INTRODUCTION TO BUILDING WEB APPS

BY SHAWN RIDER

Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower
Grunt Angular SASS JS HTML Git REST API
Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower
Grunt Angular SASS JS HTML Git REST API
Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower
Grunt Angular SASS JS HTML Git REST API
Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower
Grunt Angular SASS JS HTML Git REST API
Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower
Grunt Angular SASS JS HTML Git REST API
Node Yeoman Bower Grunt Angular SASS JS
HTML Git REST API Node Yeoman Bower

Table of Contents

Introduction to Building Webapps

This book is an introduction to the world of modern frontend web development tools and approaches. It covers the process of building a simple webapp from start to finish.

This book is for beginners. It tries to approach the process of building webapps with a focus on project and technical production skills.

Advanced developers, and others involved in the web industry, may find that much is missing here. We do not discuss usability, how one comes up with a winning app concept, the criteria that go into play when choosing which technology to build on, or myriad other factors that somebody building a webapp would likely agonize over.

This book takes up the philosophy that underneath it all these tools and frameworks have a lot in common. Learning to use any one set of tools will make it easier to learn new tools in the future. Likewise, understanding how to put the pieces together is likely to cause new developers to become curious about how those pieces got there.

This is the goal of a project-based approach: Readers should be able to reap immediate rewards from the instructions contained here, but they will also be led to deeper topics they can explore at length in the future.

What is in this book?

Throughout the series of projects in this book, we will build a webapp using modern frontend development tools. We will use the Yeoman workflow to provide our project bootstrapping, package management, build and deployment capabilities. This involves working with Grunt and Bower, and of course all of it runs on Node.js. These are solid tools used by many developers.

Once we are up and running, we will walk through a buildout of a simple application using AngularJS and a third party data API that can provide us useful information to share with our users.

What is not in this book?

This book is not a documentation of the ONE TRUE WAY of web development. It is a model for ONE WAY THAT WORKS. There are many OTHER WAYS THAT WORK, and those can also be very effective.

This book is not even, necessarily, trying to make the case that this is absolutely the best way to learn how to build webapps. That is another area without any dominant contender.

This book is not the last book you will ever need to read about making webapps, or web development in general.

Sincerest apologies if any of the above disappoints.

Prerequisites and Assumptions

Before you get going with this process, you should be aware that we assume you possess some knowledge already. If you do not feel comfortable with the subjects listed below, then you should probably revisit those topics in another forum (there are many options for learning these skills online for free).

If you are missing any of the technology described below, then you should definitely install whatever apps or tools are needed.

Basic HTML and CSS

It's essential for you to know basic HTML and CSS. You should be comfortable marking up content and writing styles using traditional methods. We will advance those skills to allow you to use more modern methods of putting together HTML and CSS so that you can build better webapps.

Basic Javascript

You should know how to use Javascript components on your pages, and how to write your own scripts to create the features you want. If you are at the "jQuery" level of Javascript usage, then that's probably about right: You can manage your variables, functions, objects, and logic; you can leverage plugins and other more complex things made by other people; you can follow instructions to implement somewhat complex features such as asynchronous data calls to a third party API using `jquery.ajax()` or similar.

This project will advance your Javascript skills and expose you to another way of thinking about how to organize your code and logic.

Basic command line

You should not be afraid of the Terminal app (although if you prefer to use iTerm2 you would not be alone). You probably already have CMDR installed on your Windows machine to ease the pain of using the Windows command line (and if you don't, then you might want to install that). You may have already tried working with Linux or Unix and seen what the command line looks like.

Modern web developers use the command line all the time because it allows us to easily manage the many components of our projects, and it mimics the environment our projects run in on a public server. Many developers love the command line and the speed and ease of use. Throughout this project we will regularly use the command line to manage our development tools.

Your preferred text editor

There are many opinions online about what editor is best. For this project, you do not need a "heavy" development tool (like Visual Studio or Eclipse). Many web developers prefer to use more streamlined text editors focused on providing a high quality code writing experience. You will need a text editor that you can use to edit files. This must be a plaintext editor (you cannot use Word or Pages).

Two popular choices of text editors today are Atom (a project sponsored by Github) and Sublime Text. Both editors are incredibly good, and either is an excellent choice. There are many other good choices out there, too, but this project is agnostic about which editor you use. As long as it allows you to comfortably make changes to text files, you should be fine.

Image and media editors / authoring

This project will benefit from lovely design and presentation. Lovely design and presentation often benefit from custom media (images, patterns, icons, video, etc.). Unfortunately, covering the ins and outs of how to make great media to go with a webapp would be a whole new book. This project does not specifically require any media making. For anything requiring images or other media files, you can use publicly available resources (and links to sites where you can find such resources are provided where they are referenced).

You are encouraged to learn more about making great media files, and explore the many options for creating media for your webapp. You are also encouraged to explore the many options available to buy or commission custom media work from artists and makers who do that for a living. (There's no shame in reaching out to others to make your projects as great as they can be. Remember that building webapps is a team sport, in spite of the fact that you may be doing this project on your own.)

Setting Up Your Development Environment

As developers, we may work on many different kinds of software projects. Each one will require us to use different components and tools to build out. This combination of components and tools, and the various utilities, scripts and apps that are required to run them all together, make up a "development environment".

It is best to maintain a fluid notion of what a "development environment" is, or what tools are in that environment, because it is likely to change for every project you work on. It will even evolve in long-term projects over time as new techniques become available.

Some developers are keen to push their development environments further and further, which is great for the rest of us. Invariably, those developers invent new ways of doing things that become so convenient that everyone wants to adopt their methods. But it's not crucial that you become highly skilled in conceiving and constructing complex development environments.

For many other developers, the development environment is something that should fade away to the background of their work process. They don't want to think about all the things that are going into making sites build and run: Rather, they wish to focus entirely on the project of building a great site or application. For these developers, it can never be too simple or too convenient.

It's good to know a little bit about what sort of developer you think you are: Do you like to fiddle with the pieces? Or do you want to focus on the code? This is related to another decision we must make very quickly: Do you want to work in a hosted environment? Or would you prefer to work "local"?

Hosted vs. Local

When setting up your development environment, you first have to choose whether you wish to work on your local machine (your laptop or desktop computer) or in a hosted environment (usually accessed through a web browser).

For many years, when we wanted to edit websites we logged in to a server and made changes using command line text editors like Vi and Emacs. Once desktop Linux became popular (which coincided largely with Macs converting to OS X, which is also a Unix-based operating system), it was possible for developers to begin working "local" -- on their own computers.

Working in a "local development environment" has a number of advantages: It allows for the fastest possible response from your test server (since that server is running on the same computer you are using), and it allows you to configure every small aspect of your environment. This is the way most developers work today.

The down side of local development is that it can often be complex, and, especially when working on a large project in a team setting, it can lead to wasted hours spent fixing development environments and keeping everyone running together. There are many tools and techniques that have been developed to mitigate those issues, so you may hear about things like "virtualization" and "containers" in relation to local development environments.

The alternative to local development environments is the hosted development environment. These services (like CodeAnywhere.com, Nitrous.io, and Cloud 9) offer hosted development environments that you can bring up with a few clicks of the mouse. They all offer some level of interface built on top of the development environment to allow you to make easier choices when provisioning your development environment.

On these systems, you typically create a "devbox" (or whatever creative term they use) with a base configuration. Those base configurations range from "HTML5" and "Python" to "Drupal" or "MEAN Stack". These configurations often have all the tools installed that we are installing during this first project.

How to Use This Book

This book assumes you are working in a local development environment. It provides all the information you need to get various components up and running.

If you are using a hosted development environment, then you can probably bring up a "Node.js stack" or "HTML5 stack" and have access to the core components that we need here. For most of the hosted development environments, you can also run installation commands if, for example, you can bring up a Node.js devbox, but it doesn't have Yeoman and related apps installed. The instructions provided here for installing those components should also work on your hosted development environment.

One area where there may be differences about how to set things up is when it comes to setting up Git and connecting your hosted development environment to your Github repository. You will likely need to consult the documentation for whatever hosted development environment provider you are using. (Almost all of them support connecting dev environments to Github.)

Background on Our Installation

We will work in what is known as a "Node-based" environment. That is to say, a development environment built on the Node.js platform. We will use the "Yeoman Workflow" to build our apps, and that will allow us to use several popular tools for frontend development, including Bower and Grunt.

We will also use Git and Github to manage our code and deploy our projects.

About Node.js

Node.js is a development platform that runs Javascript outside of the web browser. This is noteworthy because there are not other major implementations of Javascript outside of the web browser.

The specific Javascript engine Node.js uses is called V8, and it is the engine that is used in Google's Chrome browser. This is a very fast and reliable Javascript engine, and Node is known for being a very fast platform.

Node allows developers to create tiny Javascript applications called "modules", and then to string those modules together into larger and more complex applications. There are thousands of "Node modules", as they are known. Developers manage modules with a tool called NPM (Node Package Manager), which is installed when you install Node.js.

About the Yeoman Workflow

Yeoman is an application built on the Node.js platform, and it relies on two other programs to make it run: Bower and Grunt. All of these are Node.js modules themselves, so they can all be installed with NPM (in one command).

Each part of Yeoman Workflow provides a specific benefit to developers.

Yeoman

Yeoman is a templating tool. It provides a method for developers to create project templates, which are useful because they allow us to get "project skeletons" up and running very quickly. This process is often known as "bootstrapping" a site: bringing it up from nothing.

A project skeleton is a basic project structure that works, but without any customized features or functions. These are typically based on best practices approaches, and they involve the interlinking of several components.

In the world of Javascript there are many options for each component. With the Yeoman Workflow you are free to use any combination of components you wish, but you are also empowered with the ability to quickly create project skeletons that define the basic structure of a project.

This is useful in two ways: First, once you find a project template (or "generator" in the Yeoman lingo) that works for your development needs, you can easily re-use that template to start new projects. This allows you to get more quickly to the task of building new webapps.

Second, this is a useful learning tool because you can quickly bring up projects based on technology that you may not yet have mastered (like now). Although you have not made all the choices to put these components together, you are able to learn from the developers who have come before you. Once you learn your way around this blend of components, it will be easier to move on to new approaches and tools.

Bower

Bower is a package management tool, somewhat like NPM. It allows developers to define a "manifest" of frontend components a website makes use of. This might be jQuery or Bootstrap, or it could be AngularJS components and additions to support mobile hybrid apps. There are, as with NPM, many frontend development tools and libraries that support management with Bower.

This package manager allows you to more easily add and remove components from your website as you work through your development and maintenance. You can specify each version of the components you require, allowing you to maintain your site independent of when libraries or frameworks update themselves.

Grunt

Grunt is a build manager and task runner tool. It provides a mechanism for developers to define tasks that can be executed. A task is usually associated with some Node module, and it includes a list of configurations that tell the module how to behave in a given context.

For example, you may use Grunt to configure the `node-sass` module to process all of `.sass` or `.scss` files in your project and put the resulting `.css` files in a specific location. You might then combine that task with several others and roll it all into a command that you can execute by typing `grunt build`.

We will use Grunt in our projects to manage quite a few different tasks. As we work with it, you will come to understand the power of a tool like Grunt. There are many similar tools available for other types of development environment (`rake` in Ruby, `fab` in Python, etc.).

Please Note: You will also see another component to the Yeoman Workflow discussed: Gulp. Gulp is an alternative to Grunt that is gaining popularity. Rather than go into details about the differences between Grunt and Gulp, I'll only mention that both are very solid tools and if you wish it is entirely possible to complete this project with either tool. However, for the sake of simplicity, I am focusing on Grunt since it is the default tool that comes connected on most Yeoman generators.

About Git

Git is a source control software used by many developers for web and many other types of projects. Git is open source and free, which makes it available to everyone.

Git is also "distributed", which means each person who has a copy possesses the entire history of the project, and each developer could serve as a Git server for any other developer. This allows for a great deal of flexibility when working on projects.

Finally, Git is well-known for the way it handles branching and merging. Although these concepts will only be lightly addressed in the content of this book, these are important features used daily by most developers.

About Github

Github is a hosting service with enhanced tools to handle code repositories using Git. People are often confused about Git and Github. Think of the way Flickr is a hosting service for your images. Flickr is not a camera or illustration software -- it is a tool that allows people to host their images, organize them into galleries, and connect with others who are doing the same thing. Likewise, Github is a tool that allows you to host your code, work with it using helpful tools (like issue management, or wiki pages for documentation), and then connect with other people who are doing the same thing.

Github offers several services that go far beyond simple "Git repository management", and one of those features is called Github Pages. Github Pages is a static website hosting service, meaning that you can use Github to publish HTML, CSS and Javascript to the web and make it available for the general public.

Install Node.js

Installing Node.js is a straightforward process for most systems. On any platform, you can install Node.js using one of their pre-built installers available at http://nodejs.org/.

Please Note: Mac users should use Homebrew to install Node.js. DO NOT use the Node.js installer on a Mac unless you know what you're doing.

To install Node.js, simply download the installation package for your platform and install it like any other application.

However, for each operating system, you may consider an additional or alternative process.

Linux Considerations

Linux (or Unix) users may use whatever package manager they prefer to install Node.js. Node.js is generally available on all Linux/Unix package management systems.

Windows Considerations

Windows users may consider also installing the useful CMDR tool, which adds a lot of functionality to the default Windows command line terminal. This includes basic Unix commands that are missing by default from Windows as well as nicer coloring and prompts so you can have a more effective interface.

Since we will spend a lot of time on the command line during this project, it will be nice to have a better terminal experience.

Mac Considerations

Mac users should use Homebrew to manage packages like Node.js (and many other development tools). Homebrew is a popular package management tool for Mac and it makes installing many packages much easier. The Homebrew website (http://brew.sh) has a great installation tutorial to follow. Once you've installed Homebrew on your system, you can install Node.js with one command:

```
brew install node
```

Once that command completes successfully, you should have Node.js up and running on your Mac.

If you install Node.js using the installer on a Mac then it will likely lead to permissions errors later on and require you to use `sudo` way too much. Please use Homebrew to install Node.js on a Mac.

Install Yeoman and Friends

Once you have Node.js installed, getting Yeoman, Bower, Grunt (and even Gulp) installed is easy. You can do it all in one command by running:

```
npm install -g yo bower grunt-cli gulp-cli
```

Once that process completes, you can check on your installation by running the `yo doctor` command. The output should look something like this:

```
$ yo doctor

Yeoman Doctor
Running sanity checks on your system

✔ Global configuration file is valid
✔ No .bowerrc file in home directory
✔ No .yo-rc.json file in home directory
✔ NODE_PATH matches the npm root

Everything looks all right!
```

Now you are ready to begin bootstrapping projects from templates. Woo-hoo!

Install Git and Setup Github

It's likely that you already have Git and Github set up in your local development environment. We will use Git to version our code, and Github as both a server to host our repository and as a hosting platform to host our projects.

If you have not yet set up Git and Github, follow the links below to get these set up on your development environment.

Please note: Hosted development environments may connect to Github in a variety of ways. Please consult the documentation for your provider in order to properly connect your development environment to your Github account.

Installing Git

One of the easiest ways to install Git on your computer is to install the Github application. Here are links for the application on Windows and Mac:

- Github App for Windows
- Github App for Mac

It is recommended that if you are on a Windows or Mac machine and you have never set up Git before, you should start with the Github app. This app is useful to manage your code, and it also helps you set up your global Git preferences.

If you don't want to install the Github application, you may install Git directly using one of the downloads from the official Git website: http://git-scm.com/downloads.

If you choose to download Git directly and install it, you may wish to consult the "Setting Up Git" guide from Github.

Setting Up Github

You will need a Github account to use Github for the purposes of these projects. You may, of course, use another service for hosting your Git repositories, but this book will also leverage the Github Pages feature to deploy our application. It is probably essential to have a Github account to complete everything in this book.

Github has several great resources for learning how to use it and how to set it up. The trickiest aspect of setting up your Github account is adding your development computer's SSH keys to your Github account. This is covered in their guide: Generating SSH Keys.

In addition to setting up the basics, you may want to bookmark the Github Guides homepage as a destination for later. Github guides cover basic topics of using Github and Git, and they are very good learning resources.

Please Note: The Github guides that are linked above contain instructions for every operating system. Look for the platform links below the title of the page if it does not automatically select your platform.

Install a Generator

To test out our development environment, we should install a "generator". Generators are Node modules that tell Yeoman how to build out a project skeleton. There are hundreds of generators available. You can find a searchable index on the Yeoman website.

For our purposes, the Webapp generator will suffice. To install it, run the following command in your terminal:

```
npm install -g generator-webapp@1.1.2
```

Remember that NPM is the package manager that came with Node. That command tells your system to install the `generator-webapp` module. The `-g` flag tells it to install the module "globally" -- so it is available to run anywhere on your system.

Once that process finished running, you will be able to generate a new project skeleton based on the `generator-webapp` template.

Please Note: The latest version of the `generator-webapp` module was recently redesigned to use a task manager called `Gulp`. This book will eventually be updated to catch up to the switch to `Gulp` but for a variety of reasons it's beneficial to stick with `Grunt` for now. (`Grunt` is still used by the other generator we will use throughout this book, the `generator-angular` module.)

Create a Test Repo

For this and the next couple of sections, we will do some experimentation in a test repository before we get into our final webapp repository. In order to work through all of our experiments, we need to create a repository on Github.

Github has a great guide to creating repositories: Github Create a Repository Guide.

Follow that guide and create a new repository. (It should be empty except the README file.) Once you've completed that, return to your development machine and clone the repository to your development environment.

Clone Your Test Repo

If you are using the Github app for Mac or Windows, you can easily clone your new repository there. If you have any trouble, consult the appropriate Help guide:

- Github for Windows Help
- Github for Mac Help

If you are using a command line Git interface, you may use the following command to clone your repo. (Be sure to copy the SSH Clone URL from your repository home page and paste it into the command.)

```
git clone <SSH_CLONE_URL>
```

Once you have the repository successfully cloned to your local machine, you can move on.

NOTE FOR HOSTED DEV ENVIRONMENTS
If you are using a hosted development environment, it is very possible that you need to supply the SSH Clone URL when you first create your devbox. (This is the case, for example, with Code Anywhere.) Keep this in mind and adjust your process here to accommodate whatever your hosted service provider recommends.

Bootstrap a Test Project

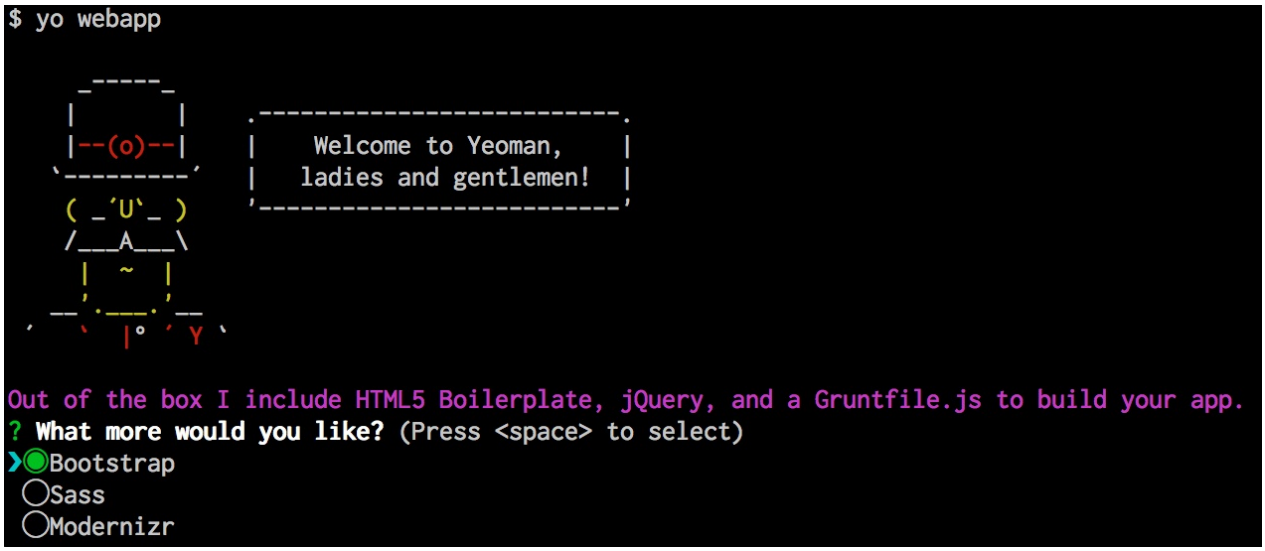
In order to get used to all the pieces of our new development environment, we will create a simple test project based on a common Yeoman generator template. The `generator-webapp` template puts the key pieces of the Yeoman Workflow into place, giving you an HTML file, Javascript files, SASS support, and more.

Getting this project up and running is not too difficult. First, you should **open your terminal and change directory into the directory where you cloned your test Github repository**. Do not run the Yeoman generators in random directories because you will end up with a mess of files. Please keep this in mind when generating your sites.

Run the following command to generate a project skeleton based on `generator-webapp`:

```
yo webapp
```

You should see the following information display:



Many Yeoman generators allow developers to select optional components and features. In this case, the generator is asking us if we want to use the Bootstrap CSS Framework, SASS CSS Preprocessor, and Modernizr, a Javascript polyfill that makes HTML5 sites work better on old browsers.

We want all of those in our project, so use the `up` and `down` arrow keys to move through the list, and the `spacebar` to select each option. Make sure each option has a bright green dot next to it. Once you've selected all the options, press `enter` to move on.

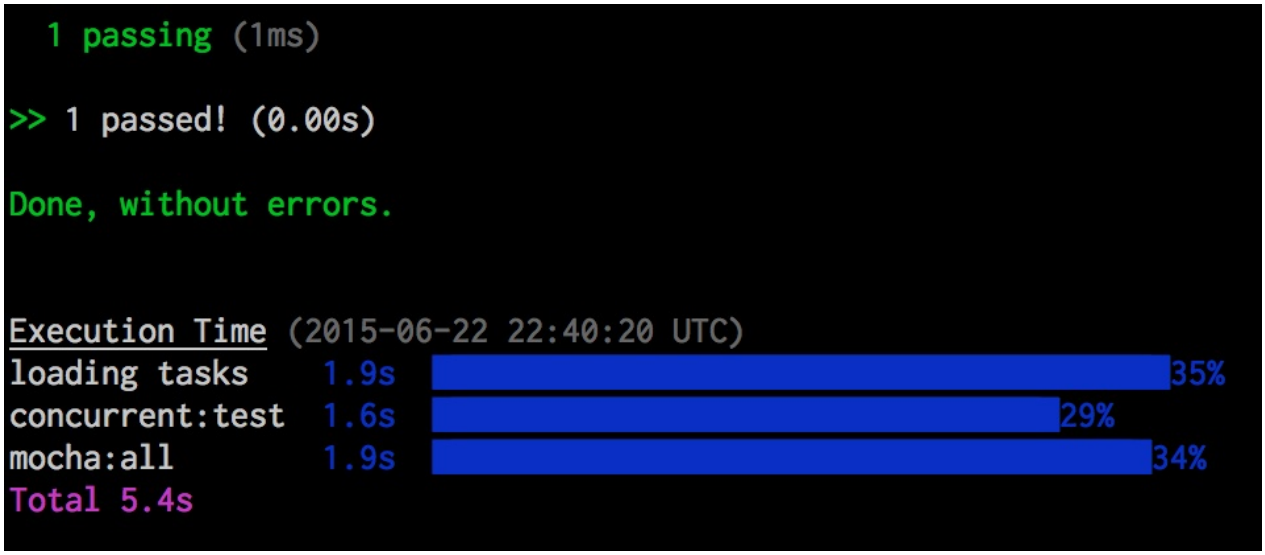
In some cases, you will need to answer additional questions. Whenever you use SASS, generators will often offer you the option of using a module called `node-sass` instead of the SASS Ruby Gem. Unless you have consciously installed the Ruby Gem, it's advisable to use `node-sass`, so when it asks us this question, we will say "y", we want to use `node-sass`.

Press `y` and then `enter` to begin the build.

Now that you have built out your project skeleton, it's a good idea to run the tests and see if everything is working. To do so, run this command in your terminal:

```
grunt test
```

That command should yield output that looks like this at the end:



If you see those results, then your site should run.

Running Your Local Dev Server

In order to work on your site you will, of course, need to run a local dev server. This can be accomplished by running the following command:

```
grunt server
```

That command will run the server in your terminal and will open your web browser to show you the default project template page.

The generator has created a set of directories and files that make up a project. There are many files in the system, but here is an abbreviated description of that structure:

```
.git/
.bowerrc
.editorconfig
.gitattributes
.gitignore
.jshintrc
Gruntfile.js
app/
bower.json
bower_components/
node_modules/
package.json
test/
```

Your website itself is located inside the `app/` directory. All of the files you will modify to make changes to your website are located in that directory. Here is what the layout of that directory and the files within looks like:

```
app/
|-- favicon.ico
|-- images/
|-- index.html
|-- robots.txt
|-- scripts/
    |-- main.js
|-- styles/
    |-- main.scss
```

You may now open up the index file in your `app/` directory within your project (`your-project-dir/app/index.html`). All of the files that form your webapp will be contained in this `app/` directory. Underneath the `app/` directory you will find `app/styles/` and `app/scripts` . In each of those directories you will find `main.scss` and `main.js` files respectively.

These three files, `app/index.html` , `app/styles/main.scss` , and `app/scripts/main.js` are the main files we will edit to customize this experiment to be our own site. For now, let's make some modifications to see how our auto-refresh preview works.

Making Changes

Open the `app/index.html` file in your favorite editor. Find the `<h1>` tag where it says "'Allo! 'Allo!" and change that to say something unique. Save the file.

Now return to your web browser. Your changes should update automatically so you see your altered text instead of the default Yeoman greeting.

Your local server is listening for changes to the files that make up your site, and when it detects a change it will update the page in your browser automatically. This can be very convenient.

This works with all your files. Open `app/styles/main.scss` and add the following style to the stylesheet:

```
h1 {
    color: green;
}
```

Now save the file and you should see that the `<h1>` on the page has now turned green.

If you look at the output in your terminal window, you will notice that Grunt is actually outputting all the stuff it's doing when you change files. It rebuilds the site almost instantly every time you save a file.

Commit and Push Your Changes

Once you've finished messing around with your new webapp a bit, commit and push our changes to your repository on Github. This way you can share your work, and you have an off-site backup. It's important to always commit and push your work before you move on to another task.

If you're using the Github app, then you will just need to go back into the app, commit and then "Sync" your changes.

If you're using command-line Git, here are the commands to commit and push your code.

First, you may wish to check the status of your repo:

```
git status
```

Then you will need to add changed files. You can add them one file at a time, or all at once like this:

```
git add -A
```

That command stages your changes, making them ready to be committed. Now we must make the commit, which sets these changes into the history of our repository:

```
git commit -m "YOUR MESSAGE HERE"
```

Finally, you must push your changes to Github. Since you cloned from Github, that server will be known as the `origin`. When you push it will automatically send your changes back to the `origin`.

```
git push
```

You should now be able to visit your repository on Github and see your changes.

Getting to Know Your CSS Preprocessor

Over the past several years, frontend developers have embraced CSS Preprocessors—tools that allow you to write and manage your CSS with more functionality.

CSS Preprocessors address three major issues when writing styles, adding functionality on top of normal CSS functions: How do we better organize our styles (both inside a single file and across multiple files)? How do we handle configuration data and re-used components? How do we ease the pain of repetitive sequences of style definitions?

There are many other doors that open and problems that are solved by trying to solve these three base issues.

For our project, we are using a CSS Preprocessor called `node-sass`, which allows us to use "SASS" (Syntactically Awesome Style Sheets). SASS, the specification, has been implemented into several different preprocessor tools (such as `node-sass` or the `sass` Ruby Gem). SASS and SCSS (Structured CSS) are terms that tend to be used interchangeably. Technically, SCSS is the piece that adds the hierarchical (nested) capabilities, while SASS is the specification that adds imports, variables and mixin capabilities.

Nesting: Style Organization and Hierarchy

Stylesheets are tricky to manage. This is primarily a result of the flexibility we have with implementing CSS and the "cascading" priority of styles. In order to determine which style applies to an element when there are multiple styles that target that element, CSS implements a notion of "specificity". The concept of specificity and the intricacies of how it works are well described in Andy Clarke's classic essay, "CSS Specificity Wars".

The trickiness with specificity is that it can be tough when you are in a huge file to keep track of all the style definitions. The fact that many stylesheets on large projects become files with thousands of lines of style definitions adds to the problem.

Stylesheets tend to grow to huge sizes because developers want to precisely control how styles go together. If developers break styles into multiple files, and then link stylesheets to the webpage in the traditional manner (with a `<link>` tag), then other issues can crop up in terms of page speed and style linking order (which, of course, can cause major issues with styles rendering correctly.)

CSS Preprocessors help with these issues in two ways. First, when you are writing styles inside a stylesheet, you can use "SCSS" hierarchies (also known as "nesting"). (**Please Note:** SCSS is a *superset* of CSS, so you can always just write "normal" CSS and that will work.)

Here is an example to show how normal CSS might compare to the SCSS version. First, the plain CSS example:

```
.my-container a:link,
.my-container a:active,
.my-container a:visited {
    color: red;
}
.my-container a:hover {
    color: green;
}
```

The equivalent SCSS code to the example above looks like this:

```
.my-container {
    a {
        &:link, &:active, &:visited {
            color: red;
        }
        &:hover {
            color: green;
        }
    }
}
```

As you work with hierarchical definitions of styles, you should also keep in mind the "3-levels Rule". This is a rule of thumb that you should avoid having your styles go more than 3 levels deep in hierarchy. This will help you in two ways: First, you will have more easily readable stylesheets; second, the styles that are produced by the preprocessor will not be as efficient as shorter styles. Remember that writing styles is always a balance of being specific enough and not over-specifying.

It's also easy to see how styles can be better organized within a file because we can use larger element styles (`.my-container` in the example above) to contain styles affecting the child elements.

The other way in which preprocessors help you with hierarchy and organization is via the `@import` command.

Imagine that you have a large web project and you have a stylesheet that has possibly thousands of lines of style definitions. If you look at those styles, you could probably group them according to what function they perform in your project (menus, buttons, content styles, etc.). Using your preprocessor, you can now break up those giant files into much smaller files.

Imagine that you have a set of style definitions in a file called `_buttons.scss` in the same directory as your `main.scss` file:

```
// filename: _buttons.scss

.button {
    .button-green {
        color: green;
    }
}
```

To include those styles in your `main.scss` file, you would use the `@import` command:

```
@import('buttons');
```

This is better than using traditional imports because the files are combined the way you want them to be before they are ever delivered to users. With traditional imports, the files are downloaded separately by the user.

Most of the time, the `main.scss` file becomes an index of imports. It might look something like this:

```
// CSS Style Resets
@import('reset');

// Site Components
@import('buttons');
@import('typography');
@import('content_styles');
...
```

To reflect the files mentioned in the example above, your styles directory would look like this:

```
styles/
|-- _buttons.scss
|-- _content_styles.scss
|-- main.scss
|-- _reset.scss
|-- _typography.scss
```

Please note: The underscores preceding some of the filenames indicate that those files are "partials". This is a convention in SCSS, so you will see it often. Also notice that you do not need to include the underscore or the `.scss` filename extension in the `@import` statement.

Variables

Variables are a key piece of creating flexible, consistent systems. Rather than rewriting key information several times across many files, variables allow us to create an information reference we can use everywhere.

It can be difficult to imagine how useful this will be, but here is an example. One of the most important aspects of any site design is the color palette. In normal CSS, we must define colors as words, hex codes or RGB values. Each time a color is used, we must input the same value. This can be tricky, especially when developers get lazy and use tools to select colors from web pages. Often the tool selects a slightly different shade of the color, so you may see stylesheets that reference several slightly different variations of a color.

Traditionally, we might see styles like this:

```
.my-class { color: #aa0000; }
.my-background { background-color: #ab0010; }
.my-success-link { color: green; }
```

If we are using SASS, we can create variables to represent colors:

```
$brand-primary: #aa0000;
$brand-success: green;
...
.my-class { color: $brand-primary; }
.my-background { color: $brand-primary;}
.my-success-link { color: $brand-success;}
```

In the SASS example, we could easily change the values of the variables wherever they are defined, and that change would then ripple through all of our styles. So if the `$brand-primary` color changed to "blue" then it would have an immediate and consistent effect on the site styles.

Colors are just one vivid example where having the ability to define variables would be very useful. There are many other situations where defining variables is incredibly useful and helpful for maintaining flexibility and responsiveness in your styles.

Mixins (Functions)

Another major issue facing frontend developers working on defining styles is repetitive code that needs to be retyped, sometimes with very slight modifiers, over and over again. In most programming, we handle this with "functions". The functions implemented in most CSSS authoring frameworks (like SASS) are called mixins.

A common use case for using a mixin is when using a feature that is supported by all your target web browsers, but which is still only accessible with a "vendor prefix". (This is often the case when features are nearing finalization, but can sometimes be a situation that can last for years.) If we wanted to get rounded corners on our boxes, mixins allow us to write something like this:

```
@mixin border-radius( $radius ){
    -webkit-border-radius: $radius;
    -moz-border-radius: $radius;
    -ms-border-radius: $radius;
    border-radius: $radius;
}

.rounded-box {  @include border-radius( 10px ); }
```

This approach allows us to write one line of code to invoke the mixin, rather than the four lines of style attributes we would need to use. Note that the mixin also accepts a variable (`$radius`), which means we could use this mixin to generate any size of rounded corners.

These three enhancements are powerful enough to have made CSS authoring frameworks a standard part of the frontend development toolkit over the past few years. The future of the CSS standard has no indication that any of these features are going to be rolled into CSS proper in the near-term, so these frameworks and techniques are likely to remain a part of our work.

