Contents

Preface

In this book we'll be covering rapid prototyping from the perspective of the front-end developer (although designers, managers, etc., will probably get a lot out of the content too). The approach will favor using a tool-based approach and be aimed more towards the developer with minimal design ability. We are not advocating getting rid of old approaches, but simply adding a newer and faster one to our repetoire.

The pace of the book will be quite fast as we'll favor brevity so we can dig deeper in to "doing" and cover more materials. Therefore, we'll be making heavy use of web links to places to get more detailed coverage on topics we might seem to breeze past.

Programming Language and Syntaxes

Most of the examples will be in some type of language or syntax familiar to front-end developers. In addition we will learn some tool-specific syntaxes:

- HTML, CSS, JavaScript, etc.
- Compass and Sass (the .scss version) and possibly a bit of LESS
- Tool specific syntaxes (e.g. Grunt, Bower, Yeoman, etc.)
- Command line

Assumptions

This book assumes experience with web technologies like HTML, CSS, and JavaScript.

Resources

I've generally listed any resources as clickable web links that you can learn more from, or, as links to where you might purchase the book that I'm referencing.

Line breaks in code

I've taken the liberty of purposely wrapping long lines that won't fit within the width of the page.

Contributions

I am definitely open to collaborative authorship (hey, I did put it on github!), provided that other authors follow the general style and spirit of the book. At some point, I'll try to define this all in a more concrete way. If you do want to contribute, you'll probably want to have a good look at the commented Makefile, and also notice the use of "extra lines" between code samples. I've managed to find workarounds for the somewhat finicky pandoc/docbook tool-chain (and I'm thankful that it works at all since these tools really make life so much easier!)

Special acknowledgement

I feel obliged to commend Addy Osmani on his countless community contributions, particularly in developer education, and would like to acknowledge that seeing the impact of his work is part of what inspired me to write works like this myself. If you haven't already, you should check out his book: Essential JS Design Patterns is where I shamelessly lifted the pandoc build process being used here!

Introduction

Why rapid prototyping?

Many projects for the web go from the meeting room, to a wireframing tool or sketch pad, and then quickly move to the Adobe Creative Suite where high fidelity mocks are created. Then, these mocks must be painfully sliced and diced in to assets. Finally, a front-end developer meticulously hand crafts "pixel-perfect" pages ready for web consumption. While getting pristine looking sites may still involve some of these steps, we now have quicker alternatives.

Frameworks like HTML5 Boilerplate, Twitter Bootstrap, and Zurb's Foundation make it ridiculously easy to quickly put together acceptable web page prototypes. Even more full-blown workflow tools like Yeoman can integrate—not just rapid prototyping—but also things like library updating, image optimization, compression, minification, deployment, etc.

Once we've got an initial prototype of our web site, we can choose to iterate further, fully customizing it, or, scrap the whole idea and go back to the drawing board. The ability for lean start-ups to "fail-fast", pivot, adapt, get to market fast, etc., are all key. The rapid prototyping approach we'll be discussing in this book provides some tools to help the developer to cope with these time sensitive realities.

So, there you have it, my elevator pitch on why rapid prototyping is an important addition to your tool kit. Yes, a lot more could be said (see links in the section below), but I'll purposely depart early from any sort of long-winded persuasive argument so we can get to the fun stuff... "doing".

More info please!

Erica Heinz gave a nice presentation on utilizing rapid prototyping which is quite persuasive. Here are her slides.

Jeff Gothelf and Josh Seiden are advocating an approach to UX which they call LeanUX. Also see these slides.

Here's a nice slide-deck which proposes ditching wireframing altogether to instead use Twitter Bootstrap to prototype a responsive web site.

Standing on the shoulders of giants

Every front-end developer has that perfect combination of tools and libraries that he or she uses to improve workflow. Sometimes its that special combination of plugins to trick out their favorite editor or IDE (e.g. the perfect Vim, Sublime Text, or WebStorm set up); sometimes its an assortment of custom shell scripts; sometimes its a certain combination of libraries that all projects get started with, etc. As we continue to learn of new tools, libraries, and best practices, we further pepper our set ups. But it turns out to be *really hard* to keep up with "the cool kids". This is why an *opinionated tool* such as Yeoman helps out so much. We're greeted with an assortment of battle-tested components that help us maintain best practices while speeding up our workflow. But we do need to do a bit of homework ourselves to really benefit... or do we?

In a recent interview with netmagazine.com, Paul Irish, one of the core developers of Yeoman, addresses an interesting question regarding the use of libraries we don't fully understand:

> "...there's a common sentiment that says: 'If you don't know how something works, you shouldn't use it.' JavaScript libraries serve a purpose of papering over browser APIs to create a much more functional API to interface with. But these libraries sit on top of the browser so, if we take the same argument that says you shouldn't use something you don't [understand], are we going to apply that to the entire browser as well? It's hard to say that you should understand the entirety of the browser before you use it," he adds.

Obviously, we want to generally understand "what's going on", but we don't want to get so offtrack with researching how things work that we completely sideline our current projects. We have to strike a balance.

Interestingly, the Yeoman tool itself has so many sub-components, that only the most seasoned front-end developer will come to it understanding them all. It's easy to feel bit overwhelmed at first. One of the goals of this book is to help the reader understand some of these tools individually so they can use workflow tools like Yeoman more effectively.

We'll discuss:

- Twitter Bootstrap
- Compass/Sass
- Modernizr
- RequireJS
- JSHint
- OptiPNG and JPEGTran

And of course we'll look at Grunt and Bower (both of which are now part of Yeoman 1.0 itself). Once you've got a handle on these tools, you may later choose to, say, swap Zurb Foundation in place of Twitter Bootstrap, LESS for Sass, etc. But the general philosophy and approach will be close enough to what you've learned that you'll be able to do so confidently.

Please be forwarned that we won't be going extremely "deep" on these tools but, rather, we will provide enough of an introduction to gain a general understanding of what the tool does and how to get started with it. You should plan to consult each tool's documentation for in depth coverage.

Setting up

In this chapter we'll briefly introduce CSS preprocessors (Compass, Sass, LESS, etc.), Twitter Bootstrap, and end with a look at a full-blown workflow tool called Yeoman:

- What is a CSS preprocessor and why should I use one?
- What are Compass and Sass?
- What exactly is Twitter Bootstrap and how can I benefit from it?
- What is Yeoman, and how can it help my rapid prototyping workflow?

