Community Experience Distilled

HTML5 iPhone Web Application Development

An introduction to web-application development for mobile within the iOS Safari browser

Alvin Crespo

[PACKT] PUBLISHING

HTML5 iPhone Web Application Development

An introduction to web-application development for mobile within the iOS Safari browser

Alvin Crespo

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

HTML5 iPhone Web Application Development

First published: May 2013

Production Reference: 1170513

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-102-4

www.packtpub.com

Cover Image by Asher Wishkerman (wishkerman@hotmail.com)

Credits

Author
Alvin Crespo

Reviewers
Dale Cruse
Faraz K. Kelhini

Acquisition Editor
Joanne Fitzpatrick

Lead Technical Editor
Neeshma Ramakrishnan

Technical Editors
Amit Ramadas
Neha Shanbhag

Project Coordinator
Arshad Sopariwala

Proofreaders
Paul Hindle
Chris Smith

Indexer
Rekha Nair

Graphics
Ronak Dhruv

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Author

Alvin Crespo is a creative technologist strongly focused on delivering compelling user experiences through the use of frontend technologies. Utilizing the latest industry standards, he strives to move the Web forward promoting open source technologies. Having worked in startup and agency environments, he has helped build and architect complex applications for both medium and large-sized companies.

First and foremost, I would like to thank my lovely wife, Janice Smith, for helping me produce this book. This has only been possible through the love and support you have given me. To my friends and family who have been there throughout the process, my love and endless thanks cannot express how awesome you all are.

About the Reviewers

Dale Cruse is the co-author of *HTML5 Multimedia Development Cookbook* by *Packt Publishing* and the technical editor of several other HTML5 books. He started his career in 1995 as a U.S. Army photojournalist. Since making the switch to purely digital at CBSNews.com, he's created web and mobile experiences for some of the most well-known clients in the world, including 20th Century Fox, Bloomingdale's, and MINI Cooper. Currently, he juggles being both a senior developer at Allen & Gerritsen and being a New York Yankees fan in South Boston. An in-demand speaker, you can't get him to shut up on Twitter at `@dalecruse`.

Faraz K. Kelhini has more than a decade of software development experience in a broad range of disciplines. His core expertise and interest lies in web technologies, including PHP (as well as frameworks like Symfony and Zend framework), Python, HTML5, CSS3, JavaScript (as well as frameworks like jQuery and MooTools), and Linux/Unix operating systems. He is a professional consultant, editor, and writer who specializes in technical presentations, workshops, online content publishing, and knowledge transfer. Faraz has more than 100 articles to his credit within prominent publications such as Developer.com, .net magazine, and Smashing Magazine.

When not pursuing a new technology or idea, Faraz loves practicing his DSLR photography skills. More information on his related writings, presentations, and useful tools can be found at `http://eloux.com`.

I would like to thank you, the reader; I hope that you enjoy this book and produce a fantastic HTML5 iPhone App of your own. I look forward to hearing your feedback and seeing what you come up with! My thanks also go to my entire friends and family.

Table of Contents

Preface

Web applications have come a long way since the 90s, where static HTML pages were delivered to the client. In those days, web pages had a strict client-server model, where much of the processing was done on the server and the client only presented the information with little to no interaction. Such information was only accessible through desktop computers on very slow networks.

Those days have passed, and we are now connected in ways that were impossible before. From cellphones that can still make calls on the subway, to tablets presenting the latest article from your favorite newspaper 30,000 feet in the air; we are now in a digital age where information is easily accessible through innovative technologies. Yet, we still struggle to create a seamless interaction between technology and the physical world.

Although we have devices that are sensitive to our touch, can detect our location, and have the ability to monitor our vital signals, it is still up to us to make applications that will change the world. The hard work that goes into creating these applications usually requires large teams, complex business roles, and costly expenses.

For a brief period of time, developing these applications presented a challenge to many entrepreneurs who were looking to drive change. A staggered mobile market, which continues to this day, contributed to limited development resources. What we saw was an increase in the advancement of these technologies, but very few people who understood or were even interested in learning all the languages felt the necessity to create a cross-platform application.

However, it was only a matter of time until a single platform would arrive and change the world forever. HTML5, and its implementation across devices, helped drive the force necessary to deliver a platform that allowed developers to innovate and change the world. Leveraging this technology in our applications allows us to push the limit of the hardware while creating something that many users can enjoy, no matter what device they prefer to use.

Over the years, I have come to realize that device agnostic applications will become the norm. We have seen competitors adopt these standards with little to no impact on their success; in fact, it can be argued that it has done the opposite. For these reasons, this book was written to provide you with the techniques to create applications, based on open standards, and for the successful creation of device agnostic software.

What this book covers

Chapter 1, *Application Architecture*, helps you to learn how to create a standard architecture for iPhone web application development. We will customize the standard HTML5 Mobile Boilerplate for our needs throughout the book.

Chapter 2, *Integrating HTML5 Video*, helps you to learn the basics of implementing an HTML5 video player within your web application. We'll review the specification and implement an exposed API to tap into.

Chapter 3, *HTML5 Audio*, explains an implementation of the HTML5 Audio API. We'll create an audio player that utilizes the same principles from Chapter 2 to create a reusable component.

Chapter 4, *Touch and Gestures*, helps you to learn about touch and gesture events, including the similarities and differences. We'll go over a couple of examples and more importantly, the specification to properly integrate our application's user experience.

Chapter 5, *Understanding HTML5 Forms*, explains the new features in HTML5 forms, ultimately understanding its uses for our iOS web applications. We'll review the new inputs, their interactions, and the behaviors expected from the iOS operating system.

Chapter 6, *Location-aware Applications*, will have Geolocation as the key point, from the specification to the full implementation in the Safari iOS browser. We'll create an example that utilizes this feature and demonstrate how we can utilize it in our own applications.

Chapter 7, *One-page Applications*, is jam-packed with information on how to create a seamless experience in your application. We'll go over the principles of the MVC design pattern and create an example that utilizes its full potential.

Chapter 8, *Offline Applications*, will cover key topics such as Caching, History, and local storage. The essentials will be covered and the details exposed in order for us to create true offline applications.

Chapter 9, Principles of Clean and Optimized Code, will have us sidestepping the development process to refine our craftsmanship. We'll go over best practices, industry supported techniques, and ways to improve our code for the overall benefit of our applications.

Chapter 10, Creating a Native iPhone Web Application, reviews how we can create the native application we have learned previously. Applying the same techniques we'll create native applications based on open standards.

What you need for this book

This book aims to provide web application development solutions specifically for iOS. With that in mind, you will need an iPhone and/or iPad, preferably an Apple computer with Mac OS X 10.8 and above. You will definitely need a text editor or IDE of your choice, including Xcode 4 and above with the iOS simulator installed. And ultimately, you'll be testing your applications in the most modern web browsers, including Safari.

Who this book is for

This book is intended for beginner to intermediate level developers who are diving into web application development specifically for iOS. This book begins with introductory level material, with each chapter advancing over each topic. The topics covered will give you a good idea on how to approach the development process and the steps needed to achieve those goals.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "Although we've written this code previously, let's briefly review the structure of the `MediaElement` class."

A block of code is set as follows:

```
<div class="audio-container">
    <audio controls preload>
        <source src="../assets/nintendo.mp3" type='audio/mpeg;
codecs="mp3"'/>
        <p>Audio is not supported in your browser.</p>
    </audio>
    <select>
        <option value="sample1.mp3" selected>Sample1</option>
        <option value="sample2.mp3">Sample2</option>
        <option value="sample3.mp3">Sample3</option>
    </select>
</div>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Then download the zip file, by clicking on the **Download as zip** button."

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Application Architecture

In this chapter, we will create a standard architecture for our iPhone application. We will base it on the HTML5 Mobile Boilerplate and customize it for the needs of the several projects in this book. From marking up our content in HTML5 to creating a JavaScript framework, we'll create static pages that help us focus on the foundations of iPhone Web Application development.

In this chapter, we will cover:

- Implementing the HTML5 Mobile Boilerplate
- Creating a preliminary architecture
- Customizing our framework
- Creating semantic markup
- Structuring our stylesheets
- Responsive design principles
- Establishing our JavaScript architecture
- Routing to a mobile site
- Home screen icons
- Introducing our build script
- Deploying our project

Implementing the HTML5 Mobile Boilerplate

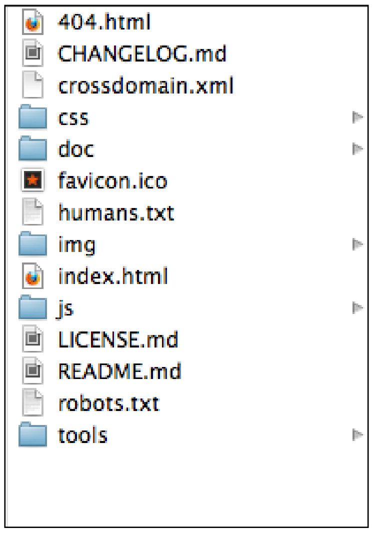
When you begin development, it is always critical to start with a basic framework that can be molded to the needs of your project. In many cases, we develop these frameworks in-house where we work, or perhaps for our own personal projects. However, the open source community has presented us with a great framework we can use in our projects—the HTML5 Mobile Boilerplate. This framework is based on the well-known HTML5 Boilerplate, and has been optimized for mobile including a lean HTML template; the utilization of `Zepto`, and use of tools and helpers optimized for mobile.

Downloading and installing the HTML5 Mobile Boilerplate

The first step we need to take is to download the HTML5 Mobile Boilerplate, which is located here:

`http://html5boilerplate.com/mobile/`

Once the boilerplate is downloaded, you should see the following structure from the unzipped archive file:



```
404.html
CHANGELOG.md
crossdomain.xml
css
doc
favicon.ico
humans.txt
img
index.html
js
LICENSE.md
README.md
robots.txt
tools
```

The Preliminary Directory Structure

The next step is to take these files and place them in the directory of your choice. For example, I have placed my files in the following directory on my Mac:

```
/Users/alvincrespo/Sites/html5iphonewebapp
```

Next, we'll want to use a build system that helps us create multiple environments, ease the deployment process, and overall make things easier when we want to optimize our site for testing and/or production.

According to the documentation for the HTML5 Mobile Boilerplate, there are two different types of build system, such as the Node Build script and the Ant Build script. In this book, we'll be using the Ant Build script. I would recommend using the Ant Build script since it has been around for a while and has the appropriate features that I use in my projects, including CSS Split, which will help split up the main CSS file that comes with the boilerplate.

Integrating the build script

To download the Ant Build script, go to the following link:

```
https://github.com/h5bp/ant-build-script
```

Then, download the zip file by clicking on the **Download as zip** button. When you have downloaded the Ant Build script, copy the folder and its contents to your project.

Once your Ant Build script directory is fully transferred over to your project, rename the directory containing the build script to `build`. At this point, you should have your project completely set up for the rest of the applications in this book. We will cover how to utilize the build script later on in this chapter.

Creating our application framework

With every project, it's important to create a framework that adjusts to your project's needs. It's critical to think about every aspect of the project. From the required document to the team's strengths and weaknesses, it's important we establish a solid foundation that helps us build and adjust accordingly.

Modifying the boilerplate

We'll now modify our boilerplate for the needs of the projects we will be building. For simplicity, we'll remove the following items from the folder:

- `CHANGELOG.md`
- `crossdomain.xml`
- `README.md`
- `/doc (Directory)`

Now that the directory has been cleaned up, it's time to take a look at some of the boilerplate code and customize it for the needs of the projects in this book.

Customizing our markup

First, open up the application in your favorite text editor. Once we've opened up the application in the editor of our choice, let's look at `index.html`.

The index file needs to be cleaned up in order to focus on iPhone Web Application development, and also unused items such as Google Analytics need to be removed. So let's remove some code that is not necessary for us.

Look for the following code:

```
<!DOCTYPE html>
<!--[if IEMobile 7 ]>    <html class="no-js iem7"> <![endif]-->
<!--[if (gt IEMobile 7)|!(IEMobile)]><!--> <html class="no-js">
<!--<![endif]-->
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

And modify it to this:

```
<!DOCTYPE html>
<html class="no-js">
```

What we've done here is removed detection for IE Mobile. Although this may be helpful for other projects, for us it doesn't really help in creating a fully compatible application just for the iPhone. However, we also need to remove an `IEMobile` specific meta tag:

```
<meta http-equiv="cleartype" content="on">
```

The previous meta tag turns on `cleartype` (a utility that assists with the rendering of fonts) for the IE mobile. This isn't necessary for us and is not a requirement for our applications.

Now that we've removed some unnecessary markup from our page, we can go ahead and start enabling features that will enhance our application. Look for the following meta tags and enable them, by removing the comments surrounding them:

```
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
```

These directives inform our application that it can run in fullscreen and they set the status bar to black.

We can also remove the following code from the `<head>` of the document:

```
<!-- This script prevents links from opening in Mobile Safari.
https://gist.github.com/1042026 -->
<!--
        <script>(function(a,b,c){if(c in b&&b[c]){var d,e=a.
location,f=/^(a|html)$/i;a.addEventListener("click",function(a){d=a.
target;while(!f.test(d.nodeName))d=d.parentNode;"href"in d&&(d.href.
indexOf("http")||~d.href.indexOf(e.host))&&(a.preventDefault(),e.
href=d.href)},!1)}})(document,window.navigator,"standalone")</script>
-->
```

Once we've removed the previous script, your markup should now look like the following:

```
<!DOCTYPE html>
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
    <meta name="HandheldFriendly" content="True">
    <meta name="MobileOptimized" content="320">
    <meta name="viewport" content="width=device-width">
    <link rel="apple-touch-icon-precomposed" sizes="144x144"
href="img/touch/apple-touch-icon-144x144-precomposed.png">
```

```
    <link rel="apple-touch-icon-precomposed" sizes="114x114"
href="img/touch/apple-touch-icon-114x114-precomposed.png">
    <link rel="apple-touch-icon-precomposed" sizes="72x72" href="img/
touch/apple-touch-icon-72x72-precomposed.png">
    <link rel="apple-touch-icon-precomposed" href="img/touch/apple-
touch-icon-57x57-precomposed.png">
    <link rel="shortcut icon" href="img/touch/apple-touch-icon.png">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-status-bar-style"
content="black">
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/main.css">
    <script src="js/vendor/modernizr-2.6.1.min.js"></script>
</head>
```

Now, we can focus on cleaning up our body. Lucky for us, we only need to remove one thing—Google Analytics, since we will not be focusing on tracking for iPhone Web Apps.

To do this, find the following code and remove it:

```
<!-- Google Analytics: change UA-XXXXX-X to be your site's ID. -->
<script>
    var _gaq=[["_setAccount","UA-XXXXX-X"],["_trackPageview"]];
    (function(d,t){var g=d.createElement(t),s=d.
getElementsByTagName(t)[0];g.async=1;
    g.src=("https:"==location.protocol?"//ssl":"//www")+".google-
analytics.com/ga.js";
    s.parentNode.insertBefore(g,s)}(document,"script"));
</script>
```

The only scripts that you should have on the page should be the following:

```
<script src="js/vendor/zepto.min.js"></script>
<script src="js/helper.js"></script>
```

Once we've completed the previous steps, our markup should be clean and simple as follows:

```
<!DOCTYPE html>
<html class="no-js">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
    <meta name="HandheldFriendly" content="True">
```

```
        <meta name="MobileOptimized" content="320">
        <meta name="viewport" content="width=device-width">

        <link rel="apple-touch-icon-precomposed" sizes="144x144"
    href="img/touch/apple-touch-icon-144x144-precomposed.png">
        <link rel="apple-touch-icon-precomposed" sizes="114x114"
    href="img/touch/apple-touch-icon-114x114-precomposed.png">
        <link rel="apple-touch-icon-precomposed" sizes="72x72" href="img/
    touch/apple-touch-icon-72x72-precomposed.png">
        <link rel="apple-touch-icon-precomposed" href="img/touch/apple-
    touch-icon-57x57-precomposed.png">
        <link rel="shortcut icon" href="img/touch/apple-touch-icon.png">

        <meta name="apple-mobile-web-app-capable" content="yes">
        <meta name="apple-mobile-web-app-status-bar-style"
    content="black">

        <link rel="stylesheet" href="css/normalize.css">
        <link rel="stylesheet" href="css/main.css">
        <script src="js/vendor/modernizr-2.6.1.min.js"></script>
    </head>
        <body>

            <!-- Add your site or application content here -->

            <script src="js/vendor/zepto.min.js"></script>
            <script src="js/helper.js"></script>
        </body>
    </html>
```

From here, we should examine our stylesheets and scripts for every project and optimize it as much as we can prior to beginning a project. However, this boilerplate that we will be using has been optimized by the community and continuously enhanced with support from many developers, and for our use here, both styles and scripts are good to go. If you are curious, I encourage you to look at the `normalize.css` file, which contains excellent directives for resetting a page. It would also be beneficial to review the `main.css` file that has been enhanced with this boilerplate to support mobile devices.

Now, we'll move on to establishing our framework.

Customizing our framework

It's critical for developers to establish a framework for each project they are working on, no matter how small or big the project may be. Of course, your framework should adjust to the requirements that the project demands as well. In this section, we'll establish a simple framework that we can work with throughout the use of this book.

We've gone through and cleaned up the boilerplate for our needs, now we'll go through and expand upon the boilerplate to include the files that are critical to the applications we will build.

The first application will be based on the HTML5 Video specification (`http://dev.w3.org/html5/spec-author-view/video.html`). In that application we'll create a specific functionality for our video player that includes play, pause, and fullscreen functionalities. So let's create a directory specific to this application; we'll call this directory `video`.

In this directory, we'll create an `index.html` file and copy the contents from the homepage of the `index.html` file.

Now that we have our video section created, let's create the `video.css` file inside of our `css` directory.

Then, create an `App` directory within our `/js` folder. Within the `/js/App` directory, let's create an `App.js` file. Later, we'll explain in detail what this file is, but for now it will be our main application namespace that will essentially encapsulate global functionality for our application.

Finally, let's create an `App.Video.js` file that will contain our video application functionality within the `/js/App` directory.

You will now repeat the previous steps for each of our applications; including Video, Audio, Touch, Forms, Location, Single Page, and Offline. In the end, your directory structure should have the following new directories and files:

```
/audio
    index.html
/css
    audio.css
    forms.css
    location.css
    main.css
    normalize.css
    singlepage.css
```

```
        touch.css
        video.css
    /forms
        index.html
    /js
        /App/App.Audio.js
        /App/App.Forms.js
        /App/App.js
        /App/App.Location.js
        /App/App.SinglePage.js
        /App/App.Touch.js
        /App/App.Video.js
    /location
        index.html
    /offline
        index.html
    /singlepage
        index.html
    /touch
        index.html
    /video
        .index.html
```

At this point, we should fix the references to our dependencies, such as our JavaScript and stylesheet. So let's open up `/video/index.html`.

Let's modify the following lines:

```
<link rel="stylesheet" href="css/normalize.css">
<link rel="stylesheet" href="css/main.css">
<script src="js/vendor/modernizr-2.6.1.min.js"></script>
```

Change the previous markup to the following:

```
<link rel="stylesheet" href="../css/normalize.css">
<link rel="stylesheet" href="../css/main.css">
<script src="../js/vendor/modernizr-2.6.1.min.js"></script>
```

Note that we add `../` to each dependency. This is essentially telling the page to go up one level and retrieve the appropriate files. We also need to do this for the apple-touch-icon-precomposed links, shortcut icon, and the scripts at the bottom of the page.

Our framework is now almost complete, except that they aren't connected yet. So now that we've got everything organized, let's start hooking up everything to one another. It won't look pretty, but at least it will be working and moving towards a fully functional application.

Let's start with the main `index.html` file, `/ourapp/index.html`. Once we've opened up the main `index.html` file, let's create a basic site structure inside our `<body>` element. We'll give it a class of `"site-wrapper"` and put it right below the comment `Add your site or application content here`:

```
<body>
    <!-- Add your site or application content here -->
    <div class="site-wrapper">

    </div>
    <script src="js/vendor/zepto.min.js"></script>
    <script src="js/helper.js"></script>
</body>
```

Within the wrapper containing our site, let's use the new HTML5 `<nav>` element to semantically describe the main navigation bar that will exist across all our apps:

```
<div class="site-wrapper">
<nav>
</nav>
</div>
```

Nothing too special yet, but now we'll go ahead and use the unordered list element and create a navigation bar with no styling:

```
<nav>
    <ul>
        <li>
            <a href="./index.html">Application
Architecture</a>
        </li>
        <li>
            <a href="./video/index.html">HTML5 Video</a>
        </li>
        <li>
            <a href="./audio/index.html">HTML5 Audio</a>
        </li>
        <li>
            <a href="./touch/index.html">Touch and Gesture
Events</a>
```

```
        </li>
        <li>
            <a href="./forms/index.html">HTML5 Forms</a>
        </li>
        <li>
            <a href="./location/index.html">Location Aware
Applications</a>
        </li>
        <li>
            <a href="./singlepage/index.html">Single Page
Applications</a>
        </li>
    </ul>
</nav>
```

If we copy the code that we have created in `/video/index.html` and test the page, you see that it will not work correctly. For all subdirectories, like video and audio, we'll need to change the relative path from `./` to `../` so that we can go up one folder. With this in mind, the `nav` element would look like the following within the other applications:

```
<nav>
    <ul>
        <li>
            <a href="../index.html">Application
Architecture</a>
        </li>
        <li>
            <a href="../video/index.html">HTML5 Video</a>
        </li>
        <li>
            <a href="../audio/index.html">HTML5 Audio</a>
        </li>
        <li>
            <a href="../touch/index.html">Touch and Gesture
Events</a>
        </li>
        <li>
            <a href="../forms/index.html">HTML5 Forms</a>
        </li>
        <li>
            <a href="../location/index.html">Location Aware
Applications</a>
        </li>
```

```
            <li>
                <a href="../singlepage/index.html">Single Page
    Applications</a>
            </li>
        </ul>
    </nav>
```

Now, we can copy the navigation from `/video/index.html` to the rest of the application files or to the `index.html` files we created previously. Once this is done, we will have a single site that now connects well with each other.

Believe it or not, we have a very simple website going on here. Our pages are set up with basic markup and general styles. At this point, we need a navigation that brings our pages together. However, we've barely touched on some important aspects, including semantic markup for applications, which we'll discuss next.

Creating semantic markup

Semantic markup is important for several reasons, including search engine optimization, creating maintainable architectures, making code easily understandable, and meeting accessibility requirements. However, you should be familiar with structuring your page with markup that is related to your content. There are new elements within the HTML5 specification that help to ease this process, including the `<header>`, `<nav>`, `<footer>`, `<section>`, `<article>`, and `<aside>` elements. Each one of these elements helps describe the aspects of a page and easily identifies components of your application. In this section, let's structure our applications, beginning with our Video application.

Creating the header

First, let's start by giving our main index page a title and a header that describes the page we are on. Let's open the main `index.html` file in our application at `/index.html`.

Find the `<title>` tag and enter it in `iPhone Web Application Development – Home`. Note that we use a hyphen here. This is important since it makes it easier for users to scan the content of the page and helps with the ranking for specific keywords.

You should now have the following `<title>` in the `<head>` tag of your document:

```
<title>iPhone Web Application Development - Home</title>
```

Now we want the content of the page to reflect the title as well and alert the user of their progress on our site. What we want to do is create a header that describes the section they are on. In order to achieve this, let's place the following code before the navigation we created previously. Your code should then look like this:

```
<hgroup>
    <h1>iPhone Web Application Development</h1>
    <h2>Home</h2>
</hgroup>
<nav>...</nav>
```

The `<hgroup>` element is used to group multiple headers for a section. The rank of the headers is based on `<h1>` to `<h6>`, with `<h1>` being the highest rank and `<h6>` the lowest. Therefore, the highlighted text places our `<h1>` content higher than our `<h2>`.

Also note that we are not using the `<section>` element yet. However, this page does validate using the W3C Markup Validation Service (`http://validator.w3.org/`).

We can further describe the page by wrapping our `<hgroup>` and `<nav>` elements within a `<header>` element to give the page an introductory aid. Once you do this, you should have the following code:

```
<header>
    <hgroup>... </hgroup>
    <nav>... </nav>
</header>
```

With the previous code, we have finally given our page some structure. We are describing our page with a main header for the site and a sub header for the page. We have also given the page a navigation menu, allowing the user to navigate across applications.

Creating the footer

Now let's add a `<footer>` that contains the name of this book with its copyright date:

```
<footer>
    <p>iPhone Web Application Development &copy; 2013</p>
</footer>
```

The previous code will basically relate to the nearest sectioning ancestor. Thus the footer will relate to the content before it, which we will fill in a bit later. At this point, your content should look like this:

```
<div class="site-wrapper">
    <header>
        <hgroup>...</hgroup>
        <nav>...</nav>
    </header>
    <footer>...</footer>
</div>
```

Clearing up section

You may be wondering why we are not using the `<section>` element right away for the `<div>` element that contains both the `<header>` and `<footer>` element. In this case, it's not necessarily useful since we are not creating a page where the element's contents would be listed in an outline. This is the suggestion by the W3C and is something every developer should be aware of when deciding which element to use, `<div>` or `<section>`. In the end, it comes down to the content itself and the outline the team wishes to create.

Now that we have a basic structure for our pages, we can go ahead and do the same for the rest of our applications. This will be done for you in the code provided with this book in case you wish to review a final version.

With this in mind, we will move forward with our application development, making sure that we use semantic code when and where it makes sense.

Structuring our stylesheets

Styling is extremely important in any application we build, especially since it is the first aspect of any application the user experiences. In this section, we'll start structuring our styles appropriately.

Global styling

First, let's open our `main.css` file, located in the `css` directory. When you open this file, you'll see default boilerplate styles. At this point, let's skip through these to create our own styles. We'll review those styles as we continue to develop our applications.

Find the following line in `main.css`:

```
/* =====================================================================
========
   Author's custom styles
=====================================================================
===== */
```

It's after this comment that we want to include the global styles for the semantic code we wrote previously.

Start by defining the global site styling such as the background color:

```
html{
    background: #231F20;
    border-top: 10px solid #FDFF3A;
    border-bottom: 5px solid #FDFF3A;
    width: 100%;
}
```

In the previous styling, we are making some stylistic choices like setting our background color and some borders. The important part here is that the width is defined at 100 percent for the HTML element. This will basically allow us to extend to 100 percent of the width of the phone for all our content.

Defining our global fonts

We then have to define overall fonts on the page. This will be basic for now and can continue to extend as design as per our application, but for now take a look at the following styles:

```
h1, h2, p, a {
    font-family: Arial, Helvetica, sans-serif;
    text-decoration: none;
}
```

```
h1, h2 {
    color: #A12E33;
    font-weight: bold;
    margin: 0;
    padding: 0;
}

h1 {
    font-size: 18px;
}

h2 {
    font-size: 14px;
    font-weight: normal;
}

p {
    color: #F15E00;
    font-size: 12px;
}

a,
a:visited {
    color: #F19C28;
}
```

In the previous code, you can see that we are working from a higher level down, the essential understanding of Cascading Style Sheets. We first define our headers, anchors, and paragraphs by using a specific font family and having no decoration.

As we work down the previous styles, we start to define each one more specifically, with headers having no padding or margins and a specific color. Then, when we go down further, we can see that each type of header has a specific font size and we do the same for paragraphs and anchors.

Our page layout

Once we've defined some of our fonts and site styling, we include some basic layout information for the `<div>` element containing our content:

```
.site-wrapper {
    padding: 5px 10px 10px;
}
```

Since our element automatically scales to 100 percent of the width of the screen, we tell the content to have a padding of `5px` at the top, `10px` at the left and right, and `10px` on the bottom. Alternatively, we could have written the following styles:

```
    padding-top: 5px;
    padding-left: 10px;
    padding-right: 10px;
    padding-bottom: 10px;
```

The former is known as a shorthand property setting and is considered best practice.

Using content with :before and :after

Since we also want to make sure our second header is differentiated in some form, we can use a CSS3 pseudo class selector and property to define the before and after content, as following:

```
hgroup h2:before,
hgroup h2:after {
    content: " :: ";
}
```

> Keep in mind that the `:before` and `:after` pseudo selectors are supported in Safari 3.2 and above.

The previous selector targets the `<h2>` elements within the `<hgroup>` element and appends the content we have defined in the property before and after it, as per the `:before` and `:after` pseudo class selector.

Styling our navigation

Next, let's style our navigation to look and feel a bit more useable.

```
nav ul {
    padding: 0;
}

nav li {
    list-style: none;
}
```

```
nav a {
    display: block;
    font-size: 12px;
    padding: 5px 0;
}
```

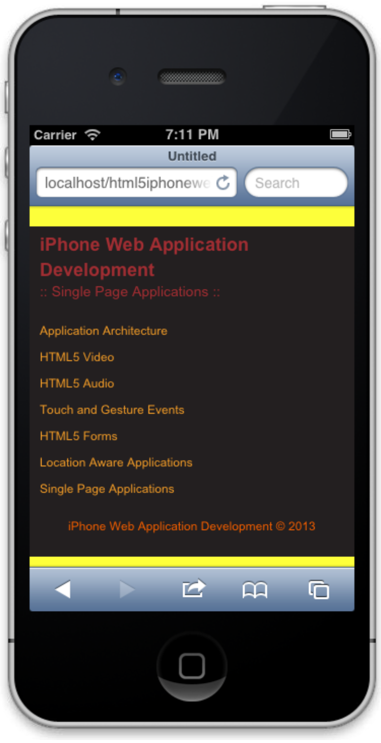
Here we remove the padding off the `<ul>` element and then remove the default styling option from each list element. Finally, we make sure each anchor is displayed correctly by setting the font size to `12px` and add padding to the top and bottom of each anchor to allow for easy selection on the iPhone.

Finally, we'll add some styling to our footer.

```
footer p {
    text-align: center;
}
```

Very simply, we're aligning the paragraph within the footer to center. Since we've defined the default styles for our paragraph in our fonts section, the styling gets picked.

When the previous styles are applied properly, your result should be similar to the following display:



iPhone Web Application Development

:: Single Page Applications ::

Application Architecture

HTML5 Video

HTML5 Audio

Touch and Gesture Events

HTML5 Forms

Location Aware Applications

Single Page Applications

iPhone Web Application Development © 2013

Responsive design principles

Responsive design is the key to our mobile applications. Given the fact that many mobile experiences now surpass those viewed on desktop, it is essential we create applications that fit our evolving technological landscape. Lucky for us, the HTML5 Mobile Boilerplate comes with preliminary styles that we can modify.

Media queries to the rescue

First, let's open up our `main.css` file in our `css` directory.

Next, scroll down to the bottom of the file and you should see the following styling:

```
/* ======================================================================
========
   EXAMPLE Media Queries for Responsive Design.
   Theses examples override the primary ('mobile first') styles.
   Modify as content requires.
======================================================================
===== */

@media only screen and (min-width: 800px) {
}

@media only screen and (-webkit-min-device-pixel-ratio: 1.5),
       only screen and (min-resolution: 144dpi) {}
```

Although this styling gets us off the ground, for iPhone development, we need some more customization. The first media query is specific for tablet devices, and the second media query helps us by targeting devices with higher resolution, such as the iPhone 4.

What we want to do is make this a bit simpler. Since we are only targeting iPhones, this is what we can replace the previous code with:

```
/* iPhone 4 and 5 Styles*/
@media only screen and (-webkit-min-device-pixel-ratio: 2) { }
```

The previous code will target both the iPhone 4 and 5. We specifically target these two devices by checking the `–webkit-min-device-pixel-ratio` property on the device, and if it is true it means we can serve high definition graphics.

Another aspect we want to check is our viewport settings in the `index.html` pages we've set up. Luckily, we cleaned this up earlier and it should have the following:

```
<meta name="viewport" content="width=device-width">
```

The previous code snippet will basically resize our content based on the width of the device.

At this point, we should be set for implementing responsive styling later on in our applications. Now that our styling is set for our applications and is general enough to expand upon, let's start adding the framework behind the scripts.

Responsive images

Images are an extremely important part of any application. It helps showcase the features of a product and exemplifies information you want the user to understand. However, today's varying amount of devices require content to respond correctly. On top of that, we need to be able to deliver content that is appropriate for the experience, meaning we need to tailor to higher resolution devices so that the highest quality content reaches that audience.

There are multiple techniques for delivering the appropriate content. However, the one you choose depends on the requirements of your project. In this part, we'll review the traditional responsive web design principle of resizing an image according to its content and/or container.

Fluid images

In this technique, the developer sets all the images to a maximum width of 100 percent. We then define the container of the image to adjust accordingly.

Fluid width images

To achieve full width images, we can do the following:

```
<body>
<img src="img/batman.jpeg" alt="Its Batman!">
</body>
```

The markup is pretty simple, we essentially wrap an image into an element that extends the full width of what we need. In this case, the body will extend 100 percent in width.

Next, we'll define the style of the image as follows:

```
img {
    max-width: 100%;
}
```

With this simple CSS declaration, we are telling our images to have their maximum width set to 100 percent of the containing content. This will automatically resize the image as the device's width changes, which is essential if we want to make sites responsive to the user's device.

Full width images

In this case, we want the image to stay its full width, but we also need it to cut off accordingly.

To achieve this, we can start by simply creating a `div` with a `class`, in this case we add a class of `overflow`:

```
<div class="overflow"></div>
```

We can then create the styling that keeps the image at full width and cuts off based on the resizing of the content:

```
overflow {
    background: transparent url('img/somgimg.jpg') no-repeat 50% 0;
    height: 500px;
    width: 100%;
}
```

This is a bit complex, but essentially we attach the image with a `background` property. The key here is to make sure we center it using 50 percent. The height property is just to show the image, and the width tells the container to be 100 percent related to its content.

These are the two techniques we use when implementing a traditional responsive design. We'll be implementing these techniques much later when we create the video and image galleries.

Establishing our JavaScript architecture

When establishing a JavaScript architecture for your application, there's a lot to think about, including possible changes in the near or short term, security, ease of use and implementation, documentation, and more. Once we can answer the various questions we have, we can then decide on the pattern (module, facade and/or mediator, and so on). We also need to know what library or framework would be best suited for us, such as `jQuery`, `Zepto.js`, `Backbone.js`, or `Angular.js`.

Luckily for us, we'll be keeping it plain and simple in order to deliver an effective application on an iPhone. We'll be utilizing `Zepto.js` as our supported library to keep it light. We'll then build upon Zepto by creating a custom JavaScript framework that follows a modular pattern.

Structuring our app functionality

First, let's open up our application directory in our preferred text editor.

Next, open the `App.js` file we created earlier within our JavaScript directory. The `App.js` file should be completely empty, and it shouldn't be included anywhere. This is where we will begin writing our framework.

Namespacing our application

If you're new to JavaScript, you have most likely created most of your code in the global scope—perhaps laying out most of your JavaScript inside of script tags. Although this may achieve some of your goals, when working on large scale applications we want to avoid such practices. Some of the reasons we want to namespace our applications is for maintainability, efficiency, and portability.

Let's start out by checking for the `App` namespace; if it exists we'll use what's there, if it does not exist, then we'll make an empty object. The following code shows how we can achieve this:

```
var App = window.App || {};
```

Immediately Invoked Function Expressions

Great! We are checking for the `App` namespace, now let's define it. Let's include the following code after the check:

```
App = (function(){}());
```

The previous code is doing several things, let's take it one step at a time. First, we're setting the `App` namespace to what is known as an **Immediately Invoked Function Expression** (**IIFE**). We are essentially creating a function that is wrapped by parentheses and immediately invoking it after the closing brace.

When we use the previous technique, or IIFE, we create a new execution context or scope. This helps in creating self-containing code that will hopefully, not impact other code on the site. It protects us and helps us follow the modular pattern efficiently.

Let's extend the previous functionality by passing in the window, document, and Zepto objects, as follows:

```
App = (function(window, document, $){
}(window, document, Zepto));
```

I know that this may be a bit confusing, but let's take a second to think through what we're doing here. First, we are setting some parameters in the function named `window`, `document`, and `$`. Then, we are passing in `window`, `document`, and `Zepto` when we invoke this method. Remember, we discussed previously that this creates a new scope or execution context? Well, this becomes useful to us because we can now pass in references to any object that might be global.

How is this useful to us? Well, imagine if you wanted to use the actual `Zepto` object over and over again it would be kind of tiring. It's not that difficult to type `Zepto`, but you can just namespace it to the dollar sign and keep it simple.

Use strict

Ok, so we've got our module setup. Now let's continue to extend it by including the `use strict` directives:

```
App = (function(window, document, $){
    'use strict';
}(window, document, Zepto));
```

This directive helps us debug our applications by making changes to how JavaScript runs, allowing certain errors to be thrown instead of failing silently.

Default options

Default options are a great way of giving your codebase some extensibility. If, for example, we want to customize or cache an element related to the application itself then following are the defaults we will use:

```
var _defaults = {
'element': document.body,
    'name': 'App',
    'videoOptions': {},
    'audioOptions': {},
    'touchOptions': {},
    'formOptions': {},
    'locationOptions': {},
    'singlePageOptions': {}
};
```

Let's look at these defaults briefly. First we will create a `defaults` variable, which will contain all the defaults for our application(s). Inside it, we have defined a default location to be referenced for our application with the `'element'` default set to `document.body`—which gets our body element in **DOM** (**Document Object Model**). We then create a custom name for our application called `'App'`. After this, we create empty objects for our video, audio, touch, form, location, and single page applications—to be built later. These empty objects will be extended as we continue through the book.

Defining the constructor

Now we need to define our constructor after the `use strict` directive. This constructor will take a single parameter named `options`. We will then extend the defaults with the parameter `options` and store these settings that can be retrieved later, if needed. We will then ultimately cache the `'element'` option as a `Zepto` object.

```
function App(options) {
    this.options = $.extend({}, _defaults, options);
    this.$element = $(this.options.element);
}
```

Here is what the previous code is accomplishing. First, we are using the keyword `this`, which is a reference to what will be, an instance of App itself. Thus, `this` is the context of the object itself. Hopefully, this is not too confusing and will become clear as we go on. In this case, we are using `this` to define an object `options`, which will contain the merged contents of `_defaults` and any custom options we pass into the constructor.

Note, when we pass an empty object, or {} into `$.extend()` as the first parameter, we are telling `Zepto` to merge `_defaults` and `options` into a new object, thus not overwriting the `_defaults` object. This is useful when we need to do some sort of check in the future with the default options.

Once we've defined the options, we then cache the element with `this.$element`, where `$` in front of `element` is just for my reference so that I immediately recognize a Zepto object versus a plain JavaScript object.

The prototype

Ok, so we've created our `App` namespace, constructed an IIFE to contain our code and defined our constructor. Now, let's start creating some public methods that can be accessed to make this a bit modular. But before we do that, let's try to understand JavaScript's `prototype`.

Think of `prototype` as a live object that can be accessed, modified, and updated whenever and however you like. It can also be thought of as a pointer, because JavaScript will continue to go down the chain until it finds the object or it will return `undefined`. The prototype is simply a way of extending functionality to any non-plain object.

To make things a bit more confusing, I mentioned that non-plain objects have prototypes. These non-plain objects would be Arrays, Strings, Numbers, and so on. A plain object is one where we simple declare an empty object as follows:

```
var x = {};
```

The `x` variable does not have a prototype, it is simply there as a key/value storage similar to our `_defaults` object.

If you haven't yet understood the prototype, don't worry, it's all about getting your hands dirty and getting some experience. So, let's keep moving and getting our applications to work.

At this point, your `App.js` file should look like the following:

```
var App = window.App || {};
App = (function(window, document, $){
    'use strict';
    var _defaults = {
        'element': document.body,
        'name': 'App',
        // Configurable Options for each other class
        'videoOptions': {},
        'audioOptions': {},
        'touchOptions': {},
        'formOptions': {},
        'locationOptions': {},
        'singlePageOptions': {}
    };
    function App(options) {
        this.options = $.extend({}, _defaults, options);
        this.$element = $(this.options.element);
    }
}(window, document, Zepto));
```

Defining public methods

Now we need to create some public methods by typing into the prototype. We'll create a `getDefaults` method, which returns our default options; `toString` will overwrite the native `toString` method so we can return a custom name. Then we'll create initialization methods to create our other applications, and we'll name these `initVideo`, `initAudio`, `initLocalization`, `initTouch`, `initForms`, and `initSinglePage` respectively.

```
App.prototype.getDefaults = function() {
    return _defaults;
};

App.prototype.toString = function() {
    return '[ ' + (this.options.name || 'App') + ' ]';
};

App.prototype.initVideo = function() {
    App.Video.init(this.options.videoOptions);
    return this;
};

App.prototype.initAudio = function() {
    App.Audio.init(this.options.audioOptions);
    return this;
};

App.prototype.initLocalization = function() {
    App.Location.init(this.options.locationOptions);
    return this;
};

App.prototype.initTouch = function() {
    App.Touch.init(this.options.touchOptions);
    return this;
};

App.prototype.initForms = function() {
    App.Forms.init(this.options.formOptions);
    return this;
};

App.prototype.initSinglePage = function() {
    App.SinglePage.init(this.options.singlePageOptions);
    return this;
};
```

At this point we have several methods we can access publicly when we create an instance of `App`. First, let's review the code we implemented previously, specifically this line that gets duplicated, but customized based on the `init` method:

```
App.Touch.init(this.options.touchOptions);
```

For every `init` method we have created a call to the appropriate application, for example, `App.Touch`, `App.Forms`, `App.Video`, and so on. Then we pass it the options we've defined in the constructor that merged our defaults, for example, `this.options.touchOptions`, `this.options.formOptions`, `this.options.videoOptions`, and so on.

Note, we haven't created these classes yet for Video, Forms, Touch, and others, but we will be creating these soon.

Returning our constructor/function

The last thing we need to do in `App.js` includes returning the constructor. So, after all the public methods defined previously, include the following code:

```
return App;
```

This bit of code, although simple, is extremely important. Let's look at a stripped down version of `App.js` to better understand what's going on:

```
App = (function(){
    function App() {}
    return App;
}());
```

As mentioned earlier, we are creating an `App` namespace that gets set to the immediately invoked function expression. When we do this, we create a new scope inside this function.

This is why we can have a function or constructor with the name `App` as well and have no conflicts or errors. But if you recall, our function `App` is also an object, just like everything in JavaScript is an object. This is why, when we return our function `App` the `App` namespace gets set to the constructor. This then allows you to create multiple instances of `App`, while centralizing your code inside of a new scope that is untouchable.

Integrating a custom module template

Now, to get the rest of our architecture together we need to open up every other `App` file in the JavaScript directory we are in (`/js/App`).

When we have these files open, we need to paste the following template, which is based on the script we've written for `App.js`:

```
var App = window.App || {};

App.Module = (function(window, document, $){
    'use strict';

    var _defaults = {
        'name': 'Module'
    };

    function Module(options) {
        this.options = $.extend({}, _defaults, options);

        this.$element = $(this.options.element);
    }

    Module.prototype.getDefaults = function() {
        return _defaults;
    };

    Module.prototype.toString = function() {
        return '[ ' + (this.options.name || 'Module') + ' ]';
    };

    Module.prototype.init = function() {

        return this;
    };

    return Module;

}(window, document, Zepto));
```

When we have each template in, we must then change `Module` to the appropriate type, that is Video, Audio, Location, and so on.

Once you are done with pasting in the section and changing the names, you should be all set with the basic JavaScript architecture.

Including our scripts

One of the last items you will need to take care of is including this basic architecture into each `index.html` file. In order to do this, you will need to paste the following code at the bottom of the page, right after the inclusion of `helper.js`:

```
<script src="js/App/App.js"></script>
<script src="js/App/App.Audio.js"></script>
<script src="js/App/App.Forms.js"></script>
<script src="js/App/App.Location.js"></script>
<script src="js/App/App.SinglePage.js"></script>
<script src="js/App/App.Touch.js"></script>
<script src="js/App/App.Video.js"></script>
<script src="js/main.js"></script>
```

We are basically including each script of the framework. What's important here is to always include `App.js` first. The reason for this is that `App.js` creates the `App` object and directly modifies it. If you include it after all the other scripts, then `App.js` will overwrite the other scripts because it's directly affecting the `App` object.

Initializing our framework

The last item we need to take care of is `main.js`, which includes the initialization of our application. We do this by wrapping our code in IIFE and then exposing the instance to the `window` object. We do this with the following code:

```
(function(window, document) {
    'use strict';

    var app = new App({
        'element': document.querySelector('.site-wrapper')
    });

    window.app = app;

}(window, document));
```

What we've seen earlier is an IIFE being assigned to an object. Here we don't see that because it's not necessary. We just want to make sure our code would not affect the rest of the code, which in most cases would not happen because of the simplicity of this project. However, as a best practice I try to self contain my code in most cases.

The difference in the previous code is that we see the initialization of our framework here:

```
var app = new App({
    'element': document.querySelector('.site-wrapper')
});
```

We do that by using the `new` keyword, creating a new instance of `App`, and then passing it an object, which will be merged into our default options we previously wrote.

> `querySelector` is a JavaScript method that is attached to the document object. This method accepts a selector that we would normally use in CSS, parse DOM, and find the appropriate element. In this case, we are telling our application to self contain itself to the element with the `site-wrapper` class.

When we finally initialize our application, we then attach `app` to the `window` object:

```
window.app = app;
```

This basically makes it accessible anywhere in our application by attaching it to the `window` object.

We are now done with the framework for our application. Although we don't have anything being manipulated on the page, or have attached any events that correlate with a user's input, we now have a solid foundation for coding that follows best practices, is effective, efficient, and easily accessible.

Routing to a mobile site

Unless we are making a completely responsive site where the styles of the site shift based on the dimensions of the device, we most likely will need to do some sort of redirect to a mobile friendly version of our site.

Lucky for us, this can easily be achieved in several ways. Although I won't cover in detail the ways in which we can achieve this, here are a few techniques that might help out when deciding how to move forward.

> Since this book is geared towards the frontend, routing to a mobile site will be briefly covered with PHP and htaccess. We can always perform this process on the frontend, but it should be avoided for SEO and page-ranking purposes.

Redirecting via PHP

In PHP we could do the following type of redirect:

```
<?php
    $iphone = strpos($_SERVER['HTTP_USER_AGENT'], "iPhone");
    if ($iphone) {
        header('Location: http://mobile.site.com/');
    }
?>
```

In this example we are creating a variable, `$iPhone`, and giving it a Boolean value of true or false. If `iPhone` is found in the user agent, which may or may not be the best technique to use, then we tell the page to redirect using the `header()` method in PHP.

Again, there are other ways of achieving this, but this will get you off the ground and running.

Redirecting via htaccess

We can also detect the iPhone and redirect it by putting these instructions on the server using an `htaccess` file:

```
RewriteEngine on
RewriteCond %{HTTP_USER_AGENT} iPhone
RewriteRule .* http://mobile.example.com/ [R]
```

In this example, we are turning on the rewrite engine, creating a rewrite condition that checks for the `iPhone` text in the user agent, and then creates a rewrite rule if the condition is met.

In essence, if we want to redirect to a mobile version of our site, we need to be able to detect the type of device, not its dimensions, and then redirect appropriately.

Home screen icons

If you're creating an application that should mimic the feeling of being a native application, or to simply increase the experience of a web app—it is a good idea to have bookmark icons that represent your application.

At the moment, we do support this feature with the following markup in our `index.html` files:

```
<link rel="apple-touch-icon-precomposed" sizes="144x144"
href="img/touch/apple-touch-icon-144x144-precomposed.png">
<link rel="apple-touch-icon-precomposed" sizes="114x114"
href="img/touch/apple-touch-icon-114x114-precomposed.png">
<link rel="apple-touch-icon-precomposed" sizes="72x72"
href="img/touch/apple-touch-icon-72x72-precomposed.png">
<link rel="apple-touch-icon-precomposed" href="img/touch/apple-
touch-icon-57x57-precomposed.png">
<link rel="shortcut icon" href="img/touch/apple-touch-icon.png">
```

These directives inform Safari that we have home screen icons for the appropriate devices. Starting from top to bottom we are supporting retina display, first-generation iPad and non-Retina iPhone, iPad Touch, and even Android 2.1+.

To put it simply, we have an application that users can bookmark to their home screen, allowing them to instantly access the web application from their home screen.

Introducing our build script

Earlier, we installed our build script along with the HTML5 Mobile Boilerplate. We'll now explore the build script a bit further by customizing it for our purposes. We'll need to make sure our styles, scripts, images, and markup are optimized for deployment. It will also be necessary for us to set up multiple environments to test our application thoroughly.

Configuring our build script

Let's start by configuring the build script for our needs, this way we'll have a custom build script that works for us and gets us going immediately.

Minifying and concatenating scripts

First, let's make sure our scripts get concatenated and minified. So let's open all our `index.html` files and wrap all our scripts at the bottom of the page with the following comments:

```
<!-- scripts concatenated and minified via ant build script-->
<script src="path/to/script.js"></script>
<!-- end scripts-->
```

The previous comments are used by the `ant` task, or build script, to find all JavaScript files being used, concatenate, and minify them. The process will also use a timestamp for the newly optimized JavaScript file in order to bust caching on the server.

Minifying and concatenating styles

By default, the Ant Build script minifies and concatenates our styles. However, if we want to retain comments that identify a particular section of our app, such as the video or audio section, then we need to do something that will keep those comments.

The comments can be used to identify a section, and it can be written as follows:

```
/*!
  Video Styling
*/
```

Write the previous comments for each stylesheet.

We then need to add each of our stylesheets to the project properties so that each can be minified by the YUI compressor. To do this, we need to open up the `project.properties` file located in `/build/config`.

Then find the following line:

```
file.stylesheets  =
```

Once we've found that line, let's add all our `css` files as follows:

```
file.stylesheets  = audio.css,forms.css,location.css,singlepage.
css,touch.css,video.css
```

Note, that there are no spaces after each file. This is necessary for the build script to process.

This is all we need to do at the moment for optimizing our styles.

Creating multiple environments

Typically a project will run on a development, test, and production environment. The test environment should be closest to production in terms of configuration, allowing us to effectively reproduce any issues that might come up.

In order to build our environments correctly, let's go through the process of building our project. First, let's open up `Terminal`, a program that allows you to interact with the operating system of any Unix style computer through a command-line interface.

Navigating our directories

Once the terminal is up and running, we have to navigate to our project. Here are a couple of commands that will help you navigate:

```
cd /somesite
```

The previous command means we are changing our directory from the current directory to the `somesite` directory, relative to where you're now.

```
cd ../somesite
```

This command tells us to change the directory, but going up a level with `../` and then going into the `somesite` directory.

As an easier example to understand, my project exists in `/Sites/html5iphonewebapp`. So what I can do is use the following command to enter my project:

```
cd /Users/somuser/Sites/html5iphonewebapp
```

This changes the directory for me to the project where I am developing this application.

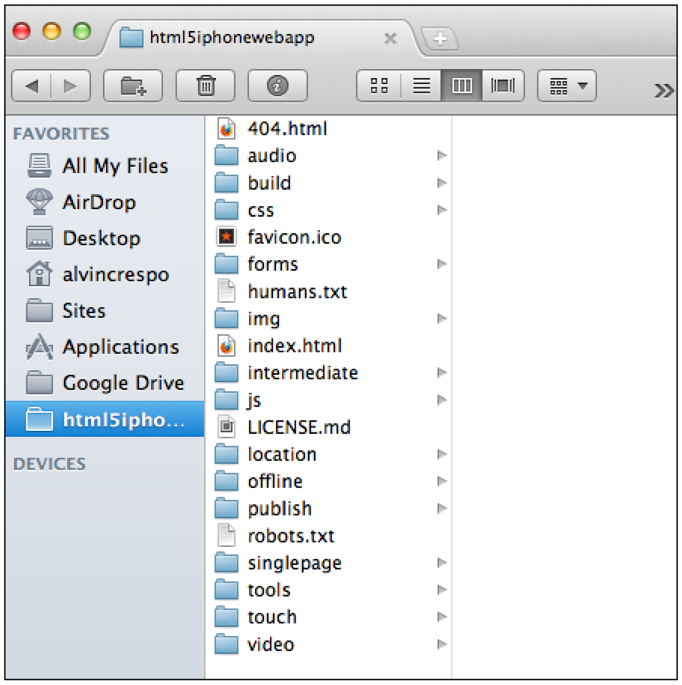
Building our project

Once we've entered the project directory, we can start building our project. By default, the Ant Build script creates a production environment, which optimizes all parts of the process.

```
    ant build
```

This command tells us to build our project, and as explained creates our production version in a directory labeled `publish`. You will also notice that when you run that command, your terminal will update, letting you know what step in the process the build is in.

Once the build is complete, your directory structure should look similar to the following screenshot:



404.html
audio
build
css
favicon.ico
forms
humans.txt
img
index.html
intermediate
js
LICENSE.md
location
offline
publish
robots.txt
singlepage
tools
touch
video

The `publish` directory represents the production environment. You will also see that an intermediate directory has been created; this is your test environment.

However, let's say you wanted to have full control of the build and wanted to create your environments manually, then one can do the following in the terminal:

```
ant build -Denv=dev
```

This command, `ant build -Denv=`, lets us define which environment we want to build and does it accordingly.

We now have a project that is ready to be built upon. There were many steps in this process, so I encourage you to practice this process in order to develop a good architecture and deployment process that works for you and/or your team.

Summary

In this chapter, we saw how to use the HTML5 Mobile Boilerplate for our projects, from downloading the default package to customizing it for our needs. We also took a couple of simple steps to establish a solid architecture for our JavaScript, CSS, and HTML. As a bonus, we went over including a build process and customizing it for our project. We then quickly reviewed best practices for JavaScript applications and gave a couple of tips on how to direct users to a separate mobile site. We are now prepared for in-depth development of the mobile web applications.

2
Integrating HTML5 Video

Media distribution is essential for any web application; delivering rich experiences that alter the user's perception. Many times we are asked to put a still image on a site, and other times we are asked to include video galleries that allow the user to switch videos easily through some sort of unique navigation. Previously, we were able to do this with Flash and other plugin based technologies, but with the wide support of HTML5 video, we now have the ability to deliver video without the condition of downloading a plugin.

One thing to keep in mind is that HTML5 video and audio share the same specification. This is because they are both considered a media element (`http://www.whatwg.org/specs/web-apps/current-work/multipage/the-video-element.html#media-element`). What this means is that both video and audio share some attributes and methods, making it easier to implement them within our applications.

Either way, let's get going by learning how we can configure our server to deliver our videos correctly.

In this chapter, we will cover:

- Configuring our server for video distribution
- Implementing HTML5 video
- Listening to HTML5 video events
- Creating a thorough JavaScript video library
- Customizing HTML5 video controls

Configuring the server

Before implementing our video, we need to make sure our server knows what media types we will be serving. Doing this now helps to avoid headaches later on when we don't know why we are getting network errors. So let's get to it.

Video formats

First, we need to know what file types we will be serving up. In our examples we will be using MP4, but it's always a good idea to allow what is supported. Make sure you have WebM, OGV, and MP4 formats for your videos. But first, before we go on, let's side step and learn a little bit about these formats.

