Front-End FUNDAMENTALS

Joe Fender | Carwin Young

Front-End Fundamentals

A practical guide to front-end web development.

Joe Fender and Carwin Young

This book is for sale at http://leanpub.com/front-end-fundamentals

This version was published on 2015-02-11

Thanks to,

Sally Rose, Carrie Young and Justin Harrell

Table of Contents

About the Authors

Joe and Carwin started working together in 2013 at Lullabot, an interactive strategy, design and development company.

At Lullabot, they work with decision makers, developers, and designers at all stages of a web project. They have guided the development of sites for clients such as Martha Stewart, The GRAMMYs, MTV UK, Sony Music, Lifetime Television, The George Lucas Educational Foundation, Leo Laporte's TWiT network, and many others.

Joe Fender

Joe Fender lives in London, UK and splits his focus between front-end and back-end development. Specializing in PHP, AngularJS and Drupal, Joe was the lead architect at Studio Umi, the largest Drupal shop in Japan before joining Lullabot to work on awesome websites such as Drupalize.Me and MSNBC.

In his spare time, Joe likes to collect shoes and work on side projects. But never at the same time.

Carwin Young

Carwin Young is an expert front-end developer living in Springfield, Missouri. He has over 8 years experience working on the front-end. He was the second front-end developer to work at Lullabot where he has played a key role on building websites for clients such as MSNBC, BravoTV and the GRAMMYs.

Carwin loves to share his expertise. He has spoken on topics such as Rapid Drupal Development at DrupalCamp Chicago, is a co-organizer for the Springfield Drupal meet-up and is a regular contributor to the Front-end Rapport, an online magazine for front-end developers.

Introduction

Front-end web development is evolving. Long gone are the days of writing static HTML files with self-contained CSS. As web developers in this modern world, we need to understand and utilize an endless number of frameworks, plugins, techniques and more. Knowing the right tools to use and when to use them is key to building successful solutions.

Despite our best efforts, it can be difficult to keep up with the speed of advancing web development. We sure as hell can't slow it down. And why would we want to? It's so exciting to not know where technology is headed and what could be around the next corner. We should both prepare for change and embrace it. And that is exactly where front-end developers excel. We build what the consumer sees and because of that, we want to use the most cutting-edge technologies to create the biggest impact.

This book came to be from the lessons we, the authors, have learnt through trial and error whilst working on front-end development projects ranging from enormous high-traffic sites such as MSNBC, The GRAMMYs and BravoTV to startups and experiments such as Drupalize.Me and BracketCloud. We want to share with you the tools that we use and the standards that we follow. Our ultimate goal is for you to walk away understanding the core concepts of front-end development so you can confidently go and work on your own projects.

This book is somewhat opinionated based on our experience, so you'll find some of the content heavily geared towards a particular way of doing things. Of course, there are many ways of accomplishing the same task in the world of front-end development; that's what makes it so accessible and yet so complicated. Learning one way of doing something can make learning the alternatives a much more palatable task, so we encourage you to look into any and all of the technologies presented herein and even go on to see if something else might better fit your needs.

Who This Book Is For

The content in this book is aimed at those who are new to front-end web development. Regardless of whether you are a beginner or expert developer, the concepts outlined in this book are essential to embracing the rapidly evolving web. We won't be covering the general basics of web development so you will most likely want to get up to speed on the fundamentals of web programming such as HTML, CSS and JavaScript. We will also be utilizing the command line, so some basic knowledge of how to use that on your operating system is recommended.

If you hear yourself asking any of the following questions, this book will likely be very useful for you!

- “What actually is front-end development?”
- “How can I be a more efficient web developer?”
- “What is Grunt/AngularJS/SASS/<*insert popular tool name here*>?”
- “How do I fit all of these front-end tools together into a project?”
- “How can I build a front-end application that can scale?”

An Overview of This Book

Every web project that you work on is different and there is no ‘one size fits all’ set of tools. However, throughout the chapters in this book, we’ll introduce you to some of the most popular, useful and powerful tools used in front-end web development so that when you’re done reading you will be able to apply the concepts and techniques to whatever you are working on.

1. Getting Started

The aim of the first chapter of this book is to help get you acclimated with front-end development in general. You’ll learn what’s expected from a front-end developer and get a short overview of the types of projects a front-end developer might be tasked with. We’ll even take you through setting up your computer for local development.

2. Frameworks

This chapter jumps right into some of the more popular JavaScript frameworks and template languages in use today. We’ll cover the details of what exactly a ‘framework’ is and talk about the MVC pattern before getting into the details of installation and practical usage of 3 popular frameworks: AngularJS, Backbone.js and Ember.js.

3. Styling

There’s a lot more to styling than just throwing CSS at your markup; that just leads to a big mess of unreadable spaghetti code sprinkled with `!important` parmesan. Here we’ll cover CSS preprocessors like Sass, organizational methods for your styles, and the kind of impact your CSS may have on a project’s performance.

4. Dependency Management

Getting your whole team on the same page code-wise can be challenging. Heck, even making sure you’re using the same version of that awesome Ruby gem after an update can be a pain. This chapter on dependency management covers a number of really handy front-end package managers like `npm`, `Bower`, and `Bundler`. You’ll learn what a package manager is and how to use it effectively to keep your project on track.

5. Automation

Do you like doing the same repetitive tasks over and over? No? Neither do we.

The automation chapter is about just that, automation. We’ll go over how you can remove the tedium from front-end development and just get to work. In this chapter you’ll learn how to perform tasks like checking your JavaScript or CSS for errors whenever you save a file, minify your code to shave off some file size, and even how to automatically refresh your browser whenever something changes. Once you get started, you'll never look back.

Example Code

This book contains many code snippets for demonstration purposes. Code may appear differently depending on the device you are using to read this book. Here is an example of a code snippet so you know what to expect:

```
var gulp = require('gulp'),
  uglify = require('gulp-uglify');

// This task uglifies our JS files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

You will also notice that we reference code inline `like this` throughout the book.

Contacting the Authors

If you have any feedback or questions, please feel free to contact us via the Front-End Fundamentals Google Group.

You can either create a new topic via the web interface or you can send an email to the mailing list address: front-end-fundamentals@googlegroups.com

We will do our best to consider each and every new message received. Please understand that depending on the volume of requests, it may be difficult for us to respond immediately.

Getting Started

Traditionally, when web developers started building websites in the 90s, they would wear all the hats. They would set up the server, write all the server-side code (in PHP) that would eventually output HTML and finally sprinkle on some CSS to make things look pretty. If you have a background in web development, this may sound very familiar to you.

Due to the advancement of modern web technologies and computer hardware, we’re able to create more sophisticated experiences than ever. Because of this complexity, the role of a web developer has naturally evolved and split into separate disciplines. Specifically, a line was drawn between the back-end and front-end. Back-end being server-side programming and front-end being client-side. Front-end should not be confused with web design. Although front-end developers can benefit from possessing designer chops, their role isn’t to design a website but rather to implement a design given to them. Depending on the size of team you work with, you may find the lines between these roles blurred. You should definitely read this great article on the confusion which revolves around the definition of a front-end developer.

The divide between back-end and front-end is still new. We’re not yet completely comfortable with it. Some back-end developers still do a lot of front-end work, or vice-versa. Sometimes it can be tricky to figure out how to communicate from either side of the divide. However, when you look at job postings for web developers these days you’ll often see them listed as either for a back-end or front-end candidate. The industry as a whole has embraced the concept of role separation for web developers which can only lead to good things as we can now specialize in our discipline of choice.

Being a front-end developer is so exciting. You’re on the ‘forefront’ so to speak and have a direct impact on how the user feels about and interacts with the product or service that the websites you build portray. You can instantly see the results of your work and because of this you will receive a lot of feedback, sometimes harsh. But take it with a grain of salt and use it constructively to improve on yourself and the work that you do.

Expectations of a Front-End Developer

What you’re going to realize by the time you’ve finished this book is that the front-end world is big. There are so many people working towards the same goal but with different points of view on how to get there. That is the beauty of open-source. You’re going to feel a little overwhelmed but don’t let that faze you. Learn to love trying new things and embrace change as it comes and you’re going to do really well. Here is a summary of the various things you should consider when trying to be the best front-end developer you can:

- Rather than setting out to master a single tool, try to learn a little about each of them and how they work together first. When you've grasped the bigger picture, it is much easier to specialize. If you're reading this book, you're definitely on the right track.
- Love to learn. To keep up with the speed of evolving web technologies, you'll have to pick up new things regularly or you may fall behind.
- Be a good communicator. Even if you're working alone, you'll find that there are many opportunities online to interact with other developers and get support.
- Google is your friend. When anyone asks me "How do you go about solving a problem?" my answer always starts with Google and that is perfectly OK! Google is the keeper of much knowledge and you should learn to use it in abundance.
- Don't reinvent the wheel. You're going to find someone else has probably thought of a similar solution to the problem you're facing. It may not be the perfect match but by bending your project to make room for pre-existing solutions, you're also benefiting from the community and support that comes with them.
- Cross-everything. You're responsible for making sure what you build actually works and isn't dependent on the user having a specific browser or device. Build cross-browser, cross-platform and cross-device. Ask outside the box questions like "What if I am accessing this website from a mobile on a slow 3G connection?" or "What if I am viewing on an iPhone 4 in landscape?".
- Test and then test again. Test for visual regressions (e.g. using Wraith or PhantomCSS), run unit and functional tests on your JavaScript (Karma or Jasmine are good places to start) and test for general performance (read the Yahoo best practices and try YSlow). A well tested site makes all the difference so make sure to create time for testing.
- Think about accessibility. For example, ask yourself how will a blind person or someone who is colorblind interact with your website? Here is a useful checklist of things to think about.
- Document everything. Not only does it help educate new developers about the theory behind your code but the act of writing comments will help you to spot flaws in your logic.
- Save time with automation. There are a lot of repetitive tasks in front-end development that can be automated to save time. We find it so important that we've even written an entire chapter on automation in this book!
- Learn about collaboration tools. As a web developer, you are going to use Github at some point. So you would be better off looking into Git and its various popular workflows such as git-flow or Github flow.
- Read about Continuous Integration. Although you may not be able to incorporate it into every project, learning about its principles will help you approach code deployment in a more intelligent way.
- Have a good eye for design. A somewhat overlooked asset of front-end developers is being able to provide feedback to designers. Seeing as we're the ones who actually build the interactions, we will probably use the website a lot more than the designers ever will. They will greatly appreciate any feedback you can give so don't be hesitant to point out flaws in the UI.
- Learn about server-side web development concepts. Although you may not be touching the back-end, it can help if you know what is going on and how the data you

are using actually got to you.

Setting Up Your Local Environment

A lot of front-end development happens on either a local machine or a non-production web server set up for multiple people to use. Long gone (hopefully) are the days of just editing CSS on a live site by overwriting CSS files over FTP. As developers, we prefer to do everything locally so we don't need to worry if we're going to accidentally ruin our production site. Also it's just plain faster since there's no need to connect to the internet.

Setting up your local environment could probably be a book in itself (there are a ton of viable options), but we're going to cover what we think is probably the easiest method: MAMP.

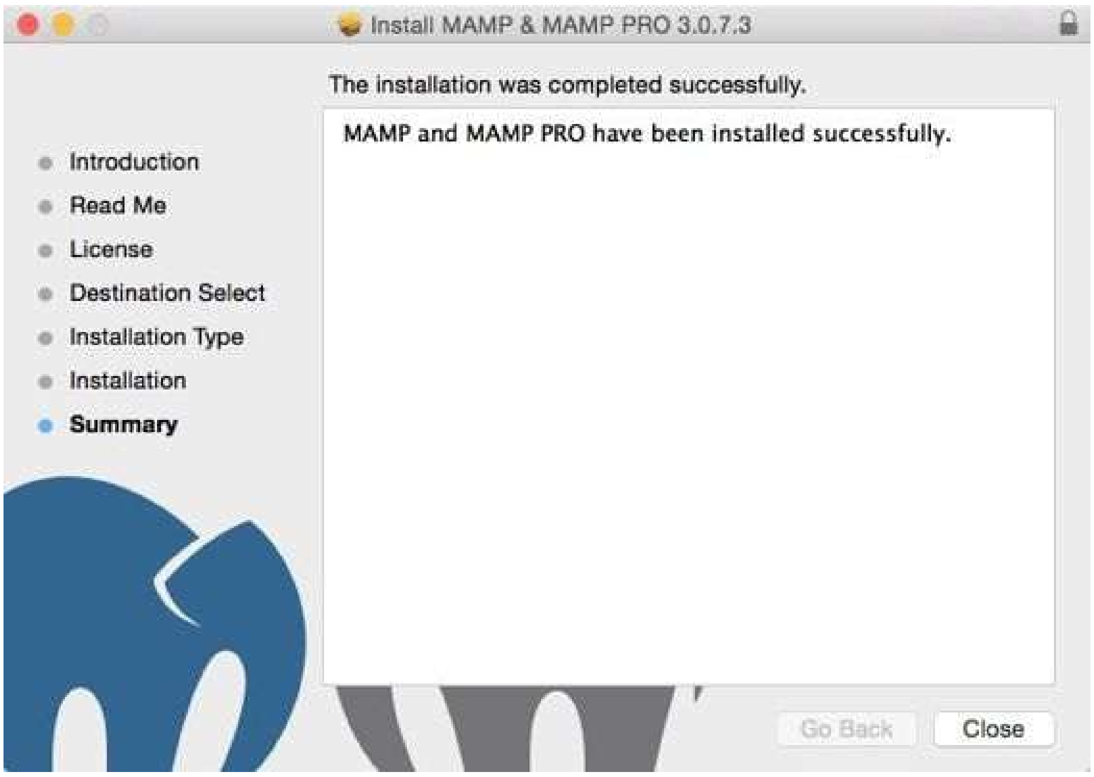
MAMP

MAMP is an easy to use, free application for Mac and Windows based machines. MAMP will install PHP, MySQL, and the Apache webserver in a simple to manage environment and you can start up your web server with a click of a button.

Installation

Let's walk through a first run:

1. The first step in the installation process is to go download the MAMP installer for your operating system.
2. Once the download completes, run the installer program that you downloaded.
3. Walk through the installation process. Once it finishes, you'll be ready to start using MAMP*.


The installation was completed successfully.

MAMP and MAMP PRO have been installed successfully.

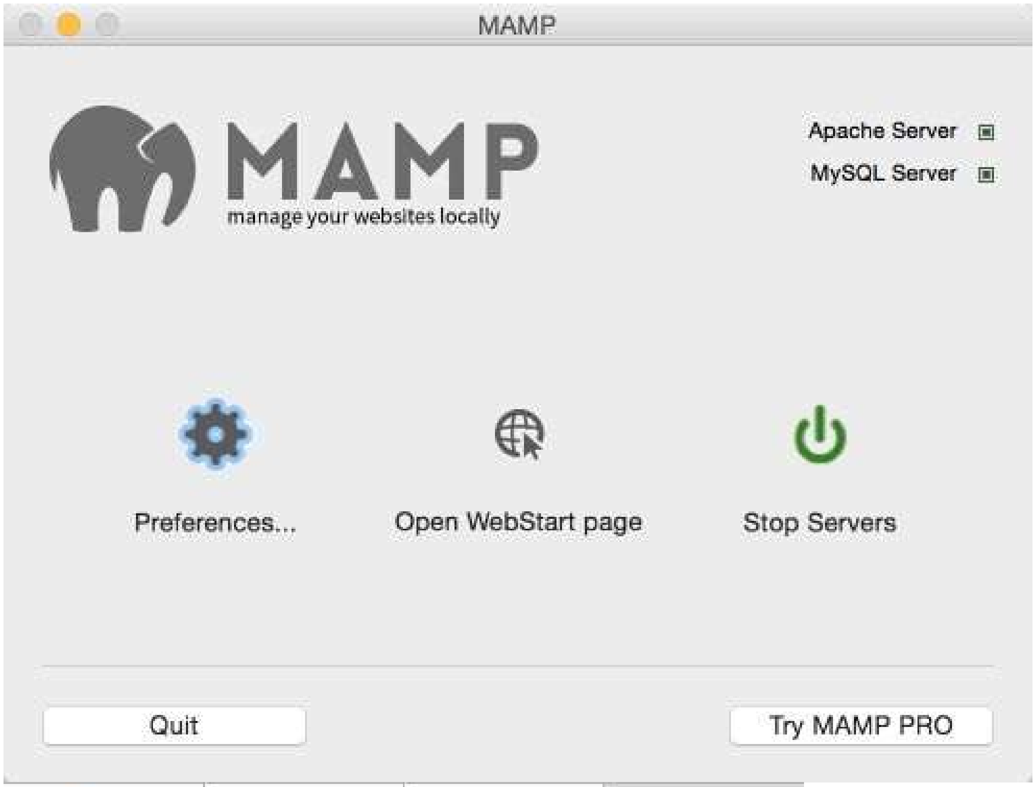
*: The MAMP installer also installs MAMP PRO which has some extra features if you choose to pay for them.

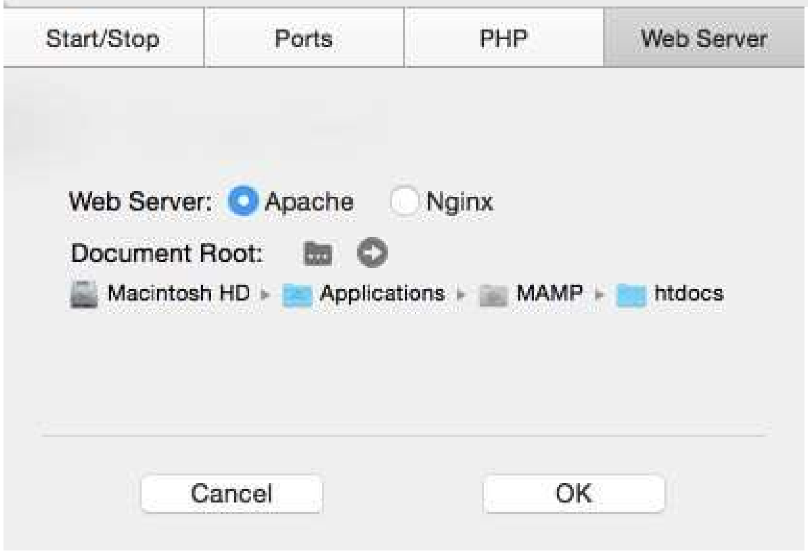
Setup

Once you've made it through the installer, run the MAMP application.

When MAMP is open, you'll see a "Start Servers" button on the right side of the screen. You can click it now and you'll be in business with your document root set, which is the place all your site files will go, to the default path. In the case of OSX, the default document root is in the `htdocs` directory wherever you installed MAMP (typically in `/Applications/MAMP`).

If you want to change your document root to another location, change the version of PHP that the server uses, or a number of other settings, make sure to stop the servers and head to the Preferences pane of MAMP.





Web Server: [✓ Apache] Nginx

Document Root:
Macintosh HD ▸ Applications ▸ MAMP ▸ htdocs

The default configuration of MAMP is fine for getting started, so if you don't need to customize anything just start the servers and you should be able to access a test page at http://localhost:8888/MAMP/. This page isn’t on the internet anywhere, it’s on your local machine! If that link works for you, you’re ready to start developing. Just create a simple html page inside your document root folder, and you should be able to access it at http://localhost:8888/your_file.html.

For a more detailed installation and set-up guide, take a look at the MAMP user guide.

Summary

MAMP is a great tool and one that’s easy to set up and get started with. There is a big wide world of web server solutions for developing locally; heck, if you’re on a Mac you have Apache/PHP/MySQL already! There just isn’t a great interface for putting those things to use, so many developers prefer MAMP. In the end, however you go about serving your HTML/CSS/JS to a browser is fine. What’s important is that it works for you.

Frameworks

JavaScript allows you to write client-side scripts for manipulating the DOM and for interacting with the user and their browser. JavaScript is commonly written procedurally using libraries such as jQuery. As HTML, browsers and JavaScript have become more powerful, so has the complexity of JavaScript files. But this growth can quickly lead to spaghetti code and a whole bunch of maintenance nightmares as it is quite common in web development for a site's entire JavaScript to be maintained in a single file. Over the years, developers have attempted to tackle the issue of growing demand for JavaScript by creating 'frameworks' to approach implementation in a more modular way. Frameworks help to organize code.

There are a large number of JavaScript frameworks available. In this chapter you'll be introduced to some of the most popular ones.

The majority of frameworks incorporate a variation of the 'Model-View-Controller' (MVC) architectural pattern. It's worth noting that you may come across a few in a the wild that introduce their own take on MVC, or they may have an entirely different structure. If you have a background in software engineering you may already be familiar with MVC, but let's go over it briefly so we're all on the same page.

Model-View-Whatever Architecture

MVC is a popular architectural pattern that is used in many JavaScript frameworks. As previously mentioned, there are some frameworks out there that put their own spin on the MVC pattern. These frameworks implement what is sometimes referred to as an MV* or MVW (Model-View-Whatever) architecture.

However, it is important to know what MVC is, and why it has been such an important part of the way applications are built. MVC separates the different aspects of an application into three components:

- **Model**

  Models represent an application's data. You can think of each model as a different category of data that an application will store. For example, a typical web application may have `User` and `Article` models. Models do not output anything, as this is the job of 'views', but they are commonly responsible for telling views when their data has been updated.

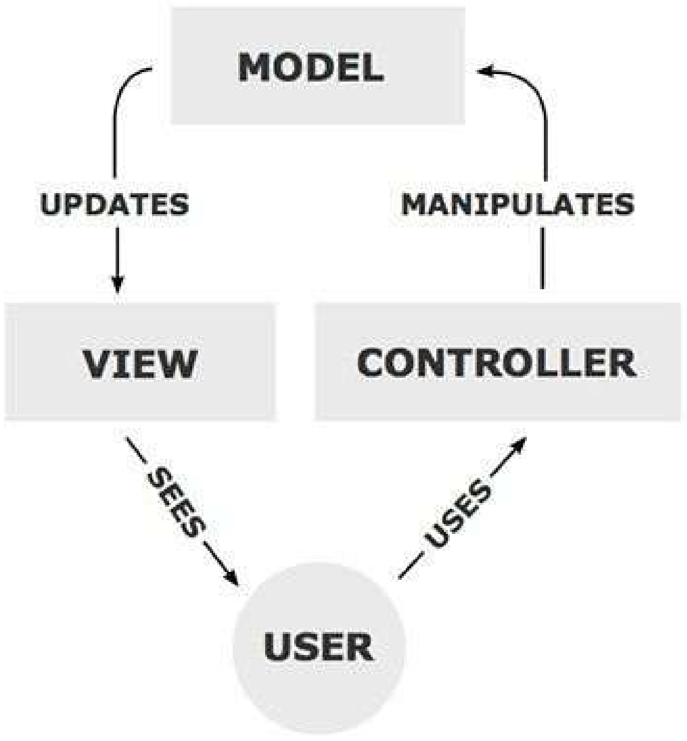
- **View**

  Views handle output. In most implementations they will output the markup that represents the user-interface. Views request information from models in order to generate this output, and models keep them up to date.

- **Controller**

  Controllers are responsible for handling user input such as form data submission or click events. They can send requests to models in order to manipulate their data.

The following diagram may help you visualize the various components of an MVC from the user's perspective:



If you fancy reading more about JavaScript architectural patterns, Addy Osmani's book Learning JavaScript Design Patterns is highly recommended.

Single-Page Applications

When learning about JavaScript frameworks, one term you might see floating around is 'single-page application', or 'SPA'.

When building an SPA, the UI and application logic is essentially shifted from the server to the client. All the resources for an application such as HTML, CSS and JavaScript are downloaded in a single page load. As the user navigates a site, the browser will not reload the page but instead the SPA is responsible for rendering the different parts of a page at the right time.

Creating an SPA provides a more fluid user experience than the more traditional approach of requesting a new page every time the user navigates somewhere. By moving all the application logic to the front-end, the requirements of the server are drastically reduced. Instead of the server rendering each page every time there is a page reload, an SPA dynamically requests only the required raw data behind the scenes, and uses that to render the various views.

JavaScript frameworks can make it fairly easy to build an SPA. In fact, some of them build their functionality entirely around the concept. However, there are a few disadvantages that you may want to consider before taking the plunge.

For example, SPAs need JavaScript. If the user disables JavaScript in their browser, the site won't render. It's difficult to say how many users actually do this but according to certain studies it's less than 1%. So you have to ask yourself if it is actually worth supporting those users and if not, what experience can you provide for them instead?

You'll also need to consider robots. Maybe not the kind of robots you're thinking of but the ones that Google, Facebook and Twitter can send to find out about a page. These robots scrape a page for meta tags and other information about content so they can display them appropriately in search engines or social networks. More recently, Google robots have started to render JavaScript. This means that when a Google robot hits an SPA, it should correctly render the page first using JavaScript. However, other robots (such as Facebook's) are yet to implement this functionality, and this can cause shared pages from a site to produce undesirable results. You'll need to come up with a way to render data server-side for these robots. Services exist to solve this exact problem, such as Prerender.io.

Before you start to use JavaScript frameworks to build SPAs, think carefully about the requirements of your application. It can be quite easy to over-complicate things. For example, if you're building an application where there is a heavy emphasis on user interaction, such as a ticket management system or a to-do list, then you'll definitely benefit from using a JavaScript framework and SPA. However, if you're creating a page which only requires very basic interaction, such as a portfolio page giving details about you and your work, then it could be overkill. You may find that the traditional approach to web development (i.e. rendering on the server-side using a language such as PHP and then adding on the client-side interactions using jQuery) would make life a lot easier.

AngularJS

AngularJS is an open-source web application framework that is maintained by Google. It is a great fit for dynamic web applications and SPAs.

Overview

The way that AngularJS (more commonly referred to as Angular) approaches web application development is unique. It promotes HTML syntax extension to include additional tag attributes to elements. These attributes, known as 'directives', allow you to attach new behavior to DOM elements. Angular comes packaged with several directives for common activities such as data binding, repeating or hiding DOM elements, event handling and form management. You can also write your own directives, but that will be covered later.

Installation

As with most JavaScript frameworks, to install Angular all you need to do is include the library on your page. Let's work through an example together to illustrate this. In our example, we'll lay the foundations for an application that will allow users to view the timezones for different cities around the world. Let's call it Timezone Finder. You can find the source code for Timezone Finder in this repository: https://github.com/fender/TimezoneFinder.

Create a new folder for the project and create an `index.html` file at its root:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timezone Finder</title>
  </head>
  <body>
    <p>Hello world! Welcome to Timezone Finder.</p>
  </body>
</html>
```