Let's discuss some of the core tools we'll be using throughout the remainder of this book...

What is a CSS preprocessor?

A CSS preprocessor is simply a tool that takes text you've written in the preprocessor's language (usually a super-set of CSS), and converts it into valid CSS. Because the preprocessor language is, essentially a super-set of CSS, it adds useful mechanisms such as variables, nesting, mixins, basic math, etc.

Just taking the variable feature, for example, you might define a color variable in one place and then reference it later as needed:

```
$dark: #333;
...
.foo { background-color: $dark }
.bar { background-color: $dark }
.baz { background-color: $dark }
```

Later, if you decide you'd like $dark to be, well, a bit darker, you could simply redefine the initial declaration like so:

```
$dark: #191919;
```

Now, `.foo`, `.bar`, and `.baz` will all be updated to use the new background-color the next time your .scss file is converted to CSS.

I'd be remiss not to mention that the three most popular CSS preprocessors today are LESS, Sass, and Stylus. All have their merits but we'll primarily be using Sass in this book.

Why use a CSS preprocessor?

If you've done much web development, you're already aware that CSS can get unruly fast! Using a preprocessor affords a nice means of keeping CSS organized and maintainable. This point is best proven by example—so let's move on to discussing Sass.

What is Sass?

Sass is an open source tool that allows its metalanguage—also called Sass—to be interpreted into CSS. It has two syntaxes, `.sass` and `.scss`. We'll only be covering the `.scss` syntax which is a super set of CSS that provides conveniences such as: variables, nesting, mixins, selector inheritance, and much more. [1] In a bit, we'll examine exactly what those mechanisms are and how they work. But for now, let's take a look at Sass's complimentary technology Compass. [2]

What is Compass?

Compass is a combination of things. It's a workflow tool for Sass that sets up relative paths (such as the relative path to your images via the image_url property; it does this via a config.rb configuration file). It then "watches" changes you make in your .scss files compiling those in to valid CSS. Compass also provides is a vast library of reusable Sass mixins for grids, tables, lists, CSS3, and more. Lastly, Compass is a full scope platform for building frameworks and extensions. [3]

Again, we'll be going over how to use Compass in more detail soon, but first let's have a quick look at Twitter Bootstrap...

Twitter Bootstrap: Not just a CSS framework!

Twitter Bootstrap is an open source framework that contains a set of CSS boiler plate templates for typography, buttons, charts, forms, tables, navigation and layout, etc. This CSS depends on a small set of HTML class name conventions such that any web author can "hook into" these styles by simply providing the proper markup. It also features a 12-column responsive grid so your site can adapt to different devices. It's currently the most popular GitHub project and used by big hitters such as NASA and MSNBC. [4]

In addition to interface components, the Bootstrap framework provides a plethora of JavaScript plugins that support dynamic UI components such as Modal, Tab, Tooltip, Popover, Alert, Carousel, Typeahead, Dropdown, and more. It's only real dependency is jQuery.

Setting Up For Development

In this section we will be setting the stage for things to come by installing Compass/Sass, Twitter Bootstrap and any other dependencies along the way. We'll first show how you to set up some of the tools individually, and then show how you can do it all at once with Yeoman. If you're already sure you want to use Yeoman feel free to skip to that section (but first ensure you have Git and Ruby installed).

In this section we'll cover:

- Installing Compass and Sass
- Installing Twitter Bootstrap

Installing Ruby

In order to use Compass and Sass you'll need to first install Ruby. If you're on OS X you already have it. For Linux users I'm going to assume you're adept enough on the command line to get Ruby installed yourself. Windows users can download an executable installer.

Installing Compass and Sass

With that done, you should be able to open a command line and use the `gem` command. If you install Compass you get Sass installed for free. You'll need to use the command line terminal.

Windows

```
$ gem install compass
```

Linux / OS X

```
$ sudo gem install compass
```

OS X GUI Installer

If you're on a Mac, you can optionally use Chris Eppstein's graphical installer package

Sass Only

If for some reason don't want to install Compass you can install Sass individually as follows:

```
$ gem install sass
```

Installing Twitter Bootstrap

If you already have extensive experience with Bootstrap and just want to get on with using Sass to customize Bootstrap, you may choose to skip (or perhaps skim) this section.

Before combining technologies such as Bootstrap and Sass, it's useful to play with them in isolation to get a better understanding of how they work. In that spirit, let's download a "vanilla version" of Bootstrap (not adapted for Sass) and have some fun. *Don't worry, we'll soon get to using things like Yeoman, sass-bootstrap, etc.*

There are a couple ways to get Bootstrap. One is simply to go to their site and download the zip:

- Go to http://twitter.github.com/bootstrap/
- Click the huge **Download Bootstrap** button
- Extract the downloaded file and ensure you see the `css`, `img`, and `js` directories
- Go to the Twitter Bootstrap examples page
- Right-click any of the examples you'd like to play with and 'Save Link As'
- Save the .html file to the top level of the same directory you extracted Bootstrap to
- Open the .html file in an editor and search for: `../assets/` and replace with empty string (empty string... as in blank!)

This should have found any link or src tags with relative paths like:

```
<link href="../assets/css/bootstrap-responsive.css" rel="stylesheet">
```

and replaced them with relative paths that look like:

```
<link href="css/bootstrap-responsive.css" rel="stylesheet">
```

Now double click that file and it should look as it did when you previewed it on their web site. If you're unfamiliar with Twitter Bootstrap, feel free to start hacking away off that static file now, or skip to the exercises section below.

Alternate Install

If you're more of the command line type you've probably already cloned their repo, but if not try this (you'll need to have and internet connection and Node.js and Git installed):

```
$ git clone git://github.com/twitter/bootstrap.git && cd bootstrap && npm install && make &&
```

That will clone the Bootstrap repository, put you in the cloned directory, install all the node packages that Bootstrap requires, build Bootstrap's LESS files, compile it's documentation, etc., and run the full test suite... whew!

Alternatively, if you happen to have nodejs and Twitter's package manager Bower installed you might just do:

```
$ bower install bootstrap .
```

If you've elected to use one of these command line methods to download Bootstrap, you should still go download an example .html file from the Twitter Bootstrap Examples page and ensure you can get it to render properly on your local system by replacing any invalid relative paths. Then do the exercises that follow.

Workflow Alternatives

This section will discuss some workflow alternatives for designers, and Yeoman for our command line lovers.

Give me a GUI please!