> We're not going to get too deep into explaining the different types that are widely supported, but keep in mind that Theora, WebM, and H.264/MPEG-4 are the most widely supported formats. Both Theora and WebM are free, with WebM development being supported by Google. The implementation for Theora in browsers has lagged due to worry about patents, while WebM has been widely supported by Mozilla Firefox, Google, and Opera due to its royalty-free and open video compression features.
>
> Things get a bit hairy when it comes to H.264. Although a great format for having high quality, good speed, and being a standard for video compression, patents encumber it. For this reason, it has lagged behind in support in popular browsers for a long time. Eventually, each browser came to support this format, but not without controversy.

Video format directives

Next, depending on the server type, we need to include specific directives to allow our file types. In this example, we are using an Apache server, thus the following syntax:

```
AddType video/ogg .ogv
AddType video/mp4 .mp4
AddType video/webm .webm
```

The previous code would be added to either an `.htaccess` file on the server or to the `httpd.conf` file. Either way, the `AddType` directive informs the server what types it should and can serve. So when we go line by line, we can see that we are adding the `video/ogg` type with extension `.ogv`, and we do this for MP4 and WebM.

Taking these initial steps helps us to avoid any network issues when we go ahead and implement video on our site using HTML5. If you're curious how we use these types, don't worry, that's exactly what we'll be going over in the next section.

A simple HTML5 video

We've been dying to do something cool in our web application, so let's get to it. Let's start out by including a video on our site in the simplest manner possible, and without any complicated interactions!

Single video format

First, let's open up our `index.html` file located in the `video` subdirectory of the `Chapter 2` project. If you skipped the first chapter, don't worry about it, the `Chapter 2` source files follow along and help keep you going.

Once we have our `index.html` file, we want to include the `video` element within our content area, right after the `<header>` element. This is simple, we can do it like this:

```
<video src="../assets/testvid.mp4" controls preload></video>
```

The previous code works similar to an image element. We define a `src` attribute that instructs the browser where to find the video, and then we define a `controls` and `preload` attribute, which directs the browser to display default native controls and to preload the video. Simple, eh?

Supporting multiple formats

This is all we need to put video on our site, but of course, things aren't always so simple. As we discussed earlier, browsers can support one or none of the formats that we specify. Of course, right now we have good browser support, but we want to make sure that our application is solid, so we need to make sure that we deliver the appropriate file. To do this, we can modify the previous code as follows:

```
<video poster="testvid.jpg" controls preload>
    <source src="testvid.webm" type='video/webm'/>
    <source src="testvid.ogv" type='video/ogg'/>
    <source src="textvid.mp4" type='video/mp4'/>
    <p>Fallback Content</p>
</video>
```

Here, we have introduced a new attribute, `poster`. The `poster` attribute is an image we show at the beginning of the video or in cases where the video doesn't load. Things now get a bit complex when we move the source element inside of the `video` element. But, if we check everything out, we're basically defining multiple source videos and their types. The browser will then choose the appropriate one to display. What might confuse you is the paragraph element containing the `Fallback Content` text. This is here if everything else fails and/or if the browser does not support the HTML5 video.

If this is a bit confusing, don't worry about it too much because iPhone's mobile Safari supports MP4 and is honestly all you need for your applications. So if we want to keep it simple, we can use the following code for our iPhone applications, and this is exactly what we do in this book:

```
<video src="testvid.mp4" controls preload>
    <p>Video is not supported in your browser.</p>
</video>
```

Now that we have a simple video playing in our application, we might want to pick up on the events of our video.

Listening to HTML5 video events

It's very likely that you will want full control of your application, or at least monitor what may be going on. You will usually find yourself attaching events or listening to them for various reasons. From tracking to enhancing an experience, events are how we can drive interactivity on the page. With HTML5 video, we can use the native browser to monitor the status of the video from start to finish. You have the opportunity to listen for when a video has finished loading and when the user has paused a video.

Let's review the events that are available to us. What you will find is that the events we use with videos can also be transferrable for audio. This is because, as we've learned previously, both video and audio elements are categorized as media elements in the HTML5 specification. Here is a table of events that are available to us:

Event Name	**Condition**
`loadedmetadata`	The duration and dimensions of the media resource have been determined.
`loadeddata`	Media data can now be rendered for the first time.
`canplay`	Playback of the media data can resume.
`seeking`	The seeking attribute of the media resource has been set to true.
`seeked`	The seeking attribute of the media resource has been set to false.
`play`	The element is not paused. This is fired when the `play()` method has returned or when the `autoplay` attribute has caused the element to begin playback.

Event Name	Condition
`ended`	The end of the media resource was reached and playback has stopped.
`pause`	The `pause()` method has returned and the element has been paused.
`timeupdate`	The playback position of the media resource has changed in some way.
`volumechange`	Fired when either the volume or muted attribute has changed.

There are more events that are defined by the specification, however these are the events we will listen to from our previous simple implementation. So let's get started.

Video markup review

First, open your `index.html` file inside of the `video` directory. In this file, you have to make sure your content looks like the following:

```
<div class="site-wrapper">
    <header>
        ....
    </header>
    <div class="gallery">
                    <video src="../assets/testvid.mp4" controls
preload></video>
    </div>
    <footer>
        ...
    </footer>
</div>
```

Don't pay attention to the ellipsis, these are just in there to make the code shorter in the text. What you want to make sure of is that you have the simple `<video>` element implementation from the last section.

Attaching video events

Now the fun starts. Let's start extending our JavaScript to include the listeners. Let's open the `App.Video.js` file located in the `/js` directory under the `App` folder. If you haven't been following along from our architecture chapter, don't worry, what's important for you to understand here is that we have created a structure for our applications and that the `App.Video.js` file will contain all functionality for the video application.

Find the constructor for the `App.Video` class. This should be on line 16 of your text editor, and should currently look like the following:

```
function Video(options) {
    // Customizes the options by merging them with whatever is passed
in
    this.options = $.extend({}, _defaults, options);

    //Cache the main element
    this.$element = $(this.options.element);
}
```

Again, as a review we are passing in an object that we call `options` into our constructor. From here, we create a property called `options` that is for that instance of `Video`, and this property will get set to an extended or merged version of the options and defaults using Zepto's extend method. Then, we cache the element that was sent via the merged options. This might be a bit confusing, but it is a very well recognized pattern in JavaScript applications.

Since we have verified that our constructor exists and is doing fine, we now want to add the previous listeners. We can easily do this using the native `addEventListener` method as follows:

```
this.options.element.addEventListener('canplay', function(e){
    console.log('video :: canplay');
});

this.options.element.addEventListener('seeking', function(e){
    console.log('video :: seeking');
});

this.options.element.addEventListener('seeked', function(e){
    console.log('video :: seeked');
});

this.options.element.addEventListener('ended', function(e){
    console.log('video :: ended');
});

this.options.element.addEventListener('play', function(e){
    console.log('video :: play');
});
```

```
this.options.element.addEventListener('pause', function(e){
    console.log('video :: pause');
});

this.options.element.addEventListener('loadeddata', function(e){
    console.log('video :: loadeddata');
});

this.options.element.addEventListener('loadedmetadata', function(e){
    console.log('video :: loadedmetadata');
});

this.options.element.addEventListener('timeupdate', function(e){
    console.log('video :: timeupdate');
});
```

There are a couple of things to note here. First, we are using `this.options.element` instead of the cached version `this.$element`. We do this because we actually want the element and not a `Zepto` object. Second, we are calling `addEventListener` and passing it two parameters. The first parameter is a string that defines the event we want to listen to. The second parameter is a callback function, which gets called every time that the event we specified in parameter one fires.

> Note that we are using the `console.log()` method. It is similar to `alert()` except without all the annoyances. It helps debug better and outputs to a console that lets us keep track of all our log output. Using this method is a good way of debugging our applications and testing out functionality before going further.

Your constructor should now look like this:

```
function Video(options) {
    // Customizes the options by merging them with whatever is passed
in
    this.options = $.extend({}, _defaults, options);

    // Cache the main element
    this.element = options.element;
    this.$element = $(this.options.element);
```

```
        this.options.element.addEventListener('canplay', function(e){
            console.log('video :: canplay');
        });

        this.options.element.addEventListener('seeking', function(e){
            console.log('video :: seeking');
        });

        this.options.element.addEventListener('seeked', function(e){
            console.log('video :: seeked');
        });

        this.options.element.addEventListener('ended', function(e){
            console.log('video :: ended');
        });

        this.options.element.addEventListener('play', function(e){
            console.log('video :: play');
        });

        this.options.element.addEventListener('pause', function(e){
            console.log('video :: pause');
        });

        this.options.element.addEventListener('loadeddata',
    function(e){
            console.log('video :: loadeddata');
        });

        this.options.element.addEventListener('loadedmetadata',
    function(e){
            console.log('video :: loadedmetadata');
        });

        this.options.element.addEventListener('timeupdate',
    function(e){
            console.log('video :: timeupdate');
        });
    }
```

Initializing our video

Now that we have a preliminary video class defined, we need to initialize it. So let's go ahead and open up `main.js`, where our initialization code should be located. It should look something like this:

```
(function(window, document) {
    'use strict';

    // Create an instance of our framework
    var app = new App({
        // Custom Option, allowing us to centralize our framework
        // around the site-wrapper class
        'element': document.querySelector('.site-wrapper')
    });
    // Expose our framework globally
    window.app = app;
}(window, document));
```

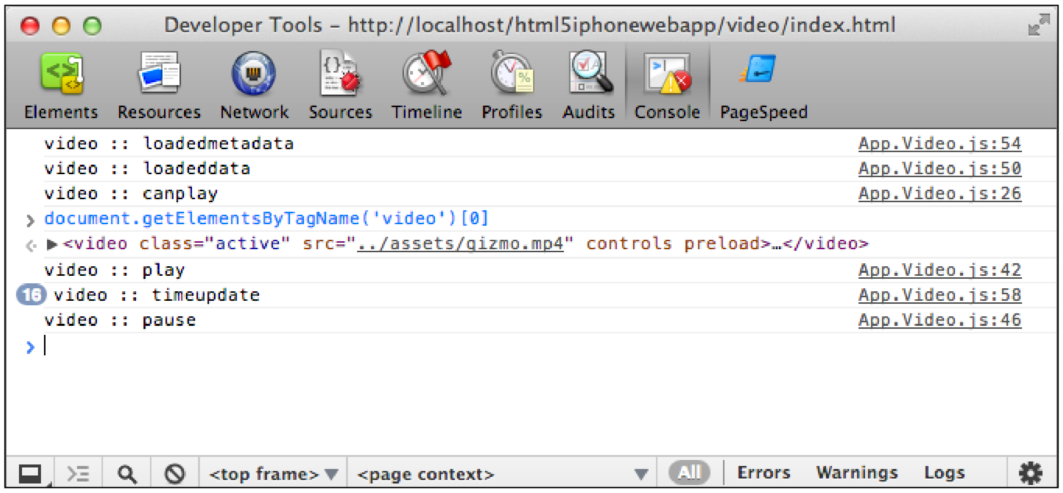
We created this in the previous chapter, but let's go over it briefly. Here we are creating a closure, passing it the `window` and `document` objects. Inside, we set the interpreter to read our code strictly. We then create an instance of the overall `App` class and then expose it to the `window` object.

Now we need to add the initialization of the `Video` class. To do this, let's put the following snippet of code after we declare a new instance of `App`, as follows:

```
new App.Video({
    'element': document.getElementsByTagName('video')[0]
});
```

This snippet creates a new instance of the `App.Video` class or the `Video` class, and passes in a simple object containing the element. The way we retrieve an element is through the use of the `getElementsByTagName` method attached to the `document` object. We tell the method to look for all the video elements. The interesting part is `[0]`, which tells the results of the lookup to only get the first one in the array that is returned.

If we load our page and test the video, we should have the log output we defined earlier in our console, similar to the following screenshot:



```
video :: loadedmetadata                                    App.Video.js:54
video :: loadeddata                                        App.Video.js:50
video :: canplay                                           App.Video.js:26
> document.getElementsByTagName('video')[0]
< ▶<video class="active" src="../assets/gizmo.mp4" controls preload>…</video>
video :: play                                              App.Video.js:42
16 video :: timeupdate                                     App.Video.js:58
video :: pause                                             App.Video.js:46
>
```

Video log output

We've got the preliminary aspects of our `Video` class going, from events to initialization. However, we need to tidy it up a bit if we are going to make it reusable for our application, and if we ever want to extend its functionality. So let's take some time creating a fully functional JavaScript video library that will work in our iPhone web applications.

Creating a JavaScript video library

At the moment, we have a very simple `Video` class that caches an element and then attaches multiple events that are defined by the media element specification for HTML5. We have defined the essentials of a video player and now need to abstract it a bit further so it can be reused and managed much better. Following some conventions and creating a flexible framework will help us move faster and more effectively.

First, let's think about some things we may want from this class:

- An events method that attaches the proper events
- Callback methods that can be defined, for example, `onPlay`, `onPause`, and `onEnded`

- Public methods that can be called from outside the instance
- Chainable methods similar to jQuery where you can call one method after another, for example, `fadeIn().fadeOut().show().hide()`

Having a list of items that abstract the behavior of a class is a step in the right direction for establishing a solid framework or library. Now let's start by creating callbacks.

Centralizing our events

First, let's tackle how we attach the events for our `Video` class. Previously, we added these events to the constructor, and although that is a fine technique, it can definitely be improved by specifying a function that handles the attachment of events onto an instance of a `Video` object.

So, let's create a private method called `attachEvents` in the `Video` class that can only be accessed within the `App.Video` closure or IIFE. When we create our method `attachEvents`, we should just place all our event handlers into it. We then want to call the `attachEvents` method after we initialize `this.$element`. When you're done doing this, your code should look like the following:

```
function Video(options) {
    this.options = $.extend({}, _defaults, options);

    // Cache the main element
    this.element = options.element;
    this.$element = $(this.options.element);

    attachEvents();
}

function attachEvents() {
    // All your event handlers go here
}
```

In the previous code, the `attachEvents()` function will contain our event handlers created earlier. For the sake of brevity, I'm omitting them for now.

Now, if we run this code we'll most likely run into a couple of errors. This is actually normal and is known as a scope issue. In order to fix the problem, first we'll sidestep to understand what's going on behind the scenes.

Scope in JavaScript

If you're new to JavaScript, scope will most likely confuse you sooner or later. If you're at an intermediate or advanced level in JavaScript, you might still have scope issues. This is completely normal and is something that we all come across. Either way, let's take our current `Video` class and analyze it to understand scope in context.

JavaScript has function-level scope, this means that every time a new function is created, we create a new scope. Scope can be quite confusing, but with practice it becomes easier. For now think of scopes as a reference to your current position, aware of itself and its environment, but unaware of newly created scopes inside it. If it sounds confusing, it can be when you get started. But let's go over some code to gain a better understanding.

So, let's start with the global scope:

```
// Global Scope
var x = 10;
(function($){
    // New Scope
    console.log(x);
}(Zepto));
```

In this example, a stripped down version of `App.Video`, we can see that the global scope is around the closure. When we create a closure, a new scope is created. The cool thing here is that anything outside of the closure can get accessed. So when we do `console.log` inside the closure, we should get back `10`.

Whenever you create a new function scope, you can pass it parameters that essentially namespace the value you're sending. In this case, we pass in `Zepto` and we tell the new function scope to define the dollar sign as an instance of `Zepto` inside that scope. Hopefully that explains scope a bit more clearly, if not, don't worry about it; it does take time and patience to understand this concept.

So, the problem with our event handlers is the fact that the new function scope, inside of `attachEvents`, does not have a reference to `this.options`. The keyword `this` is relative to the window object because of the new scope. The reason it doesn't have a reference is because our constructor is a completely different scope, and the two are not speaking with each other. To fix this problem, we can use the `.call()` method, which will change the reference of the `this` keyword to reflect the `Video` function scope. It can be done by changing the call of `attachEvents` like so:

```
attachEvents.call(this);
```

If you run your code now, you should not get any errors. If you do, take a look at the finalized version of the code in order to do a comparison and figure out the issue.

Exposing functionality

Later on in this chapter, we will be exploring custom user interfaces that help us override the default functionality of a video player. However, in order to do this we need to make sure some functionality is exposed. In JavaScript, in order to make methods public outside of the closure, we need to attach methods to the prototype of `class`—in this case, `Video`.

We can already see that we have two methods that are exposed within all our classes; these include `getDefaults` and the overriding function `toString`. Let's start extending the prototype by adding the `play`, `pause`, `stop`, `mute`, `unmute`, and `fullscreen` methods.

```
Video.prototype.play = function() {
    return this;
}

Video.prototype.pause = function() {
    return this;
}

Video.prototype.stop = function() {
    return this.pause();
}

Video.prototype.mute = function() {
    return this;
};

Video.prototype.unmute = function() {
    return this;
};

Video.prototype.fullscreen = function() {
    return this;
}
```

I'm sure you have noticed the lack of code in these methods, and that's fine. What we want to understand is that we can extend the `Video` prototype, and that we can add chaining to our methods by returning the instance with the `return this` line.

Let's start adding functionality to our methods, beginning with `play`:

```
Video.prototype.play = function() {
    this.element.play();

    return this;
}
```

Here we are getting the element we've cached in the constructor of `Video` by calling the `play` method. You might be wondering where this `play` method is coming from? Well, the HTML5 specification defined a `play` method for media elements, including video and audio. Thus, we can tap into this method using `this.element.play()`. We can do the same with the `pause` method as follows:

```
Video.prototype.pause = function() {
    this.element.pause();
    return this;
}
```

Again, we have a method defined by the HTML5 specification for pausing a media element. Things get a bit confusing when we define a `stop` method as follows:

```
Video.prototype.stop = function() {
    return this.pause();
}
```

It's the same as previous; we actually haven't made any changes. Let me explain, the specification does not define a `stop` method, so it's up to us to create one so that we can provide that functionality. But it's not too difficult because we've defined a `pause` method that does a similar action. So all we need to do is call `this.pause()`, because this is an instance of `Video` and we have a `pause` method defined. The neat thing here is that we don't need to return `this`, because the pause method already returns `this`, and so all we need to do is return the results of calling the `pause` method. I know it's a bit confusing but over time, if this is the first time you're doing this, it will become clear.

Now, on to our `mute` and `unmute` methods:

```
Video.prototype.mute = function() {
    this.element.muted = true;
    return this;
};
Video.prototype.unmute = function() {
    this.element.muted = false;
    return this;
};
```

The only difference in these methods is that we are setting a property on the video element to `false`. In this case, we are setting the muted property to either `true` or `false`, depending on what method you call.

Here is where things get a bit complicated:

```
Video.prototype.fullscreen = function() {
    if (typeof this.element.requestFullscreen === 'undefined') {
        this.element.webkitRequestFullScreen();
    } else {
        this.element.requestFullscreen();
    }
    return this;
}
```

This is a bit more complicated, and probably a bit frustrating. Trust me, many within the industry are feeling the pain. What we need to understand here is that the browser we are dealing with, Safari, runs on an engine called WebKit—an open source web browser engine.

WebKit is extremely popular and widely supported. The issue is that while it does a great job at implementing the latest and greatest features, many of these are experimental and thus, they have a prefix added to them. We see this a lot in CSS (Cascading Style Sheets) using `-webkit`. But we also face the same problem in JavaScript, `webkit[standardMethodName]`.

While this may be awesome, we need to make sure that we have backward compatibility for newer versions that strip out that prefix. This is why, in the previous method, we do a check on the standard method name, and if it doesn't exist, we use the `-webkit` prefix. Otherwise, we use the standard version.

Integrating callbacks

Callbacks are quite useful in any library or framework, and you've probably already seen something like it using jQuery or some other popular framework. In essence, a callback is a method that gets called once the method has completed. For example, in the `Zepto` method, `fadeout` accepts two parameters, the first being the speed, and the second parameter is a function that gets called when the fading has completed. This can be seen as follows:

```
$('.some-class').fadeout('fast', function(){
    // Do something when fading is complete
});
```

The second parameter in the previous code is not only a callback function, but also an anonymous function. An anonymous function is just a function without a name. In this case, it is executed every time the `fadeOut()` effect is finished. We can rewrite the previous code as follows:

```
$('.some-class').fadeOut('fast', someFadeOutFunc);
function someFadeOutFunc(){
    // Do something when fading is complete
}
```

Since we have created a method called `someFadeOutFunc`, when `fadeOut` is complete we'll just call that function instead of creating a new one. This is more efficient and manageable from an architecture standpoint.

The first step in the process of creating callbacks is to define where we may want callbacks in our code. In this case, we may want a callback for each action taken in the video player, so we'll create the following callbacks:

- `onCanPlay`
- `onSeeking`
- `onSeeked`
- `onEnded`
- `onPlay`
- `onPause`
- `onLoadedData`
- `onLoadedMetaData`
- `onTimeUpdate`
- `onFullScreen`

Ok, so now that we know which callbacks we want in our code, let's implement them in the constructor, right before the `attachEvents` method:

```
this.callbacks = {
    'onCanPlay': function(){ },
    'onSeeking': function(){},
    'onSeeked': function(){},
    'onEnded': function(){},
    'onPlay': function(){},
    'onPause': function(){},
    'onLoadedData': function(){},
    'onLoadedMetaData': function(){},
    'onTimeUpdate': function(){},
    'onFullScreen': function(){}
};
```

What we've done here is attached a property, known as `callbacks`, onto an instance of `Video`. This property contains an object that has key/value pairs for each callback we want to implement, with the value being an empty anonymous function.

Extending callbacks

Although we can have our callbacks within the class, the problem is that they are not extensible, meaning the developers using your `Video` class won't be able to extend your callbacks. In order to make them extensible, we need to put them within our `_defaults` object:

```
var _defaults = {
    'element': 'video',
    'name': 'Video',
    'callbacks': {
        'onCanPlay': function(){ },
        'onSeeking': function(){},
        'onSeeked': function(){},
        'onEnded': function(){},
        'onPlay': function(){},
        'onPause': function(){},
        'onLoadedData': function(){},
        'onLoadedMetaData': function(){},
        'onTimeUpdate': function(){},
        'onFullScreen': function(){}
    }
};
```

The drawback is that now we would need to use `this.options.callbacks` in order to access the callback we want. This can be easily fixed by doing the following in our constructor:

```
this.callbacks = this.options.callbacks;
```

This will still allow us to access the callback, but only from the extended object.

Using callbacks

Now that we have our callbacks, and have made them extensible, we can go in and integrate them into our event handlers. But first, we need to make our event handlers as private methods in this `Video` class and call our custom callbacks as follows:

```
function onCanPlay(e, ele) {
    this.callbacks.onCanPlay();
}

function onSeeking(e, ele) {

    this.callbacks.onSeeking();
}

function onSeeked(e, ele) {

    this.callbacks.onSeeked();
}

function onEnded(e, ele) {

    this.callbacks.onEnded();
}

function onPlay(e, ele) {

    this.callbacks.onPlay();
}

function onPause(e, ele) {

    this.callbacks.onPause();
}

function onLoadedData(e, ele) {
    this.callbacks.onLoadedData();
}

function onLoadedMetaData(e, ele) {
    this.callbacks.onLoadedMetaData();
}

function onTimeUpdate(e, ele) {
    this.callbacks.onTimeUpdate();
}
```

At this point, we have our callbacks fully integrated into our library. Now, we just need to call them by modifying the `attachEvents` handler as follows:

```
function attachEvents() {
        var that = this;
        this.element.addEventListener('canplay', function(e){
onCanPlay.call(that, e, this);  });
        this.element.addEventListener('seeking', function(e){
onSeeking.call(that, e, this); });
        this.element.addEventListener('seeked', function(e){ onSeeked.
call(that, e, this);  });
        this.element.addEventListener('ended', function(e){ onEnded.
call(that, e, this);  });
        this.element.addEventListener('play', function(e){ onPlay.
call(that, e, this);  });
        this.element.addEventListener('pause', function(e){ onPause.
call(that, e, this);  });
        this.element.addEventListener('loadeddata', function(e){
onLoadedData.call(that, e, this);  });
        this.element.addEventListener('loadedmetadata', function(e){
onLoadedMetaData.call(that, e, this);  });
        this.element.addEventListener('timeupdate', function(e){
onTimeUpdate.call(that, e, this);  });
    }
```

There are a couple of concepts being implemented here. First, we've replaced `console.logs` with the actual private methods that we have defined previously. Second, we used the `call` method to change the scope of the `private` method by passing in `that`, then we sent in `event` and `element` as parameters.

Tying it all up

We have everything we need, such as event handlers, exposed functionality, callbacks, and even chainable methods. That's all good, but now we need to make it work. This is where the magic comes in.

To verify, your `Video` class should look something like this:

```
var App = window.App || {};

App.Video = (function(window, document, $){
    'use strict';

    var _defaults = { ... };
```

```
        // Constructor
        function Video(options) {
            this.options = $.extend({}, _defaults, options);

            this.element = options.element;
            this.$element = $(this.options.element);

            this.callbacks = this.options.callbacks;

            attachEvents.call(this);
        }

        // Private Methods
        function attachEvents() { ... }

        // Event Handlers
        function onCanPlay(e, ele) { ... }
        function onSeeking(e, ele) { ... }
        function onSeeked(e, ele) { ... }
        function onEnded(e, ele) { ... }
        function onPlay(e, ele) { ... }
        function onPause(e, ele) { ... }
        function onLoadedData(e, ele) { ... }
        function onLoadedMetaData(e, ele) { ... }
        function onTimeUpdate(e, ele) { ... }

        // Public Methods
        Video.prototype.getDefaults = function() { ... };
        Video.prototype.toString = function() { ... };
        Video.prototype.play = function() { ... }
        Video.prototype.pause = function() { ... }
        Video.prototype.stop = function() { ... }
        Video.prototype.mute = function() { ... };
        Video.prototype.unmute = function() { ... };
        Video.prototype.fullscreen = function() { ... }

        return Video;

    }(window, document, Zepto));
```

[Please note that the ellipsis in the previous code denote that there should be functionality. Due to page count limitations, we can only showcase a brief summary of the code up to this point. If you need to see what has been done, please review the previous sections or checkout the source code with this book.]

If your file looks like this, perfect! If it doesn't look quite like this, don't worry about it, this is why we have the source code attached with this book. At this point, we're ready to initialize this library on our page.

Let's open up our `main.js` file; the file should be located under the `js` directory. We need to make the following addition:

```
new App.Video({
    'element': document.getElementsByTagName('video')[0],
    'callbacks': {
        'onCanPlay': function(){ console.log('onCanPlay'); },
        'onSeeking': function(){ console.log('onSeeking'); },
        'onSeeked': function(){ console.log('onSeeked'); },
        'onEnded': function(){ console.log('onEnded'); },
        'onPlay': function(){ console.log('onPlay'); },
        'onPause': function(){ console.log('onPause'); },
        'onLoadedData': function(){ console.log('onLoadedData'); },
        'onLoadedMetaData': function(){ console.
log('onLoadedMetaData'); },
        'onTimeUpdate': function(){ console.log('onTimeUpdate'); },
        'onFullScreen': function(){ console.log('onFullScreen'); }
    }
});
```

Let's go through this quickly. First, we create a new instance of `App.Video`, passing in one parameter— a simple object. Second, the object we are passing in contains two objects: the `video` element we want on the page, and a callbacks object that overrides defaults. The first parameter is using the built in method `getElementsByTagName` to get all the instances of the `video` element, and then we get the first one found using `[0]`. This is because that method returns an array. The second parameter, `callbacks`, contains the function callbacks we want to be called on this instance of `App.Video`. All we want to do in these methods is log out the method being called.

From here on, when the instance is initialized the `Video` library we've defined will merge the simple object we've passed in and take it from there. Almost like magic, except we've created it.

One last item to take care of is to make sure we only initialize a video when we are on the video page. If we are on a non-video page in our application, this code will produce an error. This is because there are no video elements and we have not added error detection. This is a good thing to have, but will not be covered in this book. So, let's do the following in `main.js`:

```
if(document.querySelector('video') !== 'null') {
    new App.Video({
        'element': document.getElementsByTagName('video')[0],
        'callbacks': {
            ...
        }
    });
}
```

In the previous code, we are wrapping our initialization code within an `if` statement, checking to make sure that we are on the video page. The way we do a check is by using the built in method on the document object called `querySelector`. This method accepts a CSS type selector, in this case we are sending the `video` selector, telling it to get all the instances of a `video` element. If the result coming back is not null, then we initialize.

Now we don't need to do anything with the markup, this code will run and we should be good. If for some reason you come up with any errors, please take a look at the source code provided with this book. Next, let's consider overwriting the default controls of this video player to give us more control over the functionality.

Customizing HTML5 video controls

We probably want more input into the video controls, from styling to video functionality, such as adding a stop button. In order to do this we need to modify our markup a bit. We should do the following with the video:

```
<div class="video-container">
    <video src="../assets/testvid.mp4" controls preload>
        <p>Video is not supported in your browser.</p>
    </video>
</div>
```

All we did here was add a class containing `div` around the `video` element and added a class of `video-container` to it. Now we want to add some responsive styling to the `video` element, so let's open up `video.css` and add the following styles:

```
video {
    display: block;
    width: 100%;
    max-width: 640px;
    margin: 0 auto;
}

.video-container {
    width: 100%;
}
```

The first selector will apply to all the `video` elements on the page, and we are telling each element to have a width of 100 percent relative to its container, but to only have a max width of `640px`. The margin property helps it center itself on the page or to the container. The next selector, `video-container`, just specifies the width to be 100 percent. This styling will resize the player accordingly; you can check it out by resizing your browser.

For this example, we'll use basic controls using the anchor element. Keep in mind that you can use any sort of styling or markup to style your controls, just remember that we have exposed our video playback in our `Video` class, so to keep it short and simple, we'll just demonstrate how you can do this using the anchor element.

In our `video-container` dive, we want to append the following markup:

```
<div class="video-controls">
    <div class="vc-state">
        <a class="vc-play vc-state-play" href="#play">Play</a>
        <a class="vc-pause vc-state-pause" href="#pause">Pause</a>
    </div>
    <div class="vc-track">
        <div class="vc-progress vc-track-progress"></div>
        <div class="vc-handle vc-track-handle"></div>
    </div>
    <div class="vc-volume">
        <a class="vc-unmute vc-volume-unmute" href="#volume">Volume
On</a>
        <a class="vc-mute vc-volume-mute" href="#volume">Volume Off</
a>
    </div>
    <a class="vc-fullscreen" href="#fullscreen">Fullscreen</a>
</div>
```

The previous markup is what we will be using for our controls. They are pretty straightforward, but let's review a couple of decisions that were made here. One, we have a surrounding `div` with a class of `video-controls` to help define where all our controls will exist. Two, each type of control is prefixed with `vc`, standing for video controls. Three, in this example we have four types of controls, namely a state, track, volume, and full screen control. The final point is that some of these controls have show/hide functionality, for example, play and pause should only show when the others cancel out.

For styling, we can add the following styles to the `video.css` file:

```
.video-controls {
    margin: 12px auto;
    width: 100%;
    text-align: center;
}

.video-controls .vc-state,
.video-controls .vc-track,
.video-controls .vc-volume,
.video-controls .vc-fullscreen {
    display: inline-block;
    margin-right: 10px;
}

.video-controls .vc-fullscreen {
    margin-right: 0;
}

.video-controls .vc-state-pause,
.video-controls .vc-volume-unmute {
    display: none;
}
```

In this bit of styling, we self-contain all video control styling to the `video-controls` class. This helps in maintaining styles in a modular pattern. Again, following responsive design principles, we tell the controls to have a width of 100 percent. Then, each type of control is set to display as inline-blocks, similar to `float`. Lastly, we tell all default controls that are not supposed to appear initially to have a display of none. Now, we need to add interactivity to our controls.

First, let's create an `App.VideoControls` class that follows our entire framework:

```
var App = window.App || {};

App.VideoControls = (function(window, document, $){
    'use strict';

    var _defaults = { };

    function VideoControls(ele, options) {
        this.options = $.extend({}, _defaults, options);
        this.ele = ele;
        this.$ele = $(ele);

        this.init();
    }
    return VideoControls;

}(window, document, Zepto));
```

As you can see, there's not much difference here. The only thing is that we now have an `init` method that gets called. This is to separate out initialization functionality elsewhere so that the constructor is not completely populated with code. Now we need to add the following defaults:

```
var _defaults = {
    // Supported Features
    'features': ['play', 'pause', 'fullscreen', 'mute', 'unmute',
'playpause'],
    // State of the controls
    'state': 'paused',
    // State of the sound
    'sound': 'unmuted',
    // Customizable Classes or Classes associated with Elements
    'classes': {
        'state': {
            'holder': 'vc-state',
            'play': 'vc-state-play',
            'pause': 'vc-state-pause'
        },
        'track': {
            'holder': 'vc-track',
            'progress': 'vc-track-progress',
            'handle': 'vc-track-handle'
        },
        'volume': {
            'holder': 'vc-volume',
            'mute': 'vc-volume-mute',
            'unmute': 'vc-volume-unmute'
        }
    },
    // Customizable Events or Dispatched Events
    'events': {
        'onPlay': 'videocontrols:play',
        'onPause': 'videocontrols:pause',
        'onFullScreen': 'videocontrols:fullscreen',
        'onMute': 'videocontrols:mute',
        'onUnmute': 'videocontrols:onUnmute'
    }
};
```

Just as a review of these defaults, the first default is a features array, allowing the developer tapping into this code to customize what we need initialized. The second default keeps the state of the controls, namely play, pause, and so on. The third is a state specifically for sound. The classes default allows us to use custom classes, thus the developer using this `videocontrols` class isn't limited to the classes we've defined in the markup. The last is an events default that defines the custom events we want to be dispatched. By including it in our defaults, the developer can now customize these as well.

[As you can notice, a lot goes into building a video player that can be reused and implemented correctly across a web application of any kind. Putting effort does help out in the end, although it is very difficult at the beginning. Now we can add and remove functionality in a much more modular way.]

Due to the fact that there's a lot of code that goes into creating custom controls that mimic those in native controls, we've decided to leave the rest of the functionality, including show/hide and triggering custom events, in the source code for your review. Don't worry though, everything is commented and if you do have questions, I encourage you to e-mail me or ask your fellow colleagues for help.

Now, we want to implement the communication between the controls and the video player. But first, we need to clean the `main.js` file a bit. So, let's remove the following code from `main.js`:

```
if(document.querySelector('video') !== 'null') {
    new App.Video({
        'element': document.getElementsByTagName('video')[0],
        'callbacks': {
            ...
        }
    });
}
```

We don't want this code in `main.js` because it will be shared among all the applications built in this book, so what we need to do is extrapolate it. For this reason, we have created another JavaScript file named `App.VideoController.js` in our `js/App` directory. This file is also included with the source code of this book.

Please open the `App.VideoController.js` file included with this book, and find the `initControls` method; it should look like the following:

```
VideoController.prototype.initControls = function() {
    // Remove Default control
    // Comment this out if you want native controls
    $(videoEle).removeAttr('controls');

    controlsEle = document.querySelector('.video-controls');

    controls = new App.VideoControls(controlsEle);

    $(controlsEle).
         on('videocontrols:play', function(){
            video.play();
        }).
        on('videocontrols:pause', function(){
            video.pause();
        }).
        on('videocontrols:fullscreen', function(){
            video.fullscreen();
        }).
        on('videocontrols:mute', function(){
            video.mute();
        }).
        on('videocontrols:onUnmute', function(){
            video.unmute();
        });

    return this;
}
```

Let's briefly review what is going on in this method to understand it better. First, we are telling our `video` element to hide its controls. This is done by removing the `controls` attribute. Then we cache our `controls` div in `controlsEle`. Next, we initialize our `App.VideoControls` class and pass it in the cached `controls` div. Finally, we add listeners to the cached video controls and listen to the custom events that we have defined in our defaults of `App.VideoControls`. These listeners then call the methods we have exposed in `App.Video` by telling the instance, `video`, to run the appropriate function.

The last issue we need to take care of is initializing this whole program. Since we removed the initialization in `main.js`, we need to start it elsewhere. The best place would be on the specific `index.html`, which would be `video/index.html`. So, let's open up this file and include the following script at the bottom of the page, right after the inclusion of `main.js`:

```
<script>
    new App.VideoController(true);
</script>
```

This was the last item to take care of. When we run our page, we should have a fully functional video player that runs off our customized controls.

Summary

Give yourself a big pat on the back, because you have accomplished quite a bit! Not only do you have a video player with customized controls, but you have built a solid video library that ties into the HTML5 specification and works on the iPhone. We have gone over the HTML5 specification for video integration, created a simple video player that used native controls, built a fully functional and modular video library, extended the video library with a controls class that customizes our experience, and finally created a controller class that hooks up both the video and the customized controls. On the way, we've taken some time to understand scopes in JavaScript, the prototype, and the usefulness of callbacks. If at any point you had some trouble with the concepts taught in this chapter, please review the source code with this book, and as always, practice makes perfect. The next chapter should be easier since we'll take the concepts we've learned here and apply them to audio.

3
HTML5 Audio

In the previous chapter, we discussed the importance of media distribution and how HTML5 has changed the web in providing both audio and video content natively in the browser. We specifically went over the HTML5 Video implementation, but we also discussed the `MediaElement` specification, which covers common APIs that are used by both video and audio.

In this chapter, we go further into the specification and abstract it, making it reusable for audio and video. But before we get to that, we will discuss the server configuration with a simple example, and then move on to more advanced implementation with a dynamic audio player and customized controls.

In this chapter, we will learn about:

- Integrating a simple HTML5 Audio example
- Configuring our server
- `MediaElement` abstraction
- Extending the `MediaElement` API for audio
- Creating a dynamic audio player
- Customizing the audio controls

Server configuration

Before we get started with the HTML5 audio element, we need to configure our server for allowing specific audio formats to play appropriately. First, let's take a moment to understand the appropriate audio formats.

Audio formats

Support for the HTML5 audio playback is similar to that of the video element, in that each browser supports different types of formats for one reason or another. Here are some tables showcasing what is supported:

- Following are the details related to desktop browsers:

Desktop browser	Version	Codec support
Internet Explorer	9.0+	MP3, AAC
Google Chrome	6.0+	Ogg Vorbis, MP3, WAV
Mozilla Firefox	3.6+	Ogg Vorbis, WAV
Safari	5.0+	MP3, AAC, WAV
Opera	10.0+	Ogg Vorbis, WAV

- Following are the details related to mobile browsers:

Mobile browser	Version	Codec support
Opera Mobile	11.0+	Device-dependent
Android	2.3+	Device-dependent
Mobile Safari (iPhone, iPad, iPod Touch)	iOS 3.0+	MPEG, MPG, MP3, SWA, AAC, WAV, BWF, MP4, AIFF, AIF, AIFC, CDDA, 32G, 3GP2, 3GP, 3GPP
Blackberry	6.0+	MP3, AAC

As we can see, there are multiple format types that are supported by various browsers, both mobile and desktop. Luckily for us, this book focuses on iPhone web applications, so for our purpose, we will only focus on delivering MP3 formats, which are supported by most browsers. Now, we need to make sure our server can play MP3s.

Audio format directives

In order to serve the correct MIME types, we need to configure our Apache server. To do this, we want to add the following directives to an `.htaccess` file:

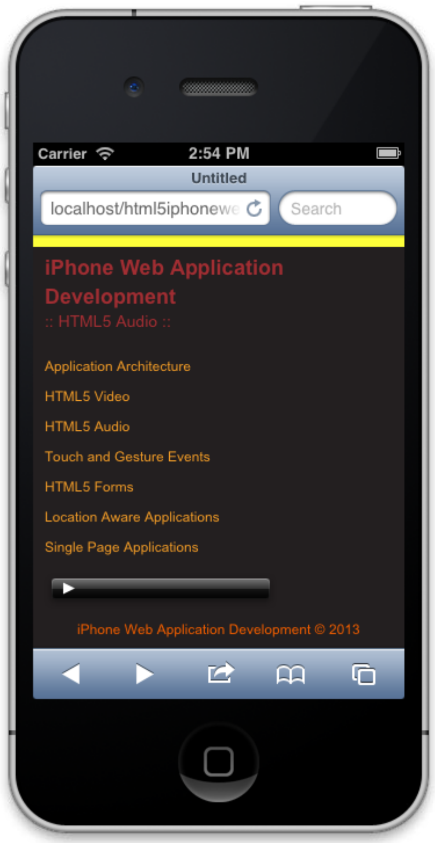
```
AddType audio/mpeg mp3
AddType audio/mp4 m4a
AddType audio/ogg ogg
AddType audio/ogg oga
AddType audio/webm webma
AddType audio/wav wav
```

Of course, for our purpose, we only need MPEG/MP3, but it's a good idea to allow these in order to take into account extensibility when supporting other browsers.

Simple HTML5 audio integration

Including audio on a page is pretty simple. We can just include the following markup in the page and we have an audio player instantly:

```
<audio controls>
    <source src="../assets/mymusic.mp3" type='audio/mpeg;
codecs="mp3"'/>
    <p>Audio is not supported in your browser.</p>
</audio>
```



The audio element

The preceding example specifies an audio element with an attribute of controls telling the browser to have an audio player with native controls for playback. Inside this element, there is a source element and a paragraph element. The source element specifies the source of the audio and its type. The attribute `src` on the source element is the relative location of the audio, and the `type` attribute specifies the MIME type and codec of the source. Lastly, we have a paragraph element, just in case the audio element is not supported.

This example is perfect to demonstrate how easy it is to have media on our pages, except it's not always as simple. Most of the time, we want complete control over our components, and sometimes it's necessary to tap into the APIs specified. We've gone over these concepts in the previous chapter, and we have developed an extensive Video class that we can use here. In the next section, we'll take a step back and abstract the code we've written thus far.

MediaElement abstraction

We've talked about how audio and video share the same API in the HTML5 specification. In this section, we'll discuss taking the video JavaScript we've written and abstracting it so that we can re-use it for audio playback.

Creating App.MediaElement.js

1. First, let's create a new JavaScript file in our `js` directory and name it `App.MediaElement.js`.
2. Next, copy over the contents of `App.Video.js` into the new `App.MediaElement.js` file.

   In this step, we want to make sure that our file reflects the `MediaElement` namespace, so we'll rename the word `Video` as `MediaElement`.

Once we've renamed everything to `MediaElement`, we probably want to remove the default elements and their names, because they won't be necessary for an abstracted class like this one. Along with those defaults, we won't need the public `fullscreen` method nor the `onFullScreen` callback.

When we make the preceding changes, our file should look like this:

```
var App = window.App || {};
App.MediaElement = (function(window, document, $){
  'use strict';

  var _defaults = {
'callbacks': {
...
}
  };

  function MediaElement(options) { ... }
  function attachEvents() { ... }

MediaElement.prototype.onCanPlay = function(e, ele) { ... }
MediaElement.prototype.onSeeking = function(e, ele) { ... }
MediaElement.prototype.onSeeked = function(e, ele) { ... }
MediaElement.prototype.onEnded = function(e, ele) { ... }
MediaElement.prototype.onPlay = function(e, ele) { ... }
MediaElement.prototype.onPause = function(e, ele) { ... }
MediaElement.prototype.onLoadedData = function(e, ele) { ... }
MediaElement.prototype.onLoadedMetaData = function(e, ele) { ... }
MediaElement.prototype.onTimeUpdate = function(e, ele) { ... }
MediaElement.prototype.getDefaults = function() { ... ;
MediaElement.prototype.toString = function() { ... };
MediaElement.prototype.play = function() { ... }
MediaElement.prototype.pause = function() { ... }
MediaElement.prototype.stop = function() { ... }
MediaElement.prototype.mute = function() { ... };
MediaElement.prototype.unmute = function() { ... };

  return MediaElement;

}(window, document, Zepto));
```

Although we've written this code previously, let's briefly review the structure of the `MediaElement` class. This class contains exposed methods that can be accessed, such as `onCanPlay`, `onSeeking`, and `onEnded`. These methods are called when the element we pass in has dispatched the appropriate event. The events we are listening to are in `attachEvents`, and they contain the shared API events, such as `canplay`, `seeking`, `ended`, and so on.

This class essentially contains only the APIs that are shared among audio and video media. If we wanted to extend it for a specific functionality, such as fullscreen, we would extend the instance of `MediaElement` or use JavaScript inheritance for the `App.Video` class.

> In this book, we do not cover true JavaScript inheritance. Given that we want to review HTML5 for iPhone web application development as a whole, we don't go into more advanced details of JavaScript architecture.

Initializing App.MediaElement.js

In order to initialize `App.MediaElement.js`, we can do the following:

```
new App.MediaElement({
    'element': someElement,
    'callbacks': {
        'onCanPlay': function(){ console.log('onCanPlay'); },
        'onSeeking': function(){ console.log('OVERRIDE :: onSeeking');
},
        'onSeeked': function(){ console.log('OVERRIDE :: onSeeked');
},
        'onEnded': function(){ console.log('OVERRIDE :: onEnded'); },
        'onPlay': function(){ console.log('OVERRIDE :: onPlay'); },
        'onPause': function(){ console.log('OVERRIDE :: onPause'); },
        'onLoadedData': function(){ console.log('OVERRIDE ::
onLoadedData'); },
        'onLoadedMetaData': function(){ console.log('OVERRIDE ::
onLoadedMetaData'); },
        'onTimeUpdate': function(){ console.log('OVERRIDE ::
onTimeUpdate'); }
    }
});
```

In the preceding code, we create a new instance of `MediaElement` and pass it an object, which gets merged with the defaults of the `MediaElement` constructor. Keep in mind that `element` will always refer either to the audio or video element. We can choose to override the default callbacks or not, as they are optional.

> Please note that we are passing in all the callbacks. This is because, since the writing of this book, `Zepto.js` contains a bug that does not do deep copying of an object if passed in the Boolean true value as the first parameter.

Now we are prepared to use this class with the audio class we've developed for this page.

Extending the MediaElement API for audio

Now that we have an abstract `MediaElement` class, we want to build on top of it to allow for audio playback. Starting with the base template we've established, we'll create an `App.Audio` class that contains all the functionality for this page; from creating an instance of `MediaElement`, to creating a drop-down menu for switching tracks and managing the volume of each track.

The base template

We can establish a base template by following the pattern we have previously established. Here is some code you can start out with as a template:

```
var App = window.App || {};

App.Audio = (function(window, document, $){
  'use strict';

  var _defaults = {
    'element': 'audio',
    'name': 'Audio'
  };

  function Audio(options) {
    this.options = $.extend({}, _defaults, options);

        this.element = this.options.element;
        this.$element = $(this.element);

        attachEvents.call(this);
  }

    function attachEvents() { }

  Audio.prototype.getDefaults = function() { ... };

  Audio.prototype.toString = function() { ... };

  return Audio;

}(window, document, Zepto));
```

Nothing is new here, we are using the same pattern that we've used previously; establishing an `App.Audio` class, an IIFE that contains the `Audio` constructor, the same `attachEvents` method to contain the events to handle, and some prototype methods that extend `Audio` (`getDefaults` and `toString`). We continue to use `Zepto` and pass in both `window` and `document` into the IIFE as a reference and then self contain our code.

Creating an instance of MediaElement

In our constructor, we need to do two things. One, we need to get the audio element on the page and cache it. And two, we need to create or initialize an instance of MediaElement based on the element on the page.

Finding and caching an audio element

To find the audio element and cache it, we can do the following:

```
this.audioElement = document.getElementsByTagName('audio')[0];
this.$audioElement = $(this.audioElement);
```

Remember that the `this` keyword is in reference to the instance of `audio` that gets returned to `App.Audio`. We then create a property on `this` called `audioElement`, which gets set to the first audio element found on the page.

Note that `getElementsByTagName`, which exists on the document, accepts one parameter, a string. This method gets all the elements on the page that match that tag in an array. In this instance, we only have one audio element on the page, so we get an array with one element found. Thus, we use `[0]` to get the first instance in that array.

Once we have the audio element, we cache it as a `Zepto` object so that we use `Zepto` once, thus increasing the performance of our application. I usually do this in most of my projects because I find myself using many of Zepto's built-in methods, especially for creating event listeners. However, this can be skipped if you don't find it useful in your case.

Initializing MediaElement

Now that we have our audio element, we can follow the code we wrote in the last section on how to initialize `MediaElement`. So you don't have to flip back, here's the code we can use:

```
this.mediaElement = new App.MediaElement({
    'element': this.audioElement,
    'callbacks': {
        'onCanPlay': function(){ ... },
        'onSeeking': function(){ ... },
        'onSeeked': function(){ ... },
        'onEnded': function(){ ... },
        'onPlay': function(){ ... },
        'onPause': function(){ ... },
        'onLoadedData': function(){ ... },
        'onLoadedMetaData': function(){ ... },
        'onTimeUpdate': function(){ ... }
    }
});
```

This is the same code we wrote previously, and the ellipsis in the callbacks should contain `console.log` we had written. The one thing you should notice is that we pass in `this.audioElement`, our cached audio element, into this instance of `MediaElement`. Also, we have now created a reference to the instance of `MediaElement` with `this.mediaElement`. Now we can publicly control the audio from the instance of `App.Audio` we will create later on.

At this point, we have a fully functional audio player built off our abstracted class of `MediaElement`. However, there's not much going on; we only have a setup that works and is extensible, but it is not in any way unique. This is where our dynamic audio player will come in to play.

Dynamic audio player

So at this point, we have an audio class that extends our `MediaElement` object with events that are exposed and therefore available for us to create dynamic content. Now, let's have some fun and create a dynamic audio player that switches tracks.

The select element

Originally, when we created this application in *Chapter 1*, *Application Architecture*, we created a navigation that is contained by anchor tags and list elements. Although this works perfectly on desktop and perhaps the iPad, it is not suitable for smaller screen devices like the iPhone. For this reason, the `select` element brings up a native component that allows you to easily navigate choices that you can select.

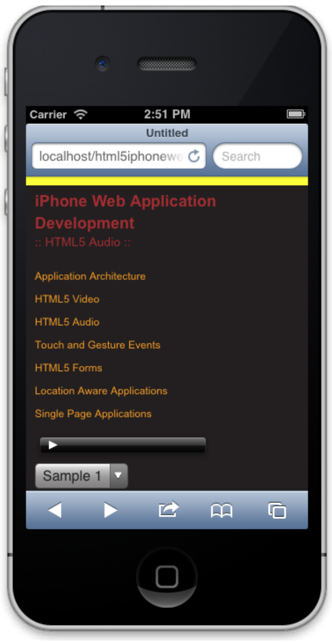
Apple's developer documentation suggests that we use the `select` element in our apps, because it has been optimized as a custom control within iOS (`http://goo.gl/T3xuY`). This is extremely useful, because it allows us to keep within the guidelines for web application design for iOS.

Now let's go ahead with the implementation. First we need to make sure to add the `select` element to our page. Right now, you should have the following markup:

```
<div class="audio-container">
    <audio controls preload>
        <source src="../assets/sample.mp3" type='audio/mpeg;
codecs="mp3"'/>
        <p>Audio is not supported in your browser.</p>
    </audio>
</div>
```

What we need to do is add the select element after the `audio` tag, like so:

```
<div class="audio-container">
    <audio controls preload>
        <source src="../assets/nintendo.mp3" type='audio/mpeg;
codecs="mp3"'/>
        <p>Audio is not supported in your browser.</p>
    </audio>
    <select>
        <option value="sample1.mp3" selected>Sample1</option>
        <option value="sample2.mp3">Sample2</option>
        <option value="sample3.mp3">Sample3</option>
    </select>
</div>
```



A select element

In the preceding code, we have added a select element that contains multiple options. These options have a `value` attribute, and the first option also contains a `selected` attribute. The value attribute should contain the track you have in your assets, and the selected attribute tells the `select` to have that option automatically selected on page load.

> In this example, we assume that all our audio is MP3. In your case this may be different, and if so, we would need to build logic into the code we will write to handle this logic. Because of the complexity that would be introduced, we focus exclusively on handling audio tracks that have a MIME type of MP3.

Switching audio tracks

Now that we have a `select` element on the page listing out the several audio tracks in an iOS suggested manner, we now want to make our player dynamic based on user input. To do this, we need to create an event listener to handle the change event.

The change event listener

The `select` element has a specific event we can listen to, namely the `change` event. This is fairly easy to accomplish with `Zepto` and our cached instance of the audio element. To add the listener, let's go to our `attachEvents` method in `App.Audio` and add the following bit of code:

```
var that = this;
this.$element
    on('change', 'select', function(e) { onSelectChange.call(that, e);
});
```

First we are creating a variable called `that`, which refers to the instance of audio. Then we get the cached element we created in the constructor and delegate the `change` event coming from any `select` element on the page. When the `change` event fires, we call the anonymous function, namely the third parameter in the `on` method. Inside this anonymous function we call a method, which we have not created, called `onSelectedChange`, and pass it in the event or the `e` reference.

> We are using Zepto's `on` method. This method can accept various parameters similar to jQuery's `on` method, but in this case, we send in the event we want to listen to, the element it should be coming from, and then finally a function that should be called. On top of this, our anonymous function is doing a call on the method, which we have discussed previously, but essentially it changes the reference of this to be audio.

The change event handler

Once we have created the listener for the `change` event, we need to define the handler. We haven't created this yet, but it entails some fairly complex functionality. Initially, this should be fairly easy now that we have an API via the `MediaElement` instance. However, we only have one audio element on the page, so we need to be able to use that one element for playback. Thus, we need to do the following in our handler:

- Create a temporary reference to our cached audio element
- Stop the playback of the audio, even if it is not playing
- Clone the cached audio element to the temporary reference
- Remove the audio element from the DOM
- Delete the cached media element, audio element, and Zepto audio element
- Change the source of the cloned audio element
- Append the cloned audio element to the DOM
- Recreate the cached media element, audio element, and Zepto audio element

Yes, this sounds like a lot of work to do in order to keep a single audio element on the page, but the code to do this is very little and involves some copy and paste because we have already written it. So, let's write some magic!

Under the event handler section, we want to include the following method:

```
function onSelectChange(e) {
    var $tempAudioElement;
    // Stop the song from playing
    this.mediaElement.stop();
    // Store the element temporarily
    $tempAudioElement = this.$audioElement.clone();
    // Now remove the element
    this.$audioElement.remove();
    // Remove from memory
    //-----
    delete this.mediaElement;
    delete this.audioElement;
    delete this.$audioElement;
    //-----

    // Change the temporary audio source
    $tempAudioElement.
        find('source').
            attr('src', '../assets/' + e.target.selectedOptions[0].
value);
```

```
        // Now attach it to the DOM
        this.$element.prepend($tempAudioElement);
        // Reset the audioElement
        this.audioElement = document.getElementsByTagName('audio')[0];
        this.$audioElement = $(this.audioElement);
        // Reset the mediaElement
        this.mediaElement = new App.MediaElement({
            'element': this.audioElement,
            'callbacks': {
                'onCanPlay': function(){ ... },
                'onSeeking': function(){ ... },
                'onSeeked': function(){ ... },
                'onEnded': function(){ ... },
                'onPlay': function(){ ... },
                'onPause': function(){ ... },
                'onLoadedData': function(){ ... },
                'onLoadedMetaData': function(){ ... },
                'onTimeUpdate': function(){ ... }
            }
        });
    }
```

If we go ahead and run the code in the browser, we should be able to switch between audio tracks with no problem. If you do have an issue, please refer to the provided source code.

