Contents

Preface

In this book we'll be covering rapid workflows that utilize modern tooling from the perspective of the front-end developer, or *rapid prototyper* (although designers, managers, and anyone who needs to build web sites efficiently might want to read too). This approach uses cutting edge front-end tooling to achieve amazingly rapid developer workflow. While we respect that you've likely aready developed you're own personal favorite workflow rituals, we will attempt to persuade you that there may be sexier approaches you should add to your repetoire.

The pace of the book will be quite fast as we'll favor brevity over long winded explanations so we can dig deeper in to "doing". 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 use some type of language or syntax familiar to many front-end developers:

- HTML, CSS, JavaScript, etc.
- Command line

But we will also be using some tool-specific syntaxes like:

- Compass and Sass (the .scss version)
- Tool specific syntaxes (e.g. Grunt, Bower, Yeoman, etc.)

Assumptions

This book assumes experience with core 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

Addy Osmani has made many community contributions in the area developer education and has encouraged others to do the same. His contributions are generally open source, and, for example, I looked at the build process he used in his Essential JS Design Patterns book, and shamelessly *lifted* the pandoc build process we're using for this book!

Introduction

What is "Rapid Workflow"?

Rapid Workflow encompasses all of the activities that allow you to rapidly develop the front-end of a web site or SPA (single-page app for the uninitiated).

Frameworks like HTML5 Boilerplate, Twitter Bootstrap, and Zurb's Foundation make it ridiculously easy to quickly put together acceptable "first draft" web pages. Even more full-blown workflow tools like Yeoman can help us to—not just rapidly author web pages—but also update libraries, perform image optimizations, compression, minification, deployment, etc.

Once we've got an initial version 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 factors to success. The rapid workflow approach we're about to discuss provides guidance on how to use modern tooling to cope with the time sensitive realities of today.

Other Information on Rapid Workflow

While we'll be looking specifically at modern tooling for the front-end developer, there are some interesting and related resources regarding the general notion of improved development workflow that you might want to look at:

- Erica Heinz gave a nice presentation on utilizing rapid prototyping which is quite persuasive. Here are her slides. Rapid prototyping is akin to the rapid workflow approach we'll be discussing, and so many of her points are applicable.

- 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.

- The blogs of Addy Osmani and Paul Irish both have a large number of articles related to improved developer workflow.

Standing on the shoulders of giants

Every front-end developer has that perfect combination of tools and libraries that he or she already uses to improve their 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 what "the cool kids" are using at any given point in time. This is why *opinionated tools* such as Yeoman and Grunt can help us out so much. We're provided an assortment of battle-tested developer tools that help us maintain best practices and speed up workflow.

In this book we'll discuss the following tools that will help us to achieve rapid workflow:

TODO: We need to work out exactly what goes on this bullet list

- Compass/Sass
- Twitter Bootstrap
- Yo, Grunt, and Bower
- Unicorn
- Modernizr
- RequireJS
- JSHint

Once you've got a handle on these core 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 the workflow power-tool Yeoman. We'll answer these questions:

- What is a CSS preprocessor and why should I use one?
- What are Compass and Sass?
- What is Twitter Bootstrap and how can I benefit from it?
- What is Yeoman, and how can I use it to improve my 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:

```scss
$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:

```scss
$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.

We'll be diving in to using Twitter Bootstrap for what it's best at: prototyping. Whether using Bootstrap in production applications is up for debate (in our opinion it adds a lot of weight once you start wanting to override its defaults), so we'll also show you some viable alternatives like Zurb Foundation and Unicorn.

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).

Since hard-disk space is so cheap these days, we feel it's worth creating a `labs` *directory for experimentation, so you can work with each tool individually before leveraging it through Yeoman. This will give you the essential background for that tool to fully benefit when using Yeoman.*

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 && grunt
```

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 at the end of this chapter.

Workflow Alternatives

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

Give me a GUI please!

Here are some GUI alternatives you might consider adding to your workflow. We feel that using command line tools such as Yeoman is, ultimately, more productive. But there's nothing wrong with combining the powers of GUI and CLI if your prefer.

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.
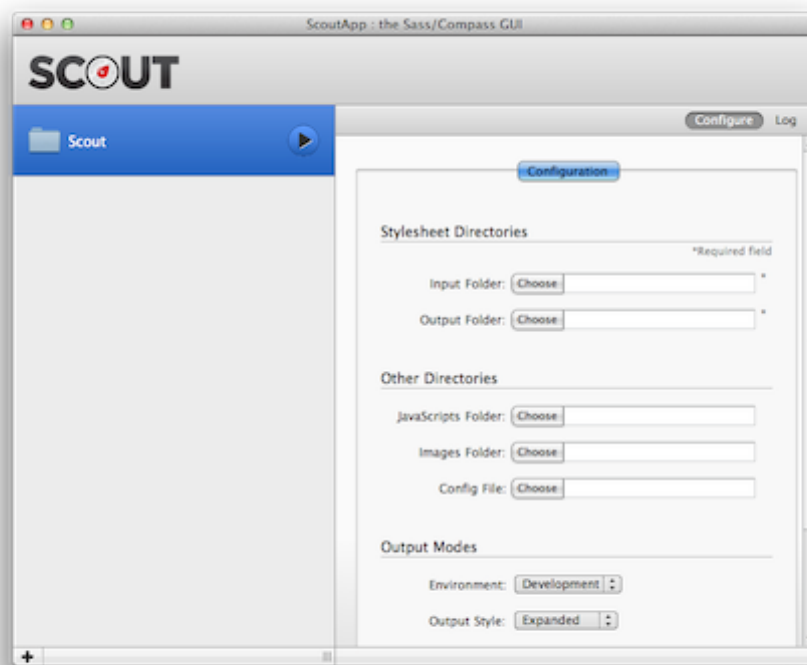
Figure 1: Opening a project in Scout

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:

```html
<!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:

```scss
@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:
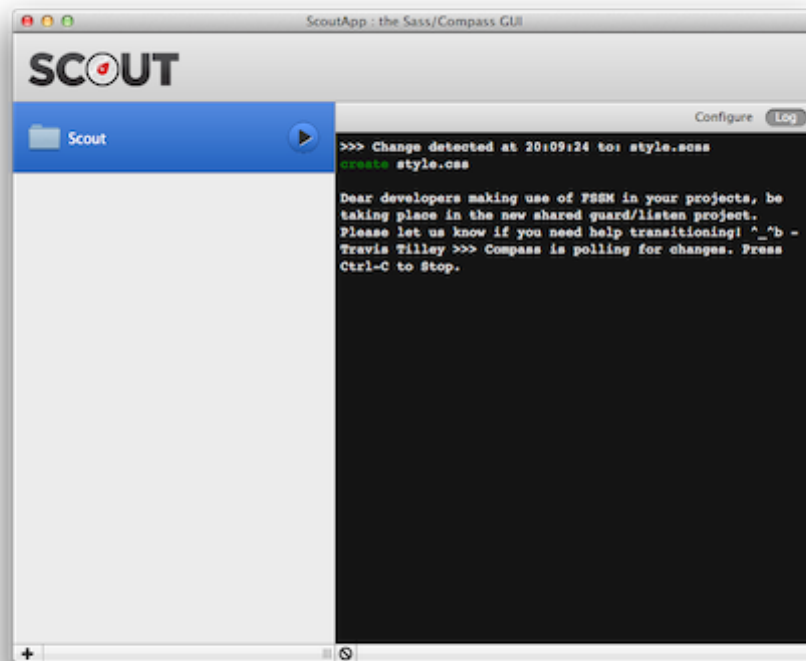

Figure 2: Scout can convert your .scss files to .css files

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 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 to use command line tool. It is still in BETA, but the author has had success using Yeoman since version 1.0 was released a few months before writing this book. 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 && npm install -g generator-webapp
$ yo webapp # answer questions choosing Bootstrap for sass and hit ENTER
$ grunt server
```

At this point, you'll have a fully prepared web app scaffolded and should be previewing your web app in a web browser. The following screen capture shows how it look when I ran this (however, since this is always changing and improving library, there's a good chance it will look different by the time you read this):

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

```
Out of the box I include HTML5 Boilerplate and jQuery.
[?] What more would you like? (Press <space> to select)
```

With a drop down of choices immediately following. We want the Bootstrap for Sass which sets up the Twttier Bootstrap port to Sass for us.

We'll soon be seeing how you can further adapt a project like this using one of the primary Yeoman tools which consist of:

- Yo—the tool for customizing projects and generating scaffolding (we just used it earlier)
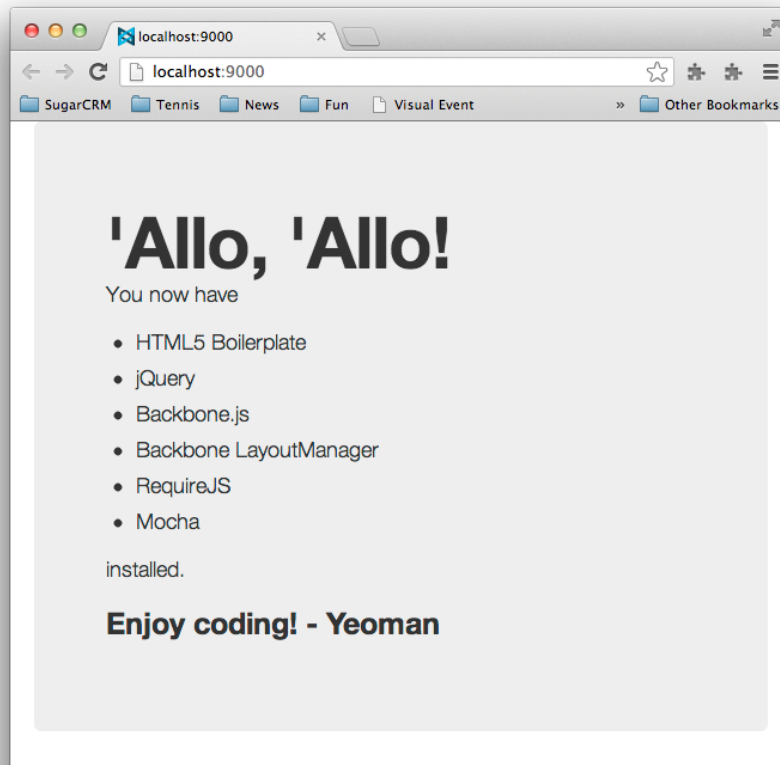
Figure 3: Grunt server loads the default project start page

- 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

The `yo webapp` line in our earlier Yeoman project initialization is, in Yeoman parlance, called a *generator*. The `webapp` generator was installed when we did `npm install -g generator-webapp`. Other generators you might want to use also 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 (again, this is always changing so expect a larger and/or different list by the time you read this!):

- Web App (the one we installed earlier)

- AngularJS

- Backbone

- BBB (Backbone Boilerplate)

- Chrome Apps Basic Boilerplate

- Ember

- Jasmine

- Mocha

- Karma

Here's another example of installing a generator:

```
$ npm install -g generator-bbb # -g installs the bbb generator globally
$ mkdir myproject && cd $_ && yo bbb
# answer any questions at prompt
$ 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 on combining Yeoman and Twitter Bootstrap to rapidly prototype a simple web page

- 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.

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 serve as a short introduction to Sass and go over the basic syntax, workflow, etc., and will additionally introduce the advantages of incorporating Compass in to your Sass workflow. 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>
```

19

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

Above, we've included our `screen.css` which will hold the styles for this exercise.