For those of you that prefer to stay away from the command line you have some GUI alternatives.

Scout

Scout is a simple GUI that sits on top of Adobe Air. Download the Scout installer for either OS X or Windows and simply follow the instructions to get it installed. Start a new project by opening up Scout and clicking the plus sign on the lower left, then navigate to the directory you'd like to create your project in. Once you've selected a directory and clicked 'Open', you're new project will show up in Scout on the left side.

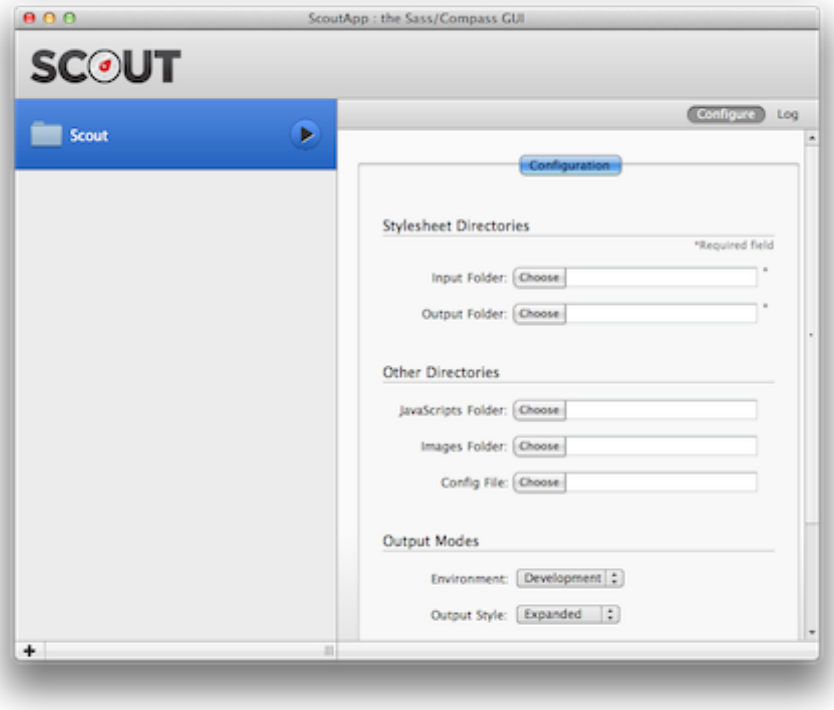

Figure 1: Opening a project in Scout

Above, I've created a folder called "Scout". As you can see, we're required to select the input and output folders. However, we haven't set those yet... let's do so.

In your project directory (the `Scout` directory in the above example), manually create the following directory structure and files (you'll use `compass create` to do much the same later but let's do it manually for now):

```
|-- css
|-- index.html
```

```
|-- sass
    |-- style.scss
```

Above we have two directories `css` and `sass` and two files `index.html` and `sass/style.scss`. The `index.html` file should contain:

```
<!doctype html>
<head><title>Compass Sass Sandbox</title>
<link href="css/style.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div class="test">This is a test.</div>
</body>
</html>
```

And the `sass/style.scss` file should contain:

```
@import "compass/reset";
$testColor: #008080;
.test {
    color: $testColor;
}
```

It should be self-evident in the above `.scss` file that we're importing Compass's reset module, defining a color variable, and then using that variable on the `.test` class we defined earlier in our markup.

Now go back to the Scout application. For the 'Input Folder' click the 'Choose' button and find the `sass` directory we defined earlier; now do the same for the 'Output Folder' but this time choose the `css` directory. The idea here is that the input files will get fetched from the `sass` directory (where we have our `.scss` files), get converted to proper CSS, and then output as `.css` files to the our output directory.

Once you've set up the input and output folders, simply click the big "play button" beside your project name to start Scout "watching" for file modifications. The first time I did this it took several seconds before I actually saw the output on the log tab showing that the style.scss file was detected and the style.css file was created:

If for some reason you don't see this try re-saving your style.scss file to force Scout to compile it.

At this point you should be able to double click on the project's index.html file and see "This is a test in teal". Not too exciting yet—I know—but we've now seen a simple Compass/Sass workflow using Scout. Try making a few more edits

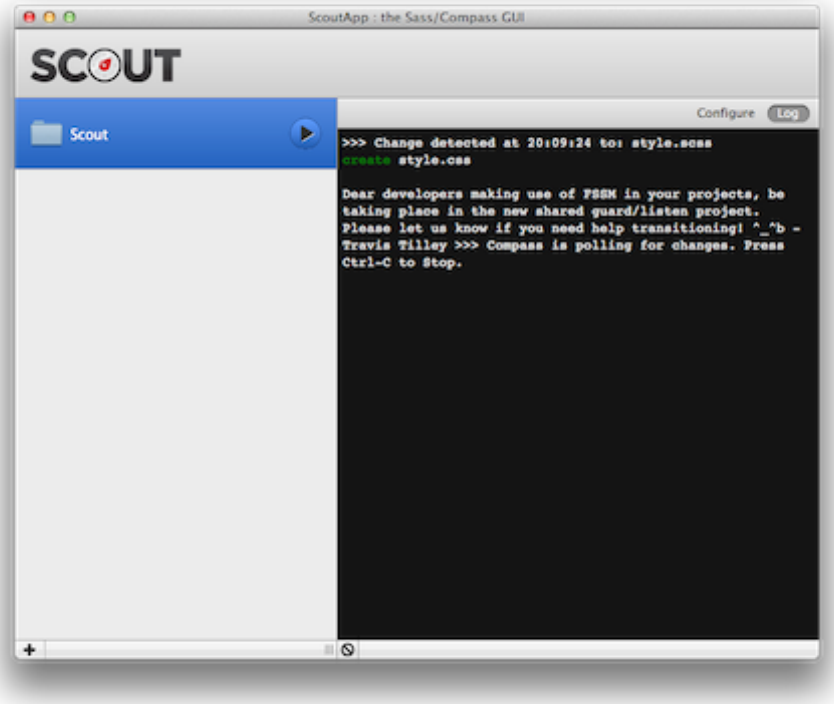

Figure 2: Scout can convert your .scss files to .css files

to the .scss file and you'll see that Scout detects them and recompiles a new modified .css file for you. Nice!

While your at it, be sure to have a quick look at the generated CSS file. Now try removing the Compass reset import line and see what's generated. You should see all of the boiler-plate reset CSS was removed (as expected), and just see the changes made on the `.test` class. Our Sass changes are reflected immediately in the corresponding CSS file.