Either way, the preceding code does exactly what we wanted. If we analyze the code closely, we can see that we are essentially tapping into the `MediaElement` class when we stop the playback. This is an example of how easy it is to play around with media elements such as audio and video now that it has been abstracted. Also note that we are using quite a few of the Zepto methods, including `clone`, `remove`, `prepend`, and `attr`. These are all methods that are useful, which is precisely the reason we cache our audio element.

You may be asking yourself what the `delete` section does in our preceding code. Essentially, this helps with garbage collection; it tells the JavaScript engine that we no longer need this and so you can recollect it. Yes, we can set them to the new values after we prepend the new audio element, but this is a sure fire way of starting fresh and not leaving anything up to guessing from the JavaScript engine.

There is a problem with the code we've written, and that is the redundancy of the recreation of the `audioElement`, `$audioElement`, and `mediaElement` objects. As we've defined this functionality previously in our constructor, we can refactor to make sure our functionality is all located in one place—this is what the next section looks at. If you already understand what is meant by the refactor of this code, you can skip this part.

Refactoring our code

Since we have the same code in two places, we are starting to introduce some redundancy. To make our application a bit more manageable, we should centralize the same functionality into one location. Doing this is not complex, and it is simpler than you would probably imagine.

For our refactor, we only need to write one method, a `setAudioElement` method. This method should be private and only available within the `Audio` class, and it should only contain the code necessary for creating the references to the `audioElement`, `$audioElement`, and `mediaElement` objects.

To do this, create the following method in our private methods section:

```
function setAudioElement() {
    return this;
}
```

Now copy the following code from the constructor, and paste it into `setAudioElement`:

```
this.audioElement = document.getElementsByTagName('audio')[0];
this.$audioElement = $(this.audioElement);

this.mediaElement = new App.MediaElement({
        'element': this.audioElement,
        'callbacks': {
            'onCanPlay': function(){ ... },
            'onSeeking': function(){ ... },
            'onSeeked': function(){ ... },
            'onEnded': function(){ ... },
            'onPlay': function(){ ... },
            'onPause': function(){ ... },
            'onLoadedData': function(){ ... },
            'onLoadedMetaData': function(){ ... },
            'onTimeUpdate': function(){ ... }
        }
});
```

Once we've done this, let's call `setAudioElement` within our constructor:

```
function Audio(options) {
    // Customizes the option
    this.options = $.extend({}, _defaults, options);
    //Cache the main element
    this.element = this.options.element;
```

```
        this.$element = $(this.element);
        // Sets the audio element objects
        setAudioElement.call(this);
        attachEvents.call(this);
    }
```

If we run our application now, it should run like normal, as if we had not changed anything. Now we need to replace the repeated code in the `select` handler to call the same method:

```
    function onSelectChange(e) {
        ....
        // Now attach it to the DOM
        this.$element.prepend($tempAudioElement);

       setAudioElement.call(this);
    }
```

Now that we've done all the refactoring we need, let's run the application on our iPhone simulator. When the page runs and you switch between audio tracks, you should not encounter any issues. There's nothing surprising here, but the cool thing is that you now a have common code centralized in one location. This is the essence of refactoring, and it helps achieve a maintainable codebase.

Initializing our Audio class

Up to this point, we've focused on the development of the `Audio` class. That's fine, but now we need to initialize all of this code.

To do this, open up the `index.html` file for the **Audio** page. This should be located at `/audio/index.html`. Once we've opened up that file, scroll to the bottom of the source code and add the following script after all the other scripts:

```
    <script>
        new App.Audio({
            'element': document.querySelector('.audio-container')
        });
    </script>
```

This is a bit different from how we initialized `App.Video`, in that we now pass in the element while `App.Video` finds the video elements within it. The reason behind the difference was to show how we can initialize our classes differently. It's up to you on how you want to initialize a class. Each has its advantages and disadvantages, but it's good to be aware of the alternatives and choose the best one your code style and project needs.

Now we have a dynamic audio player running off an abstract `MediaElement` class. On top of that, we have created a UI that is effective for this purpose and executes what is expected. But what if we want clearer control of the audio besides what is provided in the default interface? In the next section, we discover how to control our audio using the `MediaElement` class we've created previously.

Customizing HTML5 audio controls

In this section, we'll cover how to customize the controls of our audio player. As we've seen in our video player discussed in the previous chapter, it can be fairly useful to create a custom experience. For this book, we've kept it fairly simple, and we will continue following that pattern so that we can discuss the principles and get you started quickly. For audio, customizing the controls is even simpler, especially since we do not have control over the volume, which is discussed further in the following section.

Creating custom media controls

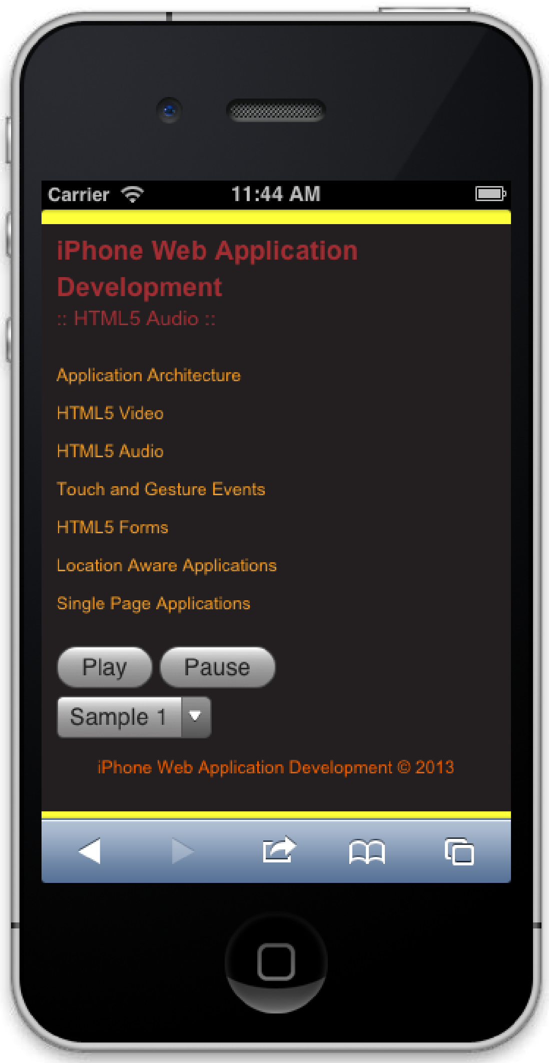
First, let's remove the `controls` attribute from our `audio` element. When we do this, you should have the following markup:

```
<audio preload>
    <source src="../assets/sample1.mp3" type='audio/mpeg;
codecs="mp3"'/>
    <p>Audio is not supported in your browser.</p>
</audio>
```

Now we need to add custom controls to the markup. We can continue doing the same thing we did in the previous chapter, except this time we abstract it with a media-controls class and keep it simple by just having a play and pause button. This should also go after the `audio` element. When we are finished, the markup should look like this:

```
<div class="media-controls">
    <div class="mc-state">
        <button class="mc-play mc-state-play">Play</button>
        <button class="mc-pause mc-state-pause">Pause</button>
    </div>
</div>
```

When you check out the application on the iPhone simulator, it should look like this:



iPhone Web Application Development

:: HTML5 Audio ::

Application Architecture

HTML5 Video

HTML5 Audio

Touch and Gesture Events

HTML5 Forms

Location Aware Applications

Single Page Applications

iPhone Web Application Development © 2013

Custom controls

What you will notice is that we now have no audio element being displayed on the page. This is because we have taken out the `controls` attribute. Don't worry about it too much; this is the expected behavior on iOS. Normally, you would create all the controls for your audio player, but for now we'll just do play and pause. As a bonus, you would probably want a track as well, but that is for a much larger discussion and does not fit the scope of this book.

Adding interactivity to our customized controls

This is where all the magic happens. We will now connect the interactivity we've built into the `MediaElement` class in order to customize our experience.

First, let's go to our `App.Audio` JavaScript file and find the `attachEvents` method. To keep it short and simple, let's include the following bit of code after our `change` event listener:

```
this.$element.
    find('.media-controls').
        on('click', '.mc-play', function() { that.mediaElement.play();
}).
        on('click', '.mc-pause', function(){ that.mediaElement.
pause(); });
```

The preceding code uses the cached `$element` to find the media controls, and then attaches the clock event onto the play and pause buttons accordingly. Inside each event listener we use the instance of `mediaElement` that has been created within the `setAudioElement` method to call the `play` or `pause` methods.

One thing to note is that our event listeners are using `that` in order to reference the instance of `mediaElement`. If you recall, we created the `that` variable at the top of the `attachEvents` method so that we have a reference of `this` inside of the event listener. As we've explained previously, JavaScript has function scope, so when we create our event listener, that function has created a new scope that sets the relation of `this` to the scope of the event. Behind the scenes, Zepto sets `this` to the target element, which would be either the `play` or `pause` element.

This is all we need to make customized controls that play and pause our audio. If we now test the application, we should be able to switch between tracks, play our track, and pause the track.

Sequential playback

In this section, we take a look at how we can build a preliminary playlist. Although this section is more bonus material, it's useful when creating some sort of a music player application where there are playlists of music we want to play sequentially. At first, it may be difficult to understand how we can do this, especially given the fact that we need user input to enable playback, but it's not really an issue. Because the load and play methods are initiated on the first song, we can just switch the source, load it, and then play the track. So let's go through it step-by-step.

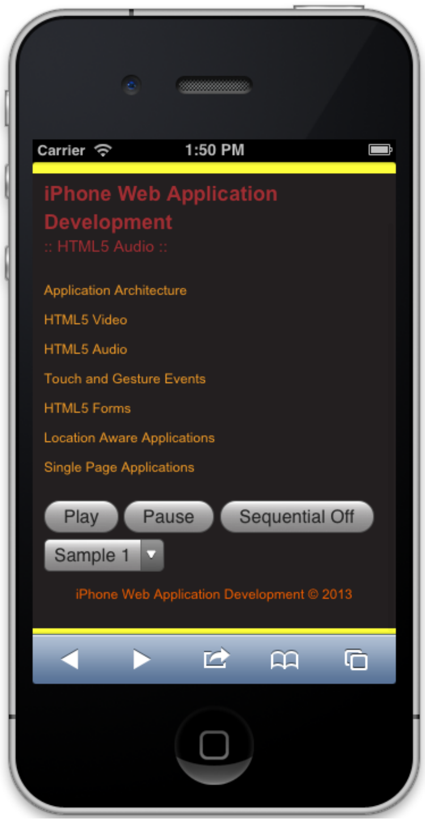
The markup

We don't really want to play music sequentially by default, this should be user initiated based on good user experience design. So, let's include another button for the user to enable or disable this feature:

```
<div class="mc-state">
    <button class="mc-play mc-state-play">Play</button>
    <button class="mc-pause mc-state-pause">Pause</button>
    <button class="mc-sequential mc-sequential-off mc-state-
sequential">Sequential Off</button>
</div>
```

All we've done in the preceding code is added another button after the play and pause buttons. This button contains the appropriate three classes that we need and the text `Sequential Off`, because we only want users to enable this feature if they want to.

When your markup is all set, you should have the following interface:



The sequential button

The JavaScript

There's a bit of work to do here, but nothing overly complicated. Here's what we want to do as a checklist:

- Create a default setting for sequential playback, and set it to false
- Create a `handleOnAudioEnded` method, with a parameter for the `Audio` class
- Call the `handleOnAudioEnded` method within the `onEnded` callback of the MediaElement initialization
- Inside the `handleOnAudioEnded` method, we should check if sequential playback is enabled
- If sequential playback is enabled, we want to update the select menu and reload the audio element
- Finally, we want to listen for the click event on the new sequential button to enable or disable this feature, while also updating the button's status as well

So, first things first, let's create the sequential default setting:

```
var _defaults = {
    'element': 'audio',
    'name': 'Audio',
    'sequential': false
};
```

Nothing too crazy, we've just added a default setting called `sequential` and set it to `false`. Next, we want to create the `handleOnAudioEnded` method that contains the functionality we've listed previously:

```
function handleOnAudioEnded(Audio) {
    if(Audio.options.sequential) {
        var $select = Audio.$element.find('select'), $next;

        // Go to next in playlist
        $next = $select.
            find('option[selected]').
                removeAttr('selected').
                    next().
                        attr('selected', 'selected');

        // Change the Selected Index
        $select[0].selectedIndex = $next.index();

        // Must be made on the audio element itself
        Audio.audioElement.src = '../assets/' + $select.val();
```

```
            Audio.audioElement.load();
            Audio.audioElement.play();
        }
    }
```

Don't worry if you don't understand the preceding code, just take the following points into account:

- The only parameter we are passing is an instance of `Audio`
- We then check if `sequential` is enabled
- Once we've verified we want sequential playback, we create two variables: `$select`, which caches the select element, and `$next`, which will cache the next song in the playlist
- Then we set the `$next` element while removing the `selected` attribute from the current option
- We update the `select` menu by setting the `selectedIndex` to the next option in `select`
- Finally, we update the audio elements source directly, load that source, and set the status to playing

This method handles the playback of the next source we want to play. We could probably improve this by adding the functionality to change the source, load, and play inside the `MediaElement` class, but I'll leave that up to you to decide and extend as needed. We could also possibly cache `select` at the class level (`Audio`), instead of doing it each time we want to play sequentially.

Note that we have not added any error checking. Currently, this code doesn't do a check for when we get to the end of the list. Perhaps we want it to loop, or maybe we want to inform the user the playlist is done? There are many use cases that we can execute here, but you get the point, which is that we can have a playlist in our application if we wanted.

Next we want to call the preceding method we created when we pass in `callbacks` to the initialization of the media element. As you may recall, we put this in our `setAudioElement`, and therefore we want to update the initialization to the following:

```
this.mediaElement = new App.MediaElement({
    'element': this.audioElement,
    'callbacks': {
        ...
        'onEnded': function(){ handleOnAudioEnded(that); },
        ...
    }
});
```

All we did here was update the `onEnded` method by calling `handleOnAudioEnded` and passing in `that`, which is a reference to the instance of the `Audio` class. Now, all we need to do is add our event listener for when the user wants sequential playback, which can be added in our `attachEvents` method:

```
this.$element.
    find('.media-controls').
        on('click', '.mc-play', function() { that.mediaElement.play();
}).
        on('click', '.mc-pause', function() { that.mediaElement.
pause(); }).
        on('click', '.mc-sequential', function(e) {
handleSequentialClick(e, that); });
```

The preceding code basically shows that we have added a `click` event listener to our sequential button, and all it does is call the `handleSequentialClick` method that accepts an event and the instance of audio with the `that` variable we created previously. Notice how we haven't created the `handleSequentialClick` method? Well, here it is:

```
function handleSequentialClick(e, Audio) {
    var $this = $(e.target);

    if(!Audio.options.sequential) {
        Audio.options.sequential = true;
        $this.
            removeClass('mc-sequential-off').
            addClass('mc-sequential-on').
            text('Sequential On');
    } else {
        Audio.options.sequential = false;
        $this.
            removeClass('mc-sequential-on').
            addClass('mc-sequential-off').
            text('Sequential Off');
    }
}
```

Put simply, this method just updates the default `sequential` option to `true` or `false`, toggling the value depending on the previous status. The method also toggles the classes of the button and the inner text, updating the user based on their interaction.

iOS considerations

Up to this point, we have customized much of the experience for both the video and audio elements. This will work perfectly for desktop devices, but there are a few points we need to take into account when working with touch devices, such as the iPhone and iPad. The good thing is that these are points that are consistent across all iOS devices, and as such should be something we take into account.

Volume

We can set the volume for audio and video elements from `0` to `1`, and we can maintain the state of the volume in our `MediaElement` library. This is good practice for an overall architecture. However, on iOS, the volume is under the user's physical control—the volume button we interact with on almost any device.

As per Apple's documentation (`http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html#//apple_ref/doc/uid/TP40009523-CH5-SW11`):

> *On iOS devices, the audio level is always under the user's physical control. The volume property is not settable in JavaScript. Reading the volume property always returns 1.*

Basically, we can't set the volume property; it will always return `1`. This is so that we don't manipulate the user's volume, and as such is only set via the user's volume control button.

Autoplay

In our application, we've also seen an example of autoplay, where we play the audio after we select a different track in our `select`. This works perfectly on a desktop, but not so much on iOS. There's a reason for this, and it's basically to protect the user's cellular data usage. This is a design decision on Apple's part, and is something we could see on other devices as well.

As per Apple's documentation (`http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html#//apple_ref/doc/uid/TP40009523-CH5-SW8`):

> *Autoplay is disabled to prevent unsolicited cellular download.*

It also states that (`http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html#//apple_ref/doc/uid/TP40009523-CH5-SW4`):

> *In Safari on iOS (for all devices, including iPad), where the user may be on a cellular network and be charged per data unit, preload and autoplay are disabled. No data is loaded until the user initiates it. This means the JavaScript play() and load() methods are also inactive until the user initiates playback, unless the play() or load() method is triggered by user action. In other words, a user-initiated Play button works, but an onLoad="play()" event does not.*

Simultaneous playback

You might be asking why we didn't go into more complicated experiences, including multiple video playback or audio tracks playing at the same time. Well, there's a good reason for that too, and it's basically because iOS limits the playback of audio or video streams to one at a time. This also comes back to the fact that we don't want to use more cellular data than necessary on the page.

As per Apple's documentation (`http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html#//apple_ref/doc/uid/TP40009523-CH5-SW10`):

> *Currently, all devices running iOS are limited to playback of a single audio or video stream at any time. Playing more than one video—side by side, partly overlapping, or completely overlaid—is not currently supported on iOS devices. Playing multiple simultaneous audio streams is also not supported.*

There are more considerations that should be taken into account when developing an iOS web application that supports audio and video media playback. We can continue covering this here, but I encourage you to visit Apple's documentation, *iOS-Specific Considerations* (`http://developer.apple.com/library/safari/#documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html`), to review all the necessary considerations. The previously mentioned documentation snippets should cover some of the concerns you had while developing the video and audio sections of this book, but it's always good to be aware of all the issues that come up.

Summary

In this chapter, we've reviewed the media element API from the perspective of audio playback on iOS. From abstracting the previous code into a `MediaElement` class making it reusable for both audio and video, to customizing the controls of the audio element, we have created a dynamic audio player that works and is constructed in a modular pattern. Along with creating an audio player, we have reviewed the considerations that must be made on an iOS device, such as the control of volume and the limitation of simultaneous playback. I hope that this chapter has helped you get started experimenting with audio, and has helped you understand that we can consolidate code and focus on delivering features that are critical within our applications by abstracting our code. In the next chapter, we shift to how we can use touch and gestures to create unique user experiences that go beyond just clickable buttons.

4
Touch and Gestures

Creating iPhone web applications, by default, involves touch interaction. This is obvious, and luckily Apple has done a great job getting us up and going quickly by mapping clicks to touch events by default. However, what if we wanted a slideshow that reacted to a swipe from the user? Or, what if we wanted to zoom into a photo, without affecting the layout of a page, when the user pinched within a defined area of our application? Well, that's all up to us as developers.

In this chapter we'll go over touch events and gestures, and use the technology to build a slideshow that is responsive to a user's touch and gestures. Most of the concepts here will be basic to help you understand these new events that were not common in traditional web development. However, we'll also dive into some more advanced features using the pinch gesture to zoom in and out of an image. But, first we need to do some adjusting to our app by reorganizing our navigation, so that it does not take up most of the screen real estate, from there we'll start our dive into touch and gestures.

In this chapter, we will cover:

- Simplifying our navigation
- Creating a responsive photo gallery
- Listening to and handling touch events
- Interpreting touch events
- Responding to gestures
- Extending touch events as a plugin

Simplifying the navigation

Our navigation is currently taking up some serious real estate, and although it worked for our previous examples, it just won't work well with the rest of the examples for this book. So, first we need to clean up this application in order to focus on the actual content of our application. We'll clean up our markup to use the `select` component. Then we'll add interactivity, so that our `select` element actually switches between pages.

Before we start coding, create an `App.Nav.js` file in our JavaScript directory. Once the file is created, let's include it at the bottom of our page with the following script tag:

```
<script src="../js/App/App.Nav.js"></script>
```

Navigation markup and styling

In this section of the chapter we look at reworking the navigation for our applications. In most cases we want to make sure to use native controls on the device, so the goal here is to provide the user the ability to use the custom select control in iOS, but give us the same flexibility to customize the look and feel while having the same interaction. We'll modify the markup, look at customizing controls, and then simulating the same experience.

The basic template

First, let's get rid of the anchor tags that we are using within the navigation. Once we're done removing those links, let's create a `select` element with options and have the values point to the appropriate pages:

```
<nav>
    <select>
        <option value="../index.html">Application Architecture</
option>
        <option value="../video/index.html">HTML5 Video</option>
        <option value="../audio/index.html">HTML5 Audio</option>
        <option value="../touch/index.html" selected>Touch and Gesture
Events</option>
        <option value="../forms/index.html">HTML5 Forms</option>
        <option value="../location/index.html">Location Aware
Applications</option>
        <option value="../singlepage/index.html">Single Page
Applications</option>
    </select>
</nav>
```

In the preceding code we have replaced anchor tags with a `select` element with options. Each option has a value that points to the specific page and within the option is the chapter name. Since we've removed the anchor tags, we need to adjust the styling.

Styling a select component

There's not much we need to do here, but remove the styling we had in place. Although it's not necessary, for best practice, you always want to remove unused styling. This helps increase the performance of your application by lowering the page load.

So let's remove the following styles:

```
/* --- NAVIGATION --- */
nav ul {
    padding: 0;
}
nav li {
    list-style: none;
}
nav a {
    display: block;
    font-size: 12px;
    padding: 5px 0;
}
```

Now, we need to add the interactivity that mimics the default actions of an anchor tag.

Navigation interactivity

Mimicking the default behavior of an anchor tag is pretty simple. Let's start by creating a basic template, such as those we've done in previous chapters, then cache the navigation and add the behavior to switch between pages. So let's get started!

The basic template

Following is our default template. As before, it's just a simple IIFE that establishes a class for our navigation. This closure accepts the `window`, `document`, and `Zepto` objects, aliasing the `Zepto` object to the dollar sign.

```
var App = window.App || {};

App.Nav = (function(window, document, $){

  var _defaults = {};

  function Nav() {}

  return Nav;

}(window, document, Zepto));
```

Caching our navigation

Now, we could just use Zepto to find the navigation in the DOM each time we need it. But following our best practices, we can cache the navigation and have a variable contained in the closure scope that can be used by private and public methods.

```
var _defaults = {},
  $nav;

function Nav() {
  $nav = $('nav');
}
```

In the preceding code, we have created a `$nav` variable that is contained within the closure scope, so we can now reference it within all methods contained in this closure. Then in the constructor we set the variable to the `nav` element.

Listening and handling the change event

Now the fun begins. We need to listen to when the `select` element's change event is fired. We have done this before for our audio player. However, we'll briefly go over how to do this here in case you haven't been following along in order.

First, let's call an `attachEvents` method that we will define next:

```
function Nav() {
  $nav = $('nav');

  attachEvents();
}
```

Now that we are calling the `attachEvents` method, we need to create it. In this method, we want to listen to the change event and then handle it:

```
function attachEvents() {
  $nav.
    on('change', 'select', handleSelectChange);
}
```

In the preceding code we use Zepto's `on` method to tell the cached navigation to listen to the change event on a `select` element, which is contained within the navigation. From there we assign a method we have not created, `handleSelectChange`. This method is a handler that we will define next.

Finally, we need to define our handler. All this handler needs to do is switch pages based on the changed value of the `select` element.

```
function handleSelectChange(e) {
  window.location = this.value;
}
```

The preceding handler accepts the event parameter, but we actually don't use it. You can remove this parameter, but usually I like to keep the parameters a handler accepts. Either way, we are telling the window object to switch locations by setting `window.location` to the value that the `select` element has been changed to.

Note that we are using `this.value` to set the location of the window object. In this case, `this` refers to the select element itself or the element the on event targets.

Initializing the navigation

Finally, all we need to do is initialize this class. Because this navigation will theoretically be on every page of our application, we can immediately create a new instance of `App.Nav` after we have created this call. So let's add the following code at the end of `App.Nav.js`:

```
new App.Nav();
```

This is all we need to mimic the behavior of our previous anchor tags. With this done, we now have plenty of screen real estate to proceed with touch events. Next, let's discuss touch events and gestures on the iPhone.

Touch and Gesture events

Touch events are easy to handle on the iPhone; however, there are a couple of "gotchas" when you start diving into when events are fired and how they are interpreted in certain situations. Luckily for us, gestures are also easily implemented with the `GestureEvent` object. In this section we'll go over touch and gestures in general, getting a fundamental understanding of the technology behind these user experiences so that in the next section, we can successfully create a swipeable slideshow.

Touch events

Touch events include one or more inputs received by your mobile device. In this book we'll focus on up to two-finger events that we can handle in several ways. iOS does a great job at interpreting these inputs; however, elements can be either clickable or scrollable as described by Apple's Developer documentation (`http://developer.apple.com/library/ios/#documentation/AppleApplications/Reference/SafariWebContent/HandlingEvents/HandlingEvents.html#pageTitle`):

> *A clickable element is a link, form element, image map area, or any other element with mousemove, mousedown, mouseup, or onclick handlers. A scrollable element is any element with appropriate overflow style, text areas, and scrollable iframe elements. Because of these differences, you might need to change some of your elements to clickable elements, as described in "Making Elements Clickable," to get the desired behavior in iOS.*
>
> *In addition, you can turn off the default Safari on iOS behavior as described in "Preventing Default Behavior" and handle your own multi-touch and gesture events directly. Handling multi-touch and gesture events directly gives developers the ability to implement unique touch-screen interfaces similar to native applications. Read "Handling Multi-Touch Events" and "Handling Gesture Events" to learn more about DOM touch events.*

This is essential to keep in mind because, depending on the kind of functionality we require, certain elements behave differently by default. If we want to modify this functionality, we need to override the defaults by attaching certain events to those elements, as described earlier. By preventing the default functionality and overriding it with our own, we can create experiences that are greatly customized to our needs. An example of this would be creating a full page parallax experience that plays an animation as we scroll.

Once we know what kind of behavior we want, there are a couple of important things we need to keep in mind. For example, events are conditional, so depending on the user interaction some gestures might not generate any events. Let's take a look at some of these events.

On scroll

A good example of a conditional event is when the user scrolls a page. In this interaction the scroll event only fires when the page stops moving and redraws. For this reason, on most parallax-driven sites the default behavior is prevented on the page and a custom scroll solution is implemented.

On touch and hold

When a user touches a clickable element and holds down their finger, an information bubble is displayed. But if you were hoping to catch this gesture, you're out of luck. Based on the official Apple documentation, no events are dispatched during this type of interaction.

On double-tap zoom

In this interaction, the user double-taps the screen and the page zooms in. You would think that there would be an event for this type of interaction, but again we don't have any events we can tie into.

If we keep in mind the exceptions discussed earlier, we should be good with developing our application and handling our touch events correctly. Now we need to know what events we can tie into for touch, including how to listen and handle them appropriately.

Supported touch events and how they work

The Apple documentation officially lists out all events that are supported on iOS, including the following touch and gesture events and when they were supported:

Event	Generated	Conditional	Available
`gesturestart`	yes	N/A	iOS 2.0 and later
`gesturechange`	yes	N/A	iOS 2.0 and later
`gestureend`	yes	N/A	iOS 2.0 and later
`touchcancel`	yes	N/A	iOS 2.0 and later
`touchend`	yes	N/A	iOS 2.0 and later
`touchmove`	yes	N/A	iOS 2.0 and later
`touchstart`	yes	N/A	iOS 2.0 and later

Based on the preceding list, we've got everything we need in order to make complex user experiences on the iPhone using mobile Safari. If you were worried how these events are handled, there's no need to be, based on the development documentation by Apple (`http://developer.apple.com/library/ios/#documentation/AppleApplications/Reference/SafariWebContent/HandlingEvents/HandlingEvents.html`) these events are delivered in the same way as any other browser:

> *Mouse events are delivered in the same order you'd expect in other web browsers (…). If the user taps a nonclickable element, no events are generated. If the user taps a clickable element, events arrive in this order: mouseover, mousemove, mousedown, mouseup, and click. The mouseout event occurs only if the user taps on another clickable item. Also, if the contents of the page changes on the mousemove event, no subsequent events in the sequence are sent. This behavior allows the user to tap in the new content.*

Now that we have a good understanding of one-finger touch events, including the exceptions and the way they work, we should take some time to understand gestures.

Gestures

Technically, gestures are touch events and so the preceding information also applies to single-touch events because panning, zooming, and scrolling are all considered gestures. But, gestures are also complex interactions that can be interpreted differently. Based on the Apple documentation (`http://developer.apple.com/library/ios/#documentation/AppleApplications/Reference/SafariWebContent/HandlingEvents/HandlingEvents.html`) we can combine multi-touch events to create custom gestures;

> *Typically, you implement multi-touch event handlers to track one or two touches. But you can also use multi-touch event handlers to identify custom gestures. That is, custom gestures that are not already identified (...)*

We've seen from the chart in the previous sections that we can listen to gestures and thus create custom experiences; however, one thing that is confusing about gestures and normal touch events is when they happen. But this is not a mystery, because Apple's documentation (`http://developer.apple.com/library/safari/#documentation/UserExperience/Reference/GestureEventClassReference/GestureEvent/GestureEvent.html#//apple_ref/javascript/cl/GestureEvent`) provides the following information to us:

(...) for a two finger multi-touch gesture, the events occur in the following sequence:

1. touchstart for finger 1. Sent when the first finger touches the surface.

2. gesturestart. Sent when the second finger touches the surface.

3. touchstart for finger 2. Sent immediately after gesturestart when the second finger touches the surface.

4. gesturechange for current gesture. Sent when both fingers move while still touching the surface.

5. gestureend. Sent when the second finger lifts from the surface.

6. touchend for finger 2. Sent immediately after gestureend when the second finger lifts from the surface.

7. touchend for finger 1. Sent when the first finger lifts from the surface.

From the preceding information, we can gather that both touch and gesture events go hand-in-hand. This allows us to do some interesting things on the frontend without any guesswork. But, how do we do this? Well, the next section tackles this by creating a photo gallery that responds to both touch and gestures.

Creating a responsive photo gallery

We'll get a better understanding of touch and gesture events if we focus on small pieces of functionality that we have already seen in traditional mobile applications, such as an interactive slideshow. We've seen this everywhere, a slideshow with next and previous buttons but also the ability to swipe from left to right or vice versa. The buttons are easy, and attaching touch events is also fairly simple; however, swiping does not come out of the box in mobile Safari, so we need to build that. So let's get started by first laying out our gallery and then styling it.

Gallery markup and styling

As with any slideshow gallery, it's essential that we create a good structure. This structure should be easy to follow and doesn't really require too many elements if we want to modularize it.

The basic gallery slide list

Let's start with something extremely basic. First, let's create a `div` with a class of `gallery`:

```
<div class="gallery"></div>
```

From here, we want a content area that will contain all our slides. You may very well be asking why we just don't dump our slides inside the parent gallery container, and the reason for this is so that we can extend our gallery with other functionality, such as a play and pause button, without compromising the structure of the slides themselves.

So let's create another `div` inside our gallery that contains a class of `gallery-content`, like so:

```
<div class="gallery">
    <div class="gallery-content">
    </div>
</div>
```

Now that we have a content area for our gallery, we want to make an unordered list of slides that contain our images. When we finally do this, our `gallery` markup should look like this:

```
<div class="gallery">
    <div class="gallery-content">
        <ul>
            <li>
                <img src="../assets/sample-image1.jpg" alt="…">
            </li>
            <li>
                <img src="../assets/sample-image2.jpg" alt="…">
            </li>
            <li>
                <img src="../assets/sample-image3.jpg" alt="…">
            </li>
            <li>
                <img src="../assets/sample-image4.jpg" alt="… ">
            </li>
        </ul>
    </div>
</div>
```

[When you look at the preceding markup, you may be in shock that I left out content for the `alt` attribute on the `image` tag. Yes, this is a bad practice, but I do this here to move a bit quicker. However, you shouldn't do this in your applications, always give your images an `alt` attribute with relative content.]

Now that we have a basic markup structure we should start styling this slideshow, but keep in mind that the preceding markup is not end all, be all. I've seen some extraordinary work on other sites, and that's cool, but we want to keep things simple here and give you a foundation to build upon. I encourage you to experiment and try new things out, but don't let the preceding markup be the final solution for you. Before we get to styling this, let's take a step back and understand why we have a content area.

Adding simple gallery controls

We don't want to complicate the styling for the content area. If we do, this can lead to some messy styling that "fixes our markup". For this reason, we created a content area and are now going to add a `controls` group to our slideshow.

So let's follow the same principle; let's create a `div` with a class of `gallery-controls` that contains two anchor tags, one for the next button and another for the previous button.

```
<div class="gallery-controls">
    <a href="#next">&raquo;</a>
    <a href="#previous">&laquo;</a>
</div>
```

Now, both the content area and the control are two separate areas that can be controlled independently. You'll see how this makes things easy for us when we start styling our gallery. For now, trust me, this will make it simpler for you to control your gallery. But now, on to our styling!

Making images responsive

We've gone over responsive design in the very first chapter of this book, and hopefully you do understand the principles. But if you don't, this chapter should give you a good idea of what we can do to make sure our application not only works on iPhone, but also on other touch devices as well.

So let's say we want our gallery to exist on mobile and desktop versions of our site, which is a highly desirable feature because now you are building a component that is reusable and device agnostic. But this also makes things difficult, not taking into account asset management, we need to calculate how big our images must be. Well, for this example, we want our images to scale to 100 percent of the slideshow's width, we want the slideshow to take up 100 percent of our screen width, and have 12 pixel padding on the sides.

In order to accomplish this, we can simply set the width of all images to 100 percent and have our gallery apply padding of 12 pixels on the sides, like so:

```
img {
  width: 100%;
}

.gallery {
  margin: 12px 0 0 0;
  padding: 0 12px;
}
```

Note that our gallery will already take 100 percent of our screen width, minus the padding we give it on the sides. For this reason you don't see a property of `width: 100%` in `.gallery`. Also, take into account that we added 12 pixels to the top of the gallery to give it some room from the main navigation. And last but not least, we are using shorthand properties here, so that we don't use padding-left, margin-top, and so on. This makes our code, not only shorter, but easier to understand and maintain.

This is all that's needed to make a responsive gallery using CSS, the rest of the styling will be applied via JavaScript. Some of you may balk at this, but it's a fairly used technique because we need to know the device's width in order to set our gallery correctly for responsive use. But before we get to that, let's start out by finishing up the styling for our gallery.

Styling our gallery

Now let's finish up the styling for our gallery in the CSS. Some of this will still apply for responsive applications, but the previous section helps define the principles. Don't worry though; I'll go over the styling for each part of this application so that you can understand it thoroughly.

First, let's make sure our gallery content scales to 100 percent in width, and because eventually our slides will float left, we want the parent container to have a height; so let's add a property of `overflow: hidden`. When you're done, your styles should look like this:

```
.gallery .gallery-content {
  width: 100%;
  overflow: hidden;
}
```

Next, we want to make sure that the unordered list also has a height for when the slides are floated left, so that this height gets applied to the gallery content. Not only that, but because we want to animate the unordered list left or right, based on user interaction, we need to make sure the position and starting `left` value are defined. When you're done applying this styling, it should look like this:

```
.gallery .gallery-content > ul {
  left: 0;
  margin: 0;
  overflow: hidden;
  padding: 0;
  position: relative;
}
```

Here we've also applied a `0` value to `margin` and `padding`. This is mostly a reset so that we don't run into any layout issues later on. `Normalize.css` applies some `padding` and `margin` to unordered lists by default, and this is good but not necessary for our application so we wipe out those values.

Now, let's focus on styling the controls of our slideshow. This next step is mostly a setup style, so that we don't run into any issues when we float elements within a container; such as what we did for the `gallery` content and unordered list previously. So let's make sure `overflow` is set to `hidden` for our controls:

```
.gallery .gallery-controls {
  overflow: hidden;
}
```

Since our controls are now set to `hidden` when elements overflow, we can float our next and previous buttons accordingly so that they are on the appropriate side of the slideshow.

```
.gallery .gallery-controls a[href="#next"] {
  float: right;
}

.gallery .gallery-controls a[href="#previous"] {
  float: left;
}
```

This is all that's needed to get the basic styling done for your slideshow. Unfortunately it still doesn't look pretty, and that's because we need to use JavaScript in order to determine the screen size, apply widths to our slides, and an overall width to our unordered list. However, there's one more thing we can do here that brings some serious performance optimization to our application, and that's using CSS3 transitions.

Before we move on, it's important to note that our CSS selectors are cascading from the `gallery div`. This is a good practice because it allows you to compartmentalize your styles. What we are doing is basically creating default styles for our gallery, and if anyone wanted to customize it, they could add their own class before `.gallery` to override these styles, allowing the gallery to be much more customizable. This is a basic CSS fundamental, but I thought I'd point it out in order to show the importance of creating styles that are modular.

Using CSS3 transitions

CSS3 transitions are extremely important to our applications. Not only because it makes things easier for us, but also because it gives us performance optimization. By default, mobile Safari uses hardware acceleration for CSS3 transitions; what this means is that the hardware will handle the rendering of these transitions, and as such we won't need to do it manually. Traditionally, we needed to do this using JavaScript, and because of that we gained no performance optimization, but now we do with CSS3 transitions. So let's use them!

This is a basic gallery, and we want to keep it simple. So let's just add our transition to the unordered list. After all, the unordered list is what we want to animate when the user swipes or initiates an action from the controls. To do this, we will use the `transition` property and use shorthand to define what property we want to animate, how long, and what transition-timing-function, otherwise known as easing method, to use.

```
.gallery .gallery-content > ul {
  left: 0;
  margin: 0;
  overflow: hidden;
  padding: 0;
  position: relative;

  -webkit-transition: left 500ms ease;
  -moz-transition: left 500ms ease;
  -ms-transition: left 500ms ease;
  -o-transition: left 500ms ease;
  transition: left 500ms ease;
}
```

The only thing we have done here is added the `transition` property to our unordered list. This property tells the unordered list to animate the `left` property within 500 milliseconds and to use the default easing method.

> Here we are defining five transition properties, each one being prefixed to the browser vendor while the last is the supported standard property. This is done so that our gallery is usable across devices. Yes, it's a bit complicated and messy, but it is a necessary evil given the fact that browser vendors have prefixed this property and only now have begun to use the non-prefixed version.

Gallery interactivity

The heart of our slideshow lies in its interactivity; from next and previous buttons, swipeable content and animation rich displays—our slideshow is dependent on JavaScript. In this section we dive deep into what makes our slideshow work; using our basic framework, we'll build a `Gallery` class that is efficient and achieves the goals stated previously. In reality, our gallery should just have functionality that allows it to resize and play in a certain direction. But, as always, this takes some setup work and then we hook everything up. So let's get started!

The basic template

First, we'll create our `Gallery` class. This class should be set up in the same way as any other class we've built. But in case you haven't been following the book in order, all we need to do is check for the `App` namespace and then create a `Gallery` class underneath it. Wrapped in a closure, we'll have some defaults and a `Gallery` function, and return it at the end of the closure declaration. As we have mentioned previously, we'll have the following:

```
var App = window.App || {};

App.Gallery = (function($) {

    var _defaults = {};

    function Gallery() {}

    return Gallery;

}(Zepto));
```

The only thing that is different here is that we are only passing in the `Zepto` object. Previously, we were passing in `window` and `document`, but for this class we won't need those two objects, so we limit it to the Zepto library.

That's all we need for now, but what's more critical is to cache the elements we will be re-using, plus they will need to be available in the closure so that they are available in private and public methods.

Caching the gallery

Caching objects is extremely helpful in our applications, especially since it increases the performance and makes our app extremely efficient. By cutting down on the number of lookups we need to do in the DOM, we allow for faster processing and create a less error-prone application.

Not only do we want to cache certain elements, but we want them to also be available in the closure so that they are accessible by all methods. To do this, all we need to do is add the cached variables after the `_defaults` variable located above our construct, like so:

```
var _defaults = {},
    $gallery,
    $slides,
    $slidesContainer,
    $slidesLength,
    $galleryControls,
    slidesWidth,
    galleryWidth;
```

In the preceding code, we can see that the gallery, its slides, the container of the slides, the number of slides, gallery controls, and slide and gallery width will be cached. However, at this point in time we haven't cached anything. So let's start assigning them the values they should have.

The best place to initialize your values would be in the constructor, or when you create an instance of a gallery. The constructor should go ahead and cache the values we need for the rest of the running application. On top of that, each variable semantically describes what it should be holding, making it easier for you to understand what's going on. Let's take a look at the following function:

```
function Gallery() {
    $gallery = this.$el = $('.gallery');

    $slides = $gallery.find('li');

    $slidesContainer = $gallery.find('.gallery-content > ul');

    $galleryControls = $gallery.find('.gallery-controls');

    $slidesLength = $slides.length;
}
```

From this function we can gather that we cache the gallery, and from that all other values are determined. For example, we use `$gallery` to find all the slides, or list items. This is extremely useful, because what we are doing is telling our application to begin with the `gallery` element and then dive into it to find the appropriate values. Otherwise, we would typically begin at the top of the document and then go down, which is extremely costly when it comes to DOM lookups.

This is a critical step in the process, because everything else should be easy as pie. So let's start hooking up some interactivity!

Connecting our controls

First, we want the user to be able to click on the next and previous buttons. However, we don't want anything to happen just yet, we just want to capture those events. As always let's start small and then work up, what we want to do is have a foundation to work with.

Attaching the events

We've previously gone over how to attach events, and in this chapter it's no different. So first create an `attachEvents` method that looks up the next and previous buttons from the gallery and then calls a `play` method. When you're done writing the code, you should have something like this:

```
function attachEvents() {
    $galleryControls
        on('click', 'a[href="#next"]', play).
        on('click', 'a[href="#previous"]', play);
}
```

Nothing is different here. We use the cached `$galleryControls` variable and tell it to listen to the `click` event coming from our next and previous buttons. When that `click` event comes from the designated element, then call our `play` method. If we run our code now, nothing would happen, except that we would probably get an error because `play` does not exist. But let's not do that; instead we'll call our `attachEvents` method in our constructor after all of the setup code takes place:

```
function Gallery() {
  // our previous code

    attachEvents();
}
```

Nothing crazy here, we're just calling `attachEvents`, a private method. Did you notice how we are using `$galleryControls` even though it's a private method? That's because that variable exists within the closure scope, so this makes it easier to manage variables without polluting the global scope of the program. If you don't yet understand what's going on here, don't worry. With time and practice this will make sense and things will just be that much easier.

Now, we still have a problem. There is no `play` method, so let's create it!

Handling our events

Because our `play` method does not exist, our application fails; so we need to write it. But what should it do? Well, for this application, we want it to determine the direction in which the gallery should play. Then we want it to animate left or right based on the current position in which the gallery is located. You might be saying, that sounds easier than you think. But in reality it is. So let's go step-by-step.

Caching variables, again

Yes, we want to cache as much as possible. Again this is a mobile application we are creating for iPhone and because of the nature of mobile, we need to optimize where we can. But what should we be caching? Well, the first thing we'll be checking is direction, and then manipulating a current left position of the unordered list. To prevent lookups of these values, let's just state a `currentLeftPos` and direction at the top of the method, like so:

```
function play(e) {
    var currentLeftPos, direction;
}
```

Simple! Now, let's determine these values. An easy way to determine direction is to have it based on the value of the element clicked. In this case, we can check for #next or #previous, the values of the `href` attributes. To make it simpler, we may want to remove the hash tag as well, just in case we ever wanted to expose this method and allow ourselves to pass in `next` or `previous`. So let's do this:

```
function play(e) {
    var currentLeftPos, direction;

    direction = $(this).attr('href');

    direction = direction.substr(1, direction.length);
}
```

Don't worry too much about the details here, but essentially since `play` is an event handler, `this` has become the target event, which would be our anchor tag. This is how we can get the `href` value from those elements. Also, don't be too nervous about the string manipulation going on there. Basically we are using `substr`, a built-in `string` method, and passing it `1` so that it starts at position one and then gets the rest of the string. This is how we are able to get the word next or previous from the `href` attribute.

Great, at this point we have determined the direction. Now we want to get the latest left position of the unordered list. To do this, we can add the following bit of code after we've set the direction:

```
function play(e) {

  // Previous code

    currentLeftPos = parseInt($slidesContainer.css('left'), 10);
}
```

Note that we are using `parseInt`, a built-in number method that accepts an integer as its first parameter and then the base as its second. We do this because when we request the value of the `left` property, we get something like `0px`, and we want the value that we are using to be an integer, not a string. So `parseInt` helps us out by taking `0px` and interpreting it as an integer of `0`.

Now it's time to create the magical part of our application. This part is a bit complex, but in the end will help us achieve the effect we are looking for. But let's first focus on getting our application to move on the next call to action. To do this, we want to set the left position of the unordered list to the current left position minus the width of a single slide. To do this, we can simply write the following code after the setting of `currentLeftPos`:

```
function play(e) {
    // Previous code

    // Next
    $slidesContainer.css({
    'left': currentLeftPos + -(slidesWidth) + 'px' });
}
```

The preceding code will do exactly as we ask it; however, there are a couple of issues we run into. First, this will always run, even if the previous button is hit. Two, there's no check for when you have reached the very end of your gallery. This can easily be added to our application like so:

```
function play(e) {
    // Previous code
```

```
        // Next
        if (direction === 'next') {
            if (Math.abs(currentLeftPos) < (galleryWidth - slidesWidth)) {
                $slidesContainer.css({
                    'left': currentLeftPos + -(slidesWidth) + 'px'
                });
            }
        }
    }
```

> You may have noticed that we are using `Math.abs` on the `currentLeftPos`. That is because we'll be getting a negative number as our value, and since we don't want to complicate the math or comparison, we simply turn it into a positive integer using `Math.abs`. Keep it simple!

In this adjusted code we check the direction, looking for `next`, and then checking to make sure the current left position is less than the gallery width minus a single slide's width. This helps prevent any errors that might come up.

Now on to implementing our `previous` functionality. For this step, we'll follow the same procedure; we'll make sure we want to go in the `previous` direction, then we'll do a comparison to make sure we don't go below the `0` mark, and finally we'll execute the code if the conditions have been met. When we're done implementing this functionality, we should have the following bit of code:

```
    function play(e) {
        // Previous code

        // Previous
        if (direction === 'previous') {
            if (Math.abs(currentLeftPos) > 0) {
                $slidesContainer.css({
                    'left':  currentLeftPos + slidesWidth + 'px'
                });
            }
        }
    }
```

The only difference here is that we are comparing against the static number `0`. This is to prevent any positive values that will cause a visual error in our gallery. Then instead of negating our numbers, we use the correct values in order to add to a negative number, thus giving the appearance of a `Previous` action.

In the end, our `play` method should look like this:

```
function play(e) {
    var currentLeftPos, direction;

    direction = $(this).attr('href');

    direction = direction.substr(1, direction.length);

    currentLeftPos = parseInt($slidesContainer.css('left'), 10);

    // Next
    if (direction === 'next') {
        if (Math.abs(currentLeftPos) < (galleryWidth - slidesWidth)) {
            $slidesContainer.css({
                'left': currentLeftPos + -(slidesWidth) + 'px'
            });
        }
    }

    // Previous
    if (direction === 'previous') {
        if (Math.abs(currentLeftPos) > 0) {
            $slidesContainer.css({
                'left':  currentLeftPos + slidesWidth + 'px'
            });
        }
    }
}
```

Are we done? Yes we are! Even though we are only switching the left position value of our unordered list. We are actually animating because, if you remember, we've told our element to transition the left property within our CSS. See how easy and effective using CSS3 properties is? With a simple declaration we have been able to minimize code and make a highly optimized version of our gallery.

Now, that we have the core of our gallery completed, let's make it responsive!

Gallery responsiveness

We're going to side-step here a bit, but it's worth the necessary effort! In this step, we're going to look into making our gallery responsive to our user's device's width. So let's get started by setting our styles.

Setting the gallery styles

Here we're going to set all the styles necessary to make our gallery responsive. There are a couple of things we need to do. First, let's create a public `setStyles` method using the `Gallery` function's `prototype`:

```
Gallery.prototype.setStyles = function() {

    return this;
};
```

The preceding method, as you may have noticed, returns the instance of `Gallery` and thus allows you to chain your methods. Next, get an individual slide's width. This width is 100 percent of the container it's in, so it should be of the same width as the gallery itself. To get this width we can do the following within `setStyles`:

```
Gallery.prototype.setStyles = function() {

    slidesWidth = $slides.width();

    return this;
};
```

Now, we can determine the full width of the gallery by multiplying the number of slides by the width each is set to, which we've already determined in the previous step. When we do this, we achieve the following code:

```
Gallery.prototype.setStyles = function() {

    slidesWidth = $slides.width();

    galleryWidth = slidesWidth * $slidesLength;

    return this;
};
```

This following step may be confusing, however it is critical because we need to manually set the width of each slide in order to float them next to each other. So what we need to do now is apply the `slideWidth` value to each slide, like so:

```
Gallery.prototype.setStyles = function() {

    slidesWidth = $slides.width();

    galleryWidth = slidesWidth * $slidesLength;

    $slides.width(slidesWidth);

    return this;
};
```

Now, we can also set the width of the slides container using the calculate gallery width. Again, we need to do this in order to keep a gallery that has slides which are floated to the left. So we'll set the slide container's width and then float all our slides to the left. Your `setStyles` method will look like this, when we code these requirements:

```
Gallery.prototype.setStyles = function() {

    slidesWidth = $slides.width();

    galleryWidth = slidesWidth * $slidesLength;

    $slides.width(slidesWidth);

    $slidesContainer.css({'width': galleryWidth});

    $slides.css({'float': 'left'});

    return this;
};
```

This is all it takes in order to set our gallery's styles in a responsive manner. However, there's one issue here; the styles cannot reset, which is needed in order to determine the widths of the slides and containers appropriately when the device's orientation or width changes in some way. Let's do some setup work in order to get this reset going.

To do this, we'll simply wrap our functionality within a method that we then pass to a public `resetStyles` method. In this technique, we are essentially sending in a `callback` that will get executed when `resetStyles` functionality has finished taking place. For now, your code should result in the following:

```
Gallery.prototype.setStyles = function() {

    this.resetStyles(function(){
        slidesWidth = $slides.width();

        galleryWidth = slidesWidth * $slidesLength;

        $slides.width(slidesWidth);

        $slidesContainer.css({'width': galleryWidth});

        $slides.css({'float': 'left'});
    });

    return this;
};
```

As you can see, all the functionality we originally created for `setStyles` has been wrapped in an anonymous function, also known as a `callback`, that will get called when `resetStyles` is done running. To get the full picture, let's continue on by creating our `resetStyles` function.

Resetting the gallery styles

Resetting an element's style is actually not that complicated, so we'll go head first into this method. Check out the following code that should be within your `reset` method.

```
Gallery.prototype.resetStyles = function(callback) {
    $slides.attr('style', null);

    $slidesContainer.attr('style', null);

    $slides.attr('style', null);

    if (typeof callback !== 'undefined') {
        callback.call(this);
    }

    return this;
};
```

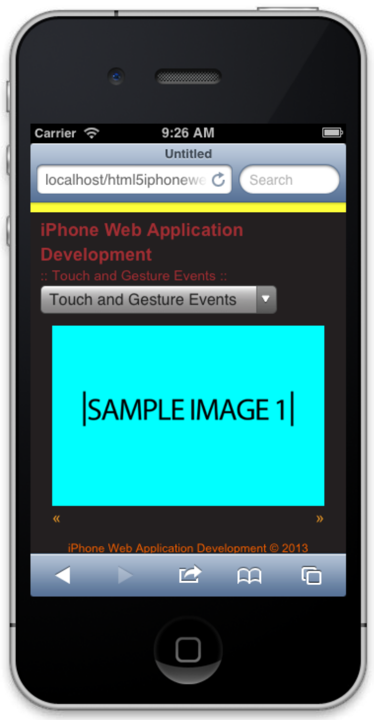
Not too crazy right? We are basically just removing the inline styles that Zepto applies when we set an element's style using JavaScript, or what we've done in our `setStyles` method. When we remove these styles, we then check for a `callback` method and execute that method. This is a good practice because, let's say we need to reset the styles of our gallery for any other reason; we don't want to create unnecessary functions for no reason.

Initializing the gallery styles

The last thing we need to do is initialize our styles. To do this, let's call `setStyles` when our code initializes in the `Gallery` constructor.

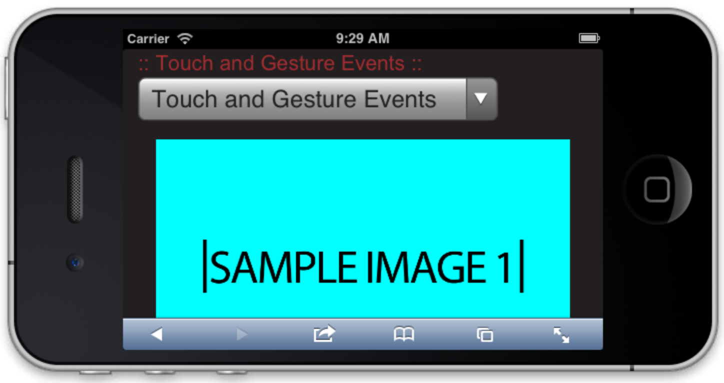
```
function Gallery() {
  // our previous code

   this.setStyles();
    attachEvents();
}
```

When we finally have our styles in place, our application should look like the following in the portrait mode:


Carrier
9:26 AM
Untitled
localhost/html5iphonewe
Search
iPhone Web Application Development
:: Touch and Gesture Events ::
Touch and Gesture Events
|SAMPLE IMAGE 1|
«
»
iPhone Web Application Development © 2013

Responsive gallery

In landscape mode, our application should like the following:



Responsive gallery

Unfortunately, your application won't look or behave like the one shown in these screenshots; this is because nothing is hooked up right now and we haven't even gone through initializing any of our code. But if you do want to get to that immediately and see how we do it, you can check out the last section in this chapter, right before our conclusion. If you follow those steps, you should have an application that will look similar to what we have just seen.

Technically, our gallery is now completely built and we can now fully interact with it using our next and previous buttons. But now, it's on to the fun we've all been waiting for, touch events!

Extending the gallery for touch

We could, by default, include touch interactivity into the `Gallery` class, but this wouldn't be reusable and couldn't be applied to other parts of the application. So in this section we'll create a new class called `Swipe`, and it will contain everything that's needed to detect swiping gestures on a specific module.

The basic template

As always we want to start out with our basic framework, similar to the other classes we've written before. To get started we can write the following basic template:

```
var App = window.App || {};

App.Swipe = (function(window, document, $){

  var _defaults = {};

  function Swipe(options) {
    this.options = $.extend({}, _defaults, options);
  }

    return Swipe;

}(window, document, Zepto));
```

The `Swipe` class is a bit different from our `Gallery` class in that it accepts the `window`, `document`, and `Zepto` objects. Another difference is that the `Swipe` constructor accepts one parameter called `options`, used to override default values that we will set soon.