Compass Configuration

Let's take the `config.rb` 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

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:

```html
<ul id="nav">
    <li>
        <a href="#">Products</a>
        <a href="#">Services</a>
        <a href="#">About Us</a>
        <a href="#">FAQ</a>
    </li>
</ul>
```
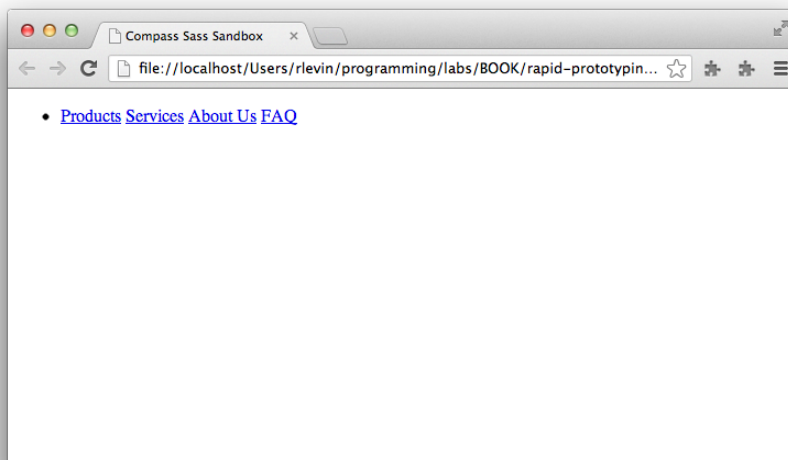
At this point you should have something very plain like:

Figure 4: Navigation bar work in progress

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:

```scss
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
```

```scss
    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):

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:

```scss
$grey: #ccc;
$white: #fff;
#nav {
    margin: 0 auto;
    padding: 0;
    list-style: none;
```
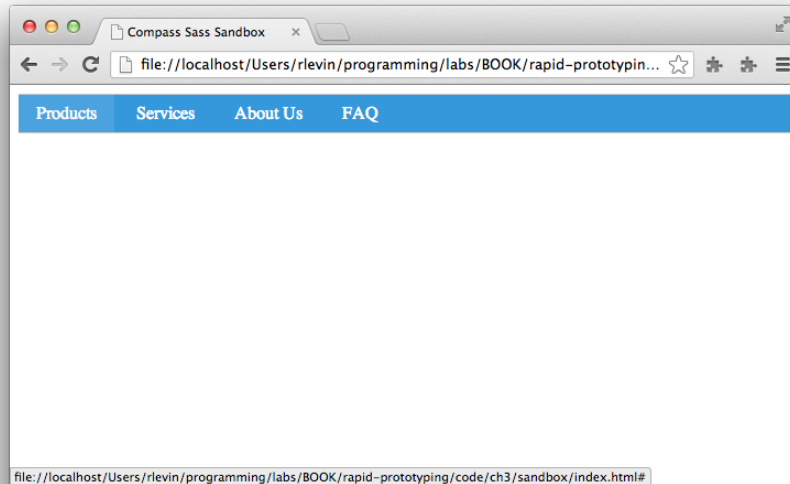
23

Figure 5: Navigation bar end result using CSS

```scss
    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`:

24

```scss
// 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:

```scss
#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;
```

```scss
            &: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:

```scss
/* 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;
```

```scss
        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%);
            }
        }
}
```

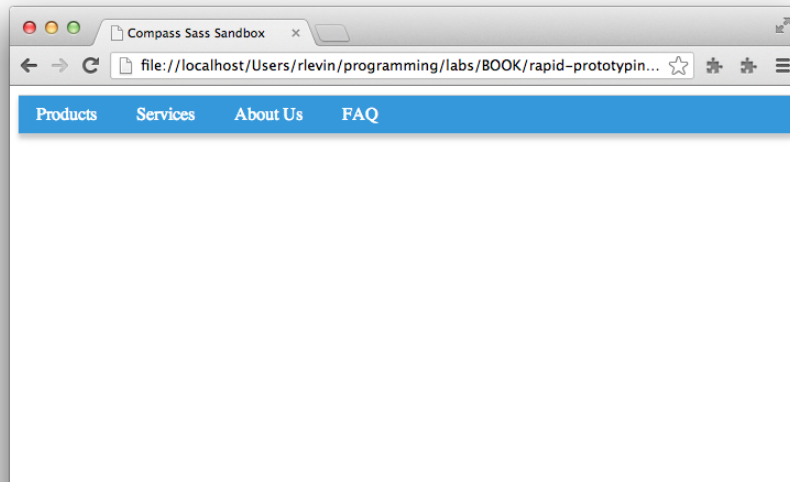Our navigation bar now has a not so subtle shadow:

Figure 6: Navigation bar with box-shadow added

While this is not an extremely impressive use of mixins, 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:

```scss
@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.

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

```scss
        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):

```css
.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
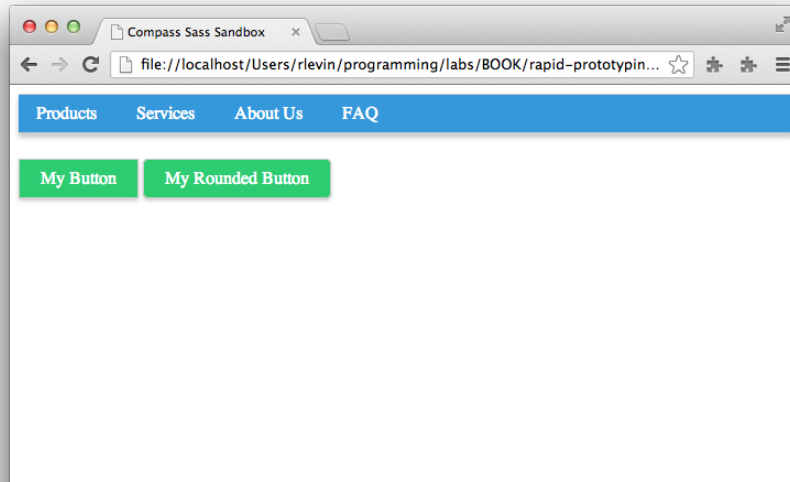
Figure 7: Buttons using placeholders

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.

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

31

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?   Partials are 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, once we create these partials our sass directory will look something like:

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

Here are the contents for these new files. . .

`sass/screen.scss`