Commercial GUI

If you're willing to fork out a small sum of money for slightly more aestetically pleasing interfaces and features, you might want to take a look at the following alternatives: CodeKit, Compass.app, or LiveReload. Keep in mind, though, that Yeoman, the tool we'll be discussing next, will also give you this sort of *watch* functionality and it's free!

Yeoman

If you're a CLI junky looking to fully optimize your front-end set up, you might want to take a look at Yeoman. Yeoman is spearheaded by none other than Paul Irish, Addy Osmani, and Sindre Sorhus. Yeoman bundles Grunt, Bower, Modernizr (and much more) in to one very convenient command line tool. It is still in BETA, so don't use unless you're "adventurous". That said, the author has had success using Yeoman since version 1.0 was released a few months before writing this guide. We'll be using Yeoman to do all our heavy lifting through-out the remainder of this book.

If you have Node.js, Git, Ruby and Compass already installed, you should be able to get Yeoman up and running with the following commands (only do this if you have an Internet connection!):

```
$ mkdir myproject && cd $_ # $_ is last argument of previous command
$ npm install -g yo grunt-cli bower # -g installs these globally
$ yo webapp # answer any questions and hit ENTER
# $npm install && bower install .. it seems this is now done for you automatically
$ grunt server
```

You used to have to manually install your dependencies via Bower and NPM, however, it now seems that this is done for us. I noticed output "whiz by" that stated:

```
I'm all done. Running bower install & npm install for you to install the required dependenci
```

Obviously, if you have dependency issues, you can run these command yourself or consult the Yeoman issue track, etc.

At this point, you'll have a fully prepared web app scaffolded and should be previewing your web app in a web browser:

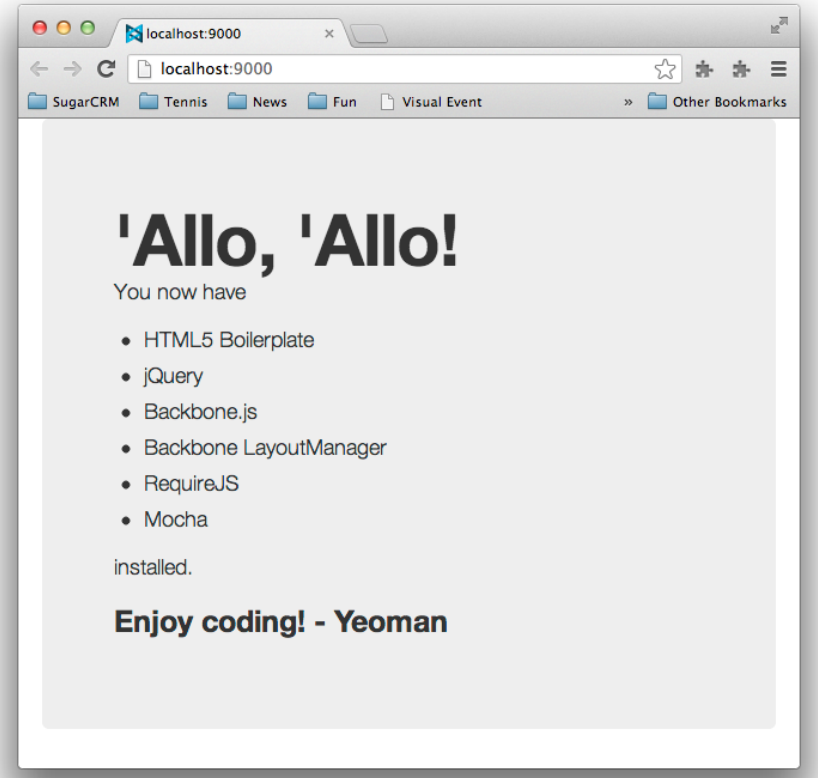

Figure 3: Grunt server loads the default project start page

When you ran the `yo webapp` command, one of the questions you were asked should have looked something like:

```
Would you like to include Twitter Bootstrap for Sass? (Y/n)
```

It's optionally setting up a Twttier Bootstrap port to Sass for us. Well, isn't that nice! Yeoman makes it incredibly convenient to set up Compass/Sass based projects fast.

You can further adapt a project like this using one of the primary Yeoman tools which are:

- Yo—a tool for customizing projects and generating scaffolding
- Grunt—you can use Grunt to create custom workflows for testing, deployment, coding standards verification, etc.
- Bower—use Bower to install and/or updated your front-end packages

Did you notice the `yo webapp` part back when we initiated our Yeoman project? In Yeoman parlance, that `webapp` thing is called a *generator*. The `webapp` generator is installed for us by default, but other generators you might want to use require you to install them yourself. You can do that via npm (that stands for *node package manager* and it's bundled with Node.js).

New generators are being added all the time, but at the time of writing this, the Yeoman docs list the following officially supported generators:

- Web App (comes by default)
- AngularJS
- Backbone
- BBB (Backbone Boilerplate)
- Chrome Apps Basic Boilerplate
- Ember
- Jasmine
- Mocha
- Karma

As we mentioned earlier, any generator besides the built in `webapp` generator needs to be installed separately. For example:

```
$ npm install -g generator-bbb # -g installs the bbb generator globally
$ mkdir myproject && cd $_ && yo bbb
$ grunt && grunt test && grunt server
```

That would install the Backbone Boilerplate generator, create a project, and then build, test, and preview it.

The above example workflows are just a couple ways you might use Yeoman to scaffold out a web app. Visit the Yeoman site (or the author's How To Code Youtube channel which has several tutorials on using Yeoman) to get more information on this lovely tool.

Exercises

Here are some easy exercises to get yourself familiarized with Bootstrap:

- Have a quick read through of the Twitter Bootstrap documentation which is only a half dozen pages or so. Don't worry about memorizing every last detail; just try to get a general feel for where they cover what, what's available, conventions they use, etc.—you'll be visiting these docs frequently

If you haven't already used Twitter Bootstrap before also do the following:

- Take the skeleton app we created above (in the section on Scout), and add jQuery and Bootstrap (in that order). The goal is to get a simple static page assembled with a form, table, and perhaps a navigation bar. Alternatively, if you've installed Compass you can create a similar file structure with the command: `compass create`. You will still have to create the `index.html` file though, so again, refer to the Scout section above for that.