Imports: Set Up Your Stylesheets

In order to make the most of our new CSS preprocessor, let's set up our stylesheets for further development. We need to establish our file structure and set up our imports to allow us to write clean, well-organized code that makes the most of the SCSS nesting capabilities to create an obvious hierarchy.

Locate your stylesheet

As we saw when we bootstrapped the site, the stylesheets for our project are located at `app/styles/`. As part of the bootstrapping process, a `main.scss` file was created and some styles were added there to support the project skeleton homepage.

Open `app/styles/main.scss` and observe the styles that have been pre-created for you. There is a `.browsehappy` style, which is an element that shows up when outdated browsers hit your page. (You can find that element, and the code that makes it appear, in the `app/index.html` file.)

You will also see several styles that affect the content inserted into `app/index.html` for demonstration purposes. These show you how SCSS can be written, and you should start editing with just altering some of the values to experiment.

If you are running your server as you edit, you should see your page automatically update in the browser whenever you save your changes. This can be very handy for rapidly working on style definitions. Of course, what's happening is that when you run `grunt server`, there is a `watch` task that runs. The `watch` task monitors files in your app for changes, and when a change is detected it kicks off the whole build process.

Pretty neat, right?

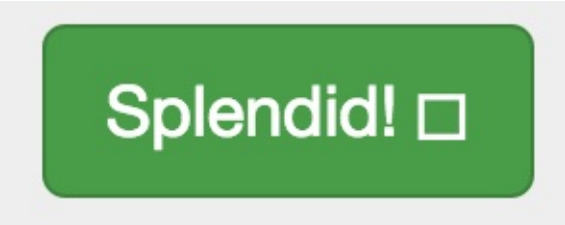
So although these style definitions are useful (you might also pay special attention to the way they write their media query to target mobile devices), they are not necessary. You may remove any of the styles that you wish. In general, it's not recommended to put styling directly into your `main.scss` stylesheet.

Icon Font Path

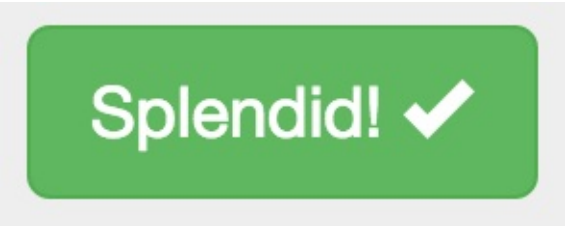
The first line of your generated stylesheet is probably a variable definition:

```
$icon-font-path: "../../bower_components/bootstrap-sass-official/assets/fonts/bootstra
p/";
```

This definition may not be correct. The path should point to wherever the Glyphicon font that comes with Bootstrap is located. You may not have noticed if the path is broken, but you can tell by looking at the button in the default web content. If you see a square, like this, then it is broken:



When the `$icon-font-path` variable is properly set, then the fonts work and the checkmark icon is visible:



Adjust the `$icon-font-path` variable so it points to the directory containing the Glyphicon fonts. (You may need to add an extra `../` to point to the proper directory--or modify the path further if you have altered your default locations.)

Bower-managed area

The next section of code in `main.scss` (starting on line 2) is the Bower-managed area. Leave these comments and import statements in place so that Bower can add components to your project properly. These lines probably look like this:

```
// bower:scss
@import "bootstrap-sass-official/assets/stylesheets/_bootstrap.scss";
// endbower
```

Don't modify the comments or the code between the comments those lines will be managed by Bower if/when we add frontend components to this project.

Setting up your `main.scss`

Your `main.scss` stylesheet should be an index of all your site's stylesheets. It should consist of a list of imports (with helpful comments about them if need be). You should strive for a clear, understandable structure. There are different philosophies about how to manage your styles and stylesheets, but they all generally boil down to managing two things:

1. Making sure styles are named and organized in a way that makes sense. Styles that affect the same objects should go together. Styles that belong to the same part or feature of the site should go together. As much as possible, we wish to avoid making any surprises or confusion for other developers who may come along.
2. Making sure individual stylesheets do not become overly cumbersome because they are way too long.

Organizing files

Some sites might organize styles by specific function, something like this:

```
/app/styles/
|-- main.scss
|-- _buttons.scss
|-- _content.scss
|-- _layouts.scss
|-- _typography.scss
...
```

This approach works fine, especially when there aren't too many styles. Once a site grows very large, it may be beneficial to break up styles into other types of organization:

```
/app/styles/
|-- main.scss
|-- components/
    |-- _buttons.scss
    |-- _menus.scss
    ...
|-- modules/
    |-- _blog.scss
    |-- _checkout.scss
    ...
|-- _reset.scss
```

There are more formal methods of organizing styles (such as SMACCS), and there are many, many hybrid and ad-hoc methods as well. It's worthwhile to spend a little time thinking about what makes sense in terms of organizaing your files.

Make some stylesheets

Since we are just experimenting and we don't need too much in terms of styles, we will do a minimal stylesheet setup. We will create the following file structure for our `app/styles/` directory:

```
/app/styles/
|-- main.scss
|-- _content.scss
|-- _variables.scss
```

Create the `app/styles/_content.scss` file and add the following style definition so we will know when our new stylesheets have taken effect:

```
body {
    color: green;
}
```

Now you need to create `app/styles/_variables.scss`. To make this file, we're going to copy a file out of the Bootstrap files that Bower installed for us. In order to customize Bootstrap, a `_variables.scss` file is provided. This file sets values that we can alter or override to make Bootstrap fit our needs better. However, we **DO NOT** want to edit the `_variables.scss` file in the Bootstap module. We **must** make our own copy of `_variables.scss` in our `app/styles/` directory.

Find the Bootstrap variables file. It should be in:

```
bower_components/bootstrap-sass-official/assets/stylesheets/bootstrap/_variables.scss
```

(Note: The `bower_components` directory should be at the root of your project.)

Copy this file into your `app/styles/` directory. Once you have done so, open the file for editing. Find the line that defines `$brand-success` (probably line 22), and alter it like so:

```
$brand-success: #AA0000;
```

This turns the color for "success" styles in our project a dark red. Once we get these stylesheets connected to `main.scss`, you will see that the body text turns green and the button turns red.

Setting up your imports

Now that we've set up our new stylesheets, let's modify `main.scss` to import those files. We will first add a line above the Bower-managed area that imports the `_variables.scss` file:

```
@import "variables";
```

Please Note: It's important to remember that in order to properly override the variables in Bootstrap, you must import your `_variables.scss` file **BEFORE** you import the `bootstrap.scss` file (which is what Bower has done for you).

Next, we will add an import below the Bower-managed area to include our `_content.scss` stylesheet:

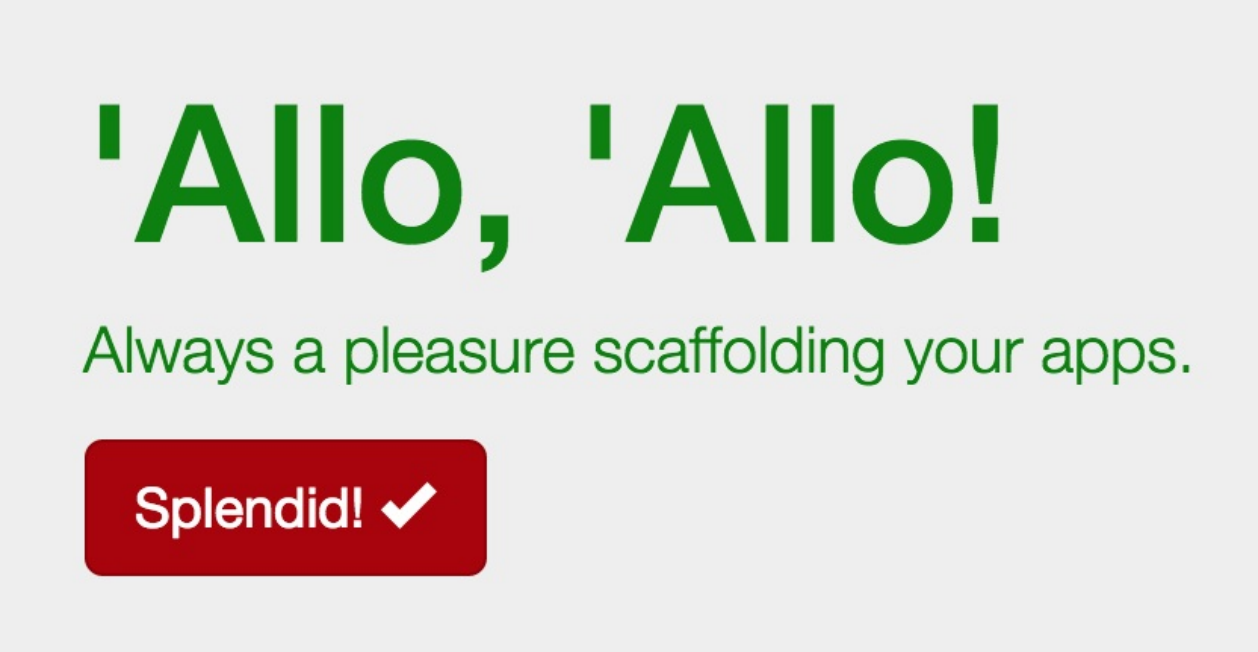
```
@import "content";
```

Once we're finished, and with some helpful comments added, our `main.scss` stylesheet should look like this:

```
$icon-font-path: "../bower_components/bootstrap-sass-official/assets/fonts/bootstrap/"
;

// Override Bootstrap variables
@import "variables";

// bower:scss
@import "bootstrap-sass-official/assets/stylesheets/_bootstrap.scss";
// endbower

// Site-specific styles
// Add additional stylesheets below
@import "content";
```

Remember: When importing stylesheets in SASS, you do not include the underscore (`_`) or `.scss` filename extension.

Once you save this file, your server should automatically update and you should see that the text on your page is green and the button has now turned dark red:


'Allo, 'Allo!
Always a pleasure scaffolding your apps.
Splendid!

Experimenting with Nesting

As mentioned before, SCSS allows you to use hierarchical nesting structure to create more specific styles. This can help you keep your files more organized (because styles that go together can be contained together), and it helps you overcome any style selector conflicts.

When nesting styles, you may define selectors within selectors. We saw the basic example earlier.

This SCSS:

```
.my-container {
    a {
        &:link, &:active, &:visited {
            color: red;
        }
        &:hover {
            color: green;
        }
    }
}
```

Makes this CSS when it is processed:

```
.my-container a:link,
.my-container a:active,
.my-container a:visited {
    color: red;
}
.my-container a:hover {
    color: green;
}
```

All of the styles above are contained within `.my-container`. You can see how the `a` selector (which selects the `<a>` elements inside `.my-container`) is defined within the curly braces marking off the `.my-container` style. This is the "nesting".

The ampersands (`&` symbols) reference the parent selector, and modify it directly. Thus, `&:link` becomes `a:link`. If it were written without the ampersand (`:link`), the result would be `a :link`, which would be an invalid selector. You can read more about referencing parent selectors on the SASS Documentation site.

Try it out

In order to try this out, let's define some styles for the `.marketing` section of the demo page. Add the following styles to your `_content.scss` stylesheet:

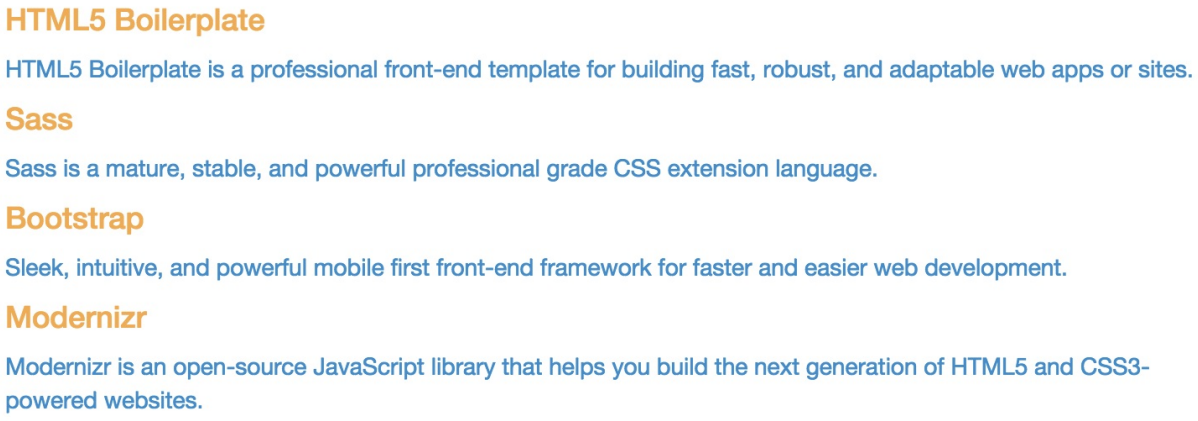
```
.marketing {
    h4 {
        color: $brand-warning;
    }
    p {
        color: $brand-primary;
    }
}
```

This code specifies that the `<h4>` elements inside the `.marketing` element should take on the `$brand-warning` color (which is a yellow), and the `<p>` elements inside the `.marketing` element should become the `$brand-primary` color (which is blue).

The CSS that will be compiled from this SCSS will look like this:

```
.marketing h4 { color: #f0ad4e; }
.marketing p { color: #428bca; }
```

And the result in the browser should look like this:



Customizing Variables

Since we're already working with Bootstrap, it makes sense to leverage as much of it as possible. This is only possible if we use the `_variables.scss` file to customize the variable values for our project.

The values in `_variables.scss` control everything from the color palette for the site, the font families and type settings that are used for your site, default style settings for buttons, inputs, and even the specific dimensions at which the Bootstrap grid will break down for different screen sizes.

Colors

The first section of the `_variables.scss` file defines colors. Notice the way the gray colors are defined:

```
$gray-darker:            lighten(#000, 13.5%) !default; // #222
$gray-dark:              lighten(#000, 20%) !default;   // #333
$gray:                   lighten(#000, 33.5%) !default; // #555
$gray-light:             lighten(#000, 46.7%) !default; // #777
$gray-lighter:           lighten(#000, 93.5%) !default; // #eee
```

First of all, variables in SASS begin with a dollar sign (`$`). Second, these color definitions are using the `lighten()` mixin. There is usually not much reason to change the gray values, but you are free to do so if you wish.

The next set of definitions is much more impactful for branding our site:

```
$brand-primary:         #428bca !default;
$brand-success:         #aa0000 !default;
$brand-info:            #5bc0de !default;
$brand-warning:         #f0ad4e !default;
$brand-danger:          #d9534f !default;
```

You may alter these colors however you desire. You can use hex codes to define the color, or `rgb()` and `rgba()` style definitions. You may also use the `lighten()` and `darken()` mixins.

Typography

The next section of `_variables.scss` worth noting is the Typography section. This defines the way fonts look on your site. The main font definitions come first:

```
$font-family-sans-serif:  "Helvetica Neue", Helvetica, Arial, sans-serif !default;
$font-family-serif:       Georgia, "Times New Roman", Times, serif !default;
//## Default monospace fonts for `<code>`, `<kbd>`, and `<pre>`.
$font-family-monospace:   Menlo, Monaco, Consolas, "Courier New", monospace !default;
```

You may alter these to use whatever fonts you have available (including any font you have made available through a webfont method like Typekit, Google Fonts, or just a regular old `font-face` definition).

In addition to the fonts themselves, you'll find the following two variables, which are important for dicating how type appears on your site:

```
$font-size-base:          14px !default;
$line-height-base:        1.428571429 !default; // 20/14
```

These values modulate many aspects of how your site appears. If you generally want to increase or decrease the size and spacing of your fonts, you should make those changes here, in the `_variables.scss` file.

Additional Variables

The `_variables.scss` file is very long and defines many settings for Bootstrap. If you want to play with these in a more interactive format, the Bootstrap Live Customizer tool is a great way to see the effect your variable changes can have on a theme.

Whenever you find yourself working against Bootstrap, it's worthwhile to check if the thing you're trying to do could be accomplished by modifying a variable. Often developers find themselves working against their frameworks because they do not realize the flexibility that has been built into the system. Take the time to get to know some of these options and play with making changes to see how you can affect the look and feel of your site.

Making the Most of Mixins

Mixins are powerful tools, and you should consider them in two ways:

1. You can create a mixin to handle any repetitive code generation task that can be defined with logic. The mechanisms provided to create, essentially, "CSS functions" are very handy.
2. You can use mixins created by others, and if you're working with a CSS framework, then it is likely to offer you mixins to use.

Since we're working with Bootstrap, we have lots of mixins at our disposal. We can learn from them. We can also take existing mixins and modify functionality to create our own.

Most of the time, we keep mixins in a separate file (or multiple files) so we don't get them lost in our stylesheets. Bootstrap stores its mixins in a `mixins/` directory containing many files.

You should be able to find the Bootstrap mixins directory in your project in:

```
bower_components/bootstrap-sass-official/assets/stylesheets/bootstrap/mixins/
```

As you look through that directory, you will notice many mixins. You can use the mixins in the `_grid-framework.scss` file to easily add grid-like features to your own styles. Or the mixins in the `_gradients.scss` file to create background gradients for your own styles. Let's examine the `_border-radius.scss` file to see the four mixins designed to make it easier for you to apply custom `border-radius` styles:

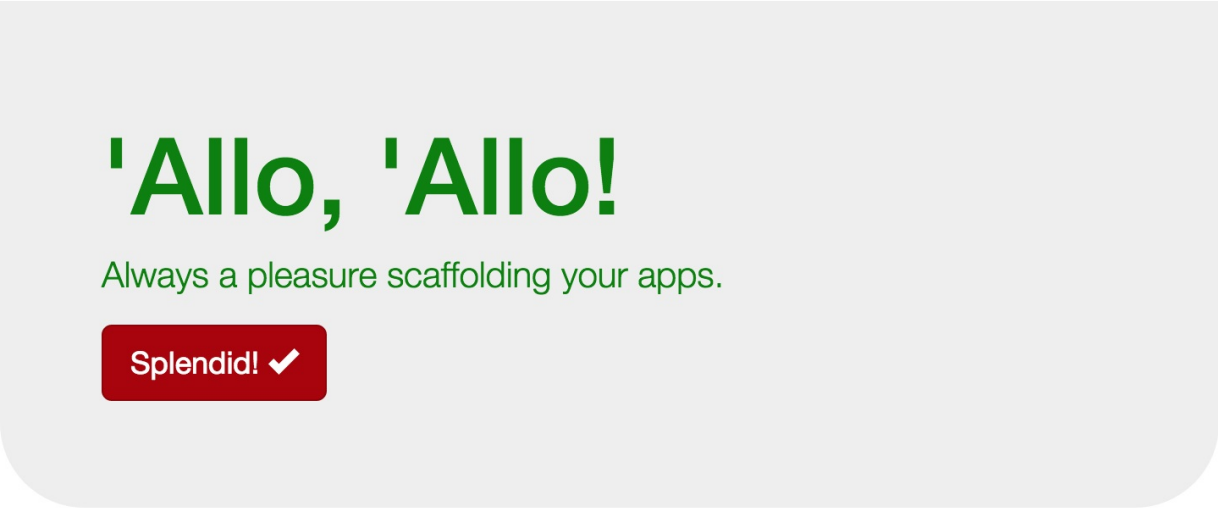
```
@mixin border-top-radius($radius) {
  border-top-right-radius: $radius;
   border-top-left-radius: $radius;
}
```

You can see the basic structure of a mixin here: The `@mixin` keyword denotes that this is a mixin definition (similar to when you use the `function` keyword in other programming languages). The name of this mixin is `border-top-radius` and it takes a value called `$radius`. That value is then used in the two lines that define the `border-radius` for the top corners of an element. Using this mixin, we could add a style definition to `_content.scss` to alter the gray jumbotron box on our web page.

Add this style to your `_content.scss` file:

```
.container .jumbotron {
    @include border-top-radius(0px);
    @include border-bottom-radius(50px);
}
```

In this selector, we are targeting the gray jumbotron box, and we are using the `border-radius` mixins we just looked at. The `@include` keyword is used to invoke the mixin, then the mixin signature is provided. In this case, we are setting the top radius so it will be squared off, and the bottom radius will be dramatically rounded, making the whole thing look a little bit like a boat:



Tips for using mixins

Since mixins can be as varied as the developers creating them, you should take time to read through a mixin and understand what it is doing. Mixins can generate a lot of code, so sometimes you don't realize how much of an impact you are having when you use an unfamiliar mixin.

Mixins also will each have a different set of parameters they expect. In the example above, the mixin requires us to send a measurement that will be compatible with a `border-radius` style attribute. Since CSS attributes vary so widely, it can be confusing to provide the right data when calling a mixin. This is another reason it's important to read through the code and see what's going on.

Whenever you find yourself writing the same sets of style attributes in a project, consider whether or not it would be worthwhile to create a new mixin. Defining a simple output mixin can be as easy as this:

```
@mixin simple-output-mixin() {
    color: $brand-primary;
    text-decoration: underline;
}
```

That mixin would allow me to define a style like this:

```
.my-content {
    @include simple-output-mixin();
}
```

And that would generate CSS that looks like this:

```
.my-content {
    color: #AA0000;
    text-decoration: underline;
}
```

If you can save yourself from re-typing lines of style defnitions over and over, then it is well worth the effort to create the mixin.

Finishing Up Our Style Experiments

Keep experimenting with writing styles and modifying this site until you get comfortable with the core features of SASS and SCSS. It can be tricky to get into these techniques at first, but they will save you time and make edits much quicker in the future.

Once you have completed your experiments, don't forget to commit and push your changes using Git:

```
git add -A
git commit -m 'Experimenting with styles'
git push origin
```

Or, if you're using the Github Application, just go into that tool, provide a commit message, then click the "Commit and Sync" button.

Additional SASS/SCSS Learning Resources

This book does not go terribly in-depth on how to use SASS and SCSS to create styles (you pretty much just read the bulk of our styles discussion). For the most part, you will apply all of the CSS knowledge that you have to your stylesheets as you normally would. But as you work through the tasks prescribed here, you may want to check out these resources for additional guidance and insight:

- Official SASS Guide
- Getting Started with SASS by David Demaree
- The Beginner's Guide to SASS by Andy Leverenz
- On SASS and Other CSS Preprocessors by Danny Smith

Building and Deploying Your Project

Now that we have a project created and we've made some changes to customize it, we now need a method of deploying that project. The process of preparing ("building") your project for deployment and then actually deploying it is the final step toward making your project available to users on the web.

There was a time when web projects often separated what were known as System Admin (sysadmin) or System Operator (sysop) duties and developer duties. The admins and ops would make sure servers were running, properly configured, and properly connected into the network. The system of servers, routers, switches and other hardware that keeps any given network center operational is known as the "network infrastructure". When companies bought servers and put them in specially designed rooms connected to lots of other equipment, it was necessary to have teams of people managing that equipment.

These days, most companies do not own their own network hardware. As the needs of websites have become both more diverse and more standardized, it is easier and easier to leverage third party solutions to keep a website or application running. Physical hosting of servers in a Network Operation Center (NOC) has given way to cloud-based hosting of virtual servers.

This evolution has transformed the way developers interact with their work. We are in the midst of the "devops" era. Devops is a term that combines development of a project (website or application) and the management of how that project is deployed and served to users. Since the optimization and securing of any project so often depends on specific details of the project, it makes sense that these two sets of tasks would come together. However, the process of building and deploying sites is still time consuming and warrants some consideration.

Tools for the Job

As with everything in web development, we begin our work here by choosing the tool we will use to help us achieve our goals. In this case, the goal is to build and deploy our static website to the Github Pages static website server. This service is free and easy to use, so it's great for our work here. You should be able to use this same approach with many other hosting solutions by modifying some of these details.

To manage bringing together all of the pieces, we will use Grunt. Grunt is a default component in the webapp generator (and the AngularJS app generator we will use later), so it is easy to set up and work with in our project. (It's worth noting, as we have earlier in this book, that Gulp is a viable alternative to Grunt. Although the specific details of implementing Gulp are quite different from implementing Grunt, the general concepts and approach here can be done with Gulp.)

Our project came with almost all of the smaller components Grunt needs to handle the "build" part of our "build and deploy" process. It is already building your site for preview when you alter files and preview them on your local development server. It is processing your SCSS into CSS using `node-sass`, and it is using a variety of tools to minify, compress, optimize and otherwise make your site happen. The good news is that we do not have to alter these processes at all, but as we work through setting up Grunt to handle our deployments we'll take a look at how all of these things are configured, too.

Grunt Build Control

The one thing our Grunt setup is missing out of the box is a way to actually deploy the built files to a server. In order to make this happen, we will use a component called `grunt-build-control` (Github repo). We will configure this component in our `Gruntfile.js` so it builds our project to a `gh-pages` branch, which Github will then automatically pick up and publish for us.

This will allow us to run a single command to publish our site, like this:

```
grunt buildcontrol
```

Of course, this will require a little effort to set up. In this chapter we will work through the process of adding `grunt-build-control` to our project. We will explore how to use `npm` to add local modules specific to our projects, and we will look at how the `Gruntfile.js` works to configure `grunt` tasks. Finally, we will publish the experimental project we've been working on to Github Pages and verify it is available to the public.

Install `grunt-build-control`

The `grunt-build-control` component is a Node.js module you can install with `npm`. In order to install it, open your terminal and change directory to the root of your project repository. Run this command to install `grunt-build-control`:

```
npm install grunt-build-control --save-dev
```

That command tells `npm` to install the `grunt-build-control` package, and it also uses the `--save-dev` flag to tell `npm` to add this module to our `package.json` file (which is the file `npm` uses to track the Node.js modules we are using in this project). By using `--save-dev` we make sure that the next developer who clones this repo and installs the dependencies will get `grunt-build-control` alongside everything else.

The installation will finish with showing you the version of `grunt-build-control` that was installed and the dependencies it had:

```
grunt-build-control@0.5.0 node_modules/grunt-build-control
├── semver@4.3.6
└── shelljs@0.2.6
```

There is not much fanfare when you are successful here, but you will receive a message if `npm` runs into an error. Verify you have not received an error message (you don't need to worry about `npm` warnings about "description" or "repository field") and you can move on.

Anatomy of a Gruntfile

We use the file `Gruntfile.js` to configure Grunt tasks. Tasks are specific commands we can run using Grunt. We can configure tasks with whatever names we want, and we can combine tasks however we want, so there is a lot of flexibility in terms of what we can do with Grunt to help us accomplish more with our development processes.

Grunt is JS

The first thing to remember as you are editing your `Gruntfile.js` is that, as the filename extension indicates, this is a Javascript file. You need to obey the rules of Javascript and have your Javascript hat on when dealing with Grunt configurations. Trailing commas, loose management of curly braces, and bad use of quotes can cause you to have syntax errors just like any other piece of Javascript. Although we don't typically do a bunch of logic in the `Gruntfile.js`, it's still a Javascript file.

Organization of `Gruntfile.js`

The Grunt configuration begins on the line that starts:

```
module.exports = function (grunt) {
```

This entire file is exporting a `function` that can be executed by Grunt. Inside this function, we declare some variables, include some additional modules via `require` statements, and then initialize Grunt with a configuration that defines all of our tasks.

Starting right below the `module.exports` line, you should see a couple of require statements:

```
// Time how long tasks take. Can help when optimizing build times
require('time-grunt')(grunt);

// Load grunt tasks automatically
require('load-grunt-tasks')(grunt);
```

Those lines are telling Grunt that we are also using these other Node.js modules. (Later on we will add `grunt-build-control` to the list.)

Below those lines comes the declaration of the `config` variable:

```
// Configurable paths
var config = {
    app: 'app',
    dist: 'dist'
};
```

This variable is crucial for Grunt, especially in our context. We are telling Grunt where our application is located as well as the name we wish to use to contain the built website files. These two definitions are used throughout the configuration of the other tasks in the Grunt file.

Finally, we come to the beginning of how we will configure our Grunt tasks, using the `grunt.initConfig()` method:

```
// Define the configuration for all the tasks
grunt.initConfig({
    // Project settings
    config: config,
```

All of the rest of this file exists within that `grunt.initConfig()` call. What follows is a massive JSON object defining all the settings for the different tasks we are using in our project. You can see the different commands we run on the command line and how they are configured.

Task definitions

For the most part to truly understand what each configured task is doing, and how to modify the functionality, it's essential to actually visit the documentation for that module and learn about the configuration options. Each module is configured slightly differently.

As an example of what a task configuration looks like, we can look at the default `mocha` task definition created as part of the project template:

```
// Mocha testing framework configuration options
mocha: {
  all: {
    options: {
      run: true,
      urls: ['http://<%= connect.test.options.hostname %>:<%= connect.test.options.por
t %>/index.html']
    }
  }
},
```

Mocha is the test running library being used in this project. This module allows you to run any tests you've created to verify the functionality of your website. The specifics of this module are not important here since we are just looking at what the configuration looks like.

You can see that the tasks are referenced by the "key" in the JSON object, so this task would be run with `grunt mocha` if you wanted to run it all alone. `mocha` references an object that contains the configuration for the `mocha` module in a structure that has been defined by the developers who created the `mocha` module.

You can note that in the definition of the `urls` property, the configuration specifies an array of strings. Those strings are leveraging a templating format that is allowed inside Grunt configurations. This templating allows us to reference the values of other Grunt tasks or components, or even the `config` object that we defined previously.

Getting comfortable

Read through the task definitions and try running some of the tasks on their own in your console. Some of them will not work properly when run alone because they need to work as a part of a bigger sequence. For example, in practice we would probably never run `grunt mocha`, but we would run `grunt test`, which in turn references and executes the `grunt mocha` task.

Note that the `copy` task is the task that controls copying files into your final build. You may, for example, discover that if you need to add a subdirectory to your project, you will need to specify that directory to be copied, too. For performance and control reasons, we typically write our `Gruntfile.js` files to be very specific about what to copy and where. When you add to or alter your projects file structure you will likely need to update some of these configurations.

Spending a little time reviewing and experimenting with these pre-existing Grunt tasks is a great way to get comfortable with this tool. You can do a lot with it, so keep it in mind when you encounter troubles or discomfort in your development process.

Configure `grunt-build-control`

In order to use `grunt-build-control` you will need to configure it. This process involves configuring a JSON object to define the options and settings we wish to use for our project.

Please Note: `grunt-build-control` can be used to deploy to a variety of places. We are only going to explore one method of configuring this tool. Be aware that you may be able to use this tool for other purposes, too.

Let Grunt know that `grunt-build-control` is available

Recently the Yeoman Webapp Generator has been updated to use a new module called `jit-grunt` to manage dependencies for your Grunt tasks. If you see this in your `Gruntfile.js` (near the top), then your project is using `jit-grunt`:

```
// Automatically load required grunt tasks
require('jit-grunt')(grunt, {
  useminPrepare: 'grunt-usemin'
});
```

If those lines exist in your `Gruntfile.js` then you will need to add a line to let Grunt know that the `buildcontrol` task uses the `grunt-build-control` module like so:

```
// Automatically load required grunt tasks
require('jit-grunt')(grunt, {
  useminPrepare: 'grunt-usemin',
  buildcontrol: 'grunt-build-control'
});
```

Please Note: If your Gruntfile.js does not use `jit-grunt` then you can ignore this step and continue to configure your `buildcontrol` task below.

Configure the `buildcontrol` task

In order to use `grunt-build-control`, it's necessary to create a JSON object to configure the task. The order of your tasks in the `grunt.initConfig()` doesn't matter, but it's important you do not break another task configuration. To keep things easy, I advocate placing this

definition right before the definition for the `watch` task (which is what Grunt uses to know when you've changed files when you are running the local server).

This should looks something like this:

```
  // Define the configuration for all the tasks
  grunt.initConfig({

    // Project settings
    config: config,

    buildcontrol: {
      options: {
        dir: 'dist',
        commit: true,
        push: true,
        message: 'Built %sourceName% from commit %sourceCommit% on branch %sourceBranc
h%'
      },
      pages: {
        options: {
          remote: 'git@github.com:your_github_user/your_webapp.git',
          branch: 'gh-pages'
        }
      }
    },

    // Watches files for changes and runs tasks based on the changed files
    watch: {

    ... more code ...
```

Let's look more closely at the configuration we just specified. First, we define general `options` for the `buildcontrol` task to use every time it runs: It will build to the `dist` directory. It will make a commit inside that directory and push it if need be. The message it uses in the commit will be the one specified above.

In addition to these general options, we can specify specific build targets. In this case, we are defining `pages` as our only target. We could define multiple targets if we needed. (For example, you may have a staging site and a production site. In that case, you could define to `buildcontrol` build targets: one for `staging` and one for `production`.)

For our purposes, we want to specify the remote server (using the SSH URL that Github gives us) and the remote branch (`gh-pages` , of course). This tells the `buildcontrol` task that it should deploy the results of the build process to our `gh-pages` branch. Once that happens, Github will automatically make that `gh-pages` branch available to the public.

Be sure to update the `remote` value with the SSH URL to your own repository.

Final `Gruntfile.js` result

Once you've configured your `Gruntfile.js` the top of it should look like this:

```
// Generated on 2015-07-11 using
// generator-webapp 1.0.1
'use strict';

// # Globbing
// for performance reasons we're only matching one level down:
// 'test/spec/{,*/}*.js'
// If you want to recursively match all subfolders, use:
// 'test/spec/**/*.js'

module.exports = function (grunt) {

  // Time how long tasks take. Can help when optimizing build times
  require('time-grunt')(grunt);

  // Automatically load required grunt tasks
  require('jit-grunt')(grunt, {
      useminPrepare: 'grunt-usemin',
      buildcontrol: 'grunt-build-control'
  });

  // Configurable paths
  var config = {
    app: 'app',
    dist: 'dist'
  };

  // Define the configuration for all the tasks
  grunt.initConfig({

    // Project settings
    config: config,

    buildcontrol: {
      options: {
        dir: 'dist',
        commit: true,
        push: true,
        message: 'Built %sourceName% from commit %sourceCommit% on branch %sourceBranc
h%'
      },
      pages: {
        options: {
          remote: 'git@github.com:your_github_user/your_webapp.git',
          branch: 'gh-pages'
        }
      }
```

```
    },

    // Watches files for changes and runs tasks based on the changed files
    watch: {

    ... more configuration ...
```

Once you've added this configuration to your project, you may proceed to deploy your project on Github Pages.