Head over to the official AngularJS home page and click the download button. Google hosts Angular on its CDN so you're able to leverage that quite easily with a simple copy-and-paste of the given CDN URL. Alternatively, you can download the file directly or, if you prefer to host it locally, you can use Bower (see the Dependency Management chapter).

Let's return to our example and add the CDN reference to our `index.html`:

```
<head>
  <meta charset="utf-8">
  <title>Timezone Finder</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular.mi\
n.js"></script>
</head>
```

Next, we can create a `app.js` file that will be responsible for kick-starting the Angular application. But first, let's just add a single line to output the included version of Angular to the browser console so that we can confirm it has been loaded successfully:

```
console.log(angular.version.full);
```

We need to make sure we include `app.js` in our page *after* Angular has been included:

```
<head>
  <meta charset="utf-8">
  <title>Timezone Finder</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular.mi\
n.js"></script>
  <script src="./app.js"></script>
</head>
```

Visit the site on a local web server and you should see the Angular version number output to your browser console.

Modules

Any custom functionality that you wish to implement with Angular must be defined in a module. When Angular starts it looks for a default module to auto-bootstrap the application. You can tell Angular which module to use by using the ngApp directive.

In our example, let's add the directive to our `<body>` tag:

```html
<body ng-app="timezoneFinderApp">
```

Angular refers to directives in lowerCamelCase (e.g. ngApp) but when you write them in your markup you need to write them in lower case and separate words with hyphens. In our example, we've named our module `timezoneFinderApp` so let's go ahead and create this module in our `app.js` script file:

```javascript
var app = angular.module('timezoneFinderApp', []);
```

That's all it takes to initialize a new module. However, nothing will actually happen just yet. Let's move on to controllers to really get this party started!

Controllers

Angular controllers allow you to write functionality that interacts with a scope object. It can take a little time to get your head around scopes and how they work but it's important to remember that they are the glue between controllers and views. Both controllers and directives have access to the scope object but not with each other directly.

The best way to understand this is to return to our Timezone Finder example. Let's write our first controller directly in `app.js`:

```javascript
var app = angular.module('timezoneFinderApp', []);

app.controller('MainController', ['$scope', function ($scope) {
  // Defines our cities.
  $scope.cities = [
    { name: 'London', timezone: 'GMT' },
    { name: 'Tokyo', timezone: 'JST' },
    { name: 'Melbourne', timezone: 'EDT' },
    { name: 'Los Angeles', timezone: 'PST' },
    { name: 'New York', timezone: 'EST' }
  ];

  // The default city.
  $scope.selectedCity = $scope.cities[0];
}]);
```

We've named our controller `MainController` since, for this example, we're only going to need a single controller. However, for larger applications you'll often end up wanting to name them in relation to the functionality that they implement.

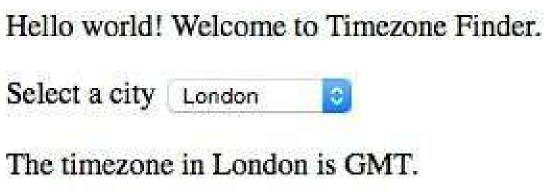
You'll see that we've attached two variables to the `$scope` object. `$scope.cities` is an array of city objects, each with a name and a timezone. `$scope.selectedCity` is a reference to the first item in the cities array.

The magic starts to happen when we switch over to our markup. Let's open up the `index.html` and add something like this inside the `<body>`:

```html
<p>Hello world! Welcome to Timezone Finder.</p>
<div ng-controller="MainController">
```

```
  <label>Select a city</label>
  <select ng-model="selectedCity" ng-options="city.name for city in cities"></se\
lect>
  <p>The timezone in {{selectedCity.name}} is {{selectedCity.timezone}}.</p>
</div>
```

We have used several new directives here. Firstly, we attached `MainController` to the DOM using ngController. We've then got a `<select>` element that implements ngModel which sets up data-binding between the element's selected option and `$scope.selectedCity`. When choosing a different option, the scope object will automatically update any other place where it is being used, such as in the message below the `<select>` element. Lastly, ngOptions is a directive specific to `<select>` elements. If we run our application we should see something like this:



Hello world! Welcome to Timezone Finder.

Select a city London

The timezone in London is GMT.

An interesting point to mention here is we've actually just covered all parts of an MVC architectural pattern within Angular. The models are the properties we set on the `$scope`. The HTML with data-bindings that we wrote are our views, and the `ngController` specifies our controller function that contains our business logic.

Routing

Angular is exceedingly well suited for building SPAs. Angular 'routing' allows you to create multiple views and have them loaded separately depending on the URL path being requested by the client.

Routing functionality is not baked into the AngularJS core library. It is provided as a separate module called ngRoute. Just like when we included the core library, we can use the CDN address to grab the file. We'll update our `<head>` to look something like this:

```
<head>
  <meta charset="utf-8">
  <title>Timezone Finder</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular.mi\
n.js"></script>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular-ro\
ute.min.js"></script>
  <script src="./app.js"></script>
</head>
```

Now we can add a new page to our application which will demonstrate how routing can be implemented in Angular. Let's have the new page display a list of all the available cities and their respective time zones. We can call it the 'Full List' page. We'll need to create a new template (i.e. an HTML markup file) for the page. Create a file called `list.html` in the root directory and place this markup inside:

```
<h1>Full List</h1>
<ul>
  <li ng-repeat="city in cities">
```

```
      The timezone in {{city.name}} is {{city.timezone}}.
  </li>
</ul>
```

That’s all we need to add. Don’t worry about `<html>` tags or anything like that. This file is just a stub for other templates to use. Now, depending on the URL used to access the page, we will ask Angular to include the markup into `index.html`. Before setting up our routing, let’s also create a file called `welcome.html` in the root directory and add the following familiar markup:

```
<h1>Welcome</h1>
<p>Hello world! Welcome to Timezone Finder.</p>
<label>Select a city</label>
<select ng-model="selectedCity" ng-options="city.name for city in cities"></sele\
ct>
<p>The timezone in {{selectedCity.name}} is {{selectedCity.timezone}}.</p>
```

Let’s move on to the `index.html`. We’re now able to remove the welcome markup from inside our `<body>`:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timezone Finder</title>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular.\
min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.2.27/angular-\
route.min.js"></script>
    <script src="./app.js"></script>
  </head>
  <body ng-app="timezoneFinderApp">
    <div ng-view></div>
  </body>
</html>
```

We’ve added a new element with the ngView directive attached. This directive tells Angular where to include the template that is required for the current route (i.e. URL path) being accessed. When the route changes, the directive will automatically update the element with the contents of the relevant template.

Now we can jump over to our JavaScript file `app.js` to add the logic for routing. First of all, we need to create a dependency on the ngRoute module. Without doing that, we’re unable to use the functionality that the module provides. For more information on how this works in Angular, check out the official documentation on dependency injection. Add the dependency to the first line of `app.js`:

```
var app = angular.module('timezoneFinderApp', ['ngRoute']);
```

Now let’s implement our routing logic. We’ll do this inside our modules `config` block. This method allows us to set up configurations for our module before any controllers are executed:

```
var app = angular.module('timezoneFinderApp', ['ngRoute']);

app.config(['$routeProvider', function($routeProvider) {
  $routeProvider.
    when('/welcome', {
      templateUrl: 'welcome.html',
      controller: 'MainController'
    }).
    when('/list', {
      templateUrl: 'list.html',
      controller: 'MainController'
    }).
    otherwise({
      redirectTo: '/welcome'
    });
}]);

app.controller('MainController', ['$scope', function ($scope) {
...
```

As you can see, we've injected the `$routeProvider` service into our `config` block. Then, using the `$routeProvider.when()` function, we were able to define our custom routes. We've set up two routes for each template that we have. When the URL path matches either of those route definitions, it will grab the markup from the `templateUrl` and place it inside the ngView directive that we placed inside `index.html`.

We can also tell Angular to load a specific controller for either of our routes. These don't have to be the same but for our application example we only need to execute the `MainController` controller. We've also used `$routeProvider.otherwise()` to redirect to the welcome page when no other route definition is matched. Read the [$routeProvider documentation] to learn about all the available options.

If we run the application again, we might notice that the URL updated to include the `/welcome` route because we accessed the page without a route and our `redirectTo` kicked in. The full URL will actually look something like `http://localhost/TimezoneFinder/#/welcome`. The hash is added by Angular by default as part of its [$location] service hashbang mode functionality. However, we can turn on the HTML5 mode which is based on the [HTML5 History API] and this will remove the hash. Pretty much all modern browsers support this functionality and Angular will fall back to using the hashbang mode for older browsers. See [Can I Use] for a full list of browser support.

Let's update our `config` block:

```
app.config(['$routeProvider', '$locationProvider', function($routeProvider, $loc\
ationProvider) {
  $locationProvider.html5Mode(true);
  $routeProvider.
    when('/welcome', {
...
```

Now, when we access the application via the root URL we should be redirected to the welcome page without a hash appearing in the URL. Note: If you are accessing the application in a sub-directory (e.g. `http://localhost/TimezoneFinder/`) then you will need to add the `<base>` tag to your `index.html` inside the `<head>` as follows:

```
<head>
  <meta charset="utf-8">
  <title>Timezone Finder</title>
  <base href="/TimezoneFinder/" />
...
```

Ok. So to be able to navigate between the routes, let's add a navigation menu to our `index.html`.

```html
<body ng-app="timezoneFinderApp">
  <ul>
    <li><a href="welcome">Welcome</a></li>
    <li><a href="list">Full List</a></li>
  </ul>
  <div ng-view></div>
</body>
```

Run the application again and you'll see that we can navigate between the different views. Although removing the hash from the URL makes them look pretty, you might notice that if you try to access a defined route directly that you will receive a 404 Not Found error message. This is because Angular is serving all of its files through `index.html` file. You'll need to configure your web server to handle rewrites. Check out this very useful Github wiki page that explains how to do this for various popular web server software.

Custom Directives

Angular provides a whole bunch of useful directives out of the box. You can get pretty damn far without ever needing to write your own. However, it can be very useful to know how to write your own directives. Even if you don't plan to use a custom directive, writing one may help you understand the inner goings-on of Angular.

Let's create a directive for our Timezone Finder application. If you look inside our template files, we are rendering the message `The timezone in {{selectedCity.name}} is {{selectedCity.timezone}}.` both on the welcome and full list pages. If we wanted to change the markup for these messages, we would have to change code in two separate places. To prevent this, let's create a directive that will render the message within any element it is attached to.

We can open the `app.js` file and add the directive block as follows:

```javascript
app.directive('cityTimezone', function() {
  return {
    scope: {
      city: '=city'
    },
    template: 'The timezone in {{city.name}} is {{city.timezone}}.',
  };
});

app.controller('MainController', ['$scope', function($scope) {
```

Remember that the Angular standard is to name directives in lowerCamelCase. We've called our directive `cityTimezone`. It requires a `city` to be passed as a parameter. The template markup is returned directly but you could use a separate template HTML file using templateUrl if you wanted to.

Returning to our example, let's add the new directive to our template files starting with `welcome.html`:

```html
<h1>Welcome</h1>
<p>Hello world! Welcome to Timezone Finder.</p>
<label>Select a city</label>
<select ng-model="selectedCity" ng-options="city.name for city in cities"></sele\
ct>
<p city-timezone city="selectedCity"></p>
```

And then our second template file, `list.html`:

```html
<h1>Full List</h1>
<div ng-controller="MainController">
  <ul>
    <li ng-repeat="city in cities" city-timezone city="city"></li>
  </ul>
</div>
```

Visit the application again and nothing will have changed from an end user's perspective. However, if we wanted to improve the message or its functionality in the future, we now only need to change code at a single location.

There are plenty of other configuration options for directives so be sure to read the official documentation to find out more.

Advanced

It would be great to cover Angular in more depth but this isn't an Angular book. However, before using Angular in your own projects, it's recommended that you gain some understanding of the following:

- **Services** - Angular services are great for separating out functionality that is to be shared throughout an application. There are endless implementations, such as storing logged-in user details, accessing external APIs and others. Read the Services documentation.
- **Filters** - Angular filters are used for formatting output before it is displayed to users. They can be used in various parts of an application and you can even build your own. Read the Filters documentation.
- **Testing** - Get into the habit of writing unit tests as you add functionality; it can save you a lot of time whilst debugging and testing your application as it grows. You'll need to learn about some additional tools for testing such as Karma, Jasmine or Protractor. For more information, see the unit testing and end to end testing documentation pages.

If you're serious about learning AngularJS then this awesome list of education resources is highly recommended.

Another recommendation to check out is AngularJS Batarang, a Chrome extension for debugging and profiling Angular applications.

Backbone.js

Backbone.js is another popular JavaScript framework for building web applications. It was created by the amazing mind of Jeremy Ashkenas who is also the creator of CoffeeScript. A fair number of large-scale applications have been built using Backbone such as Airbnb, Pinterest and Soundcloud.

Overview

Backbone is often referred to as a library rather than a framework since it plays well with other JavaScript libraries instead of trying to reinvent the JavaScript object model. Backbone comes with four core components: models, collections, views and routers. The combination of these components make up the core architecture of Backbone but you can extend any of their functionality to meet an application's requirements.

Backbone is very light. The production minified version of the library comes in at just over 6kb. In comparison, AngularJS is over 40kb. Not that a 30kb difference is anywhere near the end of the world or anything, but it does say something for the simplicity of Backbone. However, Backbone requires Underscore.js (5kb) and you should include jQuery (30kb+) to make full use of the History API with Backbone.Router and DOM manipulation in Backbone.View.

Just as we did in the AngularJS section earlier in this chapter, we're going to learn a bit about Backbone by using it to rebuild our application example, Timezone Finder. After we've installed Backbone we're going to introduce each of the core components by adding another piece to the application.

Installation

Just as with any other JavaScript library, it needs to be included on the page. Here is a sample `index.html` that includes a CDN version of Backbone, its hard dependency Underscore.js, and soft dependency jQuery. Let's make this file now in our project root:

```html
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timezone Finder</title>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></\
script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.7.0/unde\
rscore-min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/backbo\
ne-min.js"></script>
    <script src="./app.js"></script>
  </head>
  <body>
    <p>Hello world! Welcome to Timezone Finder.</p>
  </body>
</html>
```

We’ve also included `app.js` which will store our custom scripts. That’s it! Now we can continue to build Timezone Finder again, but this time in Backbone.

Models and Collections

Models in Backbone let you define your data structure and related logic. In our Timezone Finder application, we’re going to display cities and their related time zones. So it makes sense to have a `City` model that stores its name and timezone.

Sets of models are called ‘collections’. So in our example application, we’re going to create a collection that holds a set of cities. Let’s call it `CitiesCollection`.

Let’s open up `app.js` and add the following code.