For guidance on how to refactor the simple index.html page example, first have a look at the Getting Started page * Also have a a look at the Layouts section for guidance on how to control your widths and flow * See the Base CSS section for guidance on forms and tables * See the Navbar docs for guidance on the navigation bar

If you're more of a visual learner see the author's video for an example

- If you've installed Yeoman and/or Compass, and you're adventurous, try adding a few `mixins` as described in this tutorial on getting started with Compass. But don't worry, this is "extra credit"—you can of course wait until we properly discuss Compass and Sass later in the book.

Summary

In this chapter we've:

- Discovered CSS preprocessors
- Discovered and installed Compass and Sass
- Discovered and installed Twitter Bootstrap
- Saw some Compass and Sass GUI alternatives
- Played a bit with Yeoman

It's now time to delve in to the syntax of Compass and Sass. Let's go get our hands dirty, shall we!

Introduction to Compass and Sass

This chapter will server as a short introduction to Sass and go over the basic syntax, workflow, etc., as well as introducing the advantages of incorporating Compass as well. This will be a whirlwind tour (not for the feint of heart); if you'd like a more gentle and complete introduction to Compass and Sass do have a look at Sass and Compass for Designers by Ben Frain.

Topics covered:

- Compass/Sass Workflow
  - Compass Essentials: config.rb, watch, and CSS3 mixins
- A sandbox to play in (setting up a test script)
- Variables
- Nesting
- Mixins
- Placeholders
- Structure
  - @import
  - Utilizing partials

Using Compass to Create a Project

Remember in the last chapter how we manually created a directory structure like:

```
|-- css
|-- index.html
|-- sass
    |-- style.scss
```

Well, don't be mad, but with Compass with can create all of that less the index.html page with the following simple command:

```
$ compass create --css-dir "css"
```

The only reason we needed the `--css-dir "css"` part was to force Compass to use the directory name "css" (instead of "stylesheets" which is the default). You can also pass options for where to find your JavaScript and images like `--javascripts-dir "path_to_your_javascript"` and `--images-dir "path_to_your_images"`. These get used to build the config.rb file that we'll be looking at in a bit.

What happens upon issuing the above command is Compass scaffolds out a minimal project for you that looks like:

```
|-- config.rb
|-- css
    |-- ie.css
    |-- print.css
    |-- screen.css
|-- sass
    |-- ie.scss
    |-- print.scss
    |-- style.scss
```

Notice that this is essentially what we did manually before with the added benefit that they've added the IE and print files as well (we'll be ignoring the IE stylesheet for the time being). What we had previously named `sass/style.scss` is now `sass/screen.scss`. If you look at the config.rb file you'll notice that our CSS and Sass directories are where you'd expect:

```
css_dir = "css"
sass_dir = "sass"
```

No surprises there.

A Sandbox to Play In

As Compass is really only concerned with stylesheet related things, we didn't get an index.html page so let's create that now:

```
<!doctype html>
<head>
    <title>Compass Sass Sandbox</title>
    <link href="css/screen.css" media="screen, projection" rel="stylesheet" type="text/css"
    <link href="css/print.css" media="print" rel="stylesheet" type="text/css" />
</head>
<body>
    <div class="test">This is a test.</div>
</body>
</html>
```

Above, we've included our `screen.css` or `print.css` files dependent on if we're printing or not. Let's now have a look at that config.rb configuration file.

Compass Configuration

Let's take the configuration file section by section:

```
http_path = "/"
css_dir = "stylesheets"
sass_dir = "sass"
images_dir = "images"
javascripts_dir = "javascripts"
```

This should be fairly self-evident but the `http_path` sets the root HTTP directory of the project from which all other paths will be relative to. If you wanted to add fonts too you might add the line:

```
fonts_dir = "fonts"
```

Let's have a look at the output style line:

```
# output_style = :expanded or :nested or :compact or :compressed
```

Let's assume we have the following defined in an .scss file:

```
.test {
    color: #ddd;
    .klass: {
        color: #aaa;
    }
}
```

By default, the output style will look something like this:

```
/* line 7, ../sass/screen.scss */
.test {
  color: #ddd;
  .klass-color: #aaa;
}
```

Notice that nothing's changed other than we have a comment that identifies the line and file. Changing the output_style to `:nested` we get:

```
/* line 7, ../sass/screen.scss */
.test {
  color: #ddd;
  .klass-color: #aaa; }
```

Notice that resulting css is nested. Now let's try `:compact`

```
/* line 7, ../sass/screen.scss */
.test { color: #ddd; .klass-color: #aaa; }
```

Notice that the CSS was all put on one line. Last let's finally try `:compressed`

```
.test{color:#ddd;.klass-color:#aaa}
```

Notice that the comment was removed along with all white space.

I'd suggest just leaving this parameter alone for the time-being as it will be easier to debug. Before generating production code you'd likely want to use `:compressed` to ensure speedy delivery. While there are more configuration options available, this should be enough for our purposes. If you need more detail have a look at the Compass Reference documentation which covers the options in detail.

Sass Features

Now that we're properly utilizing Compass to help us with our Sass workflow, let's dive in to the Sass syntax itself. We'll cover just enough of the features to get you started (as always, for more detailed coverage we suggest you go the the reference documentation).

We're going to use a simple navigation bar to go over a few Sass syntax concepts. Let's go ahead and replace the content within our `<body>` tags from:

```
<div class="test">This is a test.</div>
```

to the following navigation markup:

```
<ul id="nav">
    <li>
        <a href="#">Products</a>
        <a href="#">Services</a>
        <a href="#">About Us</a>
        <a href="#">FAQ</a>
    </li>
</ul>
```

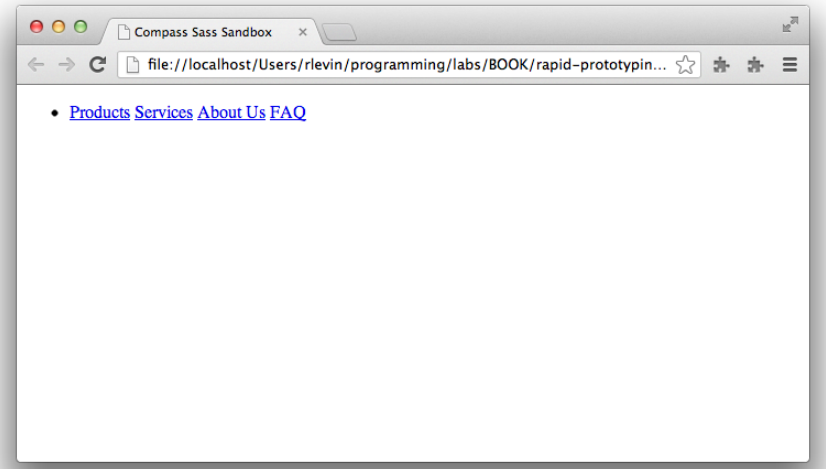

Figure 4: Navigation bar work in progress

At this point you should have something very plain like:

Now let's go to http://flatuicolors.com/ and grab some colors for our navigation bar (feel free to use your own colors...I've chosen blue colors). Since you're probably already familiar with CSS, let's go ahead and show how we might style the navigation bar with vanilla CSS.