Using `grunt-build-control`

Now that you have installed and configured `grunt-build-control` in your project, you can deploy your site to Github Pages for all the world to see. Huzzah!

Review and commit your changes

In order to help you maintain a somewhat more sane approach to deploying your projects, `grunt-build-control` will not work when you have outstanding changes to files that are part of your repository. In order to deploy your work, you will need to commit all of your changes.

(NOTE: If you need a refresher on how to do this, refer to the Git Reference in the appendix.)

Build your project

Build the latest version of your project using the `build` task:

```
grunt build
```

This will update all the files in our `dist/` directory with the latest changes.

Deploy your project

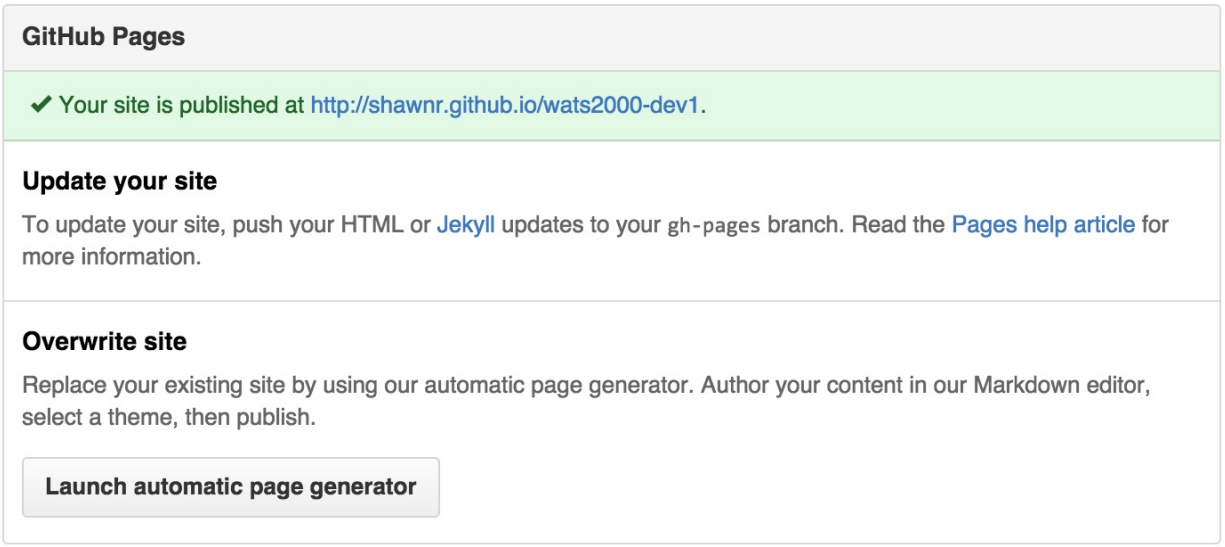
Now you are ready to deploy. Run the `buildcontrol` task to deploy:

```
grunt buildcontrol
```

That command will kick off the deployment process. You will see some information scroll down the screen, and it should all end up with a message that looks something like this:

```
Pushing gh-pages to git@github.com:shawnr/wats2000-dev1.git
To git@github.com:shawnr/wats2000-dev1.git
 * [new branch]      gh-pages -> gh-pages

Done, without errors.


Execution Time (2015-07-06 22:14:58 UTC)
loading tasks       1.1s  ▇▇▇▇▇▇▇▇ 16%
buildcontrol:pages  5.9s  ▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇▇ 84%
Total 7s
```

Once you're done, you should return to the Github page for your repository. You should see the new `gh-pages` branch has been created by `grunt-build-control`. If you click into the "settings" page for your repository, you should see that your site built successfully on Github Pages:

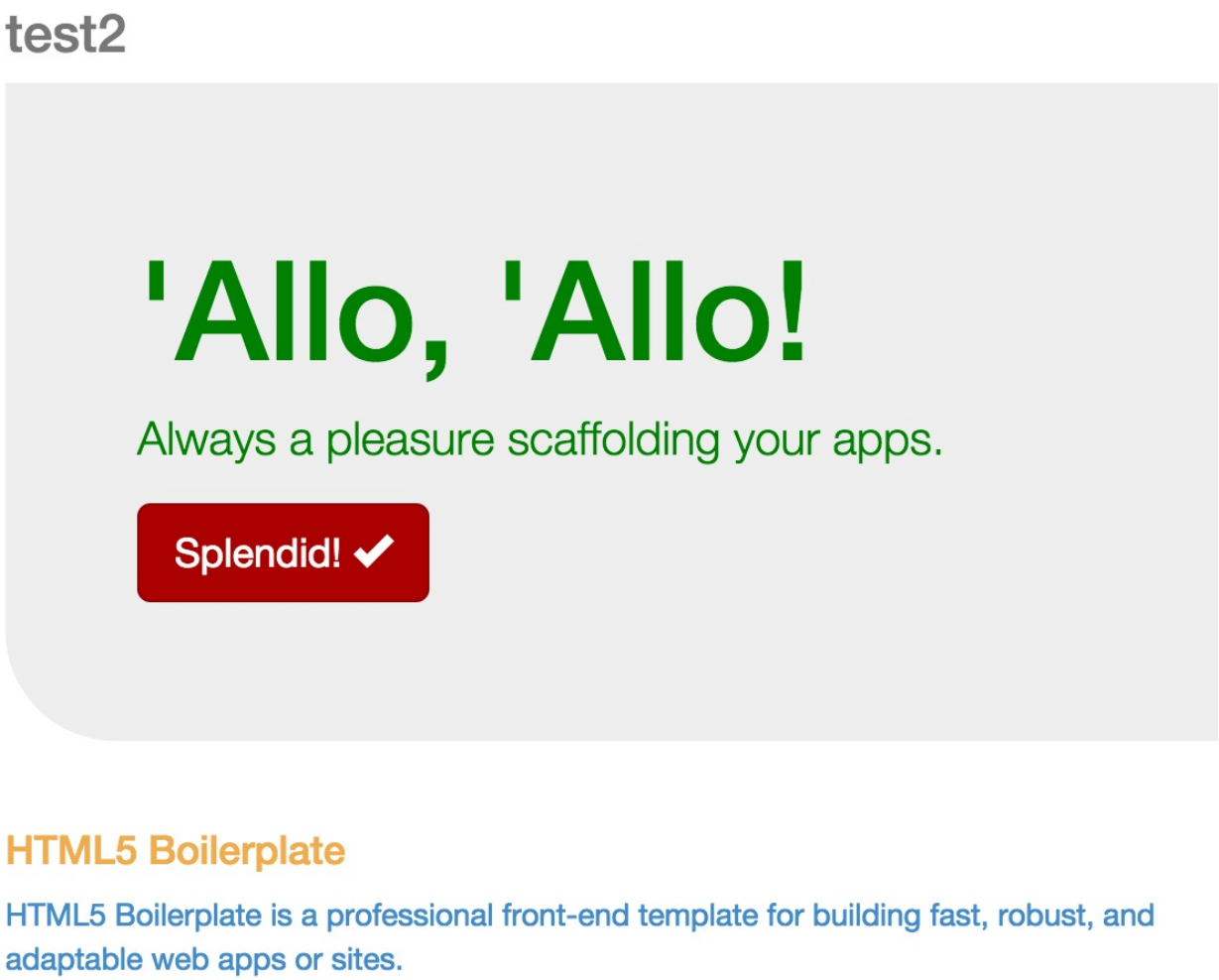


If there is an issue with deploying your site, then Github will alert you in that box. Usually if there is an issue with deploying your site you can fix it by making a small modification and re-deploying.

If you see the green bar, you should click the URL and visit your newly deployed website!

Après Deployment

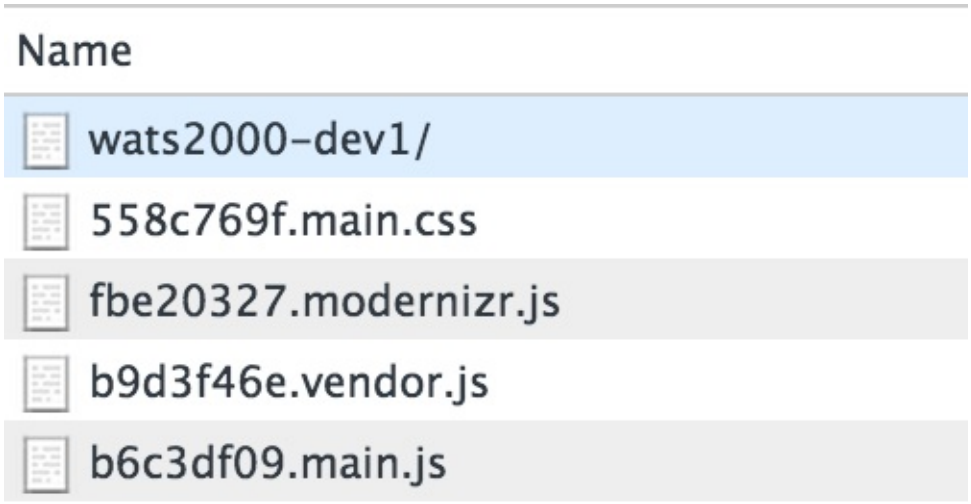
Now that you've deployed your project, when you visit the URL shown in our Github Pages Settings box, you should see something like this:



This shows the changes we made in the last chapter to everyone. If you inspect this deployment you will see several things that are different from when you normally preview your site using your local server.

File combination and versioning

Part of the value of a tool like SASS is that it allows us to edit many files (so we have the benefit of good file organization) but combine those before delivery so our users only have to download one file (which makes a huge difference in how fast our site downloads). If you look at the files being downloaded for our site, you will notice there are only a few of them:



Note that there is only one `main.css` file, in spite of the fact that we are now editing three different files for our styles (`main.scss` , `_content.scss` , and `_variables.scss`). Our SASS processor (`node-sass`) has helpfully combined all those files for us.

Likewise, the Javascript from our third party ("vendor") components has also been gathered up into a single file. Some components must remain separate, such as `modernizr.js` , and we want to keep our code separate from vendor code (hence the `main.js` and `vendor.js`), but we have still saved our users from making several other file requests form our server. That means they each have received our website more quickly than they otherwise would have.

Also note that these filenames have big numbers in front of them. This is a form of file versioning that helps us make sure our users are always getting the latest version of our site. When working with Javascript and CSS files, the caching technology of the web can make it difficult for users to get the latest versions of those files after we have made updates. There are many systems in place to help this process along, but in the end the best way to make sure users receive the correct files is to make sure each one has a uniquely versioned name.

Unfortunately, thinking about versions and re-linking files because we changed the name is not something we want to fuss with as developers. So we offload this task to a Grunt task and let it be: no fuss required. Yay for tools!

Minification

All HTML, CSS and Javascript is "minified" when we deploy. This means that extra spaces, linefeeds, and other formatting niceties are removed to make the file as small as possible. In some situations it might even involve rewriting the names of variables or functions to be shorter and smaller. There are many ways that minification helps compress files to make them as small as possible so they can be delivered to our users as quickly as possible.

Here is an example of what HTML looks like after it has been minified:

```
<!DOCTYPE html> <html class=no-js> <head> <meta charset=utf-8>
<title>test2</title> <meta name=description content=""> <meta
name=viewport content="width=device-width"> <link rel="shortcut icon"
href=/6df2b309.favicon.ico> <!-- Place favicon.ico and apple-touch-
icon.png in the root directory --> <link rel=stylesheet
href=styles/558c769f.main.css> <script
src=scripts/vendor/fbe20327.modernizr.js></script>  <body> <!--[if lt
IE 10]>
      <p class="browsehappy">You are using an <strong>outdated</strong>
browser. Please <a href="http://browsehappy.com/">upgrade your
browser</a> to improve your experience.</p>
    <![endif]--> <div class=container> <div class=header> <ul
class="nav nav-pills pull-right"> <li class=active><a href=#>Home</a>
</li> <li><a href=#>About</a></li> <li><a href=#>Contact</a></li> </ul>
<h3 class=text-muted>test2</h3> </div> <div class=jumbotron> <h1>'Allo,
'Allo!</h1> <p class=lead>Always a pleasure scaffolding your apps.</p>
<p><a class="btn btn-lg btn-success" href=#>Splendid! <span
```

If you use the "Inspect Element" tool in your browser, then you will still see the HTML broken into a nice visual display, but that's because the developer tools are able to reformat the HTML. When you view source, you will see the compressed HTML like you see above.

More

More happens when we build the site, too. Tests are run. If they fail, it will cause us issues. Fortunately, we have no tests written, so that cannot prevent us from deploying. But when we get around to adding tests to our project, that will be a great way to help us not make a mistake by deploying broken code.

Grunt is also looking at all of our images and trying to optimize those. Right now we are not using any images, but if we did then they would be optimized and compressed as much as possible before deployment. This is a good thing for our users.

There are many other things that Grunt can do for us as part of the build and deployment process. As you encounter challenges in your development, keep in mind the things Grunt can do for you and on the lookout for opportunities to get more out of a tool you're already using.

Congratulations!

We have now completed the prep work for our webapp project. In the next chapter we will bootstrap our first AngularJS application, and it will use all the pieces we have prototyped here.

Web Application Frameworks: An Overview

This section covers the broad topic of "web application frameworks", and aims to give you a broader understanding of why we are doing things the way we are. This topic is really still too broad for a single section, so we will cover many of the foundational concepts somewhat lightly. The hope is that given a general understanding and then some practical experience working within a framework, you will then be able to go out and engage more effectively with the many other great learning resources already available for various specific frameworks and approaches.

What is a "framework"?

Framework is a term used to address a collection of pre-created code that can be used to augment an application designed for the end user. Frameworks are designed for developers, and they aim to "take care" of the repetitive (or "boilerplate") code that we often must generate to get a project up and running.

Take, for example, the Django framework. Django is a framework written in Python, which means that if you are using it, then you should be writing your application in Python. Django is a framework designed to create websites. It contains code that can speed your development by giving you systems for handling HTTP requests, routing URLs, processing templates, accessing databases, and much more. If you wish to make a new page, or connect to your database, there are pieces in Django that make that process much easier.

However, Django, itself, is not an application. It's not like Wordpress, for example, which is a content management system. When you "install" Django, you are really just adding it to a project (just like we have added Javascript modules to our webapp project already). In order to have anything you could use to, say, post a blog online, you would need to write additional Python code to define your site and the content you wish to make available.

When you install Wordpress, you have a functional blog site that you can then modify into any number of types of content-based website.

This is a major difference between an application and a framework: All by themselves, frameworks are nothing but a box of parts waiting to be assembled into a functioning application.

Frameworks often also dictate what we build. For example, if your goal were to build an in-browser game, Django would likely not be helpful for easing the development of the player's interactions or your game screen display. Frameworks for handling games and animation have very different pieces in the box than frameworks designed for handling HTTP requests and database connections.

Frameworks are also often used in conjunction. In the example described above, you may use a framework like Django to run the data and HTTP server portion of your game (so you can store user profiles, high score tables, etc.) and then you could use a framework like Crafty.js for actually making the game run.

Why do I need one?

There was a time when no websites used frameworks. In the beginning there was the "static website": that is, sites where all the files are uploaded and as you navigate the site you move between real files present on the server's storage drives.

Quickly, nascent web developers created what would ultimately come to be known as "static website generators", which were scripts that would process templates written in various formats to build stacks of real files on the server's storage drives. (It's worth nothing that this approach has remained with us today, and has enjoyed a renaissance as storage costs have plummeted and cheap or free static website hosting has become plentiful.)

Developers also quickly moved into creating web-oriented frameworks across many favorite languages. We saw the fast rise of things like ASP.NET and JavaEE Servlets. These large technologies dominated for the first decade of the Web, and served the primitive web well. But it was quickly realized that we needed a better way to handle the growing complexity of building websites.

In the mid-2000s there was another boom in web application framework design. Microsoft dropped the "ASP" and released .Net, a greatly improved framework that uses C# as the programming language. Java continued to ride high as a language with the release of several popular frameworks including Struts, Web Objects, and Google Web Toolkit. Even Perl, which was the language of the "CGI-BIN", got into the act with frameworks like Catalyst gaining popularity around the time.

In addition to the "enterprise" level of web application frameworks on offer from major players working with well-supported languages, several serious competitors came out that have ultimately had a major effect on the web world. PHP took off in the mid-2000s, opening the door to low-cost dynamic backend web development for many more potential developers and websites. PHP frameworks like CakePHP, Symfony, and Zend remain popular today.

As PHP boomed, so did other languages online. Python and Ruby both saw a serious increase in popularity beginning around 2005 that continues today. In the Python world early frameworks like ZOPE, web2py, and Pylons laid the groundwork for getting Python onto web servers. The Django framework has continued to dominate Python-based web development.

Ruby's most popular web application framework is Rails, popularized by a group of bombastic creators and enthusiastic supporters. Rails was the first popular framework to boast robust database integration so that developers did not have to hand code SQL in order to make their apps run. This made Rails immensely friendly to developers who wish to rapidly create content types and use them in innovative ways.

The approaches pioneered in the mid-2000s have continued to grow and evolve. Although Django and Rails are both dominant frameworks today, there are numerous alternative frameworks (such as Flask and Sinatra) that aim to serve various types of websites within various types of contexts.

Although the BEST way to build a webapp is far from settled, one thing we appear to have learned over the years is that if you're going to build webapps, you will likely benefit from using a framework.

What about Javascript?

You may have noticed that all of the discussion so far has focused on backend languages. That's because the story of Javascript frameworks is both shorter and longer than the summary given above. Although serious usage of Javascript frameworks has not been at the foreground of web development for as long, over the past five years Javascript has become an even more important component of building websites.

Javascript is everywhere, and developers are using it in amazing new ways every day.

A Brief History of Javascript in Web Development

The web is a strange place. We use all sorts of technologies to create websites that do all sorts of things. We use Java and Ruby and Python and PHP and myriad other languages; we use nginx and Apache and Tornado and Twisted and all sorts of servers made for specific purposes; we use mySQL and Postgresql and MongoDB and Cassandra to store data for use in dynamic applications. This list just barely hints at the surface of how diverse and complex the server-side components of any given web application can be.

On the client-side, things are quite a bit different.

Web browsers are built to adhere to standards. Most of the core standards and specifications governing the Web are created by the World Wide Web Consortium (W3C). The W3C publishes standards that, for example, govern how Hypertext Markup Language (HTML) should be interpreted, how Cascading Stylesheets (CSS) work, and what APIs are revealed on the Document Object Model (DOM). These standards then dictate what browsers do when they receive HTML, CSS and Javascript from the server.

And that's a key point: The browser itself only understands HTML, CSS and Javascript. Functionality can be expanded using plugins and extensions, but those are browser-specific and proprietary. At the fundamental level, all websites boil down to the HTML, CSS and Javascript.

HTML is the core building block of any given webpage. The HTML provides a structure that defines the content on the page so the browser "knows" what exists in that page. CSS is a means of configuring the visual presentation of HTML elements. HTML structure allows CSS to reference content on the page using element names, classes, or IDs, tying the HTML and CSS together directly.

Javascript interacts with the page using the DOM, which is a programmatic representation of the structure of the HTML. The browser downloads the HTML, reads it, interprets it, and then holds the dynamic, programmatic representation of that HTML, the DOM, in memory while the page is being viewed.

Of the three primary web technologies that make a page live and breathe in your browser, Javascript is the only one that is a programming language. HTML is a markup language: it describes content using HTML elements. CSS is a configuration language: it defines settings

for visual presentation of HTML elements. Javascript is a programming language: it allows you to create logical definitions and structures that can provide dynamic response to user actions.

Because Javascript is the only programming language that is interpreted and executed in the web browser, it enjoys a special place in the world of web development. This is why Javascript is so important.

Early Javascript

Javascript was created by Brendan Eich at Netscape and was introduced in 1995 through the Netscape browser. Microsoft adopted Javascript in 1996 with Internet Explorer 3, and since then Javascript has been a part of the web.

In 1997 Ecma International, a standards organization based in Europe, released the first ECMAScript standard, lending Javascript governance outside of a single company. ECMAScript standards have been released steadily ever since, with the latest being ECMAScript v. 6, which released in June 2015.

At first, Javascript was used for very basic features, many of which have since become absorbed into other technologies. For example, at one point it was common to use Javascript to handle "image rollovers"—making images switch when the user moved the mouse cursor over them. Using this technique brought about a revolution in web design with sites that had more graphic navigation bars that could highlight current and active menu options.

Javascript had the ability to modify elements in the DOM, but the native APIs for working with the DOM were clunky and limited. In the early 2000s Javascript developers created libraries, which grew into frameworks, to make manipulating the DOM much easier. Numerous libraries competed, including MooTools, Dojo, and ExtJS, but the library that became a framework that ruled Javascript development for many years was jQuery.

jQuery introduced the `$()` (often pronounced "dollar sign query"), which was a shorthand way of referencing the main `jquery` object. The "query" part of jQuery came from the fact that developers could use CSS notation to select DOM elements, allowing them to "query" the DOM for the elements they want.

With jQuery, one may write:

```
$('nav a').on("click", function(el){
    el.addClass('demo');
});
```

That code would add a "click" listener to all of the links that are inside the `<nav>` element. To do the same thing with native APIs would take many more lines of complex code.

jQuery has been a great solution for making dynamic web pages in many ways:

- It provides a great set of tools for developers.
- It generally plays nicely with unrelated Javascript, so you can use it alongside lots of other tools.
- It has good support for making "AJAX" (asynchronous Javascript) requests.
- It has a "plugin" system that allows developers to create custom functionality and complex modules to make specific effects happen, and those plugins can be leveraged in a fairly standardized way.

Recent Javascript

Recently, Javascript has enjoyed another surge of popularity. Already firmly established since the creation of frameworks like jQuery, Javascript has moved everywhere the web has moved. As televisions, in-car navigation systems, and mobile devices have all leveraged web technologies, Javascript has become a part of those ecosystems.

In 2009 Node.js was created, giving developers a very fast and robust implementation of a Javascript engine that could run on the server. This has opened the door to a whole new wave of module development, with thousands of modules now created for Node.js. The architectural concepts pushed by Node.js have led to changes in how webapps and websites use interdependent Javascript modules.

The rise of Node.js popularity has led to Javascript being placed on all sorts of embedded machines, too. There is a whole subculture built around gadgets, like drones and hobby robots, powered by Javascript code.

The enhanced interest in using Javascript has led to other noteworthy developments, too. The entire development environment and stack that we are using for this project is Javascript-based. As we build our pages we use a huge array of Javascript tools to help us, and we write Javascript that runs in the browser for our users as they use our sites.

Javascript, again, is everywhere.

Evolving Approaches to Web Development

Over the years, web development has evolved. As a teacher, I feel like watching students learn HTML, CSS and Javscript looks a lot like reviewing the history of the Web. As a world, we have learned how to build websites and webapps together, over the past 25 years.

Along the way we've come up with ideas that have helped us build better websites and webapps at the time. However, it's important as developers that we do not become complacent. There is always a new challenge, a new approach, and a new role to serve. How we are working in this book is the result of this continued evolution of web development.

The challenge of progressive enhancement

The generation of Javascript libraries that birthed tools like jQuery in the mid-2000s was an era built around "progressive enhancement" (and, sometimes, progressive enhancement's less appealing cousin, "graceful degradation"). Developers could not be sure that all users would have browsers supporting Javascript, and if they did standards adoption among browser makers could be spotty, so the Javascript might not work. Because of this, developers had to always guard against breakage.

To help with this, tools like jQuery were made to enhance the vanilla HTML and CSS that would normally be delivered to the page. Properly done, pages could be created that were totally usable in a traditional, "click and refresh", way if the Javascript failed to load. But if the Javascript succeeded at loading, then pages would come alive with background updates, fancy interfaces, and more.

This approach is still worthwhile, and for many purposes it may still be a great idea to think about a project from a progressive enhancement point of view. For example, if you are building a site that is dedicated to showing off specific content, or providing access to media files, there is a lot that can be conveyed in traditional HTML. That content can be loaded and accessible to a wide range of devices, then Javascript, which is very reliable these days, can enhance the content to provide modern features.

But there are limitations to progressive enhancement. For example, what if you have a web application, and this application relies on the interaction of the user with Javascript to have any value? We aren't imaginging a site that wants to show you a video; imagine instead a site that wants to help you make a video. What is the value of a "static" or "traditional" page in that context? Probably not much.

What webapps want

Dynamic webapps have a whole different set of considerations. If Javascript is not enabled, then the entire app is going to be unusable. So why bother having robust fallback content? (Other than a helpful error message, of course.)

Other issues come up when you build webapps:

- With jQuery, we could make a quick call to see if, for example, a content item was favorited, or to send a new favorite to the server in the background. But what if we want to build an app where all of the screens have access to the data for objects in the system, and where we keep all that data synchronized across multiple views? This becomes very complex using only jQuery or similar solutions. We need a better way of **managing and modeling data** for use in our app.
- With progressive enhancement, we could enhance pages by making tab layouts or other custom effects. But what if we want to provide consistent headers and footers? What about slide-out menus that are the same across multiple views? These are the sorts of interface elements that exist in applications, but not so much in content-based websites. We need a better way to handle **templating views** so we can better manage our interface.
- In order to handle the data and render the views, we must write logic that applies to all views and all template renders. With jQuery and a traditional approach, we could not easily divide logic for syncing data after changes, updating views when the user interacts, or modulate template rendering based on user preferences or application state. We could accomplish these things, but it usually required repetitive coding and created a huge opportunity for mistakes. We require a **control and logic mechanism** that allows us to abstract and encapsulate Javascript logic into discrete, reusable components the application can utilize as needed.
- Since we do not want to make a page request to the server on each click, we cannot rely on standard links to be interpreted by a web server to move our users around our webapp. This provides us with a challenge of knowing which views to show our users as they move through our application. We require a **routes mechanism** that can be used to define specific "locations" inside our webapp.

These are just a few of the major challenges we face when we try to build more app-like websites, and the old methods just don't work.

Fortunately, developers are never short of "new methods", and we are amid a boon of what are commonly known as "Single Page Application Frameworks". Webapps (also often known as "Single Page Applications (SPAs)") are the way developers are thinking these days, and developers have many different approaches to solving all of these challenges.

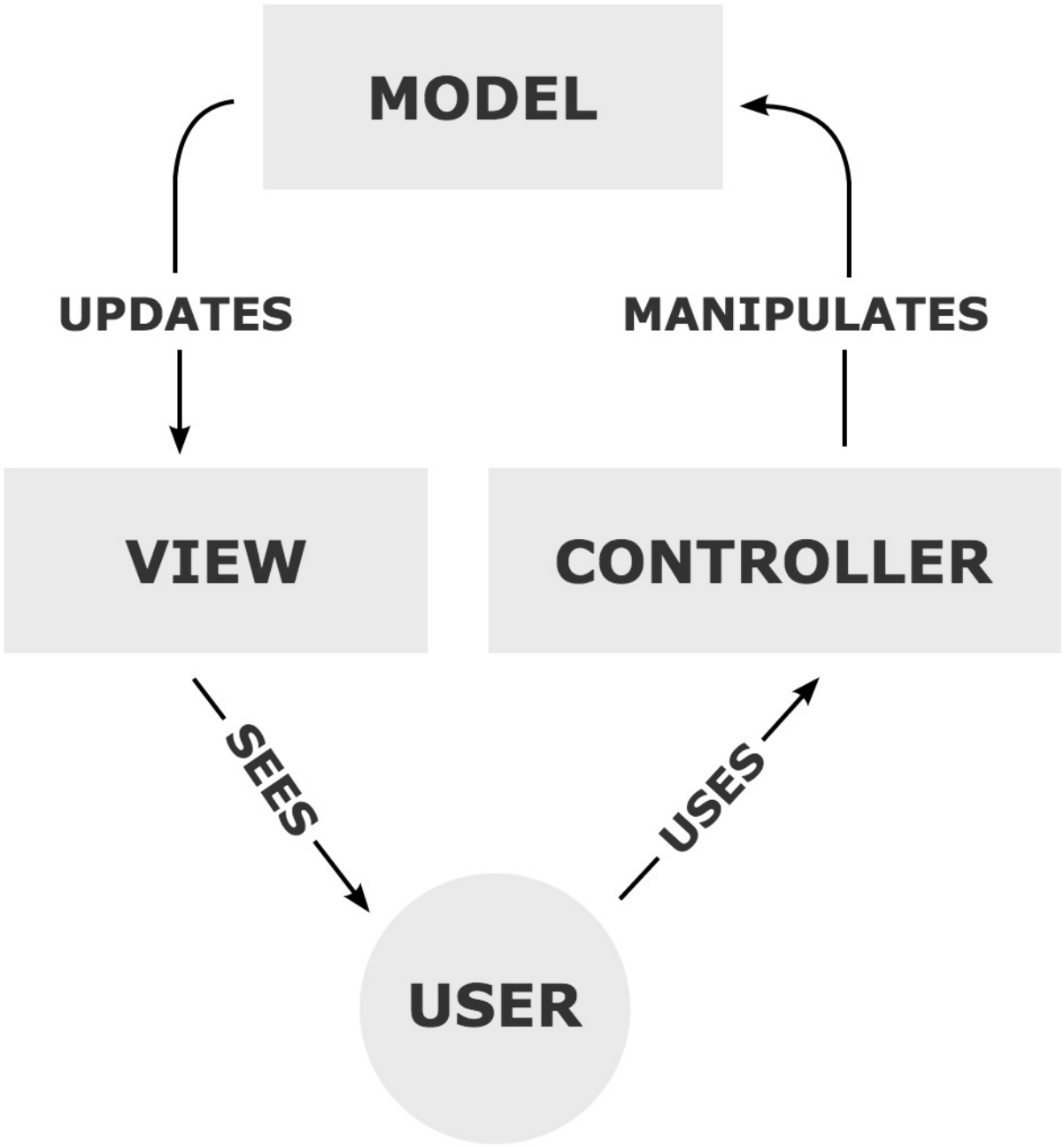
Model View Controller

Application frameworks all have an "opinion" about how projects should work. That is, they all define a way that the pieces of the framework go together, and that method of organization is often unique to the specific framework. If you, for example, have become familiar with Rails, you would not necessarily know how to build a website with Sinatra, although those are both Ruby-based web frameworks.

Fortunately, there are some common concepts that tend to be at work in these application frameworks, and understanding these base concepts is useful for learning any new framework.

The MVC approach

Most web application frameworks will be described as "MVC" frameworks. This means they, in some way, implement the "Model View Controller" approach to application design. This is an approach that has been in use since the beginning of graphical computing, and it defines three basic concepts that can be applied to almost any application.



(Ilustration used courtesy Regis Frey.)

Model

The "model" represents the data in the system. All the information and the algorithms that make that information useful are rolled up into the concept of the model. It is our model of whatever concept, object, or event we are working with in our application. This model retrieves data, stores data, and manipulates data to make it useful to our users.

As an example, imagine a Twitter application. The "model" would define how to fetch data from Twitter, how to manipulate the data, and how to post data back to Twitter. The model is busy stewarding the data as it moves through the application.

View

The "view" is made up of the information we show to the user. This view is informed by the model, and it provides an audiovisual interface to the information the user is working with.

In the example above, our Twitter app may have several views: We might view our feed with all the posts from the accounts we follow. We might also have a view that shows just one Tweet. And we might have a view that shows the profile information for a Twitter user.

Controller

The "controller" provides an method for the User to interact with the application. The "controller" communicates the user's intention to the model, which then updates the data in the system and, in turn, updates the view the user sees.

In our example Twitter app, one controller element that could be presented to the user would be a form for sending a new Tweet. Other controller elements could include the Favorite and the Retweet buttons. All of these interface elements would result in the controller sending information back to the model that would ultimately result in the data in the model being updated and the views being updated as well.

The virtuous cycle

The three components of an MVC system work together to keep the application responsive to the user and constantly updated. Although this core concept is incredibly popular, frameworks that implement the idea of an MVC approach are diverse and varied. It's critical to understand the basic concept here, but also critical to learn the specifics of whatever framework you are working with.

A Summary of Popular Javascript Application Frameworks

There are many Single Page Application Frameworks (SPAFs) available for building websites with Javascript. These were created to ease the pain of making more responsive webapps that can run in the browser. The developers of these frameworks have borrowed, modified, imagined, and otherwise generated a wide range of approaches to addressing the challenge of building a webapp.

What follows is a short list of some of the popular SPAFs that are not AngularJS. This book will use AngularJS as the primary example framework, so we will cover it on consecutive pages. For now, let's survey the landscape.

Backbone.js

Backbone was one of the first Javascript-based SPAFs to come out. It tends to be farily "unopinionated", which means that it does not make many requirements about how you organize your code. Backbone is still a very popular tool, and there are many supporting libraries and modules that can enhance Backbone apps (such as Marionette and Thorax).

Ember

Ember is a framework designed to speed development and remove mundane boilerplate code from your list of TODOs. It provides you with a more opinionated setup, meaning that it uses standard approaches to things and assumes you want to do that, too. It is built around some popular templating libraries, and boasts a full set of included features.

React + Flux

React provides a tool for making interactive Javascript components for your webapp that keep track of their state and their data. Flux is an application framework used by Facebook to build their webapps. Flux makes use of React to create highly interactive and responsive interfaces.