```
// Define our model and default values.
var City = Backbone.Model.extend({
  defaults: {
    id: '',
    timezone: ''
  }
});

// Define our collection.
var CitiesCollection = Backbone.Collection.extend({
  model: City
});

// Create our collection of cities.
var citiesCollection = new CitiesCollection([
  { id: 'London', timezone: 'GMT' },
  { id: 'Tokyo', timezone: 'JST' },
  { id: 'Melbourne', timezone: 'EDT' },
  { id: 'Los Angeles', timezone: 'PST' },
  { id: 'New York', timezone: 'EST' }
]);
```

Notice the `extend` function; you’ll see this being used a lot in Backbone’s core functionality. In the above code snippet, we’re using it to extend the functionality of the `Backbone.Model` and `Backbone.Collection` components. You can look at `extend` as the Backbone way to define properties for the different components of your application. It’s worth noting that its use is not required. If we had removed the `extend` function from `Backbone.Model` the code would still work but there wouldn’t be any default values when a new `City` is created.

Views

Backbone views give you a place to organize code related to your interface. They let you write the logic behind the presentation of model data but don’t contain HTML markup or CSS. Backbone doesn’t really have controllers from the common MVC pattern but instead views usually contain much of that logic. There are no restrictions on how you render your views. It is recommended to use a JavaScript templating engine. Underscore.js has its own templating engine which is simple, lightweight and perfect for our example application. You can also look into other options such as Mustache or Handlebars. To quote Eleanor Roosevelt, and now the Spiderman movie script, “With great freedom comes great responsibility.” You may find yourself writing a fair amount of boilerplate code for your applications in Backbone which might be seen as a good thing because it gives you full control and understanding of the logic behind your application.

Let’s go ahead and jump back over to Timezone Finder to implement our first view. Open up `index.html` and add the initial markup to the `<body>` tag:

```html
<body>
  <div id="view"></div>
</body>
```

The `#view` element will be responsible for rendering our view. Switch over to `app.js` and append the following script to the file, and then together we’ll go through what we’ve actually done:

```javascript
// This view renders a select element containing the different cities. When a
// city is selected a message outputs the relative timezone.
var SelectCityView = Backbone.View.extend({
  // This is the DOM element the view will be rendered within.
  tagName: 'div',

  // Declare templates required for this view.
  template: _.template(
    '<p>Hello world! Welcome to Timezone Finder.</p>' +
    '<label>Select a city</label>' +
    '<select id="cities"></select>' +
    '<p id="message"></p>'
  ),

  option_template: _.template('<option value="<%= id %>"><%= id %></option>'),

  message_template: _.template('<p>The timezone in <%= id %> is <%= timezone %>.\
</p>'),

  // This is automatically called when the view is created.
  initialize: function() {
    this.render();
    this.renderMessage();
  },

  // Renders our view and the select element.
  render: function() {
    this.$el.html(this.template());
    _.each(this.collection.models, function(item) {
      var option = this.option_template(item.toJSON());
      this.$el.find('#cities').append(option);
    }, this);
    return this;
  },

  // Renders the selected city message.
  renderMessage: function() {
    var message = this.message_template(this.model.toJSON());
    this.$el.find('#message').html(message);
    return this;
  },

  // Listens to the 'change' event on our select element.
  events: {
    "change #cities": "citySelected"
  },

  // When a city is selected we update the message.
  citySelected: function() {
    var id = this.$el.find('#cities').val();
    this.model = this.collection.get(id);
    this.renderMessage();
  }
});

// Wait for the DOM to be ready before creating our view.
$(document).ready(function() {
  var selectCityView = new SelectCityView({
    collection: citiesCollection,
    model: citiesCollection.get('London')
  });
  $('#view').html(selectCityView.el);
});
```

We started by defining `SelectCityView` which contains our view. As we did with model and collection, we `extended` a view to include all of our custom logic. Backbone views are quite simple and come with only a handful of properties.

A view must be associated with a DOM element. The `el` property of a view contains a reference to this element. There are two different ways to initialize `el`. You can create a new element by using a combination of `tagName`, `className` and `id` properties. Alternatively, you can reference an element that already exists in the page. In our snippet above, we specified a `tagName` so that a new `div` element was created for the view.

Using the Underscore.js template function, we have set up a few templates to be used within our views. Templates can contain placeholders and we can pass in a hash containing the values to populate them.

Next, we've got `initialize`. This is an optional property that acts as a constructor for our view. It is run automatically when the view is created so you'll often see it being used to call `render` functions.

The next two properties, `render` and `renderMessage` are functions that output our HTML markup. We've used them to dynamically create the various select `<option>` elements and the selected city message. It's common practice to place `return this` at the end of `render` functions so that you can chain functionality together, e.g. `view.render().renderMessage()`.

The `events` property allows us to bind to various events. In our example, we're listening for the `change` event on our `select` element. For the full catalog of available events, see the official documentation. When a change is detected we call our custom `citySelected` function where we use jQuery to return the selected items value, update the local `model` variable to contain the selected city and then re-render our message by calling `renderMessage()`.

Lastly, we wrapped the actual creation of our view in a `$(document).ready`. We did this because our `<script>` tags are in the page `<head>` and would cause our JavaScript to compile before the DOM is ready. Alternatively, you can place your `<script>` tags just before `</body>` to have them be called later in the page life cycle. As mentioned earlier, the `el` property contains a reference to the DOM element for our view. So we're able to use jQuery to populate `#view` with our views rendered HTML.

Routers

In Backbone, you are able to utilize `Backbone.Router` to connect URLs to actions and events in applications. Most browsers now support the History API but for those that don't, the router gracefully falls back to using hash fragments instead.

For most applications, you'll usually only need to have one router. In our example, let's go ahead and define our router in `app.js`:

```
// Defines our Router which is responsible for handling the switching between
// our various content views.
var AppRouter = Backbone.Router.extend({
  routes: {
    '': 'showWelcome',
    list: 'showList'
  },

  showWelcome: function() {
    var selectCityView = new SelectCityView({
      collection: citiesCollection,
      model: citiesCollection.get('London')
    });
    this.showView(selectCityView);
  },

  showList: function() {
    var cityListView = new CityListView({
      collection: citiesCollection
    });
    this.showView(cityListView);
  },

  showView: function(view) {
    if (this.currentView) {
      this.currentView.remove();
      this.currentView.unbind();
    }

    this.currentView = view;
    $('#view').html(this.currentView.el);
  }
});
```

The `routes` property creates key-value pairings for the various application routes in the format `path: 'callback'`. Our default home route (i.e. defined as `''`) will call `showWelcome` which in turn will render `SelectCityView`. We've also added a `list` route that will render a new view called `CityListView` which we'll create in a moment. To make things a little more simple, we've created a helper function called `showView` that manages which view is currently being displayed. We also have to destroy views when changing between them to prevent memory leaks and binding events multiple times.

Let's create that missing `CityListView` in our `app.js`:

```
// This view renders a list of all cities and their timezones.
var CityListView = Backbone.View.extend({
  tagName: 'div',

  template: _.template('<h1>Full List</h1>' +
    '<div id="list"></div>'
  ),

  message_template: _.template('<p>The timezone in <%= id %> is <%= timezone %>.\
</p>'),

  initialize: function() {
    this.render();
  },

  render: function() {
    this.$el.html(this.template());
    _.each(this.collection.models, function(item) {
      var message = this.message_template(item.toJSON());
      this.$el.find('#list').append(message);
    }, this);
    return this;
  },
});
```

In comparison with `SelectCityView`, this view should be fairly self-explanatory. It renders a list of messages for each of the cities in our collection.

Now that we have a router and both views in code, we're going to need to have a way to switch between the views. Let's add a small menu to our `index.html`:

```
<body>
  <ul id="menu">
    <li><a href="/">Welcome</a></li>
    <li><a href="/list">Full List</a></li>
  </ul>
  <div id="view"></div>
</body>
```

The last piece of the puzzle is to update our `$(document).ready` function:

```
// Wait for the DOM to be ready before creating our view.
$(document).ready(function() {
  Backbone.history.start({
    pushState: true
  });

  // Create our Router and automatically show the welcome view.
  var appRouter = new AppRouter;
  appRouter.showWelcome();

  // Capture menu item clicks to trigger route changes.
  $('#menu a').click(function(e) {
    e.preventDefault();
    appRouter.navigate($(this).attr('href'), {trigger: true});
  });
});
```

We call [Backbone.history.start] to route the initial URL and start listening for route changes. This is required for us to set up routing with `pushState` functionality. Next, we create the router and call `showWelcome` so that view is displayed by default. Lastly, we're listening to clicks in our menu and calling navigate to change the route.

That's it! We've successfully created our application in Backbone. Take it for a spin on your local web server and you should notice very similar results to the application that we created in the AngularJS section earlier in this chapter.

We've only touched the surface of what is possible with Backbone but hopefully the takeaway from your brief encounter is positive. Although there is a learning curve, it's an extremely powerful JavaScript framework that gives you plenty of freedom to scale. If you'd like to learn more about Backbone, Developing Backbone.js Applications by Addy Osmani is strongly recommended. Also, you might be interested in the Backbone Debugger Chrome extension once you get started building Backbone apps.

The final popular framework to introduce to you in this chapter is 'Ember'.

Ember

Ember.js is another open-source JavaScript framework which was originally created by Yehuda Katz and Tom Dale in 2011. Although it's the last framework covered in this chapter, it's by no means the least. With a strong community and a brilliant architectural pattern, Ember is the choice framework for many developers.

Overview

The core of the Ember architecture lies within the URL. Quoting from the Ember core concepts:

> "It's important to remember what makes the web special. Many people think that something is a web application because it uses technologies like HTML, CSS and JavaScript. In reality, these are just implementation details. Instead, the web derives its power from the ability to bookmark and share URLs. URLs are the key feature that give web applications superior shareability and collaboration. Today, most JavaScript frameworks treat the URL as an afterthought, instead of the primary

> reason for the web’s success. Ember.js, therefore, marries the tools and concepts of native GUI frameworks with support for the feature that makes the web so powerful: the URL.”

Because of its strongly opinionated architecture, many people find Ember easier to learn. It may also be more straightforward because of the excellent official documentation.

It is worth noting that Ember has a dependency on Handlebars. So if you were hoping to use another templating language, you may need to look elsewhere. You’ll also need jQuery, but what’s new right?

Just as we did with Angular and Backbone, let’s go ahead and rebuild our Timezone Finder application, but this time in Ember!

Installation

We’ll create `index.html` in our application’s root directory and include the Ember library and its dependencies: jQuery and Handlebars:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timezone Finder</title>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></\
script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/2.0.0/hand\
lebars.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/ember.js/1.9.0/ember.min\
.js"></script>
    <script src="./app.js"></script>
  </head>
  <body>
    <p>Hello world! Welcome to Timezone Finder.</p>
  </body>
</html>
```

For future projects you may be interested in checking out the Starter Kit offered on the Ember homepage which provides you with a barebones starting point for your application. However, for Timezone Finder we’ll just create everything ourselves.

We’ll create an `app.js` file and add the following single line to get started:

```
App = Ember.Application.create();
```

Every Ember application has only one Ember.Application object. So when using Ember, before you do anything else you have to call `create`.

Router

As explained earlier, the URL is a core concept of Ember’s architecture. By default, Ember creates an `index` route which maps to `/` (i.e. the root). So for our application we’ll only need to create one extra route for our Full List page which lists all cities and their timezones. Let’s append the following to our `app.js`:

```
App.Router.map(function() {
  this.route("list");
});
```

Using `App.Router` we are able to map as many routes as we want. We've named our route `list` which means that, by default, its path will be `/list`. Ember also sets up several other components automatically for each route created and these will be introduced as we dive deeper into the application.

Templates

Templates are responsible for presenting the markup for the user interface. Ember templates are exclusively written in Handlebars so you will benefit from spending some time getting familiar with their semantics before building your own real-world application.

Returning to our example, we'll start by adding the navigation menu to the top of the application. We're going to add our templates directly to our `index.html`:

```
<body>
  <script type="text/x-handlebars">
    <ul id="menu">
      <li>{{#link-to 'index'}}Welcome{{/link-to}}</li>
      <li>{{#link-to 'list'}}Full List{{/link-to}}</li>
    </ul>
  </script>
</body>
```

Using the Ember Handlebars link-to helper it's easy to create links to the various routes within an application. In our example we've added a link to the front page route `index` and our custom `list` route. Now let's set up the templates for each of those routes.

When requesting the URL for a route mapped in `App.Router`, Ember will automatically look for a Handlebars template to display for that route. Let's add a template for both `index` and `list` routes, again, in our `index.html`:

```
<script type="text/x-handlebars" id="index">
  <h1>Welcome</h1>
  <p>Hello world! Welcome to Timezone Finder.</p>
</script>

<script type="text/x-handlebars" id="list">
  <h1>Full List</h1>
</script>
```

We've set the `<script>` element's `id` attribute to match the route name. But before this will work, we have to create something called an `outlet`. The outlet helper tells the application where to render the template that was picked for a specific route. Let's add the outlet helper to our `index.html` just below the navigation menu:

```
<script type="text/x-handlebars">
  <ul id="menu">
    <li>{{#link-to 'index'}}Welcome{{/link-to}}</li>
    <li>{{#link-to 'list'}}Full List{{/link-to}}</li>
  </ul>

  {{outlet}}
```

```
</script>
...
```

Now, when accessing the application you'll be able to switch between the two templates using the links in the navigation menu. That was pretty quick and easy right? Let's start adding some logic to our application.

Models

Each template is backed by a model. Models are objects that contain data to be displayed to the user via the template. The most common use of a model is to link it up to an API so that you're getting data directly from a service, but for the purpose of Timezone Finder we're just going to define our model data locally. Let's append a `cities` variable in our `app.js` file so that it can be used throughout our application:

```
var cities = [
  { name: 'London', timezone: 'GMT' },
  { name: 'Tokyo', timezone: 'JST' },
  { name: 'Melbourne', timezone: 'EDT' },
  { name: 'Los Angeles', timezone: 'PST' },
  { name: 'New York', timezone: 'EST' }
];
```

Next, we're going to tell Ember that we want to use `cities` as the model for both our `index` and `list` route templates. The way to do this is to create a `Route` object. That can be a little confusing, but just remember that a `Route` object is not the same as a route defined in your `Router` object. Again, we'll append the following to our `app.js`:

```
App.IndexRoute = Ember.Route.extend({
  model: function() {
    return cities;
  }
});

App.ListRoute = Ember.Route.extend({
  model: function() {
    return cities;
  }
});
```

Now that Ember knows which model to provide our route templates with, we can switch back over to `index.html` and update our Handlebars templates:

```
<script type="text/x-handlebars" id="index">
  <h1>Welcome</h1>
  <p>Hello world! Welcome to Timezone Finder.</p>
  <label>Select a city</label>
  {{view "select" content=model optionValuePath="content.name"
optionLabelPath="\
content.name" selection=selectedCity}}
  <p>The timezone in {{selectedCity.name}} is {{selectedCity.timezone}}.</p>
</script>

<script type="text/x-handlebars" id="list">
  <h1>Full List</h1>
  <ul>
    {{#each}}
      <li>The timezone in {{name}} is {{timezone}}.</li>
    {{/each}}
  </ul>
</script>
```

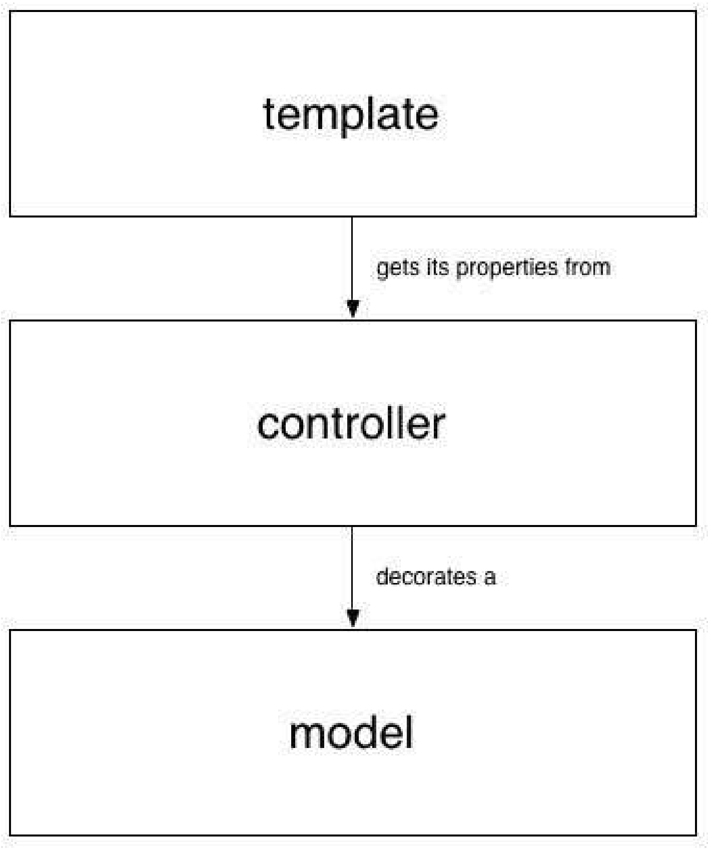
For our `index` route, we're using the Ember.Select view class which renders a `<select>` element for us. We've explicitly set the options to be populated from `model` which is defined in `App.IndexRoute`. As each item in our cities model is an object, we've had to explicitly set the option value and option label. The selected value will be stored in `selectedCity` so that we use it to populate a message below our `<select>` element.

For the `list` route, it is very easy to iterate over the model object and output each city's properties using the Handlebars each helper.

Controller

Something that can take a little while to understand is that although templates get their values from models, a controller lies between them. The controller is responsible for 'decorating' a model with display logic before its values are passed to the template. By default, the controller between a model and a template simply passes through the data. In our example, this is the case with both the `index` and `list` routes. We didn't create any controllers because we didn't need to decorate or alter the data. The important point to remember here is that templates receive all of their data from controllers and not models.

Although we don't need to implement controllers in Timezone Finder, this simple diagram taken from the official documentation sums up the concept quite nicely:



Advanced

There is plenty more sophistication to Ember than was touched on here. As previously mentioned, go and check out their excellent documentation if you're interested in learning more. Just as with Angular and Backbone, there is a popular Chrome extension that you can use to debug your Ember application. Check out Ember Inspector for more details.

Summary

Hopefully this chapter was as enjoyable to read as it was to write! All three of the popular frameworks introduced in this chapter are great fun to work with. Each of them help you to think about developing your application in a smart way through maintainable and scaleable code. We've only looked at three frameworks but there are plenty more out there in the wild that you may come across in your travels. Again, you can find the source code for the various versions of Timezone Finder available here: https://github.com/fender/TimezoneFinder

So the next question you're probably going to ask is, 'which is the best framework for my project?' That's something only you are able to answer really. Each project is different and when picking a framework you need to consider your specific requirements. Here are a few recommended things to think about:

- Which frameworks meets your requirements? For example, if you specifically do not want to use Handlebars, then you can rule out Ember. Or, if you don't want to use jQuery, you may want to consider Angular.
- As with any application you build, you're going to need help. So do some research to see how active and mature the framework is. Are there regular updates? How big is the community? Are there good resources for finding answers to common problems?
- Try it! Often, the best way to find out if something is right for your project is to try it first. You might find one framework compliments the way you think and develop more than another.
- Check out TodoMVC. This is a very useful tool that offers the same to-do application in a large number of different frameworks for you to compare. It was created by Addy Osmani who also wrote a brilliant article on selecting a framework: Journey Through The JavaScript MVC Jungle.

Styling

Styling your markup on the front-end is pretty straightforward on the surface. You have a selector in your DOM and you want it to look a certain way and using the descriptive language of CSS, you tell it so. The challenge many new front-end developers face isn't generally due to the complexity of CSS itself, it's the *nature* of CSS that can make it so challenging.

The C in CSS stands for 'cascading', meaning that styles applied to one element are inherited by other elements further down the DOM tree. If you apply a `font-size` to your document's `<body>` and then look at the `font-size` of some `<li>` further down the DOM, you'll notice they're the same. The `font-size` was inherited. This cascading nature is both the best and most frustrating feature of CSS.

Many times, you'll want to style something like an item list, `<ul>`, the same way, no matter where it appears in the DOM. Easy! But when the design you're trying to implement has different variations of an element based on where it appears, you may start running into specificity problems.

Specificity in CSS is actually an intended feature of the language. For example, if we wanted our all of our item list elements in a footer to have a few pixels of margin on the left side, we simply need to be more specific about our selector. What started out as:

```css
ul { margin-left: 10px; }
```

Would become:

```css
.footer ul { margin-left: 10px; }
```

Now, if we want all of our item lists in a different parent element to have no margin at all then we can do the same thing:

```css
.block ul { margin-left: 0; }
```

Keeping those two CSS selectors in mind, what happens when we have the following markup?

```html
<body>
  <div class="block">
    <h1>Hello World!</h1>
  </div>
  <footer class="footer">
    <div class="block">
      <ul>
        <li>List item</li>
        <li>List item</li>
      </ul>
    </div>
  </footer>
</body>
```

In the example above, the `<ul>` is styled entirely by the cascade. If we declared `.footer ul` last, the `<ul>` has margin, if we declared `.block ul` last, the `<ul>` has no margin. The reason for this is that both selectors have the exact same specificity, and CSS falls back to the cascade applying the last thing declared that matches our selectors. To further complicate the issue, different types of selectors have different levels of specificity. For instance, the three most common selector types follow this order of specificity:

1. element (least specific)
2. class
3. id (most specific)

In our example above, `div ul` would always be overridden by `.footer ul` because the class is more specific than the element, even though `div ul` has a match.

So if we're aware of how all this works, why is CSS such a trouble spot in front-end development? Usually the answer to that comes from a couple different sources.

Firstly, the larger a page is the more opportunity for selector conflicts there are. It's tough to know just where everything on a page *might* appear, so when we're faced with the need to have an instance of a selector styled just so, the first thing many front-end developers do is get more specific and add a new style declaration. If that gets overridden by something even more specific or lower in the cascade, the easiest option is to throw `!important` on the style that seems too difficult to override. That causes frustration down the road when a change comes in that needs to affect something, including the thing that was overwritten. Largely, this problem can be solved with proper planning (if you have that luxury) or by choosing an authoring methodology to minimize future frustration (more on that later).

Secondly, the more people who are in your codebase writing styles, the more opportunities for conflict and duplication there are. More people, more problems. Johnny might be able to make the styling adjustments he needs one way without ever knowing that he's totally mangled the styles for something Carl wrote earlier. This problem boils down to communication. Like the first issue, this can be somewhat solved by planning or choosing a methodology like BEM, but documentation, and even writing *tests* for your CSS which run automatically, can be equally as useful.