Place the following CSS in `sass/screen.scss`:

```
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    background-color: #3498db;
    border-bottom: 1px solid #ccc;
    border-top: 1px solid #ccc;
}
#nav li a {
    display: inline-block;
    padding: 0.5em 1em;
    text-decoration: none;
    font-weight: normal;
    color: #fff;
}
#nav li a:hover {
```

```
    color: #fafafa;
    background-color: #4aa3df;
}
```

As you can see, this is plain old CSS. As Sass's `.scss` syntax is just a super-set of CSS, we can use any valid CSS within a Sass `.scss` file.

Now go ahead and either run `compass compile`, or, better yet, start a *watch* on our Sass files by issuing: `compass watch`. You should do this from another terminal tab in the same project directory so changes get immediately detected and compiled to CSS.

You should see something like the following (this will be our, ahem, amazing end result for the exercise):

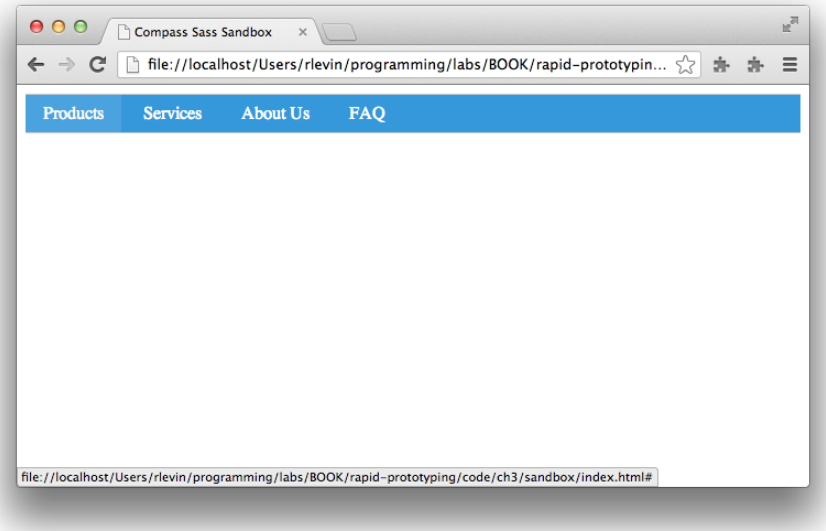

Figure 5: Navigation bar end result using CSS

Ok, so now that we have a working example of a pleasant (if extremely simple) navigation bar using CSS. Let's now see how we might *refactor* this CSS using Sass to make our lives easier.

Variables

If you look at the CSS we have at this point, you should notice that we have some *duplication* in our hex colors (e.g. `#ccc` is repeated twice, and `#fff` and `#fafafa` are definitely related). We can do better.

You've already seen in an earlier chapter that we can define variables in Sass using `$variable_name`. Let's do that by adding a `$white` and `$grey` at the top of the `sass/screen.scss` file, and then reference those variables as needed:

```
$grey: #ccc;
$white: #fff;
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    background-color: #3498db;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
}
#nav li a {
    display: inline-block;
    padding: 0.5em 1em;
    text-decoration: none;
    font-weight: normal;
    color: $white;
}
#nav li a:hover {
    color: darken($white, 2%);
    background-color: #4aa3df;
}
```

darken *towards the bottom should be fairly self-evident, but it's what is called a "mixin" (something we'll talk about shortly); for now, just now that it will darken the white by 2% leaving us with the* ***#fafafa*** *we had earlier.*

Let's go ahead and do a bit more by replacing our `background-color` with a `$blue` variable. This time we'll be using the `lighten` mixin which does the inverse of `darken`:

```
// Base Colors
$grey: #ccc;
$white: #fff;
$blue: #3498DB;//peter-river blue
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    background-color: $blue;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
}
```

```
#nav li a {
    display: inline-block;
    padding: 0.5em 1em;
    text-decoration: none;
    font-weight: normal;
    color: $white;
}
#nav li a:hover {
    color: darken($white, 2%);
    background-color: lighten($blue, 5%);
}
```

So we now have some color variables defined which can be used throughout the rest of our project as it grows. The next thing we can work on is the structure.

Nesting

In our CSS so far, we have a lot of repetition of the `#nav` selector. Let's use Sass's nesting feature to organize that into a more readable structure:

```
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    background-color: $blue;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
    li a {
        display: inline-block;
        padding: 0.5em 1em;
        text-decoration: none;
        font-weight: normal;
        color: $white;
        &:hover {
            color: darken($white, 2%);
            background-color: lighten($blue, 5%);
        }
    }
}
```

Most of this should be self-evident, but what's happening, is that Sass is smart enough to take that nested structure and convert it in to valid CSS. In fact, here's the CSS that's created from the above `.scss` file:

```
/* line 5, ../sass/screen.scss */
#nav {
  margin: 0 auto;
  padding: 0;
  list-style: none;
  background-color: #3498db;
  border-bottom: 1px solid #cccccc;
  border-top: 1px solid #cccccc;
}
/* line 12, ../sass/screen.scss */
#nav li a {
  display: inline-block;
  padding: 0.5em 1em;
  text-decoration: none;
  font-weight: normal;
  color: white;
}
/* line 18, ../sass/screen.scss */
#nav li a:hover {
  color: #fafafa;
  background-color: #4aa3df;
}
```

It looks pretty much like what we had before as the nested `#nav` blocks have been properly converted for CSS, and of course, our variables have been evaluated and converted to their actual hex values.

What's that `&:hover {` bit you say? Let's break that down:

- The `&` means "insert my parent selector here"
- So the `&` will essentially be replaced with an `a`
- the `:hover` is just a plain old `:hover` pseudo-class, so this will be turned in to: `a:hover`.