Meteor

Meteor is both a client and server-side solution for building webapps using pure Javascript. The server is designed to integrate closely with your webapp so you can quickly build custom APIs to interface with your data stores and then use them in your webapp.

TODO MVC Comparison

These are just a few of the popular Javascript SPAFs that are in use today. You can find many more, along with a sample project to compare across SPAFs on the TODO MVC website. This site shows the implementation of the same TODO webapp in many different SPAFs. You can compare how the code looks in each one and get a great idea of the differences between frameworks.

An Overview of AngularJS

AngularJS is a Single Page Application Framework (SPAF) created and maintained by engineers at Google. It is used to build many webapps, and it is even used to build mobile apps on many platforms.

AngularJS is considered an opinionated framework because it dictates much of your project structure in the interest of providing a lot of support for rapid development. AngularJS uses an MVC approach. Data Models are implemented in a way that works well with third party APIs, and data persists and is updated throughout the application. Views are handled through templates, which are important parts of an AngularJS app. Finally, Controllers are attached to different elements in each View and can power all sorts of features.

AngularJS is supported by a large community of developers, and there are many modules available. Most things you will want to do in AngularJS have probably already been created and listed on ngmodules.org, the main site for listing AngularJS modules.

Getting into AngularJS

Learning any framework takes time and effort. It's important to be diligent about figuring out *how* developers using your chosen framework approach a problem. There are always many ways to fulfill to any given requirement, but being able to do it in a way that "fits" with the framework you're using is crucial to long-term success. If you can learn to work with the framework, then it will help you go faster and build better apps. If you constantly work against the framework, then you will constantly be frustrated and will move more slowly.

AngularJS, like any framework, has a "philosophy" or "worldview". That view is centered around powerful templates, revealing functionality through custom HTML elements ("Directives"), keeping data streaming through the system, and keeping logic contained in discrete locations. The end result of this is that developers are given easy to manipulate components that can easily be moved around to dramatically alter visual presentation and user experience without requiring any changes to the underlying code.

This gives AngularJS exceptionally powerful templates, and allows both highly trained and more novice developers to contribute productively to a project.

Core concepts in AngularJS

In order to set us up for moving through the work described in the next several chapters, it's useful to get a grasp on some of the core concepts of AngularJS. Over the next few pages we will discuss these basic principles of the framework:

- Data Models
- Routes
- Controllers
- Views

Data Models

In AngularJS, Models represent the data that the system will work with. Models can be defined at almost any level of the application, and all sorts of components in the system can "watch" and react to models and changing data.

In this example from the AngularJS documentation, you can see how models are defined quickly using the `ng-model` attribute:

```
<div ng-app ng-init="qty=1;cost=2">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="qty">
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="cost">
  </div>
  <div>
    <b>Total:</b> {{qty * cost | currency}}
  </div>
</div>
```

(Code borrowed from angularjs.org example.)

In this example, two inputs are put on the screen: one for "quantity", the other for "cost". As the user fills in this information, the total cost will be calculated and displayed. In this case, the models being used are very simple. They are named `qty` and `cost` and they are defined using the `ng-model` attribute on the `<input>` elements.

This is a very simple example that defines the models right in the templates. AngularJS data models will often have some more complex mechanisms for populating data and keeping it updated. If you are working with a third party API, then you will likely want to create a `service` or `ngResource` to help keep your data connected across all your views, or you may use built in tools like `$http` (AngularJS's http request library) to manage that data connection.

That means the Javascript defining your data models and how you get your data could be quite complex. Different apps require different data connections, so there is no one correct way to set up data, and through this project we will explore a couple of useful approaches.

Routes

In any webapp, we face a challenge of locating different "screens" or "views" of our app. These are often equated to pages of the site, and in the case of webapps we often still want to retain the ability to link precisely to content within the app. This requires us to have URLs that make sense and can be used by the application to return users to specific screens or content. Much of web technology is built around having URLs that point to specific information, and web users tend to appreciate tools like their browser's back button. It's crucial that we do not break the way URLs function when we build webapps.

In AngularJS, there is a mechanism for providing `routes`, which allows us to, essentially, create "locations" within our webapp. The Javascript code to define routes might look something like this:

```
phonecatApp.config(['$routeProvider',
  function($routeProvider) {
    $routeProvider.
      when('/phones', {
        templateUrl: 'partials/phone-list.html',
        controller: 'PhoneListCtrl'
      }).
      when('/phones/:phoneId', {
        templateUrl: 'partials/phone-detail.html',
        controller: 'PhoneDetailCtrl'
      }).
      otherwise({
        redirectTo: '/phones'
      });
  }]);
```

(Code sample borrowed from angularjs.org documentation.)

In this example, we can see that two different views are being configured: One view exists at `/phones` and will show a list of all phones in the system, controlled by the `PhoneListCtrl` controller. The other view exists at `/phones/:phoneId` and will show the information for just one phone model using the `PhoneDetailCtrl` controller. The `otherwise` clause tells the webapp that if the URL patterns do not match either of the two defined views, then route the user to the `/phones` view (the list of all the phones).

Note that each `route` defines three details:

1. The route pattern ("/phones") that can be matched in the URL.
2. The `templateUrl` attribute, which defines which template will be rendered.

3. The `controller`, which defines which controller will be invoked.

This kind of routes definition would allow us to share links to specific phones, make bookmarks to specific phones, and would otherwise allow us to use this webapp like we expect.

Controllers

In AngularJS, Controllers are the primary Javascript code that determines how some specific component or View functions. In view templates, we define which controllers affect which components on the page. It's possible to have a controller entirely dedicated, for example, to a "favorite" button. Or, we might have a controller attached to the template as a whole (such as the `PhoneListCtrl` in the previous example).

Controllers handle the logic that makes the view function and they make data available to the templates by defining the `$scope` variable. In AngularJS, each Controller has a `$scope`, which is a data object used to contain all of the information available to the template. In a template, using the `ng-model` attribute to define a new model is actually the same as defining that model in the `$scope` object in the Controller for that View.

This in-template `ng-model` definition:

```
<input type="text" ng-model="username">
```

is actually creating this value:

```
$scope.username
```

If we were to initialize the value in the Controller for this example View, then we would write something like this:

```
$scope.username = "Initial Value"
```

And that would cause the input to be populated with the name "Initial Value" when the View first loads.

In the Controller for our view, we can define the different functions that could be carried out. Look at the example we saw previously for calculating total cost:

```
<div ng-app ng-init="qty=1;cost=2">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="qty">
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="cost">
  </div>
  <div>
    <b>Total:</b> {{qty * cost | currency}}
  </div>
</div>
```

That same functionality could be rewritten like this. First, in the template:

```
<div ng-app ng-init="qty=1;cost=2">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="qty" ng-change="calculateCost()">
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="cost" ng-change="calculateCost()">
  </div>
  <div>
    <b>Total:</b> {{totalCost | currency}}
  </div>
</div>
```

And then in the Javascript controller:

```
$scope.calculateCost = function() {
    $scope.totalCost = $scope.qty * $scope.cost;
};
```

In the updated example, AngularJS will watch for changes to the two text inputs, and when a change is detected the `calculateCost()` function will be executed. That function will update the value of `$scope.totalCost`, which will update the display in the template.

It's worth pointing out that in the Javascript controller we reference these variables as part of the `$scope` object (like so: `$scope.qty`, `$scope.cost`, `$scope.calculateCost()`), but when we reference these `$scope` values in the templates, we can refer directly to the values and drop the `$scope`. So in the template we can just refer to `qty` or `cost` or `calculateCost()`.

Views

Views in AngularJS are very powerful, but they can be a little confusing to wrap your mind around. When we talk about the "view", we are really talking about the rendered version of the template. When AngularJS templates are rendered, they are processed with all of the different components needed. That means the controllers, irectives, services, and all of the other parts of your AngularJS app combine with the template to create the view.

When working with views, you will move between different components to adjust functionality. You may need to update the controller for a specific feature, or create a new directive to handle some custom formatting.

But in AngularJS views are orchestrated by the template. Templates bring together HTML, AngularJS directives, set up model relationships and much more. The behavior and presentation of a view can be radically altered by modifying the template, giving template developers a lot of strength in AngularJS apps.

Here is a simple example of an AngularJS template from angularjs.org:

```
<html ng-app>
 <!-- Body tag augmented with ngController directive  -->
 <body ng-controller="MyController">
   <input ng-model="foo" value="bar">
   <!-- Button tag with ng-click directive, and
          string expression 'buttonText'
          wrapped in "{{ }}" markup -->
   <button ng-click="changeFoo()">{{buttonText}}</button>
   <script src="angular.js">
 </body>
</html>
```

Notice that it mostly looks like HTML except for the special attributes that start with `ng-` and the use of curly braces to output variable values (`{{buttonText}}`). The custom attributes are called "directives" in AngularJS lingo, and they represent complex Javascript functionality that can be accessed easily. For example, as a template writer all you have to do is use the `ng-model` directive to define the model for the input, but behind the scenes AngularJS is executing a complex set of code to attach the value of that input to a `$scope` variable and make that available to everything else in this view.

These are aspects of AngularJS templates that will become familiar as you work on building your webapp. As with HTML before it, there are lots of things to absorb, so the point is not to learn every single possible feature. Start with the few things you will use all the time, and

then build out your knowledge from there.

The End of the Tour

There is much to learn about AngularJS and using it to build a webapp. Once you have learned one framework, it will be easier to move on to other frameworks. As long as you know one thing, it is always easier to learn the next thing.

For now, immerse yourself in AngularJS, but come back later to explore some of these other frameworks. This is an exciting, dynamic space right now, and the only thing that is for sure is the continued pace of change.

Additional Resources

AngularJS is a huge topic. Luckily, there are many developers using it and writing about it. As you work through this book and other AngularJS projects, you may find it useful to consult some of these general resources:

- AngularJS Docs
- NG Modules
- Learn AngularJS

Bootstrapping an AngularJS Project

Now that we've gotten to know how to use this Yeoman/Bower/Grunt environment, and we have figured out how to add and modify elements (so we can do things like deploy to Github Pages), it's time to get into working with our selected application framework.

For the project we build out in the remainder of this book, we will use the following "stack", or combination of tools and modules:

- AngularJS
- SASS
- Bootstrap CSS/JS Framework

Those are the major components, but we will be using additional elements throughout. We will add `grunt-build-control` again so we can deploy to Github Pages. We will also add several custom AngularJS modules that will be managed using Bower. These modules will help us with things like making sure we maintain accessibility and using the localStorage system to create permanent data storage for users.

In order to get working on our project, we must bootstrap a project skeleton. That involves a few steps.

Install `generator-angular` for Yeoman

AngularJS has an official generator, which we will make use of. We will install this first.

Bootstrap an AngularJS project

Once we have the generator installed, we can bootstrap a new project skeleton with Yeoman. This will build out the basic files and templates we will use to build our site.

Set up SASS

By default, the AngularJS project generator is configured to use a tool called "Compass" to process your SASS styles. Compass is a great tool, but it is a Ruby Gem, which means in order to install it you must have Ruby installed on your computer. If you already use Ruby on projects, it may be much easier to just install Compass and continue with the work.

If you do not already use Ruby, you may prefer to stick with a pure Javascript solution. Since we don't use Ruby anywhere else in this project, this book will show you how to convert your AngularJS app to use SASS instead of Compass, which is set up by the project generator.

Set up `grunt-build-control`

We will use `grunt-build-control` again to handle deployment to Github Pages. This will be configured pretty much just like we did before, but in order to do that we'll also need to create a Git repo on Github.

Install `generator-angular`

Just as we installed the `generator-webapp` previously, we must first install the Yeoman generator for making AngularJS project skeletons. Installing the AngularJS generator is just like installing any Node.js module. You may consult the NPM package page for `generator-angular` for further details about the generator.

```
npm install -g generator-karma generator-angular
```

If you already have these generators installed, be sure to update them:

```
npm update -g generator-karma generator-angular
```

Once that process has finished successfully, you are ready to move on.

A little more about `generator-angular`

The AngularJS project generator contains support not just for generating a brand new project, but also for building new components of a project as you work on it. This is useful for keeping projects in order, and also for helping novice developers make quicker progress on project builds.

Consult the `generator-angular` NPM package page for more information about all of the commands the generator offers, and be aware that we will use those commands as we work through building out our project here. When you see those `yo angular:<command>` commands come along, that's what is happening: We are using the generator to generate a new component of our existing project.

Note: Some of you, dear readers, may have learned about the problems that can be caused when generating a project inside of a project. The reason that causes problems is that in order to facilitate features like `generator-angular` makes use of, Yeoman looks in the parent directories for a project definition. When it finds one, it tries to run the command as part of the existing project. Needless to say, if you're not running a command that works within the existing project, you will see problems.

Boostrap Your AngularJS Project

Bootstrapping your AngularJS project is just like boostrapping any other project using Yeoman. Once you've installed `generator-angular`, you will want to bootstrap your project.

Preparing to bootstrap the project

Previously, we had bootstrapped our projects into a Git repository we had already made. Make a new Git repository and clone it to your development space. If you generate a repository on Github it will ask you if you want to make a README file. You do not need to create one, but you may if you wish.

Clone the repository you created on Github to a location where none of the parent directories have Yeoman projects inside of them. Open your terminal and change to the root directory of your project.

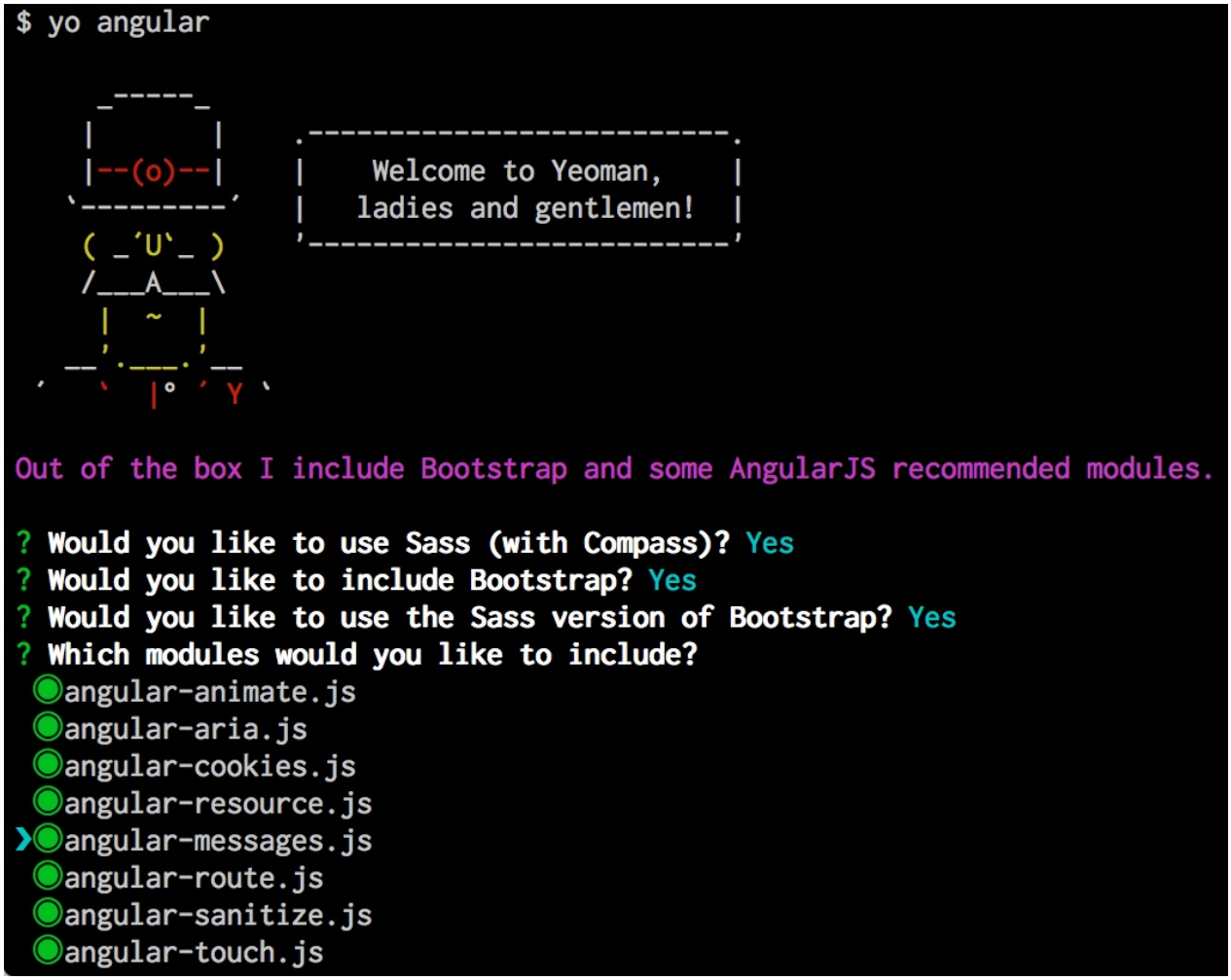
Throughout the rest of this book, I'm going to make the assumption that you have the following structure for your work space:

`~/Projects/angular-app/`

I'm assuming that you have your home directory (represented by the `~` symbol), a `Projects` directory (which contains all your development work), and then your repository (which is called `angular-app`). Whenever you see this path, you can know I'm just referencing the root directory of your AngularJS webapp repository.

Now that you're in the right location, run the bootstrap command:

```
yo angular
```



That will bring up the standard Yeoman bootstrap interface. Answer all the questions Yes (even the one about using Compass) and select all the Angular modules: we will explore many of them.

Once you hit enter after selecting all the modules, Yeoman will proceed to build out the project.

Please Note: At the time of writing, there is a small issue where you may be prompted to overwrite `package.json` at some point during the installation process. This is an issue with the `generator-angular` scripts, and you should answer `Y` to overwrite the file. Unfortunately, during the installation, when it prompts you to enter that response, there is not time to actually do so. Keep watching the progress on the bootstrapping and **eventually it will stop and look like it has frozen up**. The process hasn't actually froze. In order to continue, type "Y" and press `return`. You should see the final results of the bootstrap process ending with a success message.

Configure SASS

In order to run the server on your site, Grunt will need to build your stylesheets. Unfortunately, the `generator-angular` template forces us to set up the project to use Compass. Compass is a great CSS preprocessor, but it has one downside for us here: It requires us to use Ruby. If you already have Ruby installed, or if you're interested in working on Ruby projects, feel free to install Compass and continue using Compass. Compass will give you additional features you can leverage that we won't cover in this book.

Since we have no dependencies outside of Javascript for our work here, we will use a Node.js module called `grunt-sass` to process our CSS. This uses a pure-Javascript SASS processor to provide speedy, high quality treatment of our `.scss` files.

(Note: The approach here mimics the approach described by Ravel Antunes in his article.)

Install `grunt-sass`

Installing `grunt-sass` is the same as installing any Node.js module into your project. We will use `npm` and we will send the `--save-dev` flag to save this into our `devDependencies` list.

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

Configure `Gruntfile.js`

We must remove the configuration for the `compass` task and add new configurations for a `sass` task in our `Gruntfile.js`. This will involve changing Gruntfile.js in several places. Pay close attention to where you are making edits. If you mix up a curly brace or a comma, you'll get errors when you run `grunt` commands.

Configure `sass` task

Remove this `compass` config:

```
// Compiles Sass to CSS and generates necessary files if requested
compass: {
  options: {
    sassDir: '<%= yeoman.app %>/styles',
    cssDir: '.tmp/styles',
    generatedImagesDir: '.tmp/images/generated',
    imagesDir: '<%= yeoman.app %>/images',
    javascriptsDir: '<%= yeoman.app %>/scripts',
    fontsDir: '<%= yeoman.app %>/styles/fonts',
    importPath: './bower_components',
    httpImagesPath: '/images',
    httpGeneratedImagesPath: '/images/generated',
    httpFontsPath: '/styles/fonts',
    relativeAssets: false,
    assetCacheBuster: false,
    raw: 'Sass::Script::Number.precision = 10\n'
  },
  dist: {
    options: {
      generatedImagesDir: '<%= yeoman.dist %>/images/generated'
    }
  },
  server: {
    options: {
      sourcemap: true
    }
  }
},
```

And replace that entire configuration with this new configuration:

```
// Compiles Sass to CSS and generates necessary files if requested
sass: {
    options: {
        includePaths: [
            'bower_components'
        ]
    },
    dist: {
        files: [{
            expand: true,
            cwd: '<%= yeoman.app %>/styles',
            src: ['*.scss'],
            dest: '.tmp/styles',
            ext: '.css'
        }]
    },
    server: {
        files: [{
            expand: true,
            cwd: '<%= yeoman.app %>/styles',
            src: ['*.scss'],
            dest: '.tmp/styles',
            ext: '.css'
        }]
    }
},
```

Update `watch` task by adding a `sass` sub-task

Look for the `watch` task definition, which contains a list of things that `grunt` runs while it is running the development server. This task is what defines which files get watched for changes and what happens when a change is detected.

Within the `watch` task definition, find the `compass` sub-task:

```
compass: {
    files: ['<%= yeoman.app %>/styles/{,*/}*.{scss,sass}'],
    tasks: ['compass:server', 'autoprefixer:server']
},
```

Replace that definition with a new `sass` sub-task:

```
sass: {
    files: ['<%= yeoman.app %>/styles/{,*/}*.{scss,sass}'],
    tasks: ['sass:server', 'autoprefixer']
},
```

Update the `concurrent` task

Finally, locate the `concurrent` task definition. This task defines things that should happen simulatneously in order to speed up processing and building files. This is part of what makes the dev environment so fast.

The current `concurrent` task config looks like this:

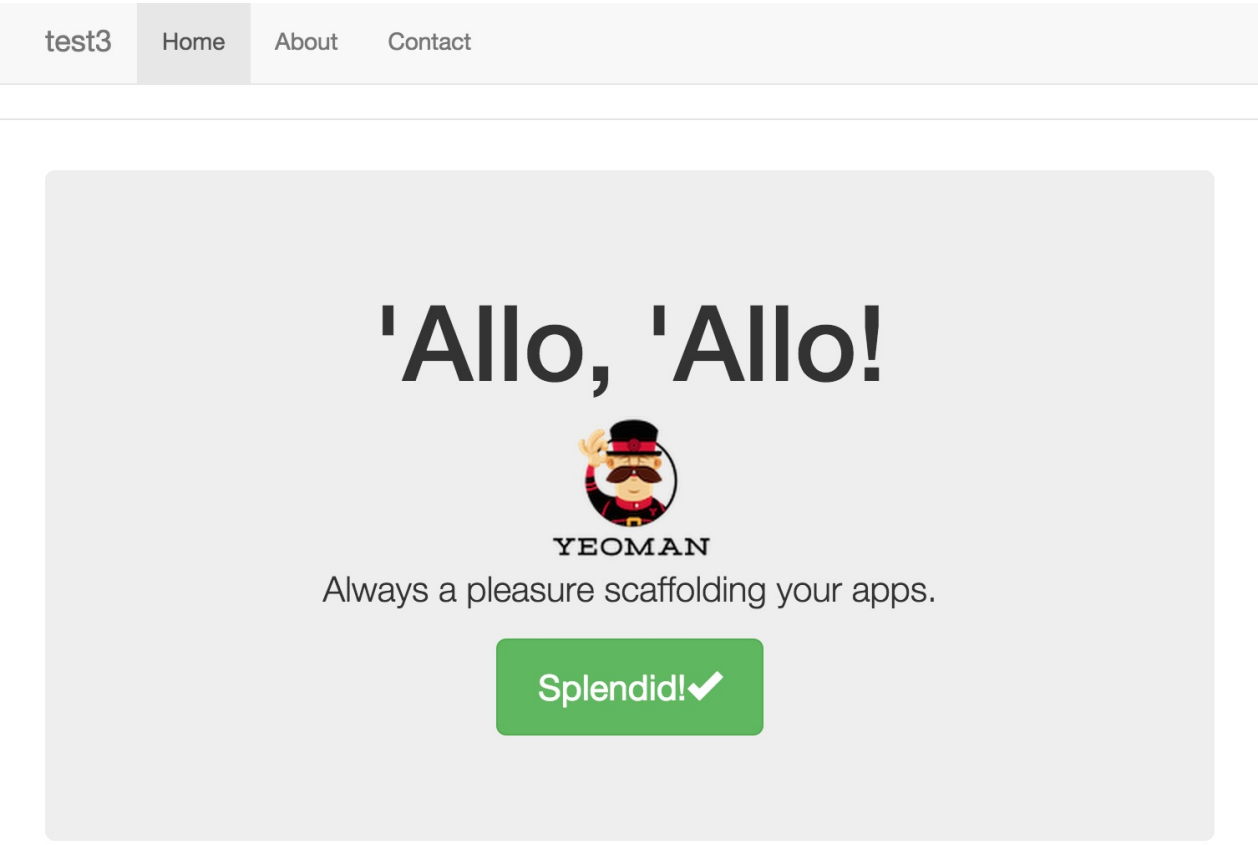
```
// Run some tasks in parallel to speed up the build process
concurrent: {
  server: [
    'compass:server'
  ],
  test: [
    'compass'
  ],
  dist: [
    'compass:dist',
    'imagemin',
    'svgmin'
  ]
},
```

Replace that config with a new `sass` config:

```
// Run some tasks in parallel to speed up the build process
concurrent: {
  server: [
    'sass:server',
    'copy:styles'
  ],
  test: [
    'copy:styles'
  ],
  dist: [
    'sass',
    'copy:styles',
    'imagemin',
    'svgmin'
  ]
},
```

Test your project

Now that you've updated `grunt` to use `grunt-sass` instead of `compass`, you can try out your new project. Run `grunt serve` to view your project in a web browser. You should see something like this:



If you see a page like that, then you have successfully updated your project to use `grunt-sass` and you are almost done setting up your project environment.

Exploring Your AngularJS Project

Before we configure your project to deploy, it's worthwhile to take a look at the files in your project and get to know where things are and what's what. Open your project in your favorite text editor so you can browse the files and peek at their content.

If you look at the directory listing of all your files, you'll see something like this:

```
.bowerrc
.editorconfig
.gitattributes
.gitignore
.jshintrc
.tmp
.travis.yml
.yo-rc.json
Gruntfile.js
README.md
app/
bower.json
bower_components/
node_modules/
package.json
test/
```

Those files mostly look like the files we had in the webapps we were previously experimenting with. This is because most projects that use a Yeoman template inherit similar-looking structures. This is handy because it makes it easier to learn new tools.

Many more modules

If you look inside `bower.json` you will notice there are many more components listed in it than there were in our other webapps. Yours should look something like this:

```
{
  "name": "yourAppName",
  "version": "0.0.0",
  "dependencies": {
    "angular": "^1.3.0",
    "bootstrap-sass-official": "^3.2.0",
    "angular-animate": "^1.3.0",
    "angular-aria": "^1.3.0",
    "angular-cookies": "^1.3.0",
    "angular-messages": "^1.3.0",
    "angular-resource": "^1.3.0",
    "angular-route": "^1.3.0",
    "angular-sanitize": "^1.3.0",
    "angular-touch": "^1.3.0"
  },
  "devDependencies": {
    "angular-mocks": "^1.3.0"
  },
  "appPath": "app",
  "moduleName": "yourAppName",
  "overrides": {
    "bootstrap": {
      "main": [
        "less/bootstrap.less",
        "dist/css/bootstrap.css",
        "dist/js/bootstrap.js"
      ]
    }
  }
}
```

Remember that we want to let Bower manage this file, so we won't edit it by hand. But looking in the `bower.json` file on a project is a great way to start to understand what components are involved.

Looking inside `package.json` is another way to see what is being used in a project. This file will list a whole different set of resources. We will interact more directly with many of these components throughout our work on this project.

App structure

Inside the `app/` directory is where all the code for your webapp is stored. If you look at the structure within that directory, you will see something like this:

```
.buildignore
.htaccess
404.html
favicon.ico
images/
|----yeoman.png
index.html
robots.txt
scripts/
|----controllers/
|--------about.js
|--------main.js
|----app.js
styles/
|----main.scs
views/
|----about.html
|----main.html
```

We will work within this structure for the rest of the book. But before we get ahead of ourselves, let's take a quick stroll through these files and directories to see what's there.

`index.html`

Note that the `index.html` file still provides the starting point for a browser to interact with our webapp. This page is loaded first, and you will find that it still must set up the basics for viewing the page: It defines a viewport, doctype, sets up meta tags, and includes our scripts and styles. This looks a lot like the `index.html` file we used in our previous experiments, but the difference now is that most of our work in AngularJS will take place in our `views`, which are partial templates. This `index.html` file will rarely be touched once it's all set up.

Styles

The `styles` directory will work pretty much like it did in our previous webapp experiments. We will likely want to copy over the Bootstrap _variables.scss file and set up some partial stylesheets. We will work on that in the next chapter. For now we will leave the styles as-is.

Scripts

The `scripts` directory is where much of our work will take place. This is where all the Javascript we write for our app goes. By default we have an `app.js` file, which defines our application for AngularJS. The `app.js` file is used in several different ways, and we'll explore that a lot more over the coming chapters.

Also in the `scripts` directory is a `controllers` sub-directory. The `controllers` are what handles manipulating data and setting up the Javascript functions that are available to our `views`. By default, the Yeoman generator has set us up with two views: `about` and `main`. These two views each have a unique controller, so within the `scripts/controllers/` directory we find two corresponding Javascript files.

Views

Finally, we come to the `views` directory. Inside are two `.html` files. If you look inside those `.html` files you will notice that they are not actually entire HTML documents. They are, in fact, "partials" -- meaning that they are partial HTML templates. This is what AngularJS uses to render new HTML into the overall single page app structure. These parts of HTML files will be brought into our `index.html` file as the user interacts with our app.

The best way to get a feel for how this is going to work is to make a few changes and see. So that's what we'll do next.

Change Making

In order to get a little taste of how the AngularJS views system works, we will make some simple changes to our views.

First, open the `app/views/main.html` file. Notice that it contains the familiar HTML defining the jumbotron and "'Allo! 'Allo!" message we see on so many Yeoman templates. Change some of the text to custom text of your own. (Don't overthink it!)

Notice that when you save changes to this file, your web browser preview will update. (You did run `grunt serve` in your project directory before starting to make changes, right?)

Since this project has Bootstrap connected, you can use Bootstrap-specific styles or HTML structures to modify your page. Try making a few changes.

Once you're satisfied that you see how `main.html` is working, open `app/views/about.html` and make some changes there, too. Click back and forth between the Home and the About screens. Do you see how your changes are being inserted into a very specific place in your application? This is the area that your views have to work with.

If you feel like you're running on steam and you need to conclude this phase of our project, feel free to continue. But if you'd like to experiment with a simple AngularJS-powered enhancement to a template, then try this extra stretch goal.

Stretch Goal: Make an adding machine.

To test out some of the core features of AngularJS, we will recreate the example we had in the previous chapter.

To do this, we will add a couple of `<input>` elements and then do a little math using the AngularJS template tags.

Paste this code somewhere in either your `main.html` or `about.html` files:

```
<div ng-app ng-init="firstnum=1;secondnum=2">
  <input type="number" min="0" ng-model="firstnum">
    plus
  <input type="number" min="0" ng-model="secondnum">
    equals
  <b>{{firstnum + secondnum}}</b>
</div>
```

We will get into these concepts in more depth in coming chapters, but for now, here is a brief explanation of what's happening in that code (and this matches with the information we saw about Models in the previous chapter).

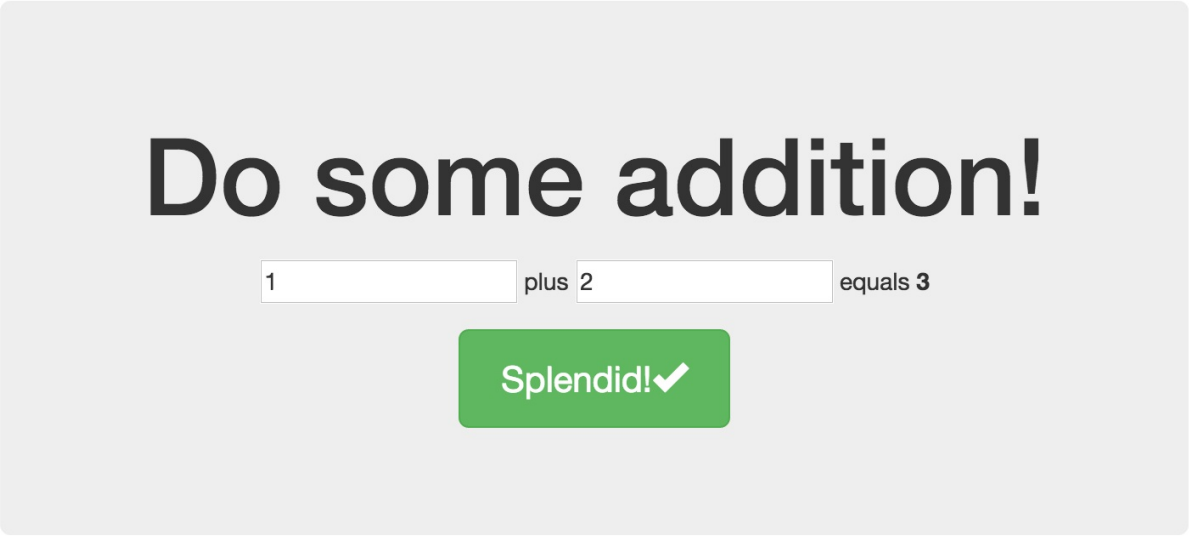
First, we make a `<div>` element and we add the `ng-app` attribute to let AngularJS know that inside this `<div>` is going to be an AngularJS app.

Then we add in a couple of `<input>` elements. Each input defines a new variable, which AngularJS calls a "model", so we use the `ng-model` attribute to assign a name to whatever value gets typed into those `<input>` elements.