Responsive Web Design

In the last couple of years, there has been an explosion in the popularity of this term. So what does it mean? If you're new to the world of front-end development, you may not have a fully grasped it. In short, Responsive Web Design (RWD) is the concept that page designs should be flexible and should accommodate any screen size. Whether it's your mobile phone or your TV, the page should adapt and the content should be readily viewable and accessible.

The task of actually figuring out how a layout should adjust and change to accommodate various screens is often a designer's job, but the task of making those flexible layouts a reality falls to a front-end developer.

Responsive web design is a humongous topic and there are lots of tips and tricks and methods for making it all happen. So let's go over some of the basic things you'll probably want to understand to get started.

Media Queries

A media query is a statement that limits the scope of your styles to a particular media type (like `screen` or `print`) and/or a media feature (like `min-width` or `height`). This is really important to understand when you're getting into the world of RWD because they give you the power to change your styles based on certain conditions.

A media query can be applied as an attribute on the `<link>` element that imports your stylesheet:

```
<link rel="stylesheet" href="/path/to/your/style.css" media="(min-width: 400px)">
```

Using this method, your `style.css` stylesheet will only be used when the width of the page is `400px` or greater.

You can also use media queries directly in your stylesheets as wrappers around a block of selectors:

```
.foo {
  color: blue;
}

@media (min-width: 400px) {
  .foo {
    color: green;
  }
}
```

In the above statement, `.foo` will be colored blue until the page width is `400px` or greater. Once the page width hits `400px`, the color of `.foo` will change to green.

Media queries can be chained together using `and` to increase specificity. This next media query makes sure that `style.css` is used only when the page is being viewed on a `screen` and the width is between `400px` and `601px`:

```
<link rel="stylesheet" href="/path/to/your/style.css" media="screen and (min-wid\
th: 400px) and (max-width: 600px)">
```

That same media query written in the stylesheet would look like this:

```
@media screen and (min-width: 400px) and (max-width: 600px) {
  /* your styles */
}
```

You can also separate media queries by using a comma to represent 'or'. This media query will apply `style.css` when the orientation is landscape *or* the page is at least `400px` wide:

```
<link rel="stylesheet" href="/path/to/your/style.css" media="(orientation: lands\
cape), (min-width: 400px)">
```

You can find a detailed account of media query features and types at the Mozilla Developer Network.

Relative Units

A good deal of the visual flexibility of elements on a page comes from defining their dimensions not with pixels, but with relative units. Haven't heard of relative units? Fear not, here's a quick overview of the units you'll see used most often.

`em` and `rem` Units

The `em` unit represents the value of the font size of the current element's parent. Take a look at the following example:

```css
.form {
  font-size: 16px;
}

.form input {
  width: 2em;
}
```

What would the width of `.form input` be? That's right, `32px`, or 2x the font size of `.form`. `em` units are scalable, so when a user zooms in with their browser anything styled with an `em` unit will scale proportionally.

The `rem` unit is a bit newer, and represents the *root* font-size of the document, which is the font size set on `<html>`. Why would you want such a unit? Well, a problem that often crops up when using `em` units is a sort of nesting mess. Here is another example to demonstrate:

```css
html {
  font-size: 16px;
}

.form input {
  width: 2em;
}

.form input span {
  width: 2em;
}
```

You may think that the width of `.form input span` would also be `32px` but actually it is `64px`. However, if we switched out `em` for `rem` units then both widths would indeed be `32px`. Using an `rem` unit can keep text scalable and provide a sane base for nested `em`s.

`%` Units

Using a `%` value works in the way you'd expect. If you have an element's width set to `100%` and its parent width is `200px`, your element's width is also going to be `200px`. This is a pretty common way to define dimensions in the RWD world. If you define the majority of your dimensions with percentages rather than absolute pixel values, you'll be well on your way to creating flexible pages.

`vw` and `vh` Units

These two units are also relatively new in the front-end space and are very handy indeed. They function somewhat similarly to the `%` unit, but instead of being based on their parents they are based on the 'viewport' (the dimensions of the browser window). `1vw` is equal to 1/100th of the current viewport width and `1vh` is equal to 1/100th of the current viewport height. If you wanted to have an element be exactly the height of the viewport, you could set its height to `100vh` which equates to 100% of the viewport height.

There are a handful of other really interesting relative units out there, but you'll typically deal with the ones presented above in our day to day work. If you're interested in more details about all the relative units and their support in various browsers, check out the Mozilla Developer Network.

Summary

Building front-end applications with RWD principals can get pretty complicated. Wrapping your head around *em* and *%* units can be tricky, and getting everything on the page to look right all the time can be a painstaking process. To be honest, it was a bit of a struggle to figure out what, if anything, should be said about responsive web design in this book. The topic is massive, heavily debated, and constantly evolving. But something needed to be said to bring newer front-end developers up to speed on *what* it is if not exactly *how* to implement it. That topic is worthy of an entire book itself and there just so happens to be a fantastic one written by Ethan Marcotte simply titled *Responsive Web Design*. You can find this seminal work at A Book Apart in both paperback and ebook forms.

Authoring Methodologies

A 'methodology' is a group of procedures for accomplishing something and, particularly if you're working with a team, something that is incredibly helpful when authoring CSS for your project. Choosing or even crafting your own methodology is a perfect way to keep people from stepping on each other's toes in the codebase (mostly) and it takes some of the "How the hell do I do this?" thinking out of the equation as well.

So what exactly do we mean when we talk about 'authoring methodologies' for CSS? Generally, we mean a documented strategy for how our code will be organized and how we'll approach styling something. Will we use ID selectors for every major section? Should we totally ignore the cascade to avoid specificity problems? How should we name our classes? These are the kinds of things choosing a methodology is all about.

Unfortunately, nobody can just tell you which methodology is the best. There's no right answer. Every single project will likely have different needs and those needs have to be analyzed before you can begin to think of how to approach meeting them. Moreover, there is no one single methodology that can ever be perfectly adhered to. There will always be exceptions to whatever you decide to do but, if you document those exceptions as you come across them, their impact on the whole of the project is usually pretty minimal.

Below are two common methodologies, grouped by how the authors perceive the spirit of those methodologies.

Components and Patterns

A component based methodology centers around the idea that certain individual chunks of styles can be group together to form a 'component'. These components should typically be presented the exact same way no matter where they appear in the DOM. A good example of this might be a small upcoming events calendar which should always look the same, whether in the footer or a sidebar. Let’s write it up:

Markup

```html
<div class="calendar-component">
  <ul>
    <li>Jan <span>01</span></li>
    <li>Feb <span>29</span></li>
  </ul>
  <ul>
    <li>New Year's!</li>
    <li>Leap Day!</li>
  </ul>
</div>
```

CSS

```css
.calendar-component {
  width: 400px;
}
.calendar-component ul {
  width: 60%;
  display: inline-block;
  padding: 0;
}
.calendar-component ul:first-child {
  width: 30%;
}
.calendar-component li {
  list-style: none;
}
.calendar-component span {
  font-weight: bold;
}
```

That isn’t exactly the most beautiful component on the web, but it’s completely self contained. Nothing we have written above will affect anything that isn’t directly related to our calendar component. Our component may still be affected by more generic styles (like our `<ul>` declarations earlier in this chapter), but if we separate the various parts of our designs into these unique components right from the beginning of a project, then we can prevent that from happening.

Taking this a step further, let’s now imagine we have an identically styled side-by-side list component. It contains similarly formatted content, but instead of a calendar it’s a T-shirt sizing chart. The simplest option would be to duplicate our markup for that component, classes and all, with the different content. Blamo! We’ve just created a re-usable CSS pattern.

Now, `calendar-component` isn’t exactly the most generic name to be applying to a T-shirt sizing chart, so for developer sanity it’s recommended that you try to generalize component names so they can be re-used and understood more easily. Reducing duplicated CSS is pretty important due to the nature of the cascade and the tendency for styles to show up in odd places. If your project happens to support Internet Explorer 6-9 then duplication can be even more important to avoid because IE has a 4095 total selector limit - and then it *just stops reading them*.

BEM

BEM is a popular component writing methodology developed by Yandex that is, in many ways, similar to the object oriented programming methodology used in other languages. BEM stands for ‘block’, ‘element’ and ‘modifier’; these are the categories your CSS selectors will fall into. Blocks are the largest chunk of styles that make up a particular visual element on a page. Elements, in regards to BEM, are sub-sections of a particular block, and modifiers, as you might guess, describe any variations of a BEM block or element.

The BEM methodology itself isn’t limited to CSS since it’s also applicable for XML, JSON, and probably other things, but we’re mostly concerned with the CSS usage.

Our calendar component, described above, would equate to what BEM calls a block while the inner HTML would be our element.

The typical pattern for naming classes using BEM is to include our block name in every selector and name each element of the block using __ to separate them. Modifiers are denoted by `--`.

Using BEM our CSS might look something like this:

```css
.calendar-component {
  width: 400px;
}
.calendar-component__list {
  width: 60%;
  display: inline-block;
  padding: 0;
}
.calendar-component__list:first-child {
  width: 30%;
}
.calendar-component__list-item {
  list-style: none;
}
.calendar-component__day {
  font-weight: bold;
}
```

When comparing the visual display of the block to our CSS selectors, it is all perfectly clear at a glance what styles are affecting what pieces.

If we look at that third selector `.calendar-component__list:first-child` you might notice that it’s the perfect opportunity to use what BEM calls a modifier since all it’s really doing is modifying an otherwise perfectly normal `.calendar-component__list`.

BEM convention would be to create a different class and apply it to the the element that varies from its companions. Our HTML could change to something like this; let’s call our variation of the `.calendar-component__list .calendar-component__list--smaller` since the variation is defining a smaller width:

```html
<div class="calendar-component">
  <ul class="calendar-component__list calendar-component__list--smaller">
    <li class="calendar-component__list-item">Jan <span class="calendar-componen\
t__day">01</span></li>
    <li class="calendar-component__list-item">Feb <span class="calendar-componen\
t__day">29</span></li>
  </ul>
  <ul class="calendar-component__list">
    <li class="calendar-component__list-item">New Year's!</li>
    <li class="calendar-component__list-item">Leap Day!</li>
  </ul>
</div>
```

Now, if we swap `.calendar-component__list:first-child` with our new modifier class `.calendar-component__list--smaller` we'll be in business!

So why would you want to do all this? Well, there are probably a lot of opinions out there regarding what makes this so useful, but a few of the more immediately digestible reasons are:

1. BEM makes scanning CSS documents easier
2. BEM avoids deep selector and inheritance hell
3. BEM makes it simple to look at a visual element and figure out exactly which CSS selector is styling it

An interesting thing about writing your CSS entirely with BEM is that you're essentially removing the C from CSS and ignoring the cascading feature. This means you end up writing a lot more classes, but you also avoid some of the trouble that the cascade can cause.

If you want to read more about BEM than this short introduction can provide, check out the official website.

Utility

A utility based methodology centers around the idea that a single class should do very few things and that any number of CSS classes can be applied to an element to obtain the desired output. Using a utility class paradigm is actually fairly straightforward, although organization can become quite a hassle due to the sheer number of utility classes that get crafted throughout the course of a site build.

Before we talk about why this might be useful, let's take a look at some utility class examples:

```
.small-text {
  font-size: 11px;
}

.underlined {
  text-decoration: underline;
}

.black-bg {
  background-color: #000000;
}

.float-left {
  float: left;
}
```

These are all fairly standard examples of utility classes. Each class presented above does only one thing, although it is perfectly acceptable for them to do more, so if you wanted some paragraph of text to be 11px and underlined, you'd need to apply both `.small-text` and `.underlined` to the markup:

```
<p class="small-text underlined">Lorem ipsum</p>
```

If this sounds insane, nobody can blame you. The usefulness of this methodology is tough to see at first, so let's talk about why we'd go this route.

Firstly, it's very quick to style something new if you've got an army of utility classes already set up in your stylesheet. If you need your text to be red, by God, just throw a `.red-text` class on it.

Secondly, using utility classes means there is very little repetition in your stylesheets which means a smaller overall file size which, in turn, means a faster load time for your stylesheet. With a component based methodology you may find yourself repeating `float: left;` on lots of elements, but using utility classes you may only have one `float: left;` in your entire document.

Using utility classes exclusively brings its own set of challenges to a project however, as it becomes the responsibility of HTML authors to style their content as well as create it. The benefits of ease and reduced repetition in your stylesheets might be negligible when compared with the trouble of creating new content using utility classes. If your workflow makes it easier to modify HTML classes than update stylesheets, utility classes might be for you.

Mixing and Matching

Probably the most useful way to author your styles is to take a bit from both component and utility methodologies.

Building your visual patterns as components is one of the best ways to make sure your styles scale well. It takes a lot of thought out of the authoring process by providing what are, essentially, templates of style. By creating one-off utility classes that don't really need to be embedded into a particular component, you can ease some of the pain of modifying a component as needed.

Let's jump back to our calendar component example and introduce a new scenario that mixes the two methodologies.

Let's say that on our site we have multiple instances of our calendar component sprinkled in various areas of the layout. One in a sidebar, one in the masthead and another in the footer (we REALLY want people to know about our upcoming events okay?). However, when a user visits our site on a browser less than `400px` wide we want to hide the component in the footer without actually affecting the other instances on the page. This is the perfect spot for a utility class because we have a scenario in mind that could be applied to any component. That scenario is "hide this thing when the site width is less than `400px`."

Our utility class might look like this:

```
.hide-under-400 {
  display: none;
}
@media only screen and (min-width: 400px) {
  .hide-under-400 {
    display: block;
  }
}
```

Now we can apply `.hide-under-400` to our component, or any other component, and have it disappear from view whenever our browser window is less than `400px` wide.

Preprocessors

A preprocessor is a program that takes some kind of input and spits out another kind. In the world of front-end development, we have a number of these programs at our disposal which makes writing CSS (or JavaScript and HTML) a more pleasant experience. The basic idea is that preprocessors let you write your styles using a different syntax which then spits out into plain old CSS which your browser can read.

The main benefit of using preprocessors is that most come with features not normally found in CSS (at the time of writing, anyway). The features themselves are a little different depending on which preprocessor we're looking at, but most come with a way to write variables for storing values and nesting support so, among other things, you don't need to write lengthy selectors.

Sass (Syntactically Awesome Style Sheets)

At the moment, the Sass language is widely used in the front-end community. Sass sports a number of fantastic features and has two syntaxes to choose from; the original indented syntax Sass and the more recent SCSS syntax which use the `.sass` and `.scss` file extensions respectively. The original version of Sass is built on Ruby although other options are available including PHP, C, and Java. Sass even has a scripting language called SassScript under the hood so if you're comfortable with such things, you could even write your own features.

Installing

To get started using Sass, you'll need to install the program on your computer or use a third-party application that contains it such as CodeKit or Scout. In this section we'll cover the typical way to install it with Ruby, but for a more thorough overview of all the options out there, visit Sass' installation page.

This section also assumes that you have Ruby already installed on your machine. Installing Ruby is outside the scope of this book, but if you're using a Mac the good news is that you've already got it. Windows and Linux machines have their own installation processes.

Once you've got Ruby on your system, simply head to your command prompt (Terminal in OSX, cmd in Windows) and install the Sass gem:

```
gem install sass
```

That's all there is to it! You can make sure everything is installed properly by running `sass -v`. If you get output like so, then you're all set:

```
sass -v
Sass 3.4.9 (Selective Steve)
```

Using Sass

Now that you've got Sass installed, you'll find it's fairly simple to use. The pattern for using Sass from the command line is easy:

```
sass [options] [input] [output]
```

So if you've got a `style.sass` file in your current directory you could run:

```
sass style.sass style.css
```

to have Sass compile your `style.sass` stylesheet into `style.css`.

Using the SCSS syntax is a little trickier, and you'll need to pass an option to the command to tell it that you're using SCSS instead of Sass. Assuming you have `style.scss` instead of `style.sass`, that command would look like this:

```
sass --scss style.scss style.css
```

You can watch a file or an entire directory for changes to your `.sass` files and have the program automatically compile CSS any time it detects changes using the `--watch` option. This is really quite handy if you're working on a project and don't want to keep running the command manually to see the changes you're making.

To watch a single file:

```
sass --scss --watch style.scss:style.css
```

To watch an entire directory:

```
sass --scss --watch your_scss_directory:your_compiled_css_directory
```

The `sass` command has a number of other useful options available to customize the output of your compiled CSS. You can see the full list by running:

```
sass -h
```

The Sass Syntax

The original Sass syntax is not directly compatible with CSS. That means you can't just change your file extensions to `.sass` and expect things to run. Sass uses indentation to separate property declarations from selectors, and new lines instead of semicolons to denote the termination of a property. As a quick example, let's compare the same style declaration in CSS and Sass:

CSS

```
.class-name {
  color: purple;
  background: blue;
}
```

Sass

```
.class-name
  color: purple
  background: blue
```

This can be a little confusing to actually use at first, but if you're starting a new project you may find it preferable to use the Sass syntax because you actually end up writing fewer overall characters.

The SCSS syntax

Unlike the Sass syntax, the SCSS syntax **is** compatible with CSS. You can copy and paste any generic CSS into a `.scss` file and have absolutely no issues, although of course it doesn't work the other way around. The SCSS syntax, for the most part, uses the exact same syntax that CSS does. You group sets of properties on a selector with braces and you terminate property declarations with a semicolon. Indention doesn't matter at all, although you should be consistent for your own mental well being.

Many people prefer the SCSS syntax because it's so compatible with existing CSS. If you've already got a well established project and you'd like to start switching to Sass, using the SCSS syntax is an easy way to get started.

Variables in Sass

Variables in Sass are denoted by a `$` which should feel pretty familiar if you've ever used a language like PHP. Both the SCSS and Sass syntaxes use the same format for declaring a variable, although in the Sass syntax you won't need to use a semicolon at the end:

Sass syntax

```
$my-blue-variable: #0000ff
header h2
  color: $my-blue-variable
```

SCSS syntax

```
$my-blue-variable: #0000ff;
header h2 {
  color: $my-blue-variable;
}
```

CSS output

```
header h2 {
  color: #0000ff;
}
```

Nesting Selectors in Sass

Sass has the concept of nesting selectors within one another to avoid having to write really long strings of selectors to get the specificity you want. The easiest way to understand this concept is to visualize it. Let's say we want to target this specific CSS selector: `#main article .author`. To do that we can write our styles for `.author` inside the styles for `article` and write those inside `#main`. To do that with Sass, we'd write:

Sass syntax

```
#main
  background: #ffffff

  article
    margin: 0 auto
    color: #000000

    .author
      background: #cecece
      padding: 10px
```

SCSS syntax

```
#main {
  background #ffffff;

  article {
    margin: 0 auto;
    color: #000000;

    .author {
      background: #cecece;
      padding: 10px;
    }

  }

}
```

CSS Output

```
#main {
  background: #ffffff;
}
#main article {
  margin: 0 auto;
  color: #000000;
}
#main article .author {
  background: #cecece;
  padding: 10px;
}
```

Selecting parents with `&`

Another handy feature of nesting that Sass provides is a way to reference the parent selector using `&`. This is particularly handy when you want a nested selector to have a different style when its parent is slightly different, for instance, when the parent has an extra class or state. An easy example of this parent selector usage is styling the various states of a link:

Sass syntax

```
a
  color: blue

  &:hover
    color: green

  .some-class &
    color: purple
```

SCSS syntax

```
a {
  color: blue;

  &:hover {
    color: green;
  }

  .some-class & {
    color: purple;
  }

}
```

CSS Output

```
a {
  color: blue;
}
a:hover {
  color: green;
}
.some-class a {
  color: purple;
}
```

Another interesting feature of the `&` selector is its ability to create combinatorial explosions of selectors using `& + &` within a nested list of selectors. This will generate every possible permutation of the parent selectors.

Note: The + operator in CSS denotes sibling elements.

Sass syntax

```
ul, quote, p

  & + &
    margin: 0
```

SCSS syntax

```
ul, quote, p {

  & + & {
    margin: 0;
  }

}
```

CSS Output

```
ul + ul,
quote + ul,
p + ul,
ul + quote,
quote + quote,
p + quote,
ul + p,
quote + p,
p + p {
  margin: 0;
}
```

Mixins in Sass

Mixins are a really handy way to group sets of properties together so you can reuse them. A common practice is to use mixins for writing vendor prefixes (e.g. `-webkit-border-radius`, `-moz-border-radius` etc), but you can use them to group any kind of properties you want.