Just remember that anytime you see that `&`, you can say in your head "insert parent selector here".

Caution: This nesting feature, while attractive from an organizational and aestetic perspective, can be a bit dangerous. You should avoid too many levels of nesting (perhaps 4). Read more information on why at this article on the "Inception Rule".

Mixins

Mixins are a feature of Sass that allow you to create re-usable CSS. To define a mixin you must use the `@mixin` directive followed by the mixin's name and

any arguments it accepts in paranthesis. This is all followed by a CSS block containing the contents of the mixin. An example should make this all a bit clearer:

```
@mixin box-shadow($def) {
    -moz-box-shadow: $def;
    -webkit-box-shadow: $def;
    box-shadow: $def;
}
```

Above we have the `@mixin` directive followed by the name `box-shadow` followed by a parameter list of just one `$def` argument. This is all followed by the block which defines the CSS rules (in this case it's just some convenience for dealing with vendor prefixes).

To use this mixin we might do:

```
    @include box-shadow(0px 4px 5px $grey);
```

In fact, let's add this to our navigation bar with the following in `sass/screen.scss`:

```
// Base Colors
$grey: #ccc;
$white: #fff;
$blue: #3498DB;//peter-river blue
@mixin box-shadow($def) {
    -moz-box-shadow: $def;
    -webkit-box-shadow: $def;
    box-shadow: $def;
}
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    background-color: $blue;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
    @include box-shadow(0px 4px 5px $grey);
    li a {
        display: inline-block;
        padding: 0.5em 1em;
        text-decoration: none;
        font-weight: normal;
        color: $white;
```

```
        &:hover {
            color: darken($white, 2%);
            background-color: lighten($blue, 5%);
        }
    }
}
```

Our navigation bar now has a subtle shadow:

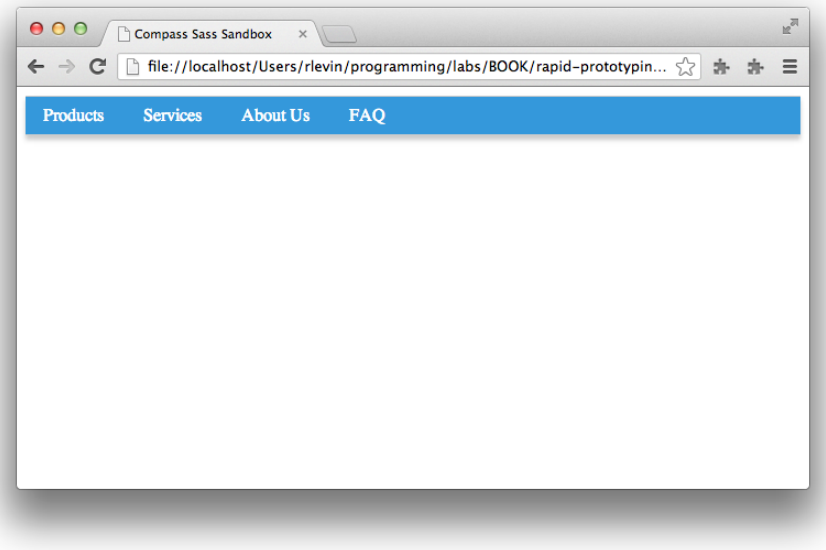

Figure 6: Navigation bar with box-shadow added

While this is not an extremely impressive mixin, it does provide an example of how they work. *It just so turns out, that Compass provides its own box-shadow mixin that we'll be levaraging later.*

The possibilities with mixins are truly endless as we can see by this next example refactoring where we've moved out the horizontal navigation CSS in to it's own mixin and then included that in our `#nav` rule:

```
@mixin horizontal-navbar {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    li a {
        display: inline-block;
        padding: 0.5em 1em;
```

```
        text-decoration: none;
        font-weight: normal;
        color: $white;
        &:hover {
            color: darken($white, 2%);
            background-color: lighten($blue, 5%);
        }
    }
}
#nav {
    background-color: $blue;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
    @include box-shadow(0px 4px 5px $grey);
    @include horizontal-navbar;
}
```

Mixins aren't all perfect though. Anytime you use a mixin using the `@include` directive, keep in mind that the rules in your mixin's block will all get copied over potentially causing duplication. This isn't always a problem, but if it concerns you, you'll be happy to hear that we have an alternative feature in Sass called *placeholders* which circumnavigate this issue alltogether.

Placeholders

Placeholders look like class and id selectors but instead use `%` before the name. They work in tandem with the `@extend` directive which allows a selector to inherit from another.

```
%button {
    padding: .5em 1.2em;
    font-size: 1em;
    color: $white;
    background-color: $green;
    text-decoration: none;
    border: 1px solid $grey;
    @include box-shadow(0px 2px 4px $grey);
    &:hover {
        color: darken($white, 5%);
        background-color: lighten($green, 10%);
    }
}
.button {
    @extend %button;
}
```

```
.rounded-button {
    @extend %button;
    @include border-radius(5px);
}
```

The way `@extend` works is that Sass uses CSS inheritance to re-use the block so duplication is avoided. For example, the above `.scss` code gets converted to the following CSS (pay attention to the first line indicating that the `.button` and `.rounded-button` inherit the same rules):

```
.button, .rounded-button {
  padding: 0.5em 1.2em;
  font-size: 1em;
  color: white;
  background-color: #2ecc71;
  text-decoration: none;
  border: 1px solid #cccccc;
  -moz-box-shadow: 0px 2px 4px #cccccc;
  -webkit-box-shadow: 0px 2px 4px #cccccc;
  box-shadow: 0px 2px 4px #cccccc;
}
.button:hover, .rounded-button:hover {
  color: #f2f2f2;
  background-color: #54d98c;
}
.rounded-button {
  -moz-border-radius: 5px;
  -webkit-border-radius: 5px;
  -ms-border-radius: 5px;
  border-radius: 5px;
}
```

As you can see, the two buttons share the CSS they have in common with only the border-radius being additionally added to `.rounded-button`. It's easy to imagine adding some text shadow, linear gradients, etc., to have a nice `%button` that all of you're project's buttons can `@extend` from.

Structure

If you've been following along with the code changes, you should notice that our `sass/screen.scss` has become a bit of a hodgepodge of variables, mixins, etc. Fortunately, we can use Sass's *partials* and `@import` directives to create a more modular project structure. While our little experiment is still pretty small, it should be evident that as a project grows, keeping things organized is of paramount importance. Let's go ahead and do this refactoring now.