Default options and modularizing swipe events

There are a couple things we want to do inside the `Swipe` class. First, we want to make sure it only applies to the particular container and not the entire document. Then, we want to be able to cache certain values like the initial x position of the touch and the end x position. These cached values should also be available in the closure scope, so that they are available across all methods.

Here are the defaults we would like to have and the cached values that will be available in the closure's scope:

```
var _defaults = {
  'el': document.body,
  '$el': $(document.body)
},
el,
$el,
delta,
initXPos,
endXPos,
threshold = 30;
```

What we're basically saying in the preceding code is that the default element, the swipe functionality, should be attached to is the document's `body` element. From here we make sure that we can access these cached elements in the closure's scope. Finally we set up some variables that will store the information about the touch gesture we will be listening for.

Now in our constructor we want to override these defaults and make sure some of these initial values will exist in the global scope:

```
function Swipe(options) {
  this.options = $.extend({}, _defaults, options);
  $el = this.$el = this.options.$el = $(this.options.el);
  threshold = this.options.threshold || threshold;

  this.init();
}
```

Here we are using Zepto's `extend` method to create a new object that contains a merge of the options parameter into the defaults object. We then make sure that the closure's scope contains the cached element the swipe class will be attached to. Finally we check if a custom threshold was passed in and override the default 30. After all this, we call an initialize method at the end of the constructor so that the Swipe class starts automatically.

Listening to touch events

Now we need to attach the appropriate events to the `Swipe` class. These events will be based on the touch events we covered earlier, but they will be used in such a way that mimics swipe gestures. To achieve this, we first need to listen to the `touchstart`, `touchend`, and `touchmove` events and assign event handlers to each of these. We can do all of this inside the `init` method that we are calling from the constructor.

So first let's create our `init` method on the Swipe's `prototype`, and let's make sure we are returning the instance at the end of the method:

```
Swipe.prototype.init = function() {

  return this;
};
```

Inside this method we want to listen to the touch events mentioned previously and make sure they have event handlers. To do this, we'll use Zepto's `on` method and attach the events to the element we've cached:

```
Swipe.prototype.init = function() {
  this.options.$el.
    on('touchstart', handleTouchStart).
    on('touchend', handleTouchEnd).
    on('touchmove', handleTouchMove);

  return this;
};
```

In the preceding code we pass in the event as a string to the `on` method's first parameter, and then assign an event handler, which we have not yet created. What you'll also notice is that these methods are chainable, allowing us to attach several events all at once. This is why we return `this` at the end of our public methods, so that we can allow ourselves to make calls synchronously.

Handling touch events

Now we need to create the event handlers we've assigned to each listener. We'll go through one handler at a time so that we can explain how the swipe gesture is created from these touch events. The first we want to look at is the `touchstart` handler.

When we place our finger on the phone, the first thing we want to do is store the initial x position of the finger. To access this information, there is a `touches` array on the event triggered. Because we only want to use the first touch, we need to access the first touch in the `touches` array. Once we get the first touch, we can get the x position by using the `pageX` attribute on that object. This is what the functionality for `handleTouchStart` will look like:

```
function handleTouchStart(e) {
    initXPos = e.touches[0].pageX;
}
```

As you can see the `handleTouchStart` method accepts one parameter, the event object. We then set `initXPos` to the `pageX` attribute on the first touch in the `touches` array on the event object. That might sound like a huge mess, but basically we are just accessing the objects we need to in order to hold the initial x value of your touch.

Next, we want to create the `handleTouchMove` event handler. This handler will contain the same concepts as `handleTouchStart`, but instead of the initial x position, we want to update the ending x position. This can be seen in the following code:

```
function handleTouchMove(e) {
  e.preventDefault();
    endXPos = e.changedTouches[0].pageX;
}
```

There are a couple of differences here that I'll explain. First, we prevent the default behavior of a touch move. This is to stop any weird behavior from happening and is usually suggested when we want to create a unique experience, such as a swipeable gallery.

Another difference you'll notice is that we are looking into the `changedTouches` object on the event. This is because the `move` event does not contain a `touches` object. Although a bit contentious, this helps keep track of each touch and the changed attributes of that specific touch. So if I had multiple touches, then my `changedTouches` object would contain each changed touch appropriately.

Up until now, all we have done is set the initial and ending x position. Now we need to use those values to create a `delta` value that is then used to trigger a swipe in the left or right direction. This is what our `handleTouchEnd` event handler will be doing for us.

Here's the code that `handleTouchEnd` should contain:

```
function handleTouchEnd(e) {
    endXPos = e.changedTouches[0].pageX;
    delta = endXPos - initXPos;

    if(delta > threshold) {
        $el.trigger('SwipeLeft');
    }

    // The *-1 converts the threshold to a negative integer
    if(delta < threshold*-1) {
        $el.trigger('SwipeRight');
    }
}
```

Now let's go over this code, one line at a time. First we do the exact same thing that `handleTouchMove` does, and that's setting the end x position. Next, we set our delta value, which is the difference calculated by subtracting the end x position from the initial x position. Now we do a comparison; if the delta is larger than the threshold, then trigger a custom event called `SwipeLeft`. Our next comparison is a bit more confusing but basically we check to see if the `delta` value is less than a negative threshold. This is so that we can detect a swipe in the right direction.

Our `Swipe` class is now complete. We have created the necessary functionality that listens to our touch events and then mimics a gesture that we can tie into to. But we haven't actually connected it to our gallery, which is the last step in the process. Feel proud that you've reached this point, because now the easy stuff happens!

Putting it all together

Okay, so at this point we have a gallery and the ability to detect swipe gestures using touch events. But right now, nothing is really connected, and in fact we haven't initialized our `Gallery` class so nothing should be working right now. But this is what this last section is about; we'll go through initializing our `Gallery` class, adding the `Swipe` functionality and then reacting to our swipe events.

The JavaScript

The first thing we want to do is open our `App.Touch.js` file, as you recall this file pertains to the functionality of our touch page and thus is the file that will contain all our initialization. When we have this file open, go to the `init` method, or if it's not created already, then create it and initialize an instance of `Gallery`:

```
Touch.prototype.init = function() {
  var that = this;

  // Initializing Gallery
  this.gallery = new App.Gallery();

  return this;
};
```

Now that we have initialized our `Gallery` class, the gallery should instantly initialize. But keep in mind we have not modified our markup to include this file. So even at this point, you won't see the fruits of your labor. But let's make sure we continue the setup work. In this next step, we want to initialize our `Swipe` class and make sure it sets itself to the `gallery` element:

```
Touch.prototype.init = function() {
  // Previous code

  // Initializing Swipe
  this.swipe = new App.Swipe({
    'el': document.querySelector('.gallery')
  });

  return this;
};
```

Now, even at this point our gallery does not respond to swipe events. That is because our swipe functionality only detects touches and dispatches those custom events we set up previously, so what we need to do is listen for those events on the gallery and then tell it to play the next or previous slide:

```
Touch.prototype.init = function() {
  // Previous code

  // Listen to the swipe and then trigger the appropriate click
  this.swipe.$el.
```

```
      on('SwipeLeft', function(){
        that.gallery.$el.find('a[href="#previous"]')
.trigger('click');
      }).
      on('SwipeRight', function(){
        that.gallery.$el.find('a[href="#next"]').trigger('click');
      });

    return this;
  };
```

In the preceding code, we listen for the `SwipeLeft` and `SwipeRight` events that get dispatched by our instance of swipe. When either event is dispatched, based on the event, we simulate a click on either the previous or next button. In this way we are able to give the appearance that the user is swiping throughout our gallery, while at the same time eliminating any kind of complications.

When you are done writing your `init` method, it should look like this:

```
  Touch.prototype.init = function() {
    var that = this;

    // Initializing Gallery
    this.gallery = new App.Gallery();

    // Initializing Swipe
    this.swipe = new App.Swipe({
      'el': document.querySelector('.gallery')
    });

    // Listen to the swipe and then trigger the appropriate click
    this.swipe.$el.
      on('SwipeLeft', function(){
        that.gallery.$el.find('a[href="#previous"]')
.trigger('click');
      }).
      on('SwipeRight', function(){
        that.gallery.$el.find('a[href="#next"]').trigger('click');
      });

    return this;
  };
```

The markup

The final item that needs to be taken care of is the markup on the page—the scripts being included. To make things simpler and get you to finally run your application correctly, here's what you need to include on your page:

```
<script src="../js/vendor/zepto.min.js"></script>
<script src="../js/helper.js"></script>
<!-- BEGIN: Our Framework -->
<script src="../js/App/App.js"></script>
<script src="../js/App/App.Nav.js"></script>
<script src="../js/App/App.Gallery.js"></script>
<script src="../js/App/App.Swipe.js"></script>
<script src="../js/App/App.Touch.js"></script>
<!-- END: Our Framework -->
<script src="../js/main.js"></script>
<script> touch = new App.Touch(); </script>
```

The difference here, compared to other pages, is that we are only including the items we need, including `App.Nav.js`, `App.Gallery.js`, `App.Swipe.js`, and `App.Touch.js`. In comparison to other pages, we were including the entire framework, but we don't really need to do that for this page or any pages moving forward. One thing to note is that we have also created a global touch object that gets set to an instance of our `App.Touch` class. This is so that we can reference it in the debugger easily, but this should be replaced with `App.touch`, so that it is not polluting the global namespace.

We've reached the end! At this point you should have a fully functional gallery that has swipeable interactivity. Now pat yourself on the back; it's been a long journey, but I hope you can appreciate the fact that we have created reusable, modular code that is completely self-contained. On top of that, our gallery is completely responsive and adapts to users' devices, allowing them to enjoy the experience consistently.

Summary

In this chapter we have restyled our main navigation, gone over the fundamentals of both touch and gesture events, and then implemented both types of events using a responsive photo gallery that will adapt to the user's device. We have also gone through attaching these events and handling them appropriately for our requirements in the slideshow. From here on out you should have a good understanding about how to use touch events to create unique experiences on the iPhone, as well as on other mobile devices. Next, let's take a look at some special interactions that come with handling forms in HTML5 on the iPhone.

5
Understanding HTML5 Forms

In this chapter we take a look at forms using the latest HTML5 technology, including new input types and form attributes. We'll briefly review some of the new input types that we'll be using in our sample forms. From there we'll discuss some of the new attributes in the specification, while also looking at the `autocapitalize` attribute specifically for mobile devices. Before we dive into our sample forms, we consider the layout of forms on iOS devices and the limitations that come up when interacting with these forms. Finally, we create some sample forms, develop some simple validations, and then style our form specifically for iOS - and WebKit-supported browsers.

Once we've reviewed all of these features and have gone through our sample forms, we should have a solid grasp on HTML5 forms and how they relate to developing a web application for iOS.

Here are the topics that we will cover in this chapter:

- New HTML5 input types
- New HTML5 form-specific attributes
- Form layout for iPhone
- Form validation
- Form styling for iOS

So, let's start by going over the new standard HTML5 input types.

HTML5 input types

HTML5 introduces several new input types that speed up the development of our applications. In total there are 13 new input types introduced with the HTML5 specification, including `datetime`, `datetime-local`, `date`, `month`, `time`, `week`, `number`, `range`, `email`, `url`, `search`, `tel`, and `color`. Unfortunately, only 10 of these new inputs are supported on iOS, but there's no need to worry since the type defaults to text automatically. This doesn't help us too much, but it does allow us to create polyfills for the types we need but aren't supported. However, either way, following is a breakdown of all the input types supported on iOS and a description of what each does:

Input type	Description
`button`	Represents a button with no additional semantics.
`checkbox`	Represents a state or option that can be toggled.
`date`	Represents a control for setting the element's value to a string representing a date.
`datetime`	Represents a control for setting the element's value to a string representing a global date and time (with time zone information).
`datetime-local`	Represents a control for setting the element's value to a string representing a local date and time (with no time zone information).
`email`	Represents a control for editing a list of e-mail addresses.
`file`	Represents a list of file items, each consisting of a filename, a file type, and a file body (the contents of the file).
`hidden`	Represents a value that is not intended to be examined or manipulated by the user.
`image`	Represents either an image from which the UA enables a user to interactively select a pair of coordinates and submit the form, or alternatively a button from which the user can submit the form.
`month`	Represents a control for setting the element's value to a string representing a month.
`number`	Represents a precise control for setting the element's value to a string representing a number.
`password`	Represents a one-line plain-text edit control for entering a password.
`radio`	Represents a selection of one item from a list of items (a radio button).
`range`	Represents an imprecise control for setting the element's value to a string representing a number.
`reset`	Represents a button for resetting a form.
`search`	Represents a one-line plain-text edit control for entering one or more search terms.

Input type	Description
`submit`	Represents a button for submitting a form.
`tel`	Represents a one-line plain-text edit control for entering a telephone number.
`text`	Represents a one-line plain text edit control for the input element's value.
`time`	Represents a control for setting the element's value to a string representing a time (with no time zone information).
`url`	Represents a control for editing an absolute URL given in the element's value.
`week`	Represents a control for setting the element's value to a string representing a week.

These details are available at:

- `http://www.w3.org/TR/html-markup/input.html`
- `https://developer.apple.com/library/safari/#documentation/AppleApplications/Reference/SafariHTMLRef/Articles/InputTypes.html#//apple_ref/doc/uid/TP40008055-SW1`

Even though there are plenty of inputs we can experiment with here, we will only be focusing on the new `email`, `number`, `datetime`, and `range` types. The sample forms in this book will also contain the regular types, including `text`, `password`, and `submit`.

Now that we have a good grasp of what is supported and have a reference of information for what types might fit our needs, let's go ahead and review the attributes we can also take advantage of.

HTML5 attributes for forms

There are many attributes we can use in HTML5, but to keep this part simple we'll focus on the new attributes we can use on inputs and forms alike. The following attributes are defined in the latest HTML5 specification, except for `autocapitalize`, and are also supported on iOS:

Input attributes	Description
`autocapitalize`	Specifies the auto-capitalization behavior of text elements.
`autocomplete`	Specifies whether the element represents an input control for which a UA is meant to store the value entered by the user (so that the UA can prefill the form later).

Input attributes	Description
`min`	The expected lower bound for the element's value.
`max`	The expected upper bound for the element's value.
`multiple`	Specifies that the element allows multiple values.
`placeholder`	A short hint (one word or a short phrase) intended to aid the user when entering data into the control represented by its element.
`required`	Specifies that the element is a required part of form submission.

You can find details on these attributes at:

- `http://www.w3.org/TR/html-markup/global-attributes.html#global-attributes`
- `https://developer.apple.com/library/safari/#documentation/AppleApplications/Reference/SafariHTMLRef/Articles/Attributes.html#//apple_ref/doc/uid/TP40008058-SW2`
- `http://www.w3.org/TR/html-markup/form.html#form.attrs.autocomplete`

Not all form attributes are listed in the preceding table; only the latest supported attributes that are defined in the HTML5 specification are listed. This is to give us a good idea of the latest and greatest. If, however, you would like to gain a broader sense of what's supported, I encourage you to review the preceding sources that detail out this information and provide a thorough explanation of each attribute in the specification.

We now have a basic understanding of the latest attributes supported on iOS. We can now briefly review some design considerations and then jump straight into some sample HTML5 forms to see how the latest input types and attributes work together to simplify our development process.

Form layout for iPhone

In this section, we briefly cover some design considerations when we are creating a form for iOS. You may or may not have full control of the design of your form; however, to make it simpler to understand the limitations that may come up, the following table helps demonstrate the limited amount of screen real estate we have when working with forms. Hopefully, this will help you explain these limitations so that adjustments could be made. Let's take a look at the following table:

UI control	Pixel dimensions
Status bar	20 in Height
URL text field	60 in Height
Form assistant	44 in Height
Keyboard	216 in Portrait Height
	162 in Landscape Height
Button bar	44 in Portrait Height
	32 in Landscape Height

The details regarding these controls can be found at `https://developer.apple.com/library/safari/#documentation/AppleApplications/Reference/SafariWebContent/DesigningForms/DesigningForms.html`.

Based on these values, we need to adjust our forms for certain dimensions when these controls appear. For example, if all of these controls appear, except for the button bar, and we have an available height of 480 pixels, then our screen real estate ends up being a whopping height of 140 pixels.

As you can see it's a challenge to create usable forms for iOS, but not impossible. There are some interesting techniques that we can use to accommodate forms within our applications. But the best technique is simplicity. Make sure that you don't require your user to provide lots of information at once; so instead of requiring a name, e-mail, password, and password confirmation with a date of birth, you just require a username, password, and email address. Keeping it simple goes a long way in our applications and helps improve the user experience.

We now have a fair understanding of the limitations that come up when designing forms for iOS, but now let's jump into functionality and see how we can create some simple forms.

Sample HTML5 forms

Now we're going to take a close look at some code, including the markup, scripts, and styles. Some of this you may already know and for the most part the only emphasis here will be on the new HTML5 inputs and attributes. We'll look at how they get implemented into a form, what their effect is on the UI controls, and how to leverage this new technology into our scripts. But first, let's do some setup work so that everything is consistent across our pages.

Setup work

The first thing we need to do is open up the `index.html` file for our forms page. Once we have this open, you'll see that we have the old template that we initially created at the beginning of this book. As our applications have evolved, we must update this template to reflect those changes, so let's do the following tasks:

- Include the forms styling (`forms.css`) after our main styles
- Update the navigation to reflect our new menu
- Include our navigation script (`App.Nav.js`) and our forms script (`App.Forms.js`)

Including our forms styling

Currently, we do not have any styling for this page, but we should include our page-specific stylesheet. When we do this, our head should look like this:

```
<!DOCTYPE html>
<html class="no-js">
<head>
    [PREVIOUS META TAGS]

    <link rel="stylesheet" href="../css/normalize.css">
    <link rel="stylesheet" href="../css/main.css">
    <link rel="stylesheet" href="../css/forms.css">
    <script src="../js/vendor/modernizr-2.6.1.min.js"></script>
</head>
```

Updating the navigation

Again, as with the previous chapter, we need to update our navigation to reflect the new select menu. This helps us save screen real estate for our application. When we update our navigation, our markup will be updated to the following code:

```
<nav>
    <select>
        <option value="../index.html">Application Architecture</
option>
        <option value="../video/index.html">HTML5 Video</option>
        <option value="../audio/index.html">HTML5 Audio</option>
        <option value="../touch/index.html">Touch and Gesture Events</
option>
        <option value="../forms/index.html" selected>HTML5 Forms</
option>
        <option value="../location/index.html">Location Aware
Applications</option>
        <option value="../singlepage/index.html">Single Page
Applications</option>
    </select>
</nav>
```

Including our navigation and forms scripts

Now that we have our navigation in place, let's include the navigation script and while we're at it, let's include the page-specific script for our forms:

```
<script src="../js/vendor/zepto.min.js"></script>
<script src="../js/helper.js"></script>
<!-- BEGIN: Our Framework -->
<script src="../js/App/App.js"></script>
<script src="../js/App/App.Nav.js"></script>
<script src="../js/App/App.Forms.js"></script>
<!-- END: Our Framework -->
<script src="../js/main.js"></script>
```

As you can see we are only including the necessary scripts for this page to function.

The forms

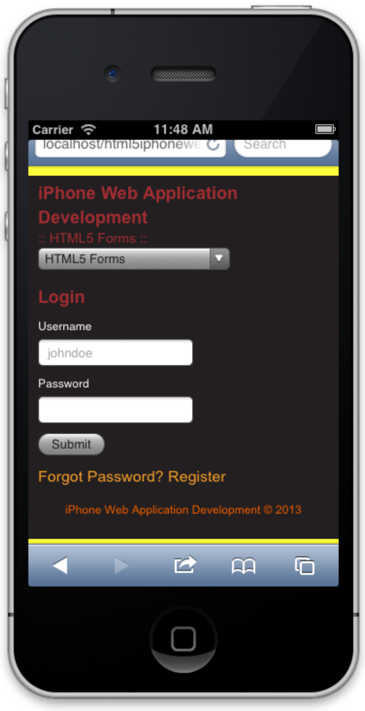
We will be developing three different forms on the page, including a login, register, and profile form. They're pretty basic and will mostly demonstrate the implementation of forms. After each bit of code, we'll review the new inputs and give some background information on how they affect our markup and user interface. For this part, don't concern yourself with the overall structure; meaning don't worry about the containing `div` of the form or the section with the header. The structure won't be discussed and is mostly there as a guideline for you. So, let's start with our login form.

The login form

The following is the structure for our **Login** form. Review this carefully, mostly focus on the `form` element and how it utilizes the `autocapitalize` attribute, and then look at how the required attribute is implemented across our username and password fields:

```
<!-- BEGIN: LOGIN CONTAINER -->
<form autocorrect="off" autocapitalize="off">
    <div class="error-messaging"></div>
    <label for="login-username">Username</label>
    <input name="username" id="login-username" type="text"
placeholder="johndoe" required>
    <label for="login-password">Password</label>
    <input name="password" id="login-password" type="password"
required>
    <input type="submit" value="Submit">
</form>
<!-- END: LOGIN CONTAINER -->
```

When we look at the final product, which is not at this point since we haven't styled our form, it should look somewhat like this:



Our login form

As you can see, we have `autocapitalize` set to off on the `form` element. This basically tells mobile Safari to not capitalize any of the inputs within it. We could easily set this to `off` on each individual input, but for the simplicity of this demonstration we've kept it on the `form` element.

Another cool thing that's going on here is that we've set `required` on both username and password. This is great because it won't submit the form unless these fields are filled out. In the old days, we would need to set a class of `required` and then check it with JavaScript; now we don't with the advent of HTML5.

> I know some of you may be shocked, but you won't receive any notice that a field is required in iOS. According to the developer documentation it's not supported. So why mention it here? Well because if we truly wanted to support multiple mobile devices, it's still a good idea to include this attribute so that our application is device-friendly, and if Apple chooses to support it in the future, we are future-proofing our application. Again, this has to be weighed by you and possibly by your team, but having this attribute conforms to the HTML 5 specification—it's just not supported on iOS, yet.

We can also see that the `placeholder` attribute is being used to apply some default text to our text inputs. Keep in mind that `placeholder` is exactly that: a placeholder. It is not setting the value of our input, so the value is still empty.

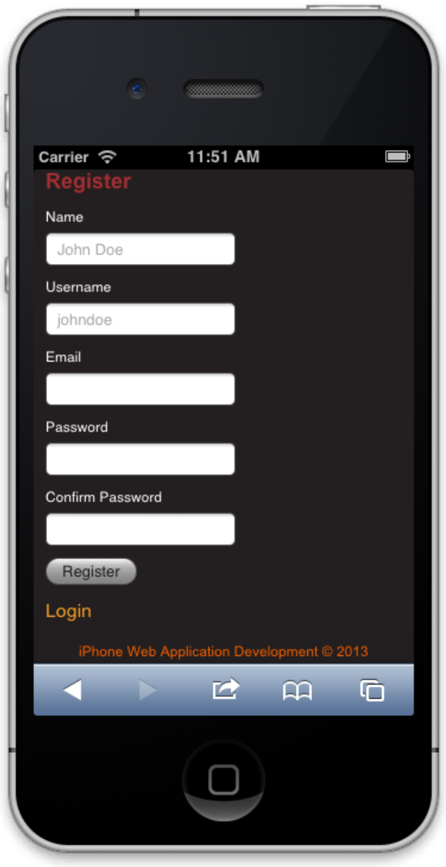
The registration form

Now we move on to our registration form. In this form we'll collect the user's name, username, e-mail, password, and confirmation password. Again, don't focus on structure. Concentrate on how the `autocorrect` attribute is implemented on the `form` element and then the use of the `email` input type.

```
<!-- BEGIN: REGISTER CONTAINER -->
<form autocorrect="off" autocapitalize="off">
    <div class="error-messaging"></div>
    <div class="field">
        <label for="register-name">Name</label>
        <input name="name" id="register-name" type="text"
placeholder="John Doe">
    </div>
    <div class="field">
        <label for="register-username">Username</label>
        <input class="required" name="username" id="register-username"
type="text" placeholder="johndoe">
    </div>
    <div class="field">
        <label for="profile-email">Email</label>
        <input class="required" type="email" id="profile-email"
autocorrect="off">
    </div>
    <div class="field">
        <label for="register-password">Password</label>
```

```
            <input class="required" named="password" id="register-
    password" type="password">
        </div>
        <div class="field">
            <label for="register-password-confirm">Confirm Password</
    label>
            <input class="required" named="password" id="register-
    password-confirm" type="password">
        </div>
        <input type="submit" value="Register">
    </form>
    <!-- BEGIN: REGISTER CONTAINER -->
```

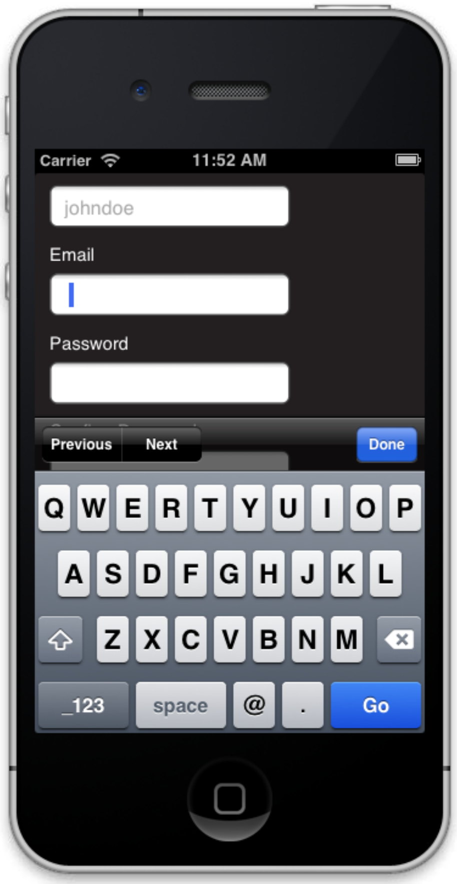
When we have completed this section and some of the preliminary styles, our form will look like this:



Our registration form

In this form, we've turned off `autocorrect` from all form fields. Again, we can do this individually with every element, but to make things simpler we've chosen to add it to the `form` element.

The last point to take into account is the use of the input type, `email`. When we start using some of the customized input types, our user interface accommodates. For example, when we click on the `email` input type, we'll see that the controls change to include the `@` symbol:



The email input type

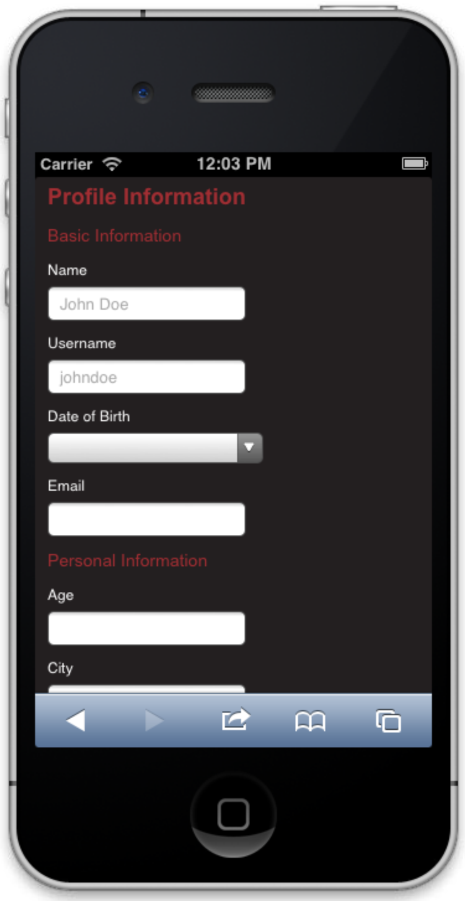
Now, let's take a closer look at some other input types to see how this affects our user interface.

The profile form

The following form is a sort of combination of both the login and registration forms with some extra fields. However, there are a couple of differences, so let's focus on just what's changed. In this example, we'll see that we've changed `autocapitalize` to `sentences` and have set `autocorrect` to `off` on only the fields we want it to apply to. On top of that, we start using the `datetime`, `number`, and `range` input types. The last change we've made is to apply the `required` fields using a class instead of an attribute—this will be explained further in the implementation of our scripts. For now, review the markup and then read on to view the explanations.

```
<!-- BEGIN: PROFILE UPDATES -->
<form autocapitalize="sentences">
    <div class="error-messaging"></div>
    <h2>Basic Information</h2>
    <div class="field">
        <label for="profile-name">Name</label>
        <input name="name" id="profile-name" type="text"
placeholder="John Doe">
    </div>
    <div class="field">
        <label for="profile-username">Username</label>
        <input name="username" id="profile-username" type="text"
placeholder="johndoe" autocorrect="off">
    </div>
    <div class="field">
        <label for="profile-dob">Date of Birth</label>
        <input type="datetime" id="profile-dob">
    </div>
    <div class="field">
        <label for="profile-email">Email</label>
        <input type="email" id="profile-email" autocorrect="off">
    </div>
    <h2>Personal Information</h2>
    <div class="field">
        <label for="profile-age">Age</label>
        <input type="number" id="profile-age">
    </div>
    <div class="field">
        <label for="profile-city">City</label>
        <input type="text" id="profile-city" placeholder="Boston">
    </div>
    <div class="field">
        <label for="profile-state">State</label>
        <select name="state" id="profile-state">
            <!-- OPTIONS GO HERE -->
        </select>
    </div>
    <div class="field">
        <label for="profile-zip">ZipCode</label>
        <input type="number" min="0" id="profile-zip">
    </div>
    <h2>Professional Information</h2>
    <div class="field">
        <label for="profile-skills-markup">HTML5</label>
        <input type="range" min="0" max="5" id="profile-skills-
markup">
    </div>
    <div class="field">
        <label for="profile-skills-styles">CSS3</label>
        <input type="range" min="0" max="5" id="profile-skills-
styles">
    </div>
    <div class="field">
        <label for="profile-skills-scripts">JavaScript</label>
        <input type="range" min="0" max="5" id="profile-skills-
scripts">
    </div>
    <h2>Bio Information</h2>
    <label for="profile-bio">About Yourself</label>
    <textarea id="profile-bio" name="about"></textarea>
    <div class="field">
        <label for="register-password">Password</label>
        <input class="required" named="password" id="register-
password" type="password">
    </div>
    <p>Provide your password to confirm.</p>
    <input type="submit" value="Update Profile">
</form>
```

Our final product will look like this after our styling:

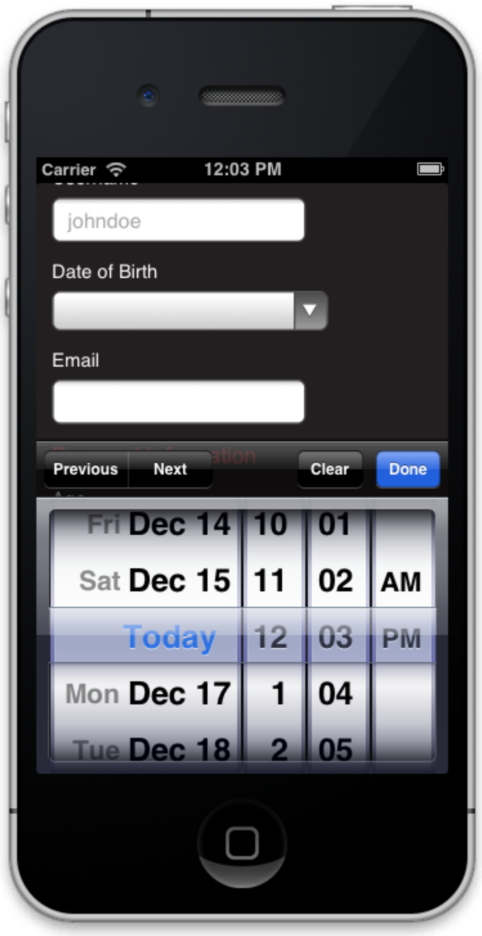


Our profile form

In this example, we've set `autocapitalize` to `sentences` on the `form` element. This helps us out because now we've defined exactly what we want to be capitalized, and that is sentences only. This is described in Apple's documentation and can be further explored there. As for `autocorrect`, we've set it on the individual items because we may want it to correct on `textarea`. Again, we could have chosen to set `autocorrect` to `off` on the `form` element and then set it to `on` within the `textarea`, but this is a matter of choice and is completely up to you as the developer. Now let's review the several input types.

The datetime type

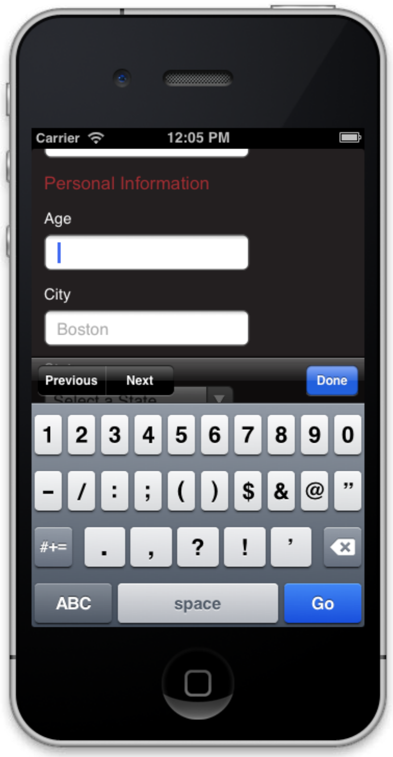
In this example we use `datetime` for the **Date of Birth** field. This is great because our UI accommodates exactly how we expect it to in order to provide exact information:



The datetime input type

The number type

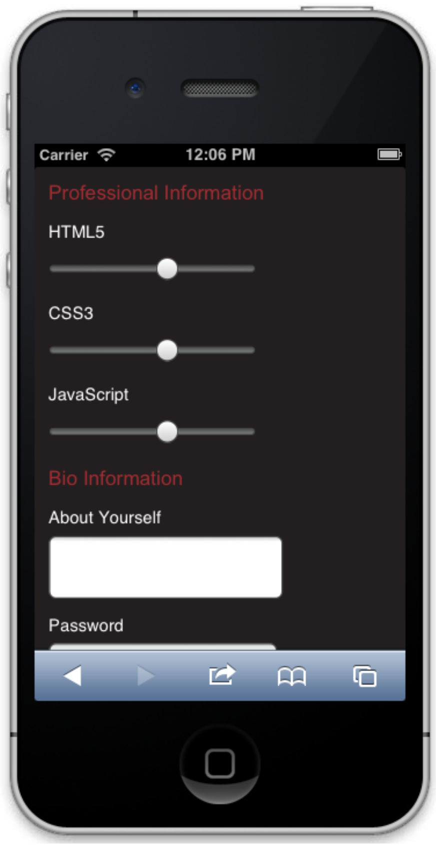
The `number` input type also manipulates our UI, so that we have a selection of numbers as defaults in our controls:



Our number input type

The range type

The `range` input type is an extremely useful control within our forms. Again, this type provides a custom UI that allows us to use system defaults, instead of JavaScript, to provide the type of value we're looking for:



The range input type

Now we've finished reviewing some of the new input fields and attributes in HTML5 and how they affect our iOS web application's UI. Next is using JavaScript to validate our form. Again, this will be very basic and will introduce us to how we can set up a reusable component for forms and won't directly tie into these new inputs and attributes. This is because these custom inputs and attributes were part of the specification to help speed up development, thus your need to use scripts for validation should be limited. Either way, let's move forward and take a quick look into our scripts.

Form validation

In this section, we review the JavaScript written for this page. There's nothing that's really new or that pushes the boundary; it is explicitly meant to demonstrate how to use the framework we've developed in this book to create self-contained code that validates multiple forms and makes it easier for you to extend. So let's get started by reviewing the basic template.

The basic template

The following is the basic template we've been using. A standard namespacing technique that extends the `App` namespace with a `Form` class will contain all of our functionality.

```
var App = window.App || {};

App.Form = (function(window, document, $){
    'use strict';

    var _defaults = {
            'element': 'form',
            'name': 'Form'
        };

    function Form(options) {
        // Customizes the options by merging them with whatever is
passed in
        this.options = $.extend({}, _defaults, options);

        this.init();
    }

    //----------------------------------------------------
    //  Private Methods
    //----------------------------------------------------

    //----------------------------------------------------
```

```
        //----------------------------------------------------
        //  Event Handlers
        //----------------------------------------------------

        //----------------------------------------------------

        //----------------------------------------------------
        //  Public Methods
        //----------------------------------------------------
        Form.prototype.getDefaults = function() {
            return _defaults;
        };

        Form.prototype.toString = function() {
            return '[ ' + (this.options.name || 'Form') + ' ]';
        };

        Form.prototype.init = function() {
            // Initialization Code

            return this;
        };

        return Form;

    }(window, document, Zepto));
```

Just keep in mind that the code is self-contained in an immediately invoked function expression or IIFE/closure. When we initialize `App.Form` the `Form` constructor will be called and our public method, `init`, will initialize any code that we write within it. So let's get started there, by attaching the appropriate events.

Initializing our forms

We need to initialize our forms, but we don't really need to create a new object for each one. What we can do is have it event-driven and then handle our validation using the attributes we've written for each input. But let's look at our event setup.

Attaching events

First, let's perform attaching events:

```
this.$element.
  on('submit', 'form', handleFormSubmission);

this.$cache.loginFormContainer.
  on('click', 'a[href="#forgot-password"]',
handleForgotPasswordClick).
  on('click', 'a[href="#register"]', handleRegisterClick);

this.$cache.registerFormContainer.

  on('click', 'a[href="#login"]', handleLoginClick);
```

In the preceding code we've got a couple of things going on. First, we look for the submission of any form on the page. Then, we're going to call the `handleFormSubmission` method, which we'll write in a moment, when we submit the form. The following event listeners are basically a show/hide for the login and register buttons.

Nothing new or groundbreaking here, we're basically doing some setup work and can always come back to this if we need to. The key here is that we have not created a new instance of an object for each form, instead we've generalized our code to just listen for the `submit` event on each form. Now let's create, or set up, our handlers and then write the functionality for them.

Event handlers

Now, let's take a look at the event handlers.

```
function handleFormSubmission(e) {
  e.preventDefault();

  // Code goes here
}

function handleForgotPasswordClick(e) {
  e.preventDefault();

  // Code goes here
}
```

```
function handleRegisterClick(e) {
  e.preventDefault();

  // Code goes here
}

function handleLoginClick(e) {
  e.preventDefault();

  // Code goes here
}
```

We haven't done anything new here, the only steps we've taken are to stub out our code so that we know where each piece of functionality will exist. From here, we look at the validation code for the submission of each form. We will not be looking at the show/hide functionality of each form, but you do have the source code that accompanies this book in case you are curious about how that works.

Validating our input

We're going to take a look at the `handleFormSubmission` method and go step-by-step in order to understand how we are validating our fields. If you feel confused at any step of the process, don't worry about it. We've all been here, and I myself struggle sometimes with form validation and how it should be handled on a project-by-project basis.

First, let's start by caching the variables we'll be using:

```
function handleFormSubmission(e) {
  var $target, errors, $required, fields, $errorText, i, required_
fields_length;
}
```

These variables describe themselves, a standard practice since we want to understand what is going on and thus, attaching meaningful names to our variables is a must.

Now, we need to prevent default behavior of our form; meaning we don't want the form to submit just yet. To do this, let's do the following:

```
function handleFormSubmission(e) {
  var $target, errors, $required, fields, $errorText, i, required_
fields_length;
  e.preventDefault();
}
```

We have added `e.preventDefault`, which tells the event to prevent the event's default behavior in the browser. Next, we want to define the target, empty out any previous error messaging, create an empty errors object, and then find all required elements. This can be done with the following code:

```
function handleFormSubmission(e) {
  // Previous code
  $target = $(e.target);
  $target.find('.error-messaging').empty();
  errors = { 'required': [], 'invalid': [] };
  $required = $target.find(':required');
}
```

Note that our `errors` object contains two arrays: a `required` array and an `invalid` array. This `errors` array will keep track of what's gone wrong; for example, if a field is `required` and the value is `empty`, then we'll populate the `required` array within the `error` object, but if an input is filled out but is not valid, then we'll to populate the `invalid` object within the `errors` object.

Now, remember when we added the `required` class but not the `required` attribute to our profile form? Well the preceding code wouldn't catch that, so we'll run into issues. In order to prevent that, we can do the following:

```
function handleFormSubmission(e) {
  // Previous code
  if ($required.length === 0) {
    $required = $target.find('.required')
  }
}
```

This code helps solve our issue with the `required` class, but does have a logical flaw. Can you find that flaw? I'll leave that up to you as possibly a teaser for you to solve. The next step in this process is to find all our `form` elements, and then find the `required` fields and check that they've been filled out:

```
function handleFormSubmission(e) {
  //Previous code
  fields = $target[0].elements;

  i = 0, required_fields_length = $required.length;
  for (i; i < required_fields_length; i++) {
  if ($required[i].value === '') {
      errors.required.push($($required[i]).prev('label').text() + ' is
required.');
    }
  }
}
```

At this point, we are basically populating our `invalid` array inside the `error` object if the field is empty. If the field is empty, we collect the value of the label associated with that field and attach a customized message that will be presented to the user.

> Unfortunately, specific validation won't be covered, such as e-mail, numerical, and other constraints. However, there is room here for you to explore and add to this bit of code, but hopefully this is enough for you to understand validation, requirements, and how to handle these use cases in your code.

The last step is to check for errors, and if they do exist present these errors to the user so that they can correct them accordingly:

```
function handleFormSubmission(e) {
  //Previous code
  if (errors.required.length === 0 && errors.invalid.length === 0) {
    console.log('Form Requirements and Validations Passed');
    return;
  } else {
    $errorText = $('<ul />');

    if (errors.required.length !== 0) {
      $errorText.append('<li>' + errors.required.join('</li><li>') +
'</li>');
    }

    if (errors.invalid.length !== 0) {
      $errorText.append('<li>' + errors.invalid.join('</li><li>') +
'</li>');
    }

    $target.find('.error-messaging').append($errorText);
  }
}
```

Our check is pretty simple, we essentially check if the `invalid` and `required` arrays are empty within the `error` object. If they are, we want to continue with the submission—which in this case would be an AJAX call. Otherwise, we want to create an unordered list containing the errors and then append them to the form, so that the user understands what went wrong without a page refresh.

Hopefully, this section has helped you understand the approach in validating a form. With the latest support of the HTML5 specification, much of the work is taken care of by the browser. This speeds up development by lessening the development of customized components and helps us focus on delivery. Now as a bonus feature, we move on to styling our form.

Form styling for iOS

In this section we look at styling our form. If we currently test out our form on an iOS device or even a desktop browser, it won't be pretty. In fact you might be a little upset at how ugly it really is. So, let's style it and make everyone happy. We'll start with basic styling that helps achieve a good look. Then we'll consider how we can customize our components using CSS3 features.

Basic styling

Styling forms is pretty easy. We can simply use the elements themselves, but there is a single "gotcha". You might notice the fact that we've specified `[type="datetime"]` in one of our selectors. This is because the `datetime` input type shows a select menu type of UI in iOS and thus, the typical input selector won't apply. Otherwise, there's not much to the basic styling that really pops out here, and it basically gives us the styles we've seen in the previous images when we discussed the input types used in our forms.

```
/*!
  Forms Styling
*/

label {
    color: #FFF;
    font-family: 'Helvetica', 'Arial', sans-serif;
    font-size: 12px;
    display: block;
    margin: 10px 0 5px 0;
}

input, select, input[type="datetime"], textarea {

    font-size: 13px;

    display: block;
    margin: 0;
    padding: 5px 8px;
}

input[type="submit"] {
    margin: 10px 0;
}

.form-container {
   display: none;
```

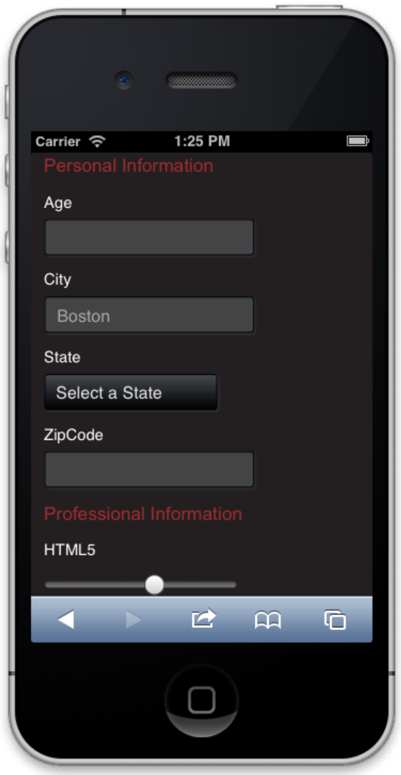
```
    margin: 15px 0;
}

.form-container.active {
  display: block;
}

form h2 {
    margin: 10px 0 5px 0;
}

.error-messaging ul {
  list-style: square outside;
  margin: 5px 0 0 0;
  padding: 0 0 0 12px;
}

.error-messaging li {
    color: #A12E33;
    font-family: 'Helvetica', 'Arial', sans-serif;
    font-size: 12px;
}
```

Custom styling

This is where a lot of the magic happens. In this section, we use custom CSS3 styles to customize our components. The following styles will customize our inputs, selects, and give us a more stylized form that matches our current styling. Some things you may want to keep in mind when reviewing the styling are the use of the CSS3 `gradient` properties as `background` and the use of `border-radius`.

```
/*!
  Forms Styling
*/

label {
    color: #FFF;
    font-family: 'Helvetica', 'Arial', sans-serif;
    font-size: 12px;
    display: block;
    margin: 10px 0 5px 0;
}

input, select, input[type="date-time"], textarea {
    background: rgb(69,72,77);
    background: -moz-linear-gradient(top, rgba(69,72,77,1) 0%,
rgba(0,0,0,1) 100%);
    background: -webkit-gradient(linear, left top, left bottom, color-
stop(0%,rgba(69,72,77,1)), color-stop(100%,rgba(0,0,0,1)));
    background: -webkit-linear-gradient(top, rgba(69,72,77,1)
0%,rgba(0,0,0,1) 100%);
    background: -o-linear-gradient(top, rgba(69,72,77,1)
0%,rgba(0,0,0,1) 100%);
    background: -ms-linear-gradient(top, rgba(69,72,77,1)
0%,rgba(0,0,0,1) 100%);
    background: linear-gradient(to bottom, rgba(69,72,77,1)
0%,rgba(0,0,0,1) 100%);

    font-size: 13px;
    color: #e5e5e5;

    border: 1px solid #000918;

    -moz-border-radius: 3px;
    -webkit-border-radius: 3px;
    -ms-border-radius: 3px;
    -o-border-radius: 3px;
    border-radius: 3px;

    display: block;
    margin: 0;
    padding: 5px 8px;

    -moz-box-shadow: 1px 1px 1px #333;
    -webkit-box-shadow: 1px 1px 1px #333;
    -ms-box-shadow: 1px 1px 1px #333;
    -o-box-shadow: 1px 1px 1px #333;
    box-shadow: 1px 1px 1px #333;
}

input[type="text"],
input[type="number"],
input[type="email"],
input[type="datetime"],
input[type="password"],
textarea {
  background: -webkit-gradient(linear, left top, left bottom, color-
stop(0, #42422F), color-stop(0.09, #444));
}

input[type="submit"] {
    margin: 10px 0;
}

.form-container {
   display: none;
   margin: 15px 0;
}

.form-container.active {
  display: block;
}

form h2 {
    margin: 10px 0 5px 0;
}

.error-messaging ul {
  list-style: square outside;
  margin: 5px 0 0 0;
  padding: 0 0 0 12px;
}

.error-messaging li {
    color: #A12E33;
    font-family: 'Helvetica', 'Arial', sans-serif;
    font-size: 12px;
}
```

When we apply the preceding styles, we get the following UI:



The range input type

As you can see, we've given our form a completely new look and feel and have easily styled the select component, something that is not easily done on desktop browsers. Going on from these styles, I would recommend checking out the `-webkit-appearance` property that essentially allows you to further customize your form and gives you much more control in terms of how components can get styled. However, at this point you should have a solid foundation to build HTML5 forms for iOS.

Summary

In this chapter, we have reviewed the latest HTML5 input types and attributes specifically for our sample application. We then discussed the layout of forms on iOS and its limitations. Finally, we developed a couple of forms and attached a very basic validation script that used these latest input and attributes to our advantage. As a bonus, we went into styling our forms specifically for WebKit browsers, including mobile Safari on iOS.

We should now have a solid grasp of forms on iPhone and iPad, and how they can be used with the latest HTML5 technology for our advantage. This chapter helped to demonstrate the use of forms and the considerations we need to take in order to create a user-friendly form. On top of that, we now move into location awareness in our next chapter and will use some of the concepts learned here to extend the experience.

6
Location-aware Applications

Geolocation is a widely requested feature in today's applications, giving accurate location-specific information to the user. In this chapter, we'll review the Geolocation API in the HTML5 specification. With this knowledge, we'll go ahead and build a wrapper that allows us to easily tap into this feature. Once we have a thorough understanding of how we can retrieve the user's location, we put our newly found knowledge to use with a simple application that uses the Google Maps API. At the end of this chapter you should have a thorough understanding of the Geolocation specification, have a simple example of its implementation, and as a bonus you should have gained some experience in using the Google Maps API. So let's start out by exploring the specification.

In this chapter we will cover:

- Geolocation specification
- Retrieving the user's current location
- Watching the user's location
- Handling geolocation errors
- Google Maps API
- Tying Google Maps with geolocation
- Customizing Google Maps

Geolocation specification

Location-based services have been around for quite a while and have evolved over time. In essence these services strive to provide features that allow the use of time and location in various types of programs. However, until now there has not been a useful set of tools for the frontend. Therefore, the **W3C** (**World Wide Web Consortium**), has tried to standardize the API that retrieves geographical location from a client-side device, whether that is your desktop computer, mobile phone, or tablet.

Implementation

The Geolocation API defines a high-level interface to location information associated only with the device hosting the implementation, such as latitude and longitude. The API itself is agnostic of the underlying location information sources.

(As mentioned at `http://dev.w3.org/geo/api/spec-source.html#introduction`.)

Common ways that browsers have implemented the Geolocation API involve **Global Positioning Systems** (**GPS**), IP addresses, WIFI and Bluetooth MAC addresses, and basic user input. Due to the various ways these technologies work and the varying degrees browser vendors choose to implement the specification, there is no guarantee that this API will return the location of the user or device. Therefore it is up to you, as the developer, to make sure your users are also aware of this limitation and that reasonable expectations are explained to all involved.

Scope, security, and privacy

When it comes to implementing Geolocation into our apps the only part we'll need to worry about is the scripts. There is no need to provide any sort of markup and there's no need to query or tap some external resource or API. The implementation of Geolocation is strictly on the scripting side and is directly associated with the device in use. It is also useful to know that the position is delivered in terms of World Geodetic System coordinates or latitude and longitude.

Security and privacy concerns must also be considered when exposing a user's location. From the security methods used to retrieve and store this information to how it is distributed among other parties, each device implementing it must provide a mechanism that protects the user's privacy. Therefore, the following considerations are required by the W3C specification:

- Permission from the user is required to send the location information.
- Location information must only be requested when necessary.
- The user must approve of retransmitting location information.
- The party holding this information must disclose to the user that they are collecting location data, including its purpose, security, accessibility, sharing (if the data will be shared among other parties), and the length of time such data will be stored.

Keep in mind that applications written for mobile Safari do not have direct access to the device. They can only query the browser to access the device on their behalf. Therefore, your application is requesting the browser for specific information and the browser does the work for you, but you never have one-on-one communication with the device itself.

Overall, the specification takes into account the concerns that arise with sharing personal information, such as Geolocation, with other parties. However, these considerations do not take into account the complexity that might arise when a user inadvertently grants permission or if the users decides to change their mind. For these reasons, the specification does make the following recommendation:

> *Mitigation and in-depth defensive measures are an implementation responsibility and not prescribed by this specification. However, in designing these measures, implementers are advised to enable user awareness of location sharing, and to provide easy access to interfaces that enable revocation of permissions.*

(As mentioned at `http://www.w3.org/TR/geolocation-API/#implementation_considerations`.)

With these concerns and considerations in mind, we now briefly dive into describing API. In the following section, we'll look at how API is built and specifically look at parts that will be utilized in the application built in this chapter.

API descriptions

At this point of the chapter you may be wondering why we haven't looked at the code yet, and although that is a valid concern, my goal here is to help you understand the Geolocation API thoroughly and to guide you through the actual W3C specification. So in this chapter we look at four interfaces or exposed behaviors that define the `Geolocation` specification, including the `Geolocation`, `PositionOptions`, `Position`, `Coordinates` and `PositionError` interfaces. If you get confused by any of the information provided here, there's no need to worry. Consider this section more of reference material that can help you increase your knowledge on the subject.

The Geolocation interface

The Geolocation object is used to determine the location of the device. When we instantiate the Geolocation object a user-agent algorithm is used to determine the location, then a `position` object is created and populated with the data. If we look at the W3C specification, the Geolocation is defined as this:

```
interface Geolocation {
    void getCurrentPosition(PositionCallback successCallback,
            optional PositionErrorCallback errorCallback,
            optional PositionOptions options);

    long watchPosition(PositionCallback successCallback,
            optional PositionErrorCallback errorCallback,
            optional PositionOptions options);

    void clearWatch(long watchId);
};
```

(As seen at `http://www.w3.org/TR/geolocation-API/#geolocation`.)

The previous code is not JavaScript and is a description of the API or **Interface Definition Language** (**IDL**). If its confusing, don't worry, I felt the same way when I first looked at a specification page. However, what you are looking at here is the description of the Geolocation object. When you read the previous code, you should gather the following information:

There are three methods:

- `getCurrentPosition`, which accepts three parameters, two of which are optional
- `watchPosition`, which accepts three parameters, two of which are optional
- `clearWatch`, which accepts one parameter

You should now know that there are three methods associated with the Geolocation object and each has a specific purpose as described by the function name. So let's go over these three methods, beginning with `getCurrentPosition`, which as you may have guessed obtains the current location of the device or attempts to.

The getCurrentPosition method

As described earlier, this method accepts three parameters, two of which are optional. The first parameter should be a `callback` method for a successful request. The second and third parameters are completely optional. The second parameter, if defined, is another `callback` method for when an error occurs. The last parameter is an `options` object as defined by the `PositionsOptions` interface.

The watchPosition method

The `watchPosition` method also accepts three parameters, which are the same as the `getCurrentPosition` method parameters. The only difference here is that this method will continuously fire the `successCallback`, or the first parameter, until the `clearWatch` method is called. Keep in mind that the `successCallback` will only fire if the position has changed and is thus not dependent on any time option. This method also returns a long value that defines the watch operation—this is what is used to clear it with the `clearWatch` method.

The clearWatch method

As we've already discussed, `clearWatch` is used to stop the process set by `watchPosition`. To use this method we must use the long value returned by `watchPosition` and send it in as a parameter to `clearWatch`.

The PositionOptions interface

We have seen that the `PositionOptions` object is used to pass an optional parameter to both the `getCurrentPosition` and `watchPosition` methods. This object is defined by W3C as follows:

```
interface PositionOptions {
    attribute boolean enableHighAccuracy;
    attribute long timeout;
    attribute long maximumAge;
};
```

(As seen at `http://www.w3.org/TR/geolocation-API/#position-options`.)