```scss
// Partials imports
// note that in a "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`

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

`sass/base/_mixins.scss`

```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;
```

```scss
        text-decoration: none;
        font-weight: normal;
        color: $white;
        &:hover {
            color: darken($white, 2%);
            background-color: lighten($blue, 5%);
        }
    }
}
```

sass/base/_placeholders.scss

```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

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

sass/layout/_nav.scss

```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;
}
```

As you can see, not much has actually changed and we've simply restructured our Sass. But by using partials as we have (using th`@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.

Yeoman

In this chapter we'll be having a look at the workhorse workflow tool Yeoman. Yeoman's core consists of three tools: Yo, Grunt, and Bower. We will go over each in depth.

*If you haven't read the **Setting Up** chapter, go read that and ensure you're properly set up to use Yeoman.*

What is Yeoman?

At its heart, Yeoman is a workflow power-tool that provides a suite of popular front-end tools to facilitate developing with best practices and efficient workflow. It's pioneers, Addy Osmani and Paul Irish, are deeply commited to making the modern web development process better and faster. While there are a ton of tools available in the Yeoman eco-system, its imperative to learn its core tools first. Let's dig in.

Yo

The first core tool we'll look at is Yo, Yeoman's scaffolding tool that helps you to efficiently bootstrap projects. It not only scaffolds out your app, but also gives you sensible default configuration files that get you up and running quickly.

With Yo, there are three main things to learn:

- How to use Yo to bootstrap a project

- How to install custom generators

- How to create your own custom generators

We'll put off learning how to build a custom generator for a later chapter and now have a look at common `Yo` use.

Installing Generators

Perhaps you'd like to use the venerable [Backbone Boilerplate](#) via Yeoman. To do so you would first need to install the generator:

```
npm install -g generator-bbb
```

*This assumes you've installed node.js (which bundles **npm**).*

Next you would simply create a project directory and issue the `yo bbb` command as follows:

```
mkdir project && cd $_ && yo bbb
```

As usual, you can build and preview your Yeoman project using grunt as follows:

```
grunt server
```

As you may have noticed, we've installed the `generator-bbb`, but used `bbb` when issuing the `yo` command. This seems to be an undocumented idiom as `generator-angular` would be triggered with `yo angular`, etc.

Grunt

The next core tool we'll have a look at is Yeoman's build tool extraordinaire [Grunt](#) authored by "Cowboy" [Ben Alman](#). Grunt allows you to efficiently build and deploy your application, run tests, and preview your changes instantaneously. It's used by Twitter, jQuery, and [many other high-profile projects](#). Grunt has a mountain of features and plugins available and we'll only be covering some of them here; of course you can always reference their [docs](#) once you're comfortable with the basics.

Note that Grunt's sweet-spot is task-based configuration management (much like Ant or Make but in JavaScript/Node.js), where you're files are built as appropriately for production, testing, development environments, etc. As such, much of the documentation and tutorials you'll find on Grunt will already address these sorts of tasks. In order to get a more general understanding of Grunt, we'll be using much simpler if not slightly pedantic examples. Using this basic knowledge as a point of departure, you should then be able to utilize Grunt for more practical purposes.

The basics of how Grunt's command line system works are as follows:

- you run the Grunt CLI in a project directory that contains a `Gruntfile.js`

- the Grunt CLI runner loads the grunt library

- using the configurations found in that `Gruntfile.js` it then executes the task(s) as you've requested

If you're in a project directory and its dependencies have not yet been loaded, you'll need to do so with:

```
npm install
```

However, before we get in to usage, let's have a quick look at some of the parts of Grunt.

package.json

The `package.json` file is just a simple JSON file that declaratively specifies which dependencies should be included in your Grunt based project. See Grunt's documentation on the package.json for more information and an example on how this works. Luckily, when using Yeoman, we get a nice default `package.json` which can serve as a starting point for further customization.

Besides hand editing the `package.json` file itself, you can choose to install new packages with the following general syntax:

```
npm install PACKAGE --save-dev
```

(where `PACKAGE` is grunt plugin you want to install. Obviously, the plugin comes as an `npm` package). This will both install the package and add it to your `package.json` configuration file.

Gruntfile

The `Gruntfile.js` is simply a JavaScript file that contains your project's configuration and tasks. What's a task? A task is a single chunk of work you need to get something done. For example, one task might be to compress certain types of images, another to copy files from one directory to another, etc. We'll learn more about these by example so hold tight and it will all make more sense.

This `Gruntfile.js` should be placed at the root directory of your project (as with the `package.json` file discussed earlier). Taking a very high-level look at the contents of a typical `Gruntfile.js` we will find the following components:

- An outer "wrapper" function

```
module.exports = function(grunt) {
    // ...
};
```

- An initial configuration section:

```
grunt.initConfig({
  // ...
});
```

If you're familiar with JavaScript (and you better be for this book), you'll notice that `initConfig` is passed an *object literal*. That object literal represents a "configuration object" that will thereafter be available throughout the `Gruntfile.js` as `grunt.config`. The general purpose of this section is to set up any global initialization or bootstrapping required by your Grunt tasks.

- Default task