Finally, we output the sum of the two numbers. AngularJS templates use the double curly brace (`{{ ... }}`) syntax to output the value of a variable to the template. Hence, the `{{firstnum + secondnum}}` line. Notice that output is going to return a number. We could put whatever tags around it we want. Right now, there are `<b>` tags around it, but you could just as easily turn it into a Bootstrap label by changing it to something like this:

```
<span class="label label-success">{{firstnum + secondnum}}</span>
```

When you preview this in your browser you should see something like this:



Type in two numbers and the sum total will change on its own. This is one of the cool things about AngularJS. We will dive into much more as we continue to explore.

Configure `grunt-build-control`

For the most part, we will configure `grunt-build-control` just like we did for our webapp experiments. We will still be deploying our AngularJS sites to Github Pages because although we are creating much more complex Javascript applications, we are still only building a static website that requires no dynamic server to deliver. This is another of the amazing things about working with Javascript webapps.

Install `grunt-build-control`

The `grunt-build-control` component must be installed into each project where you want to use it. Install it as usual and don't forget to add the `--save-dev` flag so your project remembers that you are using `grunt-build-control`. Run this command to install `grunt-build-control`:

```
npm install grunt-build-control --save-dev
```

Let Grunt know that `grunt-build-control` is available

The `generator-angular` template uses `jit-grunt` to manage dependencies for your Grunt tasks. In this case, look for the definition that resembles this code:

```
// Automatically load required Grunt tasks
require('jit-grunt')(grunt, {
    useminPrepare: 'grunt-usemin',
    ngtemplates: 'grunt-angular-templates',
    cdnify: 'grunt-google-cdn'
});
```

Modify that `require()` definition to add a `buildcontrol` line, like this:

```
// Automatically load required Grunt tasks
require('jit-grunt')(grunt, {
    useminPrepare: 'grunt-usemin',
    ngtemplates: 'grunt-angular-templates',
    cdnify: 'grunt-google-cdn',
    buildcontrol: 'grunt-build-control'
});
```

Configure the `buildcontrol` task

You also need to create the task object inside of `grunt.initConfig()`. To keep things easy, I advocate placing this definition right before the definition for the `watch` task (which is what Grunt uses to know when you've changed files when you are running the local server).

This should looks something like this:

```
  // Define the configuration for all the tasks
  grunt.initConfig({

    // Project settings
    yeoman: appConfig,

    buildcontrol: {
      options: {
        dir: 'dist',
        commit: true,
        push: true,
        message: 'Built %sourceName% from commit %sourceCommit% on branch %sourceBranc
h%'
      },
      pages: {
        options: {
          remote: 'git@github.com:your_github_user/your_webapp.git',
          branch: 'gh-pages'
        }
      }
    },

    // Watches files for changes and runs tasks based on the changed files
    watch: {

    ... more code ...
```

Once you've completed these steps you may commit your code and deploy it.

Commit and deploy

Commit your code using the same process you have before. Once you commit you should run:

```
grunt build
```

and then

```
grunt buildcontrol
```

Once the `buildcontrol` command completes, you can visit your project on Github Pages to see the files deployed. You should see the same thing you saw in your local development browser, and your addition calculator (if you went that far) should be fully functional.

Conclusion

You have now bootstrapped a new project skeleton for your AngularJS webapp. You have modified the initial project skeleton to use `grunt-sass` instead of `compass`, and you have configured deployment scripts to use `grunt-build-control` to deploy your files to Github Pages whenever you wish.

You have explored the files created by the Yeoman generator, and you have scratched the surface of making cool AngularJS apps.

You are well on your way to creating a fresh, new webapp. Congratulations!

See you in the next chapter.

Adding Data to the App

Most websites make use of a custom backend component. For example, many websites use a server-based application to manage business logic and data, which is then stored in a database that also resides on some remote server. The web frameworks we discussed a couple chapters back are largely designed to make that process of connecting data, logic, and template rendering much easier for developers.

For our purposes here, we do not actually require a custom backend component. We can create a useful application for our users that leverages local storage technology to keep user preferences and settings, and leverages third party data APIs to provide all the interesting content for our user. We can do all of this with no custom backend "engine" or database or anything else. We require only a static web server that can deliver our HTML, CSS and Javascript files to our users.

The big picture concept

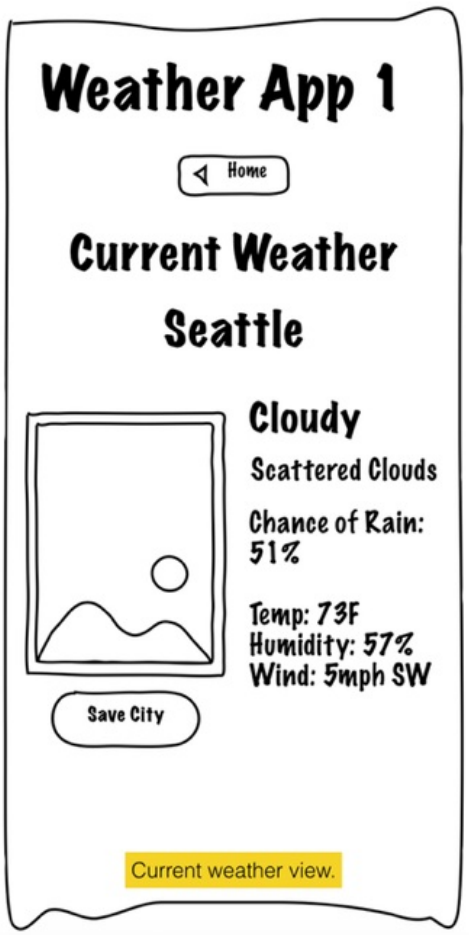
For this project, we will build a simple weather forecast application. It will allow users to enter their city, state, and/or country to select where they wish to see a forecast from, and then it will show them the current weather and a forecast for some number of hours or days.

We will use OpenWeatherMap.org to provide our data. OpenWeatherMap (OWM) is a site that seeks to provide weather information in the same way that OpenStreetMaps.org provides street map information or Wikipedia provides encyclopedic information. OWM makes an API available for free to developers that will suit our needs for this educational project.

Ultimately, we will provide a way for users to store a list of cities so they can easily return to our app and pull up their preferred weather forecasts. We will do this using local storage techniques, which means the data will not sync for users between browsers or devices. For now, this will be an acceptable limitation (although it's easy to work around that with a little server-side storage).

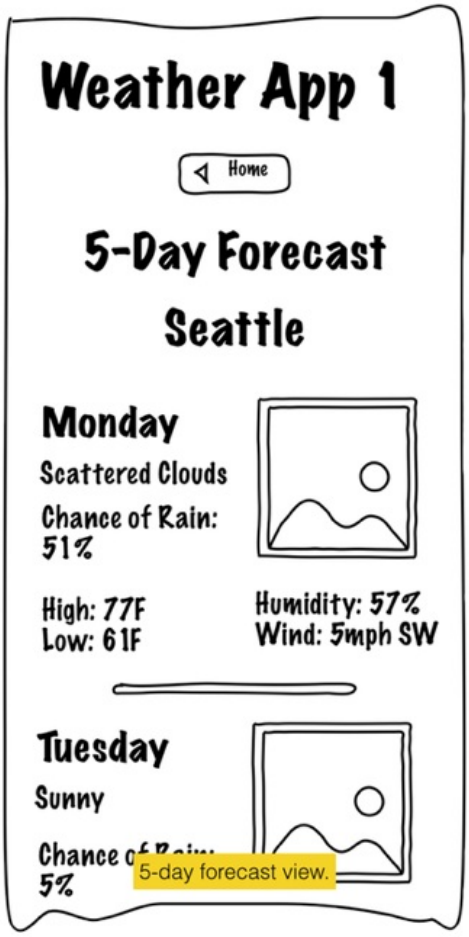
Here are some wireframe illustrations of the webapp we will work on for the rest of this book:

Home view with no saved cities.



Current weather view.

Home view with saved cities.



5-day forecast view.

This set of wireframes shows three different views (and two states of the "home" view), which we will build out over the next few chapters. There are many ways to enhance and otherwise customize this app to allow you to explore the areas of webapp building that you are most interested in. You can work on your interface design skills, experiment with animation to enhance the experience, or continue to build out a more robust weather application that can do more for the user. You can even use this basic app to build something like DoINeedAnUmbrella.com or other weather-based jokes and artwork.

Put the data in the app

Whatever else you decide to do with your weather application, it is essential to set up the data piping so that you can start to make use of all this rich weather information you can receive from OWM. In order to do that, you'll need to have an understanding of how an API works, how you can use it, and what you can do with it. We will cover all this and more as we set up your data models in this chapter.

An Introduction to Data APIs

The topic of Data APIs is really too broad to be covered thoroughly here. It is a topic full of nuance and importance, especially for developers creating Data APIs. When designing an API, it is as complex as designing a framework or code library: You are trying not only to solve a problem, but to solve it in a way that is understandable and accessible to everyone.

Luckily, we do not need to design an API for our purposes here. We need only to "consume" an API: That is, we will make requests to an API service and we will receive responses from that service that contain data we can use in our Javascript application. This process requires us to get to know the shape of some data returned from a chosen service, but it does not require us to understand the finer points of how APIs are put together.

Just so we aren't diving in completely blind, we will review some of the basics of how APIs work so that we can more easily understand the directions we see in our chosen API's documentation.

Websites for machines

If you compare human beings and computers, you will notice many differences. In fact, there is not much in common between what a human hopes to get from the world and what a computer hopes to get from a world. (Mostly because computers don't "hope".)

When trying to understand the concept of Data APIs that run on the web, it can be tough to get a mental image of what's really happening. The explanation that has always helped me is to imagine building a website for machines.

When building a website for humans, we care about things like information architecture and clear presentation of content that utilizes visual design to reinforce hierarchy, relationships, and meaning. Machines (computers) also need a solid information architecture, but the way we present data for machines is very different from how we present data for users.

To give machines what they need, APIs respond with information that is structured so it can be interpreted by whatever software is making the request. Typically, this means JSON or XML formatting, which are discussed below. Both of these formats lack the visual component that is essential to delivering information for humans. Machines are terrible at interpreting graphic design, but they are very good at keeping track of intricate hierarchies as long as the structure is consistent and syntactically correct.

When we use APIs we are asking our software to do some web browsing for us, and the information it retrieves makes our own application much more valuable to the humans using it.

REST

There are many kinds of APIs, but the dominant architecture on the web today is called "REST". You may see references to "RESTful APIs" or "REST-based APIs"—these are all ways of talking about an API that uses REST architecture.

REST stands for "Representational State Transfer", and although it sounds quite complex, it's something that you encounter every day. REST is the way the web works, and the concept was, in fact, defined by Roy Fielding (co-founder of the Apache web server project) as he worked on defining the HTTP protocol. REST feels natural on the web because REST is the way the web works.

Requests

REST is an architectural principle based on the client-server model. REST assumes that there is a client (web browser, Javascript app, mobile app, etc.) that wishes to consume data from a server (web server, API service, etc.).

When we make a request to a web server or API service, we are essentially sending our own file of information. The request is structured in a way similar to any HTML response we get. Requests have a "method", "headers", and a "message body". Each of these parts of the request is taken apart and inspected by the web server (or API service) in order to understand how to respond.

Methods

Methods are the "verbs" of the web. RESTful services use the base methods defined in the HTTP specification to handle data. These methods are:

- **GET**—Retrieves information for the client without altering any data on the server. This is commonly known as the "read" method.
- **POST**—Creates an entirely new record or data object on the server based on information sent by the client. This is often called the "create" method, and it is the most common method for HTML forms to use in order to send data back to the server.
- **PUT**—Updates the specified record with the data contained in the request. This is known as the "update" method.
- **DELETE**—Deletes the record specified by the client on the server. Most people don't

feel the need to rename the "delete" method.

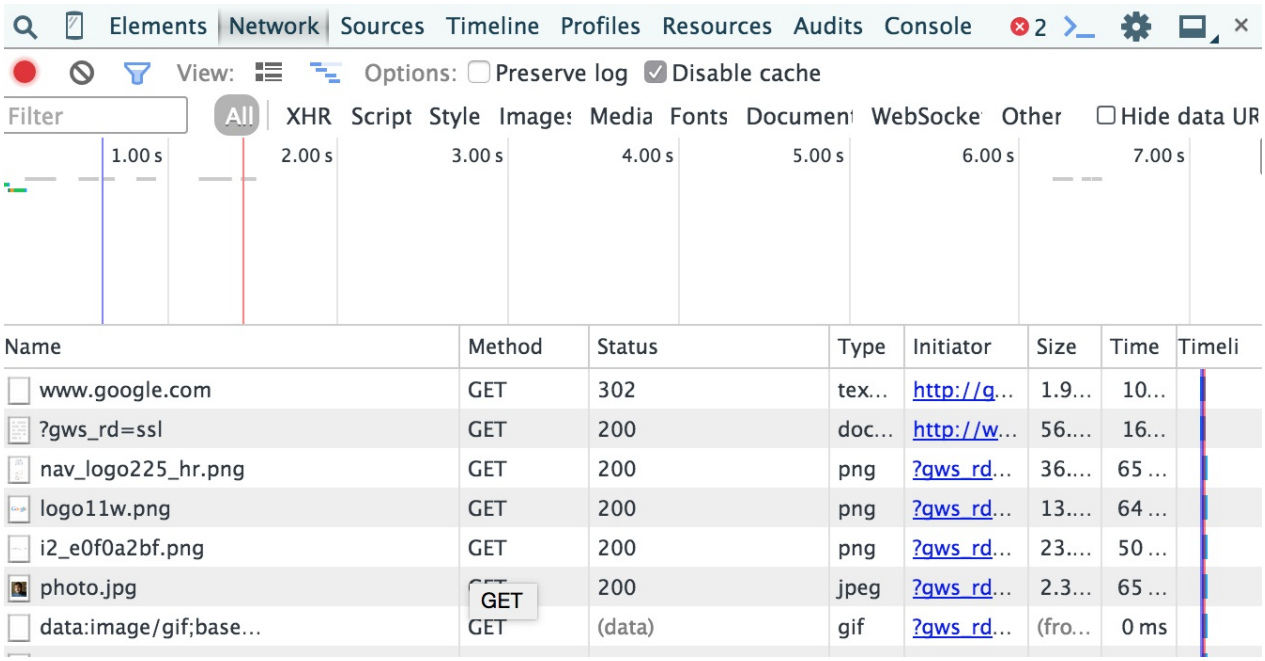
In working with any data content, we often use the CRUD acronym to describe basic functions of handling data: Create, Read, Update, Delete. The basic methods of HTTP mirror this concept very well, making them efficient at handling our data management needs.

Headers

In any request, the headers are of importance. Headers contain all sorts of data used by the server to determine the proper response. Each header has a meaning and an impact on the request, but for the most part we never pay much attention to the headers being sent each time we access a website. Here is a snippet of the headers sent when requesting `google.com`:

```
GET / HTTP/1.1
Host: google.com
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_4) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/43.0.2357.134 Safari/537.36
X-Chrome-UMA-Enabled: 1
X-Client-Data: CIm2yQEIgfjbJAQiptskBCMG2yQEI7YjKAQifl8oB
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

You can easily see the request is using the GET method, and you can probably figure out quite a few more details about the request (language, browser, etc.). You can view the headers used in any request made by your web browser by looking at the `Network` tab in the Chrome developer tools:



Click any of those requests and you'll see a summary of the request and response including all headers and data. (Please note: Being able to look at what your browser is doing like this can be very helpful when trying to figure out why your API-dependent app is not working properly.)

Sometimes when working with data APIs you will need to set a header. For example, authorization to use an API may involve you sending your API token as an HTTP header. This header is sometimes specified like so:

```
Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b
```

Most libraries you use to consume data APIs will provide a convenient way to set a header to a specified value. Many frameworks are specifically designed to minimize how much manual management of headers you must perform. When you set configuration options and other setup details you may actually be specifying that the framework or library will include or exclude specific headers. It's worthwhile to be cognizant of what headers are being used in your API requests.

Message Body

The request contains a message body, which contains any data the client is sending to the server. In the case of POST and PUT methods, data is sent to the server for interpretation and then handling (usually it is being saved into the database).

When HTML forms are sent using the GET method, the data from the form fields is serialized into "query string parameters". These are attached with a question mark (`?`), like so:

```
http://domain.com/catalog/search?q=slippers&brand=fuzzybunny&gender=male&size=14
```

In the example above you can see several query string parameters have been defined: `q`, `brand`, `gender`, and `size`. It is very common to use query string parameters to represent things like search filters. On many ecommerce websites you will notice that the filters you choose in the catalog are easily identifiable in the query string parameters for the catalog page you are viewing.

The URL above could have come from an HTML form with fields called `q`, `brand`, `gender`, and `size` that was sent using the GET method. This is handy in cases where we want to be able to copy/paste a URL and have our friend see the same results (in this case, a selection of very large fuzzy bunny slippers).

In other situations, it would not be proper to send the data contained in an HTML form as part of the URL. The URL of a request can never be encrypted, since it must be used to route all of the pieces of the request from client to server. Therefore, any data sent using the GET method is going to be visible to all the network nodes those bits pass through. In the case of sensitive information (such as passwords or credit card numbers), it's much better to use POST and PUT so that the message body content can be properly encrypted via HTTPS and much less visible to any nefarious agents.

State(less)

REST is "stateless". (So is the web.) This means that each request to a web server (or a REST API service) must define all of the parameters to receive the desired response. In RESTful services, all of these parameters are packed into the URL and form data. Imagine the number of times you've seen a URL that looks like this:

```
http://domain.com/profile/username/edit/
```

Presumably, if `domain.com` follows best practices, that URL would allow us to edit the profile for the user with the specified `username`. The server responding to this request would be able to fetch the data for the specified `username` and serve us back a web page designed to allow us to edit that information. No extra information is required to receive our desired response, and no information from this request will be used in any other response the server sends.

When we edit the information on the page and click submit, we are making a brand new request to the web server. This request defines all of the information the server requires in order to respond to our request. This time, when we send data to the server we include

"form data", which is the data collected using HTML `form` elements (`input` , `select` , etc.). This form data provides the information we want the server to put into the database for us.

Data

REST APIs respond with data that can be contained in an HTTP response. This generally means text. The text can return a URL reference to a media file, of course, but then you must actually use that URL properly in your HTML to allow the user to see that media. You might insert an image tag, or a video tag, or whatever other HTML structure is needed to display the media.

The text responses returned by REST APIs tend to either be formatted as JSON (Javascript Object Notation) responses or XML (eXtensible Markup Language) responses. The trend today is toward JSON since it is more easily consumed by Javascript applications, and for our purposes we will expect JSON responses.

Here is a sample API response from OpenWeatherMaps.org for the query `api.openweathermap.org/data/2.5/weather?q=seattle,wa,us&units=imperial&id=4717560d` :

```
{
  ...

  "weather": [
    {
      "id": 701,
      "main": "Mist",
      "description": "mist",
      "icon": "50d"
    }
  ],

  ...

  "main": {
    "temp": 61.05,
    "pressure": 1018,
    "humidity": 63,
    "temp_min": 57.2,
    "temp_max": 64.99
  },
  "wind": {
    "speed": 4.54,
    "deg": 220
  },
  "clouds": {
    "all": 90
  },

  ...

  "id": 5809844,
  "name": "Seattle",
}
```

In this example, I have removed a few pieces of the response to make it a little easier to read. (The missing pieces are indicated by the `...` lines.)

As you read through this example response, it's pretty easy to see some interesting data fields. We can see that this data object contains several "root level" or "base" attributes: `weather`, `main`, `wind`, `clouds`, `id`, `name`. This is the "current weather" data for Seattle. We are given the ID, which is a more reliable way of not getting confused with the other cities called "Seattle". We also have a weather summary (at the moment "Mist"), current temperature (61.05F), cloud coverage (90%), and more. With this information, we can build a solid weather report.

When this data object is received by our AngularJS app, AngularJS will parse the information into a Javascript object. We can call that object whatever we want (perhaps `weatherReport` ?) and we can pipe that into our view templates like this:

```
<h1>Weather report for {{weatherReport.name}}</h1>
<p>Current conditions: {{weatherReport.weather.main}}, {{weatherReport.main.temp}}F</p>
```

And then our users would see this rendered in our webapp:

Weather report for Seattle

Current conditions: Mist, 61.05F

This is just a quick preview of how handy it is to use a tool like AngularJS to interact with data APIs that deliver JSON responses. We will explore much more about how to make this all work in our webapp over the next few pages of this chapter.

Prerequisites for Using Data APIs

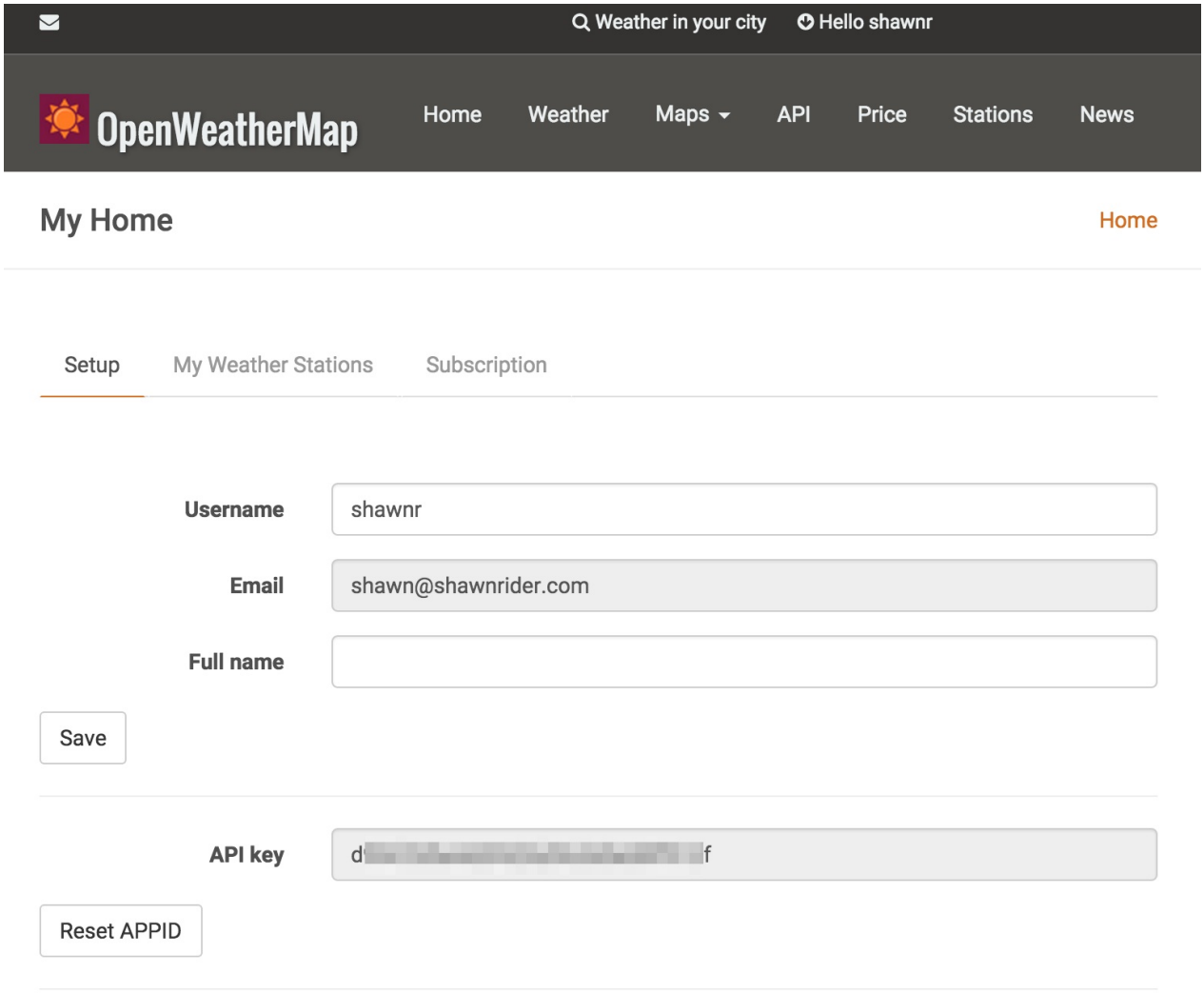
Most APIs you might leverage in a Single Page Application (SPA) will require you to sign up, and possibly even go through some approval process, in order to consume their data. For the purposes of experimentation, it's good to stick with "open" data APIs, many of which are provided by various government agencies, non-profit organizations, and various other outlets. Many for-profit companies offer very liberal API usage to developers, while others restrict their API access to approved partners.

Open Weather Map

For the purposes of this book, I will leverage the data available from OpenWeatherMap.org (OWM). This is a service run by a private company, but with very liberal usage policies. You can learn more about OpenWeatherMap.org on their About Us page. (Please note: Neither I nor this book is in any way affiliated with OWM. For the purposes of the work we will do here, you may substitute any other REST API service that will return JSON. There are a lot of those.)

In order to use OWM's API, you must register for an account. When you register for an account, you will be given an API key, which you can find on your "My Home" page after you have logged in. (Click the "Hello" link in the upper right of the site if you can't find your "My Home" page.

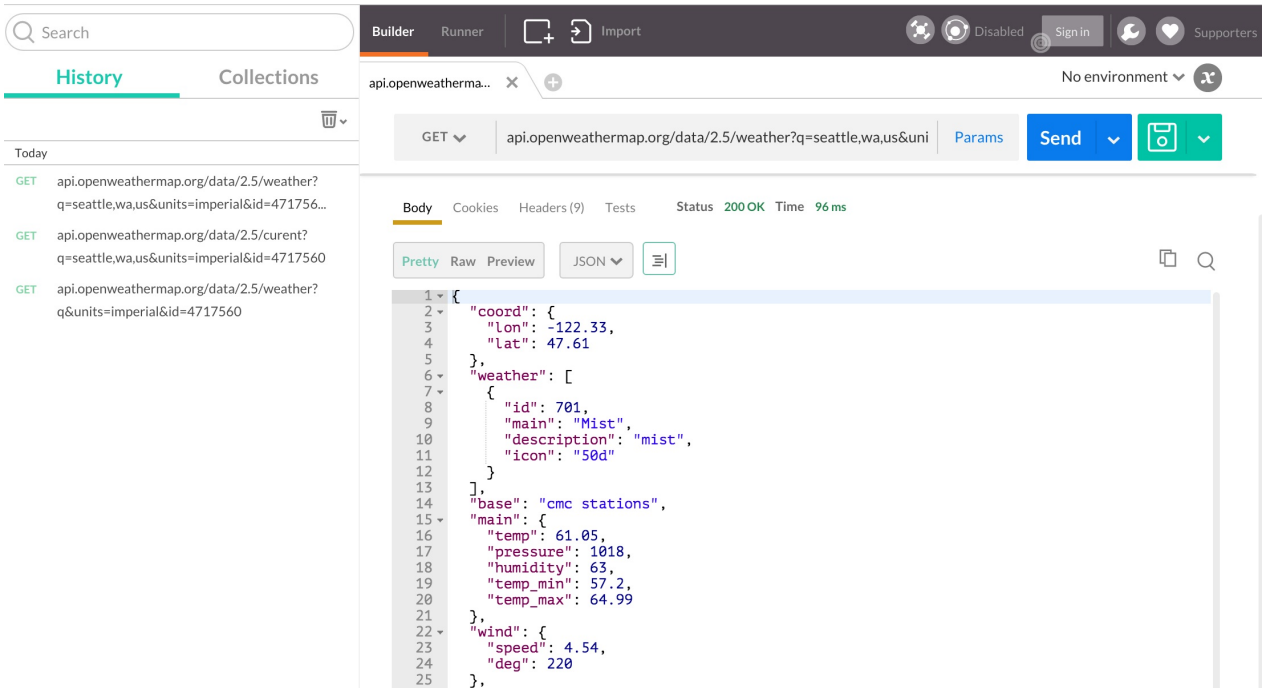
The My Home page looks like this (please note the API Key field):



Now that you have your API key, you are ready to continue with the work in this chapter using the OWM API.

Exploring the API

Whenever I begin using a new data API, I like to spend a little time exploring the API documentation and trying requests using an API browser tool like Postman for Google's Chrome browser.



Postman allows you to construct HTTP requests to any API. You can set necessary headers, query parameters, form data, and more. Results are clearly displayed so you can truly begin to understand the shape of the data that comes back from the API. Each API will deliver a different type of data object, and all the attributes contained in these responses become available to you inside your AngularJS app once you set up your data models properly.

That means you can use an API browser tool to get a clear view of what information exists in the data coming back from your API service. Since the API data objects are translated directly into your AngularJS models, you can easily see how to access the specific information you require for your purposes.

These tools also help you form proper API requests. If your request is not properly formed, then you will not get the data you need back from the API service. Luckily, AngularJS makes it easy to create base request templates so you do not need to constantly be writing and rewriting requests, but it can still be tricky to work out how to form your request in the first place. A good API browser tool will help you formulate correct requests and will allow you to know exactly what you are getting back. You can even use it to help test edge cases and learn how the API responds if, for example, you send some data value it does not expect.

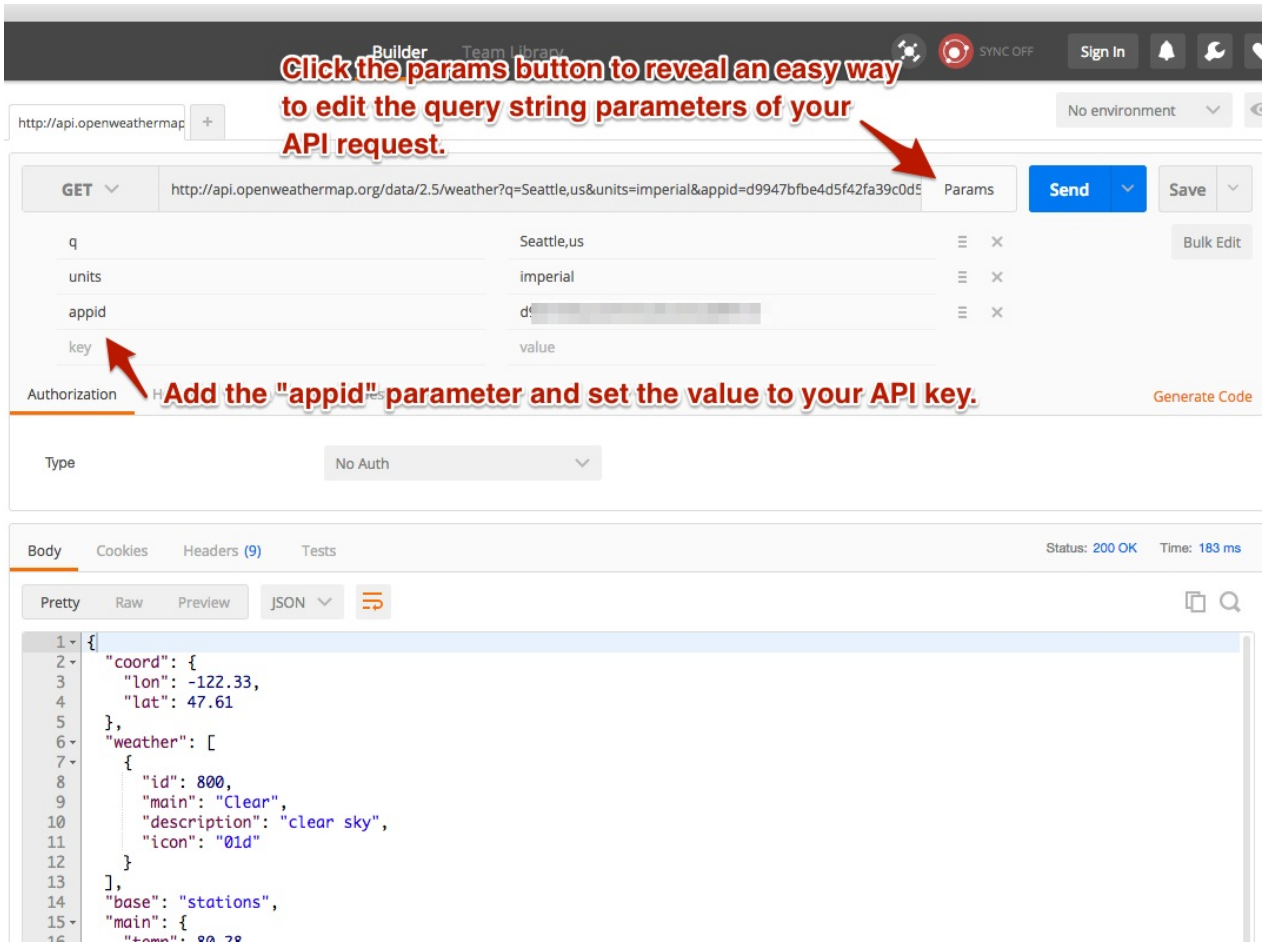
The Postman Documentation site has a lot of useful information, tutorials, and guides for using Postman to explore API services. You are encouraged to read up on how to use this tool, or find a similar tool you prefer and use that one. It is very essential to have a way to view the data coming into your app, and to test your app's API queries, apart from your custom Javascript code.

We will be using the OWM "current weather" feature today. You can read more about how this feature works on the current weather documentation page. More information about how the OWM API works can be accessed from their API documentation homepage: http://openweathermap.org/api.

You will want to review those pages. Try using Postman to explore the current weather API endpoint. Plug this URL into Postman and do a GET query to see results like those shown above:

```
http://api.openweathermap.org/data/2.5/weather?q=Seattle,us&units=imperial
```

(Don't forget that you'll need to set your API key using a query string parameter. Other API services will require you to configure Postman to authenticate in other ways, so pay attention to those requirements if you're using a different API.)



Try changing the units to "metric" or alter the location to see results for a different city. You should be able to see clear changes in the data returned. Can you make the API send results for Paris, France? How about Paris, Texas? Keep exploring until you feel like you have a basic understanding of how this API works.

Using other APIs

You can use the patterns described in this book to use any other REST API that serves JSON results. There are many, many APIs that fit that description. Some APIs require you to do more arduous authentication or to have your app approved before you will be granted developer privileges. Other APIs have steep charges for using them, and although you may be able to develop against them reasonably it would be prohibitively expensive to release a website using that API. These considerations and more should inform your decision as you look for APIs.

As you look for APIs to use in your projects, some of these resources may help:

- US Government's 18F project to encourage developers to use US Government Data APIs
- Mashape Directory of Public APIs
- Programmable Web API Directory

Create Data Models

In order to use the data from our API in our webapp, it is necessary to create data models. There are several way to accomplish this goal, but we will explore a technique that is both commonly used and well-suited to our needs. We will use the `ngResource` module to create a connection between our data API and the data we use in our views.

We will use the Yeoman `generator-angular` module to create stub files of Angular Services and Controllers. By using the Yeoman generator, we will have an easier time of making sure that all of our source files are being properly included in our application.

If you use the tools to help maintain your dependencies, then you will have a much easier time. If you choose to edit files directly you may find yourself getting errors and running into issues more often.

As you move through these steps, realize that you can repeat this process for as many API objects as you wish to use in your app. You can create resources that talk to entirely different APIs if you wish. Each resource represents a unique stream of data into a model that you can use in any of your application's views.

Generate a factory service

In order to manage data from the server, we will use a "factory service". That is, we will create an AngularJS Service using the `factory()` method. To generate this service, you will need to use the command line `yo` application. Use `cd` to change directory into your project root (the same directory as your `Gruntfile.js` is located).

Once you're in the correct location, run the following command and you should see similar results:

```
$ yo angular:factory current
    create app/scripts/services/current.js
    create test/spec/services/current.js
```

That command asks the Angular generator to create a new factory service in your app. It makes two files and adds references to those files in the appropriate places (including your `index.html`).

The factory we made is for the `current` data model. This model will contain the data for the "current weather" at a given location. This model will NOT contain the data for the "weather forecast" at a given location (that would be defined as another data model and factory service).

Configure your resource

Now that you have a `current` factory service in `app/scripts/services/current.js`, you need to add some logic to it so it does something. By default, the service that has been generated looks like this:

```
angular.module('yourApp')
  .factory('current', function () {
    // Service logic
    // ...

    var meaningOfLife = 42;

    // Public API here
    return {
      someMethod: function () {
        return meaningOfLife;
      }
    };
  });
```

We want to replace this with code that connects to our chosen API service. The file `app/scripts/services/current.js` should look like this:

```
angular.module('yourApp')
  .factory('current', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/weather?q=:location&units
=imperial&APPID=YOUR_API_KEY_HERE', {}, {
      query: {
        method:'GET',
        params:{
          location: 'Seattle,us'
        },
        isArray:false
      }
    });
  });
```

Let's take apart what's happening in that factory definition. First of all, you can see that we access the base `angular` object, and the `module()` method. We specify our app module: `yourApp`. Alter that to match with the name of your app.

This factory is named `'current'` and you can see that the function definition takes in `$resource` as a parameter. This is how we refer to the `ngResource` module within our Javascript code. This module provides us with the ability to write the `$resource` statement that we are returning as the "Public API".

First, we specify the base API URL. We can replace parts of this URL with variable names. In this case, I have replaced the actual "query" with `:location`. We could, for example, also replace the `units` value with a variable name so we could allow our users to switch between Metric and Imperial units.

Please also note that you must replace YOUR_API_KEY_HERE with your actual API key from OpenWeatherMap.org.

Once we have defined the base API URL, we can move on to specify a Javscript `object` that will contain all of the methods we wish to define as part of our `current` data model. By specifying a `query` object here, we are allowing ourselves to use the `query()` method on the `current` model. So when we use the `current` model in a Controller we can write:

```
current.query({location: 'Chicago,us'});
```

That line would query the OWM API service for the current weather in Chicago.

The `query` object in our `$resource` definition specifies a few other things, too:

```
query: {
    method:'GET',
    params:{
      location: 'Seattle,us'
    },
    isArray:false
}
```

The HTTP method that will be used to make the API request is GET, which is appropriate since this is a "read only" service. (We cannot alter the weather via API call... yet!) We also let AngularJS know that we do NOT expect these results to be a Javascript Array (as opposed to an Object) by setting `isArray:false`.

The interesting part of the `query` definition is the `params` object. The `params` object allows us to specify default values to be filled in if the `query()` method is called without those params specified. In this case, we are supplying `'Seattle,us'` as the `location` value so

that when the query is run with no parameters it still returns results. This is a useful way of providing user-friendly defaults and minimizing the likelihood for your app to break if not all params are specified.

If you wish for a param to be left blank if it is not supplied you can set the default to be equal to `null`:

```
params: {
    location: 'Seattle,us',
    bogus: null
}
```

Now what?

Now that you have your `$resource` defined as a factory service, you can use this `current` data model in any view you choose by adding the `current` service to your AngularJS Controller parameters. We will do that next.

Put the Data in the View

If you have constructed your `current` data model properly then you should now be able to use it in any Controller to pipe data into your webapp's views. In order to prove out our data integration, we will make a simple form interface so users can look up the weather for their city.

Set up `MainCtrl` controller in `main.js`

In the file `app/scripts/controllers/main.js` there is a definition for the `MainCtrl` module. This is the module that controls the data and functionality on the main page of our site. We want to get rid of the default code that Yeoman generated for us and replace it with something that looks like this:

```
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, current) {
    $scope.current = current.query();
  });
```

Let's take a look at this code more closely. First of all, we are once again appending to the `'yourApp'` module, so you'll need to replace that with the name of your app. In the definition of `MainCtrl` we added a few things to the function parameters: `$scope` and `current`.

As discussed in our previous overview of AngularJS, the `$scope` object is a special AngularJS object that manages the context of the view associated with this controller. The `$scope` object contains all of the variables defined for use in this view, as well as the different functions that can be added to HTML elements inside of our view. We will make extensive use of `$scope` in all of our controllers.

The other parameter references our `current` data model. Now that we've added it to the `MainCtrl` function parameters, we can reference `current` from inside `MainCtrl`. That's exactly what we do on the next line:

```
$scope.current = current.query();
```

This line sets a variable in `$scope` called `current` equal to the results of the `query()` method on the `current` data model. Since we are not supplying any parameters in our `current.query()` call, the API should respond with our default query (for 'Seattle,us').

We should now have access to the `{{current}}` variable in our views.

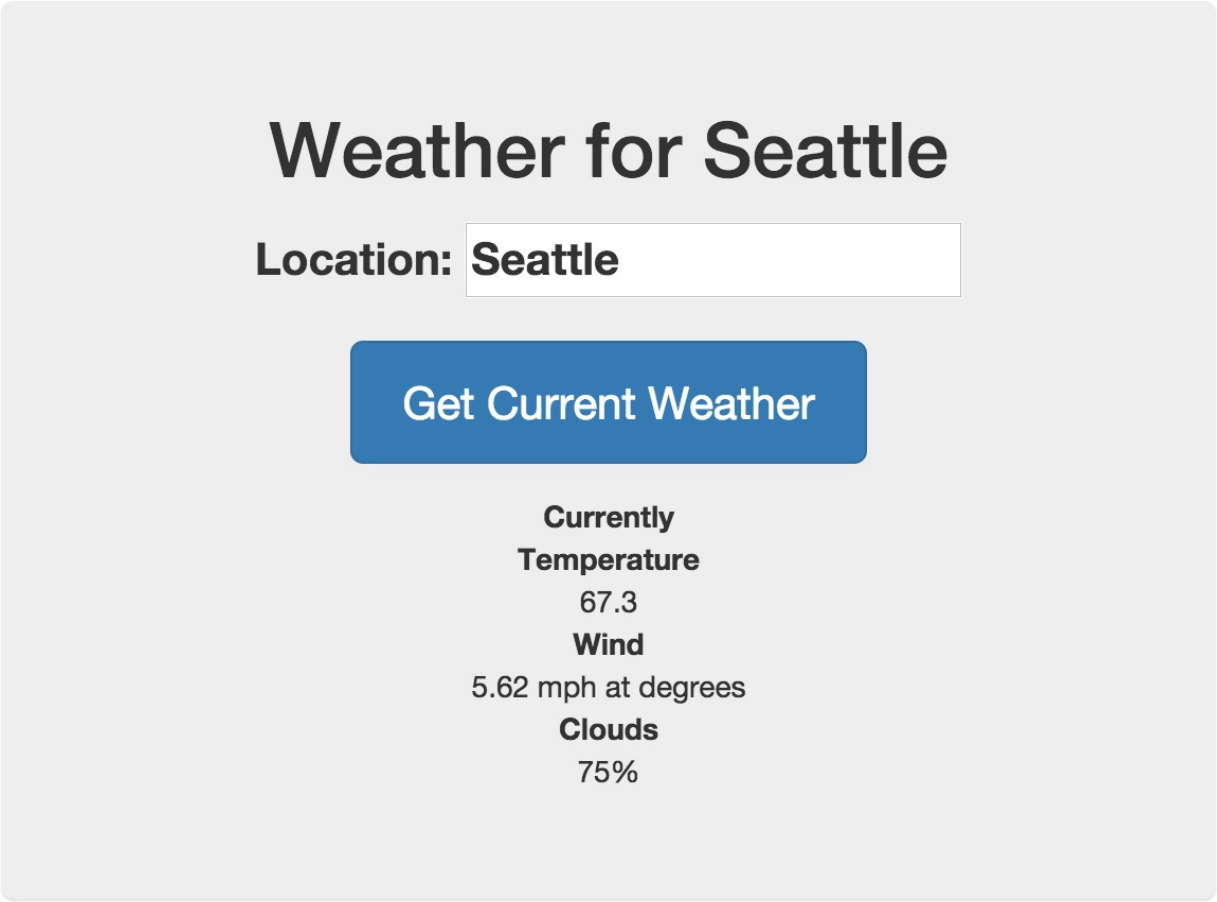
Set up the view

In order to display this information, we must modify our view so that it will show the data we have fetched from our API. Here is how the `app/views/main.html` should look in order to show some data from the OWM API:

```
<div ng-app class="jumbotron" ng-controller="MainCtrl">
  <h1>Weather for {{current.name}}</h1>
  <p class="lead">
    <div ng-init="location='Seattle'">
        <p>
          <label for="location">Location:
            <input type="text" name="location" ng-model="location">
          </label>
        </p>
        <p>
          <button class="btn btn-lg btn-primary" ng-click="refreshCurrent()">Get Curre
nt Weather</button>
        </p>
      <dl>
        <dt>Currently</dt>
        <dd>{{current.weather[0].main}}</dd>
        <dd>{{current.weather[0].description}}</dd>
        <dt>Temperature</dt>
        <dd>{{current.main.temp}}</dd>
        <dt>Wind</dt>
        <dd>{{current.wind.speed}} mph at {{current.wind.deg}} degrees</dd>
        <dt>Clouds</dt>
        <dd>{{current.clouds.all}}%</dd>
      </dl>
    </div>
  </p>
</div>
```

As you can see, there are variable outputs peppered throughout this HTML. The `<h1>` heading shows the name of the city where the current weather is coming from. The content shows values from inside the data returned by the API. You can see that we are using the same field names and heirarchy as we saw in the data results when we were testing them using the Postman API browser.

If you run your server and view your webapp in the browser, you should now see the default results of the 'Seattle,us' query:



Congratulations! It's been a long road to get here, but this is the first time we're really working with serious data and a living application. Well, almost living...

Refresh the data

Of course, one of the hallmarks of living apps is that you can do things with them, and right now if you type in the name of another city and click the "Get Current Weather" button, nothing happens. Right now, this button is not connected to anything. Notice in the code above the button contains a `ng-click` attribute:

```
<button class="btn btn-lg btn-primary" ng-click="refreshCurrent()">Get Current Weather
</button>
```

That `ng-click` attribute tells AngularJS that when this button is clicked it should execute the `$scope.refreshCurrent()` function. However, that function does not yet exist, so right now nothing happens. Let's fix that.

Since the HTML for our view already contains the `ng-click` attribute on our button, all we need to do is create the `$scope.refreshCurrent()` function in our `MainCtrl` inside of `app/scripts/controllers/main.js`. Here is what that controller will look like once we've

edited it:

```
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, current) {
    $scope.current = current.query();

    $scope.refreshCurrent = function(){
        $scope.current = current.query({
            location: $scope.location
        });
    };
  });
```

Notice that all we've done is add a function called `$scope.refreshCurrent`. This function takes no parameters, and it only has one line: It sets the value of `$scope.current` to the result of `current.query()`, and it sends `current.query()` a `location` parameter that is equal to the value of `$scope.location`.

This might seem quite complex, so let's walk through what happens in the view as the user interacts with the webapp.

First, the user loads the view. The `$scope.current` value is populated with the default data returned by `current.query()`. The user sees this default data displayed in their browser.

Second, the user types a new location name into the text input. Since AngularJS has already bound `$scope.location` to that text input, the value of `$scope.location` is updated constantly as the user types. However, since the user must click the button to refresh the data, the information being displayed does not change.

Third, the user clicks the "Get Current Weather" button, which executes the `$scope.refreshCurrent()` function. That function queries the `current` resource again, but this time it sends the `query()` method a `location` parameter:

```
$scope.current = current.query({
    location: $scope.location
});
```

Since `$scope.location` was already updated as soon as the user typed the new location into the text box, this new API request sends the updated location to OpenWeatherMaps.org, which responds with the new data.

Finally, since AngularJS knows that all the places where we display data are bound to that `$scope.current` variable, then it automatically updates everywhere that data is used in our view as soon as it receives the new data. We do not have to manually refresh the view or

update any HTML elements by hand, which is a major advantage to using a framework like AngularJS.

Now that we have this all connected, you should be able to try this in your web browser and see how it works.

Changes in this Chapter

Since this chapter involves several changes to several files, it can be tricky to make sure you've made all the required changes. Doublecheck your work against this page to make sure you didn't miss anything.

Note: Every refrence to `yourApp` should be replaced with the name of your webapp.

app/scripts/services/current.js

```
'use strict';

/**
 * @ngdoc service
 * @name yourApp.current
 * @description
 * # current
 * Factory in the yourApp.
 */
angular.module('yourApp')
  .factory('current', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/weather?q=:location&units
=imperial&APPID=d9947bfbe4d5f42fa39c0d5e08ff915f', {}, {
      query: {
        method:'GET',
        params:{
          location: 'Seattle,us'
        },
        isArray:false
      }
    });
  });
```

app/scripts/controllers/main.js

```
'use strict';

/**
 * @ngdoc function
 * @name yourApp.controller:MainCtrl
 * @description
 * # MainCtrl
 * Controller of the yourApp
 */
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, current) {
    $scope.current = current.query();

    $scope.refreshCurrent = function(){
        $scope.current = current.query({
            location: $scope.location
        });
    };
  });
```

app/views/main.html

```
<div ng-app class="jumbotron" ng-controller="MainCtrl">
  <h1>Weather for {{current.name}}</h1>
  <p class="lead">
    <div ng-init="location='Seattle'">
        <p>
          <label for="location">Location:
            <input type="text" name="location" ng-model="location">
          </label>
        </p>
        <p>
          <button class="btn btn-lg btn-primary" ng-click="refreshCurrent()">Get Curre
nt Weather</button>
        </p>
      <dl>
        <dt>Currently</dt>
        <dd>{{current.weather[0].main}}</dd>
        <dd>{{current.weather[0].description}}</dd>
        <dt>Temperature</dt>
        <dd>{{current.main.temp}}</dd>
        <dt>Wind</dt>
        <dd>{{current.wind.speed}} mph at {{current.wind.deg}} degrees</dd>
        <dt>Clouds</dt>
        <dd>{{current.clouds.all}}%</dd>
      </dl>
    </div>
  </p>
</div>
```

Data Conclusion

This ends our first foray into getting data inside of our app. Using a third party data API, we have successfully wired up a factory service that makes a data resource available to use throughout our application. Any controller that needs to leverage the current weather forcast may make use of the `current` service and will be able to easily query for weather data.

In order to achieve this, we have used `ngResource`, `$resource`, and OpenWeatherMaps.org. We now have a pattern established that can be followed to create additional resources, or to use this resource more in our webapp. Each resource contains its own configuration, methods, and API endpoints. You can leverage multiple API services inside a single app by creating separate resources for each one.

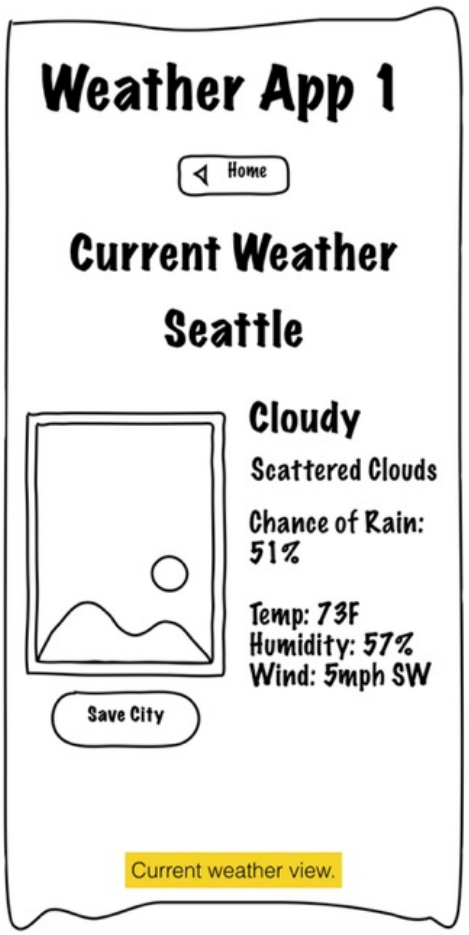
All of these possibilities are opened up with the simple architecture we've laid out here. AngularJS helps us keep our views updated, and makes it much easier to access our third party data APIs.

Now that we have data, we need to effectively make use of it. We have a "proof of concept" in place, and a pattern we can use to finish out our work, but we are not yet done. In the next chapter we will explore how to manage and work with data in ways that are useful to our users.

Managing and Using Data in Your App

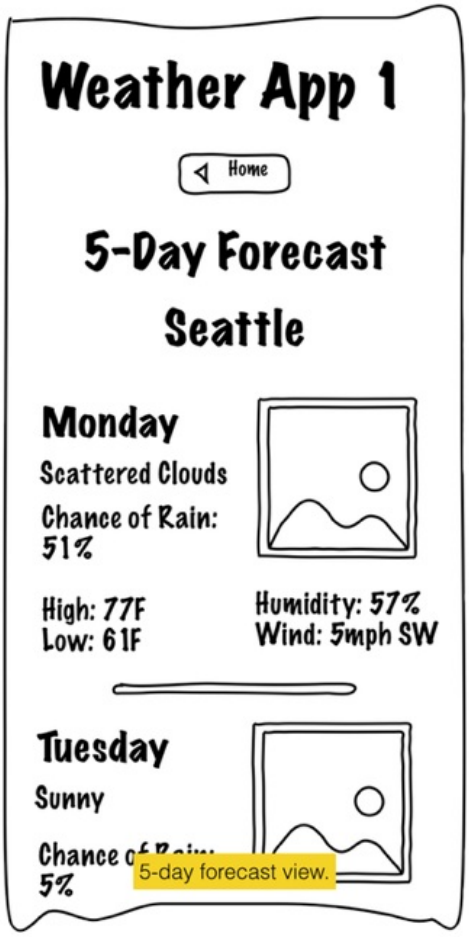
In this chapter we will dig into more of how to manage and use data in our app. We will add the remaining functionality that will make our app a real, functional weather tool for our users. As in the previous chapter, we will be working toward our basic concept of a weather app. This concept is outlined in the four wireframes we used previously:

Home view with no saved cities.



Current weather view.

Home view with saved cities.



5-day forecast view.

So far, we have laid the groundwork for how we can call API endpoints on the OpenWeatherMaps.org API in order to get data into our app. However, in addition to displaying the current weather in a location, we want to provide a more user-friendly experience.

The core features of our app will be:

1. Find a city you wish to view weather for.
2. View the current weather for that city.
3. View the forecast for that city.
4. Save that city into a list of quick-links so the user can select from their list of preferred locations rather than typing in the city name each time.

These are the features that will make up our core application, and once we have these, we will have an app that offers some real utility to our users. This is what is called in the web industry an "MVP" or "Minimum Viable Product". It is not a stopping point in terms of our app's entire life, but it is a major milestone where our app is delivering enough basic functionality to be useful.

We can start to share our MVP and get feedback from users about what to improve next. You probably already have ideas about things that would be handy to have in the app (like removing favorites? or setting a default city so users can see weather data right away?).

Covering ground

In this chapter we are going to cover a good amount of ground. We will create a couple of new views, a few new model data resources, and we will rearrange where we put the current weather functionality we have already built. For the most part, we are repeating what we have done previously, and repetition is very valuable at this point. If you feel you need more repetition, you are encouraged to add an additional view using one of the many other endpoints available through OpenWeatherMap.org (historical data, weather station data, etc.).

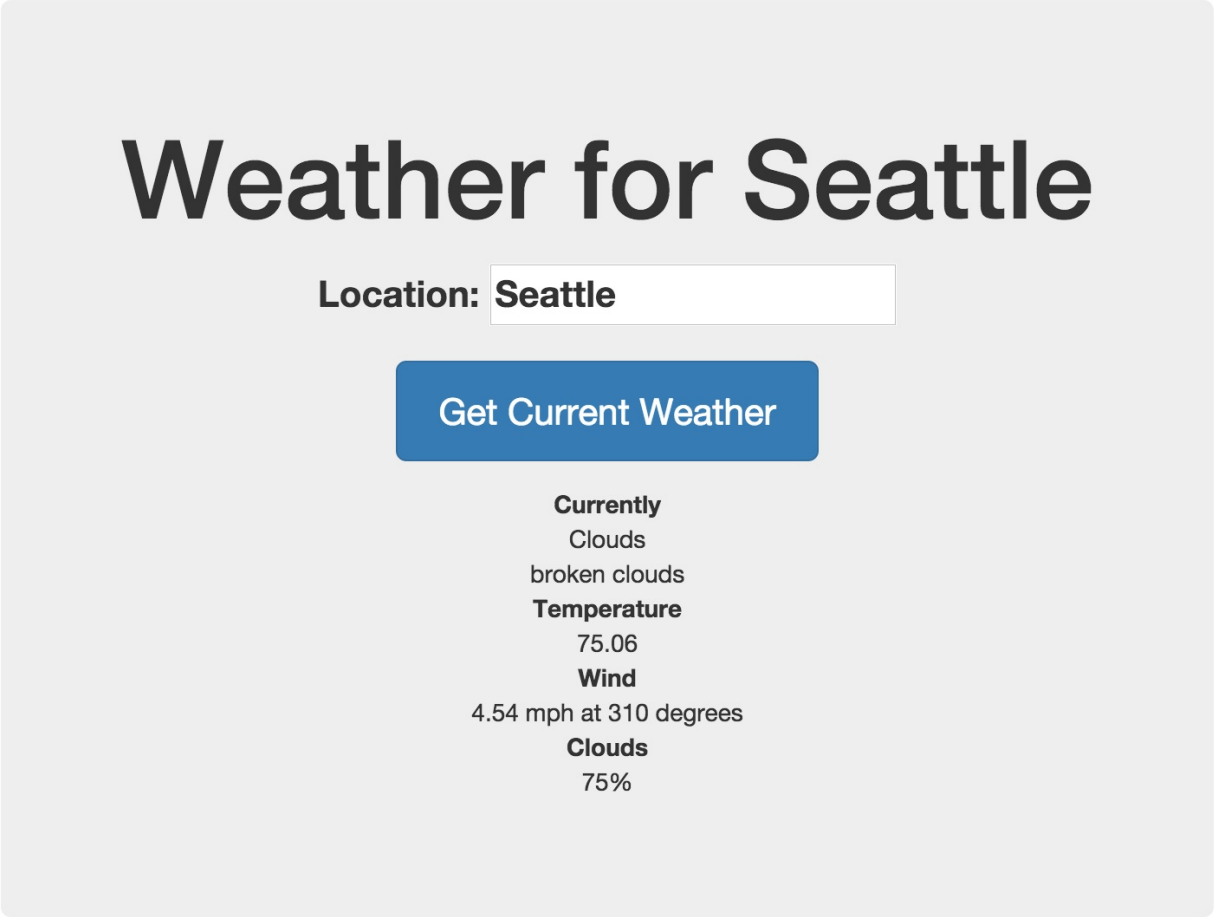
One more thing

The one new thing we will bring into play is an AngularJS module, ngStorage. This module allows you to use a browser feature called `localStorage`, which stores data in the user's web browser. This is a modern approach to storing information locally for users, and as a developer you can use it to enhance your site in many ways.

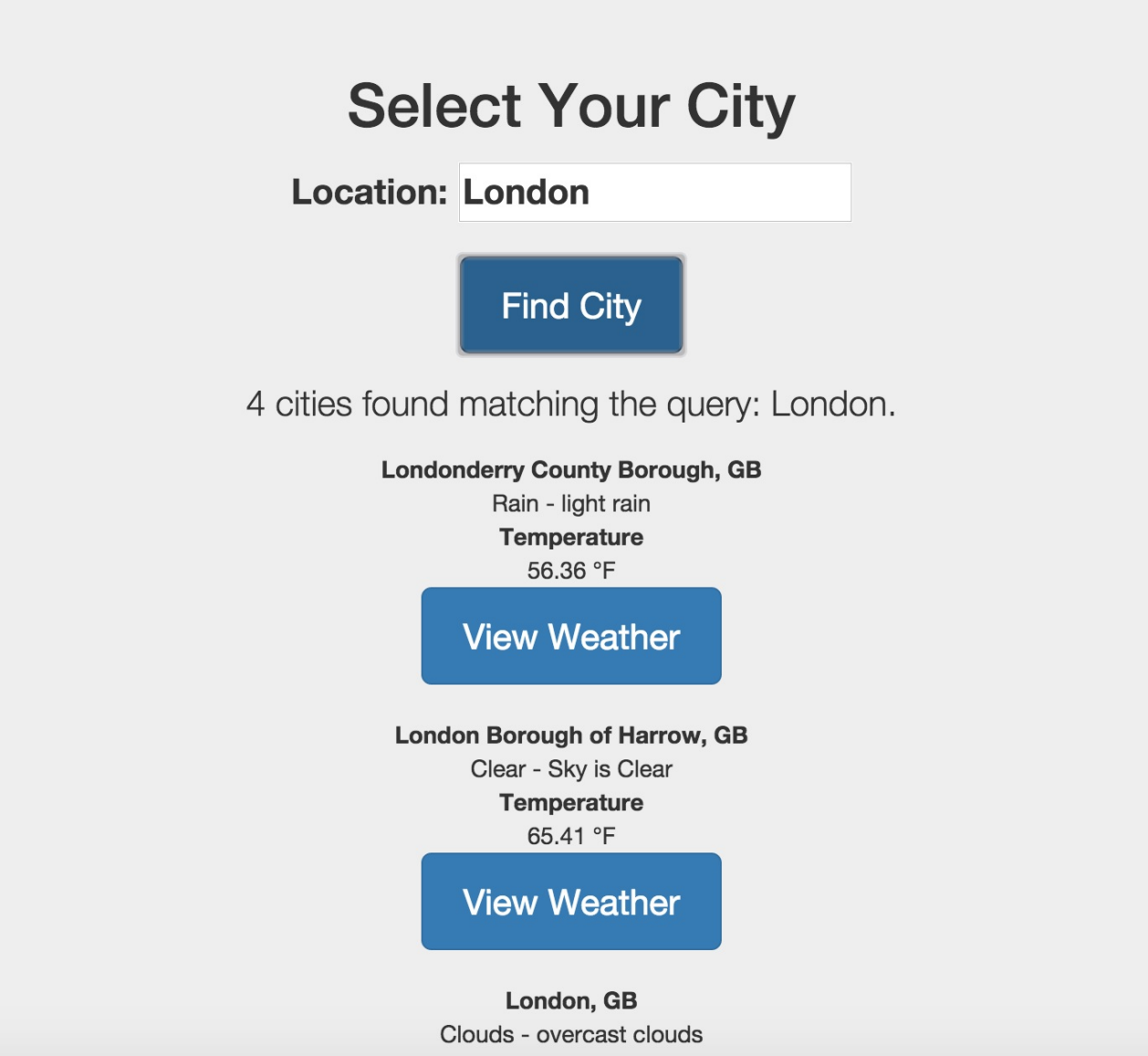
In the context of our weather app, we will use the `$localStorage` object to store the list of favorite cities so we can keep those handy for our users. By using the `ngStorage` module, which gives us the `$localStorage` object, we can easily keep our data synced and safe with fewer lines of code required.

Revising the Home Screen (`main`)

Our current "home" screen is known within our app as `main`. It is the `app/views/main.html` file powered by the `app/scripts/controllers/main.js` file, which contains the `MainCtrl` Controller object. Our home screen currently shows you a text input and the current weather data:



We will revise this output so that on the home screen the user is prompted to search for a city, and then they are shown a list of cities they have saved to their preferred locations list. It will end up looking like this:



In order to do this, we will need to revise the home screen once again. We will alter the `MainCtrl` object so that it handles the city search. We will leverage a different OpenWeatherMap.org API endpoint called the "Find API". The Find API allows you to search for cities so you can, for example, show your user both Paris, France and Paris, Texas to allow them to specify which "Paris" they mean. City names are not unique in the world, so this sort of search functionality is crucial to making a useful weather app.

Home screen update tasks

In order to update the home screen, we will do the following:

1. We will make a new model data resource called "citysearch". This resource will be set up to make calls to the "Find API" on OpenWeatherMap.org.
2. We will revise the `MainCtrl` controller to call the "citysearch.search()" method to retreive data about cities.

3. We will show the user a list of cities we've found, and allow them to click in to a more detailed weather data view (based on the current weather and forecast data we will pull in later this chapter).

For now, we will ignore the "save city" feature. We will put the "save city" button on one of the detail pages about the city's weather.

Find a City: Create the `citysearch` Resource

In order to get a weather report, users must be able to find a city. As we mentioned, it's important to provide users a way to search and specify the exact city they want. We don't want to confuse Paris, France with Paris, Texas. So when a user searches for "Paris", we will show them both as options.

Once the user specifies which "Paris" they want to see weather for, we will use the "ID" for that city to make all of our calls to the OpenWeatherMap.org API. The ID is unique for each city, so if we make a call for data related to a specific city ID, then we know that we will get the correct information.

Set up the `citysearch` model data resource

In order to perform the city search call, we will need to create a `citysearch` resource that we can use. To do this, we will follow the same basic steps as we followed to create the `current` resource we used in the last chapter.

First, run the Yeoman generator to generate the resource:

```
yo angular:factory citysearch
```

That command, as it did previously, will create a new file in `app/scripts/services/` called `citysearch.js`. Inside that file, you will find the stub of a resource. You will want to modify that stub to use the `$resource` object and add in the OpenWeatherMap.org API URL pattern. This will follow the same pattern as we used in the previous chapter. The proper URL to perform a city search using the OWM Find API should look like this:

```
http://api.openweathermap.org/data/2.5/find?q=paris&type=like&mode=json&APPID=YOUR_APP
_ID_HERE
```

Once we are done creating the resource definition, it should look like this:

```
angular.module('yourApp')
  .factory('citysearch', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/find?q=:query&type=like&mode=json&APPID=YOUR_APP_ID', {}, {
      find: {
        method: 'GET',
        params: {
          query: 'seattle'
        },
        isArray: false
      }
    });
  });
```

Looking at the code we've generated, you can see that we have mostly recreated the same setup as in the `current` resource (defined in `app/scripts/services/current.js`). You can identify that we have replaced the string `paris` in our example with the template placeholder `:query` and we supply a default value of `'seattle'` if no other value is specified. That means our app will, by default, return `citysearch` results for "seattle", but we can alter that whenever we make the search call.

Also note that we are defining a `find()` method here. So when we access this data, we will use code that looks like this:

```
$scope.citiesFound = citysearch.find({
    query: $scope.searchText
});
```

Wire it up

Now that we have this resource defined, we will wire it into our `MainCtrl` so we can use it on our home screen.

Wiring Up `citysearch` on the Home Screen

Now that we've got the `citysearch` resource defined, let's try it out. In order to do that, we will need to alter the `MainCtrl` Controller (in `main.js`) and the `main.html` template. We will set it up so that we use the existing text input and button on the home screen, but instead of immediately calling the `current` resource, we will show the list of cities we have found so the user may select one.

Set up `MainCtrl` in `main.js`

In order to search for cities on the home screen, we must use the `citysearch` resource in the `MainCtrl` Controller. We must also rework the way we assign variables and set up our refresh function. Here is what your `MainCtrl` object should look like after you finish your edits:

```
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, citysearch) {
    $scope.citiesFound = citysearch.find();

    $scope.findCities = function(){
        $scope.citiesFound = citysearch.find({
            query: $scope.location
        });
        $scope.searchQuery = $scope.location;
    };
  });
```

You will notice that we are now using `$scope` and `citysearch` in this controller. We have create a `$scope` variable called `citiesFound` that we can use to loop through the results of our search. We initialize that value to the results of a default search. We also define a `$scope` function called `findCities()`, which performs a search and updates a `$scope` variable called `searchQuery`. This `$scope.searchQuery` variable is an example of the kinds of variables you can use to improve the user interface. We will store the value of `$scope.location` that we searched for so that we can provide good feedback to our user about the search results.

There is not too much code here, and for the most part it is not very different from the code we wrote to fetch the current weather. Hopefully these lines are starting to feel familiar and you're gaining some comfort with some of these common patterns.

Set up the `main.html` template

Now that we have the `MainCtrl` Controller all ready to power our view, we need to set up our template. This is also a fairly straightforward conversion, although you will probably want to pull up some sample queries in Postman or another API browser so you can get an idea of the data structure you receive when using the Find API. This is different, but similar to, from the structures returned by the Current Weather API.

Here is what the template should look like after you finish your edits:

```
<div ng-app class="jumbotron" ng-controller="MainCtrl">
  <h1>Select Your City</h1>
  <p class="lead">
    <div ng-init="location='Seattle'">
        <p>
          <label for="location">Location:
            <input type="text" name="location" ng-model="location">
          </label>
        </p>
        <p>
          <button class="btn btn-lg btn-primary" ng-click="findCities()">Find City</bu
tton>
        </p>
    </div>
    <div ng-if="searchQuery">
      <p class="lead">{{citiesFound.count}} cities found matching the query: {{searchQ
uery}}.</p>
      <dl ng-repeat="city in citiesFound.list">
        <dt>{{city.name}}, {{city.sys.country}}</dt>
        <dd>{{city.weather[0].main}} - {{city.weather[0].description}}</dd>
        <dt>Temperature</dt>
        <dd>{{city.main.temp}} &deg;F</dd>
        <dd><a ng-href="/#/current/{{city.id}}" class="btn btn-lg btn-primary">View We
ather</a></dd>
      </dl>
    </div>
  </p>
</div>
```

We use the same text input box and button, but we will attach our `findCities()` function to the button this time. You will notice that we have changed the text label, as well as the text for the title of the page.

We wrapped the results in a `<div>` that is controlled by a `ng-if` directive:

```
<div ng-if="searchQuery">
```

This directive only shows the element and its children if the conditional contained in the `ng-if` attribute is true. In this case, the results of our search will only show if the `searchQuery` variable contains a value. So when you first load the page, you will see no results, but after you submit your query and the response returns from the API the results will be shown.

Since we anticipate there will be multiple results, we use the `ng-repeat` directive to specify a loop in the `ng-repeat` attribute:

```
<dl ng-repeat="city in citiesFound.list">
```

In this case we loop through the `city` objects in `citiesFound.list` Array (which is where the OpenWeatherMap.org API puts the list of cities). Using the `ng-repeat` directive causes the HTML element it's a part of to repeat for every item in an Array.

We output all the different useful information about each city so our user can understand what city has been found. The OpenWeatherMap.org API gives us some brief weather data for each city we found, so we can display some quick, useful information to our users.

Finally, we output a button to click on to view the weather for a given city:

```
<a ng-href="/#/current/{{city.id}}" class="btn btn-lg btn-primary">View Weather</a>
```

This link makes use of the `ng-href` directive, which parses variables inside of the `ng-href` attribute. This allows us to create a link to view current weather info about a given city. In our case, we are building the URL with the `/current/` directory followed by the ID of the city: `{{city.id}}`. When our users click this link, we will be able to show them the current weather data for the city.

Try it out!

Now that we have all the pieces in place, you should be able to test your creation in your browser. You should now be able to search for cities using whole names, or parts of names, and see all the cities you can find. OpenWeatherMap.org tracks weather for about 200 thousand cities, so there should be plenty to try out. Each city should come up with the name, country, and weather summary.

Once you've marveled at your amazingly awesome new city search tool, we can move on to building the current weather view in its proper location.

Revisiting the `current` Data Resource

We built the current data resource in the previous chapter. In order to fit our current needs, we will want to alter the API URL we are using in one small way: We will switch from doing a text match on the name of the city to calling data for a specific city ID. We will use the ID returned from the `citysearch` service when we ultimately make this call.

`current` resource

The `current` resource should be defined in the `app/scripts/services/current.js` file. That file should contain this resource definition:

```
angular.module('yourApp')
  .factory('current', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/weather?id=:cityID&units=
imperial&APPID=YOUR_API_KEY', {}, {
      query: {
        method:'GET',
        params:{
          cityID: '4717560' // Paris, France ID
        },
        isArray:false
      }
    });
  });
```

This resource will still default to find results for the ID `'4717560'`, which is the ID for Paris, France. Note that the URL has been altered. Rather than sending the `q` parameter, we are sending the `id` parameter, and we are populating that parameter with the value of `cityID` in our `query()` method. When we use this in a controller, it will look something like this:

```
$scope.currentWeather = current.query({
    cityID: 1234567
});
```

That code would set a `$scope` variable called `currentWeather` equal to the API response from OpenWeatherMap.org.

Next Step: Create the view

In order to actually use this data in our app, we need to create a "current weather" view, which will involve creating a route that can take parameters. Let's do that now.

Creating the `current` Route

To create a route for our "current weather" view, we will use the Yeoman generator once again. This time, we will run the following command in our project:

```
yo angular:route current
```

This command will create a controller called `CurrentCtrl` in `app/scripts/controllers/current.js` and a template called `current.html` in `app/views/current.html`. It will also add this entry to your `app/scripts/app.js` file:

```
.when('/current', {
    templateUrl: 'views/current.html',
    controller: 'CurrentCtrl',
    controllerAs: 'current'
})
```

We will need to alter this route in one small way to get things working. We will add a "route parameter" called `:cityID` to the path for this route. That route parameter is something we can pick up in our controller, so we can use that information to change the response this view delivers.

Here is how your route definition will look after you add that parameter:

```
.when('/current/:cityID', {
    templateUrl: 'views/current.html',
    controller: 'CurrentCtrl',
    controllerAs: 'current'
})
```

Set up the controller

Now that we have the route defined and all our basic files in place, we can edit them to bring our current weather view to life. First, jump into `app/scripts/controllers/current.js` and begin editing your `CurrentCtrl` Controller.

You will need to add `$scope` and `$routeParams` to your controller. We will use those to access and store info. You will also need to add `current` to your controller, so you can access the current weather data API from OpenWeatherMap.org.

Once you have all those pieces available in your controller, you can easily write the code needed to set the `$scope.cityID` and make your data call:

```
angular.module('yourApp')
  .controller('CurrentCtrl', function ($scope, $routeParams, current) {
    $scope.cityID = $routeParams.cityID;

    $scope.currentWeather = current.query({
        cityID: $routeParams.cityID
    });
  });
```

Notice that we are storing the `cityID` in a specific `$scope` variable, more to have it easily handy than anything else. The `$routeParams` object makes any route parameters available for your use. Since we called our parameter `cityID` in the route definition, we can access it with that same name as `$routeParams.cityID`.

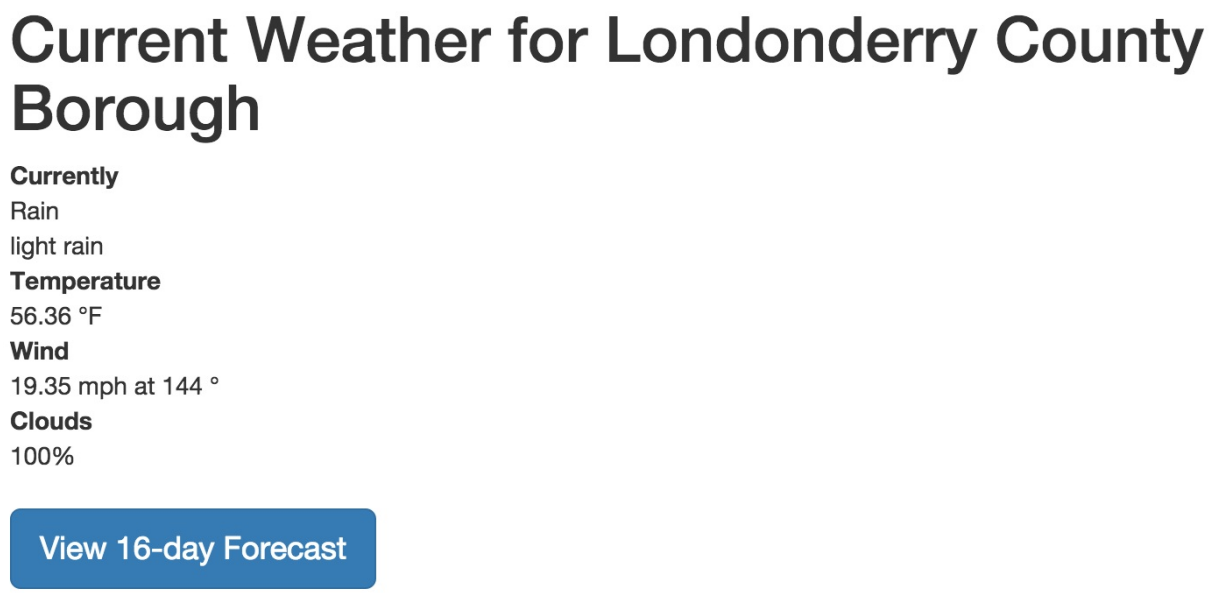
Using our updated data resource, we can easily get our current weather data. Now that we have that data streaming into the `$scope.currentWeather` variable, we can build our view template.

Set up the view template

The view template created for us by Yeoman is the file `app/views/current.html`. Inside you will find some placeholder content. You can replace that with content that presents the current weather data to our users:

```
<h1>Current Weather for {{currentWeather.name}}</h1>

<dl>
    <dt>Currently</dt>
    <dd>{{currentWeather.weather[0].main}}</dd>
    <dd>{{currentWeather.weather[0].description}}</dd>
    <dt>Temperature</dt>
    <dd>{{currentWeather.main.temp}} &deg;F</dd>
    <dt>Wind</dt>
    <dd>{{currentWeather.wind.speed}} mph at {{currentWeather.wind.deg}} &deg;</dd>
    <dt>Clouds</dt>
    <dd>{{currentWeather.clouds.all}}%</dd>
</dl>

<p><a ng-href="/#/forecast/{{cityID}}" class="btn btn-lg btn-primary">View 16-day Fore
cast</a></p>
```

Since we already figured out where the data is located inside the object returned by the OWM Current Weather API it was easy to place all of our information on the page. We can now click from the home screen to the current weather screen, and you should see something that looks like this in your browser:



Let's do that again!

Now you can search for cities and then click in to view more weather details. Notice that at the bottom of the template code above there is a link to a whole new route: `/*/forecast/{{cityID}}`. We will retrace our steps to add this additional view in order to get more practice. Our approach will be identical to the approach we've taken to create this current weather view.

Create the `forecast` data resource

In order to use the 16-day Forecast API from OpenWeatherMap.org, we will once again create a new factory service using the Yeoman generator:

```
yo angular:factory forecast
```

Once we have that, inside of `app/scripts/services/forecast.js` we will create the following resource definition:

```
angular.module('yourApp')
  .factory('forecast', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/forecast/daily?id=:cityID
&cnt=16&units=imperial&APPID=YOUR_API_KEY', {}, {
      query: {
        method:'GET',
        params:{
          cityID: '4717560' // Paris, France ID
        },
        isArray:false
      }
    });
  });
```

As always, you will need to substitute in your own API key and the name of your AngularJS app, but otherwise you should have something similar in your own code. Once you have the basics of working with a particular API settled, it should become trivial to create these resources.

Creating the `forecast` Route and View

Creating the forecast view is almost identical to creating the current weather view. The two start with the same basic Yeoman command:

```
yo angular:route forecast
```

Once you've run that command you should be able to see the controller file (`app/scripts/controllers/forecast.js`), view template (`app/views/forecast.html`) and the changes inside of `app/scripts/app.js` :

```
.when('/forecast', {
    templateUrl: 'views/forecast.html',
    controller: 'ForecastCtrl',
    controllerAs: 'forecast'
})
```

Once again, we will modify that route definition so it will accept a route parameter called `cityID` :

```
.when('/forecast/:cityID', {
    templateUrl: 'views/forecast.html',
    controller: 'ForecastCtrl',
    controllerAs: 'forecast'
})
```

Now that we have `cityID` coming through as a route parameter, we can set up the `ForecastCtrl` Controller.

Edit `ForecastCtrl` Controller

As we did with the `CurrentCtrl` Controller, we will set up our `$scope` variables so we can display our desired data in the `forecast.html` template. The controller will end up looking very much like the `CurrentCtrl` Controller:

```
angular.module('yourApp')
  .controller('ForecastCtrl', function ($scope, $routeParams, forecast) {
    $scope.cityID = $routeParams.cityID;

    $scope.forecastData = forecast.query({
        cityID: $routeParams.cityID
    });
  });
```

This is almost identical to the controller for the current weather view. The two views ultimately perform the same task: They both accept a `cityID` and then make a query to the OpenWeatherMap.org API using that ID so they can show the data in the template.

Edit the `forecast.html` template

Now that we've got the data ready for our template, we can modify the HTML in our `app/views/forecast.html` file so it will display our forecast properly. The most dramatic change between this display and the display of our current weather data is that we have weather information for multiple days. This means we will need to loop through the results and display all of the weather information (similar to how we looped through the `citysearch` results previously).

Here is what the template should look like after you are done making edits:

```
<h1>16-day Forecast for {{forecastData.city.name}} {{forecastData.city.country}}</h1>
<dl ng-repeat="prediction in forecastData.list" class="weather-forecast">
    <dt>Forecast for {{weather.dt*1000 | date:'MMM dd, yyyy'}}</dt>
    <dd>{{prediction.weather[0].main}}</dd>
    <dd>{{prediction.weather[0].description}}</dd>
    <dt>Temperature</dt>
    <dd>Min: {{prediction.temp.min}} &deg;F Max: {{prediction.temp.max}} &deg;F</dd>
</dl>

<p><a ng-href="/#/current/{{cityID}}" class="btn btn-lg btn-primary">View Current Weat
her</a></p>
```

Notice that the 16-day Forecast API returns a slightly different data structure than the other API endpoints, so it is worthwhile to observe some of these responses and get an idea for how the data is arranged.

Of special note in the forecast data results are the list of forecasts (for up to 16 days of weather predictions) and the use of `min` and `max` temperatures in the forecasts.

In the example above, you can see that we are repeating a `<dl>` element for each day of weather predictions. Within each of those repeating `<dl>` elements, we are outputting some data. Most of this data is pretty standard, although the `weather.dt` value gets some special treatment using the AngularJS `date` filter.

The `date` filter allows us to convert a timestamp into a human-readable date. The OpenWeatherMap.org API formats the date as a "Unix Timestamp" that is based in seconds. This number represents the number of seconds that have taken place since January 1, 1970 (also known as the "Unix Epoch"). This is a common way of recording time.

The `date` filter is designed to take a timestamp and convert it to a human-readable format, but it is also designed to expect the timestamp to be in *milliseconds*, which is how timestamps are counted in Javascript. (And, yes, this is the number of milliseconds since January 1, 1970.)

In order to make the `date` filter work, we must multiply our timestamp by 1000, hence the formatting you see above:

```
{{weather.dt*1000 | date: 'MMM dd, yyy'}}
```

That variable is filtered through the `date` filter and becomes:

```
Aug 03, 2015
```

(or something similar).

Finally, notice that we kept a link at the bottom of this forecast view to return to the current weather view for our chosen city. (We can also click the "home" link in the top navigation to return home.)

Try it out

Now that you have set up your template, you can test out your view and you should see results streaming in from the OpenWeatherMap.org API. Add a little CSS to your `app/styles/main.scss` file and you can easily come up with something like this:

16-day Forecast for Londonderry County Borough GB

Forecast for Aug 03, 2015
Rain
light rain
Temperature
Min: 56.64 °F Max: 62.74 °F

Forecast for Aug 04, 2015
Rain
light rain
Temperature
Min: 55.51 °F Max: 60.21 °F

Forecast for Aug 05, 2015
Rain
moderate rain
Temperature
Min: 54.41 °F Max: 61.45 °F

Forecast for Aug 06, 2015
Rain
light rain
Temperature
Min: 53.1 °F Max: 57.96 °F

Forecast for Aug 07, 2015
Rain
light rain
Temperature
Min: 50.23 °F Max: 56.44 °F

Forecast for Aug 08, 2015
Rain
light rain
Temperature
Min: 50.59 °F Max: 61.63 °F

Huzzah! You almost have your whole basic feature set complete now. A user can find a city, view the current weather, and view a 16-day forecast. Those are the same basic features you'll find on most weather apps and websites. Congratulations! Now we can add the final touch: Allowing users to "save" their preferred cities.

Store and Retrieve Locations

Most weather apps and websites allow you to store a list of your preferred locations so you can easily refer to them when you return to the app. We will allow our users to store their chosen locations and then access those cities in a list on the home screen of our location. They can then click into one of the cities from their list, or they can search for other cities.

Add the ngStorage module

In order to use the `localStorage` feature of modern web browsers, we will install the ngStorage module for use in our project. This will make the `$localStorage` object available to our controllers so we can use locally stored data easily throughout our application.

We install ngStorage with Bower:

```
bower install --save ngstorage
```

You may notice that running this command adds a line to your `app/index.html` file with a `<script>` tag pointing to the ngStorage file. In addition to that, we must manually add the `ngStorage` module to our list of required modules in our app definition (which is in `app/scripts/app.js`). The first part of your app definition should look like this after you add the `ngStorage` module:

```
angular
  .module('test3App', [
    'ngAnimate',
    'ngAria',
    'ngCookies',
    'ngMessages',
    'ngResource',
    'ngRoute',
    'ngSanitize',
    'ngStorage', // added to enable localStorage features
    'ngTouch'
  ])
  .config(function ($routeProvider) {
  //  ... additional code...
```

This addition makes the ngStorage Javscript module and it's components available in our application, but we must still let our controllers and other object know about the object before they use it.

Modify the `currentCtrl` Controller

The `currentCtrl` Controller is the controller in charge of rendering our current weather view in the `app/views/current.html` template. This controller will be used to "save" our city. In order to add a "save" button to the template, we need to make a function that will add the city to our list of saved cities. And in order to make that happen, we need the `$localStorage` object to be available to the `currentCtrl` Controller.

First, add the `$localStorage` dependency to your controller declaration:

```
.controller('CurrentCtrl', function ($scope, $routeParams, current, $localStorage) {
```

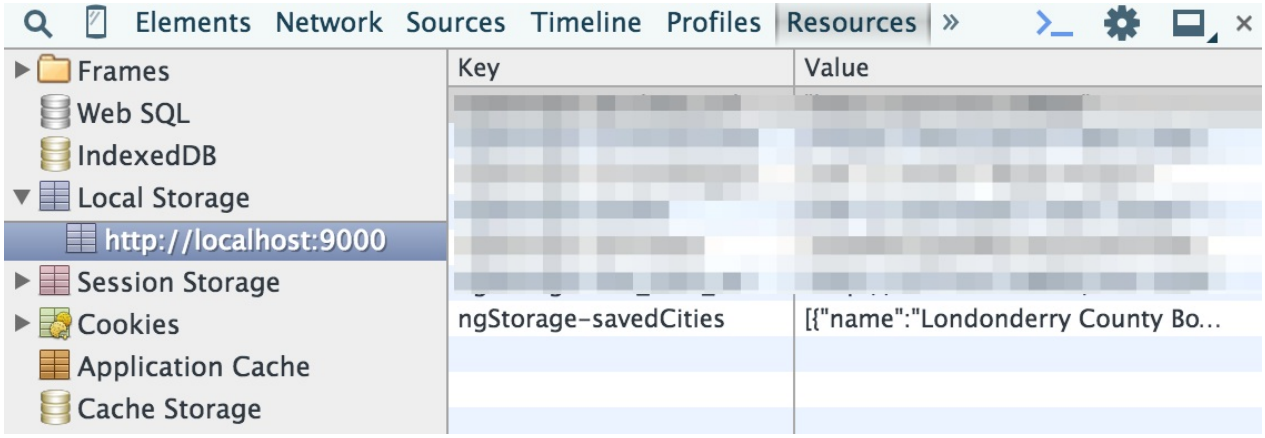
Once you have that in place, you can use it in a function that can be applied to a save button via the `ng-click` directive. Here is what that function will look like:

```
$scope.saveCity = function(city){
    var cityData = {
        'name': city.name,
        'id': city.id
    };
    if (!$localStorage.savedCities){
        $localStorage.savedCities = [cityData];
    } else {
        // We have already saved some cities.
        // Check to make sure we haven't already saved the current city.
        var save = true; // Initialize the save decision variable.
        // Use this loop to check if we've already saved the city.
        for (var i=0; i < $localStorage.savedCities.length; i++){
            if ($localStorage.savedCities[i].id == cityData.id) {
                // This is a duplicate, so don't save (variable set to false).
                save = false;
            }
        }
        if (save==true){
            $localStorage.savedCities.push(cityData);
        } else {
            console.log('city already saved');
        }
    }
};
```

You can add a button to the `app/views/current.html` template like this:

```
<p><button class="btn btn-sm btn-primary" ng-click="saveCity(currentWeather)">Save City
</button></p>
```

Once you put all those pieces together, you should be able to save the city. Now, we haven't added any interface indication that the city is saved, so if you're looking to verify that our city was saved, you should inspect the page in developer tools and click to the Resources tab. You can then twirl down the "Local Storage" list and view local storage for the domain your site is running on. You should clearly see the entry made when you clicked the button:



Once you've made those changes, we can output the list of saved cities on the home screen so a user can easily click into the weather data for any of their saved cities.

Show List of Saved Cities

Now that we have a way for users to save their preferred locations, it is critical to give them a way to view and select those saved cities. We will display the saved cities list below our search box on the home screen. The home screen view is controlled by the `MainCtrl` Controller, and rendered in the `app/views/main.html` template. We will modify these two files to display saved cities to our users.

Modify the `MainCtrl` Controller

In the `MainCtrl` Controller we will have to make a couple of changes. As with the `CurrentCtrl` Controller, we will need to make `$localStorage` available to the controller:

```
.controller('MainCtrl', function ($scope, citysearch, $localStorage) {
```

Once we've added `$localStorage` to the controller declaration, we can add a quick mapping to a `$scope` variable so we can use `$localStorage` data inside of our template. Add this line to your controller:

```
$scope.storage = $localStorage;
```

That allows us to reference `storage.savedCities` in the template to access our list of saved cities. here is the entire `MainCtrl` Controller:

```
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, citysearch, $localStorage) {
    $scope.citiesFound = citysearch.find();
    $scope.storage = $localStorage;

    $scope.findCities = function(){
        $scope.citiesFound = citysearch.find({
            query: $scope.location
        });
        $scope.searchQuery = $scope.location;
    };
  });
```

Add saved cities to the template

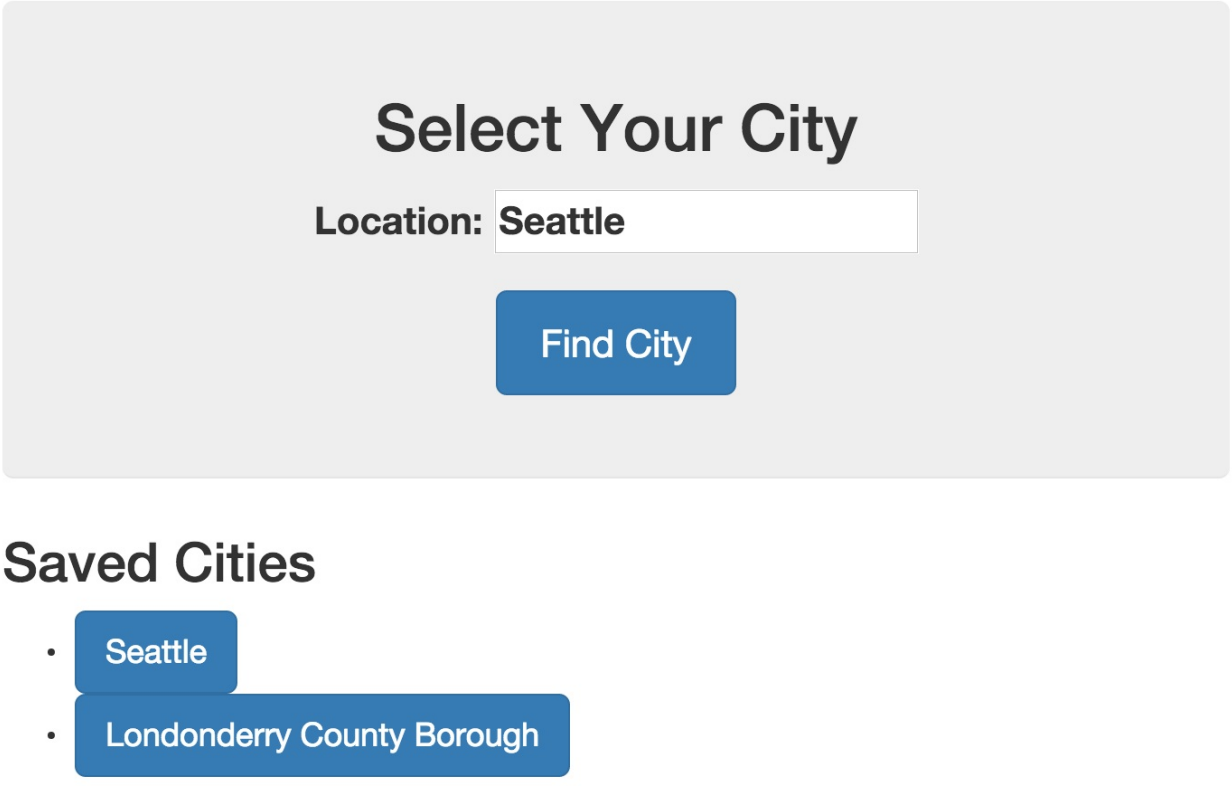
Now that we have made the `$localStorage` data available to our template, we can modify `app/views/main.html` to show a list of saved cities with links to view their current weather. To facilitate that change, we can add a section below our search results display:

```
<div ng-if="storage.savedCities">
  <h2>Saved Cities</h2>
  <ul class="saved-cities-list">
    <li ng-repeat="city in storage.savedCities">
      <a ng-href="/#/current/{{city.id}}" class="btn btn-lg btn-primary">{{city.name}}
</a>
    </li>
  </ul>
</div>
```

Looking at the code above, we can see that this div will only display if there is data stored in the `storage.savedCities` variable (look at the `ng-if` directive to see where that is specified). If we do have some saved cities, then we will show a heading and a list of links.

Notice that the `<li>` elements repeat for every `city in storage.savedCities` . On the `<a>` elements we use the `ng-href` directive to specify the URL for viewing that city's current weather data, and we use the `city.name` variable to provide the link text.

Once we have made those changes, view your app and click into the current weather for any city. On the current weather screen, click "Save City" and then return to the home screen. You should now see your saved city listed below the search box like this:



If you see something resembling the image above, then congratulations. You have now completed all of the basic features to make your weather app run.

Conclusion

Now that we have completed the basic features of our app, we can focus on the many, many improvements that will make this app stand out above the competition. Using the patterns that we've established so far we could add a few lines of code here and there to make major improvements in usability and user experience.

If you want to take a moment to practice more of what we've done here, I encourage you to think about how you would approach these tasks using, more or less, the same patterns we've established here:

- Make the save city button turn into a "remove saved city" button if the user has already saved the city. (And actually remove that `cityData` object from the list if the user wishes.)
- Allow users to save/unsave cities from the forecast view.
- Add an additional API call to look up the weather data for saved cities so you can show quick weather reports for the user's favorite cities.
- Use data coming back about the weather to add styles to the page so that the current and forecast views change their appearance based on the actual weather.
- Find some public domain or open source weather imagery and icons and incorporate those.

The possibilities are rich when working with data as textured and dynamic as weather.

Congratulations on making it through the most ambitious chapter so far! Check the weather where you live and plan a picnic or a hike or something!

File Changes in this Chapter

Here is a list of files that are altered in this chapter along with code examples showing what they should look like when you are finished. Reference these examples as you work to help you debug and evolve your code.

app/scripts/app.js

```
angular
  .module('yourApp', [
    'ngAnimate',
    'ngAria',
    'ngCookies',
    'ngMessages',
    'ngResource',
    'ngRoute',
    'ngSanitize',
    'ngStorage',
    'ngTouch'
  ])
  .config(function ($routeProvider) {
    $routeProvider
      .when('/', {
        templateUrl: 'views/main.html',
        controller: 'MainCtrl',
        controllerAs: 'main'
      })
      .when('/about', {
        templateUrl: 'views/about.html',
        controller: 'AboutCtrl',
        controllerAs: 'about'
      })
      .when('/current/:cityID', {
        templateUrl: 'views/current.html',
        controller: 'CurrentCtrl',
        controllerAs: 'current'
      })
      .when('/forecast/:cityID', {
        templateUrl: 'views/forecast.html',
        controller: 'ForecastCtrl',
        controllerAs: 'forecast'
      })
      .otherwise({
        redirectTo: '/'
      });
  });
```

app/scripts/services/citysearch.js

```
angular.module('yourApp')
  .factory('citysearch', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/find?q=:query&units=imper
ial&type=like&mode=json&APPID=YOUR_API_KEY', {}, {
      find: {
        method: 'GET',
        params: {
          query: 'seattle'
        },
        isArray: false
      }
    });
  });
```

app/scripts/services/current.js

```
angular.module('yourApp')
  .factory('current', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/weather?id=:cityID&units=
imperial&APPID=YOUR_API_KEY', {}, {
      query: {
        method:'GET',
        params:{
          cityID: '4717560' // Paris, France ID
        },
        isArray:false
      }
    });
  });
```

app/scripts/services/forecast.js

```
angular.module('yourApp')
  .factory('forecast', function ($resource) {
    // Service logic
    // ...

