following screenshot shows our result:



This is a much improved result! Now we'll enhance the navigation experience.

Adding ScrollSpy to the navbar

Let's configure our top navbar to indicate our location on the page. We'll add Bootstrap's ScrollSpy behavior to the navbar.



Refer to Bootstrap's ScrollSpy plugin documentation at `http://getbootstrap.com/javascript/#scrollspy`.

By default, the ScrollSpy plugin requires a Bootstrap `nav` component. Bootstrap's navbar contains a nav component already. The relative position is required too. You should set `position:relative;` in your CSS for the element you're spying on. In our situation, we'll have to set the relative position for the body element.

You can easily initiate the ScrollSpy plugin by adding data-attributes in the HTML code. First add `data-spy="scroll"` to the element you want to spy and then add the data-target attribute with the ID or class of the parent element of any Bootstrap `.nav` component.



Data-attributes in HTML5 allow use to store extra information into standard semantic HTML elements. Read more about data-attributes in HTML5 at the following URL: `https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes`.

ScrollSpy requires a resolvable ID target in our HTML code. We've already added the target in the preceding steps. For instance, the Intro section had `id="welcome"` in the section element as follows:

```
<section class="jumbotron" id="welcome">
```

The above HTML code can be found in the `html/includes/intro.html` file. The `id="welcome"` declaration corresponds with the resolvable id targets in the navbar links, as can be seen in the `html/includes/page-header.html` file. The HTML code of the `Welcome` should look as follows now:

```
<a class="nav-link active" href="#welcome">Welcome <span
class="sr-only">(current)</span></a>
```

Now perform the following steps to initiate the ScrollSpy plugin in our project:

1. Edit the `scss/app.scss` file to set the relative position of the body element. Add the following SCSS code at the end of this file:

```
body {
  position: relative;
}
```

2. Then open `index.html` in your editor.
3. Add these ScrollSpy data-attributes to the `body` tag:

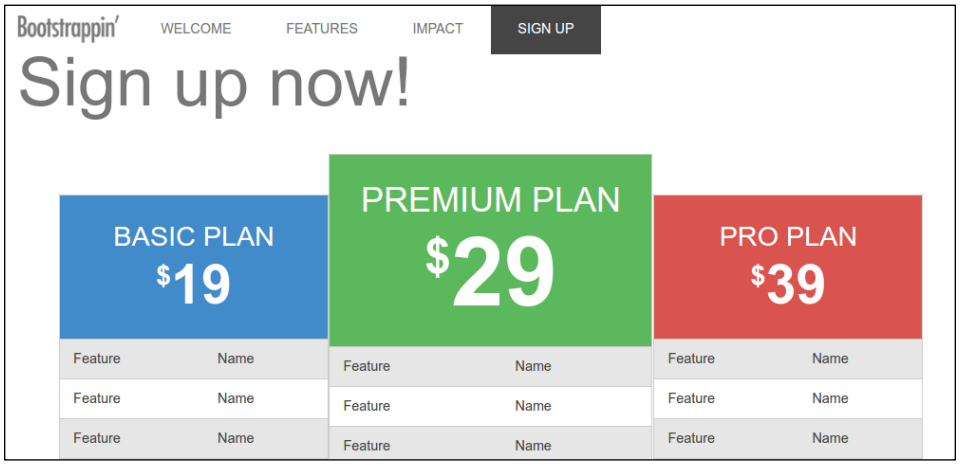
```
<body data-spy="scroll" data-target=".navbar">
```

4. Edit the file and set the resolvable ID targets in the navbar links. At the end, your HTML code should look like this:

```
<ul class="nav navbar-nav">
  <li class="nav-item">
    <a class="nav-link active" href="#welcome">Welcome <span
        class="sr-only">(current)</span></a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#features">Features</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="#impact">Impact</a>
  </li>
  <li class="nav-item">
```

```
            <a class="nav-link" href="#signup">Sign up</a>
          </li>
        </ul>
```

With the resolvable id targets and the new data-attributes in place, save the file, refresh your browser, and scroll up and down the page. You should see your main navigation respond as it should, indicating your position on the page as shown in the following screenshot:



In the preceding step, we used data-attributes to initiate Bootstrap ScrollSpy behavior. You can also initiate the plugin via JavaScript by performing the following steps:

5. First add the `position: relative;` declaration for the body element in your CSS/SCSS:

```
body {
  position: relative;
}
```

6. Then call the ScrollSpy via JavaScript/jQuery as follows:

```
$('body').scrollspy({ target: '.navbar' })
```

Animating the scroll

Now let's animate the page scrolls that will be triggered by clicking on the navbar page anchors. We'll use jQuery to accomplish this.



jQuery is a JavaScript library and provides you with an API for HTML document traversal and manipulation, event handling, and animations. The `jQuery animate()` API call lets you create custom animations of CSS properties. You can read more about jQuery's animations at the following URL: `http://api.jquery.com/animate/`.

Animating the page scroll requires adding a few lines to our `main.js` file:

1. Open `js/main.js`.
2. Add the following lines within `$(document).ready(function() {`:

```
$('#nav-main [href^=#]').click(function (e) {
  e.preventDefault();
  var div = $(this).attr('href');
  $("html, body").animate({
  scrollTop: $(div).position().top
  }, "slow");
});
```

3. Save the file and refresh your browser.

What have we done here? We have done the following using the power of jQuery:

- Selected the links in our `.navbar` element that use page anchors as their targets and set a click event on them:

```
$('#nav-main [href^=#]').click(function (e) {}
```

- Prevented the default click behavior as follows:

```
e.preventDefault();
```

- Animated the scrolling behavior, setting its duration to slow as shown in the following snippet:

```
$("html, body").animate({
    scrollTop: $(div).position().top
}, "slow");
```

Click on one of the nav items and you should see it animate the scroll!