```
grunt.registerTask('default', ['task1', 'task2']);
```

The `registerTask` method is a general purpose means for registering Grunt tasks. It just so happens that here we've specified 'default' task which will be run by Grunt when we issue `grunt` without specifying any additional arguments.

Another example of using `registerTask`, can be seen from the boiler-plate generated by Yeoman when you run `yo webapp`. If you look at that generated `Gruntfile.js` it will contain something like:

```
grunt.registerTask('test', [
    'clean:server',
    'concurrent:test',
    'connect:test',
    'mocha'
]);
```

In this case, the task name is called `test`, and will, in turn, run a set of test related tasks.

We'll learn more about the power of registering tasks later, but this should suffice for having an initial understanding of what's going on. You may want to take a quick break and try the following from an arbitrary directory of your choosing:

```
mkdir tmp && cd $_ && yo webapp
cat Gruntfile.js # or just open in an editor
```

Using your editor or "grep-fu", find all the places in that file that use `grunt.registerTask`. It should be contextually evident what going on. Congratulations! You've now got a bit more of an idea what task Yeoman is helping you with out of the box.
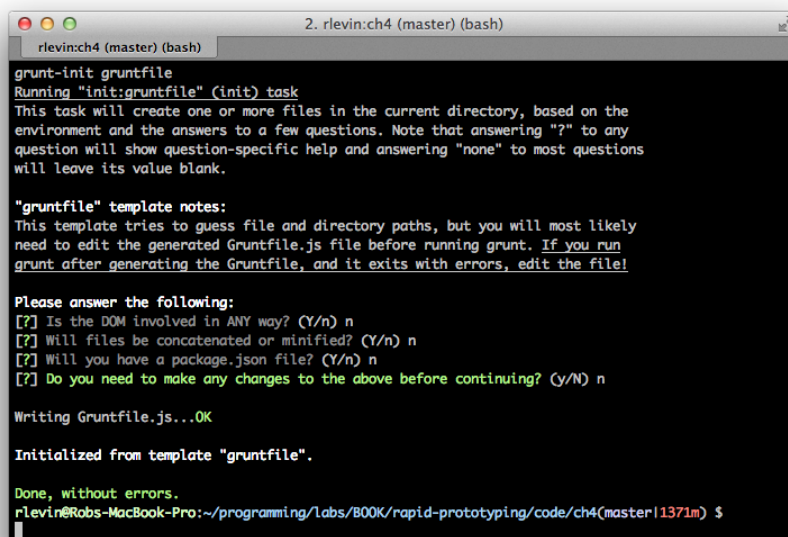
Grunt Init

Let's now perform the obligatory *Hello World* using Grunt.

As Grunt 0.4 onwards has completely modularized the Grunt system, each component is installed independently. Let's grab some convenient packages for Gruntfile authoring:

```
git clone https://github.com/gruntjs/grunt-init-gruntfile.git ~/.grunt-init/gruntfile
npm install -g grunt-init
mkdir myproject && cd $_
grunt-init gruntfile
```

For this exercise go ahead and answer No for all of Grunt init's questions. This should place a boiler-plate `Gruntfile.js` file in your `myproject` directory.

Figure 8: Initializing an empty Grunt project

As you can see, we've used `grunt-init gruntfile` to scaffold out an initial `Gruntfile.js`. Let's remove everything in the file for this exercise except for the following "wrapper" function:

```
'use strict';
module.exports = function(grunt) {
};
```

Now create a package.json with the following:

```
{
  "name": "myproject",
  "version": "0.0.1"
}
```

This is the bare minimum you'll need to supply for your `package.json`. Let's go ahead and ensure we have grunt installed for the project as a development dependency:

```
npm install grunt --save-dev
```

Our `package.json` file should now contain an entry for Grunt itself:

```
{
  "name": "myproject",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "~0.4.1"
  }
}
```

Go ahead and run grunt:

```
$ grunt
Warning: Task "default" not found. Use --force to continue.

Aborted due to warnings.
```

This makes sense, we haven't created any tasks, and issuing `grunt` with no arguments causes Grunt to look for a "default task". Let's create one now.

```
'use strict';
module.exports = function(grunt) {
    grunt.registerTask('default', 'Hello world example', function() {
        grunt.log.writeln('Hello world.');
    });
};
```

This should be painfully self-evident, but we've went ahead and registered the needed required task, and provided a callback function that simply prints our Hello World. Now try issuing grunt again:

Figure 9: Grunt Hello World

I'm a bit of an odd duck so don't be thrown off. . . I like to run my commands on the line following my PS1 prompt so I have a bit more room.

So, just to be clear that you understand the venerable default task, rename the task from default to foo, and re-run grunt. You should get the same error we had before since we no longer have a default task. But this time, issue grunt foo which will ask grunt to run the foo task.

Image Optimization

Let's go ahead and create a slightly more useful task that optimizes images by leveraging the grunt-contrib-imagemin plugin Grunt plugin. The plugin, in turn, uses OptiPNG to optimize .png images, and jpegtran to optimize jpg images. We'll be using jpg images here, but the process is essentially the same for png images.

First use the same process we used in the Hello World example above to grunt-init a Grunt project. Then install the imagemin plugin:

Figure 10: Grunt "foo" task

```
npm install grunt-contrib-imagemin --save-dev
```

Ensure that your `package.json` looks something like:

```
{
  "name": "image_opt",
  "version": "0.0.1",
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-imagemin": "~0.1.4"
  }
}
```

To recap, we have both `grunt` and `grunt-contrib-imagemin` included as development dependencies. There's a chance that your versions will be later than what's listed here as that should pull in the "latest" versions available.

To see this all in action, you'll want to create an `images` directory off your project's root directory, and place some images in there (I tested with `jpg` images, but `png` images should work with this too).

Now put the following in your `Gruntfile.js`:

```
'use strict';
module.exports = function(grunt) {
  grunt.initConfig({
    imagemin: {
```

42

```
      dist: {
        options: {
          progressive: true,
          // Only for pngs
          optimizationLevel: 7
        },
        files: [{
          // Assumes running from project root
          expand: true,
          cwd: './images',
          src: '*.{png,jpg,jpeg}',
          dest: './images/compressed',
        }]
      }
    }
  });
  // Include the imagemin npm
  grunt.loadNpmTasks('grunt-contrib-imagemin');
  // Register imagemin task as default task
  grunt.registerTask('default', ['imagemin:dist']);
};
```
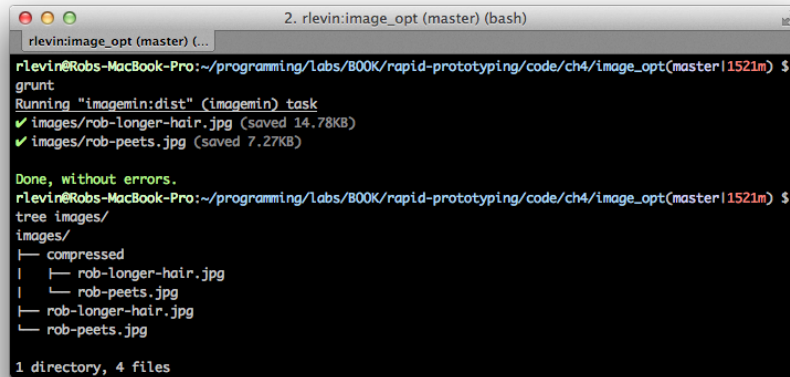
As explained earlier, the `initConfig` is simply an initial set up section that you can define globally accessible properties etc. Here's we've went ahead and put the `imagemin` task in `initConfig`. Nested within that task is a "target" called `dist`. In this example, there's only one target, but we could have included another one like `supercrunch` with higher compression levels, etc.

The `loadNpmTasks` loads the imagemin Grunt plugin, and finally, `registerTask` registers our `imagemin:dist` (the colon syntax stands for `task:target`) as the default task. As you should expect, we can run this default task by simply issuing `grunt` as follows:

As you can see, our compressed images have been created in the `images/compressed` directory, and we've shaved a bit of file size off our images.

Note that the imagemin plugin was a bit buggy initially but it seems to have become stable (at least in our experiments). If you for some reason can't get this running on your platform, be sure to visit the project's issue tracker. Please be sympathetic to the maintainer, since getting image optimization tools to work across platforms is quite a challenge!

It turns out that our little exercise here is included for us when using Yeoman. For example, after scaffolding a Yeoman project, check the `Gruntfile.js` (or `Gruntfile.coffee` if applicable) for the `img` task which will optimize .png or .jpg images using OptiPNG and JPEGtran similar to how we've just done here.

Figure 11: Grunt imagemin

Going Further

While you now have a nice grounding in Grunt, we've only scratched the surface. Please have a look at the Grunt Documentation for further mastery.

Bower

Yeoman's front-end package manager, Bower, helps you to manage your application's dependencies via a small set of command lines that make installing and updating libraries a breeze. While you've probably used another package manager tool that's somewhat similar to Bower, there's a particular philosophy that makes Bower unique. From their docs:

> There are no system wide dependencies, no dependencies are shared between different apps, and the dependency tree is flat.

> Bower runs over Git, and is package-agnostic. A packaged component can be made up of any type of asset, and use any type of transport (e.g., AMD, CommonJS, etc.).

In order to get up to speed quickly, you can always ask for help:

```
bower help
```

Finding and installing packages

The most common use case you'll have for Bower is to find and install a package which can be done as follows:

```
bower search jquery
```

This will produce a ton of output since there are many packages that have the string jquery in them. You might want to use something like the following to narrow your search, for example, if you were searching for the jQuery timeago package:

```
bower search jquery | grep time
    jquery-timeago git://github.com/rmm5t/jquery-timeago.git
    ... more output omitted

# To get information on timeago:
bower info jquery-timeago
jquery-timeago

  Versions:
    - v1.2.0
    - v1.1.0
    ... more output omitted

# Now install it with:
bower install jquery-timeago
```

You can, alternatively, use tags: `bash bower install <package>#<version>`

Intuitively, if you need to later update or uninstall the package you can simply do:

```
bower update jquery-timeago
# or to remove
bower uninstall jquery-timeago
```

You can also configure your settings by editing the `.bowerrc` file in your project's root. This is a simple JSON file that specifies what `directory` to store your components, etc. For more information, see the Bower configuration documentation.

While this is most of what you'll be doing day to day with Bower, it does also offer you the ability to create your own pacakges and interact with a programmatic API. These topics are out of scope for this book so the Bower site is your best bet should you need to delve deeper.

Yeoman Custom Generators

The material in towards the end of this chapter my be considered a bit advanced in that it will assume the reader either has had some experience using Node.js, or is resourceful enough to learn it on their own. Yeoman generators leverage Node.js, and so there are a combination of API's one can use to create a custom generator:

- Yeoman's Generator API

- Node.js API

The convention is to first look for a way to achieve your task using the Yeoman generator methods provided, but then utilize Node.js if/when needed. In this example, we'll be using a bit of both. You'll likely want to also have the Yeoman generator documentation open at the same time as reading this chapter.

You'll first need to install the `generator-generator` and then run that to create your generator project:

```
npm install -g yo generator-generator
mkdir schoolme && cd $_
yo generator
```

Answer it's questions and then ensure that you can play with your generator in progress globally by issuing:

```
npm link
```

For me, this linked the files like:

```
/usr/local/lib/node_modules/generator-schoolme -> /Users/rlevin/programming/labs/generator-s
```

For kicks, let run the pre-installed Mocha tests:

```
mocha test
```

Ok, we should be good to get started. . .
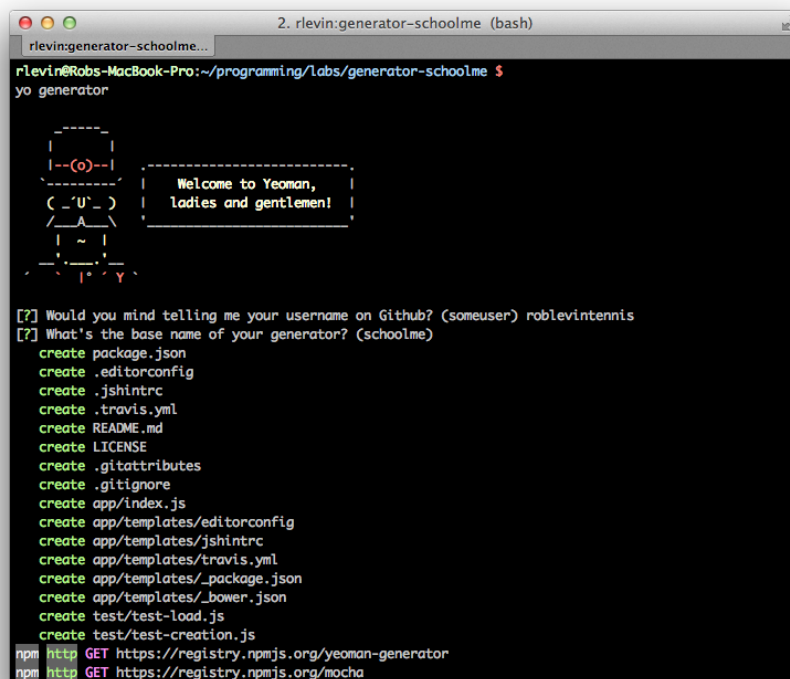
Figure 12: Yeoman's generator-generator

Figure 13: npm linking your Yeoman's generator

Figure 14: Running default generator tests

Yeoman's "chain of execution"

Take a look at the generator-schoolme/app/index.js which is our generator's main entry point. Yeoman's convention is to call the methods defined in this file from top to bottom.

- Constructor

```
var SchoolmeGenerator = module.exports = function SchoolmeGenerator(args, options, config) {
```

gets called first (but keep a mental note of the `this.on('end')` line as that registers for the asynchronous `end` callback).

- `SchoolmeGenerator.prototype` methods will be called next

- `this.end` defined in constructor will be called last

Edit that file to look as follows:

```
'use strict';
var util = require('util');
var path = require('path');
var yeoman = require('yeoman-generator');

var SchoolmeGenerator = module.exports = function SchoolmeGenerator(args, options, config) {
  // https://github.com/yeoman/generator/blob/master/lib/base.js
  yeoman.generators.Base.apply(this, arguments);

  this.on('end', function () {
    this.installDependencies({ skipInstall: options['skip-install'] });
  });

  this.pkg = JSON.parse(this.readFileAsString(path.join(__dirname, '../package.json')));
};

util.inherits(SchoolmeGenerator, yeoman.generators.Base);

SchoolmeGenerator.prototype.askFor = function askFor() {
  var cb = this.async();

  // welcome message
  var welcome =
  //... code purposely omitted
```

```javascript
    console.log(welcome);

    var prompts = [{
      name: 'fooOption',
      message: 'Would you like to do some Foo?',
      default: 'Y/n',
      warning: 'Yes: This will do foo action.'
    }];

    this.prompt(prompts, function (err, props) {
      if (err) {
        return this.emit('error', err);
      }
      this.fooOption = (/y/i).test(props.fooOption);
      cb();
    }.bind(this));
};

SchoolmeGenerator.prototype.foo = function app() {
  console.log("FOO got called");
};

SchoolmeGenerator.prototype.bar = function projectfiles() {
  console.log("Bar got called");
};
```

Let's run this. I advise that you have another console tab open so we don't later inadvertantly copy files in to our generator's project directory. You did the `npm link` step earlier so you should be able to run this from anywhere. So after opening a new console tab do:

```
cd /tmp && yo schoolme
```

After answering the promt you should see our foo and bar logs get printed as expected.

Another Simple Generator

Ok, so now that you're getting the hang of a simple generator, let's do one more, still fairly simple generator to simply port a small library of interest. Conveniently, many of the more popular projects are getting ported over to Yeoman generators daily. For example, don't wanna use Twitter Bootstrap, you'll find Zurb Foundation, etc. And it's only getting better with time. However, you may like some obscure or new library that you'd like to include in your

workflow. Well, it's really not that hard! For our second, somehwat more useful generator, I've decided I'd like to port over a sweet little color library for Sass called color-me-sass.

I went through the same general process of using the Yeoman `generator-generator`, and stripped the `app/index.js` to the following (inspired by the already existing `generator-pure`):

```javascript
'use strict';
var util = require('util');
var path = require('path');
var yeoman = require('yeoman-generator');