To define a mixin, declare `@mixin your_mixin_name` in SCSS and `=your_mixin_name` in the Sass syntax. Let’s create a mixin that adds a dashed border and some padding around an element:

Sass syntax

```
=dashed_border
  border: 1px dashed #000000
  padding: 10px
```

SCSS syntax

```
@mixin dashed_border {
  border: 1px dashed #000000;
  padding: 10px;
}
```

Now that the mixin is defined, we can use it on any selector we want by using the mixin’s name, prefixed with + in Sass and `@include` in SCSS:

Sass syntax

```
.foo
  +dashed_border
```

SCSS syntax

```
.foo {
  @include dashed_border;
}
```

CSS Output

```
.foo {
  border: 1px dashed #000000;
  padding: 10px;
}
```

Mixins can also take arguments! Let’s say we want to give the option to change the border color and the amount of padding wherever we use the mixin. To do that, we’ll need to change our mixin definitions a little bit:

Sass syntax

```
=dashed_border($border_color, $padding)
  border: 1px dashed $border_color
  padding: $padding
```

SCSS syntax

```
@mixin dashed_border($border_color, $padding) {
  border: 1px dashed $border_color;
  padding: $padding;
}
```

Now we would simply pass in a color as the first parameter and a padding value as the second:

Sass syntax

```
.foo
  +dashed_border(#dddddd, 4px)
```

SCSS syntax

```
.foo {
  @include dashed_border(#efefef, .5em);
}
```

Additionally, mixins may be defined with default argument values. A default value for an argument will be used when no argument is passed when the mixin is referenced in a selector. Let's reuse our dashed border mixin and this time define a default value for the `$border_color` argument. To do this, we'll need to change the order of our arguments in our mixin's definition because arguments with default values *must* come after arguments without:

Sass syntax

```
=dashed_border($padding, $border_color: #000000)
  border: 1px dashed $border_color
  padding: $padding
```

SCSS syntax

```
@mixin dashed_border($padding, $border_color: #000000) {
  border: 1px dashed $border_color;
  padding: $padding;
}

.foo {
  // Without a second argument, the default border color will be used.
  @include dashed_border(10px);
}

.bar {
  @include dashed_border(10px, green);
}
```

CSS Output

```
.foo {
  border: 1px dashed #000000;
  padding: 10px;
}

.bar {
  border: 1px dashed green;
  padding: 10px;
}
```

Sass mixins can also be called by using argument keywords like so: `@include dashed_border($padding: 10px, $border_color: green);`. To illustrate the usefulness of this feature, let's add a third argument to `dashed_border` called `$border_width` and give it a default value:

Sass syntax

```
=dashed_border($padding, $border_color: #000000, $border_width: 1px)
  border: $border_width dashed $border_color
  padding: $padding
```

SCSS syntax

```
@mixin dashed_border($padding, $border_color: #000000, $border_width: 1px) {
  border: $border_width dashed $border_color;
  padding: $padding;
}
```

Now suppose that in our selector `.foo`, we want to add a dashed border with a width of 5px but leave the default `$border_color` value alone. Since `$border_width` is the third argument, we cannot simply pass a value to it without also passing a value to `$border_color` unless we use the keyword functionality:

Sass syntax

```
.foo
  +dashed_border(10px, $border_width: 5px)
```

SCSS syntax

```
.foo {
  @include dashed_border(10px, $border_width: 5px);
}
```

CSS Output

```
.foo {
  border: 5px dashed #000000;
}
```

Using an argument's keyword in the reference to the mixin, you can set that argument's value no matter what position it's defined in. `@include dashed_border($border_color: green, $padding: 4px, $border_width: 8px)` would compile just fine since we're using keywords, even though we're declaring argument values in a different order than they were defined in the mixin.

Sass mixins are very powerful and there's a lot more you can accomplish with them than what's been shown here. For a complete overview of mixins in Sass, take a look at the mixin documentation. Here you'll find a lot more information about mixin arguments, like how to use variables as arguments, how to pass content blocks to a mixin, and variable scope within mixins among other handy nuggets of information.

Control Expressions

To add to the potential complexity of the language, Sass supports the usage of control expressions such as `if`, `for`, `each`, and `while`. For more information on this subject, take a look at Sass' documentation. Control expressions are not limited to usage in mixins, but this is the most common place you'll find them in the wild.

Extending

The concept of 'extending' a selector in Sass is somewhat similar to mixins. You can take all of the properties applied to one selector and extend them onto another one.

The actual usage of the extend functionality is the same in both Sass syntax and SCSS. Simply write `@extend` before the name of the selector you want to extend in the property list of your selector:

Sass syntax

```
.div_1
  color: blue

.div_2
  @extend .div_1
```

SCSS syntax

```
.div_1 {
  color: blue;
}

.div_2 {
  @extend .div_1;
}
```

The main difference between extending a selector and using a mixin is that if you use a mixin multiple times on a multitude of different selectors, your CSS output will have the same content repeated multiple times. Using our mixin example from above, that might look like:

Sass syntax

```
.div_1
  +dashed_border

.div_2
  +dashed_border
```

SCSS syntax

```
.div_1 {
  @include dashed_border();
}

.div_2 {
  @include dashed_border();
}
```

CSS Output

```
.div_1 {
  border: 1px dashed #000000;
  padding: 10px;
}

.div_2 {
  border: 1px dashed #000000;
  padding: 10px;
}
```

By extending a selector onto another one, the two selectors will be combined and only the differences will be split out into their own selectors:

Sass syntax

```
.div_1
  border: 1px dashed #000000
  padding: 10px

.div_2
  color: blue
  @extend .div_1
```

SCSS syntax

```scss
.div_1 {
  border: 1px dashed #000000;
  padding: 10px;
}

.div_2 {
  color: blue;
  @extend .div_1;
}
```

CSS Output

```css
.div_1, .div_2 {
  border: 1px dashed #000000;
  padding: 10px;
}

.div_2 {
  color: blue;
}
```

Operators

The Sass language also supports some mathematical operators that you can use when writing your styles. The supported number operators are:

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication
- `/` - Division
- `%` - Modulo

In addition to these mathematical operators, Sass includes relational and equality operators, typically for use in control expressions:

- `<` - Less than
- `>` - Greater than
- `<=` - Less than or equal to
- `>=` - Greater than or equal to
- `==` - Equal to
- `!=` - Not equal to

Partials and Importing

A common way to organize your CSS is to split out categories or specific chunks of styles into their own files. Using a component based methodology, each individual component might have its own file so that developers know exactly where to find a particular component's styles.

Out of the box CSS supports the `@import` directive which is used for including other CSS files into a CSS file. This is great for organization, but not so great for the overall performance of your page. Each usage of `@import` creates an HTTP request which will slow down the rendering of your page.

With Sass, you can create 'partials' which are essentially the same as siloed CSS files and they can be included in another Sass file in the same way that CSS can `@import` other CSS files. The difference is, when Sass comes across an `@import` directive during compile time, it concatenates the files together creating a single CSS stylesheet. In this way, you can have all the nice separation of styles without creating excess HTTP requests.

Here's an example of how to use a partial file within Sass:

Sass syntax

```
/* main.sass */
@import partial

.main_class
  background: #efefef

/* _partial.sass */
.partial_class
  color: purple
```

SCSS syntax

```
/* main.scss */
@import 'partial';

.main_class {
  background: #efefef;
}

/* _partial.scss */
.partial_class {
  color: purple;
}
```

CSS Output

```
/* main.css */
.partial_class {
  color: purple;
}
.main_class {
  background: #efefef;
}
```

The name we gave the partial file in this example, `_partial.sass` is prefixed with an underscore so that Sass doesn't create a `partial.css` file when we compile the directory, which it will do if we leave off the underscore. There is no requirement that partial files be named this way and importing a partial using the name of the file without the extension will work whether or not you've prefixed the file name with an underscore.

It is also important to note that the contents of our partial will appear in the final output in the place they were imported within the main file. In our example above, we declared `@import partial` above our declaration of `.main_class`, so the contents of our partial, `.partial_class` were compiled out above `.main_class`.

Other Sass Tools

The popularity of Sass has given rise to a number of authoring frameworks and libraries built specifically for the language. The largest players in this space at the time of writing are probably Compass and Bourbon. Each of these provides a good number of built-in mixins and functionality that you might otherwise create by hand for your projects, including layout helpers, typography utilities and CSS3 shortcuts.

The benefits of using libraries like these are many, but depending on the scope and size of your project they could very well be overkill. Tying your Sass project to a single framework or library such as these can be a limiting factor when Sass itself gets updated and you want to update to the latest version since your library will likely require a specific version of Sass to run.

Overall, tools like Compass are incredibly useful, particularly for beginners.

Summary

In this book, we're only scratching the surface of what Sass is capable of. There are other powerful features in this language that are honestly best described within Sass' own documentation and giving such a cursory overview as has been given the rest of the features presented in this book would likely be a disservice to readers. More detailed information on features we've covered as well as those we haven't covered such as lists, maps, placeholder classes, and defining custom Sass functions among others can be found here.

Less

Less.js is a JavaScript based preprocessor that at one point was king of the veritable CSS preprocessor hill. More recently, it seems to have taken somewhat of a backseat to Sass in overall popularity, but it is still widely used. Less and Sass have a very similar feature set.

Like Sass, Less is a very powerful language and just like the SCSS syntax os Sass, Less' syntax is backwards compatible with vanilla CSS making it relatively easy to pick up and use.

Less can be used via the command line, programmatically via JavaScript or on the fly via the browser by including the less.js script on your page. The latter method is not recommended due to the sheer size of the script and the fact that some browsers may compile Less into readable CSS more slowly than others. In this book, we'll concern ourselves only with the command line usage.

Installation

To install Less locally for use on the command line (the preferred method), you'll need to have the `npm` package manager installed. The installation process for `npm` can be found in the Dependency Management chapter of this book.

Once you have `npm`, installation is a breeze. Simply head to your command line and run:

```
npm install -g less
```

Many *nix systems already come with a program called `less`, so what our package manager is actually installing here is a program called `lessc` which stands for “less compiler.” To check whether or not the installation was successful, run the following and look for similar output:

```
lessc --version
lessc 2.1.2 (Less Compiler) [JavaScript]
```

Using Less

To have Less compile our code we can simply run the `lessc` program on our Less file (the extension for Less files is `.less`):

```
lessc style.less
```

This will print out the compiled result to your terminal screen as a preview. If you actually want to save your compiled styles you’ll need to send that output into its own file:

```
lessc style.less > style.css
```

The `lessc` program has a ton of options you can use to tweak the output of your compiled Less. To see them, run:

```
lessc --help
```

Variables in Less

Variables in Less are defined by prefixing a variable name with `@`:

Less

```
@a-less-variable: 10px;

.foo {
  width: @a-less-variable;
}
```

CSS Output

```
.foo {
  width: 10px;
}
```

Less variables do not need to be defined before they are used in a document. In the above example, `@a-less-variable` could have been defined below `.foo` without any issue.

An interesting, although maybe not totally intended feature of Less’ variables, is that they are *scoped*, meaning that the compiler will search for a variable definition within the local scope of a selector before moving up to its parent. In the case of nested selectors, this means moving up to the parent selector. In the case of selectors with no parent, Less will look in the *global* scope.

Here’s an example:

```
@dark-color: #000000;

.foo {
  @dark-color: #777777;
  color: @dark-color;
}
```

Above, we defined `@dark-color` twice. When the compiler hit `color: @dark-color` it first looked within that selector for a definition of `@dark-color` and on finding one, applied it to the property that referenced it. Therefore, the color used is `#777777`.

If `@dark-color` had not been defined in this *local* scope, the compiler would have moved up a level, to the *global* scope to search for a definition of `@dark-color`.

This concept of variable scope is very important to get a handle on when using Less since the compiler will use the last definition of a variable in the current scope.

In the following example, we'll define `@dark-color` a number of times and see what happens to the output:

```
@dark-color: #000000;

.foo {
  color: @dark-color;
}

@dark-color: #777777;

.bar {
  color: @dark-color;
}

@dark-color: #333333;
```

In the above example, both `.foo` and `.bar` will be output with a color of `#333333` since that was the last declaration of `@dark-color` in the current scope. Note that this is not something one would actually write and is simply being used to illustrate the fact that the Less reads variables bottom-to-top in the current scope.

Variables in Less are *constants* and generally should not be defined more than once.

Nesting Selectors in Less

Nesting in Less functions the same way as nesting using Sass' SCSS syntax, including the usage of `&` as a parent selector:

Less

```
.foo {
  color: green;

  .bar & {
    color: black;
  }

  .baz {
    color: purple;
  }

}
```

CSS Output

```
.foo {
  color: green;
}

.bar .foo {
  color: black;
}

.foo .baz {
  color: purple;
}
```

Mixins

Mixins in Less are a little different than mixins in Sass. In fact, they're a bit closer to the way Sass' extend functionality works. In Less, mixins are essentially class or ID selectors themselves which can be extended onto other classes. To mix in an ID or class' properties in a different selector, simply write the mixin selector's name optionally suffixed with `()` if there are no arguments:

Less

```
.mixin {
  border: 1px solid #000000;
}

.foo {
  .mixin();
}
```

CSS Output

```
.mixin {
  border: 1px solid #000000;
}

.foo {
  border: 1px solid #000000;
}
```

If you want to define a mixin without it getting compiled as its own selector, suffix `()` on the definition of the mixin like so:

Less

```
.mixin() {
  color: black;
}

.foo {
  .mixin;
}
```

CSS Output

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

This also works with more deeply specified mixin selectors. Due to this behavior, it is possible to namespace your mixins which is handy for categorizing mixins that all relate to some similar property. A simplistic example of this would be to namespace a set of mixins that control the direction of CSS' `float` property. The `.float` class is created as the mixin which we don't want to be outputted in our compiled CSS and nested within are `.left` and `.right`. In `.foo` and `.bar` we call the namespaced mixins:

Less

```
.float() {

  .left {
    float: left;
  }

  .right {
    float: right;
  }

}

.foo {
  .float.left();
}

.bar {
  .float.right();
}
```

CSS Output

```
.foo {
  float: left;
}

.bar {
  float: right;
}
```

There are actually a multitude of ways to use namespaced mixins in this language. For instance, all of the following would output the same CSS when added to the property block of a selector:

- `.float .left`
- `.float .left()`
- `.float.left`
- `.float.left()`
- `.float > left`

Mixin Arguments

Mixins in Less also support one or more arguments in their definition. Arguments defined on these so-called parametric mixins are either comma separated or semicolon separated and again, just like Sass, mixin arguments can have default values and can be accessed using the argument keywords where they are referenced. Below is a simple example of a Less mixin with multiple arguments:

```
.mixin-foo(@border-width, @border-style, @border-color: #000000) {
  border: @border-width @border-style @border-color;
}
```

We can call this mixin on a new selector in a number of ways. We can simply pass values in a comma separated list, specify the argument values with keywords, and completely ignore the third parameter since it was defined with a default value of `#000000`:

- `.mixin-foo(1px, solid, #0000ff)`
- `.mixin-foo(@border-width: 1px, solid, #0000ff)`
- `.mixin-foo(1px, solid)`

Commas in a mixin's argument definition can serve another purpose beyond acting as a separator; they may also act as a CSS list separator. If you decide to use commas to separate arguments, then you remove the ability to add CSS list as an argument since Less will assume each item in the list is its own argument. For this reason, it is recommended to separate arguments with semicolons rather than commas.

Let's rewrite `.mixin-foo` to include a CSS list argument:

Less

```
.mixin-foo(@border-width; @border-style; @border-color; @font-stack) {
  border: @border-width @border-style @border-color;
  font-family: @font-stack;
}

.foo {
  .mixin-foo(1px; solid; #0000ff; Helvetica, Arial, sans-serif);
}
```

CSS Output

```
.foo {
  border: 1px solid #0000ff;
  font-family: Helvetica, Arial, sans-serif;
}
```

Something to note about Less mixins is that it is completely valid to write multiple mixins of the same name with a varying number of arguments on each. When a mixin like this is referenced in a selector, Less will attempt to use any and all mixins whose number of passed arguments match the number of defined arguments. To see this functionality in action, let's define a couple versions of `.mixin-foo`:

Less

```
// Mixin A
.mixin-foo(@border-width; @border-style; @border-color; @font-stack) {
  border: @border-width @border-style @border-color;
  font-family: @font-stack;
}

// Mixin B
.mixin-foo(@line-height, @height: 100px) {
  line-height: @line-height;
  height: @height;
}
```

```less

// Mixin C
.mixin-foo(@width) {
  width: @width;
}

// Mixin D
.mixin-foo(@font-size) {
  font-size: @font-size;
}

.foo {
  .mixin-foo(100px);
}
```

CSS Output

```css
.foo {
  line-height: 100px;
  height: 100px;
  width: 100px;
  font-size: 100px;
}
```

So what happened there? Our call to `.mixin-foo` included only one parameter which matched the number of arguments defined for both *Mixin C* and *Mixin D* so `.foo` got its `width` and `font-size` properties from those two mixins respectively. Additionally, *Mixin B's* second argument had a default value defined so our call of `.mixin-foo(100px)` in `.foo` matched it as well, giving it both `line-height` and `height` values.

In the above scenario, if we had called `.mixin-foo` with four arguments, only *Mixin A* would have matched. If we had called it with only three arguments, none of the mixins would have matched and Less would have given us an error during compile time.

There's a lot more you can do with Less mixins than we have room to cover in the scope of this book. To learn more about using mixins as functions, argument pattern matching, and other nifty mixin features of Less check out the documentation.

Extending

Extending a selector onto another selector is possible by using Less' special `:extend` pseudo-class. This pseudo-class takes one or more arguments of selectors. These arguments tell Less which selector's styles you want to extend onto the current selector. In a list of selectors, any and all of them may use the `:extend` pseudo-class:

Less

```less
.foo {
  color: green;
}

.bar {
  background: black;
}

.baz:extend(.foo),
.qux:extend(.bar) {
  font-size: 1.5em;
}
```

CSS Output

```
.foo,
.baz {
  color: green;
}

.bar,
.qux {
  background: black;
}

.baz,
.qux {
  font-size: 1.5em;
}
```

In addition to the selector argument(s), the `:extend` pseudo-class may optionally be passed the keyword `all` which will match every instance of the preceding selector argument(s). To piggy-back on our previous extend example, let's create a nested instance of `.bar` and pass `.qux:extend(.bar)` the `all` keyword:

Less

```
.foo {
  color: green;
}

.bar {
  background: black;

  .norf & {
    border-radius: 8px;
  }

}

.baz:extend(.foo),
.qux:extend(.bar all) {
  font-size: 1.5em;
}
```

CSS Output

```
.foo,
.baz {
  color: green;
}

.bar,
.qux {
  background: black;
}

.norf .baz,
.norf .qux {
  border-radius: 8px;
}

.baz,
.qux {
  font-size: 1.5em;
}
```

Above, we can see that `.norf .baz` and `.norf .qux` now have the `border-radius` property in their ruleset. Without the `all` keyword, the `.qux` would have only extended the selector we gave it, `.bar`. In that scenario, `.norf .bar` would not have matched our selector argument. Selector arguments without the `all` keyword must match exactly.

The Scope of `:extend`

In the responsive driven world of front-end development we live in, you'll probably find yourself writing a lot of media queries to change the style of your content to best suit a variety of scenarios. Something to keep in mind when using `:extend` in Less is that you cannot extend a class that is outside your current `@media` block:

Less

```
@media screen and (min-width: 400px) {
  .foo {
    color: blue;
  }
}

@media screen and (min-width: 600px) {
  .bar:extend(.foo) {
    background: brown;
  }
}

.baz {
  line-height: 1.5;
}
```

CSS Output

```
@media screen and (min-width: 400px) {
  .foo {
    color: blue;
  }
}

@media screen and (min-width: 600px) {
  .bar {
    background: brown;
  }
}

.baz {
  line-height: 1.5;
}
```

In the above example, `.bar`'s extension of `.foo` is completely ignored since `.foo` does not exist in the scope of this `@media` block.

While extending across `@media` blocks is a no-go, we are able to extend a selector within a `@media` block onto a globally scoped selector. In the previous example, that's `.baz`. The caveat here is that extending a `@media` scoped selector onto a globally scoped selector will keep the `@media` scope on output. Essentially, anything you extend that comes from a selector within a `@media` block will only be extended in that `@media` block:

Less

```less
.baz:extend(.foo) {
  line-height: 1.5;
}

@media screen and (min-width: 400px) {
  .foo {
    color: blue;
  }
}
```

CSS Output

```css
.baz {
  line-height: 1.5;
}

@media screen and (min-width: 400px) {
  .foo,
  .baz {
    color: blue;
  }
}
```

Summary

We've covered a lot about Less' features in this section, but there are many many more. In a lot of ways Less and Sass are quite similar, but under the hood and in specific implementations of their shared features there are a lot of subtle differences. Less, as a language, is very mature and well documented and should suit just about any project just fine. It's easy to get started with and as you get more comfortable using it the more complex features it provides can be an enormous asset to your front-end workflow.

If you're interested in more than a quick introduction to Less, head over to the Less website and dig in.

Stylus

The Stylus preprocessor runs on `node` and boasts a huge number of quality-of-life type features. The Stylus syntax makes nearly everything optional, from colon separators between properties and values to braces around rulesets, even mixin argument parenthesis.

Stylus is relatively new on the scene and it shares a few similarities with other preprocessors. For example, Stylus supports an indented syntax that closely matches Sass' original syntax. Additionally, Stylus supports nesting selectors in the exact same way that Less and Sass do, no matter which syntax you're using.

Installation

Installing Stylus works just like installing Less or any other `node` package, by using `npm`:

```
npm install -g stylus
```

Run a quick check to see if everything worked and you're ready to get started:

```
stylus --version
0.49.3
```

Using Stylus

The `stylus` program is by itself just an interpreter. Running it without any parameters will bring you into a writing mode where you can write stylus code. When you quit the program, it'll spit out whatever Stylus you wrote into regular CSS.

If you want to be able to save your Stylus code, you'll need to store it in a file and then run the `stylus` program on that file, much like you would for Sass or Less. The typical extension for Stylus files is `.styl`, although like most other things in Stylus, that extension is optional.

Once you have a file to store your Stylus code in, you can simply run:

```
stylus your_file.styl
```

and the `stylus` program will generate a CSS file with a matching name. In this case, that's `your_file.css`.