    // Public API here
    return $resource('http://api.openweathermap.org/data/2.5/forecast/daily?id=:cityID
&cnt=16&units=imperial&APPID=YOUR_API_KEY', {}, {
      query: {
        method:'GET',
        params:{
          cityID: '4717560' // Paris, France ID
        },
        isArray:false
      }
    });
  });
```

app/scripts/controllers/current.js

```
angular.module('yourApp')
  .controller('CurrentCtrl', function ($scope, $routeParams, current, $localStorage) {
    $scope.cityID = $routeParams.cityID;

    $scope.currentWeather = current.query({
        cityID: $routeParams.cityID
    });

    $scope.saveCity = function(city){
        var cityData = {
            'name': city.name,
            'id': city.id
        };
        if (!$localStorage.savedCities){
            $localStorage.savedCities = [cityData];
        } else {
            // Check to make sure we haven't already saved the city.
            var save = true;
            for (var i=0; i < $localStorage.savedCities.length; i++){
                if ($localStorage.savedCities[i].id == cityData.id) {
                    // this is a duplicate, so don't save
                    save = false;
                }
            }
            if (save==true){
                $localStorage.savedCities.push(cityData);
            } else {
                console.log('city already saved');
            }
        }
    };
  });
```

app/scripts/controllers/forecast.js

```
angular.module('yourApp')
  .controller('ForecastCtrl', function ($scope, $routeParams, forecast) {
    $scope.cityID = $routeParams.cityID;

    $scope.forecastData = forecast.query({
        cityID: $routeParams.cityID
    });
  });
```

app/scripts/controllers/main.js

```
angular.module('yourApp')
  .controller('MainCtrl', function ($scope, citysearch, $localStorage) {
    $scope.citiesFound = citysearch.find();
    $scope.storage = $localStorage;

    $scope.findCities = function(){
        $scope.citiesFound = citysearch.find({
            query: $scope.location
        });
        $scope.searchQuery = $scope.location;
    };
  });
```

app/views/current.html

```
<h1>Current Weather for {{currentWeather.name}}</h1>
<p><button class="btn btn-sm btn-primary" ng-click="saveCity(currentWeather)">Save City
</button></p>
<dl>
    <dt>Currently</dt>
    <dd>{{currentWeather.weather[0].main}}</dd>
    <dd>{{currentWeather.weather[0].description}}</dd>
    <dt>Temperature</dt>
    <dd>{{currentWeather.main.temp}} &deg;F</dd>
    <dt>Wind</dt>
    <dd>{{currentWeather.wind.speed}} mph at {{currentWeather.wind.deg}} &deg;</dd>
    <dt>Clouds</dt>
    <dd>{{currentWeather.clouds.all}}%</dd>
</dl>

<p><a ng-href="/#/forecast/{{cityID}}" class="btn btn-lg btn-primary">View 16-day Fore
cast</a></p>
```

app/views/forecast.html

```
<h1>16-day Forecast for {{forecastData.city.name}} {{forecastData.city.country}}</h1>
<dl ng-repeat="weather in forecastData.list" class="weather-report">
    <dt>Forecast for {{weather.dt*1000 | date:'MMM dd, yyyy'}}</dt>
    <dd>{{weather.weather[0].main}}</dd>
    <dd>{{weather.weather[0].description}}</dd>
    <dt>Temperature</dt>
    <dd>Min: {{weather.temp.min}} &deg;F Max: {{weather.temp.max}} &deg;F</dd>
</dl>

<p><a ng-href="/#/current/{{cityID}}" class="btn btn-lg btn-primary">View Current Weat
her</a></p>
```

app/views/main.html

```
<div ng-app ng-controller="MainCtrl">
<div class="jumbotron">
  <h1>Select Your City</h1>
  <p class="lead">
    <div ng-init="location='Seattle'">
        <p>
          <label for="location">Location:
            <input type="text" name="location" ng-model="location">
          </label>
        </p>
        <p>
          <button class="btn btn-lg btn-primary" ng-click="findCities()">Find City</bu
tton>
        </p>
    </div>
    <div ng-if="searchQuery">
      <p class="lead">{{citiesFound.count}} cities found matching the query: {{searchQ
uery}}.</p>
      <dl ng-repeat="city in citiesFound.list">
        <dt>{{city.name}}, {{city.sys.country}}</dt>
        <dd>{{city.weather[0].main}} - {{city.weather[0].description}}</dd>
        <dt>Temperature</dt>
        <dd>{{city.main.temp}} &deg;F</dd>
        <dd><a ng-href="/#/current/{{city.id}}" class="btn btn-lg btn-primary">View We
ather</a></dd>
      </dl>
    </div>
  </p>
</div>
<div ng-if="storage.savedCities">
  <h2>Saved Cities</h2>
  <ul class="saved-cities-list">
    <li ng-repeat="city in storage.savedCities">
      <a ng-href="/#/current/{{city.id}}" class="btn btn-lg btn-primary">{{city.name}}
</a>
    </li>
  </ul>
</div>
</div>
```

Applying Animation and Interface Enhancements

In order to provide a better experience for users, it is useful to leverage some of the "sugar" that we can apply to sweeten up our application. Namely, we can use animation and more responsive messaging to allow our users to more easily track how state or data has changed in our application.

Animating for meaning

One of the side-effects of writing highly performant Javascript applications is that when we change the data on the screen it can be difficult for users to notice the alterations. It's not at all uncommon when working with an unstyled app for even developers, bleary eyed from looking closely at code, may miss that a value has changed in a corner of the screen.

In order to make it more evident that changes are happening on the page, we often turn to animation. Humans are very good at noticing even small movements, especially if everything else on a page is generally stationary. We can animate position, size, color, and shape to help us draw attention to whatever we have changed.

We will explore how to use the features of the ngAnimate module to allow us to capture animations as they are happening on our views. This primarily leverages common techniques for applying CSS animation, but it's useful to understand the many hooks AngularJS provides to trigger the correct animation at the correct time. Making good use of the ngAnimate module will allow your app to easily draw attention in whatever way makes the most sense for the user and purpose.

Messaging for clarity

Another weakness we have in our current interface has to do with clarity of what has happened when the user clicks the "Save City" button. This is a common problem we encounter when using websites. If we click a button to do something (like save a favorite), and then we receive no confirmation that action was successful, we are left wondering whether or not "it worked".

Sometimes sites rely purely on animation to indicate something worked, such as making a favorite star jiggle when you click it, but although animation is good for many things in an interface, it may be insufficient to communicate to the user "that action you just attempted

was successful" (or not).

For these sorts of transactions (and many others), we often use a concept of "messaging" in our apps. That is, we use a standard message format, design, and placement to convey updates from the system: "Successfully saved the city" or "Error retrieving data". Adding messaging like this will dramatically improve the usability of our application.

TODO

In this chapter we will add animation to our views, so that when we move from the home screen to the current weather screen, and then from there to the forecast screen, and back, we will see elegant enter and leave animations. We will also add a message to our city save so that we indicate to the user that the city was successfully saved. In the case where the user has tried to save a city that was already saved, we will indicate that, too, so they know they have already saved the city.

Animation in AngularJS

In recent versions of AngularJS, the way animating between views is handled has changed significantly. Since we are working with AngularJS above version 1.2, we will focus on the current approach to animating transitions.

In general, when working with Javascript to create dynamic interfaces, it's preferable to use CSS to make animations where possible. The features of CSS that support animation (transitions, keyframes, and animation sequence definitions) are robust and performant. Using Javascript to add/remove classes on HTML elements is a good way to trigger and control animations. By combining Javascript and CSS, you can have a great deal of control over animation.

Animation basics

Before attempting to work with animations in your AngularJS project, it's good to review the basics of animating with CSS. Animated transitions in AngularJS are triggered by classes that AngularJS applies for you when views change and items are hidden or revealed. Any animation you can define in CSS can be applied to your AngularJS components, and it's very easy to add application-wide animations that happen every time a view changes or content is added to an area of the display.

Animations in AngularJS work best if they are defined as named keyframe animations. These animations can be used inside any style definition, allowing you to define many animations and then use them as you see fit throughout your application. For more background on defining animations in CSS, check out this Mozilla Developers Network page about Using CSS Animations.

Animate.css

There are many animation toolkits out there for CSS. If you search around, you will have no trouble finding them. One set of CSS animation definitions that I like is the `animate.css` package, which is available on Github here. The `animate.css` stylesheet defines named keyframe animations that you can then use throughout your project.

Click through to the `animate.css` Github project page, and you can read a list of the animation names. If you reference those names in your application's stylesheets, you can use those animations. This is an example of what that looks like:

```
div[ng-view].ng-enter {
    -webkit-animation: fadeIn 0.5s;
    animation: fadeIn 0.5s;
}
```

We will use the `animate.css` package to add animations to our application.

AngularJS transitions

AngularJS conveniently adds and removes classes on HTML elements as it transitions users through the application. That means we can define styles for those classes that indicate transition states, and those styles may contain animations that will be triggered during a specific part of the transition. In addition, AngularJS conveniently listens to our animations and is smart about removing transition classes after the animation has completed.

It can be tricky at first to work with these transitions because they do not generally show up visibly even in developer tools, but we can see the effects of the transitions if we define proper styles. By learning which classes are applied during which transitions, we can effectively provide animations and styles that work throughout our application and lend a great degree of cohesion to the final product.

The example above indicates that the `fadeIn` animation will be triggered when the `.ng-enter` class is applied to any `<div>` element with an `ng-view` attribute. In practice, that means that any time the AngularJS view changes, this animation will be triggered.

Example of hypothetical app HTML (probably found `app/index.html`):

```
<div ng-view>
    <!-- This is where the view partial would be inserted -->
</div>
```

And corresponding hypothetical CSS example:

```
div[ng-view].ng-enter {
    -webkit-animation: fadeIn 0.5s;
    animation: fadeIn 0.5s;
}
```

AngularJS transition classes

Below is a chart of the classes that are applied during different AngularJS transitions. This chart is based on the information at Year of Moo. The AngularJS guide to Animations is also very helpful for learning about how to use animations.

Action	Starting CSS Class	Ending CSS Class	Directive that fires it
enter	.ng-enter	.ng-enter-active	ngRepeat, ngInclude, ngIf, ngView
leave	.ng-leave	.ng-leave-active	ngRepeat, ngInclude, ngIf, ngView
move	.ng-move	.ng-move-active	ngRepeat
hide an element	.ng-hide-add	.ng-hide-add-active	ngShow, ngHide
show an element	.ng-hide-remove	.ng-hide-remove-active	ngShow, ngHide
adding a class to an element	.CLASS-add	.CLASS-add-active	ngClass and class=""

As you can see, these transition classes are applied every time an item is added or removed from the screen, and you can use these classes to hook onto many different events throughout the user's interaction with your application.

It is useful to note that if you are using named keyframe animations, you only need to apply them on the "Starting CSS Class" -- the "Ending CSS Class" does not need to be defined.

Getting into it

Of course, the best way to learn about this is to get in and attempt to animate your views. On the next page we will add `animate.css` to our project and use it to apply a general animation to our view changes.

Animating View Transitions

In order to provide a more visually compelling experience for our users, and to reinforce when data in our app changes, we will create some view transition animations. We will take a fairly simple approach here, but you are encouraged to continue experimenting and using this same basic process to add more animations to different changes throughout your app. Keep in mind that you can animate any transition that happens in your application.

Add `animate.css` to the project

In order to create our animations, we will use an open source animation library called `animate.css`. You may reference the `animate.css` Github repository to see what animations are included.

To add `animate.css` to the project, we will use Bower. As usual, you will need to open your terminal and change directory into your project directory. Once you are in your project root, run the following command:

```
bower install --save animate.css
```

Bower should successfully install `animate.css` and add it to our `app/index.html` file:

```
<!-- bower:css -->
<link rel="stylesheet" href="bower_components/animate.css/animate.css" />
<!-- endbower -->
```

Now that you've installed `animate.css` as a CSS dependency in your app, you can reference the animations defined in the CSS library. (Again, look at the Github repository page for a list of animations.)

Define animations

In order to keep my stylesheets nice and tidy, I like to define transition animations in their own stylesheet. In order to do that, I will add an import statement to my `app/styles/main.scss` so my transition styles will be properly included.

Here is the line I added to `app/styles/main.scss`:

```
@import "transitions";
```

And then I will create a new file located at `app/styles/_transitions.scss`. Inside this file, I will put a style definition to create a transitional animation for every time the view changes.

The contents of `app/styles/_transitions.scss` look like this:

```
div[ng-view].ng-enter {
    -webkit-animation: fadeIn 0.5s;
    animation: fadeIn 0.5s;
}
div[ng-view].ng-leave {
    -webkit-animation: fadeOut 0.5s;
    animation: fadeOut 0.5s;
}
`
```

This is almost the same example from the previous page, and as we mentioned before, this style will make it so our content fades in when the view changes. The additional `.ng-leave` style definition will cause our current view template to fade out before the other template fades in. (Note: This effect is not actually the best for users, but it demonstrates the way different styles are applied at different times.)

Expanding and extending

It's worthwhile to go through your entire app and drop some transitional animations on any data changes that warrant the user's attention. This is likely to be all of them. Finding a balance of noticeable yet subtle enough to not be annoying is actually quite tricky, and it is worthwhile to do some trial and error experimentation.

We could essentially define the same styles as above for the elements we use to contain our list of cities, because as the application loops through the cities returned by the OpenWeatherMap.org API, it applies the `.ng-enter` and `.ng-leave` classes accordingly.

And now that we have a full animation library available to us, we could use that to create other animated styles, such as animated weather styles to indicate current conditions. The possibilities are limitless, and it is fun to try new things.

Of course, animations are only part of the enhancements we're after this week, so when you're finished experimenting with animations, let's move on to messaging.

Messaging Concepts

In web applications, being able to let the user know about changes to the system they are using is a critical aspect of designing a responsive, friendly system. In order to communicate changes that may not be possible to represent more clearly to the user through any other means, we often use "messages". Sometimes these messages take on the guise of "alerts" or "notifications", and they are used in many ways.

AngularJS allows us to use a built in messaging tool to easily define messages that can be used in a view and then to trigger them without having to write a bunch of duplicate code. This system is useful and pretty straightforward to make work. It is ideal for us because in our weather app we have a need to communicate when people successfully save a city. There is currently no indication that the app has actually saved the city when you click the "Save City" button. In order to discover that the city has actually been saved, users must click back to the home screen, which is inconvenient.

We will implement messages to indicate that we have successfully saved the city, and to indicate when the user tries to save a duplicate city. (Of course, we could also remove the "Save City" button altogether in order to prevent duplicate saves, and that would be the better solution. But for the purposes of this exercise, we're going to leave the button there.)

The basics of messaging in websites and apps

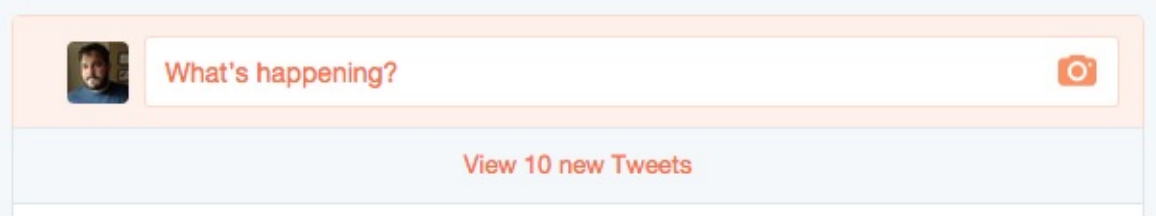
There are many ways to use messaging in websites and apps, and developers are always coming up with clever new ways to indicate changes in content and system status. But a few conventions have grown up around messaging, and for our purposes here we are not going to "reinvent" messages.

When generally thinking about messaging, there are two major types of messages: global and local. These are both useful types of messages and most apps and sites make use of both of them. Understanding when to use which type of message is important to getting messaging right in your application.

Global messages

Global messages apply to everything the user is seeing. On Twitter, for example, as you read tweets in your timeline you may notice an alert that shows up at the top of your timeline telling you that there are a number of new tweets available to read. In many email clients,

when you receive a new message, or when you file a message away, you will see an alert at the top of the screen.

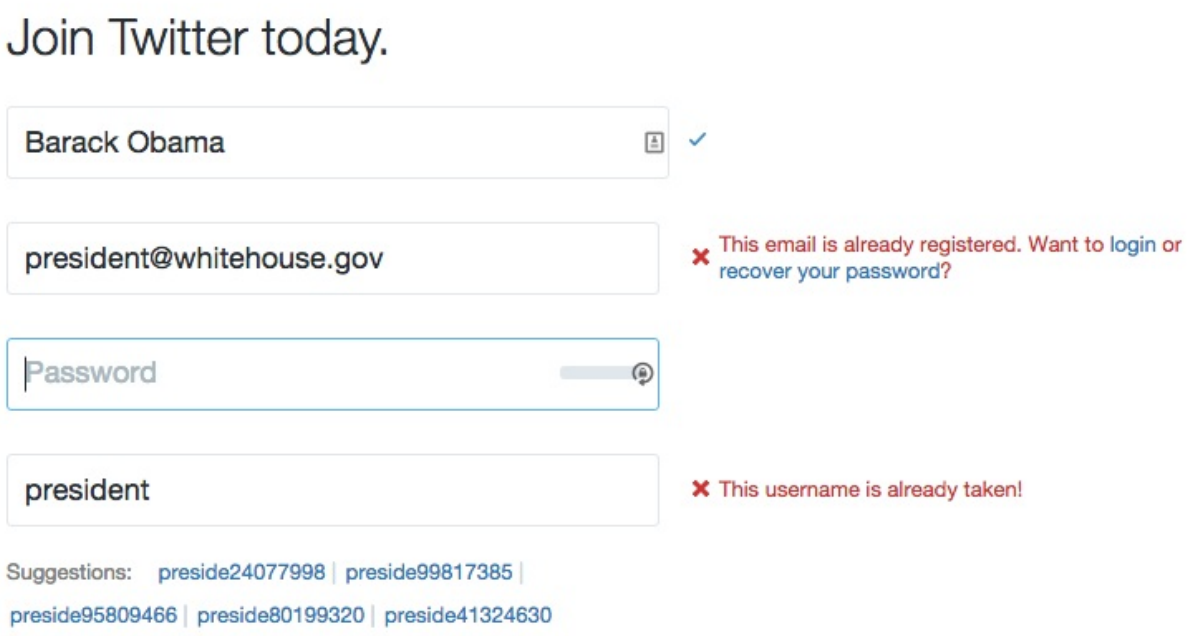


Global messages are great for letting us know that things are happening that generally apply to what we're doing. Depending on how they are presented, they can be more or less effective. For example, the alert pictured above is only visible when you scroll to the top of the page. If you are lower on the page you will never know how many tweets you have.

Local messages

Local messages appear closer to "where the action is". This is commonly seen in form field validation, especially when we're filling out more complex forms. In order to indicate where we have gone wrong, messages may be shown very close to the field (or in some way even styled as part of the form field).

Here is an example from Twitter's signup form:



As you can see in the image above, as you fill in the form the Twitter website is checking to see if your information is valid. It indicates to you clearly if you have successfully filled in the field or if you must change your information. Twitter even goes so far as to suggest alternate usernames based on the data you've filled in when your chosen name is unavailable.

Implementing messages in the weather app

In order to experiment with displaying messages using the `ngMessages` directive in AngularJS, we will add a global message to the current weather screen. When users click the "save city" button, they will see a message that indicates to them whether or not the action has been successful. This should greatly enhance the user experience, eliminating the confusion over whether or not the city was actually saved.

Here is what we hope to see after we successfully save a city:

In order to do that, we will make a couple of small tweaks to our `app/views/current.html` and `app/scripts/controllers/current.js` files.

Adding Messaging to City Saves

In order to provide a better user experience, we will add messaging to the "save city" button to indicate that the city has been successfully saved. In order to do that, we will need to make changes to the `app/views/current.html` template and the `app/scripts/controllers/current.js` file.

How `ngMessages` works

The `ngMessages` directive is a powerful tool for making different messages appear on the page. It allows you to define a set of messages in your template using HTML markup so you can style them and present them exactly as you wish. Then, in your controller, you only need to set up a Javascript object to trigger the messages.

This mechanism is supported by built in features of AngularJS, such as form validation that happens automatically based on HTML form configuration. But we can also use it however we want by setting up some simple message objects in our controllers.

We will step through the process of adding messages to the "save city" feature we created.

Create an object to control our messages

In order to know whether or not to show a message, the `ngMessages` directive expects to receive an object it can parse that contains **boolean** values (true/false). We can create this object in the function where we save the city to localStorage.

Make the following changes to `CurrentCtrl` in the file `app/scripts/controllers/current.js`:

```
$scope.saveCity = function(city){
    var cityData = {
        'name': city.name,
        'id': city.id
    };
    if (!$localStorage.savedCities){
        $localStorage.savedCities = [cityData];
    } else {
        // Check to make sure we haven't already saved the city.
        var save = true;
        for (var i=0; i < $localStorage.savedCities.length; i++){
            if ($localStorage.savedCities[i].id === cityData.id) {
                // this is a duplicate, so don't save
                save = false;
            }
        }
        if (save===true){
            $localStorage.savedCities.push(cityData);
            // Add object to trigger messages
            $scope.citySaved = {
                'success': true
            };
        } else {
            console.log('city already saved');
            // Add object to trigger messages
            $scope.citySaved = {
                'duplicate': true
            };
        }
    }
};
```

As you can see, the code above is for the `saveCity()` function, which expects to receive a `city` object and then saves that `city` into the `$localStorage.savedCities` array. We can easily define the `$scope.citySaved` object as part of this function. In the case where we save a new city, we set the `$scope.savedCity.success` value to `true`. In the case where the user has attempted to save a duplicate city, we set `$scope.savedCity.duplicate` to `true`.

These values will be interpreted by the `ng-messages` directive in our template so our messages can be shown at the proper time.

Adding messages to the view template

Now that we have the `$scope.savedCity` value defined properly in our controller, we can add the following markup to the `app/views/current.html` template:

```
<div ng-messages="citySaved">
    <p class="city-saved-alert bg-success text-success" ng-message="success">
        {{currentWeather.name}} has been saved to your list of cities.
    </p>
    <p class="city-saved-alert bg-warning text-warning" ng-message="duplicate">
        {{currentWeather.name}} has already been saved to your list of cities.
    </p>
</div>
```

The code above sets up our messages to be controlled by the `$scope.citySaved` object. First, we create a container `<div>` element, and we give it the `ng-messages` attribute. We set `ng-messages="citySaved"`, which tells the `ng-messages` directive to inspect the `$scope.citySaved` value in order to determine which messages below should be shown.

Within the `<div>` that contains our messages, we have defined `<p>` tags that possess the `ng-message` attribute. The first message, which is our "success" message, is triggered when `$scope.citySaved.success` is `true`. If that value is not `true`, then that message will not show.

Similarly, the second message is controlled by the value of `$scope.citySaved.duplicate`. If that value is `true`, then the message will be shown. If not, then it will be hidden.

Notice that the functionality of messages doesn't care about what classes, content, or HTML structures you have in your messages. Although the messages must be contained within an element that has the `ng-messages` attribute, you are otherwise free to create whatever HTML structure and assign whatever styles you wish. In this example, we are using default Bootstrap CSS styles to give the messages a "success" and "warning" look.

Try it out

If you've successfully implemented these changes to the respective files, you should be able to test it out. Click into the current weather for a city and click the "save city" button. You should see one of these messages appear. Search, view and save a few cities to get an idea of how much this improves your app. Contemplate all the different things you could do with styling to improve the look and feel of these messages. Once again, there are many possibilities to explore here. Have fun and experiment.

Files Changed in This Chapter

In order to implement the changes described in this chapter, the following files have been altered. They are quoted below so you can compare your work to a full view of the file.

app/views/current.html

```
<h1>Current Weather for {{currentWeather.name}}</h1>
<p ng-if="!citySaved"><button class="btn btn-sm btn-primary" ng-click="saveCity(curren
tWeather)">Save City</button></p>
<div ng-messages="citySaved">
    <p class="city-saved-alert bg-success text-success" ng-message="success">
        {{currentWeather.name}} has been saved to your list of cities.
    </p>
    <p class="city-saved-alert bg-warning text-warning" ng-message="duplicate">
        {{currentWeather.name}} has already been saved to your list of cities.
    </p>
</div>
<dl>
    <dt>Currently</dt>
    <dd>{{currentWeather.weather[0].main}}</dd>
    <dd>{{currentWeather.weather[0].description}}</dd>
    <dt>Temperature</dt>
    <dd>{{currentWeather.main.temp}} &deg;F</dd>
    <dt>Wind</dt>
    <dd>{{currentWeather.wind.speed}} mph at {{currentWeather.wind.deg}} &deg;</dd>
    <dt>Clouds</dt>
    <dd>{{currentWeather.clouds.all}}%</dd>
</dl>

<p><a ng-href="/#/forecast/{{cityID}}" class="btn btn-lg btn-primary">View 16-day Fore
cast</a></p>
```

app/scripts/current.js

```
angular.module('weatherApp')
  .controller('CurrentCtrl', function ($scope, $routeParams, CurrentWeather, $localSto
rage) {
    $scope.cityID = $routeParams.cityID;

    $scope.currentWeather = CurrentWeather.query({
        cityID: $routeParams.cityID
    });

    $scope.saveCity = function(city){
        var cityData = {
            'name': city.name,
            'id': city.id
        };
        if (!$localStorage.savedCities){
            $localStorage.savedCities = [cityData];
        } else {
            // Check to make sure we haven't already saved the city.
            var save = true;
            for (var i=0; i < $localStorage.savedCities.length; i++){
                if ($localStorage.savedCities[i].id === cityData.id) {
                    // this is a duplicate, so don't save
                    save = false;
                }
            }
            if (save===true){
                $localStorage.savedCities.push(cityData);
                $scope.citySaved = {
                    'success': true
                };
            } else {
                console.log('city already saved');
                $scope.citySaved = {
                    'duplicate': true
                };
            }
        }
    };
  });
```

app/styles/main.scss

```
$icon-font-path: "../bower_components/bootstrap-sass-official/assets/fonts/bootstrap/"
;
@import "variables"; // Override default Bootstrap values.
// bower:scss
@import "bootstrap-sass-official/assets/stylesheets/_bootstrap.scss";
// endbower

@import "content";
@import "buttons";
@import "transitions";
```

app/styles/_transitions.scss

```
// Styles for animated transitions of HTML elements
div[ng-view].ng-enter {
    -webkit-animation: fadeIn 0.5s;
    animation: fadeIn 0.5s;
}
div[ng-view].ng-leave {
    -webkit-animation: fadeOut 0.5s;
    animation: fadeOut 0.5s;
}
```

Conclusion

You have successfully added helpful animation and messaging to your app, which has resulted in a dramatic increase in usability. This does not mean that you are finished with your app. You now must continue to discover more ways to improve the experience for your users.

In order to continue building on your app's features and usability, you should consider pursuing some of these goals:

- Add a "save city" button to the forecast view, and duplicate the proper messaging.
- Use weather icon symbols to create animated design elements to spice up your current weather and forecast pages.
- Add proper error messaging for API calls in your services. (And implement those messages in the proper templates.)

Above all, you should continue to work on the design and visual presentation of the app. What you have here is a basically functional webapp, but it lacks the personality and attractive visual styling that will truly make users embrace it. There is a virtually unlimited terrain for exploration here, so let your imagination run wild.

Appendix

This appendix provides quick reference materials and other helpful resources you may wish to reference as you work through this book.

Git Quick Reference

Throughout this book it is assumed that you have basic Git knowledge and skills. If you are new to Git (or if you just rarely use certain commands) it can be tough to remember all the steps and commands involved. There are many great resources for learning Git, and you are encouraged to go study until you feel comfortable with Git as a tool.

Until then, feel free to consult below.

Please Note: These directions refer to command line Git. If you are using a graphical Git interface, then you should be able to follow the same basic patterns and approaches described below, but you may have different names for some of these commands (or combinations of commands). Refer to the documentation for your preferred Git client.

Committing and pushing

Whenever you finish a chunk of work that has any kind of meaning at all (a good stopping place, a completed component, etc.), you should make a commit. Think of commits as snapshots of your project at a specific time: You can always roll back to that exact state as long as there was a commit. These are the "mile markers" of our process, and we can always return to each step along the way.

The commit process

When you commit you do two things:

1. Add/remove files from the "stage". (This is called "Staging")
2. Once you're happy with what files have been "staged", commit them with a message. (This is called "Committing".)

Staging Files

To add your files to the stage, you can do so with varying degrees of specificity. It's generally best practice to be very specific and controlled when adding files to your stage for committing. A recommended process may look something like this:

Run `git status` to see what files have changes. This will return a list of files that have been altered or added.

Add each file you wish to include in the commit using the `git add` command like so:

```
git add app/myfile.html
```

If you have lots of files to add, such as several files altered within the `app/` directory, you can use wildcards to add files to the stage:

```
git add app/styles/*
```

If you know you want to commit all of the altered files (and you've reviewed what those are by running `git status`), then you can use the `-A` flag from the root of your project:

```
git add -A
```

That command will add all changed files in the repo to your stage.

Committing Files

Once you've staged your files, you can commit them. Whenever you commit, you must send a message along with your commit. You can configure which command line text editor is invoked when you commit, you can use a text file to store your commit message, or you can send the message along with the `-m` flag on the `git commit` command.

It is best practice to send more than a one line commit message, especially whenever you're making a commit that you wish to be reviewed by others. However, many developers take a shortcut and add a message to their commit command. For the purposes of our work here, this should be fine.

Before you commit your changes, you should run `git status` and make sure that the correct files are staged for commit and nothing has been omitted or accidentally added. You can fix any issues using the instructions provided along with the `status` display.

Commit your changes with this command:

```
git commit -m "YOUR MESSAGE HERE"
```

You should receive a message letting you know you have successfully made a commit. You can review the commits in your project with the `git log` command, which will print a list of the commits in any project. Run `git log` and you should see your latest commit at the top of the list.

Pushing your work

Committing is only half the battle when it comes to sharing your work with others (or publishing your work online). You also need to "push". The command `git push` pushes your repository to a "remote". By default, when you clone a repo the server you cloned the repo from is labelled `origin`. You can push back to the origin with the command:

```
git push origin
```

Once you've pushed your work, it should show up on your Github repository homepage. Review the information on that page and you should see it show up.

There are many more aspects to pushing code. You can define as many remote servers as you wish and push code independently to each one. It's also possible to set up various rules for how branches get pushed and to associate local branches with remote branches, etc. All of these details are more advanced than we need for our work here, but you are encouraged to explore them as you need them.

Branching and merging

When working with Git, it's often useful to use separate branches to contain work in progress or experimental features. This is especially useful when you are working on a project with others (you would typically keep your work in one branch until it's ready to combine with everyone else's work in the main branch).

Although branching and merging is another area where Git can become quite complex and subtle differences can matter, you can get a lot of value from these features with a base knowledge of how to use them.

Make a new branch

To make a new branch, you use the `git checkout` command with the `-b` flag like this:

```
git checkout -b new-branch-name
```

This will create a new branch based on whatever your current branch is (probably `master` in our case), and then it will `checkout` that branch so you will be "in" the `new-branch-name` branch. You may now edit files in this branch.

As you edit and modify files, you will need to commit your work like normal before switching to other branches. If you switch to another branch *without* committing your work, you may cause yourself problems, so be sure to keep a close eye on which files you've changed as you switch between branches.

To commit your work in a specific branch, you follow the exact same steps outlined above (just make sure you're in the branch you wish to commit to):

1. Stage your files.
2. Make a commit with a message.

This will make the commit to the `new-branch-name` branch. If you switch back to your `master` branch, you will see that whatever changes you made do not exist in the `master` branch.

Please Note: If you `push` the changes to your `new-branch-name` there is a good chance that when you run `git push origin` it will automatically push those changes to the `master` branch (since that is the branch we based `new-branch-name` on). This is often a very **bad** situation: We do not want to accidentally alter `master` when we mean to only share our working branch, `new-branch-name`.

To avoid the issue described above, you must tell Git to make a new remote branch when you `push`:

```
git push -u origin new-branch-name:new-branch-name
```

That command tells Git to push your code. This command reads like so: First, we invoke `git`. Then we tell it we are using the `push` command. We use the `-u` flag so that it updates our local copy of the repository with the information we are giving it in this command.

The information we're giving it is that we are specifying the `origin` as the remote server. At the end of the command we also specify that we are pushing `new-branch-name` and we want to push it to the remote branch `new-branch-name`. (That is specified by the repetition of the name with the colon at the end of the line.) If the remote branch doesn't exist, then Git will create it with this push.

Once you've used that command, you can view your repository on Github and see that the new remote branch has been created.

Change to different branches

To switch between branches you use the `checkout` command:

```
git checkout branch-name
```

Remember that you should commit your changes in your current working branch before switching to another branch.

Merge two branches

To merge two branches, you use the `merge` command:

```
git merge branch-name
```

For example, if you had completed work on your new feature in the `new-branch-name` branch, you could merge that into your `master` branch with these steps:

1. Commit all your work in the `new-branch-name` branch.
2. Checkout the `master` branch.
3. Run `git merge new-branch-name` to merge the `new-branch-name` branch into the `master`

branch.

At that point, Git would work to combine the changes from the `new-branch-name` branch into the `master` branch. If it runs into trouble figuring out which changes should stay (if, for example, the same lines of a file were modified in both `master` and `new-branch-name` branches), then Git will raise a "merge conflict".

Handling merge conflicts

Merge conflicts happen invariably throughout development. Usually they are fairly simple to resolve, and work strategies that help keep your merge conflicts simple to resolve are generally preferred. That means keeping work in small chunks, constantly moving little improvements out of working branches and into the main branches of your project so everyone has the latest code.

When merge conflicts do come up, you will receive a message in your console like this:

```
git status
# # On branch new-branch-name
# # You have unmerged paths.
# #   (fix conflicts and run "git commit")
# #
# # Unmerged paths:
# #   (use "git add ..." to mark resolution)
# #
# # both modified:      index.html
# #
# no changes added to commit (use "git add" and/or "git commit -a")
```

This message is telling you that there is a merge conflict in the `index.html` file. The trouble is that it has been modified in both branches you are dealing with, so Git cannot tell which changes should stay and which should go.

Git is asking you to fix ("resolve") the merge conflict and then add and commit those changes to make a new commit. Since Git bases everything on commits, it wants to make a commit once this conflict is resolved so it has a good point of reference going forward.

If you open `index.html` you may find something like this:

```
Please
<<<<<<< HEAD
sign up!
=======
register now!
>>>>>>> new-branch-name
```

These can be a headache to look at, but they are very sensible if you read carefully. The `<<<<<<< HEAD` marker indicates the beginning of the content that was already in your current branch (in this example, `master`). There could be many lines of code, or just a few characters.

After that content is the `=======` line, which indicates the break between the code that was already in the current branch and the code from the new branch you are trying to merge in. Below the `=======` line is all that code from `new-branch-name`.

Finally, the `>>>>>>> new-branch-name` marker indicates the end of the code coming in from the new branch.

It is your job as the developer to decide what is right or wrong. You should correct it so that it reads like you'd prefer to see it:

```
Please register now!
```

Be sure to remove all of the marker lines so they don't linger in your file and cause headaches later. Once you've finished, you should make a new commit by staging the files you fixed and committing them with a message.

Please Note: There can be more than one location in a file where there is a merge conflict. It is a good idea to use the "find" feature of your text editor to find sequences of "<<<" or ">>>" to make sure you cleaned up all those conflict markers.

Additional Resources

There are many wonderful resources for learning Git. You may wish to check out some of these:

- Git SCM Book
- Git Ready
- Git and Github Learning Resources
- Github and Code School Course

Comparing CSS Preprocessors

Many novice web developers have trouble understanding the relationship between different CSS preprocessors. This is understandable because on the one hand all these tools are quite similar, and they are also all fairly high quality. The selection of which preprocessor you use will have more to do with how it fits into your overall development patterns than any objective measure of one being a lot better than the others.

Having said that, there are differences between the preprocessors, and those are worth noting. Let's begin looking at preprocessors from the highest level: SASS vs. LESS.

SASS

SASS is the "original" CSS preprocessor. It gained a lot of interest when it debuted and it has maintained a high level of popularity. SASS was initially written as part of another Ruby Gem, and then the Compass project became the preferred Ruby Gem that developers use to process SASS.

Since SASS has been popular outside of the Ruby development community, many other preprocessors that handle SASS have been created. The one this book uses, and one of the more popular, is called `node-sass`.

LESS

LESS is an alternate take on a CSS preprocessor language, but its development was influenced by the existence of SASS. It was, initially, the only pure-Javascript solution for preprocessing CSS. It generally works a lot like SASS, with some differences that, in some cases, can be key.

SASS vs. LESS

This debate has raged for awhile and I'm not going to settle it here. If you go look at a Google Search for the term "sass vs. less" you'll find plenty of opinions.

But the prevailing trend these days seems to be towards SASS. More robust non-Ruby support (meaning, there are SASS processors written in many languages for many platforms) and continuing high adoption rates among developers really give it an edge.

What's really important?

The specific choice of preprocessor, at least between SASS and LESS, is not the most important aspect of implementing a preprocessor into your work. Unless you are working with a very unique set of cases, you will not run into a problem with either preprocessor that will make it impossible to make the styles you want.

You should focus on the things that are always most important when making choices about what components to use in projects, including:

- Is this a technology in use by others? Or will I be alone using it?
- Is this a technology my team knows and/or is enthusiastic about adopting?
- Can I plug this technology into the systems that make all the rest of my work easier (continuous integration, testing, preview, etc.)?
- Will using this technology impact my long-term maintenance in a negative way?

Since properly organizing styles, collaborating with colleagues, and delivering working code on a consistent rhythm are often the most important things to consider when building styles, it's much better to focus on those things and let your preprocessor tools get out of the way.

Resources and Links

The following is a list of references, resources, and links useful for people reading this book.

Core components of the project

- Yeoman
- Bower
- Grunt
- Node.js
- AngularJS

Additional tools and resources

- Git
- SASS
- grunt-build-control
- node-sass