Summary

Take a moment to scroll back and forth through our page, appreciating its details and resizing it to see how it adjusts to viewport dimensions.

When we consider the variety of features packed into this page - and that they all work responsively across desktop-, tablet-, and phone-sized viewports it's not a bad accomplishment!

To recap, we have given our client a beautiful, one-page marketing site with a large welcome section using Bootstrap's jumbotron styles, a bold background image, and responsive customizations, a features list making use of large-sized Font Awesome icons, a section of customer reviews with images and captions laid out in the masonry format, which adapts beautifully across viewports. We ended up with a signup section with custom-designed pricing tables built on Bootstrap styles and enhanced further to provide a visual hierarchy for medium and large viewports. As a finishing touch, we added a ScrollSpy-equipped navbar with animated scrolling behavior provided by a bit of extra jQuery. With this design, we have reached a point where there is nothing we can't do with Bootstrap.

Across this and previous projects, we have accomplished a great deal. We have learned the ins and outs of Bootstrap, and folded Bootstrap Sass and JavaScript into our own custom set of project files. We then used the robust Font Awesome icons. We also tweaked, customized, and otherwise innovated on Bootstrap styles to arrive the exact results we were seeking.

With this, we've come to the end of this learning journey. I hope you'd a smooth journey and gained a lot of knowledge on Bootstrap.

I wish you all the best for your future projects. Keep learning and exploring!

Assessments

1. Which of the following command is used to install Bootstrap CLI?
    i. `npm install -i bootstrap-cli`
    ii. `npm install -g bootstrap-cli`
    iii. `npm install -d bootstrap-cli`
    iv. `npm install -w bootstrap-cli`

2. Which of the following code is used to copy assets from source folder to destination folder?
    i.
    ```
    gulp.task('copy', function() {
    gulp.src(['assets/*/**']).pipe(gulp.dest('_site'));
     });
    ```
    ii.
    ```
    gulp.task('copy', function() {
    gulp.src('assets/**/*').pipe(gulp.dest('_site'));
     });
    ```
    iii.
    ```
    gulp.task('copy', function() {
    gulp.src('assets/*/**').pipe(gulp.dest('_site'));
     });
    ```
    iv.
    ```
    gulp.task('copy', function() {
    gulp.src(['assets/**/*']).pipe(gulp.dest('_site'));
     });
    ```

3. Which of the following command is used to load Font Awesome's CSS code from CDN?
    i.
    ```
    <link rel="stylesheet" href="https://maxcdn.
    bootstrapcdn.com/font-awesome/4.6.1/css/font-awesome.
    min.css">
    ```
    ii.
    ```
    <a href="https://maxcdn.bootstrapcdn.com/font-
    awesome/4.6.1/css/font-awesome.min.css">
    ```
    iii.
    ```
    <link  href="https://maxcdn.bootstrapcdn.com/font-
    awesome/4.6.1/css/font-awesome.min.css">
    ```
    iv.
    ```
    <a rel="stylesheet" href="https://maxcdn.
    bootstrapcdn.com/font-awesome/4.6.1/css/font-awesome.
    min.css">
    ```

4. Which of the Bootstrap's navbar classes determines the placement of the navbar?
    i. `navbar-fixed-bottom`
    ii. `navbar-fixed-left`
    iii. `navbar-fixed-top`
    iv. `navbar-fixed-right`

5. Which of the following uses the built-in lightness function of Sass to return a light (white) or dark (black) color depending on the lightness of the input color?
    i. `contrast()`
    ii. `lighten()`
    iii. `darken()`
    iv. `saturate()`

Assessment Answers

Lesson 1: Introducing Bootstrap 4

Question Number	Answer
1	iii
2	ii
3	i
4	iii
5	i
6	iii
7	iii
8	iii
9	iii
10	ii
11	ii

Lesson 2: Jumping into Flexbox

Question Number	Answer
1	ii
2	i
3	i
4	iii
5	ii
6	iv

Lesson 3: Working with Layouts

Question Number	Answer
1	iii
2	i
3	ii
4	iv
5	iv

Lesson 4: Working with Content

Question Number	Answer
1	i
2	iii
3	ii
4	iv
5	i

Lesson 5: Playing with Components

Question Number	Answer
1	iii
2	i
3	ii
4	iv
5	ii
6	i
7	ii
8	i
9	ii
10	iii
11	ii
12	i
13	i
14	ii

15	iii
16	i
17	iv
18	iii

Lesson 6: Extending Bootstrap with JavaScript Plugins

Question Number	Answer
1	iv
2	ii
3	i
4	iv
5	ii

Lesson 7: Throwing in Some Sass

Question Number	Answer
1	iii
2	ii
3	iv
4	i
5	ii

Lesson 8: Bootstrapping Your Portfolio

Question Number	Answer
1	i
2	ii
3	i
4	ii
5	iii

Lesson 9: Bootstrapping Business

Question Number	Answer
1	iv
2	i
3	ii
4	iii
5	iv

Lesson 10: Bootstrapping E-Commerce

Question Number	Answer
1	ii
2	iv
3	i
4	iii
5	i

Lesson 11: Bootstrapping a One-Page Marketing Website

Question Number	Answer
1	ii
2	iv
3	i
4	iii
5	i