To see the full list of the `stylus` program's options run:

```
stylus --help
```

Stylus Syntax

As mentioned in the introductory paragraph of this section, almost everything in Stylus is optional. What's more, you can mix and match different syntax styles in the same Stylus document without any problem.

This is both exceedingly neat and potentially disastrous, particularly for a project with multiple developers working in the code. It is highly recommended, if you choose Stylus for your project, that everyone agree on a particular syntax. Like Less and Sass' SCSS syntax, Stylus does support the regular CSS syntax in addition to all the other craziness.

Let's take some example CSS and write it in a few valid, but different, Stylus syntaxes:

```
.foo {
  color: blue
  background: black
}

.foo
  color: blue;
  background: black;

.foo {
  color blue
  background black
}
```

Every one of those different syntaxes will provide the same CSS output:

```
.foo {
  color: blue;
  background: black;
}
```

Variables in Stylus

Unlike the other preprocessors we've covered so far, you don't need to prefix any particular symbol to denote a variable in Stylus, although it does optionally let you use `$` if that helps you sleep at night:

Stylus

```
darkColor = #777777

.foo
  color: darkColor
```

CSS Output

```
.foo {
  color: #777777;
}
```

Mixins in Stylus

The syntax for Stylus mixins is a little different from the other preprocessors, but if you've been reading this chapter, the following example should be fairly easy to grasp at a glance:

Stylus

```
dashed_border_mixin(padding, border_width= 1px, border_color= #000000)
  border: border_width dashed border_color
  padding: padding

.foo
  dashed_border_mixin(5px)
```

CSS Output

```
.foo {
  border: 1px dashed #000;
}
```

In Stylus, arguments are separated by commas, default values are set with `=`, and for the most part, mixins look the same as variables.

Extending

Stylus shares it's extend functionality with Sass in both syntax and behavior:

Stylus

```
.foo
  color: green;

.bar
  @extend .foo
  background: blue
```

CSS Output

```
.foo,
.bar {
  color: #008000;
}

.bar {
  background: blue;
}
```

Summary

The Stylus language can make your own personal preferences a syntactic reality. You can really write your styles nearly any way you please. This language also sports a hefty number of features that the other preprocessors we've covered do. Although the authors haven't used Stylus exclusively in any client projects, you might find some of its flexibility very attractive. If Stylus is something you'd like to try for yourself, you can find more information on the Stylus website.

Performance and CSS

When you're trying to improve your page's performance you likely don't think of CSS as a primary concern. In the grand scheme of things, improving your CSS' performance won't provide as many quick gains as reducing the number of image requests or optimizing your JavaScript will. That said, there are still some things you can do to make browsers render your CSS more efficiently.

Likely, your largest CSS performance gains will come from optimizing how your styles get loaded onto the page. Inlining critical CSS will avoid the need to fetch a stylesheet entirely. Compressing and minifying your CSS with something like cssmin will decrease the file size of your stylesheets and lower the time it takes for the browser to download and parse that stylesheet. Optimizing the way your server actually serves your stylesheets will provide a similar gain, but caching and server configuration are outside the scope of this section.

CSS Painting

Another performance optimization can come from improving the time it takes for a browser to actually paint all of your styles onto the screen. The more things you have on the screen, the more work your browser has to do to. This is a pretty tricky topic and, for most websites, getting down and dirty with visual paint times is probably not necessary, but it's good to know about how it all works.

As the browser builds the DOM tree, it concurrently starts building up a render tree which is made up of all the visual stuff that's going to be painted onto the screen. This is how the browser knows in what order to apply your styles on the screen. Each visual element becomes a node on the render tree containing all of the styles that make it up. These styles come from inline styles, the stylesheets applied to the page and, in some cases, local stylesheets that users have added to customize their browsing experience. Browser engines all handle this process a little differently, but the concept is pretty much the same.

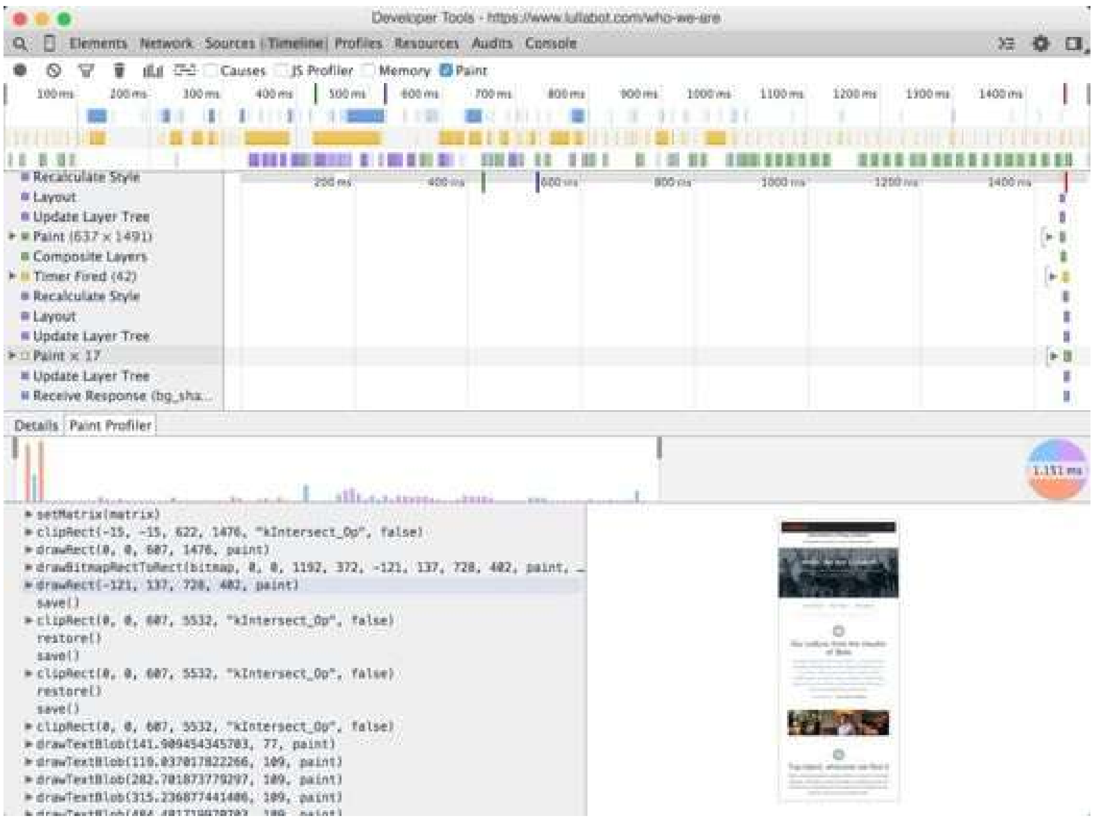
The browser will typically start at the top of a document and evaluate the style of the first element it sees, usually `body`, before painting it. At this point the browser doesn't care about anything else in the document. Once it finishes painting `body` to the screen, it'll move onto the next element and go through the same process of evaluating the style and painting it to the screen. After this child element has been painted, the browser then tries to figure out if it needs to *repaint* the parent element. If the child element changed the height of its parent, `body` will be reevaluated and painted once more.

The nodes of the render tree all get calculated individually for layout and style and that's the gist of how CSS actually gets applied. The actual *painting* part of that process follows a strict order defined by a number of conditions. These conditions are things like whether or not the element being styled is a root element, whether the element is positioned inline or block, or whether it's a child of another element. The general order of painting looks like this:

- background color
- background image
- border
- any child elements
  - background color
  - background image
  - border

You can find a lot more information about this in the [CSS specification](#).

So what can we do with this information? Now that we're aware of how the browser puts all our styles into practice, we can focus on trying to limit the amount of overall painting our browsers have to do by applying styles further up the cascade, removing unnecessary elements from the page, and tracking down styles that take a long time to paint. Chrome actually has a paint profiler in the Timeline section of its Dev Tools that makes it pretty easy to see each paint event and how long it took to happen. With this information you can quickly see the styles that take the longest time for the browser to render. More often than not, things like CSS shapes and gradients will be your primary culprits.



```
▸ setMatrix(matrix)
▸ clipRect(-15, -15, 622, 1476, "kIntersect_Op", false)
▸ drawRect(0, 0, 607, 1476, paint)
▸ drawBitmapRectToRect(bitmap, 0, 0, 1192, 372, -121, 137, 728, 402, paint, …
▸ drawRect(-121, 137, 728, 402, paint)
  save()
▸ clipRect(0, 0, 607, 5532, "kIntersect_Op", false)
  restore()
  save()
▸ clipRect(0, 0, 607, 5532, "kIntersect_Op", false)
  restore()
  save()
▸ clipRect(0, 0, 607, 5532, "kIntersect_Op", false)
▸ drawTextBlob(141.909454345703, 77, paint)
▸ drawTextBlob(110.037017822266, 109, paint)
▸ drawTextBlob(282.701873779297, 109, paint)
▸ drawTextBlob(315.236877441406, 109, paint)
```

CSS Selectors

The selectors you write to style your elements get evaluated from right to left. The more specific your selectors are the more work the browser needs to do to figure out whether or not an element on the page needs to be styled. If we had some CSS like this, the browser would go through our entire document looking for `<span>` elements. At each `<span>` it would look to see if there was a `<p>` parent element, then for each of those it would look for a `<div>` parent and so on:

```
.foo div p span {
  color: green;
}
```

This can take a very long time (comparatively) because the render tree can get pretty massive and there could be a lot of false matches. If only one `<span>` in our document exists inside a `<p>` and there are 50 `<span>`s then our browser has done a lot of needless work because it had to rule out 49 of them. BEM, introduced earlier in this chapter, avoids this problem entirely since every element typically only has one class applied to it.

It’s worthwhile noting that optimizing this part of a CSS document is probably not going to give you much bang for your buck. These days, browsers are pretty excellent at evaluating selectors, and worrying too much about the efficiency of your selectors in relation to page performance is probably a good recipe for insanity. Knowing about how the selectors work, however, is important knowledge to have tucked away.

Summary

In this section we learned a little bit about how browsers work to actually render your styles on the page. It’s some pretty interesting stuff. There’s a lot more you can do to optimize web pages, but as mentioned at the beginning of this section, your biggest performance gains will almost certainly be the optimization of your CSS’ delivery to the browser. Chasing paint times and selector efficiency might result in some lean performance gains, but as our machines and our browsers improve, these become less and less important in front-end development.

If you want to learn more about making your pages render efficiently, it’s highly recommended that you read through Google’s PageSpeed Insights. There you’ll find all kinds of great tips for optimizing images, prioritizing renderable content, and much more.

Styling Summary

In this chapter we’ve already covered a lot of topics and concepts that might help you better understand the current landscape of styling the front-end. This was possibly one of the most difficult chapters to write because this landscape is evolving almost daily, but if you have any questions or would just like to talk about best practices the Google Group for this book is a good place to start.

Dependency Management

There is an entire universe of front-end development frameworks, plugins and tools available for building applications. The open source movement brings with it a huge arsenal for building amazing applications. By using some of these available ‘packages’, we are able to develop faster and safer.

But you’ll quickly come to realize that the biggest problem now is being able to efficiently manage and keep up to date all the packages that you use. ‘Dependency hell’ is a term you may hear front-end developers use. It becomes even more difficult when working as part of a team, as it’s crucial that all developers on a project use the same package versions to avoid conflicts or bugs.

Do not fear! ‘Package managers’ are here to save the day. In this chapter you’ll be introduced to several essential package managers that are highly recommended for incorporating into your workflows.

What is a Package Manager?

All developers create software differently. Because of this, installing, upgrading and deleting software can become a chore. That is where package managers come in to play. A package manager, itself a piece of software, keeps track of what is installed and allows you to automate the process of managing software using a standard API. By adhering to the standards set out by a package manager, software developers can offer their packages to a broader audience.

Node.js and the Node Package Manager (npm)

There are a huge number of Node.js packages available for front-end development tasks. Many of the popular ones are introduced in this book, such as Grunt. For those who don’t know what Node.js is, here’s a quote from their homepage: “Node.js is a platform built on Chrome’s JavaScript runtime for easily building fast, scalable network applications.” While we’re not going to be building our own Node.js packages, we will need to install Node.js itself so that we can use Node packages.

The Node Package Manager (npm) was not included as part of Node.js originally but since version 0.6.3 it is automatically installed for us. `npm` allows us to install and manage all of the Node.js packages that are used in an application. Node.js is a huge open source community. You can browse the directory of available packages at their homepage, https://www.npmjs.org/. At the time of writing there were over 70,000 packages racking up 10 million downloads a day. It’s obvious that npm is a very popular resource.

Installing Node.js

Let's get Node.js installed. If you're on OS X and use homebrew (another Package Manager for OS X) then all you need to run is `brew install node` in the command line.

For everyone else, you can go to the Node home page and download the pre-built binary installer. Alternatively, you can clone their Git repository and build from there. Running the following commands should get the job done:

```
git clone https://github.com/joyent/node.git
cd node
./configure
make
sudo make install
```

There may be some dependency requirements and if you'd like to find out more on installing on your specific OS, check out their documentation.

We can confirm we've installed Node correctly by running the following in the command line:

```
node --version
```

As previously mentioned, npm is bundled with Node. To confirm npm is installed, run:

```
npm --version
```

The package.json

We're going to need to create a file in our project directory named `package.json` that will contain metadata relevant to our application. This file will define basic project information and, most importantly, the package dependencies, (i.e. references to the Node packages we require for our application). This file is important because it will allow both ourselves and other developers to quickly install the dependencies with a single command, `npm install`. Although it's possible to get away without having this file, one of the most important issues it solves is conflicts. If a developer has a different version of a Node package that is used in your application installed globally, then it may conflict with whichever one you actually want them to use.

The easiest way to understand the purpose of this file is to jump in and create it. Let's create an example project directory and `package.json` file:

```
mkdir myApp
cd myApp
touch package.json
```

Using your favorite text editor, open `package.json` and add the following:

```
{
  "name": "myApp",
  "description": "An all-round amazing awesome app.",
  "version": "0.0.1",
  "devDependencies": {}
}
```

As the file name suggests, this file contains JSON. So far we've defined the name of our project, its description and its version using the structure `major.minor.patch`. (For more on semantic versioning, check out http://semver.org/). This basic project metadata is required in `package.json` files. We've also defined `devDependencies` as an empty object.

This property will be where we list the package dependencies for our project. (Incidentally, you can automate the creation of your `package.json` file by using the command `npm init` but it's good to know how to write the file yourself!)

Installing packages

This is probably why you have npm in the first place. You want to install something, right? It's very easy. After you find the name of the package you want to install, navigate to your project folder and run:

```
npm install <package name>
```

This will install the package to a folder called `node_modules` within the projects directory. But, if you open `package.json`, you will notice nothing has been added to the `devDependencies` object. It's important that we add a reference to the package there so that the next time we or another developer want to set up the application for development, the referenced packages are easily installable.

Although we could edit our `package.json` manually, it's best to append the `--save-dev` parameter when installing to automate the process. Let's try this with one the most popular npm packages as an example, express:

```
npm install express --save-dev
```

This installs the package on our system (in the `node_modules` directory) but also saves a reference to the version we've installed in `package.json`. Open it up and you'll see something like this:

```
{
  "name": "myApp",
  "description": "An all-round amazing awesome app.",
  "version": "0.0.1",
  "devDependencies": {
    "express": "^4.8.7"
  }
}
```

After you've installed all of the packages that you need this way, all that a new developer has to do to make sure they have the correct version of everything installed is navigate to the project directory and run:

```
npm install
```

Pretty cool, huh?

Another thing worth mentioning is that you can also install packages globally. By default, `npm install` will place packages in the `node_modules` folder of your current working directory. Installing globally will place the modules outside of the current directory and instead place them in your global file system, (e.g., `/usr/local/lib/node_modules` on OS X). To install a package globally, you can apply the `-g` parameter:

```
npm install <package name> -g
```

Updating packages

Probably the best part of a package manager is how crazy easy it is to update stuff. Because who really loves spending time on maintenance? The `npm update` command searches the registry for newer versions of installed packages and updates them automatically. It also points dependent packages at the new version and removes the older versions. We want to make sure that package version numbers are updated correctly so we must apply the `--save-dev` parameter again:

```
npm update --save-dev
```

Alternatively, we can update just a single package at a time. If we have a large number of packages and update them all at once, it might be difficult to debug the application if something breaks. To update a specific package run:

```
npm update <package name> --save-dev
```

Listing packages

Simply looking inside the `node_modules` folder is not going to tell you every package that is installed. Installed packages may also depend on other packages and so they each may have their own `node_modules` folder too.

The `npm list` command will give you a nicely formatted hierarchical overview of what is installed with version numbers. The command also lets you filter the results (i.e. `grep`) by appending the package name:

```
npm list <package name>
```

Uninstalling packages

You can uninstall packages easily using the `npm uninstall` command. You'll need to append the package name and `--save-dev` so that the `package.json` is updated:

```
npm uninstall <package name> --save-dev
```

Getting help

You can append the `--help` parameter to any command for some quick syntax information. Alternatively, prepend `help` to get a more detailed description. For example, try:

```
npm install --help
npm help install
```

Bower

We've established that npm manages Node.js packages. Bower, built by Twitter, is a package manager that is actually a Node package itself. Bower offers a solution to front-end package management. You'll find many of the popular CSS and Javascript frameworks, libraries and plugins are available through Bower. Visit their Search page for a list of all the available packages. In a similar fashion to npm, Bower allows us to keep references to which packages are used in our app.

Installing Bower

Let's install Bower using npm:

```
npm install bower -g
```

Notice the `-g` flag which indicates that we want to install this package globally. What this means is that we will be able to use the package regardless of the directory that we are in. We install this package globally because we're not very bothered with maintaining its version across environments and we'd rather be able to use it from the command line from new project directories without having to install it again each time.

The next step is to create a file named `bower.json` which serves a similar purpose as `package.json` does for npm. This time, let's use a helper function to get started:

```
bower init
```

This will prompt us with a set of questions to which the answers are used to create our `bower.json` file. Here are the settings used for our example:

```
[?] name: myApp
[?] version: 0.0.1
[?] description:
[?] main file:
[?] what types of modules does this package expose?
[?] keywords:
[?] authors: Joe Fender <joe@example.com>
[?] license: MIT
[?] homepage:
[?] set currently installed components as dependencies? No
[?] add commonly ignored files to ignore list? No
[?] would you like to mark this package as private which prevents
      it from being accidentally published to the registry? Yes
```

The resulting `bower.json` file should then look something like this:

```
{
  name: 'myApp',
  version: '0.0.0',
  authors: [
    'Joe Fender <joe@example.com>'
  ],
  license: 'MIT',
  private: true
}
```

By default, any packages installed with Bower will be placed in a folder called `bower_components`. If we want to change the name of this folder, we can do so by creating a file called `.bowerrc` in the application's root directory and adding the new directory name. For example:

```
{
  "directory": "vendor"
}
```

Searching packages

Something neat that Bower does is allow us to search for packages from the command line. Simply run:

```
bower search <keyword>
```

This will return packages that contain the given keyword. You can also omit the keyword to get the entire list of available Bower packages.

Installing packages

Just like with npm, we can install packages using this command:

```
bower install <package name> --save
```

Notice we use `--save` as opposed to `--save-dev` used in npm. They both do the same job of ensuring that our `bower.json` file is updated with a reference to the installed package and its version. Bower does in fact let you use `--save-dev`. The difference between the two commands is that the packages will be referenced in a different object within `bower.json`. When using the `--save` parameter, packages are referenced in the `dependencies` object whereas the `--save-dev` parameter adds them to the `devDependencies` object. Most of the Node packages installed are for development purposes, such as task running, which we will dive into in the next chapter. That is why we use `--save-dev` in npm. However, the majority of the packages installed with Bower will be served directly to the clients of our applications. For example, some CSS files or a JS plugin. You can definitely use both `--save` and `--save-dev` to separate references to installed Node or Bower packages. It's totally up to you which combinations you use.

Listing packages

Listing local packages and available updates is simple:

```
bower list
```

Updating packages

To update packages to their latest version, run:

```
bower update <package name> --save
```

As with npm, the package name is optional and if omitted then all installed packages will be updated.

Uninstall packages

Again, this one is fairly straightforward:

```
bower uninstall <package name> --save
```

Bundler

Another very useful piece of the package management puzzle is Bundler. Bundler helps you to manage Ruby project 'gems'. Gems are programs and libraries written in the Ruby programming language. There are a few useful gems for front-end development that we introduce in this book, so it's recommended that you install Bundler to manage them.

Depending on your OS you may need to install Ruby first. To do so you should follow the documentation on the Ruby homepage.

Now that Ruby is set up, let's get Bundler installed from our command line:

```
gem install bundler
```

This installs the Bundler gem globally on the system. As has been the case with our other package managers, Bundler requires a file in our projects root directory that will store project dependencies. The file's name is `Gemfile` and unlike npm and Bower, has its own syntax as opposed to using JSON. Let's create the file `Gemfile` and add a single line to it in our text editor:

```
source "https://rubygems.org"
```

This tells Bundler where to look for the Ruby gems that we want to install. Alternatively, we could run `bundler init` to have Bundler create `Gemfile` for us.