What we should take from this is that we can create an object with the key/value pairs for `enableHighAccuracy`, `timeout`, and `maximumAge`. This object would look like the following in our JavaScript code:

```
var positionOptions = {
    'enableHighAccuracy': false,
    'timeout': Infinity,
    'maximumAge': 0
};
```

But what do these values mean? Well, lucky for us this is all defined in the specification. Don't worry though, here's a simple explanation for each of these options.

The enableHighAccuracy option

This option basically hints to the device that the application would like to receive the best possible results. The default is set to `false`, because if set to `true` it may result in slower response times and/or increased power consumption. Keep in mind that the user might deny this feature and that the device may not be able to provide more accurate results.

The timeout option

Timeout is defined as the amount of time, in milliseconds, to wait until the successCallback is invoked. If the amount of time to get location data exceeds this value then the error callback is invoked and a `PositionError` code of `TIMEOUT` is sent. By default, the value is set to `Infinity`.

The maximumAge option

The maximum age option is in reference to using a cached position whose age is not greater than the amount set by this option. By default this attribute is set to `0` and therefore an attempt to acquire a new position object is made each time. If this option is set to `Infinity` then the cached position is returned each time.

Now that we understand the options, we could pass this object as the third parameter to the `getCurrentPosition` and `watchPosition` methods. A simple implementation of the API would look something like this:

```
var positionOptions = {
    'enableHighAccuracy': false,
    'timeout': Infinity,
    'maximumAge': 0
};

function successCallback(position) {}

function errorCallback(positionError) {}

// Get the current position
navigator.geolocation.getCurrentPosition(successCallback,
errorCallback, positionOptions);

// Watch for position changes
navigator.geolocation.watchPosition(successCallback, errorCallback,
positionOptions);
```

Now we know how to customize our calls to the Geolocation API, but what does the data look like when a success call is made? Or, what does an error return back? These are extremely useful to know in order to develop a good wrapper around the Geolocation API. So let's take a look at the coordinates and position error interfaces.

The Position interface

The Position interface is just a container for the information returned by device implementation of the Geolocation API. It returns a `Coordinates` object and `Timestamp`. This is described in the W3C specification as follows:

```
interface Position {
    readonly attribute Coordinates coords;
    readonly attribute DOMTimeStamp timestamp;
};
```

(As seen at `http://www.w3.org/TR/geolocation-API/#position`.)

In context of what we have discussed up to this point, the position interface comes into play on the `successCallback` of the `getCurrentPosition` method. If you recall, this method accepts one parameter called `options` that is of the `position` object defined earlier. In practice, if we wanted to log out the coordinates and timestamp we could do the following:

```
function successCallback(position) {
    console.log(position.coords);
    console.log(position.timestamp);
}
```

The timestamp returned is represented as `DOMTimeStamp` and the `coords` object contains the geographic coordinates with additional information, which is defined by the `Coordinates` interface.

The Coordinates interface

As we have discussed previously, the `successCallback` for `getCurrentPosition` and `watchPosition` returns a `position` object that contains the `Coordinates` object. This `Coordinates` object contains multiple attributes, which are described in the following table:

Attribute	Description
`latitude`	Geographic coordinate in decimal degrees.
`longitude`	Geographic coordinate in decimal degrees.
`altitude`	Height of the position in meters. Null if not present.
`accuracy`	Accuracy of longitude and latitude in meters. Null if not present. Must be a non-negative real number.
`altitudeAccuracy`	Accuracy of altitude in meters. Null if not present. Must be a non-negative real number.
`heading`	Direction of travel in degrees (0° ≤ heading ≤ 360°) clockwise. Null if not present. If stationary value must be NaN.
`speed`	Magnitude of current velocity in meters per second. Null if not present. Must be a non-negative real number.

(Seen at `http://www.w3.org/TR/geolocation-API/#coordinates`.)

Now that we know the properties available through the `Coordinates` interface, we can access these properties with the following implementation.

```
function successCallback(position) {
    console.log(position.coords);
    console.log(position.coords.lattitude);
    console.log(position.coords.longitude);
    console.log(position.timestamp);
}
```

As you can see, we can access the attributes via the `position.coords` object. In this way it is extremely easy for us to access the user's current location and tie it to other APIs, which is what we will be doing shortly with the Google Maps API. Lastly, let's discuss the `PositionError` interface so that we know how to handle errors efficiently within our applications.

The PositionError interface

The `PositionError` interface comes into play when an error occurs on the `getCurrentPosition` or `watchPosition` method(s). This interface describes the codes that are sent to our error handler or callback, and a message. The W3C explains the `PositionError` interface as following:

```
interface PositionError {
    const unsigned short PERMISSION_DENIED = 1;
    const unsigned short POSITION_UNAVAILABLE = 2;
    const unsigned short TIMEOUT = 3;
    readonly attribute unsigned short code;
    readonly attribute DOMString message;
};
```

(As seen at `http://www.w3.org/TR/geolocation-API/#position-error`.)

The previous code describes two attributes that are sent over as an object to the error handler, the two attributes being `code` and `message`.

The `code` attribute could be any of the three constants,

- `PERMISSION_DENIED` (error code 1): The user chose not to let the browser have access to the location information.
- `POSITION_UNAVAILABLE` (error code 2): The location of the device could not be determined by the browser.
- `TIMEOUT` (error code 3): The total time in acquiring the location information has surpassed the specified timeout property in the PositionOptions interface.

The second parameter, `message`, would be a DOMstring or String describing the problem.

In our implementation we could then do something like this:

```
function errorCallback(positionError) {
    if (positionError.code === 3) {
        console.log("A Timeout has occurred");
        console.log("Additional Details: " +
positionError.message);
    }
}
```

As you can see, we can easily determine an error using the `PositionError` interface and customize our error messages based on the codes provided. At this point, you should have a solid foundation on which you can build. We'll now briefly go over some use cases for implementing the Geolocation API into our applications and then move into building our application for this book. You can skim over the next section, as it will only give you ideas of how Geolocation can or has been implemented.

Use cases

Before we jump into building our application, I'd like to review some instances in which Geolocation can be implemented into our applications. This will be sweet and short, but it will help you formulate ideas on how to efficiently implement this feature. Much of this is already located in the W3C specification, but I hope this will give you more insight into how useful the specification is and why you should definitely check it out when exploring new features.

Points of interest

We have always been interested in our surroundings whether that is food, beer, or entertainment. So wouldn't it be cool if we could list out possible points of interest in our applications that relate to the content the user is accessing? We can do this with the Geolocation API. By finding the user's current location and tapping into open APIs from third-party vendors we can easily find where the user is and present relevant information about the area they are currently located in.

Route navigation

We've seen this being done before, plenty of times with native applications on our phones. It's even possible that your phone comes preloaded with this feature for which so many have paid hundreds of dollars before. Now, with the HTML5 Geolocation API we can build this using the `currentPosition` method and tie it to something like Google Maps so that we can present the user with a route. If we wanted to, we could even possibly make it a live application using the `watchPosition` method, although you may run into API access limits, so keep that in mind when building your apps.

Latest information

Another useful feature in this application is to present the user with the latest information. This can be easily created if we expose an API from a backend system, but what if we went further and implemented information outside our own application based on the current position of the user. For example, if I lived in Boston and went on a trip to Seattle, I would probably want to know what's going on in Seattle and not Boston, so my application should probably handle that scenario. With the HTML5 Geolocation API we can achieve this very easily without much complication.

We now have a solid understanding of the Geolocation API, from theoretical understanding to simple implementation, we have gone over everything we need to know about Geolocation and how to work with it. Use cases have also been defined to help us find a way of integrating it into our applications, more likely than not you'll find new and innovative ways of using this piece of technology in your applications. As for now, let's gear ourselves for a simple use case scenario of pointing out the user's current location using the Google Maps API. So let's get started.

Google Maps API

Before we get on to implementing Geolocation with Google Maps, we need to do some setup work that is pretty simple. As you may already know, Google Maps provides an API that you can tap into to implement their maps into your application, making it easy for you to display information relative to a user's input or even better—their current location. However, for several reasons we need to use an API key from Google to authorize our application and keep track of the requests made from your application. In this section we'll cover the setup work and hopefully move you along quickly.

The API(s)

First of all, you need to know that there are several APIs related to Maps, including JavaScript v3, Places, iOS SDK, Android API, Earth API, and more. For our purposes we'll be using the JavaScript API v3; note that we will be using using version 3 of the API. If you would like more information on the several APIs you can visit the following page:

`https://developers.google.com/maps/`

Obtaining an API key

If you have been following along you'll notice that we need an API key for our application. Google has provided the following reasoning for this:

> *Using an API key enables you to monitor your application's Maps API usage, and ensures that Google can contact you about your application if necessary. If your application's Maps API usage exceeds the Usage Limits, you must load the Maps API using an API key in order to purchase additional quota.*

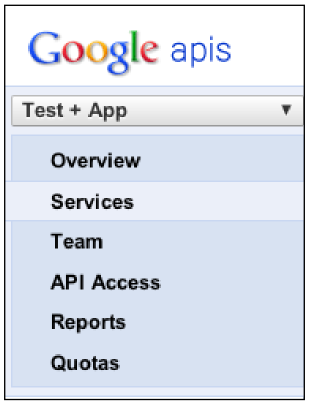
(As seen at `https://developers.google.com/maps/documentation/javascript/tutorial#api_key`.)

Activating the service

Now let's get started on creating the API key. First, log in to your Google account at the following URL:

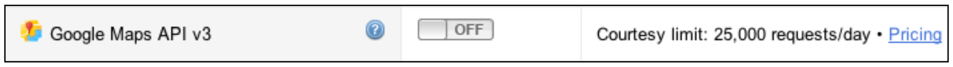
`https://code.google.com/apis/console`

Once we log in at the previous URL, we select the **Services** tab.



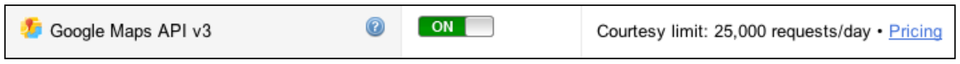
The Services tab

In the **Services** tab, we are presented with all the services provided by Google. In this list we need to activate the Google Maps API v3. It should look like this:



Inactive Google Maps API

When you click on the **OFF** button, the service will activate and should look like the following:

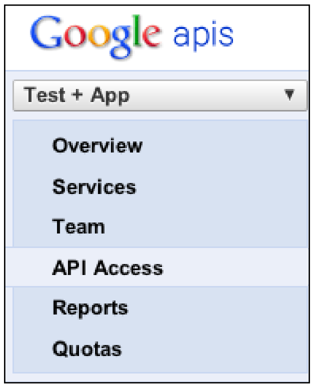


Active Google Maps API

The Google Maps API v3 service is now activated under your Google account. The next step is to retrieve the key that will be used in our implementation of the Geolocation API.

Retrieving the key

Now that the service has been activated under our Google account, let's get the key—the final step. To do this, switch to the **API Access** tab in the left-hand navigation.



The API Access tab

When we access this page we'll be presented with a **Simple API Access** section that will contain our generated key. This is the key you'll want to use to authorize your Google Maps implementation in the next section. Along with the key you'll notice that it will list the referrers, when it was activated, and who activated the key (you). To the right of all this information, you'll also notice a couple of options. These options include the ability to generate a new key, edit the referrers, and ultimately delete the key generated.

Note you also have the ability to set up an OAuth 2.0 Client ID, which will secure your applications. This is definitely recommended if you'll be dealing with sensitive information, which in a way you would be because you will be working with user location. However, the setup and use of OAuth is beyond the scope of this book, but I recommend that you take some time to learn this new authentication method and implement it within your own application once you get a solid foundation working with APIs.

With the API key in hand we are now all set to start implementing Geolocation with Google Maps. The following section will take what we've learned and use the simple methods available to us for putting a Google Map on our page. In this regard I hope it sparks your interest into the Google Maps API as it has been developed over time and is a great framework to use in almost any application. Now let's get started developing some cool stuff.

Geolocation and Google Maps

If you have been following along from the beginning of this chapter you should have a thorough understanding of the Geolocation API and have your Google account set up to tap into the Google Maps JavaScript API. If you haven't been following along that's okay as well, since this section is primarily driven to show how to implement both technologies. This section will prepare our location page within our application and then move quickly to implement Geolocation with Google Maps.

Markup preparation

In the previous chapter, we did some setup work to get our application going; we will follow the same setup work here to make sure all our pages are consistent. So let's open up the markup page related to `location` in `/location/index.html` of the source files accompanying this book. When we have this page open in our text editor, let's make the following updates to the markup:

- Update the navigation to reflect a select menu.
- Include the `location.css` file that will have the page specific styling for this page.
- Remove unused scripts from the bottom of the page.
- Include `App.Location.js`.
- Initialize `App.Location` after the inclusion of `main.js`.

Once we have made these updates, your markup should look like this:

```
<!DOCTYPE html>
<html class="no-js">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
    <meta name="HandheldFriendly" content="True">
    <meta name="MobileOptimized" content="320">
    <meta name="viewport" content="width=device-width">

    <!-- IOS THUMBS -->

    <!-- APPLE META TAGS -->

    <link rel="stylesheet" href="../css/normalize.css">
    <link rel="stylesheet" href="../css/main.css">
    <link rel="stylesheet" href="../css/location.css">
    <script src="../js/vendor/modernizr-2.6.1.min.js"></script>
</head>
    <body>
        <!-- Add your site or application content here -->
        <div class="site-wrapper">
            <header>
                <hgroup>
                    <h1>iPhone Web Application Development</h1>
                    <h2>Location Aware Apps</h2>
                </hgroup>
                <nav>
                    <select>
                        <!-- OPTIONS HERE -->
                    </select>
                </nav>
            </header>
            <footer>
                <p>iPhone Web Application Development &copy; 2013</p>
            </footer>
        </div>

        <script src="../js/vendor/zepto.min.js"></script>
        <script src="../js/helper.js"></script>
        <!-- BEGIN: Our Framework -->
        <script src="../js/App/App.js"></script>
        <script src="../js/App/App.Nav.js"></script>
        <script src="../js/App/App.Location.js"></script>
        <!-- END: Our Framework -->
        <script src="../js/main.js"></script>
        <script> new App.Location({ 'element': document.body }); </
script>
    </body>
</html>
```

Note that comments were added where more markup should exist. The markup that pertains to these sections is in the source code provided with the book. Please look there for more on what should exist in those sections.

Now that we have brought the markup up to the consistent layout of our previous pages we are ready to start customizing this application for location awareness. The next step in the process is to prepare the markup for the additional functionality we'll build in. To do this, here is what we need to do:

- Include the Google Maps API JavaScript.
- Include the `Geolocation` wrapper we'll be building.
- Create a `div` that will contain our map.

When we follow the previous directions, our markup will look like this:

```
<!DOCTYPE html>
<html class="no-js">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta name="description" content="">
    <meta name="HandheldFriendly" content="True">
    <meta name="MobileOptimized" content="320">
    <meta name="viewport" content="width=device-width">

    <!-- IOS THUMBS -->

    <!-- APPLE META TAGS -->

    <link rel="stylesheet" href="../css/normalize.css">
    <link rel="stylesheet" href="../css/main.css">
    <link rel="stylesheet" href="../css/location.css">
    <script src="../js/vendor/modernizr-2.6.1.min.js"></script>
</head>
    <body>
        <!-- Add your site or application content here -->
        <div class="site-wrapper">
            <header>
                <hgroup>
                    <h1>iPhone Web Application Development</h1>
                    <h2>Location Aware Apps</h2>
                </hgroup>
                <nav>
                    <select>
                        <!-- OPTIONS HERE -->
                    </select>
                </nav>
            </header>
            <div id="map_canvas"></div>
            <footer>
                <p>iPhone Web Application Development &copy; 2013</p>
            </footer>
        </div>

        <script src="//maps.googleapis.com/maps/api/js?key=YOUR_API_
KEY&sensor=SET_TO_TRUE_OR_FALSE"></script>
        <script src="../js/vendor/zepto.min.js"></script>
        <script src="../js/helper.js"></script>
  <script src="../js/Geolocation.js"></script>
        <!-- BEGIN: Our Framework -->
        <script src="../js/App/App.js"></script>
        <script src="../js/App/App.Nav.js"></script>
        <script src="../js/App/App.Location.js"></script>
        <!-- END: Our Framework -->
        <script src="../js/main.js"></script>
        <script> new App.Location({ 'element': document.body }); </
script>
    </body>
</html>
```

As you can see, there is not much of a difference. What we have done here is include in a new script that includes the Google Maps JavaScript. We then include another script named `Geolocation.js`, which will exist in `/js/` and finally we create a `div` with an ID of `map_canvas` that exists between the header and footer.

> Note that you will need to include the API key, which you created in the last section, into the Google Maps JavaScript URL string, replacing `YOUR_API_KEY` with the key you were provided earlier. Also keep in mind that you must set the sensor parameter to either true or false. The sensor parameter tells Google Maps that the application uses a sensor, such as a GPS, for the user's location.

Okay, so our markup is now ready. We don't need to do anything else here and so now we'll move to the JavaScript, creating our `Geolocation` wrapper first and then implementing it into our `App.Location` class. So let's take a look at how we can make tapping into Geolocation easier within our applications.

The Geolocation wrapper

In most cases, we don't want to rewrite the same methods over and over for every use case. So we create wrappers that abstract the functionality of certain technologies so we can use them easily within our apps. This is what we're going to do now, abstract the Geolocation API so we can use it with the Google Maps API.

Let's get started by creating a `Geolocation.js` file inside of our `JavaScript` directory. As you may have already noticed this is not going to exist under the `App` namespace; this is because it is an abstract class that any application might be able to use. For our purpose we'll only want to get the current position of the user and we want to be able to use this information across our application, so we'll make it global.

Here is the basic template for our `Geolocation` class:

```
(function($){

    var _self, _defaults, _callbacks;

    // Default options
    _defaults = {};

    // Stores custom callbacks
    _callbacks = {};

    /**
        @constructor
    */
    function Geolocation(options) {
        this.options = $.extend({}, _defaults, options);

        _self = this;
    }

    Geolocation.prototype.toString = function() {
        return "[object " + this.constructor.name + "]";
    }

    // Exposess the Geolocation Function
    window.Geolocation = new Geolocation();

}(Zepto));
```

This is not different from any of the code we've written previously except for how we expose this class with the following code:

```
window.Geolocation = new Geolocation();
```

Instead of returning the `Geolocation` object we basically just initialize it and set it to the `window` object, which makes it global. You'll also notice the addition of a closure scoped variable named `_callbacks`, which will contain callbacks that the user can override when extending Geolocation functionality. Now let's extend this even further by including default values for retrieving the current position, and a general properties object that will hold all returned data from the Geolocation API:

```
// Default options
_defaults = {
    'currentPositionOptions': {
        'enableHighAccuracy': false,
        'timeout': 9000,
        'maximumAge': Infinity
    },
    'props': {}
};
```

These options will be used when we retrieve the user's location. As for now, let's leave these as it is, and create a callback that the user can override when a success or error occurs with the Geolocation API:

```
// Stores custom callbacks
_callbacks = {
    'getCurrentPositionCallback': function(){}
};
```

We'll see how to implement this shortly, but for now this will be a default method that will be used to do callbacks. Next, let's check if the device/browser actually supports the Geolocation API:

```
/**
    @constructor
*/
function Geolocation(options) {
    this.options = $.extend({}, _defaults, options);

    if(navigator.geolocation) {
        this.geolocation = navigator.geolocation;
    }

    _self = this;
    _self.props = this.options.props;
}
```

This is a fairly simple check for geolocation support, and essentially we just create a property on Geolocation called `geolocation` that will be set to the API if it exists. In this way, we don't have to do `navigator.geolcation` every time within the class. Also, it will make it easier to double-check if the geolocation functionality exists later on. At this point, we're ready to expose the `getCurrentPosition` method from the Geolocation API:

```
Geolocation.prototype.getCurrentPosition = function(callback) {
    if (typeof callback !== 'undefined') {
        _callbacks.getCurrentPositionCallback = callback;
    }

    if (typeof this.geolocation !== 'undefined') {
    this.geolocation.getCurrentPosition(currentPositionSuccess,
currentPositionError, _self.options.currentPositionOptions);

        return this;
    }

    return false;
};
```

The previous method is public and accessible because we have attached it to the prototype of Geolocation. This method will accept one parameter, a function callback that will be called on success or error of the `getCurrentPosition` call on the Geolocation API. This method checks to see if the parameter is not undefined and then reassigns based on what was sent in. We then do a check on the `geolocation` property we set in the constructor; if it's not undefined we call the `getCurrentPosition` method on the Geolocation API and send in the appropriate parameters. We then return the instance of our `Geolocation` class. If the `geolocation` property is not defined we return a Boolean of false, so error checking could also be done at the time the developer uses this method.

Note we are passing two undefined methods `currentPositionSuccess` and `currentPositionError`, which will be defined shortly. However, also notice we sent in the default properties we defined previously into this method as its third parameter. By doing this we give the developer the ability to further customize the experience of Geolocation functionality easily. You'll see how easy it will be to customize these values when we start developing the `App.Location.js` file.

At this point, all that's left is creating the earlier callbacks. So let's create the following `successCallback`:

```
function currentPositionSuccess(position) {
    _self.props.coords = position.coords;
    _self.props.timestamp = position.timestamp;

    _callbacks.getCurrentPositionCallback.call(_self, _self.props);
}
```

The last callback is called, as you may have guessed, when we have successfully retrieved the user's location. As defined by the W3C specification, this method accepts a single parameter—a `Position` object containing the coordinates and timestamp. We expose the returned information using the property `props` defined in the constructor. Once all this information is retrieved and set, the callback `getCurrentPositionCallback` is invoked and passed the retrieved properties.

> Note we are also changing the meaning of `this` inside the callback to be that of the instance of Geolocation by passing in `_self` as the first parameter to call.

Lastly, lets create our error callback:

```
    function currentPositionError(positionError) {
        _callbacks.getCurrentPositionCallback.call(_self,
positionError);
    }
```

This callback, as defined by the W3C specification, accepts one parameter, a `PositionError` object that has an error code along with a brief message. However, all we have to do is use the callback and pass this information along, similar to what was done in the `successCallback`. Except here, all we are doing is passing the `PositionError` object so that custom messages can be created outside of this wrapper.

And with that, we are done creating a simple wrapper for the Geolocation API. We can now easily tap into the API from within `App.Location.js`. So let's move on to extending the `App.Location` object and start using the Google Maps API with Geolocation.

Geolocation with Google Maps

So we're now ready to start implementing Geolocation with Google Maps using `App.Location`. We'll use the same boilerplate that has been used throughout the book to connect our `Geolocation` wrapper with the Google Maps API. To get started let's open up `App.Location.js` provided with the book's source code. When you have this open, it should look similar to the following code:

```
var App = window.App || {};

App.Location = (function(window, document, $){
    'use strict';

    var _defaults = {
        'name': 'Location'
    }, _self;

    function Location(options) {
        this.options = $.extend({}, _defaults, options);

        this.$element = $(this.options.element);
    }

    Location.prototype.getDefaults = function() {
        return _defaults;
    };

    Location.prototype.toString = function() {
        return '[ ' + (this.options.name || 'Location') + ' ]';
    };

    Location.prototype.init = function() {
        // Initialization Code

        return this;
    };

    return Location;

}(window, document, Zepto));
```

There's nothing new here if you have been following the book in order. But as a review, we have declared a new namespace called `Location` under the general `App` object. This namespace will contain all the functionality for our location page so it's perfect as a controller between Google Maps and Geolocation functionality. So let's start by caching the map element creating a closure scoped reference to the `Location` instance and then initializing it. The constructor should then look like this:

```
function Location(options) {
    this.options = $.extend({}, _defaults, options);

    this.$element = $(this.options.element);

    // Cache the map element
    this.$cache = {
        'map': this.$element.find('#map_canvas')
    };

    _self = this;

    this.init();
}
```

Here we have created a `$cache` property onto the instance of `Location`, this `$cache` property will contain the reference to the `map` element and can thus be accessed using this property. We then create a closure scoped self variable that reference the instance of `Location`. Finally we initialize our code by calling the `init` method located on the instance's prototype.

The next step in the process is to retrieve the user's current location using our `Geolocation` wrapper. We'll add this bit of code to the `initialize` method, as follows:

```
Location.prototype.init = function() {
    // Initialization Code
    Geolocation.getCurrentPosition(function(args){
        if(args.toString() !== '[object PositionError]') {
            _self.initGoogleMaps();
        } else {
            console.log("An ERROR has occurred: " + args.message);
        }
    });

    return this;
};
```

Here we can finally see the implementation of our `Geolocation` wrapper and how easy it is to integrate within our applications since the `Geolocation` class has taken care of validating and verifying the setup. The great part about this is that our callback actually handles the errors; by checking the object type of `PositionError` we are able to continue with the integration of Google Maps or logging out the error returned. Of course our way of handling the error should be more elaborate for the user, but for this case it helps identify how easy it is to go with this approach in our applications.

Now, let's take a look at how we can implement Google Maps with a successful callback by looking at the `initGoogleMaps` method called earlier:

```
Location.prototype.initGoogleMaps = function() {
    this.latlng = new google.maps.LatLng(Geolocation.props.coords.
latitude, Geolocation.props.coords.longitude);

    this.options.mapOptions.center = this.latlng;

    this.map = new google.maps.Map(this.$cache.map[0], this.options.
mapOptions);

    this.marker = new google.maps.Marker({
        'position': this.latlng,
        'map': this.map,
        'title': 'My Location'
    });

    this.infowindow = new google.maps.InfoWindow({
        'map': this.map,
        'position': this.latlng,
        'content': 'My Location!',
        'maxWidth': '140'
    });
```

There's a lot going on here but, believe it or not, we're pretty much done. So let's go through this step by step.

First, we are setting the `latlng` property to a new instance of the `LatLng` class that is part of the Google Maps API. This `class` constructor returns an object representing a geographic point (`https://developers.google.com/maps/documentation/javascript/reference#LatLng`). Although we already have the coordinates from the Geolocation API, we need to make sure that we create a Google Maps instance of `LatLng` because it will be used in the following methods.

Now, before moving on we need to sidestep for a moment. The Google Maps API is extremely extensive and customizable, allowing us to customize the look and feel of the map in pretty much every area. To explore this a bit more, let's create a `mapOptions` object on the defaults that will customize our map for mobile:

```
var _defaults = {
    'name': 'Location',
    'mapOptions': {
        'center': '',
        'zoom': 8,
        'mapTypeId': google.maps.MapTypeId.ROADMAP,
        'mapTypeControl': true,
        'mapTypeControlOptions': {
            'style': google.maps.MapTypeControlStyle.DROPDOWN_MENU
        },
        'draggable': true,
        'scaleControl': false,
        'zoomControl': true,
        'zoomControlOptions': {
            'style': google.maps.ZoomControlStyle.SMALL,
            'position': google.maps.ControlPosition.TOP_LEFT
        },
        'streetViewControl': false
    }
}, _self;
```

Now, we won't jump into this extensively but keep in mind that there are many options available to you that can be explored and optimized for our iPhone web application. I encourage you to visit the following URL and explore these options so that you are familiar with what is available to you:

`https://developers.google.com/maps/documentation/javascript/reference#MapOptions`

Let's return to the `initGoogleMaps` method that we were describing previously. Continuing from the initialization of the `LatLng` class, we now define the center property on the `mapOptions` object we just created. This property is set to the instance of `LatLng`:

```
this.options.mapOptions.center = this.latlng;
```

Now that we have defined all the properties we need to create Google Map, we initialize the `Map` class part of the Google Maps API:

```
this.map = new google.maps.Map(this.$cache.map[0],
this.options.mapOptions);
```

This method accepts the `div` element we created and have cached in our JavaScript, as its first parameter. The second parameter will be the `options` object we created. The reason we set the `center` property on the `mapOptions` object is because the initialization of the map would display the user's location. We have now completed the implementation of the Geolocation and Google Maps API.

Summary

In this chapter, we reviewed the Geolocation specification as defined by the W3C. We then used this information to build a wrapper so we could tap into the API easily. As a bonus we reviewed the Google Maps API, created an access key and then used our Geolocation wrapper to determine the user's location and display it to the user. You should now have a good understanding of determining a user's location and using it effectively. In the next chapter, we'll get into one-page application development, using the concepts we've learned and extending it using some additional open source libraries.

7
One-page Applications

Until this point we have developed individual pages with related static content. In this chapter we kick it up a notch by diving into one-page-application development. We've seen this in many of our web applications, including Pandora, Mint, and NPR. We'll cover the foundations of one-page-application development, from an introduction to MVC, Underscore, and Backbone to creating architecture with our sample application and utilizing the methods taught in the first section of this chapter. Once you complete this chapter you should have a solid understanding of concepts behind one-page-applications, which will allow you to continue to extend on this knowledge and help guide you on your way to building complex applications. So let's get started by first learning about MVC.

In this chapter, we will cover:

- MVC Architecture
- Introduction to `Underscore.js`
- Introduction to `Backbone.js`
- Creating a one-page application

Model-View-Controller or MVC

Model-View-Controller (**MVC**) is a heavily used design pattern in programming. A design pattern is essentially a reusable solution that solves common problems in programming. For example, the **Namespace** and **Immediately-Invoked Function Expressions** are patterns that are used throughout this book. MVC is another pattern to help solve the issue of separating the presentation and data layers. It helps us keep our markup and styling outside of the JavaScript; keeping our code organized, clean, and manageable—all essential requirements for creating one-page-applications. So let's briefly discuss the several parts of MVC, starting with models.

Models

A model is a description of an object, containing the attributes and methods that relate to it. Think of what makes up a song, for example the track's title, artist, album, year, duration, and more. In its essence, a model is a blueprint of your data.

Views

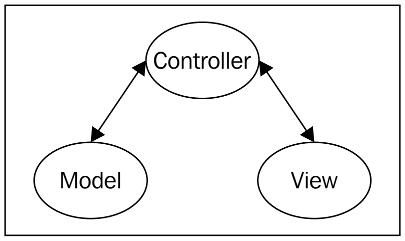
The view is a physical representation of the model. It essentially displays the appropriate attributes of the model to the user, the markup and styles used on the page. Accordingly, we use templates to populate our views with the data provided.

Controllers

Controllers are the mediators between the model and the view. The controller accepts actions and communicates information between the model and the view if necessary. For example, a user can edit properties on a model; when this is done the controller tells the View to update according to the user's updated information.

Relationships

The relationship established in an MVC application is critical to sticking with the design pattern. In MVC, theoretically, the model and view never speak with each other. Instead the controller does all the work; it describes an action, and when that action is called either the model, view, or both update accordingly. This type of relationship is established in the following diagram:



This diagram explains a traditional MVC structure, especially that the communication between the controller and model is two-way; the controller can send data to/from the model and vice versa for the view. However, the view and model never communicate, and there's a good reason for that. We want to make sure our logic is contained appropriately; therefore, if we wanted to delegate events properly for user actions, then that code would go into the view.

However, if we wanted to have utility methods, such as a `getName` method that combines a user's first name and last name appropriately, that code would be contained within a user model. Lastly, any sort of action that pertains to retrieving and displaying data would be contained in the controller.

Theoretically, this pattern helps us keep our code organized, clean, and efficient. In many cases this pattern can be directly applied, especially in many backend languages like Ruby, PHP, and Java. However, when we start applying this strictly to the frontend, we are confronted with many structural challenges. At the same time, we need this structure to create solid one-page-applications. The following sections will introduce you to the libraries we will use to solve these issues and more.

Introduction to Underscore.js

One of the libraries we will be utilizing in our sample application will be `Underscore.js`. Underscore has become extremely popular in the last couple of years due to the many utility methods it provides developers without extending built-in JavaScript objects, such as `String`, `Array`, or `Object`. While it provides many useful methods, the suite has also been optimized and tested across many of the most popular web browsers, including Internet Explorer. For these reasons, the community has widely adopted this library and continually supported it.

Implementation

Underscore is extremely easy to implement in our applications. In order to get Underscore going, all we need to do is include it on our page like so:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
        <title></title>
        <meta name="description" content="">
        <meta name="viewport" content="width=device-width">
    </head>
    <body>
        <script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.0/
jquery.min.js"></script>
        <script src="//cdnjs.cloudflare.com/ajax/libs/underscore.
js/1.4.3/underscore-min.js"></script>
    </body>
</html>
```

Once we include Underscore on our page, we have access to the library at the global scope using the _ object. We can then access any of the utility methods provided by the library by doing `_.methodName`. You can review all of the methods provided by Underscore online (`http://underscorejs.org/`), where all methods are documented and contain samples of their implementation. For now, let's briefly review some of the methods we'll be using in our application.

_.extend

The `extend` method in Underscore is very similar to the extend method we have been using from `Zepto` (`http://zeptojs.com/#$.extend`). If we look at the documentation provided on Underscore's website (`http://underscorejs.org/#extend`), we can see that it takes multiple objects with the first parameter being the destination object that gets returned once all objects are combined.

> *Copy all of the properties in the source objects over to the destination object, and return the destination object. It's in-order, so the last source will override properties of the same name in previous arguments.*

As an example, we can take a `Song` object and create an instance of it while also overriding its default attributes. This can be seen in the following example:

```
<script>
    function Song() {
        this.track = "Track Title";
        this.duration = 215;
        this.album = "Track Album";
    };

    var Sample = _.extend(new Song(), {
        'track': 'Sample Title',
        'duration': 0,
        'album': 'Sample Album'
    });
</script>
```

If we log out the `Sample` object, we'll notice that it has inherited from the `Song` constructor and overridden the default attributes `track`, `duration`, and `album`. Although we can improve the performance of inheritance using traditional JavaScript, using an `extend` method helps us focus on delivery. We'll look at how we can utilize this method to create a base architecture within our sample application later on in the chapter.

_.each

The `each` method is extremely helpful when we want to iterate over an `Array` or `Object`. In fact this is another method that we can find in `Zepto` and other popular libraries like `jQuery`. Although each library's implementation and performance is a little different, we'll be using Underscore's `_.each` method, so that we can stick within our application's architecture without introducing new dependencies. As per Underscore's documentation (`http://underscorejs.org/#each`), the use of `_.each` is similar to other implementations:

> *Iterates over a list of elements, yielding each in turn to an iterator function. The iterator is bound to the context object, if one is passed. Each invocation of iterator is called with three arguments: (element, index, list). If list is a JavaScript object, iterator's arguments will be (value, key, list). Delegates to the native forEach function if it exists.*

Let's take a look at an example of using `_.each` with the code we created in the previous section. We'll loop through the instance of `Sample` and log out the object's properties, including track, duration, and album. Because Underscore's implementation allows us to loop through an `Object`, just as easily as an `Array`, we can use this method to iterate over our `Sample` object's properties:

```
<script>
    function Song() {
        this.track = "Track Title";
        this.duration = 215;
        this.album = "Track Album";
    };

    var Sample = _.extend(new Song(), {
        'track': 'Sample Title',
        'duration': 0,
        'album': 'Sample Album'
    });

    _.each(Sample, function(value, key, list){
        console.log(key + ": " + value);
    });

</script>
```

The output from our log should look something like this:

```
track: Sample Title
duration: 0
album: Sample Album
```

As you can see, it's extremely easy to use Underscore's `each` method with arrays and objects. In our sample application, we'll use this method to loop through an array of objects to populate our page, but for now let's review one last important method we'll be using from Underscore's library.

_.template

Underscore has made it extremely easy for us to integrate templating into our applications. Out of the box, Underscore comes with a simple templating engine that can be customized for our purposes. In fact, it can also precompile your templates for easy debugging. Because Underscore's templating can interpolate variables, we can utilize it to dynamically change the page as we wish. The documentation provided by Underscore (`http://underscorejs.org/#template`) helps explain the different options we have when using templates:

> *Compiles JavaScript templates into functions that can be evaluated for rendering. Useful for rendering complicated bits of HTML from JSON data sources. Template functions can both interpolate variables, using <%= … %>, as well as execute arbitrary JavaScript code, with <% … %>. If you wish to interpolate a value, and have it be HTML-escaped, use <%- … %>. When you evaluate a template function, pass in a data object that has properties corresponding to the template's free variables. If you're writing a one-off, you can pass the data object as the second parameter to template in order to render immediately instead of returning a template function.*

Templating on the frontend can be difficult to understand at first, after all we were used to querying a backend, using AJAX, and retrieving markup that would then be rendered on the page. Today, best practices dictate we use RESTful APIs that send and retrieve data. So, theoretically, you should be working with data that is properly formed and can be interpolated. But where do our templates live, if not on the backend? Easily, in our markup:

```
<script type="tmpl/sample" id="sample-song">
    <section>
        <header>
            <h1><%= track %></h1>
            <strong><%= album %></strong>
        </header>
    </section>
</script>
```

Because the preceding script has an identified type for the browser, the browser avoids reading the contents inside this script. And because we can still target this using the ID, we can pick up the contents and then interpolate it with data using Underscore's `template` method:

```
<script>
    function Song() {
        this.track = "Track Title";
        this.duration = 215;
        this.album = "Track Album";
    };

    var Sample = _.extend(new Song(), {
        'track': 'Sample Title',
        'duration': 0,
        'album': 'Sample Album'
    });

    var template = _.template(Zepto('#sample-song').html(), Sample);

    Zepto(document.body).prepend(template);

</script>
```

The result of running the page, would be the following markup:

```
<body>
    <section>
        <header>
            <h1>Sample Title</h1>
            <strong>Sample Album</strong>
        </header>
    </section>
    <!-- scripts and template go here -->
</body>
```

As you can see, the content from within the template would be prepended to the body and the data interpolated, displaying the properties we wish to display; in this case the title and album name of the song. If this is a bit difficult to understand, don't worry about it too much, I myself had a lot of trouble trying to pick up the concept when the industry started moving into one-page applications that ran off raw data (`JSON`).

For now, these are the methods we'll be using consistently within the sample application to be built in this chapter. It is encouraged that you experiment with the `Underscore.js` library to discover some of the more advanced features that make your life easier, such as `_.map`, `_.reduce`, `_.indexOf`, `_.debounce`, and `_.clone`. However, let's move on to `Backbone.js` and how this library will be used to create our application.

Introduction to Backbone.js

To add structure to our one-page application, we will be using `Backbone.js`, a light framework that helps us apply the MVC design pattern. `Backbone.js` is one of the many MVC-type frameworks that help frontend development stick to best practices of separating out the data from the views or in particular, the DOM. On top of that, our applications can become quite complex for one-page apps. `Backbone.js` helps alleviate these issues and gets us going quickly. So let's start with discussing how MVC applies to this framework.

MVC and Backbone.js

There are many types of JavaScript frameworks that apply MVC differently, it is no different for Backbone. Backbone implements `Models`, `Views`, `Collections`, and `Routers`; it also includes an `Event`, `History`, and `Sync` system. As you can see, Backbone does not have a traditional Controller that was discussed earlier, but we can interpret `Views` as controllers. As per Backbone's documentation (`http://backbonejs.org/#FAQ-mvc`):

> *(…) in Backbone, the View class can also be thought of as a kind of controller, dispatching events that originate from the UI, with the HTML template serving as the true view.*

This type of MVC implementation can be a bit confusing, however our sample application will help clear things up. For now let's dive into Backbone models, views, and collections. In the following sections we'll go over how each part of Backbone gets implemented and the parts we'll be using to build our application.

Backbone models

As in any MVC pattern, the Model is critical, containing the data and logic, including properties, access controls, conversions, validations, and more. Keep in mind that we write models on a daily basis, and in fact we have created a number of models throughout this book (`MediaElement`, `Video`, `Audio`, and so on). Backbone models are similar to a boilerplate in that they provide utility methods that we would otherwise have to build ourselves.

Let's take the following code as an example:

```
function Song() {
    this.track = "Track Title";
    this.duration = 215;
    this.album = "Track Album";
};

Song.prototype.get = function(prop) {
    return this[prop] || undefined;
}

Song.prototype.set = function(prop, value) {
    this[prop] = value;

    return this;
}

var song = new Song();

song.get('album');
// "Track Album"

song.set('album', 'Sample Album');
// Song

song.get('album');
// "Sample Album"
```

In the preceding example, we have created a `Song` model, the same as in the previous section, that has several properties (`track`, `duration`, and `album`) and methods (`get` and `set`). From there we create an instance of `Song` and use the methods created to get and set the `album` property. This is great; however, we needed to create those methods manually. That is not what we want to do; we already know we need those methods, so we just want to focus on the data and extending it. This is where Backbone models come into play.

Let's analyze the following Model:

```
var SongModel = Backbone.Model.extend({
    'defaults': {
        'track': 'Track Title',
        'duration': 215,
        'album': 'Track Album'
    }
});

var song = new SongModel();

song.get('album');
// "Track Album"

song.set('album', 'Sample Album');
// SongModel

song.get('album');
// "Sample Album"
```

The preceding code shows how quickly we get off the ground writing our applications. Behind the scenes, Backbone is a namespace and has a model object attached to it. Then, using Underscore's `extend` method, we return a copy of `Backbone.Model`, that has merged default properties attached to it, to the variable `SongModel`. Then we do the same as earlier, using `get` and `set`, with the desired output in the comments.

As you can see it's pretty simple to get started using Backbone, especially if you just wanted a way to organize your data without building custom functionality for each and every application. Now let's look at views inside Backbone and how it can actually help us separate the data from our UI.

Backbone views

Backbone views are a bit different than models in such a way that they are more for convenience. If we look at the Backbone documentation and compare the *Views* and *Models* sections, we'll find that Views are a bit more bare bones, but again are useful in organizing our applications. To see why these are still useful, let's look at the following code:

```
var $section = $('section');

$section.on('click', 'a', doSomething);

function doSomething() {
    // we do something here
}
```

Typically, this is how we would cache our elements on the page and delegate events for particular user interactions. However, what if this could be done with less setup work? In the following code, we transform the preceding code into a typical Backbone view setup.

```
var SongView = Backbone.View.extend({
    'el': document.querySelector('section'),

    'events': {
        'click a': 'doSomething'
    },

    'doSomething': function(e){
        console.log($(e.currentTarget).attr('href'));
    }
});

var view = new SongView();
```

As you can see, Backbone takes care of the setup work for you. It caches the element selected and delegates the events for you behind the scenes. Literally, all you need to do on your end is the setup and quickly move on to the next step; now you'll notice that your development time decreases while your efficiency increases, and this is just the preliminary steps into Backbone. Now, the magic happens when we connect the Model and View together. To see this in action, take a look at the following code:

```
var SongModel = Backbone.Model.extend({
    'defaults': {
        'track': 'Track Title',
        'duration': 215,
        'album': 'Track Album'
    }
});

var song = new SongModel();

var SongView = Backbone.View.extend({
    'el': document.querySelector('section'),

    'events': {
        'click a': 'doSomething'
    },

    'initialize': function() {
        this.model.on('change:track', this.updateSongTitle, this);

        this.$el.$songTrack = this.$el.find('.song-track');
        this.$el.$songTrack.text(this.model.get('track'));
    },

    'doSomething': function(e){
        console.log($(e.currentTarget).attr('href'));
    },

    'updateSongTitle': function() {
        this.$el.$songTrack.text(this.model.get('track'));
    }
});

var view = new SongView({
    'model': song
});

song.set('track', 'Sample Track');
// The DOM Updates with the right value
```

In this code snippet we have finally connected a single model to one view. The way we have done this is by passing in the instance of the model into the instance of the view:

```
var view = new SongView({
    'model': song
});
```

When we do this, we associate the Model and the View. But we also need to do something with that Model, and usually we want to display the data associated with it. So in this example, we create an `initialize` method that gets called as a constructor. In this method, we use Backbone's built-in event system to track any changes associated with the Model's `track` property and call `updateSongTitle` accordingly. While we're at it, we change the context of the event handler by passing in `this` as the third parameter and then cache the element displaying the song's track.

In the end, when you change the instance of the song's `track` property, the DOM updates accordingly. We now have the basics we need to build our application. But let's take a look at Backbone collections to understand how keeping track of our data increases the efficiency of our application.

Backbone collections

Now, up until this point we have worked with a single Model, which is great but in most cases we work with sets of data. This is why Backbone collections exist, to manage an ordered set of models. Backbone collections also tie into Underscore's methods, allowing us to work with these sets easily and efficiently with no setup work.

Let's look at the following code:

```
var SongModel = Backbone.Model.extend({
    'defaults': {
        'track': 'Track Title',
        'duration': 215,
        'album': 'Track Album'
    }
});

var SongCollection = Backbone.Collection.extend({
    'model': SongModel
});

var SongView = Backbone.View.extend({
    'el': document.querySelector('section'),
    'events': {
        'click a': 'doSomething'
    },

    'initialize': function() {
        this.collection.on('change', this.updateDetected, this);
    },

    'doSomething': function(e){
        console.log($(e.currentTarget).attr('href'));
    },

    'updateDetected': function() {
        console.log("Update Detected");
    }
});

var collection = new SongCollection();

for (var i = 0; i < 100; i++) {
    collection.add(new SongModel());
}

var view = new SongView({
    'collection': collection
});
```

This sample code is very similar to the code produced in the previous section. The difference here is that we have created a `SongCollection` that takes models of type `SongModel`. Then we create an instance of this collection, add 100 models to it via our `for` loop, and finally attach the collection to our View.

Our View has also changed in such a way that we have attached the `change` event to our collection, and created a more general listener that gets called whenever a Model is updated within the collection. Therefore, when we execute the following code the View lets us know that something was updated:

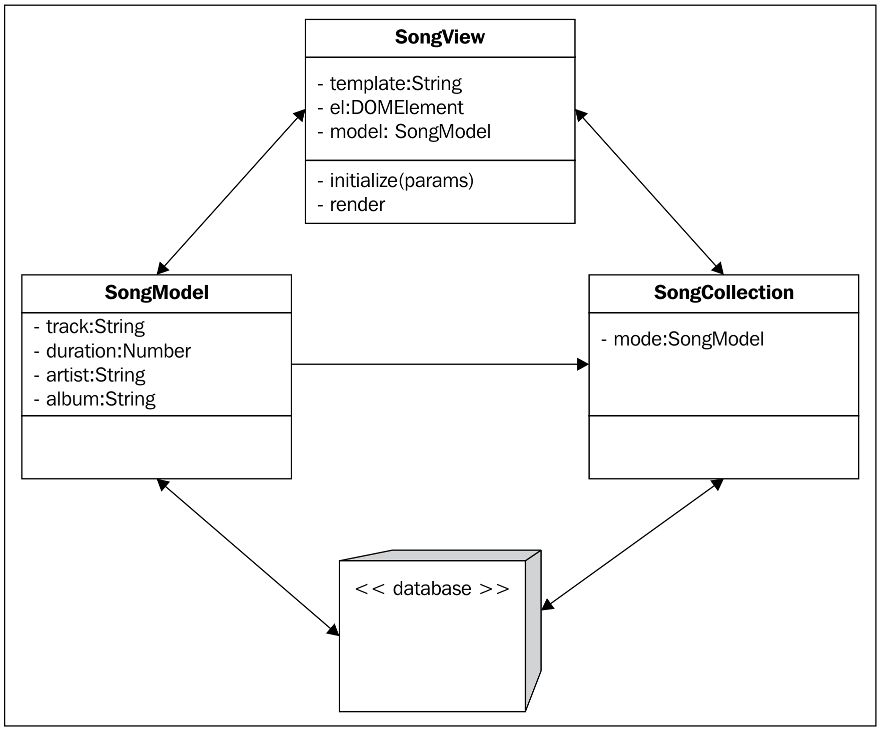
```
    collection.models[0].set('album', 'sample album');
    // "Update Detected"
```

Server-side interactions

It's not easy seeing how a Backbone application connects to the server, especially since we have so much going on in the frontend code. But, if YOU take a look at the documentation provided on the Backbone.js website (`http://backbonejs.org/#Sync`), we know that Models contain all the functionality for manipulating the data. In fact, Models connect to the database and can sync with it.

> *Backbone.sync is the function that Backbone calls every time it attempts to read or save a model to the server. By default, it uses (jQuery/Zepto).ajax to make a RESTful JSON request and returns a jqXHR. You can override it in order to use a different persistence strategy, such as WebSockets, XML transport, or Local Storage.*

But, Models aren't the only ones that can connect to the server. As the documentation continues to read, a model or collection can begin a sync request and interact with it accordingly. This is a bit different than a traditional MVC implementation, especially since collections and models can interact with the database. To better display Backbone's implementation of MVC, the provided image helps display the relationship between the different types of objects:



This is pretty much what we have created previously; a view, model, and controller. The implementation is slightly different, but we can see that there is a clear separation between the presentation layer and data because the view never directly interacts with the database. If this is a bit confusing, it's because it is and is another level of complexity that, when understood, will help guide you to coding Zen.

You are now fully prepared to create a one-page application using `Underscore`, `Backbone`, and `Zepto`. But, there is a problem. These libraries help speed up our development and increase efficiency, but don't actually provide a solid structure for our applications. This is what we tackle in our sample application. Next, we will discuss architecture, implementation, and optimization needed for one-page applications.

Our sample application

We have now been introduced to `Underscore.js` and `Backbone.js`, and have a good understanding of what these libraries provide and how they help with application development. However, we still need a way of structuring our applications, so that they are easily extended and most importantly, managed. So in this part of the chapter, we'll start building out a sample application that ties everything together and gets you going on building out one-page applications quickly.

Application architecture

Our sample application will do two things. One, it will allow us to see user information, such as a profile and dashboard. Two, it will have a playlist of songs that can be listened to using the HTML5 Audio media element. We can think of these requirements as almost two applications: a user application for managing user data and another application that manages the playback of media. However they will be related, in such a way that the user will have a playlist of songs related to them.

Basic sample architecture

Let's begin implementing the preceding architecture. First, we know that there will be two applications, similar to our `App` objects, so let's start by defining these:

- `js/Music/`
- `js/User/`
- Within the JavaScript (`js`) folder, we should create the preceding two folders: `Music` and `User`. These two folders will contain the code for the User and Music applications accordingly. To help manage our backbone files, we'll then create `models`, `views`, and `collections` folders for each.
- `js/Music/`
    - `views/`
    - `models/`
    - `collections/`
- `js/User/`
    - `views/`
    - `models/`
    - `collections/`

Great! Now we can start by creating a main JavaScript file that will contain the namespace for each application; each namespace being `User` and `Music` accordingly.

- `js/Music/`
    - `views/`
    - `models/`
    - `collections/`
    - `Music.js`
- `js/User/`
    - `views/`
    - `models/`
    - `collections/`
    - `User.js`

Now, most of our views are going to be the same with functionality that is very familiar. For example, there will be a global navigation bar that contains three links that will initiate a hide/show for each section, hiding the current section and showing the next. We don't necessarily want to code the same thing over and over again, so it would be nice to have a base view that our applications can inherit from. To do this, we'll create a folder called `views` within our `App` folder:

- `js/App/`
    - `views/`
    - `BaseView.js`

Okay, so this is basically our JavaScript framework for this sample application. Of course, there are other ways of setting this up, and perhaps they are even better—which is great. For our purpose, this fits the bill and helps demonstrate some structure within our applications. Now, let's start by looking at our markup.

Application markup

Let's open up the `index.html` file pertaining to this chapter; it should be located at `/singlepage/index.html`. Now, if we haven't already done so, let's start by updating the global navigation of the site, which we have done previously for other chapters. If you need reference material, look at the previous chapter's finished source code, provided with this book, and update the markup as necessary.

Our markup, when updated, should look like this:

```
<!DOCTYPE html>
<html class="no-js">
<head>
    <!-- Meta Tags and More Go Here -->

  <link rel="stylesheet" href="../css/normalize.css">
  <link rel="stylesheet" href="../css/main.css">
    <link rel="stylesheet" href="../css/singlepage.css">
  <script src="../js/vendor/modernizr-2.6.1.min.js"></script>
</head>
  <body>
    <!-- Add your site or application content here -->
        <div class="site-wrapper">
            <header>
                <hgroup>
                    <h1>iPhone Web Application Development</h1>
                    <h2>Single Page Applications</h2>
                </hgroup>
```

```
                    <nav>
                        <select>
                            <!-- Options Go Here -->
                        </select>
                    </nav>
                </header>
                <footer>
                    <p>iPhone Web Application Development &copy; 2013</p>
                </footer>
            </div>

            <!-- BEGIN: LIBRARIES / UTILITIES-->
        <script src="../js/vendor/zepto.min.js"></script>
            <script src="../js/vendor/underscore-1.4.3.js"></script>
            <script src="../js/vendor/backbone-0.9.10.js"></script>
        <script src="../js/helper.js"></script>
            <!-- END: LIBRARIES / UTILITIES-->
            <!-- BEGIN: FRAMEWORK -->
            <script src="../js/App/App.js"></script>
            <script src="../js/App/App.Nav.js"></script>
            <!-- END: FRAMEWORK -->
      </body>
    </html>
```

Now, let's begin modifying this code to fit our application. First, let's start by adding in a `div` after the header with a class of `content`:

```
     <div class="site-wrapper">
        <header>
            <hgroup>
                <h1>iPhone Web Application Development</h1>
                <h2>Single Page Applications</h2>
            </hgroup>
            <nav>
                <select>
                    <!-- Options Go Here -->
                </select>
            </nav>
        </header>
        <div class="content"></div>
        <footer>
            <p>iPhone Web Application Development &copy; 2013</p>
        </footer>
    </div>
```

When we're done doing that let's modify the scripts, by including our entire application that we created previously. This means we are including the `Music` and `User` application scripts, and the `BaseView`. The scripts section of our markup should then look like this:

```
<!-- BEGIN: LIBRARIES / UTILITIES-->
<script src="../js/vendor/zepto.min.js"></script>
<script src="../js/vendor/underscore-1.4.3.js"></script>
<script src="../js/vendor/backbone-0.9.10.js"></script>
<script src="../js/helper.js"></script>
<!-- END: LIBRARIES / UTILITIES-->
<!-- BEGIN: FRAMEWORK -->
<script src="../js/App/App.js"></script>
<script src="../js/App/App.Nav.js"></script>
<script src="../js/App/views/BaseView.js"></script>
<!-- END: FRAMEWORK -->
<!-- BEGIN: MUSIC PLAYLIST APPLICATION -->
<script src="../js/Music/Music.js"></script>
<script src="../js/Music/models/SongModel.js"></script>
<script src="../js/Music/collections/SongCollection.js"></script>
<script src="../js/Music/views/SongView.js"></script>
<script src="../js/Music/views/PlayListView.js"></script>
<script src="../js/Music/views/AudioPlayerView.js"></script>
<!-- END: MUSIC PLAYLIST APPLICATION -->
<!-- BEGIN: USER APPLICATION -->
<script src="../js/User/User.js"></script>
<script src="../js/User/models/UserModel.js"></script>
<script src="../js/User/views/DashboardView.js"></script>
<script src="../js/User/views/ProfileView.js"></script>
<!-- END: USER APPLICATION -->
<script src="../js/main.js"></script>
<script> Backbone.history.start(); </script>
<!-- END: BACKBONE APPLICATION -->
```

Note that we have started the Backbone history API. Although we have not discussed this thoroughly, this is essential for keeping state in our applications. The details for the implementation of the History API in Backbone are beyond the scope of this book, but it is highly encouraged for those of you looking to harness the power of offline storage using Backbone. For now, keep in mind that this is essential for routing.

Creating templates

Now our markup is close to completion, but we are still left with what the rest of the application will be made of; and that is where templating will come in. The next step is to include the templates that will be required for our application, including the audio player view, playlist, song, dashboard, profile, and shared navigation views. So how does one specify a template on a static HTML page? Like this:

```
<script type="tmpl/Music" id="tmpl-audioplayer-view">
    <section class="view-audioplayer">
        <header>
            <h1>Audio Player</h1>
        </header>
        <div class="audio-container">
            <audio preload controls>
                <source src="../assets/<%= file %>" type='audio/mpeg;
codecs="mp3"'/>
                <p>Audio is not supported in your browser.</p>
            </audio>
        </div>
    </section>
</script>
```

You may be wondering why this does not cause any errors in validation or code execution within the browser. Well to help clear things up, the `type` attribute of our `script` tag is a non-supported MIME type and because of this, the browser ignores all content within this `script` block (`http://www.whatwg.org/specs/web-apps/current-work/multipage/scripting-1.html#script-processing-prepare`). Because the code within the block doesn't get executed, we can include our HTML templates to be used later on. Keep in mind that we have attached an ID that we can use to target this element using Zepto. And also note the source for the audio element, specifically `<%= file %>`. This will be used by Underscore's `template` method to interpolate the data passed into the template itself. We'll get to that soon, but for now know that this is how we can set up our templates.

Okay, we now know how to create templates, so let's implement the following templates right before the inclusion of our applications' scripts. We can include the preceding template for the audio player, and after that we can include the following template:

```
<!-- Playlist View -->
<script type="tmpl/Music" id="tmpl-playlist-view">
    <section class="view-playlist">
        <header>
            <h1><%= name + "'s" %> Playlist</h1>
            <% print(_.template($('#tmpl-user-nav').html(), {})); %>
        </header>
        <ul></ul>
    </div>
</script>
```

In the playlist view template we have some pretty interesting stuff going on. Take a look at the code after the `h1` tag. We see here the `template` method from Underscore's library; it is accepting one parameter that will be an HTML string of the template `#tmpl-user-nav`, which we have not defined yet, and the second parameter is an empty object. This example is showcasing the use of a template within a template, kind of Inception-like but hopefully not too scary. Remember that we mentioned there would be a global navigation included in our applications; the preceding method helps us code once—keeping our code clean, manageable, and efficient.

Now, our playlist still doesn't contain a list of songs. That is because it will be dynamic and based on a dataset of songs; this is why we have an empty unordered list within the playlist view. But how will our songs look? Traditionally, we would just create a list (`li`) element in our JavaScript, but with templates we no longer need to do that—we can keep our markup outside our logic:

```
<!-- Individual Song View -->
<script type="tmpl/Music" id="tmpl-song-view">
    <li class="view-song">
        <strong><%= track %></strong>
        <em><%= artist %></em>
    </li>
</script>
```

See how easy it is now to keep your markup outside of your scripts? In this template, we are following the same basic principles: define a script block containing markup and create the markup that will then be interpolated to include the data we want. In this case, we want the track and artist to be output into their own elements.

Now let's create the user's dashboard:

```
<script type="tmpl/User" id="tmpl-user-dashboard">
    <section class="view-dashboard">
        <header>
            <h1><%= name + "'s" %> Dashboard</h1>
            <% print(_.template($('#tmpl-user-nav').html(), {})); %>
        </header>
    </section>
</script>
```

Again, it's the same as before. In fact we are repeating ourselves by using the same method in the playlist view to display the global navigation. By now, you have noticed that each template receives a specific ID, and that, for convention, we have defined the type of each `script` block according to its application, for example `tmpl/User` for the User application and `tmpl/Music` for the Music application.

Let's now take a look at the profile view that combines both the preceding methods:

```
<script type="tmpl/User" id="tmpl-user-profile">
    <section class="view-profile">
        <header>
            <h1><%= name + "'s" %> Profile</h1>
            <% print(_.template($('#tmpl-user-nav').html(), {})); %>
        </header>
        <dl>
            <dt>Bio</dt>
            <dd><%= bio %></dd>
            <dt>Age</dt>
            <dd><%= age %></dd>
            <dt>Birthdate</dt>
            <dd><%= birthdate.getMonth() + 1 %>/<%= birthdate.
getDate() %>/<%= birthdate.getFullYear() %></dd>
        </dl>
    </section>
</script>
```

In this view, we have the global navigation being printed out and data being interpolated. As you can see, there is no limit to what you can do in templates. But it can also be something as simple as the global navigation for our applications:

```
<script type="tmpl/User" id="tmpl-user-nav">
    <a href="#dashboard">Dashboard</a>
    <a href="#profile">Profile</a>
    <a href="#playlist">Playlist</a>
</script>
```

In this last example, there is nothing complicated happening, it is essentially the global navigation we have been expecting and it turns out—it's just markup. Now, you may be wondering why not create all of this in the DOM, hide it, and then populate it with the information you need using the built-in selector engines in `Zepto` or `jQuery`. And honestly, that is a great question to ask. But there is one main reason, performance. It is expensive to use these engines, even the built-in methods `querySelector` and `querySelectorAll`. We do not want to touch the DOM, because it is a heavy operation, especially for large-scale applications handling large data sets. And ultimately, it's messy to do DOM operations just for data population or storage. Just don't do it, it is a nasty habit to use the DOM for data and not a best practice throughout the industry.

Our templates are complete, and that concludes the markup of our application. Now we move to the fun part, our scripts. The next part will be quite complex and pretty challenging, but I promise that when we're done, you'll be a pro at one-page applications and ready to create your own quickly. The first time around is always rough, but stick through it and you'll reap the rewards.

Application scripts

In this section we'll go through the scripts needed to make our application work. We'll start with reviewing the `BaseView`, the view containing shared functionality in inherited views (`PlayListView`, `ProfileView`, and `DashboardView`). From there we'll create our Music and User applications, each having their relative models, views, and collections.

The BaseView

Let's start looking at our scripts, beginning with the BaseView file we created under the App namespace (js/App/views/BaseView.js). In this file, we'll create the BaseView class that will extend Backbone's general View class. The BaseView will then look like this:

```
(function(window, document, $, Backbone, _){

  var BaseView = Backbone.View.extend({

  });

  // Expose the User Object
  window.App.BaseView = BaseView;

}(window, document, Zepto, Backbone, _));
```

This class follows the exact same pattern as the rest of the JavaScript we have written in previous chapters, the only differences here include the inclusion of Backbone and Undescore and the way we expose the BaseView class using window.App.BaseView = BaseView.

Now, bear with me here. We will be creating several methods that will be included in any object extending the BaseView class. These methods will include show, hide, onProfileClick, onPlaylistClick, onDashboardClick, and onEditClick. As you may have guessed, some of these methods will be event handlers that will navigate to certain parts of our application. Check out the following code for the implementation:

```
(function(window, document, $, Backbone, _){

  var BaseView = Backbone.View.extend({
    'hide': function() {
      this.$template.hide();
    },

    'show': function() {
      this.$template.show();
    },
    'onProfileClick': function(e) {
      e.preventDefault();

      User.navigate('profile/' + this.model.get('username'), {
'trigger': true });
    },

    'onPlaylistClick': function(e) {
      e.preventDefault();

      Music.navigate('playlist', { 'trigger': true });
    },

    'onDashboardClick': function(e) {
      e.preventDefault();

      User.navigate('dashboard', { 'trigger': true });
    },

    'onEditClick': function() {
      console.log('onEditClick');
    }
  });

  // Expose the User Object
  window.App.BaseView = BaseView;

}(window, document, Zepto, Backbone, _));
```

Now, you may notice that the objects written here have not been created, such as the `$template`, `User`, and `Music` objects. We'll return to this in a few steps, but keep in mind that `this.$template` will refer to the instance extending `BaseView`, and that the User and Music objects will be routers that will use the built-in backbone method called `navigate` to change our application's location in the URL and to store the history of the user's interaction. To better understand how this class, `BaseView`, gets used, let's start creating the code for the `Music` object in `Music.js` (`js/Music/Music.js`).

Music application

Now let's jump into creating the first part of our application, the music application. Both the music and user applications are separated to add a higher level of maintainability and reuse. Starting with the music application, we'll create the appropriate router, collection, model, and view.

The router

Our music application begins with the `Music` class defined in our `Music.js` file located at `js/Music/`. In this file, we will extend Backbone's `Router` class, containing the routes for our music application, a sample data object for use with our models and collections, and finally an event handler for when the playlist is requested. First, let's start out with defining the class:

```
(function(window, document, $, Backbone, _){

  var Music = Backbone.Router.extend({
    // Application Routes
    'routes': {
      'playlist': 'setupPlaylist',
      'playlist/:track': 'setupPlaylist'
    }
  });

  // Expose the Music Object
  window.Music = new Music();

}(window, document, Zepto, Backbone, _));
```

Following the pattern we have established with our `BaseView` class, we extend the `Router` class in `Backbone` and define some default routes. The two routes include a general playlist route and an alternative route containing the playlist and the track number. Both routes, when called, will invoke the `setupPlaylist` method that we will define next:

```
'setupPlaylist': function(track){
  if (!this.songCollection) {
    // Create song collection on the instance of Music
    this.songCollection = new this.SongCollection(this.songs);
  }

  if (!this.playListView) {
    // Create song list view on the instance of Music
    this.playListView = new this.PlayListView({
      'el': document.querySelector('.content'),
      'collection': this.songCollection,
      'model': new User.UserModel()
    });
  } else {
    this.playListView.show();
    this.playListView.audioPlayerView.show();
  }

  if (track) {
    this.playListView.updateTrack(track);
  }
}
```

Don't worry if this code is a bit intimidating, it's actually pretty simple. First, we check if a `songCollection` object has been initialized with the instance of `Music`. If it hasn't then we create one, using a sample data object of songs—which we haven't created yet. Next, we do the same thing, checking if the `playListView` object has been created; if not we move on to creating it. Otherwise, we just show the playlist and the audio player associated with it. Lastly, we check if a track number has been passed in (related to the second route we created); if there is a track number, we update the `playListView` to reflect the track selected.

Let's focus on the initialization of the `playListView`:

```
this.playListView = new this.PlayListView({
  'el': document.querySelector('.content'),
  'collection': this.songCollection,
  'model': new User.UserModel()
});
```

Although we haven't officially created the `PlayListView` class, we can review how it gets initialized. In this case, we are attaching a property of `playListView` on the instance of `Music` with `this.playListView`. This property is going to be an instance of the `PlayListView` (`new PlayListView({})`). This new instance of `PlayListView` will accept a plain object that contains three properties: an element defined as `el`, a collection, and an instance of a `UserModel`, which has not been defined.

The last thing we need to do here is include an `initialize` method that will create a sample data object (`this.songs`), and listen for when the playlist route is called. When we call the playlist route, or navigate to it, we want both the profile and dashboard to hide; we'll do this manually within the `routes` listener:

```
'initialize': function() {
  this.songs = [{
      'duration': 251,
      'artist': 'Sample Artist',
      'added': new Date(),
      'track': 'Sample Track Title',
      'album': 'Sample Track Album'
    }, {
      'duration': 110,
      'artist': 'Sample Artist',
      'added': new Date(),
      'track': 'Sample Track Title',
      'album': 'Sample Track Album'
    }, {
      'duration': 228,
      'artist': 'Sample Artist',
      'added': new Date(),
      'track': 'Sample Track Title',
      'album': 'Sample Track Album'
    }
  ];

  this.on('route:setupPlaylist', function() {
    // This should be more dynamic, but fits our needs now
    // ---
    if (User.profileView) {
      User.profileView.hide();
    }

    if (User.dashboardView) {
      User.dashboardView.hide();
    }
    // ---
  });
},
```

Okay, so we have created the `initialize` method here, which gets called when an instance of `Music` is created. This is good, because in this method we take care of any setup work, such as creating the sample data object. The sample data object is an array of objects that will then be transformed into models by the `SongCollection` class:

```
    'setupPlaylist': function(track){
      if (!this.songCollection) {
        // Create song collection on the instance of Music
        this.songCollection = new this.SongCollection(this.songs);
      }
      // Some code defined after
    }
```

Looks familiar huh? Now we're tying up loose ends. We haven't created the `SongCollection` class yet, but as Backbone's documentation states, if an array is passed into a collection, it is automatically turned into the model specified in the collection (to be described in future steps).

The last thing this `initialize` method does, is define a listener on the route for playlist (`this.on('route:setupPlaylist', function() {});`). The event handler then hides the profile and dashboard if they have been created. Also, note that we have specified the route using `route:setupPlaylist`, but we could have easily listened to any route by just using `route`.

So I know this is a lot to digest, but we'll now connect the dots from this `Music` class, starting with collections then moving on to models and finally views. This class is the foundation for everything else that needs to get built in order to have a fully functional music application and provide the blueprint of our development.

The collection

The collection for our music application is simple. Following the basic template of what we've done previously, we will create a closure containing the `SongCollection` class. Then we will define the type of model the `SongCollection` should keep. And lastly, we'll expose the class to our `Music` object.

When we are done implementing these requirements, our class looks like this:

```
(function(window, document, $, Backbone, _){

  var SongCollection = Backbone.Collection.extend({
    'model': window.Music.SongModel
  });

  window.Music.SongCollection = SongCollection;

}(window, document, Zepto, Backbone, _));
```

See how simple it was? Now we know that this collection only keeps track of models that are of type `SongModel` and that if passed an array, it will transform the contained objects into `SongModel` types. This is all this class will do for now. Of course, you are welcome to extend it and play around with the several methods, such as the comparator, this class can harness; but for now, this is all we need.

The model

Our `SongModel` will describe the type of data we're trying to keep track of. This model will also contain a single method that will take a property of duration, in seconds, and return it in minutes. Of course, we have the option to prepare our model, when it has been initialized, but for now we'll keep it simple.

The `SongModel`, when written, will look like this:

```
(function(window, document, $, Backbone, _){

  var SongModel = Backbone.Model.extend({
    'defaults': {
      // in seconds
      'duration': 0,
      'artist': '',
      'added': 0,
      'track': '',
      'album': ''
    },

    'initialize': function() {

    },

    'getDurationInMinutes': function() {
      var duration = this.get('duration');

      if (duration === 0) {
        return false;
      }

      return this.get('duration') / 60;
    }
  });

  window.Music.SongModel = SongModel;

}(window, document, Zepto, Backbone, _));
```

From the preceding code, we can infer that the `SongModel` will have the attributes `duration`, `artist`, `added`, `track`, and `album`. Each will have a default value of an empty `String` or `0`. We can also notice that each model will have a method named `getDurationInMinutes` that can be invoked and would return the duration of that model in minutes. Again, the `SongModel` class follows the same basic architecture and best practices, returning itself to the `Music` object. Finally, we are ready to look at the views for this music application.

The view(s)

In this part, we're going to review three separate views, including the play list, song, and audio player views. Each view renders an individual part of the music application, except for playlist, which renders both the audio player and each individual song. So, let's start with the playlist view.

The playlist view

We want the play list view to do a couple of things but we'll take it one step at a time. First, let's create the `PlayListView` class, which will extend our already created `BaseView` class.

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    // Code goes here
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

Next, we want the `PlayListView` class to reference the proper template.

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    'template': _.template($('#tmpl-playlist-view').html())
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

By including a template as a property we can easily reference it using `this.template`. Keep in mind that we have not processed the template at this stage, we have simply used Underscore's `template` method to retrieve the markup. Next, we want to define an event listener for when the user clicks on a song.

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    'template': _.template($('#tmpl-playlist-view').html()),

    'events': {
      'click .view-song': 'onSongClicked'
    }
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

In this step we are telling the view to delegate all the events we've created to the view's element. In this event object, we are listening for a click event on an element with the class of `.view-song`. When this element is clicked, we want to invoke the `onSongClicked` event handler. Let's define this event handler next.

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    'template': _.template($('#tmpl-playlist-view').html()),

    'events': {
      'click .view-song': 'onSongClicked'
    },

    'onSongClicked': function(e) {
      var $target = $(e.currentTarget);

      this.$el.find('.active').removeClass('active');

      $target.addClass('active');

      Music.navigate('playlist/' + ($target.index() + 1), { 'trigger':
true });
    }
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

The event handler defined in the preceding code toggles the active class and then tells the `Music` router to navigate to the playlist route, telling it to trigger the route event and pass in the track's index. By doing this, our route is called, passed a track, and the playlist updates. However, we still do not have the `updateTrack` method defined. Let's include the following method in our class:

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    'template': _.template($('#tmpl-playlist-view').html()),

    'events': {
      'click .view-song': 'onSongClicked'
    },

    'onSongClicked': function(e) {
      var $target = $(e.currentTarget);

      this.$el.find('.active').removeClass('active');

      $target.addClass('active');

      Music.navigate('playlist/' + ($target.index() + 1), { 'trigger':
true });
    },

    'updateTrack': function(track) {
      this.audioPlayerView.render(track);

      this.setActiveSong(track || 1);
    }
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

Now we have the `updateTrack` method, and this is essentially telling the audio player's view to render the track it has received. Unfortunately our code is still not ready to be run because we haven't created this method. Also, the following method, `setActiveSong`, is not defined, so we need to do that now:

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    'template': _.template($('#tmpl-playlist-view').html()),

    'events': {
      'click .view-song': 'onSongClicked'
    },

    'onSongClicked': function(e) {
      var $target = $(e.currentTarget);

      this.$el.find('.active').removeClass('active');

      $target.addClass('active');

      Music.navigate('playlist/' + ($target.index() + 1), { 'trigger':
true });
    },

    'setActiveSong': function(track) {
      this.$el.find('.active').removeClass('active');

      this.$el.find('.view-song').eq(track - 1).addClass('active');

      return this;
    },

    'updateTrack': function(track) {
      this.audioPlayerView.render(track);

      this.setActiveSong(track || 1);
    }
  });

  // Expose the PlayListView Class
  window.Music.PlayListView = PlayListView;

}(window, document, Zepto, Backbone, _));
```

Our `setActiveSong` method is now created and essentially toggles the active class depending on the URL's track number. We could probably extrapolate and create a general toggle here for songs, but for now this meets the criteria. But we're not done yet, we still need to initialize this class and render it appropriately. Let's take a look at what the class now needs:

```
(function(window, document, $, Backbone, _){

  var PlayListView = App.BaseView.extend({
    // code before

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

    'render': function() {
      var i = 0,
        view,
        that = this;

      // Create the template
      this.$template = $(this.template(this.model.attributes));

      // Append the template
      this.$el.append(this.$template);

      // Create the audio player
      if(!this.audioPlayerView) {
        this.audioPlayerView = new Music.AudioPlayerView({
                      'el': this.el.querySelector('.view-
playlist'),
                      'model': new User.UserModel()
                    });
      }

      this.collection.each(function(element, index, list){
        var view  = new Music.SongView({
          'el': that.$template.find('ul'),
          'model': element
        });
      });
```

```
        return this;
      },

      // code after
    });

    // Expose the PlayListView Class
    window.Music.PlayListView = PlayListView;

  }(window, document, Zepto, Backbone, _));
```

The preceding code completes the class, but before we move on let's take a look at what's going on here. First, we have defined an `initialize` method. This method will be invoked after the construction of an instance of this class, therefore the render method will also be called. Typically, in Backbone, the `render` method does exactly what the function is called—renders the view.

The `render` method defined does a few things; first it compiles our template using the model that was passed in. Earlier we saw the following code:

```
// Create song list view on the instance of Music
this.playListView = new this.PlayListView({
  'el': document.querySelector('.content'),
  'collection': this.songCollection,
  'model': new User.UserModel()
});
```

As we can see, a new `UserModel` is created and passed into the `PlayListView`, and this instance is used to populate the playlist's template. Once the compilation is done, we attach the compiled template using Zepto's `append` method. What is it attaching it to, you ask? Well, the above initialization of this class is looking for an element with the class of `content`, which we defined after the header element on our page. Therefore, the `PlayListView` is going to attach itself to this `div` of class `content`.

When the template is done being attached, we check if the audio player view has been created. If not, then we create it:

```
if(!this.audioPlayerView) {
  this.audioPlayerView = new Music.AudioPlayerView({
    'el': this.el.querySelector('.view-playlist'),
    'model': new User.UserModel()
  });
}
```

Finally, once the check for the audio player view is made we get to the fun stuff. In this last part we loop through the collection that was sent over, which is an instance of `SongCollection` and the same data created in `Music.js`. As we loop through each model in the collection we create an instance of `SongView` each time, passing in the compiled template's unordered list element and passing it the current model.

Now, if that didn't blow your mind, I'm not sure what could. Either way, I hope you're still up for the challenge because we have two more views we need to look at: the `AudioPlayerView` and the `SongView`. Don't lose hope though, we have passed the greatest challenge and are ready to ride the wave.

The audio player view

We're going to build our `AudioPlayerView` next. This view needs to pick up on the template we created earlier, populating it with the track number and loading it if we directly access the URL, for example `/#playlist/2`. We also need to override a method on the extended `BaseView`, the method that needs to be overwritten is `onDashboardClick`. This is because it requires we hide the playlist and then navigate to the dashboard. So at the very basic level, this class will look like this:

```
(function(window, document, $, Backbone, _){

  var AudioPlayerView = App.BaseView.extend({
    'template': _.template($('#tmpl-audioplayer-view').html()),

    'events': {
      'click a[href="#dashboard"]': 'onDashboardClick'
    },

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

    'render': function(file){
      // Put our rendering code here
    },

    'onDashboardClick': function() {
      this.hide();
      Music.playListView.hide();

      User.navigate('/dashboard', { 'trigger': true });
    }
  });

  window.Music.AudioPlayerView = AudioPlayerView;

}(window, document, Zepto, Backbone, _));
```

As we can see, all the requirements listed in the previous paragraph have been met with this base class for the `AudioPlayerView`. However, we need to render out this view, populating it with the data provided by the URL. To do this, we need to write our `render` method like so:

```
'render': function(file){
  var audioElement;

  if (file) {
    audioElement = this.$el.find('audio')[0];

    // Must be made on the audio element itself
    audioElement.src = '../assets/' + 'sample' + (file || 1) + '.mp3';
    audioElement.load();
    audioElement.play();

    return this;
  }

  this.$template = $(this.template({ 'file': 'sample' + (file || 1) +
'.mp3', 'name': this.model.get('name') }));
  this.$template.find('audio')[0].volume = 0.5;

  this.$el.find('header').after(this.$template);

  return this;
},
```

Similar to the previous `render` method we wrote for the playlist view, the `render` method checks if a file, a number, has been passed in. If it has we populate the audio element from our template with what was passed in. Next, we compile our template, then set the volume to `0.5` and attach the player right after the header of `PlayListView`. If we review how we initialized this class, we'll notice that the audio player view delegates to the playlist view element (inside `PlayListView`):

```
this.audioPlayerView = new Music.AudioPlayerView({
  'el': this.el.querySelector('.view-playlist'),
  'model': new User.UserModel()
});
```

The song view

The last part of our music application is the `SongView`. So let's quickly review the requirements for this and see its implementation. For this view, we again want to set our template. When we initialize this view we want to attach an event handler on the model passed in, so if the model is ever updated, the view renders automatically with the update. The `render` method of this view should essentially compile the template with the model's attributes and then attach itself to the element set for this view.

When we are done implementing the preceding requirements, the view should look somewhat like this:

```
(function(window, document, $, Backbone, _){

  var SongView = App.BaseView.extend({
    'template': _.template($('#tmpl-song-view').html()),

    'initialize': function() {
      // Listen to when a change happens on the model assigned this
view
      this.listenTo(this.model, 'change', this.render);

      this.render();
    },

    'render': function() {
      this.$el.append(this.template(this.model.attributes));

      return this;
    }
  });

  // Expose the SongView
  window.Music.SongView = SongView;

}(window, document, Zepto, Backbone, _));
```

As we can see, we follow the standards set in previous view implementations. The only difference is the addition of the event listener on the model's change event. Let's remind ourselves of how this view gets initialized in `PlayListView`:

```
this.collection.each(function(element, index, list){
  var view  = new Music.SongView({
    'el': that.$template.find('ul'),
    'model': element
  });
});
```

Now we fully understand how the music application works. At this point, our page could function just with this implementation; however, I don't recommend it since errors will come up because we haven't created the user application. But we now know that our routes define the actions in our application, views are the presentation layer that implements models and collections. Models are the heart of our application, containing all the data we need in a manageable fashion. And finally, collections help us manage larger data sets of our models and because we can pass these into the view, the view itself can manage the render of this data, which is ideal for large-scale applications.

The next step of this process is to develop the user application, but hopefully things will be a bit easier. As we did in the last part, we'll start out with the router and work our way to the collection, model, and views.

User application

The user application will follow the same flow as the music application we created. Again, we'll cover the implementation of a router, model, and view. When we are done with this section we'll have subapplications that each run separately and increases the maintainability and efficiency of our one-page application.

The router

Our router for the user application will be very similar to the music application. We'll define the routes for the dashboard and profile. We will also take the time to create the homepage route of our one-page application. The route will contain the appropriate methods for setting up the dashboard and profile. It will also contain the homepage method, which will call the dashboard route. In the `initialize` method of the router we will listen to these routes and hide other views.

```
(function(window, document, $, Backbone, _){

  var User = Backbone.Router.extend({
    // Application Routes
    'routes': {
      '': 'home',
      'dashboard': 'setupDashboard',
      'profile/:user': 'setupProfile'
    },

    'initialize': function() {

    },
    'home': function() {

    },

    'setupDashboard': function() {

    },

    'setupProfile': function(name) {

    }
  });

  // Expose the User Object
  window.User = new User();

}(window, document, Zepto, Backbone, _));
```

In the preceding code, we follow our standards, and create the base template for our user application. Next, let's take a look at what the `initialize` method will contain:

```
    'initialize': function() {
      var that = this;

      this.on('route:setupDashboard route:setupProfile', function(){
        if(Music.playListView) {
          Music.playListView.hide();
        }
      });

      this.on('route:setupDashboard', function(){
        if (that.profileView) {
          that.profileView.hide();
        }
      });

      this.on('route:setupProfile', function(){
        if (that.dashboardView) {
          that.dashboardView.hide();
        }
      });
    },
```

The `initialize` method of our route meets the requirements we've listed by creating the event listeners for the routes we created. Each listener hides the sections we don't want to see, but how do we see the actual part of the application we want? Well, that's where the `setup` methods come in.

```
'setupDashboard': function() {
  if (!this.dashboardView) {
    this.dashboardView = new this.DashboardView({
                'model': this.model = new this.UserModel(),
                'el': document.querySelector('.content')
              });
    this.setupDashboard();
    return;
  }

  this.dashboardView.show();
},
'setupProfile': function(name) {
  if (!this.profileView) {
    this.profileView = new this.ProfileView({
                'model': this.model = new this.UserModel(),
                'el': document.querySelector('.content')
              });
    return;
  }

  this.profileView.show();
}
```

These methods are pretty much the same. They do a check for whether the view has been created on the instance of the router (for example `this.dashboardView` and `this.profileView`), if it has we just show that view. However, if the view has not been created, we initialize the appropriate view and then call that `setup` method again (recursive), so that we can show it since the view now exists.

> You may have noticed that we are creating a new `UserModel` that gets passed to many of our views. This is okay for now, since we want to test the meat of our application. But theoretically, one `UserModel` would be initialized and maintained throughout the application. Sounds like something you can tackle once you complete this chapter!

The last thing we need to do is include the homepage method for our application:

```
'home': function() {
  this.navigate('dashboard', { 'trigger': true });
},
```

Automatically, when you visit `/singlepage/index.html`, this route will be called. As defined in the `Backbone.js` library's documentation, an empty route refers to the home state of the application. Although we can define the `setupDashboard` method to be the callback, this is to illustrate that we can go from one route to another immediately when needed. Perhaps we could do some preprocessing here, or even create that single `UserModel` noted earlier?

The collection

Because we are only handling one user in this application, there won't be any need for creating a collection. Phew! You thought this would get a bit more difficult eh? Well, don't get your hopes up; we still have the model and view(s) to consider.

The model

As with any model in Backbone, we are just describing the data that will be handled throughout our application. This is no different for our `UserModel`, which will contain the default attributes of an instance and set the name of the person by combining the `first_name` and `last_name` attributes when it is initialized.

To meet these requirements, our `UserModel` will be defined like so:

```
(function(window, document, $, Backbone, _){

  var UserModel = Backbone.Model.extend({
    'defaults': {
      // in seconds
      'first_name': 'John',
      'last_name': 'Doe',
      'bio': 'Sample bio data',
      'age': 26,
      'birthdate': new Date(1987, 0, 2),
      'username': 'doe'
    },

    'initialize': function() {
      this.attributes.name = this.get('first_name') + ' ' + this.
get('last_name');
    }
  });

  window.User.UserModel = UserModel;

}(window, document, Zepto, Backbone, _));
```

That's really it for our model. We are just defining default values for our user and setting the name when an instance is created. Now we'll take a look at our `DashboardView` and `ProfileView`—the last two pieces for this application.

The view(s)

The user application will contain two views, including the `DashboardView` and `ProfileView`. As we have already established, each view extends the `BaseView` we created earlier. We'll need to make some changes in order to fit our experience, but overall this will be very similar to our music application view's implementation.

The dashboard view

As with our previously defined views, `DashboardView` will contain the template that will be used for the display of our dashboard, contain the events pertaining to this view and then render the template. What you'll notice here is that our events will use the event handlers defined in `BaseView`, because the `BaseView` event handlers meet the basic requirements of navigating to another view while the route listeners handle the hide functionality.

```
(function(window, document, $, Backbone, _){

  var DashboardView = App.BaseView.extend({
    'template': _.template($('#tmpl-user-dashboard').html()),

    'events': {
      'click a[href="#profile"]': 'onProfileClick',
      'click a[href="#playlist"]': 'onPlaylistClick'
    },

    'initialize': function() {

      this.render();
    },

    'render': function() {
      if (!this.$template) {
        this.$template = $(this.template(this.model.attributes));

        this.$el.prepend(this.$template);
      }

      return this;
    }
  });

  window.User.DashboardView = DashboardView;

}(window, document, Zepto, Backbone, _));
```

The code for this view is fairly simple; we've seen this pattern before and are repeating it here. Because we've defined the event handlers in `BaseView`, we don't need to redefine them here. As for the `render` method, it checks for the creation of the template, and if it exists, it populates the template with the user's data, which we passed in when creating the instance of `DashboardView` in `User.js`.

This is all we need to do for the dashboard view; like I promised, it's fairly easy once the general setup is finished. Next let's take a look at the final part of our application: the profile view.

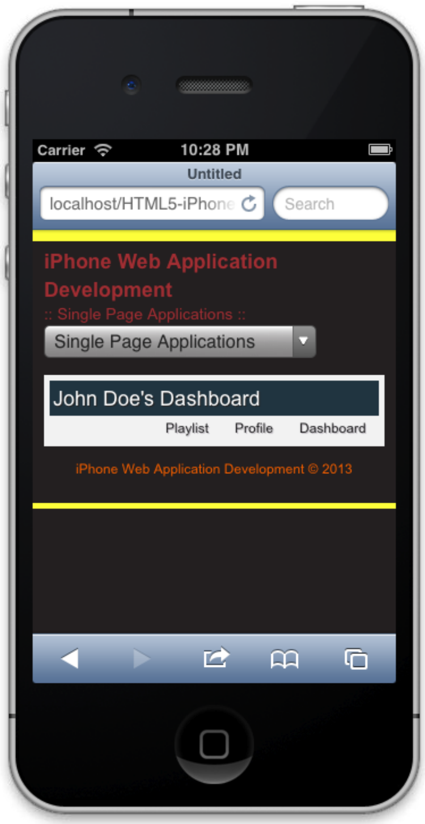
The profile view

Our profile view will be exactly the same as the dashboard view in that we have a template, some events, and a `render` method. And just like before, we won't need to define the event handlers because the basic requirement of hiding a view is being taken care of by the `BaseView` we created at the beginning of this process.

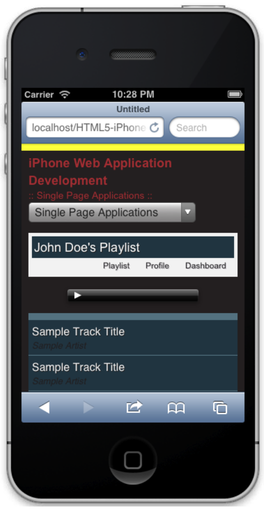
```
(function(window, document, $, Backbone, _){

  var ProfileView = App.BaseView.extend({
    'template': _.template($('#tmpl-user-profile').html()),

    'events': {
      'click a[href="#dashboard"]': 'onDashboardClick',
      'click a[href="#edit"]': 'onEditClick'
    },

    'initialize': function() {

      this.render();
    },

    'render': function() {
      if (!this.$template) {
        this.$template = $(this.template(this.model.attributes));

        this.$el.prepend(this.$template);
      }

      return this;
    }
  });

  window.User.ProfileView = ProfileView;

}(window, document, Zepto, Backbone, _));
```

And that wraps everything up. If we run the page now, we get a fully accessible application that has the dashboard view as its default view. You can then interact with the application by navigating to the profile and playlist views. As you do so, the application changes the URL and keeps history of your activity, letting you go back and forward easily. Pretty neat, huh? Here are a couple of screenshots to showcase what the final application should look like:

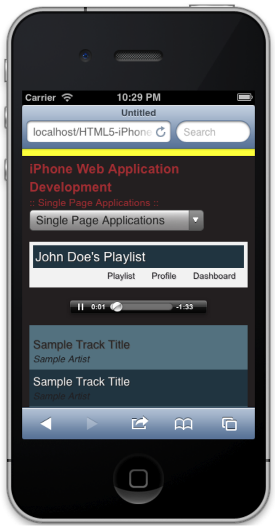
> You might be wondering about the styling for this application. Well lucky for you, the source code of this book has all of that written for you. We won't be going over the styling, since it doesn't really cover anything mobile specific and is more of a visual enhancement to display the applications we built here.



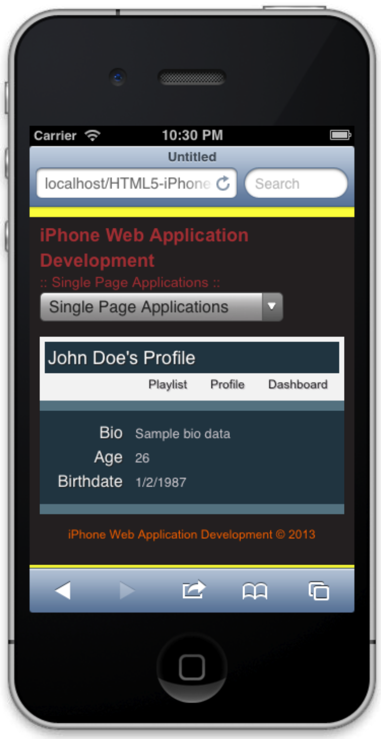
This screenshot of the application running in the iOS simulator showcases the dashboard view of the application we've written. In this view, we see our regular header and footer, including the title of the book, and a select control as the navigation. Inside the content area we see our dashboard templates rendering John Doe's dashboard and links to the playlist, profile, and back to the dashboard.



Here we are in the playlist and song views showcasing an audio control and the ability to switch between tracks. We can see the rendering of templates inside of templates (audio tracks inside of the playlist). With this example, we can see how separation of controls (models, views, and controllers) helps us distinguish logic from user interface.



In this screenshot, we see an audio track selected and playing under the **Playlist** page. It might seem like there's not a lot going on, but behind the scenes we have created a reusable application that allows user interaction without a page refresh.



Bio Sample bio data
Age 26
Birthdate 1/2/1987

iPhone Web Application Development © 2013

In this last screenshot, we see the profile view, displaying John Doe's short biography, age, and birth date. During the transition of playlist and profile we didn't see a page refresh, instead a content update. Analyzing the URL, we can see that history has been kept and thus, allowing us to use the native back button to maneuver through the one-page application.

Summary

Give yourself a pat on the back; we have finally reached the end of this chapter! It's been a good ride, and hopefully not too bad. At this point you are now prepared to tackle the development of one-page applications. From understanding the MVC design pattern to implementation, utilizing libraries such as Backbone and Underscore, you can now go and extend on this foundation by developing complex applications that tie into APIs and create a dynamically beautiful experience for the user.

8
Offline Applications

In this chapter, we will cover the basics of offline application development. Specifically, we'll discuss the application manifest, including the advantages and disadvantages, and see how we can handle offline interactions. Then we'll jump into how `localStorage` and `IndexedDB` can be used to temporarily store data on the client side. The content in this chapter will be supplemented with basic examples that will help you get up and going quickly. So, let's start out by covering how the application manifest is beneficial for us.

We will cover the following topics in this chapter:

- The application manifest
- Handling offline interactions
- The `localStorage` and `IndexedDB` APIs

Application Cache

Application Cache, or otherwise known as **AppCache**, allows you to define the resources that should be cached and available during offline use. This essentially makes your web application usable when the user is offline, and thus losing network connectivity or even refreshing the page will not affect your users' connectivity, and they will still be able to interact with your application. Let's get started by taking a look at what the application cache manifest looks like.

The manifest file

Our application manifest file contains the information regarding what resources will be cached by the file. It explicitly informs the browser of the resources you want to be made available offline, making it accessible for offline use, but also speeding up the page load through caching. The following code showcases a basic example of what a cache manifest looks like for the examples that accompany this chapter:

```
 CACHE MANIFEST

index.html

# stylesheets
../css/normalize.css
../css/main.css
../css/offline.css

# javascripts
../js/vendor/modernizr-2.6.1.min.js
../js/vendor/zepto.min.js
../js/helper.js
../js/App/App.js
../js/App/App.Nav.js
../js/App/App.Offline.js
../js/main.js
```

In the preceding example, there are a couple of things happening. First, we identify the cache manifest using the all-caps word, `CACHE MANIFEST`. This line is required and read by the browser to be exactly what it is, a cache manifest. The next line defines a file we would like to cache, `index.html`. Now, there is something we need to keep in mind; this manifest file is relative to where we are. This means that our file is located where `index.html` is in a directory (for example, `offline.appcache` is located at `localhost/`, just like `index.html`).

Moving on, we find that we can include comments in our manifest file by including the pound sign in front (`#stylesheets`). This helps us keep track of what's going on in this file. From here on out, we define the stylesheets and scripts that we would like to define relative to the page being viewed. At this time, we are looking at a real manifest file and breaking it down to understand it. As we progress through the chapter, we'll come back to this file and see how it affects the example built in this chapter.

Manifest implementation

In order to effectively use our cache manifest, we need to be able to associate it with the current page. But to do this, we also need to set up our server to handle the manifest correctly by sending over the correct MIME-type. Each type of server handles this a bit differently, and the directives may seem different, but they all achieve the same results—sending the correct type of header associated with that file.

In Apache, our configuration would look something like this:

```
AddType text/cache-manifest .appcache
```

As you can see, we have defined all files of type `.appcache` to be delivered with a content type of `text/cache-manifest`. This allows the browser to interpret the file correctly, and thus the browser associates it as `cache-manifest`. Although this is great, we are not yet done. To finish up, we need our page to define this file so that it is associated correctly.

To make our cache manifest related to our page, we need to set the `manifest` attribute on our HTML tag like so:

```
<html manifest="offline.appcache">
```

We are now done defining our application cache manifest and delivering it with the related page, but we have only briefly touched upon this technology. To fully understand its power, we need to see this in use. So we'll go ahead and create a simple example that gets us using what we have learned so far.

A simple example

Our example for this chapter will be based on the previous chapter. For this reason, we won't go into detail on the structure, styling, or scripts used to create this sample. However, it will be stripped down to an essential profile view that will allow you to understand offline applications without any additional knowledge needed from previous chapters. To begin with, let's look at our markup.

The markup

As always, the source code accompanying this book contains everything you need to get started with this chapter's goals. So, let's look at the meat of this sample and examine how this example will function, starting with the markup. If you open the index file located in the `offline` directory, you'll notice that our content should look something like this:

```
<div class="site-wrapper">
    <section class="view-profile">
        <header>
            <h1>John's Profile</h1>
            <a href="#edit">Edit</a>
        </header>
        <dl>
            <dt>Bio</dt>
            <dd>This is a little bit about myself; I like iphone web
apps and development in general.</dd>
            <dt>Age</dt>
            <dd>26</dd>
            <dt>Birthdate</dt>
            <dd>January 1st, 1987</dd>
        </dl>
        <form>
            <div class="field">
                <label for="bio">Bio</label>
                <textarea id="bio">
                    This is a little bit about me; I like iphone web
apps and development in general.
                </textarea>
            </div>
            <div class="field">
                <label for="age">Age</label>
                <input id="age" type="number" value="26">
            </div>
            <div class="field">
                <label>Birthdate</label>
                <input type="date" value="1987-01-01">
            </div>
        </form>
    </section>
</div>
```

As with any web application, especially those written in this book, this is just an excerpt of the overall application architecture, which would contain a header, footer, stylesheets, and scripts. But, the preceding markup describes a profile view that displays a user's information, including a short bio, age, and birth date. Along with this information is a form that allows you to update such information.

The experience for this application is as follows. First, when the user loads the page they should see all the information associated with them. Secondly, they will have the option to edit this information using a hyperlink with the text **Edit**. When the hyperlink is clicked, a form will appear that allows the user to edit their information. Accordingly, the **Edit** hyperlink will update to **View Profile**. Lastly, when the user clicks on **View Profile**, the form will hide and the display of the user's information will reappear.

The JavaScript

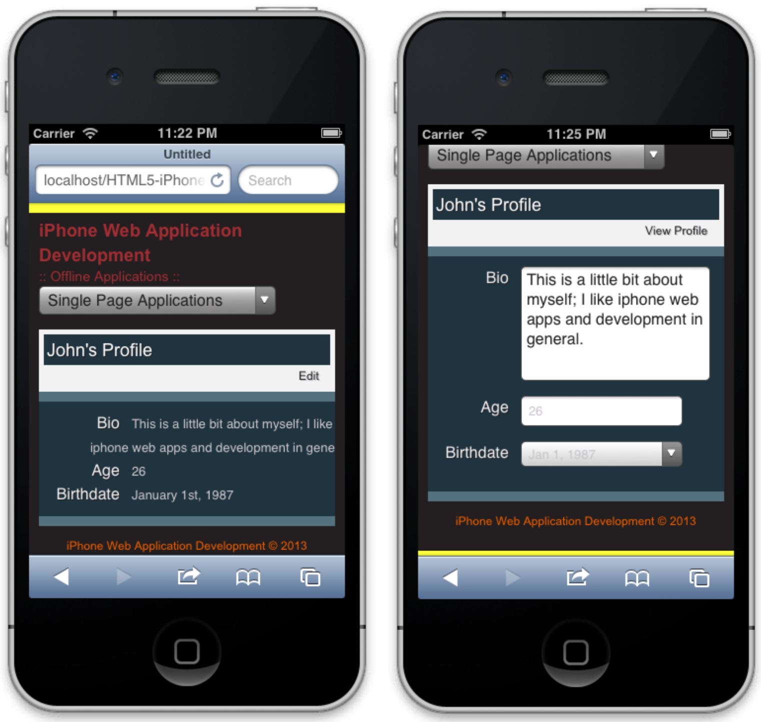
This isn't as complicated as it sounds. In fact, the script that is used to create the functionality of the page is dependent on the following script:

```
var $viewProfile = $('.view-profile'),
    $form = $viewProfile.find('form'),
    $dl = $viewProfile.find('dl'),
    $p = $('<p />');

function onEditClick(e) {
    e.preventDefault();
    e.stopImmediatePropagation();

    $form.show();
    $dl.hide();
    $(this).text('View Profile').attr('href', '#view');
}

function onViewClick(e) {
    e.preventDefault();
    e.stopImmediatePropagation();

    $form.hide();
    $dl.show();
    $(this).text('Edit').attr('href', '#edit');
}

$viewProfile.
    on('click', 'a[href="#edit"]', onEditClick).
    on('click', 'a[href="#view"]', onViewClick);
```

Let's look at what's going on in the preceding code to make sure no one is lost. First, we cache the elements appropriate to our page. In this way, we are optimizing the performance by not traversing the DOM every time we need to look something up. Then, we define our `onEditClick` and `onViewClick` event handlers, which show or hide the appropriate content blocks and then update the `text` and `href` properties of the anchor tag it relates to. Finally, we attach the `click` event on the anchor tags to the cached `$viewProfile` element.

> Note that the preceding JavaScript is an excerpt of the book's accompanying source code for this chapter. For simplicity, we have removed the closure and `Offline` class in order to better explain the meat and bones of the application being built here. Of course, you are welcome to either use the preceding code as is, or continue using the book's source code. Whichever way you choose, the desired outcome will be an application that shows or hides content based on the current state.

When the preceding code is executed and the page is loaded, these are the following states of the application:



Bio This is a little bit about myself; I like iphone web apps and development in gene

Age 26

Birthdate January 1st, 1987

Bio: This is a little bit about myself; I like iphone web apps and development in general.

Age: 26

Birthdate: Jan 1, 1987

Initial application state and application edit state

Now that we have a solid foundation of what the application should look like when we interact with it, we want to make sure that it gets cached. By implementing the techniques given at the beginning of our application, we should have an application that now functions offline.

Remember that we need to place the application cache manifest in the same directory as the sample application that is being built here. So, our application manifest needs to exist in the offline directory along with our `index.html` file. If you look at the source code for this chapter, you should see a working example of the structure and layout of our manifest and source files. Running this application on a server configured correctly will render our page offline. But the question is, how do we debug something like this? Well, this is what the next section tackles.

Debugging the cache manifest

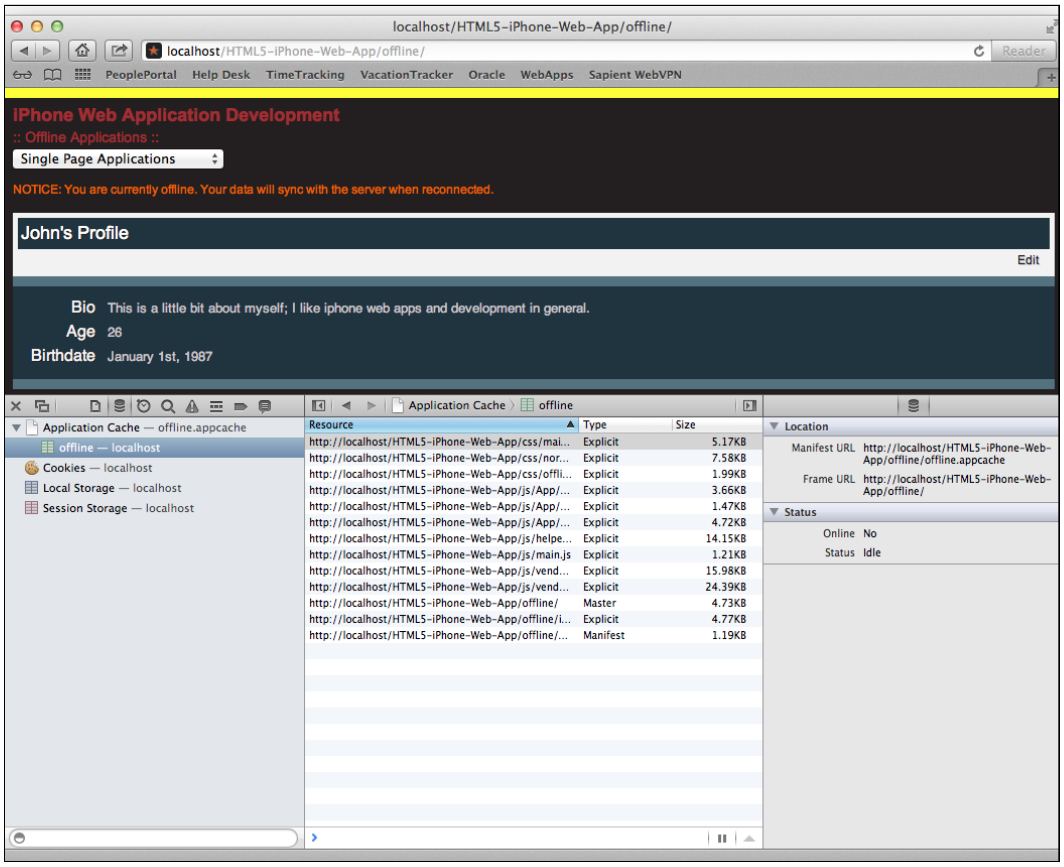
Debugging our offline applications is extremely important, after all, if our application depends on network connectivity, it's crucial we deliver an alternative experience that is successful to our users. However, it is not easy to debug an offline application. There are several reasons for this, but it's mostly based on the implementation of the application cache interface.

> *When Safari revisits your site, the site is loaded from the cache. If the cache manifest has changed, Safari checks the HTML file that declares the cache, as well as each resource listed in the manifest, to see if any of them has changed.*
>
> *A file is considered unchanged if it is byte-for-byte identical to the previous version; changing the modification date of a file does not trigger an update. You must change the contents of the file. (Changing a comment is sufficient.)*

This can be found in the Apple documentation at: `https://developer.apple.com/library/safari/#documentation/iPhone/Conceptual/SafariJSDatabaseGuide/OfflineApplicationCache/OfflineApplicationCache.html#//apple_ref/doc/uid/TP40007256-CH7-SW5`

Debugging in the browser

Based on the previous documentation, we can improve the debugging process by clearing the cache with updated resources. This could be as simple as updating a comment in our code, but to ensure that the correct assets are being cached, we can use modern browsers and use the debugger tools to review the assets being cached. Review the following screenshots to see how you can test that your assets are being cached:



localhost/HTML5-iPhone-Web-App/offline/
localhost/HTML5-iPhone-Web-App/offline/
Reader
PeoplePortal
Help Desk
TimeTracking
VacationTracker
Oracle
WebApps
Sapient WebVPN
iPhone Web Application Development
:: Offline Applications ::
Single Page Applications
Edit
Application Cache
offline

NOTICE: You are currently offline. Your data will sync with the server when reconnected.

John's Profile

Bio This is a little bit about myself; I like iphone web apps and development in general.
Age 26
Birthdate January 1st, 1987

Application Cache — offline.appcache
offline — localhost
Cookies — localhost
Local Storage — localhost
Session Storage — localhost

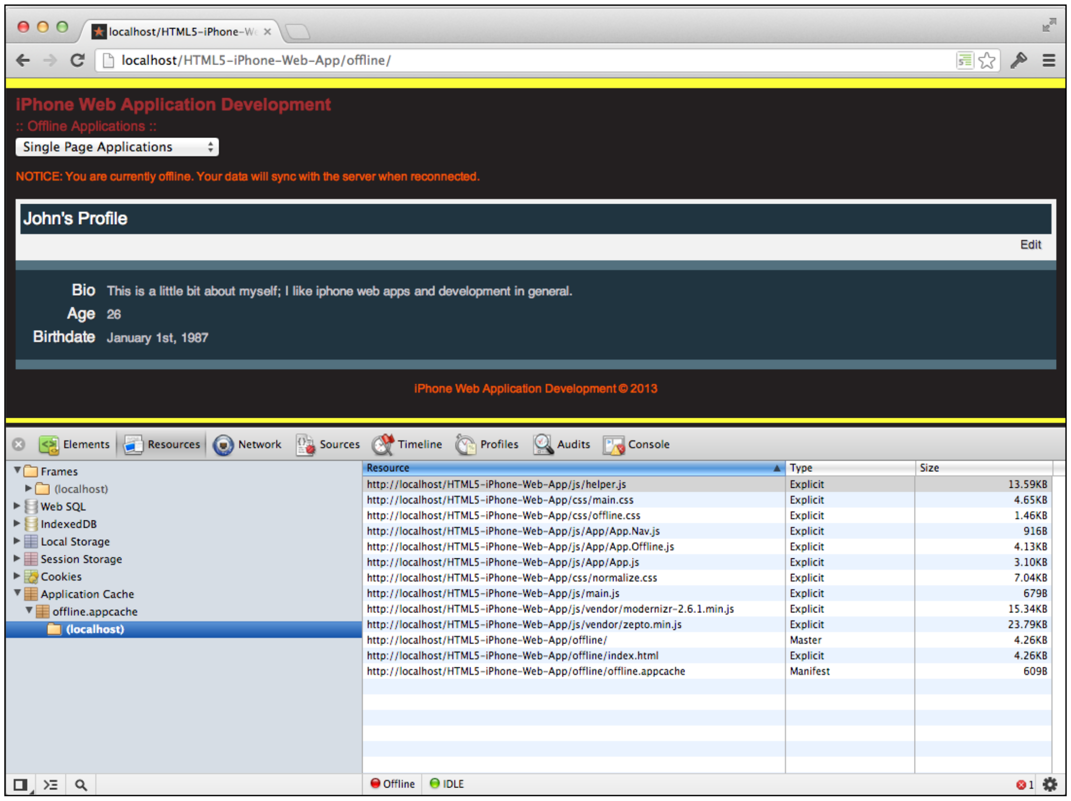
Resource	Type	Size
http://localhost/HTML5-iPhone-Web-App/css/mai...	Explicit	5.17KB
http://localhost/HTML5-iPhone-Web-App/css/nor...	Explicit	7.58KB
http://localhost/HTML5-iPhone-Web-App/css/offli...	Explicit	1.99KB
http://localhost/HTML5-iPhone-Web-App/js/App/...	Explicit	3.66KB
http://localhost/HTML5-iPhone-Web-App/js/App/...	Explicit	1.47KB
http://localhost/HTML5-iPhone-Web-App/js/App/...	Explicit	4.72KB
http://localhost/HTML5-iPhone-Web-App/js/helpe...	Explicit	14.15KB
http://localhost/HTML5-iPhone-Web-App/js/main.js	Explicit	1.21KB
http://localhost/HTML5-iPhone-Web-App/js/vend...	Explicit	15.98KB
http://localhost/HTML5-iPhone-Web-App/js/vend...	Explicit	24.39KB
http://localhost/HTML5-iPhone-Web-App/offline/	Master	4.73KB
http://localhost/HTML5-iPhone-Web-App/offline/i...	Explicit	4.77KB
http://localhost/HTML5-iPhone-Web-App/offline/...	Manifest	1.19KB

Location
Manifest URL http://localhost/HTML5-iPhone-Web-App/offline/offline.appcache
Frame URL http://localhost/HTML5-iPhone-Web-App/offline/

Status
Online No
Status Idle

Safari developer tools – resources

The developer tools in Safari, shown in the preceding screenshot, help us debug our application cache by providing a **Resource** tab that allows us to analyze the application cache for multiple domains. In this example, we can see the resources related to the domain for our sample application. When we review the application cache, we can see a list of files associated with the cache to the right. Additionally, we can see the location of the file and the status of the user; in this case, we are online and idle.



NOTICE: You are currently offline. Your data will sync with the server when reconnected.