/**
 * Port of https://github.com/RichardBray/color-me-sass
 */
var ColormeGenerator = module.exports = function ColormeGenerator(args, options, config) {
  yeoman.generators.Base.apply(this, arguments);

  this.on('end', function () {
    // this.installDependencies({ skipInstall: options['skip-install'] });
    var guide = 'Colors Installed! '.yellow.bold +
    '\nYou should now place something like the following in your _base.scss or similar:'.blu
    '\n@import "colors/reds";' +
    '\n@import "colors/peaches";' +
    '\n@import "colors/tans";' +
    '\n@import "colors/oranges";' +
    '\n@import "colors/ambers";' +
    '\n@import "colors/yellows";' +
    '\n@import "colors/limes";' +
    '\n@import "colors/greens";' +
    '\n@import "colors/turquoise";' +
    '\n@import "colors/blues";' +
    '\n@import "colors/purples";' +
    '\n@import "colors/pinks";' +
    '\n@import "colors/browns";' +
    '\n@import "colors/whites";' +
    '\n@import "colors/grays";' +
    '\n@import "colors/bootstrap";\n';
    console.log(guide);
  });
};

util.inherits(ColormeGenerator, yeoman.generators.Base);

ColormeGenerator.prototype.askFor = function askFor() {
```

```javascript
var cb = this.async();

// welcome message
var welcome =
//... code purposely omitted

console.log(welcome);

var prompts = [{
  name: 'destPath',
  message: 'Where would you like me to put the color-me-sass files?',
  default: 'css/colors',
}];

this.prompt(prompts, function (err, props) {
  if (err) {
    return this.emit('error', err);
  }
  this.directory('colors', props.destPath);
  cb();
}.bind(this));
};
```

Code Walkthrough

Let's take a quick look at how this code works. First we `require` some Node.js modules (well, Yeoman's `generator-generator` did that for us!), and then that "Yeoman generator chain of execution" we talked about earlier kicks in:

- Our constructor gets ran and registers the all important `this.on('end'` *callback*. Even though our constructor runs early in this chain of execution, the `end` callback will fire only at the end once the other `prototype` methods have completed.

- `ColormeGenerator.prototype.askFor` than is ran. That's the section of code that deals with prompts. We only care where the user wants to place the color-me-sass files over to so we prompt accordingly.

- The magical `this.prompt` callback will fire once the user answers.

- Provided there are no errors, we then copy the files over to the `destPath` (where the user said they want the files to be placed):

```javascript
this.directory('colors', props.destPath);
cb();
```

53

After the files were copied over we called the `cb()` that was assigned earlier in that `askFor` method. This is the idiom provided for dealing with asynchronous callbacks if we need them within our generator's methods.

- Finally that `this.on('end'` callback we registered earlier the constructor fires and we simply provide a confirmation message and some usage advise. Notice those pretty standard output colors are available for us to use? Nice!

In a more comprehensive generator that creates a substantial scaffolding in the **app** *directory, we'd would have copied over Bower and NPM template configuration files and* **installDependencies** *would then be used to install these (I've left this in commented out).*

As you can see this is still a very simple generator, but it serves as a nice example of how easy it is to port a libray you'd like to include in your workflow.

Generator Tests

It's quite convenient to find that the `generator-generator` already creates a very nice template for creating tests specifically for generators. Since this is a fairly simple generator, I simply verified that the expected files were getting copied over in `test/test-creation.js`

```javascript
/*global describe, beforeEach, it*/
'use strict';

var path    = require('path');
var helpers = require('yeoman-generator').test;

describe('colorme generator', function () {
  beforeEach(function (done) {
    helpers.testDirectory(path.join(__dirname, 'temp'), function (err) {
      if (err) {
        return done(err);
      }
      this.app = helpers.createGenerator('colorme:app', [
        '../../app'
      ]);
      done();
    }.bind(this));
  });
  // Remove all the _*.scss color files copied over
  afterEach(function (done) {
    // Merely using testDirectory results in removal of files:
```

```javascript
    // https://github.com/yeoman/generator/blob/master/lib/test/helpers.js
    helpers.testDirectory(path.join(__dirname, 'temp'), function (err) {
      if (err) {
        return done(err);
      }
      done();
    }.bind(this));
  });

  it('generates expected files', function (cb) {
    var expected = ['_blues.scss', '_reds.scss'];
    helpers.mockPrompt(this.app, { destPath: '.' });
    this.app.options['skip-install'] = true;
    this.app.run({}, function () {
      helpers.assertFiles(expected);
      cb();
    });
  });

});
```

We can run this spec with:

```
mocha test/test-creation.js
```

Again, we're just touching the surface on what can be done with generators, but you can consult the Yeoman documentation to dig deeper.

Unicorn

In this chapter we'll take a look at Unicorn. TBD

```
rlevin@Robs-MacBook-Pro:~/programming/labs/generator-colorme(master|754m) $
mocha test/test-creation.js


          _-----_
         |       |
         |--(o)--|    .--------------------------.
         `---------´   |    Welcome to Yeoman,    |
         ( _´U`_ )     |   ladies and gentlemen!  |
         /___A___\    '_____'
          |  ~  |
        __'.___.'__
      ´   `  |° ´ Y `

     create _ambers.scss
     create _blues.scss
     create _bootstrap.scss
     create _browns.scss
     create _grays.scss
     create _greens.scss
     create _limes.scss
     create _oranges.scss
     create _peaches.scss
     create _pinks.scss
     create _preboot.scss
     create _purples.scss
     create _reds.scss
     create _tans.scss
     create _turquoise.scss
     create _whites.scss
     create _yellows.scss
Colors Installed!
You should now place something like the following in your _base.scss or similar:
@import "colors/reds";
@import "colors/peaches";
@import "colors/tans";
@import "colors/oranges";
@import "colors/ambers";
@import "colors/yellows";
@import "colors/limes";
@import "colors/greens";
@import "colors/turquoise";
@import "colors/blues";
@import "colors/purples";
@import "colors/pinks";
@import "colors/browns";
@import "colors/whites";
@import "colors/grays";
@import "colors/bootstrap";

But please do remove any extra @imports once you have determined color scheme!
.

  1 test complete (108 ms)
```

Figure 15: Running the generator's test-creation.js spec