Installing gems

Unlike npm and Bower, there isn't a command to install gem packages. We need to manually edit the `Gemfile` and write the references. For example, say that we want to install Sass. We would edit our `Gemfile` so that it looked something like:

```
source "https://rubygems.org"
gem 'sass', '~> 3.3.6'
```

Then to install the package, we go to the project's root directory and run:

```
bundle install
```

Notice that we use `bundle` and not `bundler`. That is not a typo! If you need help figuring out how to write the version number, check out the Gemfile help page. Also, it's worth noting that gems are not installed within a local folder. Instead, they are installed globally.

Updating gems

There is some disagreement in the community about what part Bundler plays in updating gems. You might find it easiest to just manually edit `Gemfile`, change the version number and then run `bundle install` again. For more information on the reasoning behind some of the functionality decisions made in Bundler, check out their Purpose and Rationale page.

Uninstalling gems

To uninstall a gem, you will first need to run:

```
gem uninstall <gem name>
```

Then, edit your `Gemfile` and remove the reference to the gem.

Documentation

Software is only as good as its documentation. Without clear documentation of your project you are likely to run into frustrations when developing, maintaining or transferring knowledge to new developers. It's always good to start documenting projects immediately so that you don't have to play catch-up. Ideally, you should create documentation as soon as you have set up your package managers.

Start by creating a README file in the project's root directory that will explain the process required to install the package managers. As with a large number of projects these days, we may well use GitHub to maintain our code repository. If you do use GitHub, or think you might in future, you can create a `README.md` file. The `.md` file extension stands for Markdown and allows you to write formatted text files with shorthand syntax. Let's create `README.md` and add something along these lines:

```
# Getting Started

Before working on this application we'll want to make sure we have all
the required dependencies. Here is a quick guide to get started.

1. Install Node with `brew install node`
2. Install Bower globally with `npm install bower -g`
3. Install local Node packages with `npm install`
4. Install Bower packages with `bower install`
5. Install Bundler with `gem install bundler`
6. Install required Bundler gems with `bundle install`
```

You could create a script that would run these six commands behind the scenes for the user but, as with everything in this book, using the tools directly gives you a better understanding of how they work. Ultimately, this will give you more confidence when integrating them into your own projects.

Summary

In this chapter, we looked at the basics of three of the most popular front-end developer package managers: npm, Bower and Bundler. You've learnt how to install them all and how to manage their packages. We finished up by writing some documentation of the installation process.

Package managers are an extremely important part of front-end development these days. Using them in all of your projects is highly recommended, regardless of project size, so that you are able to maintain and scale them efficiently.

Automation

Time is a key factor in productivity. This chapter will explore some ways to automate your development workflow so you spend less time working and more time making awesome. You’ll be introduced to ‘task runners’ which help automate repetitive tasks. You’ll also discover ways to speed up the process of laying foundations for new projects, and you’ll pick up a few other tips to help you be a more efficient developer.

Task Runners

Task runners, also known as ‘task automation frameworks’, help you manage and execute sets of defined tasks. A task can be seen as any repetitive action you have to take when developing for the web. Tasks can include actions such as copying a file between directories or checking your JavaScript for syntax errors.

With a task runner, you can build, preview and test your application much more easily than if you did all the work yourself. The strongest attribute of task runners is their flexibility, which means you can configure and run pre-defined tasks exactly how you want. Furthermore, if you can’t find a pre-defined task that meets your needs, you can even write your own.

In this chapter you’ll learn a bit about the popular task runners Grunt, Gulp, CodeKit and Yeoman, including how to install them and how to configure them to meet your specific needs.

Grunt

Grunt is one of a front-end developer’s best friends and is the most popular and widely used task runner. It is a framework based on JavaScript and has a thriving ecosystem. In addition, there are a huge number of plugins available for Grunt. If you’re not sure where to start with task runners, then start here.

Installing Grunt

Let's work through the installation together. We'll install Grunt using `npm` from the command line. (If you haven't done so already, please read the *Dependency Management* chapter first to learn more about how to set up and use `npm`.)

The first tool we're going to install is Grunt's command line interface, `grunt-cli`. It's worth noting that this does not actually install Grunt itself. It only provides us global access to the `grunt` keyword from the command line which is then used to load and run the version of Grunt that has been installed locally within the project. It is important to remember to install `grunt-cli` globally but `grunt` locally. For more information on global vs local installation, check out this article over at the official NodeJS blog.

We'll head over to our command line and run:

```
npm install -g grunt-cli
```

To verify that Grunt CLI was installed successfully and that we can access the `grunt` command we'll run the following:

```
grunt --version
```

Now let's install Grunt. We'll need to navigate to our project's root directory first:

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

This will install Grunt locally and save a dependency reference to it from within the `package.json` file.

The Gruntfile

Now that we've installed Grunt, it's time to introduce `Gruntfile.js`. Like `package.json` for `npm`, the Gruntfile belongs in the root directory of our project. It defines our project and task configurations and tells Grunt which plugins we want to use to run these tasks. We're not going to define any tasks just yet but let's create the bare bones of the Gruntfile so that we can add to it later. Create `Gruntfile.js` in the project's root directory and add the following:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig();

  // Default tasks.
  grunt.registerTask('default', []);
};
```

You'll notice that this file, as the file name suggests, is a JavaScript file. All we've done here is called `initConfig` which initializes our Grunt configuration. This is where we will define our tasks later. We've also called `registerTask` to initialize a set of tasks labelled `default`. There are no actual tasks here yet but Grunt requires us to define this for it to run. After saving `Gruntfile.js` we can try running `grunt` in the command line from the project root. We should get a shiny green message stating:

```
Done, without errors.
```

This means Grunt was able to run all of the tasks defined in default without failure.

Yay! We’ve just run Grunt for the first time. The more you use Grunt, the more you’ll love it!

Popular Plugins

There are thousands of plugins available in the open source community for Grunt. But there are a few that have become staples for the modern front-end developer. Depending on the project, you may not need all of these, but it doesn’t hurt to know about them. Obviously, these are only a select few and may not include everyone’s favorites.

Watch, LiveReload and Connect

Quite possibly the most popular plugin is Watch. It allows you to run specified tasks whenever a watched file changes. So, for example, you could minimize your CSS files whenever one of them changes or run JavaScript tests whenever you save script changes. The Watch plugin only does the ‘watching’ part. You have to specify exactly which tasks are to be run when a file change is detected.

To install Watch, we’ll run the following from the command line:

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

Edit the `Gruntfile.js` so that the plugin gets loaded:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig();

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-watch');

  // Default tasks.
  grunt.registerTask('default', []);
};
```

All we have done here is use the `loadNpmTasks` callback to tell Grunt to load the Watch plugin.

To demonstrate how to use Watch, let’s set up something cool using LiveReload and Connect. We’ll set up a local web server that will automatically refresh any browser viewing our web site when a file change occurs. This is great because it means we don’t have to refresh every time a change is made to one of the files. The official homepage for LiveReload describes the tool as “A happy land where browsers don’t need a refresh button”!.

Usually, when setting up a local web server you have to configure software system-wide. You may have used tools such as MAMP or you may have configured a local LAMP stack yourself. But a good alternative is to use a more lightweight approach to testing web applications using a combination of Watch, Connect and LiveReload plugins.

Connect is an HTTP web server framework built for NodeJS. It’s a lightweight static web server that provides plugins known as ‘middleware’. Connect comes with a bunch of middleware such as session support, a cookie parser and more.

There is a Grunt plugin for Connect which allows us to start a Connect web server so let's get that installed:

```
npm install grunt-contrib-connect –save-dev
```

We now need to tell Grunt when and how to use the plugin. Every plugin you install should have a README file available in its root directory (in this case, `node_modules/grunt-contrib-connect/README.md`) which will outline how to configure the task. Let's edit our `Gruntfile.js` again to set up the `connect` task:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
      // Defines our Connect web server.
    connect: {
      server: {
        options: {
          port: 9000,
          hostname: 'localhost',
          base: ['dist/']
        },
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.loadNpmTasks('grunt-contrib-connect');

  // Default tasks.
  grunt.registerTask('default', []);

  // Running `grunt server` runs our Connect server until terminated.
  grunt.registerTask('server', ['connect:server:keepalive']);
};
```

Here, we told Grunt to load the plugin using `loadNpmTasks` exactly as we did with Watch. Task configurations are added within the `initConfig` callback. We've defined a task called `connect` that has a sub-task called `server` containing the settings for the task. We've defined our Connect server to run on port 9000 at `http://localhost:9000` and to serve the folder `dist` (short for distribution).

As described in the code comment near the bottom, running `grunt server` from the command line will launch the Connect server until terminated (made possible through the keepalive parameter). You will then be able to access the web application by pointing your browser to `http://localhost:9000`. However, there is currently no dist folder so let's move on to creating that and an example `index.html`:

```
mkdir dist
cd dist
touch index.html
```

Open up `index.html` to add some basic HTML such as the following:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My App</title>
  </head>
  <body>
    <p>Hello world! Welcome to my application.</p>
  </body>
</html>
```

Now, if you go to the command line, run `grunt server` and then point your browser to `http://localhost:9000` you should see the “Hello world” message.

To finish, we need to configure the Watch task so that when any change occurs in `index.html`, a browser refresh is triggered. Let’s update our `Gruntfile.js` to look something like this:

```
// Project configuration.
grunt.initConfig({
  // Defines our Connect web server.
  connect: {
    server: {
      options: {
        port: 9000,
        hostname: 'localhost',
        base: ['dist/'],
        livereload: true
      },
    },
  },

  // Watch our files for changes.
  watch: {
    options: {
      livereload: true
    },
    html: {
      files: ['dist/index.html'],
      tasks: []
    }
  },
});

// Load our Grunt plugins.
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.loadNpmTasks('grunt-contrib-connect');

// Default tasks.
grunt.registerTask('default', ['watch']);

// Running `grunt server` runs our Connect server until terminated.
grunt.registerTask('server', ['connect:server:keepalive']);
```

The first thing you may notice is we’ve added our `watch` task configuration. We’ve set up a sub-task called `html` that watches for changes to `index.html`. We haven’t actually told Watch to execute any specific tasks once a change is detected. However, we’ve enabled the `livereload` boolean both in the Connect and Watch options. This basically tells Connect and Watch to use LiveReload in tandem. Connect comes complete with the connect-livereload middleware Node package and, as of recently, Watch now comes with

LiveReload server functionality included so all we need to do is set those two booleans. Lastly, we updated our `default` task so that the `watch` task is run.

To test this, you're going to need two terminal windows. In the first, run `grunt server` and in the second run `grunt`. You should see the following status messages respectively:

```
Running "connect:server:keepalive" (connect) task
Waiting forever…
Started connect web server on http://localhost:9000
```

And:

```
Running "watch" task
Waiting…
```

Make sure your browser is looking at `http://localhost:9000` and then make a change to the "Hello world" message in your `index.html`. Don't forget to save the file! Once you save the file, switch back over to your browser and you'll see that the page had refreshed (probably before you even made it back over) and reflects the change you made. Neat huh?

The power of Watch really shines when you start to configure it for larger front-end applications. For many, it becomes the center of their automation workflow - the trigger to every task. It really is a great plug-in.

Copy and Concat

As programmers, we aim to develop our projects to be as modular as possible. This quickly leads to a lot of files and one of the biggest challenges is keeping on top of them all. Copy and Concat are a couple of the handiest file management plugins to know about.

Copy

Copy simply allows you to copy files and folders from Grunt. Copy has endless uses but for the purposes of demonstration let's create an example scenario. Let's say that we want to copy `src/index.html` to `dist/index.html` as part of our Grunt build process because the `dist` folder is the one that is to be served by our web server and `src` is where we actually edit our code. Obviously with a real application there would likely be more files to copy than just a single file but let's roll with this anyway.

Before getting started, don't forget to install the plugin:

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

This is how we might go about setting up our `Gruntfile.js` to reflect this scenario:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    copy: {
      html: {
        files: [
          {
            src: ['src/index.html'],
            dest: 'dist/index.html'
          }
        ]
      }
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-copy');

  // Default tasks.
  grunt.registerTask('default', ['copy:html']);
};
```

We've set up a sub-task within our `copy` task configuration called `html` which will copy `index.html` for us. This allows us to add the `copy:html` task to our `default` task list. Simply entering `copy` as the task name would have also worked as that would trigger Grunt to execute all of its sub-tasks, but it's preferable to explicitly call the name of the sub-task you wish to execute to prevent any issues in the future when you expand on your configurations.

To test, make sure that the `src/index.html` file exists and then simply run `grunt` from the command line. You should see the following success message:

```
Running "copy:html" (copy) task
Copied 1 files

Done, without errors.
```

To specify more advanced file paths, such as using wildcards and matching on file extensions, read the official documentation on Globbing Patterns within Grunt. Copy also has some useful options such as `encoding` to use a specific file encoding when copying or `mode` to specify copied file permissions.

Concat

Concat lets you concatenate files. Concatenation is the act of joining things together. When optimizing your applications for performance, looking at techniques to reduce the number of requests is extremely important. By concatenating files together, you are able to reduce the number of requests required by clients to get at your code.

To install Concat, we'll run the following from our project folder:

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

Let's say we want to concatenate all of the JavaScript files that we have within the folder `src/js` into a single file. This is how we would go about setting up our `Gruntfile.js`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
      // Concatenate all JavaScript files.
    concat: {
      js: {
        src: ['src/js/*.js'],
        dest: 'dist/scripts.js'
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Default tasks.
  grunt.registerTask('default', ['concat:js']);
};
```

You can test this by adding a bunch of JavaScript files in to the `src/js` folder and then running `grunt`. You may notice that Concat could possibly nullify the need for Copy as you can specify the destination of the concatenated file. However, if you wanted to run other tasks on the concatenated file before moving it to the `dist` folder, you may well still need Copy.

One configuration option of Concat that you can use is `banner`. It allows you to set a string header that will be prepended to the beginning of the concatenated file. It is worth noting that the header string is processed using grunt.template.process.

Here is an example of how to set up a banner in `Gruntfile.js` for a typical application:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    // The banner is placed at the head of our concatenated JavaScript file.
    banner: '/*! \n* <%= pkg.title || pkg.name %> - v<%= pkg.version %>' +
              '\n* Copyright (c) <%= grunt.template.today("yyyy") %> <%= pkg.autho\
r %> ' +
              '\n* <%= pkg.homepage ? pkg.homepage : "" %> ' +
              '\n*/ \n\n',

    // Concatenate all JavaScript files.
    concat: {
      options: {
        banner: '<%= banner %>'
      },
      js: {
        src: ['src/js/*.js'],
        dest: 'dist/scripts.js'
      },
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-concat');

  // Default tasks.
  grunt.registerTask('default', ['concat:js']);
};
```

What this does is load in the `package.json` JSON file into a variable called `pkg` which is then used to fill out `banner`. Depending on how your `package.json` is set up, you should see something like this at the top of the outputted `scripts.js` file:

```
/*!
* myApp - v0.0.0
* Copyright (c) 2014 Joe Fender
* http://www.myapp.com
*/
```

Linting with JSHint and CSSLint

Linting is the process of analyzing code for potential errors. In traditional methods, you would see JavaScript errors in the console if something went wrong. However, by linting your code, you are able to get feedback on your coding quality directly within your Grunt build process. Most linters will give you more information about syntax errors than your traditional browser console. Linting your code is highly recommended as it can save a lot of time debugging syntax errors.

JSHint

For JavaScript, we use grunt-contrib-jshint. To install JSHint, we'll run this from our project folder:

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

This Grunt plugin passes through all of the options you set in its configuration task directly to JSHint. That means you will want to be reading the official JSHint documentation for a list of supported options.

Here is an example of how we go about setting up our own typical JSHint configuration in `Gruntfile.js`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    // Configures JavaScript linting. See http://www.jshint.com/docs/options/
    // for more on the available options.
    jshint: {
      js: {
        files: {
          src: ['src/js/*.js']
        },
        options: {
          curly: true,
          immed: true,
          newcap: true,
          noarg: true,
          sub: true,
          boss: true,
          eqnull: true,
          strict: false,
          globalstrict: true,
          globals: {
            angular: false
          }
        }
      }
    }
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-jshint');

  // Default tasks.
  grunt.registerTask('default', ['jshint']);
};
```

We’ve told JSHint to check all of the JavaScript files in the `src/js` folder as part of our `default` task list. To test this, create an example JavaScript file that has a deliberate syntax error. Such as:

```
var example = {
  bar: 'bar',
  foo: 'foo'
}
```

Run `grunt` and you will see the following:

```
Running "jshint:js" (jshint) task

   src/js/example.js
      4 |}
          ^ Missing semicolon.

>> 1 error in 1 file
Warning: Task "jshint:files" failed. Use --force to continue.

Aborted due to warnings.
```

JSHint found the missing semicolon and gave us a very clear indication of where the syntax error was. Because there was an error, it aborted the Grunt build process.

If you have a build process that takes several seconds to complete, bring your linter task as far towards the front of the list as you can so that you’ll get snappier feedback about possible syntax errors. That way, you’re not wasting Grunt’s or your own time going through a bunch of tasks that you’ll just have to go through again once you fix the syntax error.

CSSLint

For CSS, grunt-contrib-csslint is recommended. As with JSHint, view the CSSLint documentation for a full list of configuration options. To install CSSLint we’ll run this command:

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

Here is an example `Gruntfile.js` to get CSSLint working with default settings:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Configures CSS linting. See https://github.com/CSSLint/csslint
    // for more on the available options.
    csslint: {
      css: {
        src: ['src/css/*.css']
      }
    }
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-csslint');

  // Default tasks.
  grunt.registerTask('default', ['csslint']);
};
```

To test CSSLint create a broken css file in `src/css` - perhaps something like this:

```
.boo {
  height: 100px;
  widths: 100px;
}
```

After running `grunt` you should be greeted with this lovely warning message:

```
Running "csslint:css" (csslint) task
Linting src/css/example.css…ERROR
[L3:C3]
WARNING: Unknown property 'widths'. Properties should be known (listed in CSS3
specification) or be a vendor-prefixed property. (known-properties)
Browsers: All
>> 1 file lint free.

Done, without errors.
```

It is worth noting that unlike JSLint, the Grunt build process will not fail when there is a code warning. This seems to be a design decision.

Minification with UglifyJS and CSSMin

Minification is the process of removing unnecessary characters from code (such as white space and comments) to reduce its file size. There are lots of plugins available that offer different types of minification but a couple of favorites are UglifyJS and CSSMin.

UglifyJS

UglifyJS is a JavaScript compressor and minifier. To install UglifyJS, we'll run this command from our project root:

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

We can set up the `uglify` task in our `Gruntfile.js` with only a few lines:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Minifies our JavaScript files.
    uglify: {
      js: {
        files: {
          'dist/scripts.min.js': ['src/js/*.js']
        }
      }
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-uglify');

  // Default tasks.
  grunt.registerTask('default', ['uglify']);
};
```

In the above example we've configured our `uglify` task to look for all JavaScript files in `src/js` and compress them all into the outputted `dist/scripts.min.js`. By default, UglifyJS will not only remove white space and comments but will also try to simplify the code by grouping together code of similar functionality (such as variable declarations) and by deleting unused functions.

Check out all of the task configuration options available in the plugin README, especially `mangle`. By setting this to `true`, UglifyJS reduces the names of local variables and functions to single letters. This can greatly minimize the amount of text used in the JavaScript file, thus reducing the outputted file size.

CSSMin

CSSMin is probably the most popular CSS minifier Grunt plugin. It uses clean-css to perform the actual compression.

To install CSSMin, we'll run this command from our project root:

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

Here is an example `cssmin` task for our `Gruntfile.js` that will take all of the CSS files in `src/css` and minify them into a single file, `dist/styles.min.css`:

```
module.exports = function(grunt) {
  // Project configuration.
  grunt.initConfig({
    // Minifies our CSS files.
    cssmin: {
      css: {
        files: {
          'dist/styles.min.css': ['src/css/*.css']
        }
      }
    },
  });

  // Load our Grunt plugins.
  grunt.loadNpmTasks('grunt-contrib-cssmin');

  // Default tasks.
  grunt.registerTask('default', ['cssmin']);
};
```

If you look at the UglifyJS example we did previously, you'll notice that they are very similar. Yay for standardization!

Gulp

At the time of writing this book, gulp.js is the new kid on the block. Just like Grunt, Gulp is a task runner. They basically do the same thing. Some of the key differences are:

- Gulp calls itself 'The streaming build system'. It harnesses Node's streams to greatly increase build times. For more on streams, check out the stream handbook.
- Gulp has a simpler API. It has you writing JavaScript code instead of writing configurations in Grunt.
- Gulp plugins are written for one specific task only, whereas Grunt plugins can often do multiple things. You can find a searchable list of Gulp plugins here.

Installing Gulp

Installing Gulp is fairly simple using npm:

```
npm install gulp -g
```

Let's install Gulp globally. Next, navigate to your project root folder (or create one if you haven't already) and make sure you have a `package.json` set up. Don't forget, you can do this by running `npm init` or by referring to the Dependency Management chapter of this book. Then, run the following command:

```
npm install gulp --save-dev
```

This will also install Gulp but this time locally. The important part is that we make a reference to it in the `devDependencies` section of `package.json` by using the `--save-dev` parameter.