John's Profile

Bio	This is a little bit about myself; I like iphone web apps and development in general.
Age	26
Birthdate	January 1st, 1987

iPhone Web Application Development © 2013

Resource	Type	Size
http://localhost/HTML5-iPhone-Web-App/js/helper.js	Explicit	13.59KB
http://localhost/HTML5-iPhone-Web-App/css/main.css	Explicit	4.65KB
http://localhost/HTML5-iPhone-Web-App/css/offline.css	Explicit	1.46KB
http://localhost/HTML5-iPhone-Web-App/js/App/App.Nav.js	Explicit	916B
http://localhost/HTML5-iPhone-Web-App/js/App/App.Offline.js	Explicit	4.13KB
http://localhost/HTML5-iPhone-Web-App/js/App/App.js	Explicit	3.10KB
http://localhost/HTML5-iPhone-Web-App/css/normalize.css	Explicit	7.04KB
http://localhost/HTML5-iPhone-Web-App/js/main.js	Explicit	679B
http://localhost/HTML5-iPhone-Web-App/js/vendor/modernizr-2.6.1.min.js	Explicit	15.34KB
http://localhost/HTML5-iPhone-Web-App/js/vendor/zepto.min.js	Explicit	23.79KB
http://localhost/HTML5-iPhone-Web-App/offline/	Master	4.26KB
http://localhost/HTML5-iPhone-Web-App/offline/index.html	Explicit	4.26KB
http://localhost/HTML5-iPhone-Web-App/offline/offline.appcache	Manifest	609B

Chrome developer tools – resources

The Chrome developer tools similarly help display information related to the application cache. Although the user interface is a bit different, it contains all the same components needed to review the assets associated with your application cache. This view also includes the online status of your application; in this example, we are not online and idle.

Debugging using JavaScript

The application cache can also be debugged using JavaScript, and luckily for us, the implementation of the application cache manifest is extremely easy to interact with. There are multiple events we can listen to, including `progress`, `error`, and `updateready`. When we listen to these events, we can choose to implement a supplementary experience, but for our use case here, we'll just log out the event.

```
window.applicationCache.addEventListener('cached', handleCacheEvent,
false);
window.applicationCache.addEventListener('checking', handleCacheEvent,
false);
window.applicationCache.addEventListener('downloading',
handleCacheEvent, false);
window.applicationCache.addEventListener('error', handleCacheError,
false);
window.applicationCache.addEventListener('noupdate', handleCacheEvent,
false);
window.applicationCache.addEventListener('obsolete', handleCacheEvent,
false);
window.applicationCache.addEventListener('progress', handleCacheEvent,
false);
window.applicationCache.addEventListener('updateready',
handleCacheEvent, false);

function handleCacheEvent(e) {
  console.log(e);
}

function handleCacheError(e) {
  console.log(e);

}
```

In the preceding script, we listen to the events defined by the specification (`http://www.whatwg.org/specs/web-apps/current-work/#appcacheevents`) and call either the `handleCacheEvent` or `handleCacheError` methods. In each of these methods, we just log out the event itself; however, if we wanted to, we could create an alternate experience.

There are also a number of action methods that can be employed during the implementation of the application cache process. For example, we could manually update the cache using the `update` method.

```
window.applicationCache.update();
```

Although the preceding method could be helpful for us, remember that the cache will only update if the contents themselves have changed. In reality, the `update` method triggers the download process (`http://www.whatwg.org/specs/web-apps/current-work/#application-cache-download-process`), which does not tell the browser to get the latest cache. The `swapCache` method is another action call that can be used to debug our applications by switching the cache to the latest version.

```
window.applicationCache.swapCache();
```

Keep in mind that if we make this call, the assets are not updated automatically. The way we get the updated assets are on a page refresh. Based on the specification, an easier way to do what we need here would be to do `location.reload()` (`http://www.whatwg.org/specs/web-apps/current-work/#dom-appcache-swapcache`).

At this point, we have a good idea about the application cache manifest, including how it functions, the implementation details, and ultimately how to debug it. Now we need to know how to handle offline interactions using the preceding methods and more. When we have a good understanding of both aspects, we'll be able to create simple offline applications that utilize this technology.

Handling offline applications

At this point, we have learned how to cache our files on the client side using the application manifest interface to not only speed up our site, but also make it available to our users when they are offline. However, this technique doesn't take into account what you should do to handle interactions by the user. In this case, we need to make sure our application has usable parts that can make the application seamless when they lose connection.

A simple use case

Before we proceed, let's define a simple use case for why handling an offline application is useful to both the user and us. Let's say we have a user named John, and John is commuting to work and is currently updating his profile in a web application on his iPhone. The commute involves some spotty network connections, and sometimes he does lose connection. He would like to be able to continue using the application while he is on his way to work, instead of waiting to do so at work.

Given the world we live in today, and how a spotty interaction could really cost a company a customer, we definitely want to handle this gracefully. It doesn't mean that we will provide all the services to the user while they are offline, that would be unreasonable. What this does mean is that we need to inform the user that they are offline and that certain features are currently disabled for this reason.

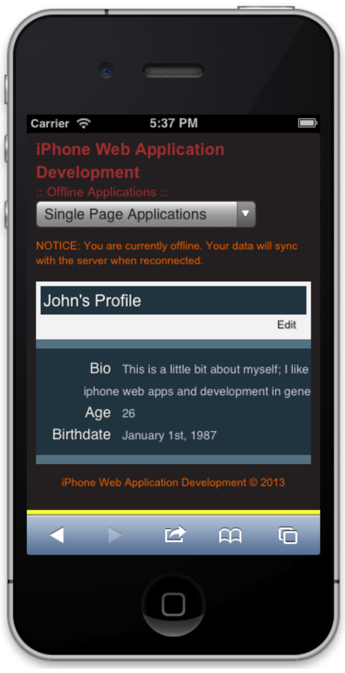
Detecting network connectivity

Now, you may be asking, "How do you detect a network connection?" Well, it's actually pretty simple. Let's take a look at the following code:

```
var $p = $('<p />');
if(!navigator.onLine) {
  $p.
    text('NOTICE: You are currently offline. Your data will sync with
the server when reconnected.');

  $('.view-profile').
    before($p);
}
```

Let's review the preceding code briefly. The first part of this code creates a cached element in memory and stores it in the `$p` variable. The next line, which is the most important, detects online connectivity by checking the `navigator` object's `onLine` property. If the person is not online, we finally set the text of that cached element and append it to our previous code.

If our application were offline, it would look like this:


Carrier
5:37 PM
iPhone Web Application Development
:: Offline Applications ::
Single Page Applications
John's Profile
Edit

NOTICE: You are currently offline. Your data will sync with the server when reconnected.

Bio This is a little bit about myself; I like iphone web apps and development in gene

Age 26

Birthdate January 1st, 1987

iPhone Web Application Development © 2013

Detecting network connectivity

Of course, this is a stripped down version of how you would handle network connectivity in your real-world applications, but it showcases that you can pick up the status of the network and determine the experience offline. This is great for us because it opens up a new world in web development that we haven't been able to explore. But of course, this also entails some complex forethought into how such experiences are handled.

Going back to the use case we defined, the user wanted to update his/her profile information, and obviously this would be extremely difficult to achieve without the necessary resources. Fortunately for us, there are two new technologies we can use to accomplish simple tasks similar to this use case. So, let's briefly go over these two technologies.

The localStorage API

Although offline status is a feature of the new HTML5 specification, it goes hand in hand with another feature of HTML5, namely storage (`http://dev.w3.org/html5/webstorage/`). Storage is something that many developers have thought previously to be a one-to-one relationship with a backend system. This is no longer true, since we are now able to save data on the client device using the `localStorage` API.

Let's cover a brief example using `localStorage` to see how it works:

```
localStorage.setItem('name', 'John Smith');
```

The code we just wrote here has multiple parts to it. First, there is a global object named `localStorage`. This object has methods that a developer can interact with, including the `setItem` method. Lastly, the `setItem` method takes two parameters, both of which are strings.

To retrieve the value of the item we just set, we would do this:

```
localStorage.getItem('name');
```

Pretty cool, huh? The only downside is that the current implementation of this describes the API as only accepting strings for each key/value pair, similar to a JavaScript object. However, we can go beyond this limitation by doing the following:

```
localStorage.setItem('name', JSON.stringify({ 'first': 'John', 'last':
'Smith' }));
```

The difference here is that we are also accessing the built-in `JSON` object that turns the object into a string so that the `localStorage` API can efficiently store this as a plain string. Otherwise, you would store `[object Object]`, which is just the type of the object.

To access this information, we would do the following:

```
JSON.parse(localStorage.getItem('name'));
```

When you run this code in the console, you should then see an object. What the `JSON` functionality has done is turn the "stringified" object into a traditional JavaScript object. This makes accessing the data we have stored easy and efficient.

As you can start to gather, we have the power to store information on the client side that we could not in the past, allowing us to temporarily allow the user to interact with certain aspects of the site while the application is offline. Combining the storage and offline features of HTML5 allows us to bring deeper interactions to our applications while meeting client and user expectations.

However, the limitation to `localStorage` is the amount of information that can be stored. For this reason, another technology exists, called `IndexedDB`. Although its support is not consistent and the technology is still in experimental stages across browsers, it's worth taking a look at. Unfortunately, we won't cover that technology in this book, due to its lack of support in iOS Safari, but it's still worth some review (`https://developer.mozilla.org/en-US/docs/IndexedDB`).

Summary

In this chapter, we covered the basics of application cache including an example of its implementation. Our review pointed out the benefits of using this new technology, but also discussed the downsides, such as the inconsistent support, mostly with older browsers, and the issues we face when testing it. We learned how to handle offline interactions and how `localStorage` and `IndexedDB` allow us to store information temporarily on the client side as a solution. In the next chapter, we'll discuss performance optimization and see how this has played out in the application developed in this book.

9
Principles of Clean and Optimized Code

Throughout the book, we have stressed the importance of optimizing from the very start of our application development. Although we have looked at topics from caching elements in JavaScript to modularizing our styles, in this chapter we want to summarize the techniques used in the book. After all, performance is highly important in our mobile applications for several reasons. In this chapter, we will cover optimizing our styles, scripts, and media. In addition to covering optimization techniques, we'll go over good coding standards that enhance the maintainability of your codebase while also enhancing performance at the same time. We'll start out by discussing stylesheets.

In this chapter, we will cover the following topics:

- Validating CSS
- Profiling CSS
- CSS best practices
- Validating JavaScript
- Profiling JavaScript
- JavaScript best practices

Optimizing stylesheets

Traditionally, styles have been "plopped" onto web applications without any forethought. Usually, we would just style our pages without any thought to modularity, reusability, and maintainability. However, this is no longer acceptable due to the extensive nature of web applications today.

In this book, we have strived to adhere to a couple of industry standards, such as modularity. However, we do have tools now that can help us validate and profile our styles. Starting with an analysis of a sample CSS file, we can then optimize those styles; this is what we aim to do in this section of the chapter.

Validating our CSS

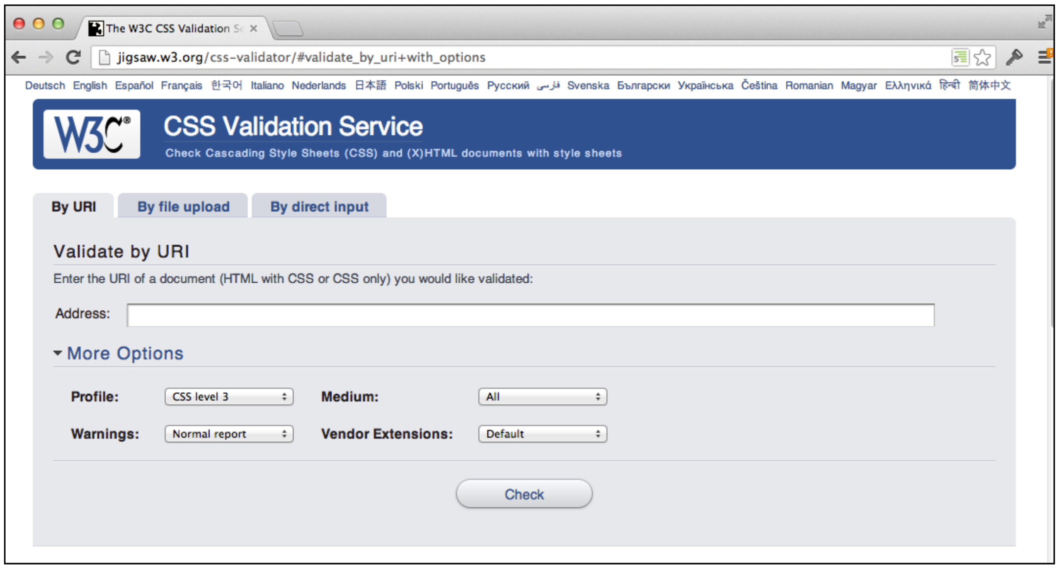
In order to optimize our stylesheets, we need to first verify that our CSS is valid and compliant with today's standards. We can use various tools out there to validate our styles, including the W3C CSS Validator and a tool called **CSS Lint**. Both these tools check your stylesheets and give you a summary of what is wrong, why it is wrong, and what you should do about it.

W3C CSS Validator

To access the W3C CSS Validator, you can visit the following URL:

`http://jigsaw.w3.org/css-validator/`

The following screenshot displays the W3C validator default view, which allows you to enter in the URI of the page containing styles. It will automatically pick up your stylesheets and validate them according to the W3C specification. However, we are not limited only to having our page crawlable on a live or production-ready site. We also have the option of uploading our stylesheets, or putting them directly into this application.


The W3C CSS Validation S
jigsaw.w3.org/css-validator/#validate_by_uri+with_options
Deutsch English Español Français 한국어 Italiano Nederlands 日本語 Polski Português Русский فارسی Svenska Български Українська Čeština Romanian Magyar Ελληνικά हिन्दी 简体中文
CSS Validation Service
Check Cascading Style Sheets (CSS) and (X)HTML documents with style sheets
By URI
By file upload
By direct input
More Options
Check

Validate by URI

Enter the URI of a document (HTML with CSS or CSS only) you would like validated:

Address:

Profile: CSS level 3

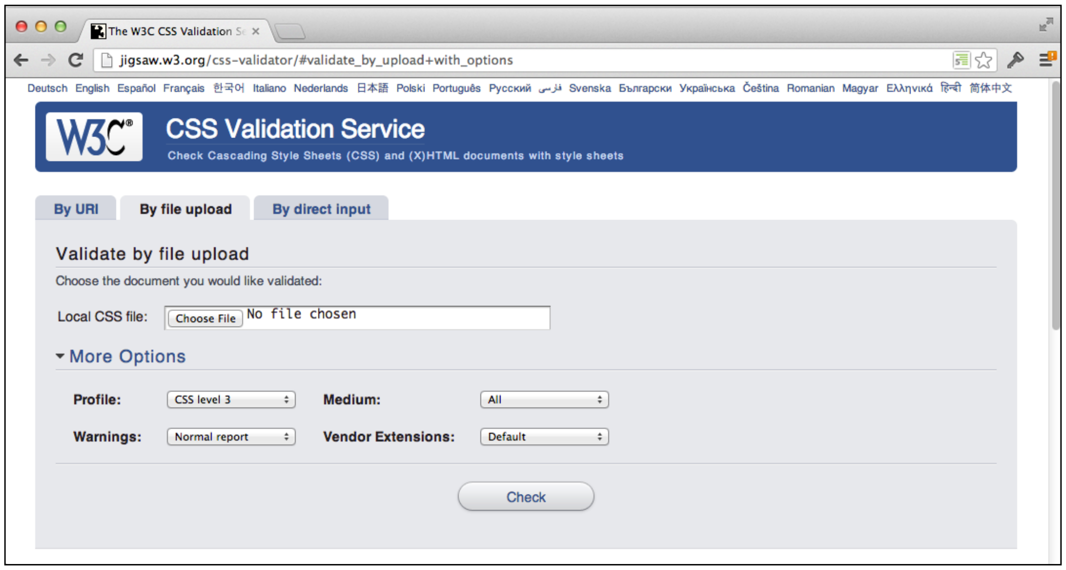
Medium: All

Warnings: Normal report

Vendor Extensions: Default

The W3C CSS Validator – URI view

In the following view, you can see that we can validate the styles through a file upload process. This will simply run those stylesheets through a processor at the backend to check if the styles are valid; once this process is done, we get our results.


The W3C CSS Validation S
jigsaw.w3.org/css-validator/#validate_by_upload+with_options
Deutsch English Español Français 한국어 Italiano Nederlands 日本語 Polski Português Русский فارسی Svenska Български Українська Čeština Romanian Magyar Ελληνικά हिन्दी 简体中文
CSS Validation Service
Check Cascading Style Sheets (CSS) and (X)HTML documents with style sheets
By URI
By file upload
By direct input
Choose File
More Options
Check

Validate by file upload

Choose the document you would like validated:

Local CSS file: No file chosen

Profile: CSS level 3

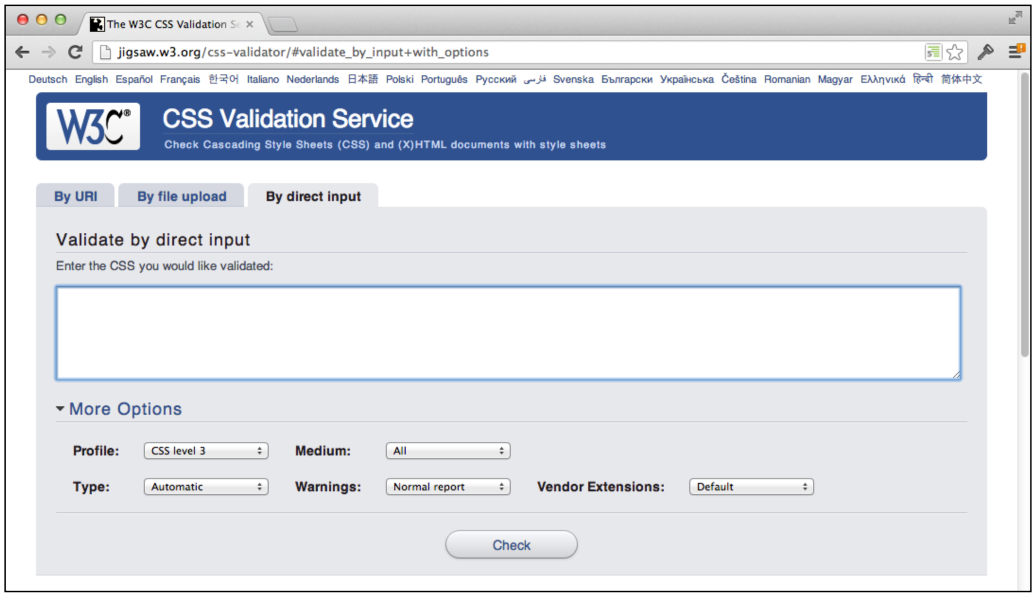
Medium: All

Warnings: Normal report

Vendor Extensions: Default

The W3C CSS Validator – file upload view

Lastly, we have the option to directly insert our styles into the tool, perhaps the fastest and easiest solution depending on the project and needs of the team or individual. We don't have to worry about styles being stripped or modified in any way; the text field will handle all your input correctly. Again, similar to the other views, the input will run through a processor and present to you the results once the **Check** button is clicked.



jigsaw.w3.org/css-validator/#validate_by_input+with_options

Deutsch English Español Français 한국어 Italiano Nederlands 日本語 Polski Português Русский فارسی Svenska Български Українська Čeština Romanian Magyar Ελληνικά हिन्दी 简体中文

CSS Validation Service

Check Cascading Style Sheets (CSS) and (X)HTML documents with style sheets

Validate by direct input

Enter the CSS you would like validated:

Profile: CSS level 3 Medium: All

Type: Automatic Warnings: Normal report Vendor Extensions: Default

The W3C CSS Validator – direct input view

Customizable options

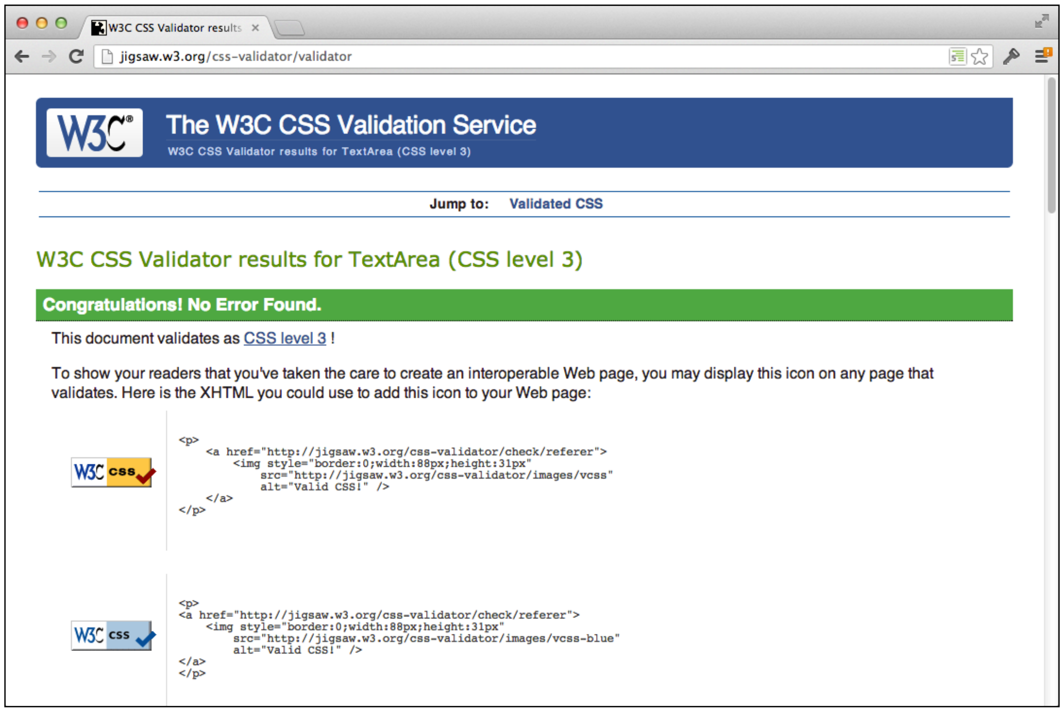
As with any good-quality assurance tool, we need to have the ability to customize the options of this tool to fit our needs. In this case, we have various options that are available to us, including:

- **Profile**: This option specifies the profile to be used when validating styles, for example, CSS Level 1, CSS Level 2, CSS Level 3, and so on.
- **Warnings**: This option specifies the warning to be presented in the report, for example, Normal, Most Important, No Warnings, and so on.
- **Medium**: This option specifies the medium the stylesheets are supposed to represent, for example, Screen, Print, Handheld, and so on.
- **Vendor Extensions**: This option specifies the way vendor extensions (`-webkit-`, `-moz-`, `-o-`) are to be handled in the report, for example, Warnings or Errors.

Validating a successful example

Let's take a look at a successful validation example. First, let's use some styles we created in a previous chapter to see if the CSS passes validation; in particular, let's use the `singlepage.css` file contents and paste it into the direct input view of the W3C validator and run it with the default options.

When we run the validator, our result should look like this:


W3C CSS Validator results
jigsaw.w3.org/css-validator/validator
The W3C CSS Validation Service
W3C CSS Validator results for TextArea (CSS level 3)
Jump to: Validated CSS

W3C CSS Validator results for TextArea (CSS level 3)

Congratulations! No Error Found.

This document validates as CSS level 3 !

To show your readers that you've taken the care to create an interoperable Web page, you may display this icon on any page that validates. Here is the XHTML you could use to add this icon to your Web page:

```
<p>
    <a href="http://jigsaw.w3.org/css-validator/check/referer">
        <img style="border:0;width:88px;height:31px"
            src="http://jigsaw.w3.org/css-validator/images/vcss"
            alt="Valid CSS!" />
    </a>
</p>
```

```
<p>
<a href="http://jigsaw.w3.org/css-validator/check/referer">
    <img style="border:0;width:88px;height:31px"
        src="http://jigsaw.w3.org/css-validator/images/vcss-blue"
        alt="Valid CSS!" />
</a>
</p>
```

The W3C CSS Validator – successful validation

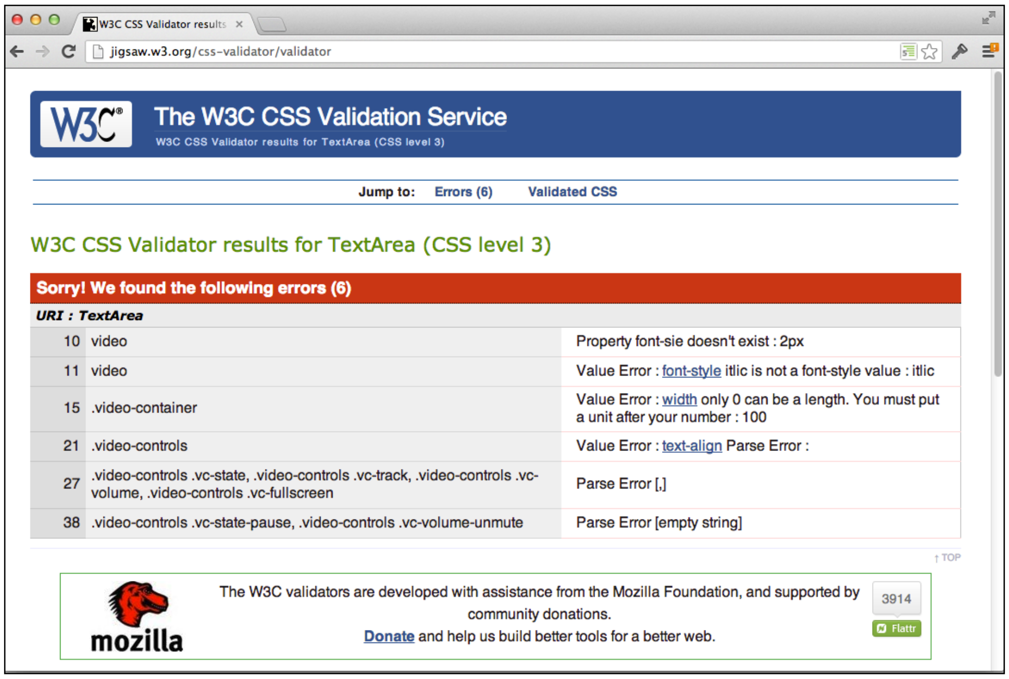
As you can see, the output is successful and passes the CSS Level 3 specification. It's so amazing that we even get badges from the validator to put on our site! But don't do that; even though you should give yourself a pat on the back, this is something that most of us don't really like on our sites. Now let's look at an unsuccessful example.

Validating an unsuccessful example

Errors happen all the time in programming, and so it's only natural for us to encounter validation errors in our styling, scripts, and markup. So, let's take a look at what an example of validation errors in the W3C CSS Validator looks like. For this example, we'll use a variation of the `video.css` file we created in *Chapter 2*, *Integrating HTML5 Video*. For the purpose of this example, I've introduced several mistakes with the following styles:

```
video {
  display: block;
  width: 100%;
  max-width: 640px;
  margin: 0 auto;
  font-size: 2px;
  font-style: italics;
}

.video-container {
  width: 100;
}

.video-controls {
  margin: 12px auto;
  width: 100%;
  text-align:;
}

.video-controls .vc-state,
.video-controls .vc-track,
.video-controls .vc-volume,
.video-controls .vc-fullscreen, {
  display: inline-block;
  margin-right: 10px;
}

.video-controls .vc-fullscreen {
  margin-right: 0;
}

.video-controls .vc-state-pause,
.video-controls .vc-volume-unmute {
  display: none;
```

When we pass the preceding styles through the W3C CSS Validator, we get the following:


W3C CSS Validator results
jigsaw.w3.org/css-validator/validator
The W3C CSS Validation Service
W3C CSS Validator results for TextArea (CSS level 3)
Jump to: Errors (6) Validated CSS

W3C CSS Validator results for TextArea (CSS level 3)

Sorry! We found the following errors (6)

URI : TextArea

10	video	Property font-sie doesn't exist : 2px
11	video	Value Error : font-style itlic is not a font-style value : itlic
15	.video-container	Value Error : width only 0 can be a length. You must put a unit after your number : 100
21	.video-controls	Value Error : text-align Parse Error :
27	.video-controls .vc-state, .video-controls .vc-track, .video-controls .vc-volume, .video-controls .vc-fullscreen	Parse Error [,]
38	.video-controls .vc-state-pause, .video-controls .vc-volume-unmute	Parse Error [empty string]

The W3C validators are developed with assistance from the Mozilla Foundation, and supported by community donations.
Donate and help us build better tools for a better web.

The W3C CSS Validator – unsuccessful validaton

In the preceding example we're getting a couple of value, property, and parse errors, all of which can be easily solved by the references given in this unsuccessful validation example. What's great about this is that instead of trying to figure out what might be breaking your layout, a simple fix on the errors shown in the screenshot might solve all your problems.

In a sense this is basically all you need to make sure that your CSS is valid and compliant across several browsers. However, what if you could prevent these errors from happening? Well, there's a tool for that, CSS Lint.

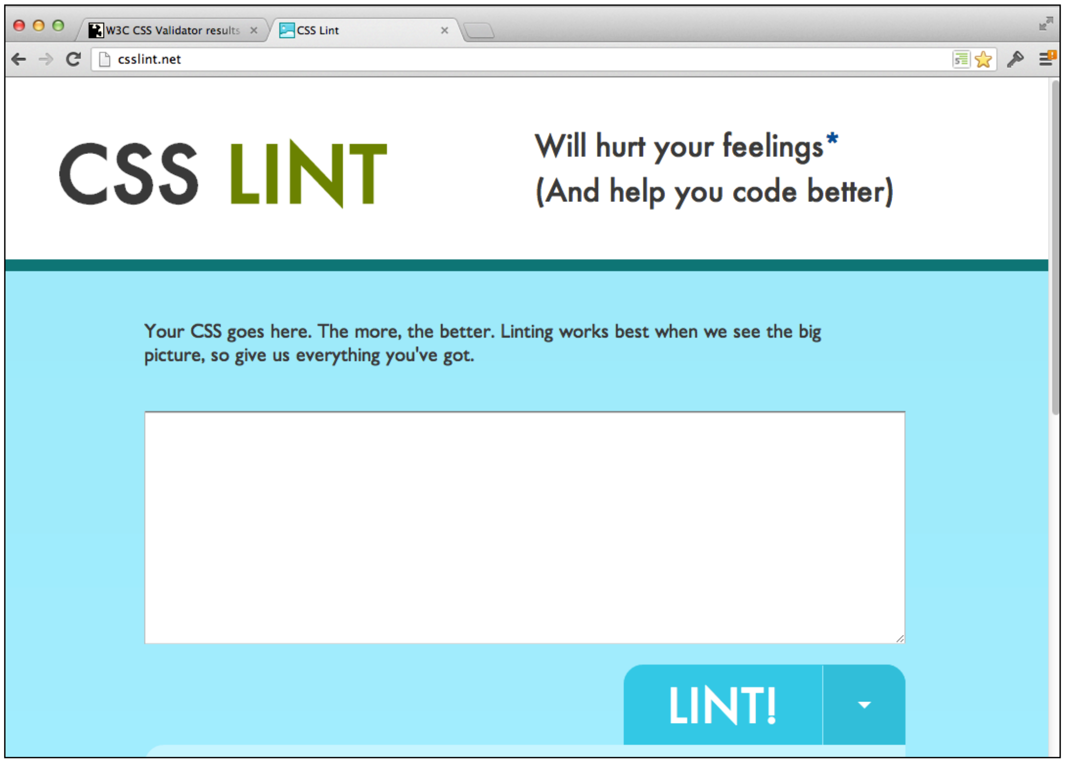
CSS Lint

In most cases, we want to avoid having errors all together when we code, and it would be helpful to catch these early on with a tool of some sort. CSS Lint is that tool, and in fact can be used right inside the text editor or IDE of your choice. CSS Lint not only checks your styling against certain principles of CSS (such as the box model), but also does a lot of the syntax checking, helping you debug your styles effectively.

> *CSS Lint points out problems with your CSS code. It does basic syntax checking as well as applying a set of rules to the code that look for problematic patterns or signs of inefficiency. The rules are all pluggable, so you can easily write your own or omit ones you don't want.*

The details regarding CSS Lint can be found at `https://github.com/stubbornella/csslint/wiki/About`.

Similar to the W3C CSS Validator, CSS Lint has its own site where you can copy and paste your styles into a text area and have the processor checkout your styling. The page in which we interact with looks like this:



Will hurt your feelings*
(And help you code better)

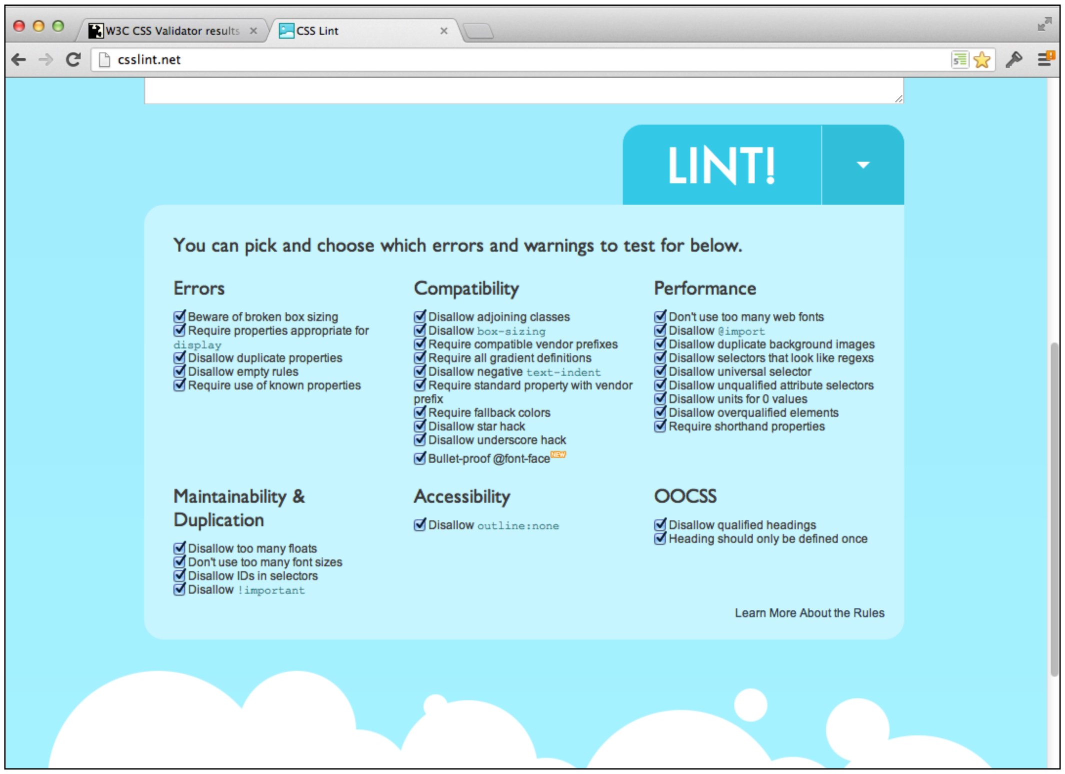
Your CSS goes here. The more, the better. Linting works best when we see the big picture, so give us everything you've got.

CSS Lint

Customizable options

CSS Lint also comes with customizable options, which are extensive, that you can customize to fit you or your team's needs. There are six sections of customizable options, including **Errors**, **Maintainability & Duplication**, **Compatibility**, **Accessibility**, **Performance**, and **OOCSS** (**Object Oriented CSS**).

The customizable options are located right below the **Lint!** button:



You can pick and choose which errors and warnings to test for below.

Errors

- ✓ Beware of broken box sizing
- ✓ Require properties appropriate for `display`
- ✓ Disallow duplicate properties
- ✓ Disallow empty rules
- ✓ Require use of known properties

Compatibility

- ✓ Disallow adjoining classes
- ✓ Disallow `box-sizing`
- ✓ Require compatible vendor prefixes
- ✓ Require all gradient definitions
- ✓ Disallow negative `text-indent`
- ✓ Require standard property with vendor prefix
- ✓ Require fallback colors
- ✓ Disallow star hack
- ✓ Disallow underscore hack
- ✓ Bullet-proof @font-face NEW

Performance

- ✓ Don't use too many web fonts
- ✓ Disallow `@import`
- ✓ Disallow duplicate background images
- ✓ Disallow selectors that look like regexs
- ✓ Disallow universal selector
- ✓ Disallow unqualified attribute selectors
- ✓ Disallow units for 0 values
- ✓ Disallow overqualified elements
- ✓ Require shorthand properties

Maintainability & Duplication

- ✓ Disallow too many floats
- ✓ Don't use too many font sizes
- ✓ Disallow IDs in selectors
- ✓ Disallow `!important`

Accessibility

- ✓ Disallow `outline:none`

OOCSS

- ✓ Disallow qualified headings
- ✓ Heading should only be defined once

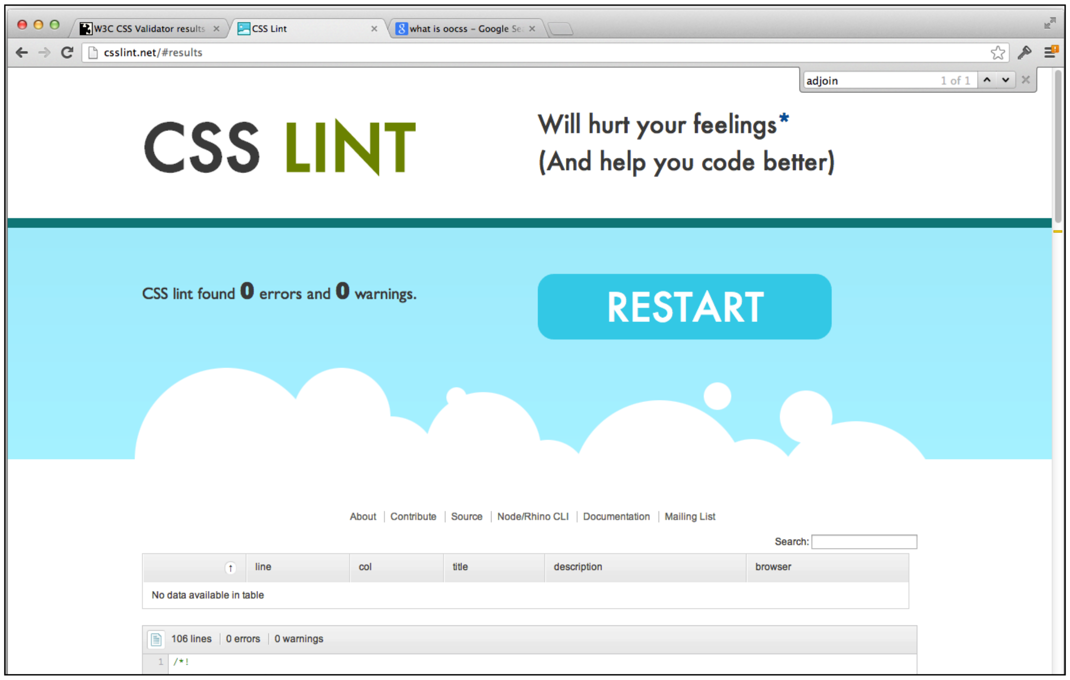
Learn More About the Rules

CSS Lint options

Checking the appropriate options enables the engine to validate against those properties. Usually these options vary between projects; for example, you may be working on an application that requires padding and width to be set on certain elements and thus, unchecking the **Beware of broken box sizing** option may be more suitable for you, so that you are not presented with multiple errors.

Validating a successful example using CSS Lint

When we customize our options and pass the page through CSS Lint with proper stylesheets that meet standards while also catering to the team's needs, we should receive a successful validation, such as the following screenshot:


W3C CSS Validator results
CSS Lint
what is oocss - Google Se
csslint.net/#results
adjoin
1 of 1
CSS LINT
Will hurt your feelings*
(And help you code better)
RESTART
About | Contribute | Source | Node/Rhino CLI | Documentation | Mailing List
Search:
line
col
title
description
browser
106 lines | 0 errors | 0 warnings
/*!

CSS lint found 0 errors and 0 warnings.

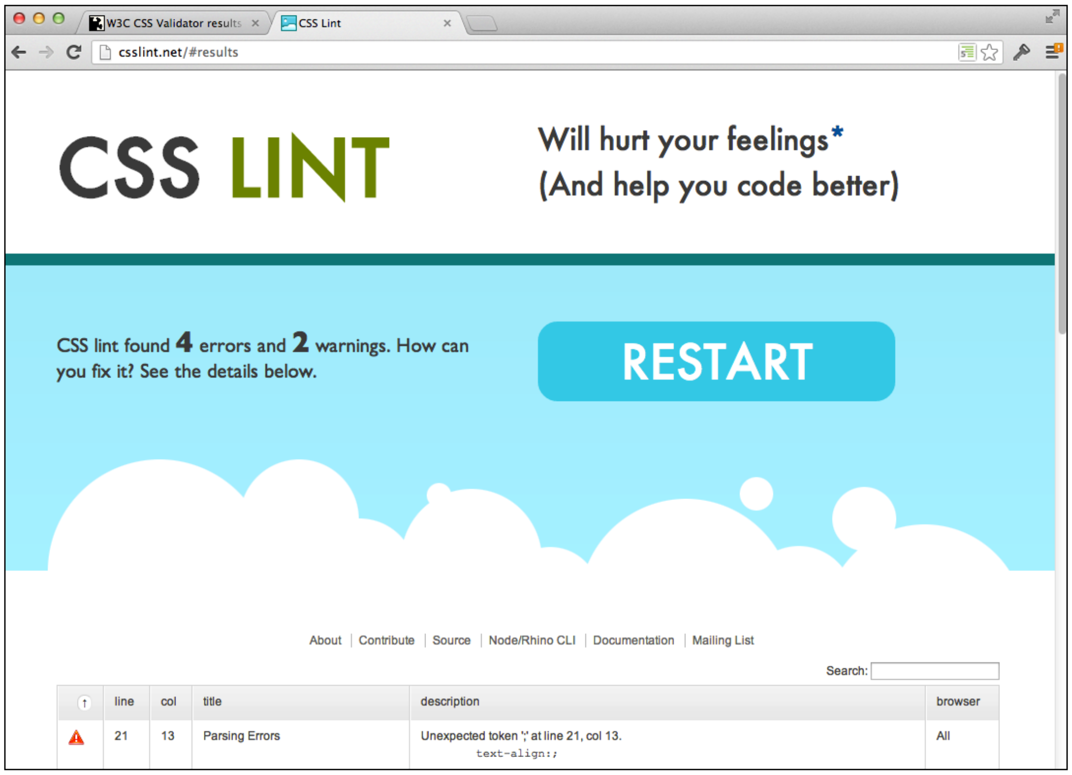
No data available in table

CSS Lint – successful validation

In the above case, our CSS styles pass and no additional information is needed. However, what happens when our CSS does not pass validation?

Validating an unsuccessful example using CSS Lint

If we take the alternative video styles we created in the previous section for the W3C CSS Validator and pass them through CSS Lint, we get the following:

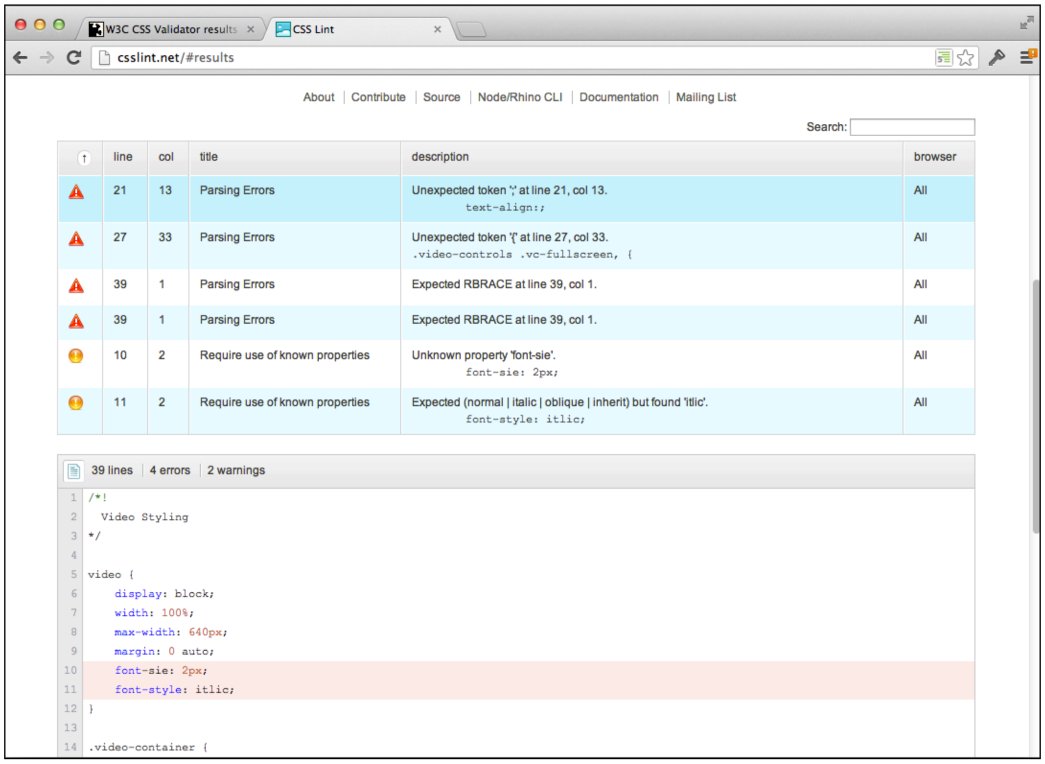

W3C CSS Validator results
CSS Lint
csslint.net/#results
CSS LINT
Will hurt your feelings*
(And help you code better)
RESTART
About | Contribute | Source | Node/Rhino CLI | Documentation | Mailing List
Search:

CSS lint found 4 errors and 2 warnings. How can you fix it? See the details below.

	line	col	title	description	browser
⚠	21	13	Parsing Errors	Unexpected token ';' at line 21, col 13. `text-align:;`	All

CSS Lint – unsuccessful validation

However, just because we received four errors and two warnings does not mean we are helpless. In fact, when we scroll down the page, we will see a list of items that need to be taken care of; it also includes the type of problem, description, and line that the error occurs on:


W3C CSS Validator results
CSS Lint
csslint.net/#results
About | Contribute | Source | Node/Rhino CLI | Documentation | Mailing List
Search:

↑	line	col	title	description	browser
⚠	21	13	Parsing Errors	Unexpected token ';' at line 21, col 13. `text-align:;`	All
⚠	27	33	Parsing Errors	Unexpected token '{' at line 27, col 33. `.video-controls .vc-fullscreen, {`	All
⚠	39	1	Parsing Errors	Expected RBRACE at line 39, col 1.	All
⚠	39	1	Parsing Errors	Expected RBRACE at line 39, col 1.	All
●	10	2	Require use of known properties	Unknown property 'font-sie'. `font-sie: 2px;`	All
●	11	2	Require use of known properties	Expected (normal | italic | oblique | inherit) but found 'itlic'. `font-style: itlic;`	All

39 lines | 4 errors | 2 warnings

```
/*!
  Video Styling
*/

video {
    display: block;
    width: 100%;
    max-width: 640px;
    margin: 0 auto;
    font-sie: 2px;
    font-style: itlic;
}

.video-container {
```

CSS Lint – unsuccessful validation listing

Integrating CSS Lint

Although we have a **graphical user interface** (**GUI**) that we can use to validate our styles, it would be easier if we could streamline the process with our personal development workflow. For example, it would be great if our CSS could be validated as we saved our stylesheet in our text editor or **integrated development environment** (**IDE**). CSS Lint is very extensible, allowing us to achieve these integrated workflows.

Some IDEs and text editor vendors have already implemented CSS Lint, including Sublime Text, Cloud 9, Microsoft Visual Studio, and Eclipse Orion. Although the installation and setup of CSS Lint into your preferred tool is out of the scope of this book, you can look up all the information needed here:

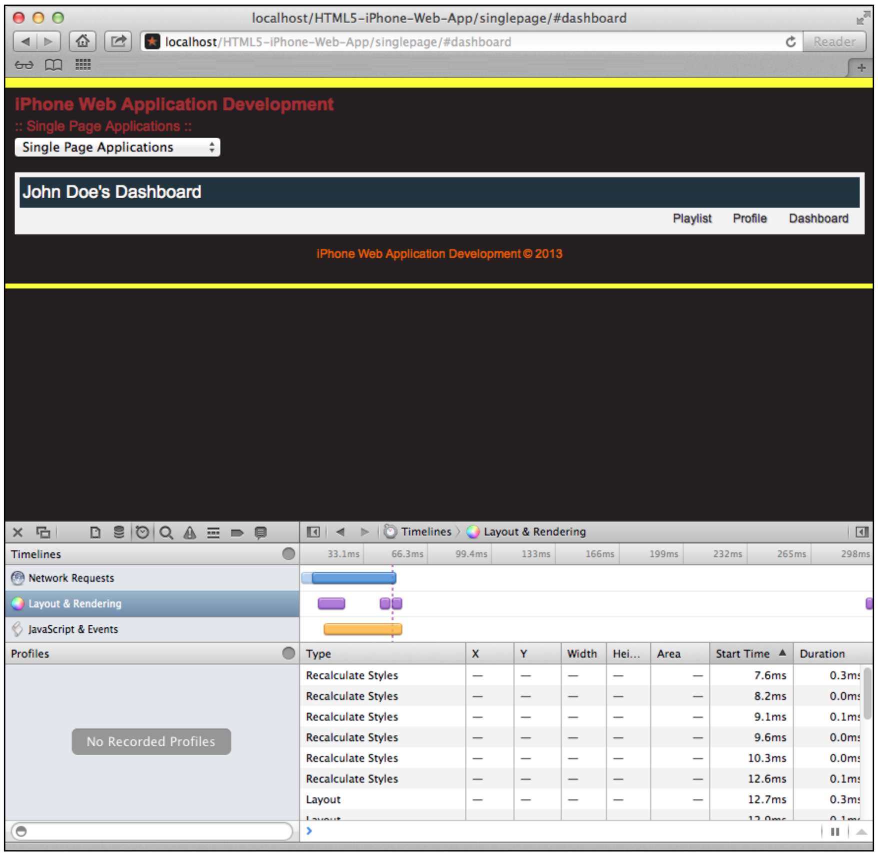
`https://github.com/stubbornella/csslint/wiki/IDE-integration`

Profiling our CSS

Previously it was extremely difficult to profile any of your CSS, in fact probably impossible. But with the advancements that have taken place in browser debugging tools, we are now able to profile stylesheets to some degree. In this section, we review how to take profiles of our styles and read the information presented to us in the Safari browser on Mac.

In the upcoming screens, we will briefly go over how profiling can be used for styles and how the Safari browser presents this information to us. We will only look at layout and rendering of our styles. Using the single-page application we built previously, we will look at the effectiveness of our styles and see the weaknesses and strengths of our styling in how it relates to the presentation layer of our application.

Let's begin by looking at the dashboard view of our single-page application with the Safari debugging tools opened and on the profile tab (clock symbol).



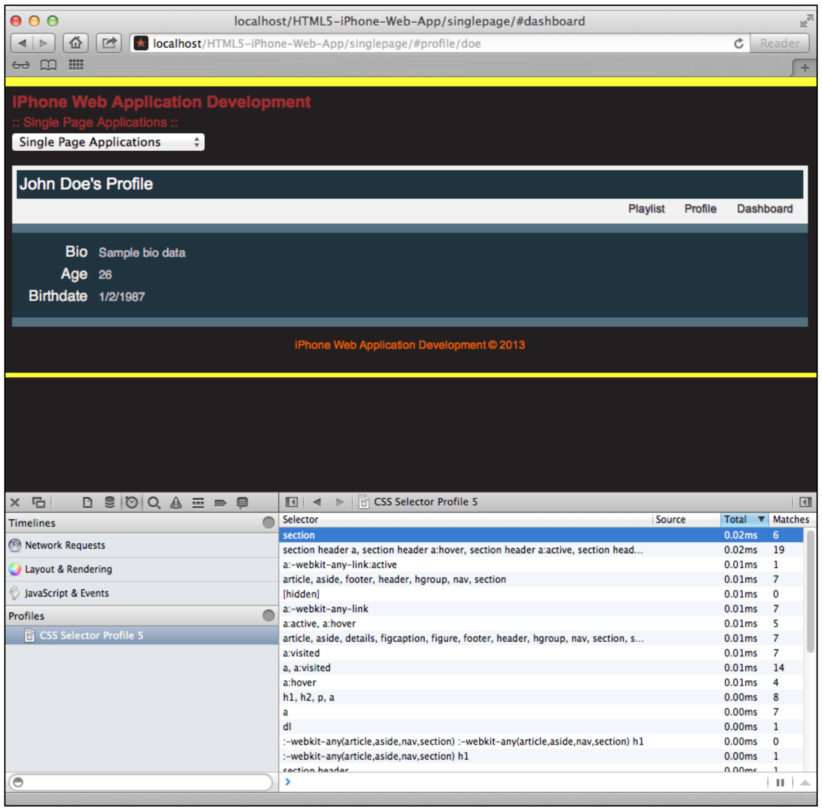
Type	X	Y	Width	Hei…	Area	Start Time ▲	Duration
Recalculate Styles	—	—	—	—	—	7.6ms	0.3ms
Recalculate Styles	—	—	—	—	—	8.2ms	0.0ms
Recalculate Styles	—	—	—	—	—	9.1ms	0.1ms
Recalculate Styles	—	—	—	—	—	9.6ms	0.0ms
Recalculate Styles	—	—	—	—	—	10.3ms	0.0ms
Recalculate Styles	—	—	—	—	—	12.6ms	0.1ms
Layout	—	—	—	—	—	12.7ms	0.3ms

Safari profiling tool

When we first load up our single-page application and view the profiling of the page load, we see three different timelines, including **Network requests**, **Layout & Rendering**, and **JavaScript & Events**. For our purposes, let's look at **Layout & Rendering**.

When we look at the **Layout & Rendering** timeline, we can see where repaints and restyle calculations were made on page load. The debugger also lets us know what type of process was run, when it was run, and the properties that were changed, including its start time and duration. This is all extremely helpful when looking for performance leaks on our page. But, what about runtime profiling? Well, the debugger has that to.

In fact, in our left-hand side bar there is a circle on the same line as the **Profiles** tab that allows us to profile either our JavaScript or CSS. This is great because when we enable this, we'll start profiling the application at run time. So, let's say we enable the profiling of our CSS and then, within our application, click on the **Profile** tab to switch page views; we'll definitely execute some changes in the view that would make our styles change. When we do this and stop our CSS profiling, we get the following results:



Selector	Source	Total	Matches
section		0.02ms	6
section header a, section header a:hover, section header a:active, section head...		0.02ms	19
a:-webkit-any-link:active		0.01ms	1
article, aside, footer, header, hgroup, nav, section		0.01ms	7
[hidden]		0.01ms	0
a:-webkit-any-link		0.01ms	7
a:active, a:hover		0.01ms	5
article, aside, details, figcaption, figure, footer, header, hgroup, nav, section, s...		0.01ms	7
a:visited		0.01ms	7
a, a:visited		0.01ms	14
a:hover		0.01ms	4
h1, h2, p, a		0.00ms	8
a		0.00ms	7
dl		0.00ms	1
:-webkit-any(article,aside,nav,section) :-webkit-any(article,aside,nav,section) h1		0.00ms	0
:-webkit-any(article,aside,nav,section) h1		0.00ms	1
section header		0.00ms	1

Safari profiling tool – runtime profiling

When we analyze our profile we see the selectors that are being utilized, the total time for it to render, the number of matches on the page, and its source. This is a great breakdown of what sort of processing took place and gives us a good idea of the amount of time each selector takes to find and render, giving us a good idea of what can be improved. Given, our application for this book is small, but if you were working on an application that included complex animations or the rendering of thousands of rows of data, this would come in useful in debugging your mobile application.