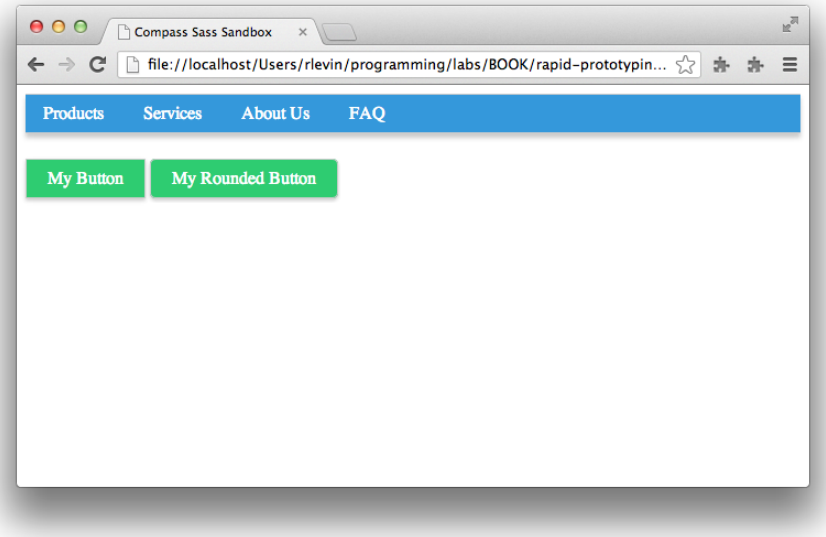

Figure 7: Buttons using placeholders

Partials and Imports In this section we're only going to look at the basics of using partials and imports, for more detailed guidance have a look at some various opinions on how to best structure your project:

- The Sass Way article on how to structure Sass projects—a great overview of how to organize your Sass based projects
- Compass best practices—some great suggestions from the Compass team
- SMACSS and Sass project structure—shows how one might use a SMACSS approach to organizing Sass based projects

Since our project is just a pedantic exercise and I don't want you to get "bogged down" in details, let's not worry too much about using the absolute perfect project structure. Instead, let's aim to simply move meaningful chunks of CSS code in to *partials* that make intuitive sense. Even this small organizational improvement will head us in the right direction.

What are partials? They're simply files that have Sass code that only pertains to one particular area of your project (e.g. colors, typography, etc.).

In our little experiment, we already have CSS for colors, buttons, navigation bar, mixins and placeholders. *In a more complete project we'd also have typography, reset, etc., but we'll purposely omit those for now to keep this simple.*

Let's go ahead and move these chunks of code in to partials now. In order to use partials with Compass and Sass we want to abide by the naming convention `directory/_name.scss`, and then we can *import* the partial dropping off the underscore and file extension like:

```
@import "directory/name"
```

The following shows our experiment converted to use partials.

Note that we'll show each file and then it's code like:

`path/to/file`

```
.foo {
    color: red;
}
```

So just to be clear, our sass directory should now look something like:

```
|-- ie.scss
|-- print.scss
|-- screen.scss
|-- base
|---- _colors.scss
|---- _mixins.scss
|---- _placeholders.scss
|-- layout
|---- _nav.scss
|-- modules
|---- _buttons.scss
```

Ok, so here are our new files...

`sass/screen.scss`

```
// Import partials...in real
// project we'd also have
// resets, typography, etc.
@import "base/colors";
@import "base/mixins";
@import "base/placeholders";
@import "modules/buttons";
@import "layout/nav";
```

`sass/base/_colors.scss`

```
// Base Colors
$grey: #ccc;
$white: #fff;
$blue: #3498DB;//peter-river blue
$green: #2ecc71;
```

sass/base/_mixins.scss

```
@mixin box-shadow($def) {
    -moz-box-shadow: $def;
    -webkit-box-shadow: $def;
    box-shadow: $def;
}
// Example of using default
// as a fallback
$default-radius: 5px !default;
@mixin border-radius($radius: $default-radius) {
    -moz-border-radius: $radius;
    -webkit-border-radius: $radius;
    -ms-border-radius: $radius;
    border-radius: $radius;
}
@mixin horizontal-navbar {
    margin: 0 auto;
    padding: 0;
    list-style: none;
    li a {
        display: inline-block;
        padding: 0.5em 1em;
        text-decoration: none;
        font-weight: normal;
        color: $white;
        &:hover {
            color: darken($white, 2%);
            background-color: lighten($blue, 5%);
        }
    }
}
```

sass/base/_placeholders.scss

```
%button {
    padding: .5em 1.2em;
    font-size: 1em;
    color: $white;
```

```
    background-color: $green;
    text-decoration: none;
    border: 1px solid $grey;
    @include box-shadow(0px 2px 4px $grey);
    &:hover {
        color: darken($white, 5%);
        background-color: lighten($green, 10%);
    }
}
```

sass/modules/_buttons.scss

```
.button {
    @extend %button;
}
.rounded-button {
    @extend %button;
    @include border-radius(5px);
}
```

sass/layout/_nav.scss

```
#nav {
    background-color: $blue;
    border-bottom: 1px solid $grey;
    border-top: 1px solid $grey;
    @include box-shadow(0px 4px 5px $grey);
    @include horizontal-navbar;
    margin-bottom: 2em;
}
```

By using partials with the `@import` directive, we've moved our Sass code into much more managable modules that will make our lives much easier as the project grows.

Summary

So we've now seen the power of Compass and Sass combined and have a better understanding of the basic syntax that's used to author Sass `.scss` files. We've purposely used these tools in isolation to get a better understanding of how they work on their own. Later, we'll show how we can tie these "CSS preprocessing tools" in with our workflow tool of choice, Yeoman. We'll also see how some of the boiler-plate Compass commands like `compass create` and `compass watch` are taken care of for us by Yeoman.

Before we look at combining the powers of Compass and Sass with Yeoman, let's first make sure we understand some of the core workflow tools that Yeoman is comprised of (Yo, Grunt, and Bower).

Yeoman

In this chapter we'll have a quick look at Yeoman's core tools: Yo, Grunt, and Bower.

TBD

Bootstrap and Sass

In this chapter we'll take a look at how we can customize Twitter Bootstrap using our favorite CSS preprocessor tool-chain, Compass and Sass.

TBD