Example Usage

To demonstrate how to use Gulp, let's set up a task to uglify JavaScript files using the gulp-uglify plugin. We'll start by installing the pluign:

```
npm install gulp-uglify --save-dev
```

Just as with Grunt, Gulp also has a single file for defining tasks. We'll create a file called `gulpfile.js` in the root of our project and use something like the following snippet to get started:

```
var gulp = require('gulp'),
        uglify = require('gulp-uglify');

```

```
// This task uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

You'll want to create a few dummy JavaScript files in the `src/js` folder to test with. From the command line, run `gulp compress` to run the `compress` task that you defined. You should then see the following status messages:

```
[11:36:18] Using gulpfile ~/myApp/gulpfile.js
[11:36:18] Starting 'compress'...
[11:36:18] Finished 'compress' after 8.87 ms
```

If you look in the `dist` folder, you'll notice that each of the JavaScript files we created have been uglified but unlike the Grunt plugin they have *not* been concatenated. This is because Gulp maintains the ideal that each plugin should only do one thing. So if we wanted to concatenate first and then uglify we would firstly need to install the gulp-concat plugin:

```
npm install gulp-concat --save-dev
```

Then, we could update our `gulpfile.js` to look something like this:

```
var gulp = require('gulp'),
  uglify = require('gulp-uglify'),
  concat = require('gulp-concat');

// This task concatenates and uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(concat('scripts.js'))
    .pipe(gulp.dest('dist'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

Now, once you run `gulp compress`, you will end up with a single concatenated and uglified file at `dist/scripts.js`.

Before moving on from Gulp, we need to know how to set up file watching. Grunt required us to download a separate plugin to do this, whereas Gulp includes watch functionality from the get-go. Let's say we want to watch for any changes to our JavaScript files and run the `compress` task once that occurs. We would need to alter our `gulpfile.js` as shown below:

```
var gulp = require('gulp'),
  uglify = require('gulp-uglify'),
  concat = require('gulp-concat');

// This task concatenates and uglifies our JavaScript files.
gulp.task('compress', function() {
  gulp.src('src/js/*.js')
    .pipe(concat('scripts.js'))
    .pipe(gulp.dest('dist'))
    .pipe(uglify())
    .pipe(gulp.dest('dist'))
});
```

```

// Watch our files for changes.
gulp.task('watch', function() {
  gulp.watch('src/js/*.js', ['compress']);
});

// The default task is used when `gulp` is called from the command line.
gulp.task('default', ['compress', 'watch']);
```

We’ve set up a new task called `watch` which is fairly self-explanatory. Also, to simplify things further, we have defined another new task called `default` which runs our other tasks. So all we need to do now is hop over to the command line and run `gulp`. Now, whenever we make any changes to our JavaScript files, the `compress` task will run automagically.

Gulp is simple and fast. Once you learn its API, the idea of code over config is a very welcome one. We’ve only touched the surface of how powerful Gulp can be. For more information, read the official documentation.

So which one should you be using? Probably Grunt. Gulp is still on the bleeding edge. Its ecosystem needs more time before it should be considered as a replacement for Grunt. That being said, Gulp is an excellent tool and hopefully will continue to develop.

CodeKit

CodeKit is a Mac app created by Bryan Jones. It aims to take a lot of the pain out of setting up an automated workflow by providing an array of the most common and useful features for web developers:

- Has a built in web server which supports live browser refreshing across all of your devices.
- Watches for code file changes and compiles them with linting and minification.
- Has built in Bower support for installing and managing components.

The most appealing part of CodeKit is that it is very easy to get up and running. All you need to do is install the app and it just works. There is no fumbling through configuration files trying to figure out why your build script isn’t working correctly. On top of that, the app’s UI is cleanly designed and a pleasure to use.

Remember though that CodeKit is a Mac only app, so if you are working with other developers you may want to check they can actually run it before diving in.

To find out more, watch these demonstration videos on the CodeKit homepage. Although there is a free trial, the app requires you to purchase a one time license for full use.

Yeoman

Yeoman defines a powerful workflow that helps you kickstart new projects by installing and configuring almost everything you could need to start developing your web applications. It greatly reduces the amount of time you need to write boilerplate code so that you can start developing sooner.

Yeoman provides an ecosystem of ‘generators’ which can be used for setting up the foundations of different types of applications. The generators are opinionated, meaning that although best standards are followed for the most part, they may not be perfectly configured for your specific needs.

There is a reason for leaving Yeoman until last. It can be quite tempting to use a generator to lay the scaffolding for your application without actually knowing everything that it is doing. Ideally, you should learn at least the basics of all the different tools that Yeoman installs and configures before actually using Yeoman. That way, it is much easier for you to configure your application after Yeoman has done its magic.

The Yeoman workflow is comprised of three separately maintained tools: Yo, Bower and Grunt. Yo is the name of the actual scaffolding app from Yeoman, Bower is covered in the Dependency Management chapter of this book and Grunt has been covered at the beginning of this chapter.

Installing Yo

Yo offers web application scaffolding, utilizing scaffolding templates referred to as generators. All we need is npm to install Yo:

```
npm install yo -g
```

Once Yo is installed, you will need to pick a generator best suited to your requirements. As an example let’s install generator-angular, which utilizes a bunch of features that may be required of a typical AngularJS front-end web application workflow:

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

As is the case with Yo, we install generators globally so that they can be accessed from anywhere in the command line. Once the generator is installed, we can create a new directory for our project and from within it run:

```
yo angular
```

You will be prompted with a few questions which will allow you to configure which tools will be installed. Once answered, sit back and relax as Yo takes care of downloading, installing and configuring everything. There is a lot of text scroll during this process so make sure your terminal window is full screen so that people around you think you're working in the matrix.

Once complete, you'll want to spend some time looking through all of the files and folders that have now been placed in your project root. In this particular generator, a whole bunch of tasks have been defined in your `Gruntfile.js` from which you can learn a lot about some of the best practices used in the developer community.

There are literally hundreds of generators available, many of which are unofficial and contributed by the community. You can view the list over at the Yeoman official page.

Other Tips

The goal of this chapter is to help you become the most efficient developer you can be. To help you on your way, here are a few tips and tricks to make your life a little easier.

z

The more projects you work on, the more frustrating it can be to navigate between all of their directories in the command line. Enter 'z', a handy bash script that helps you to jump around directories in the command line. It tracks your most frequently used directories and allows you to navigate them by simply entering part of their name. For example:

```
z myApp
```

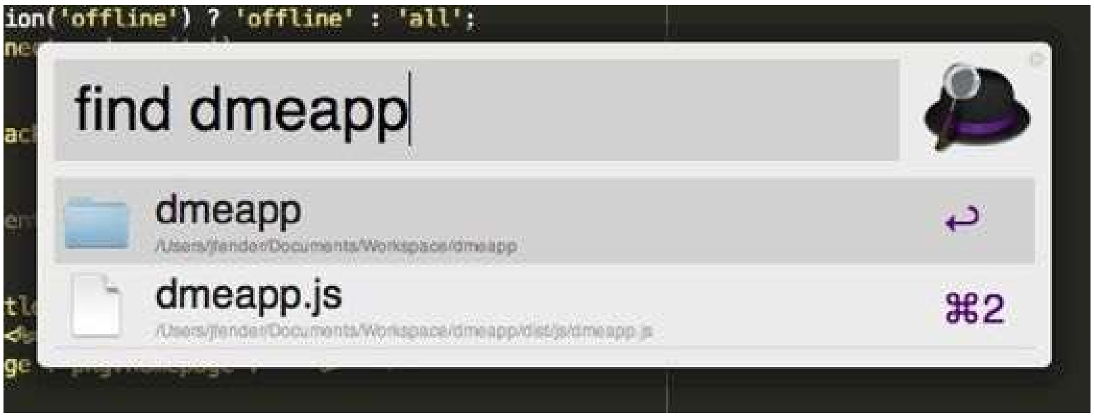
For example, this may take you to `/var/www/myApp` if that is a directory that you frequntly access.

To install, grab the `z.sh` file from https://github.com/rupa/z and write something like this in your `$HOME/.bashrc` or `$HOME/.zshrc`:

```
. /path/to/z.sh
```

Alfred.app

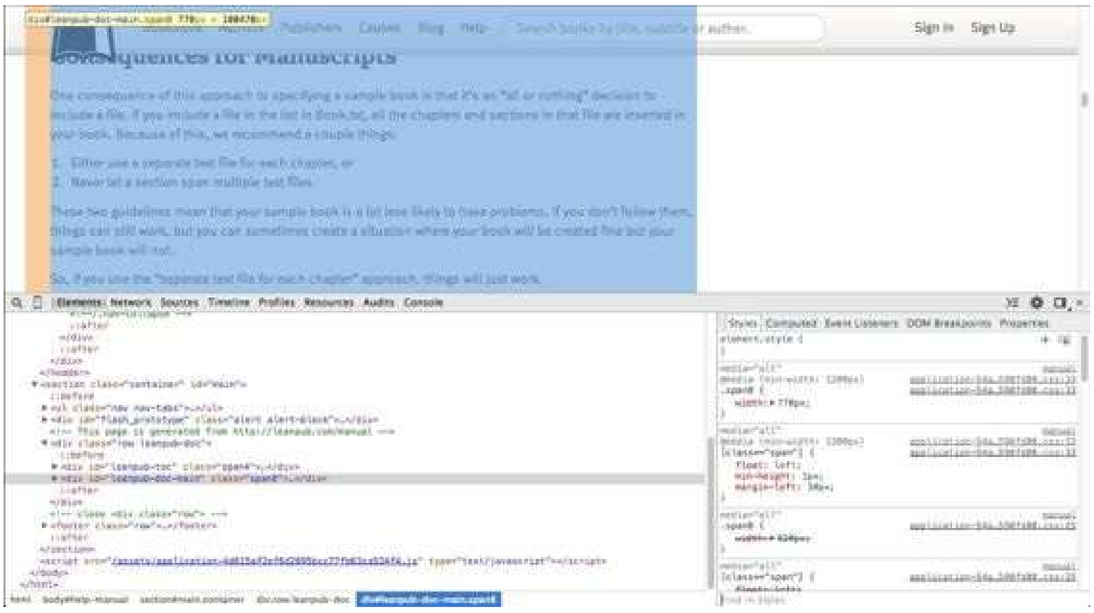
Alfred, perhaps named after the famous butler of Batman, is a Mac app that can be used as a replacement for the OS X Spotlight. Among many other useful productivity tasks, Alfred helps you saves time by letting you search the web or your computer for files and applications with a few keystrokes. Alfred comes highly recommended to any hardcore Mac user. However, it is worth noting that Apple made some great changes to the OS X Spotlight in Yosemite and you may find that sufficient enough.


find dmeapp
dmeapp
/Users/jfender/Documents/Workspace/dmeapp
dmeapp.js
/Users/jfender/Documents/Workspace/dmeapp/dist/js/dmeapp.js
⌘2

Alfred boosts productivity

Chrome DevTools

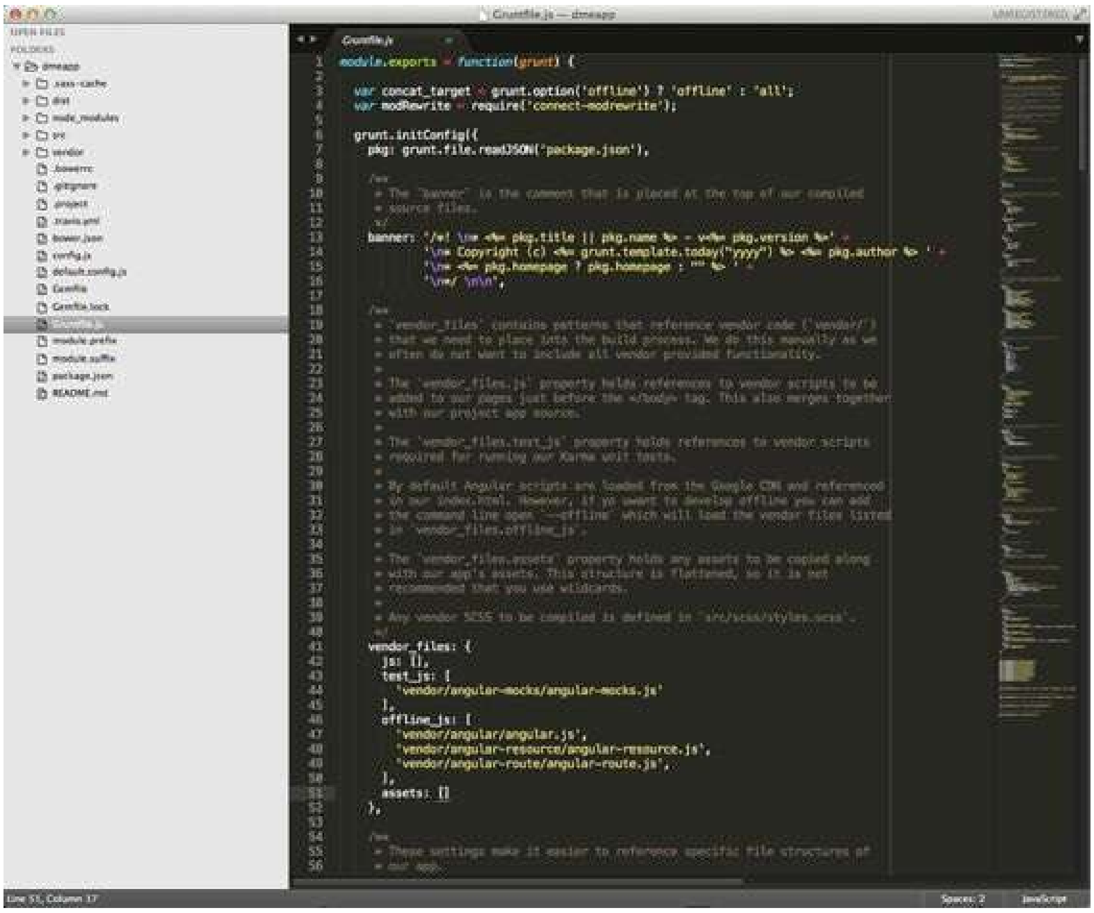
One of the strongest attributes of the Google Chrome browser is its set of Developer Tools - more commonly known as DevTools. They are a set of web debugging tools directly accessible from within your browser. Although other browsers, such as Firefox, do come complete with their own set of tools for developers, Chrome probably provides the best implementation. If you've not used DevTools before, go ahead and install Chrome and read the overview documentation.



Chrome DevTools is the essential web page debugging tool

Sublime Text

Looking for a new text editor? Something with a super slick interface and huge range of features? Look no further. Sublime Text for Mac is quickly becoming one of the most popular text editors for developers.



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

  var concat_target = grunt.option('offline') ? 'offline' : 'all';
  var modRewrite = require('connect-modrewrite');

  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),

    /**
     * The `banner` is the comment that is placed at the top of our compiled
     * source files.
     */
    banner: '/*! \n* <%= pkg.title || pkg.name %> - v<%= pkg.version %>' +
            '\n* Copyright (c) <%= grunt.template.today("yyyy") %> <%= pkg.author %> ' +
            '\n* <%= pkg.homepage ? pkg.homepage : "" %> ' +
            '\n*/ \n\n',

    /**
     * `vendor_files` contains patterns that reference vendor code (`vendor/`)
     * that we need to place into the build process. We do this manually as we
     * often do not want to include all vendor provided functionality.
     *
     * The `vendor_files.js` property holds references to vendor scripts to be
     * added to our pages just before the </body> tag. This also merges together
     * with our project app source.
     *
     * The `vendor_files.test_js` property holds references to vendor scripts
     * required for running our Karma unit tests.
     *
     * By default Angular scripts are loaded from the Google CDN and referenced
     * in our index.html. However, if yo want to develop offline you can add
     * the command line open `--offline` which will load the vendor files listed
     * in `vendor_files.offline_js`.
     *
     * The `vendor_files.assets` property holds any assets to be copied along
     * with our app's assets. This structure is flattened, so it is not
     * recommended that you use wildcards.
     *
     * Any vendor SCSS to be compiled is defined in `src/scss/styles.scss`.
     */
    vendor_files: {
      js: [],
      test_js: [
        'vendor/angular-mocks/angular-mocks.js'
      ],
      offline_js: [
        'vendor/angular/angular.js',
        'vendor/angular-resource/angular-resource.js',
        'vendor/angular-route/angular-route.js',
      ],
      assets: []
    },

    /**
     * These settings make it easier to reference specific file structures of
     * our app.
```

Sublime Text has a super slick interface

Its flagship feature, ‘Goto Anything’, has quickly become one of the main reasons for developers falling in love with Sublime Text. By simply pressing Command-P you can quickly navigate to any file in your project. For a full list of its features, and guidelines on how to install, check out the official homepage.

Summary

In this chapter, we took a fairy detailed look at the task runner Grunt and touched on other options such as Gulp and CodeKit. You’ve seen how important task runners can be, and how they can help you become a more efficient developer. We then looked at a few tips and tricks to speed up your workflow when developing.

Summary

In this book we’ve covered the basics of current front-end development practices all the way from what might be expected of a Front-end Developer in the workplace to automating your development workflow.

Before proceeding to apply the concepts you’ve picked up from this book, make sure to know all your options and pick one that is a best fit for you and your project. You may have found that something in this book conflicts with the way you’re comfortable doing things. That isn’t a problem! The concepts and tools introduced may not be the best fit for every developer or project. So by all means, continue to do what you’re comfortable with! Simply just by reading, you’ve taken an important step in improving your grasp of the front-end landscape.

As with most web technology, the front-end development world changes rapidly. So if you’d like to talk more about anything presented in this book or just share ideas, don’t forget about the Front-End Fundamentals Google Group.

Thank you very much for reading!

Appendix

Below is a condensed list of the external resources that we reference in this book.

Accessibility

- Web Accessibility Checklist: http://a11yproject.com/checklist.html

Automation

- Grunt: http://gruntjs.com
- Gulp: http://gulpjs.com
- Yeoman: http://yeoman.io/
- z.sh: https://github.com/rupa/z
- Alfred: http://www.alfredapp.com/

Deployment

- Continuous Integration: http://www.thoughtworks.com/continuous-integration

Learning Resources

- Front-end Rapport: https://flipboard.com/section/front-end-rapport-bML9IT
- List of popular AngularJS resources: https://github.com/jmcunningham/AngularJS-Learning
- Learning JavaScript Design Patterns: http://addyosmani.com/resources/essentialjsdesignpatterns/book/ by Addy Osmani
- Developing Backbone.js Applications: http://addyosmani.github.io/backbone-fundamentals/ by Addy Osmani
- Responsive Web Design: http://www.abookapart.com/products/responsive-web-design by Ethan Marcotte

Frameworks

- AngularJS: https://angularjs.org
- AngularJS Batarang Chrome Extension: https://chrome.google.com/webstore/detail/angularjs-batarang/ighdmehidhipcmcojjgiloacoafjmpfk?hl=en
- Backbone.js: http://backbonejs.org

- Backbone Debugger Chrome Extension: https://chrome.google.com/webstore/detail/backbone-debugger/bhljhndlimiafopmmhjlgfpnnchjjbhd?hl=en
- Underscore.js: http://underscorejs.org
- Ember.js: http://emberjs.com
- Express: http://expressjs.com/
- Journey Through The JavaScript MVC Jungle: http://www.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/ by Addy Osmani

Languages

- Ruby: https://www.ruby-lang.org/

Online Services

- Prerender.io: https://prerender.io
- Can I use…: http://caniuse.com/#feat=history

Package Managers

- Node Package Manager (npm): https://www.npmjs.org/
- Homebrew: http://brew.sh/
- Bower: http://bower.io
- Bundler: http://bundler.io/

Performance

- Yahoo best practices: https://developer.yahoo.com/performance/rules.html
- YSlow: http://yslow.org
- CSSMin: https://github.com/gruntjs/grunt-contrib-cssmin
- Google’s PageSpeed Insights: https://developers.google.com/speed/docs/insights/rules
- UglifyJS: http://lisperator.net/uglifyjs/

Software

- Node.js: http://nodejs.org/
- CodeKit: http://incident57.com/codekit/
- Scout: http://mhs.github.io/scout-app/
- Sublime Text: http://www.sublimetext.com/

Styling

- BEM: https://bem.info/
- Sass: http://sass-lang.com
- Compass: http://compass-style.org/
- Bourbon: http://bourbon.io/
- Less: http://lesscss.org
- Stylus: http://learnboost.github.io/stylus/
- MDN CSS media queries: https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Media_queries

Templating

- Mustache: http://mustache.github.io
- Handlebars.js: http://handlebarsjs.com

Testing

- Wraith: https://github.com/BBC-News/wraith
- PhantomJS: https://github.com/ariya/phantomjs/
- PhantomCSS: https://github.com/Huddle/PhantomCSS
- Karma: https://github.com/karma-runner/karma
- Jasmine: https://github.com/jasmine/jasmine
- JSHint: http://www.jshint.com
- JSHint Grunt Plugin: https://www.npmjs.org/package/grunt-contrib-jshint
- CSSLint: https://github.com/CSSLint/csslint
- CSSLint Grunt Plugin: https://www.npmjs.org/package/grunt-contrib-csslint
- Chrome DevTools: https://developer.chrome.com/devtools

Version Control

- Github: https://github.com
- git-flow: http://nvie.com/posts/a-successful-git-branching-model/
- Github flow: http://scottchacon.com/2011/08/31/github-flow.html
- Semantic Versioning: http://semver.org/

Web Servers

- MAMP: http://www.mamp.info/en/