Once we have a good idea of what is bottlenecking our application, we need to do something about it. Having this information gives us critical information on the performance of our application and what we should focus on. The optimization phase is based on the problem(s) and needs of the project each team or individual faces, so in the next section we discuss some optimization techniques that can be used for faster rendering and matching of our styles.

Optimizing our CSS

In this section, we briefly go over some industry standards that help us optimize our applications rendering time by providing modular styles that are efficient, maintainable, and well crafted. These standards have been widely discussed by individuals and organizations well known in the industry and have ultimately been adopted in various frameworks. Of course, the standards discussed here may change as time progresses and browsers implement better processing methods that make new techniques faster and more efficient, but this should be a good starting guide for anyone looking to create stylesheets that meet today's demands.

Avoid universal rules

Do not use the `*` selector in your rules. These select every element in the DOM and thus their traversal method is inefficient.

For example, the following is extremely inefficient:

```
header > *
```

The preceding code is inefficient for one reason, it is using a universal selector. Because CSS reads from right to left, the engine is saying "let's pick up all elements and then see if they are immediately related to the header element." Because we need to traverse the entire DOM, the rendering of this selector is extremely slower than something like this:

```
header > h1
```

Don't qualify ID or Class rules

Qualifying an ID or Class involves directly attaching the tag name with the appropriate selector, but is extremely inefficient for the same reasons as the previous rule.

For example, all of the following selectors are bad:

```
input#name-text-field

.text-field#name-text-field

input.text-field

.text-field.address-text-field
```

Although some of these may seem tempting, they are unnecessary and inefficient. However, there is an exception here; if we want to change the style by adding a class to an element, then qualifying a class might be necessary. Either way, we could do the following to correct the preceding qualifying ID or classes.

```
#name-text-field

.text-field

.text-field.address-text-field
```

As it was mentioned in the preceding paragraph, the last selector might be more useful when changing the styles of an element based on a user action via JavaScript.

Never use !important

This rule is pretty self-explanatory. It is definitely tempting to use this to override styles, but don't; it will only cause headaches as your application(s) become more extensive. For this reason, check out the next rule.

Modularize styles

It's very easy to create styles that are generic to a web application or website; however, if we start thinking in terms of modularity, we start creating styles that are reserved for a part of that application. For example, take a form and its inputs, and let's say we wanted all forms on a site to contain text fields with a brown border. We could do the following:

```
form .text-field { border: 1px solid brown; }
```

Now we have reserved all fields with the class `.text-field` inside a `form` element to contain this style. So if any input fields with the class `.text-field` are outside this selector, we can then style them any way we want. Or on the other hand we could override the style like so:

```
form .personal-information .text-field { border: 1px solid blue; }
```

Now, if we include this style after the original one, it would take precedence because we are actually using cascading principles that make our styling more efficient and manageable.

> Keep in mind that descendant selectors are the most expensive kind of selector. However they are extremely versatile, and therefore we should not sacrifice maintainability or semantics for efficient CSS.

In most cases, these rules should be enough, but you will most likely find it useful to implement some of the other best practices that have been written about in the industry. Of course, you should work with the practices that have been adopted for the framework you use, or better yet, fit your team. I have found these to be extremely helpful and a great starting point, and I encourage you to research and experiment as you see fit. Now, let's take a look at how JavaScript could be optimized for our applications.

Optimizing JavaScript

Now that we have covered the optimization of stylesheets, let's look at our scripts. JavaScript has also had a history of being dropped onto the page without any forethought or planning, and in general has led to bad reputation for the language. But again, because of the complex nature of web applications, the open source community has helped shape the language.

Throughout the book, we have adopted several industry standards, including namespaces, closures, caching variables, and so on. However, it is also essential we validate and profile our scripts so they can be optimized. In this section, we will go through this and hopefully cover the major points needed to make high-performance mobile applications.

Validating JavaScript using JSLint

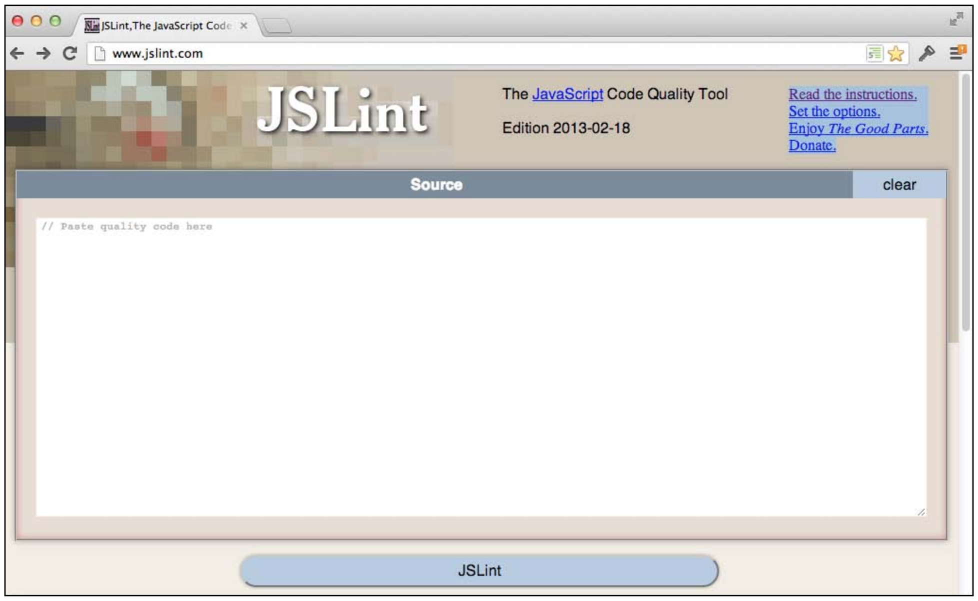
In recent years, various tools have come out to help us validate our JavaScript. Tools such as JSLint and JSHint have been created to help us as we code, similar to CSS Lint. But why should we use these tools, especially for JavaScript? JSLint's site (`http://www.jslint.com/lint.html`) mentions the reasoning behind the tool:

> *JavaScript is a young-for-its-age language. It was originally intended to do small tasks in webpages, tasks for which Java was too heavy and clumsy. But JavaScript is a surprisingly capable language, and it is now being used in larger projects. Many of the features that were intended to make the language easy to use are troublesome when projects become complicated. A lint for JavaScript is needed: JSLint, a JavaScript syntax checker and validator.*

JSLint's website also mentions the following:

> *JavaScript is a sloppy language, but inside it there is an elegant, better language. JSLint helps you to program in that better language and to avoid most of the slop. JSLint will reject programs that browsers will accept because JSLint is concerned with the quality of your code and browsers are not. You should accept all of JSLint's advice.*

To test out our JavaScript, we can easily visit JSLint's website (`http://www.jslint.com/`):

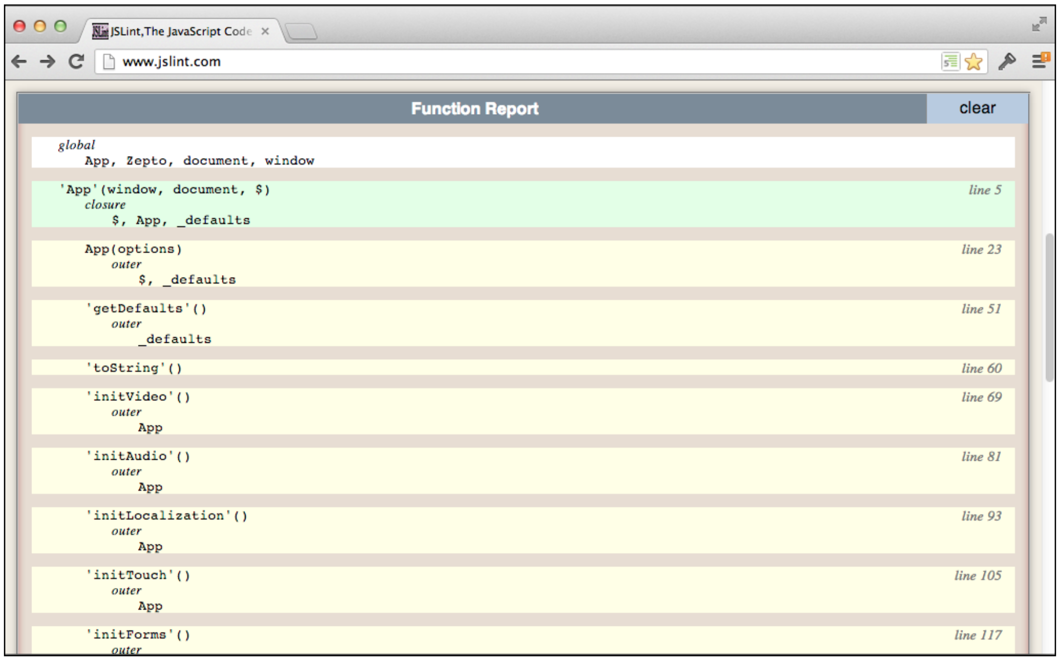


The JSLint website

As you can see, JSLint is very similar to CSS Lint, in such a way that all you really need to do is input your JavaScript onto the page and the results will be displayed to you. So let's check out what a successful and an unsuccessful output would look like.

Validating a successful example using JSLint

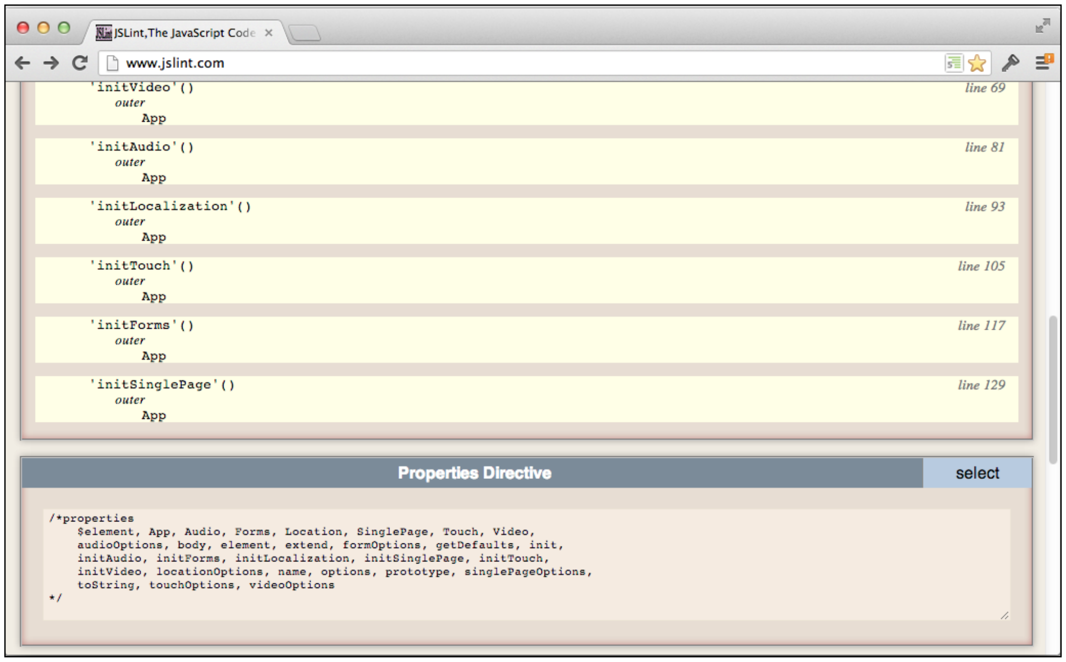
In our example, we will utilize our `App.js` JavaScript to test out with the JSLint utility. When we run this file, a successful output will detail out the methods, variables, and properties used throughout the closure. Let's look at the following screenshots:



```
global
    App, Zepto, document, window
'App'(window, document, $)                                  line 5
    closure
        $, App, _defaults
    App(options)                                            line 23
        outer
            $, _defaults
    'getDefaults'()                                         line 51
        outer
            _defaults
    'toString'()                                            line 60
    'initVideo'()                                           line 69
        outer
            App
    'initAudio'()                                           line 81
        outer
            App
    'initLocalization'()                                    line 93
        outer
            App
    'initTouch'()                                           line 105
        outer
            App
    'initForms'()                                           line 117
        outer
```

Successful validation using JSLint – methods and variables

The preceding example is the top view of a successful validation using JSLint. The validator will return to us a list beginning with a listing of all global objects. It will then continue on with a listing of the methods, variables, and some detail about each. For example, `initVideo` returns `this` or an instance of `App` and so on.



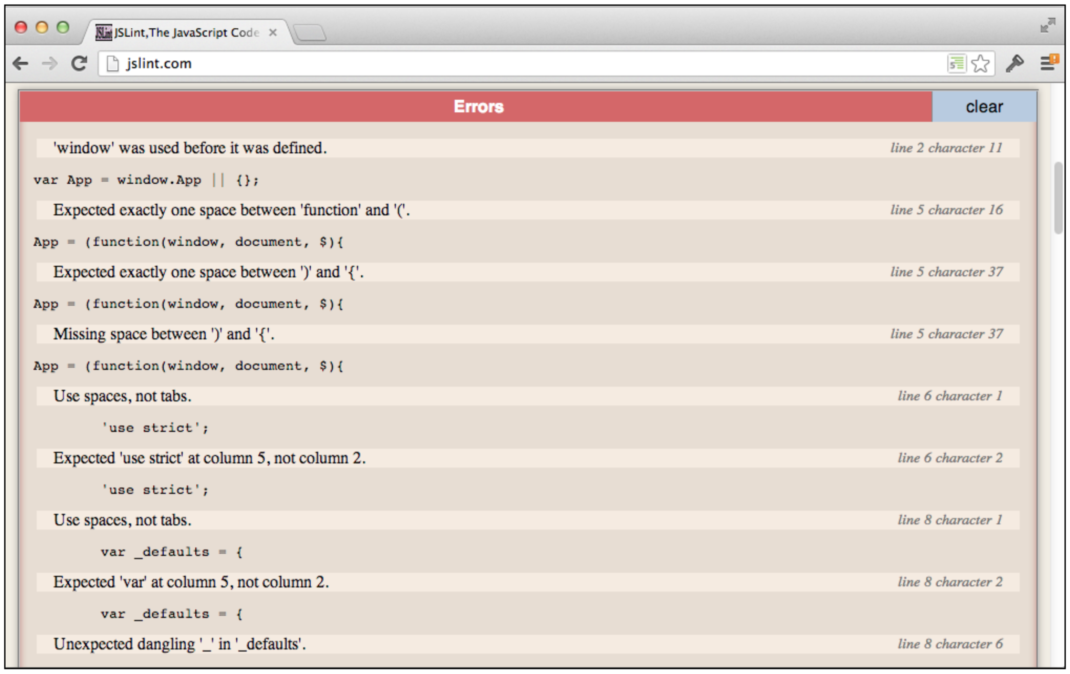
```
'initVideo'()                                   line 69
    outer
        App
'initAudio'()                                   line 81
    outer
        App
'initLocalization'()                            line 93
    outer
        App
'initTouch'()                                   line 105
    outer
        App
'initForms'()                                   line 117
    outer
        App
'initSinglePage'()                              line 129
    outer
        App
```

```
/*properties
    $element, App, Audio, Forms, Location, SinglePage, Touch, Video,
    audioOptions, body, element, extend, formOptions, getDefaults, init,
    initAudio, initForms, initLocalization, initSinglePage, initTouch,
    initVideo, locationOptions, name, options, prototype, singlePageOptions,
    toString, touchOptions, videoOptions
*/
```

Successful validation using JSLint – properties

Validating an unsuccessful example

Taking the same example as the previous one without modifying the JSLint options would produce several errors. These errors are mostly whitespace, spacing, and global objects that the processor is not aware of.



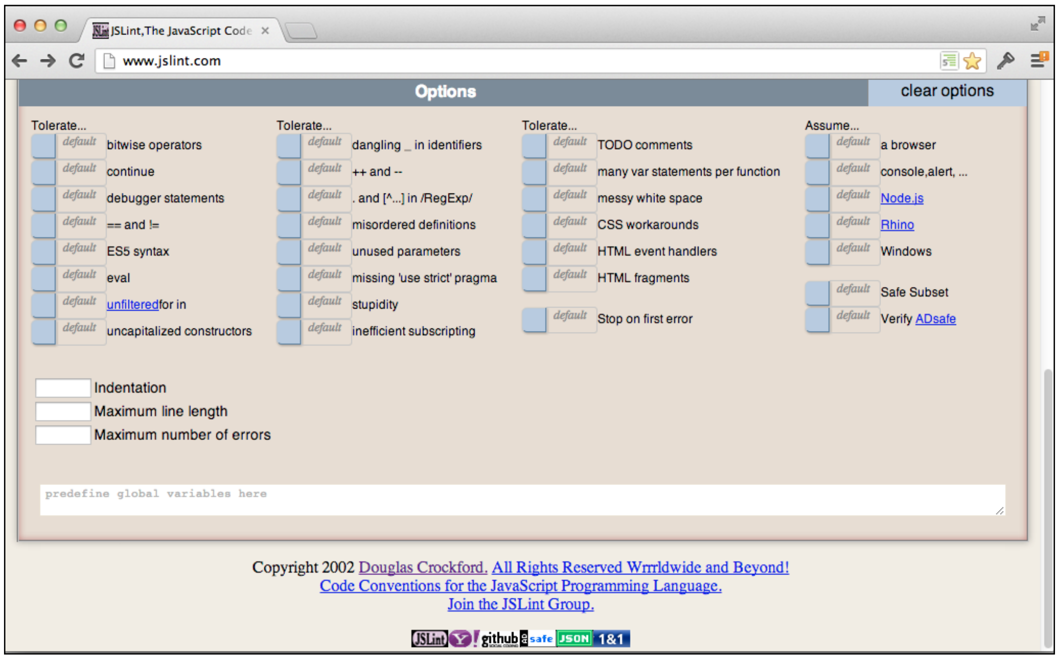
```
'window' was used before it was defined.                              line 2 character 11
var App = window.App || {};
Expected exactly one space between 'function' and '('.                line 5 character 16
App = (function(window, document, $){
Expected exactly one space between ')' and '{'.                       line 5 character 37
App = (function(window, document, $){
Missing space between ')' and '{'.                                    line 5 character 37
App = (function(window, document, $){
Use spaces, not tabs.                                                 line 6 character 1
        'use strict';
Expected 'use strict' at column 5, not column 2.                      line 6 character 2
        'use strict';
Use spaces, not tabs.                                                 line 8 character 1
        var _defaults = {
Expected 'var' at column 5, not column 2.                             line 8 character 2
        var _defaults = {
Unexpected dangling '_' in '_defaults'.                               line 8 character 6
```

JSLint – unsuccessful validation

Based on the preceding output, the errors are listed in red with a description, sample code, and line number of where the error occurs, allowing you to easily debug your application. Now, let's say we didn't want whitespace or spacing to actually affect the outcome of our validation; then we could customize the options for JSLint.

Customizable options

As with most of the tools we are discussing in this chapter, JSLint also comes packaged with options that can be customized to fit our needs. Let's briefly review some of the options that are available to us via the website.



JSLint,The JavaScript Code
www.jslint.com

Options | clear options

Tolerate...	Tolerate...	Tolerate...	Assume...
default bitwise operators	default dangling _ in identifiers	default TODO comments	default a browser
default continue	default ++ and --	default many var statements per function	default console,alert, ...
default debugger statements	default . and [^...] in /RegExp/	default messy white space	default Node.js
default == and !=	default misordered definitions	default CSS workarounds	default Rhino
default ES5 syntax	default unused parameters	default HTML event handlers	default Windows
default eval	default missing 'use strict' pragma	default HTML fragments	default Safe Subset
default unfilteredfor in	default stupidity	default Stop on first error	default Verify ADsafe
default uncapitalized constructors	default inefficient subscripting		

Indentation
Maximum line length
Maximum number of errors

predefine global variables here

Copyright 2002 Douglas Crockford. All Rights Reserved Wrrldwide and Beyond!
Code Conventions for the JavaScript Programming Language.
Join the JSLint Group.

JSLint – the Options screen

The options that are available to us are extensive, from whitespace formatting to the correctness of the toleration of `TODO` comments that we all put in our JavaScript. Of course, some of these options may not fit our needs at the time of testing, but in general, they are very helpful in keeping with a consistent coding standard that delivers cross-platform valid scripts.

Integrating JSLint

Similar to CSS Lint, JSLint can be used in your IDE or text editor—whichever environment you prefer. Many vendors have already created plugins or extensions to these tools to allow you to easily lint your code as you type or save. For example, Sublime Text has a `SublimeLinter` package that comes with CSS Lint, JSLint, and also some other tools that can help you code more efficiently. How is this possible?

> *JSLint can be run anywhere that JavaScript (or Java) can run. See for example* `https://github.com/douglascrockford/JSLint/wiki/JSLINT`.

Refer to the following for more details:

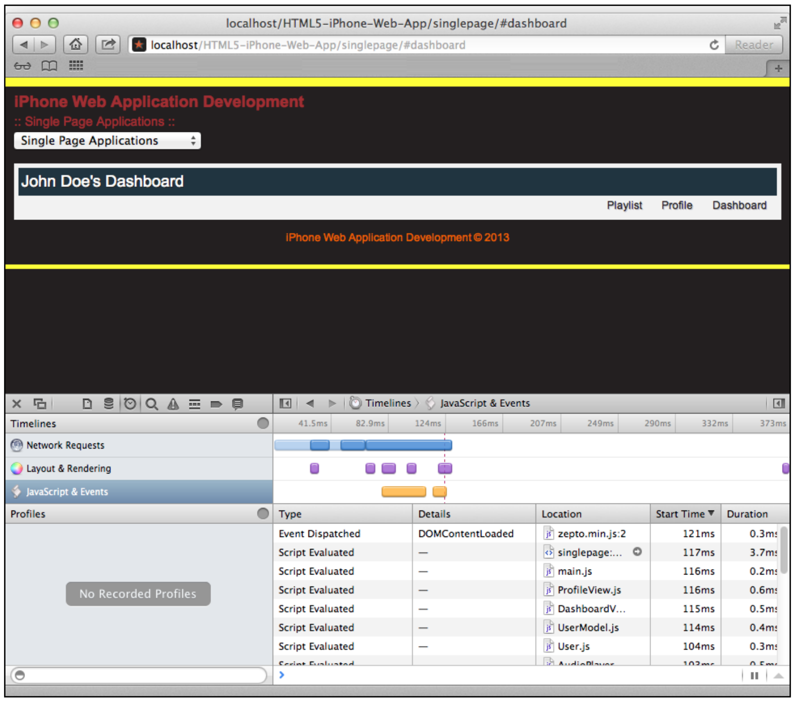
`https://github.com/douglascrockford/JSLint`

In its essence, JSLint is a JavaScript method that can be passed in code and is then evaluated by JavaScript itself, making it extremely efficient to process your code and to integrate into other environments. So if it doesn't already exist for your text editor or IDE, you can easily create an extension that helps you code with quality using JSLint.

Profiling our JavaScript

As with CSS profiling, testing the performance of your JavaScript was extremely hard in the old days of the Web. However, we don't really need to worry about it too much these days, since almost every browser debugger has implemented a way to profile your scripts. Using Safari's built-in debugging tool, we'll check out how to debug our application's script performance.

In the following examples, we will just go over the profiling of the JavaScript on the single-page application we built previously, similar to what we did for the profiling of our styles in the previous section.

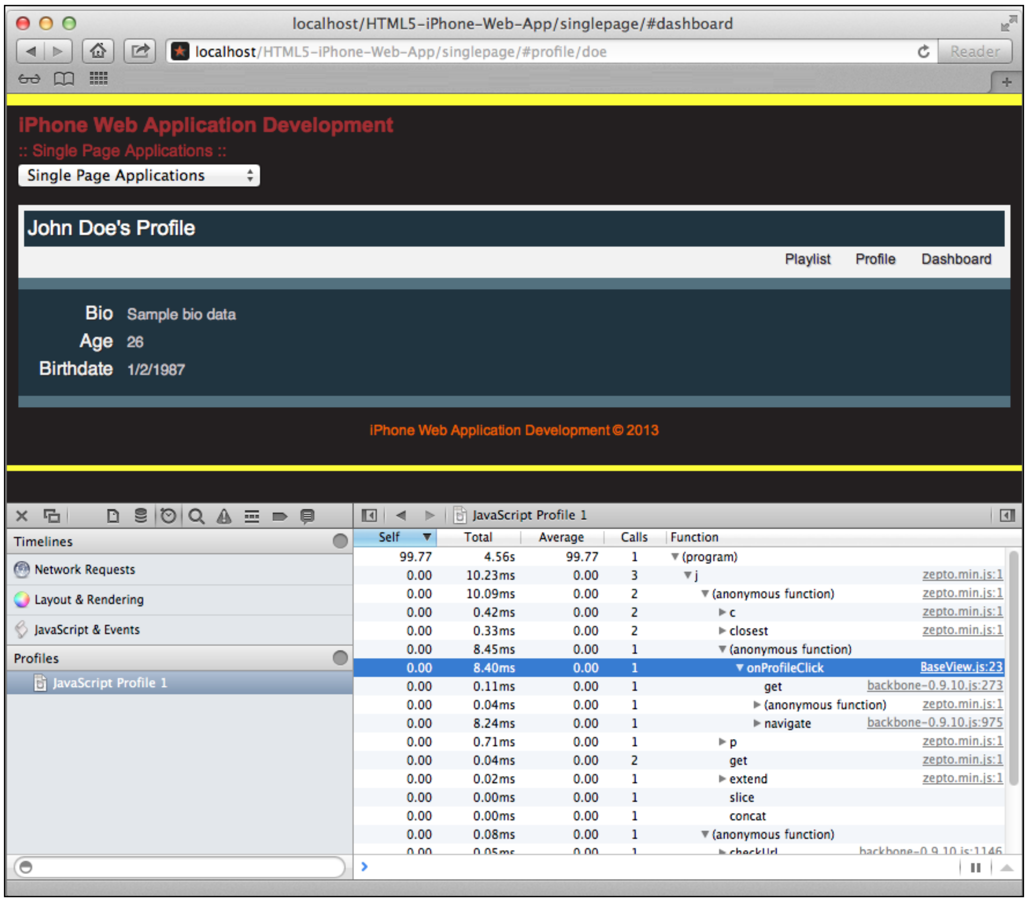


Type	Details	Location	Start Time ▼	Duration
Event Dispatched	DOMContentLoaded	zepto.min.js:2	121ms	0.3ms
Script Evaluated	—	singlepage:...	117ms	3.7ms
Script Evaluated	—	main.js	116ms	0.2ms
Script Evaluated	—	ProfileView.js	116ms	0.6ms
Script Evaluated	—	DashboardV...	115ms	0.5ms
Script Evaluated	—	UserModel.js	114ms	0.4ms
Script Evaluated	—	User.js	104ms	0.3ms

Safari profiling tool – JavaScript

The preceding screenshot is a review of the scripts on page load. When we review the **JavaScript & Events** timeline, we are given a breakdown of the type, details, location, start time, and duration of each script, contributing to the scripts timeline outcome. Although the start time is something which we definitely want to know in order to see what might be blocking scripts (other scripts), duration is probably even more important because each script can block the process of page rendering if it is not brought in asynchronously.

On top of viewing the performance of the page-load impact from scripts, we can also profile the functionality that our scripts perform. For example, let's say we wanted to detect how our methods are performing when we click on the **Profile** button within our application. Well this can be easily accomplished using the same technique as profiling your CSS, by clicking the circle in the **Profile** tab and enabling profiling for our JavaScript; we will be able to see all the methods called and their performance.


localhost/HTML5-iPhone-Web-App/singlepage/#dashboard
localhost/HTML5-iPhone-Web-App/singlepage/#profile/doe
Reader
iPhone Web Application Development
:: Single Page Applications ::
Single Page Applications
John Doe's Profile
Playlist
Profile
Dashboard
Bio
Sample bio data
Age
26
Birthdate
1/2/1987
iPhone Web Application Development © 2013
Timelines
Network Requests
Layout & Rendering
JavaScript & Events
Profiles
JavaScript Profile 1

```
JavaScript Profile 1
     Self      Total   Average  Calls  Function
    99.77      4.56s     99.77    1    ▼ (program)
     0.00    10.23ms      0.00    3      ▼ j                                    zepto.min.js:1
     0.00    10.09ms      0.00    2        ▼ (anonymous function)               zepto.min.js:1
     0.00     0.42ms      0.00    2          ▶ c                                zepto.min.js:1
     0.00     0.33ms      0.00    2          ▶ closest                          zepto.min.js:1
     0.00     8.45ms      0.00    1          ▼ (anonymous function)
     0.00     8.40ms      0.00    1            ▼ onProfileClick                 BaseView.js:23
     0.00     0.11ms      0.00    1                get                 backbone-0.9.10.js:273
     0.00     0.04ms      0.00    1              ▶ (anonymous function)         zepto.min.js:1
     0.00     8.24ms      0.00    1              ▶ navigate            backbone-0.9.10.js:975
     0.00     0.71ms      0.00    1          ▶ p                                zepto.min.js:1
     0.00     0.04ms      0.00    2            get                              zepto.min.js:1
     0.00     0.02ms      0.00    1          ▶ extend                           zepto.min.js:1
     0.00     0.00ms      0.00    1            slice
     0.00     0.00ms      0.00    1            concat
     0.00     0.08ms      0.00    1        ▼ (anonymous function)
     0.00     0.05ms      0.00    1          ▶ checkUrl           backbone-0.9.10.js:1146
```

Safari profiling tool – JavaScript runtime

Based on our preceding use case, we can detail out the performance of our application pretty easily. From what we can gather in this example, our `onProfileClick` event takes about 8.40 ms to execute, and is called once. However, the larger picture is that we can see all the methods being called and the order of this execution—great information that can be used to help detect memory leaks and performance optimization that is necessary for our application.

You can see from these very basic examples that debugging our application for performance is easier than ever before. We can profile our JavaScript, find out how our application is working, and the efficiency of our code. But now that we have this information, what can we do to improve our codebase? Well this is what we tackle in the next section, general optimization tips that we can all use to make our application perform better without sacrificing code quality.

Optimizing our JavaScript

JavaScript is highly extensible, allowing us to do almost anything we want—which is great, but can also be extremely harmful. For example, you can easily forget about using the keyword `var` in front of your variables. However, we don't want to do this because it will make our variables available at the global scope, which may cause conflicts with other scripts that might use that exact same variable name. We can also easily wrap our JavaScript in a `try...catch` statement, not exactly the best practice because we're not figuring out what the problem is. Or if we wanted to, we could `eval` or evaluate a string of JavaScript pretty easily without any error checking.

For this reason, the industry has adopted multiple best practices that are true and tested, implemented by the most used open source libraries out there, including jQuery, Backbone, and Underscore. In this section, we briefly go over the practices I have based the book on and those I believe are critical to the success of any application.

Avoid globals

It is extremely easy and tempting to create all your variables and functions in the global scope, or outside of the closures we created in our applications. But don't do that; it's a terrible idea and looked down upon in the community for several reasons. For example, if a variable is kept in the global scope, it must be maintained during the entire execution of your application, decreasing the performance of your application.

So instead of doing this:

```
Var Modal = function(){};
```

You should be doing this:

```
(function(){ function Modal(){} }());
```

The preceding technique is very similar to what we have been doing all along. In fact, this is what we called closures or **immediately invoked function expression** (**IIFE**). When we wrap a method inside the parenthesis and then invoke it using `()`, we are calling that method immediately and creating a new containing scope, so anything inside those parenthesis is not available at the global scope, making our code a bit more manageable.

Leave the DOM alone

Well, we probably won't be doing that, but we should definitely keep it to a minimum. Accessing the DOM is expensive and is an issue when it comes to application performance. So let's take a use case, such as updating a list of information.

Avoid doing this:

```
var $list = $('ul');

for (var i; i < 100; i++) {
  var li = '<li>' + i + '</li>';

  $list.append(li);
}
```

Instead, you should do this:

```
var $list = $('ul'),
  liArray = [];

for (var i; i < 100; i++) {
  liArray.push('<li>' + i + '</li>');
}

$list.append(liArray.join(''));
```

The difference between the two is that the former touches the DOM each time we create a list item, while the latter pushes each item to an array, and when it comes to appending, joins the array with an empty string, touching the DOM only once.

Use literals

This can be seen throughout our entire codebase for this book. This is more efficient since we don't use the `new` keyword. For example, instead of declaring a new variable via the new keyword, we just use the `Array` literal, like so:

```
var arr = []; // not new Array();
var str = ''; // not new String('');
```

Modularize functionality

To keep code modular, you want to make sure that each function or class has a specific set of functionality it is supposed to achieve. Most of the time, each function should probably be about 10 to 15 lines of code that achieves a certain goal.

For example, you may write the following bit of functionality:

```
function init() {
  var $list = $('ul'),
    liArray = [];

  for (var i; i < 100; i++) {
    liArray.push('<li>' + i + '</li>');
  }

  $list.append(liArray.join(''));

  $list.on('click', function() {
    // do something
  });
}
```

Instead of writing the preceding code, we can do this:

```
function populateLists() {
  var $list = $('ul'),
    liArray = [];

  for (var i; i < 100; i++) {
    liArray.push('<li>' + i + '</li>');
  }

  $list.append(liArray.join(''));

  return $list;
}

function attachListsEvents($list) {
  $list.on('click', doSomething);
}
```

```
function doSomething(e) {
  // do something
}

function init() {
  var $list = populateLists();

  attachListsEvents($list);
}
```

As you can see, the code has been modularized to do specific sets of functionality, allowing us to create methods that run a specific set of instructions that each method, by name, describes. This is great for maintaining our codebase while delivering effective functionality.

Summary

In this chapter, we have considered performance optimization for various parts of our applications, including styles, scripts, and media. We went over validating, optimizing, and profiling our styles and scripts. In addition, we briefly covered how we can optimize our media, including images, audio, and video. Now that we have firm understanding of the techniques used throughout the book to optimize our applications, in the next chapter, we will take a look at frameworks that can help us deliver native applications using HTML5, CSS3, and JavaScript.

10
Creating a Native iPhone Web Application

In this chapter, we look at transferring our native applications for iOS Safari to a native environment using the PhoneGap framework. We'll dive into setting up our development environment, including the setup of the Xcode IDE and use of the iOS Simulator. We'll build a `HelloWorld` example to demonstrate how easy it is to get going quickly, and then transfer our single-page application, built in *Chapter 7, One-page Applications*. Once we have a solid foundation on native application development, we'll enhance the single-page application by tying into native functionality using PhoneGap's Contacts API to bring in our contacts and display some of their information.

We aim to help you achieve a consistent look and feel for your native application using a single codebase. The goals here are to get you started on native application development using the programming languages you have grown to love and understand for the Web. With that in mind, let's start by setting up our development environment.

In this chapter, we will cover:

- Xcode installation
- Using the iOS Simulator
- Implementing PhoneGap
- Creating a `HelloWorld` example
- Transferring a current application, including CSS, JavaScript, HTML, and assets
- Tying into native functionality using the Contacts API in PhoneGap

Setting up the development environment

As with any workflow to creating software, our development environment is critical. So, let's take some time setting up the environment preferred by many engineers to create native applications. In this section, we'll go over the installation of Xcode, and an overview of the Integrated Development Environment (IDE). We'll continue by setting up the PhoneGap framework and lastly look at how the iOS Simulator plays a critical roll in testing our app. As a bonus, we'll look at configuring our application to fit our needs in this chapter. So let's get started.

Getting started with Xcode

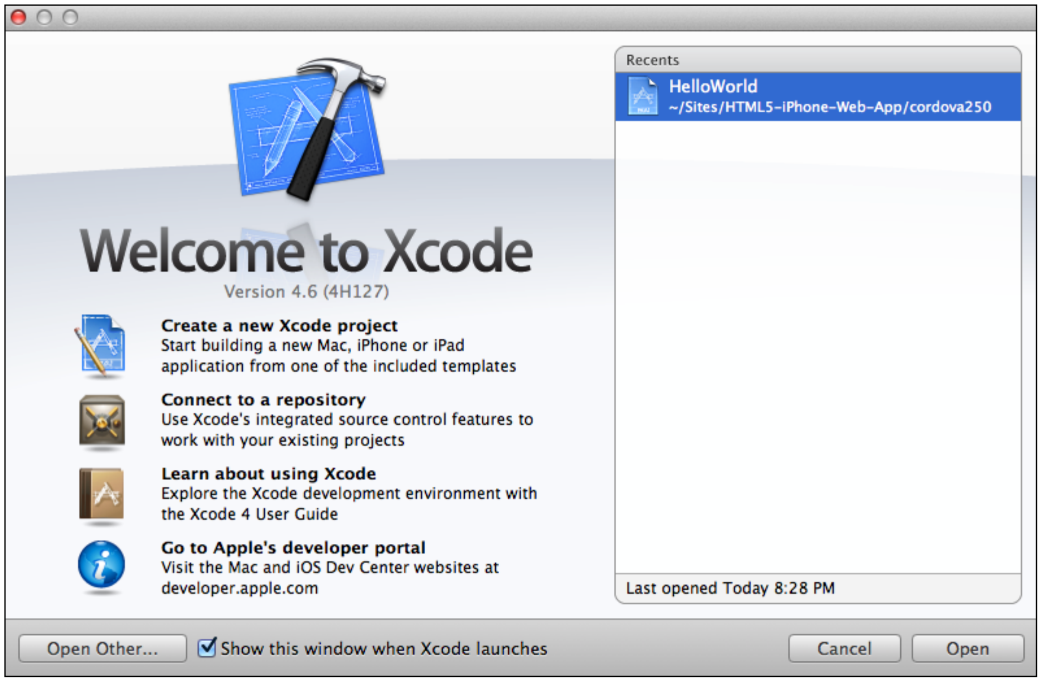
Xcode is the preferred IDE for native application development on the iOS operating system, as it is actively supported by Apple and specifically tailored to the OS X and iOS operating systems. The IDE is provided by Apple and can be used to create Mac OS X, iPhone, and iPad applications. While it can also be used for other various types of development, these three platforms are most commonly associated with Xcode. By default, your Mac does not come with Xcode, so we'll need to install it.

Installing Xcode

Luckily for us, Xcode is extremely easy to install. We can install the IDE through the Mac App Store (`https://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12`). When the installation is complete, we will have various pieces of software installed on our computer, including the Instruments analysis tool, iOS Simulator, and the latest **Mac OS X and iOS software development kit** (**SDK**).

Xcode IDE overview – the basics

By default, the Xcode IDE gets installed in the applications directory; launch it by double-clicking on the icon displayed. The icon is a hammer that is diagonally located over a blue technical drawing that has a pencil, brush, and ruler forming the letter A. When the application launches, we will see the welcome screen.


Welcome to Xcode

Version 4.6 (4H127)

Create a new Xcode project
Start building a new Mac, iPhone or iPad application from one of the included templates

Connect to a repository
Use Xcode's integrated source control features to work with your existing projects

Learn about using Xcode
Explore the Xcode development environment with the Xcode 4 User Guide

Go to Apple's developer portal
Visit the Mac and iOS Dev Center websites at developer.apple.com

Recents

HelloWorld
~/Sites/HTML5-iPhone-Web-App/cordova250

Last opened Today 8:28 PM

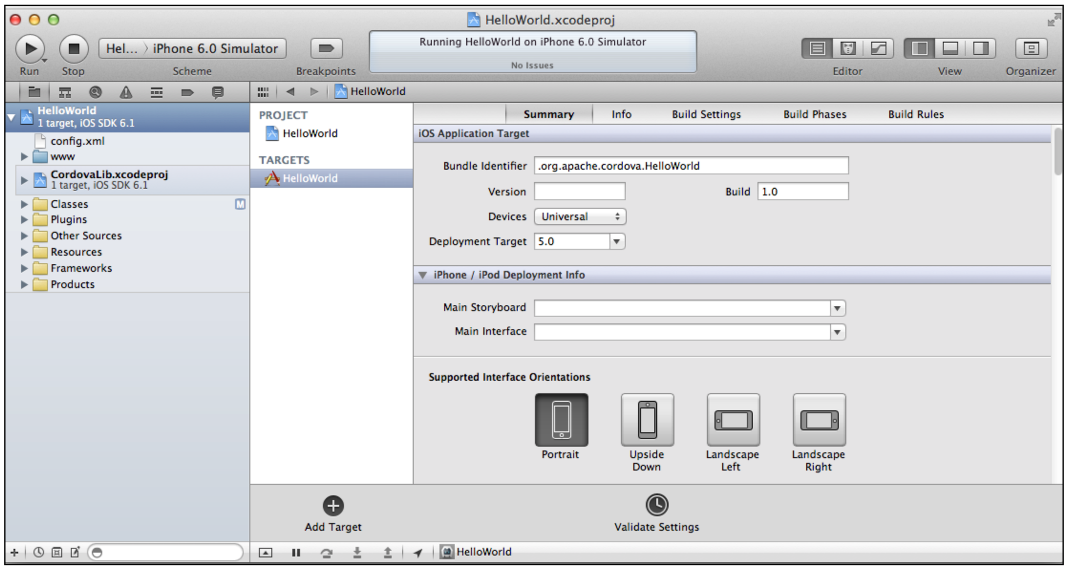
The welcome screen

This is the welcoming screen for Xcode, and it lists out recent projects and the ability to create a new project, connect to a repository, learn about Xcode, or check out the Apple developer portal. On your screen you will most likely not have the `HelloWorld` project listed in the preceding screenshot, which is what we will be building, and if it's your first time this should be empty.

Because this section is to get us familiar with Xcode itself, don't hesitate about the next few screens. The next screens are what we will be building out, but are only meant to help us recognize certain parts of the Xcode application to make it easier for you to use.

The Xcode workspace

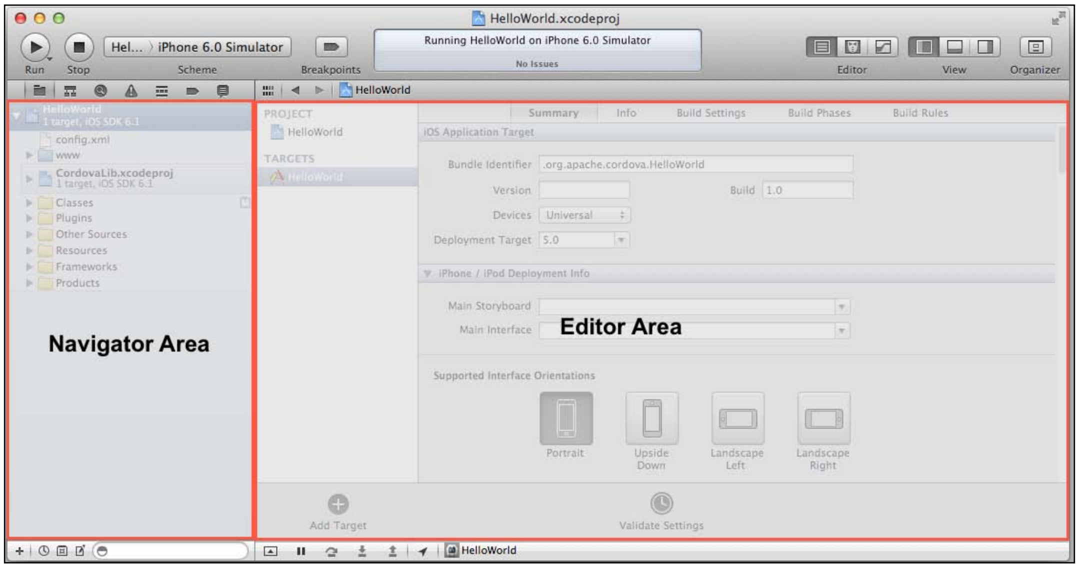
Now, let's go over the user interface of Xcode in order to understand how we can leverage this powerful tool. First of all, as we already know, we are introduced to the welcome screen when we open the application. You can choose to disable this feature by unchecking the **Show this window when Xcode launches** checkbox on the welcome screen. But when we have a project that we have created open, this is what it looks like:



Bundle Identifier .org.apache.cordova.HelloWorld
Version
Build 1.0
Devices Universal
Deployment Target 5.0

The project display

Looks pretty simple right? Well that's good, because this is known as the workspace and this is critical because Xcode aims to have all development efforts be located within one central window in the IDE, helping us consolidate and speed up our development process. But recognize the two critical aspects of this workspace: the Navigator Area located to the left where all of our files are located, and then the Editor Area where we can edit the project we are in.



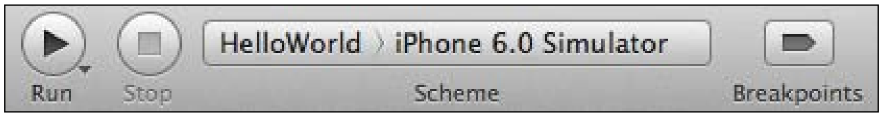
Bundle Identifier .org.apache.cordova.HelloWorld
Version
Build 1.0
Devices Universal
Deployment Target 5.0
Main Storyboard
Main Interface

The Navigator and Editor Areas

The preceding screenshot helps demonstrate the two most critical aspects of Xcode when it comes to developing an application. Keep in mind that, depending on the file selected, your Editor Area will change. For example, in the preceding screenshots we have a GUI that allows us to set properties of our project.

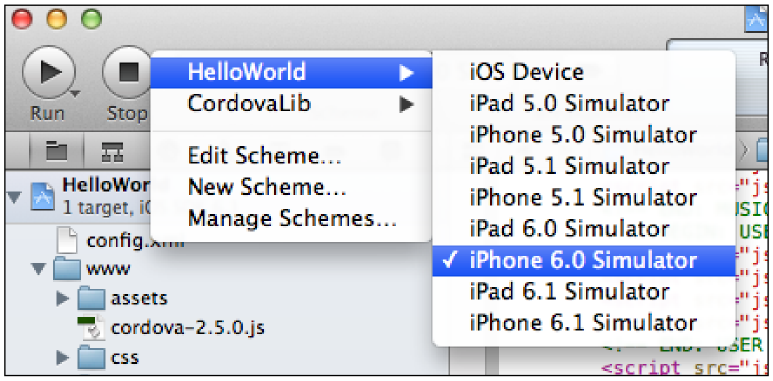
The Xcode toolbar

The toolbar in Xcode has various features that we'll use constantly when developing native applications. For example, in the following screenshot, there are **Run**, **Stop**, and **Breakpoints** buttons, including a **Scheme** selector. These actions are extremely important when debugging your application(s). The **Run** button does exactly what it says, it runs your application. **Stop** on the other hand will stop all activities with the running application. And the **Breakpoints** button will display our breakpoints in the Editor Area.



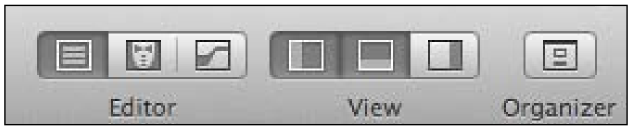
The toolbar displaying Run, Scheme, and Breakpoints

The **Scheme** selector lets you choose which application to test and in what environment to test it. In our example application, `HelloWorld` will be tested using the iPhone 6.0 Simulator, but we have various options to choose from. Looking at the following screenshot, we can see that, if installed, we can test our application using the iPad Simulator and various versions of it, and the iPhone Simulator.



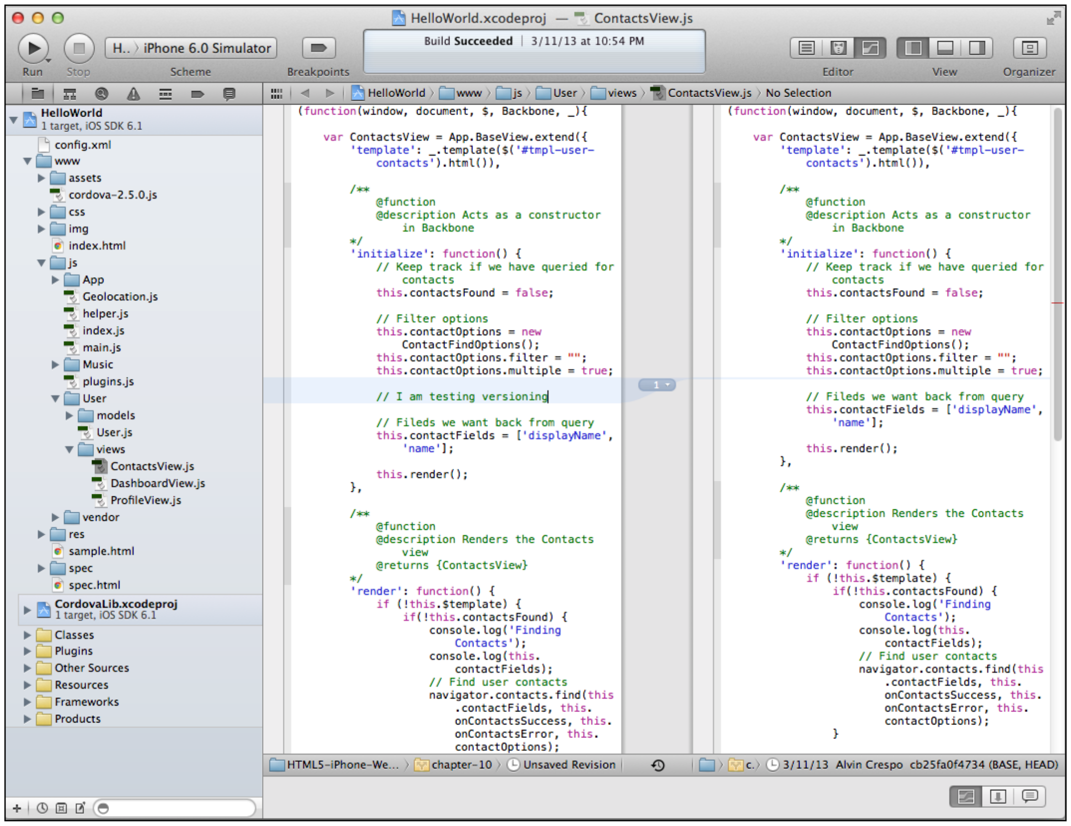
The toolbar Scheme selector

The toolbar also has various actions to the right of the IDE, including the editor views, general views, and organizer. The default editor view is a text editor component, allowing us to do basic editing of our source files. The middle editor view is the assistant editor, which we will not be covering. The last editor view is the version editor.



The toolbar project display options

The version editor is great for our use as developers, allowing us to edit files and see the versioning happening immediately. For example, in the following screenshot we can see that a comment was added, and that the original versioned file is informing the user of where the changes have been made, allowing us to see live edits of the same file.

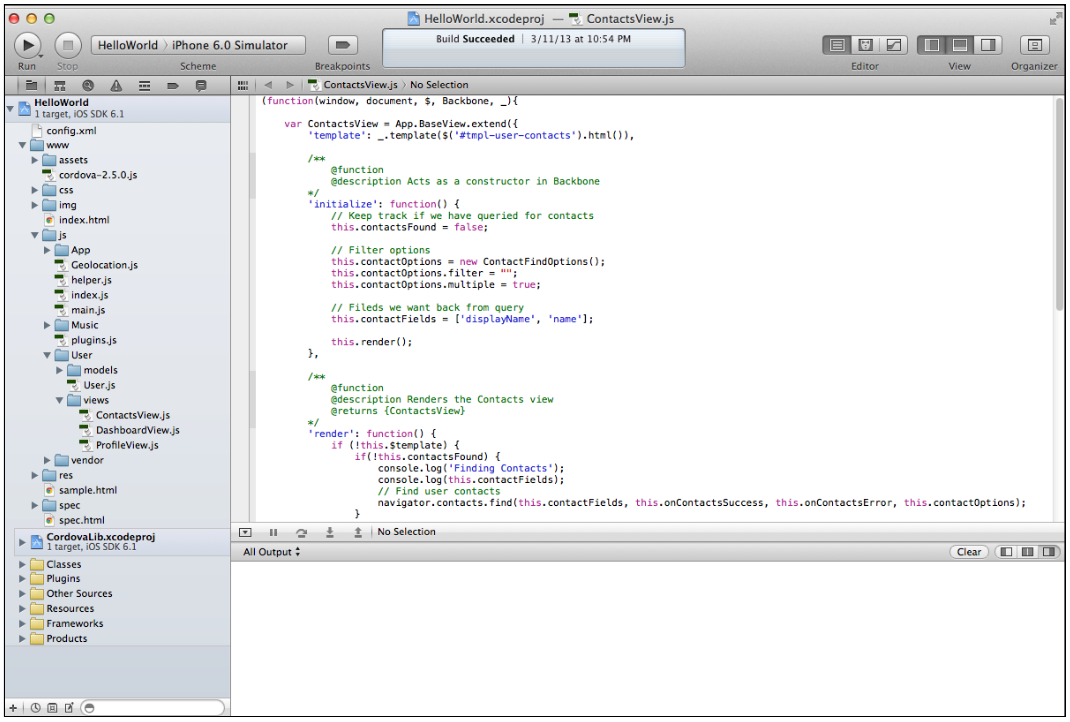


```
(function(window, document, $, Backbone, _){

    var ContactsView = App.BaseView.extend({
        'template': _.template($('#tmpl-user-
            contacts').html()),

        /**
            @function
            @description Acts as a constructor
                in Backbone
        */
        'initialize': function() {
            // Keep track if we have queried for
                contacts
            this.contactsFound = false;

            // Filter options
            this.contactOptions = new
                ContactFindOptions();
            this.contactOptions.filter = "";
            this.contactOptions.multiple = true;

            // I am testing versioning

            // Fileds we want back from query
            this.contactFields = ['displayName',
                'name'];

            this.render();
        },

        /**
            @function
            @description Renders the Contacts
                view
            @returns {ContactsView}
        */
        'render': function() {
            if (!this.$template) {
                if(!this.contactsFound) {
                    console.log('Finding
                        Contacts');
                    console.log(this.
                        contactFields);
                    // Find user contacts
                    navigator.contacts.find(this
                        .contactFields, this.
                        onContactsSuccess, this.
                        onContactsError, this.
                        contactOptions);
```

```
(function(window, document, $, Backbone, _){

    var ContactsView = App.BaseView.extend({
        'template': _.template($('#tmpl-user-
            contacts').html()),

        /**
            @function
            @description Acts as a constructor
                in Backbone
        */
        'initialize': function() {
            // Keep track if we have queried for
                contacts
            this.contactsFound = false;

            // Filter options
            this.contactOptions = new
                ContactFindOptions();
            this.contactOptions.filter = "";
            this.contactOptions.multiple = true;

            // Fileds we want back from query
            this.contactFields = ['displayName',
                'name'];

            this.render();
        },

        /**
            @function
            @description Renders the Contacts
                view
            @returns {ContactsView}
        */
        'render': function() {
            if (!this.$template) {
                if(!this.contactsFound) {
                    console.log('Finding
                        Contacts');
                    console.log(this.
                        contactFields);
                    // Find user contacts
                    navigator.contacts.find(this
                        .contactFields, this.
                        onContactsSuccess, this.
                        onContactsError, this.
                        contactOptions);
                }
```

The project versioning display

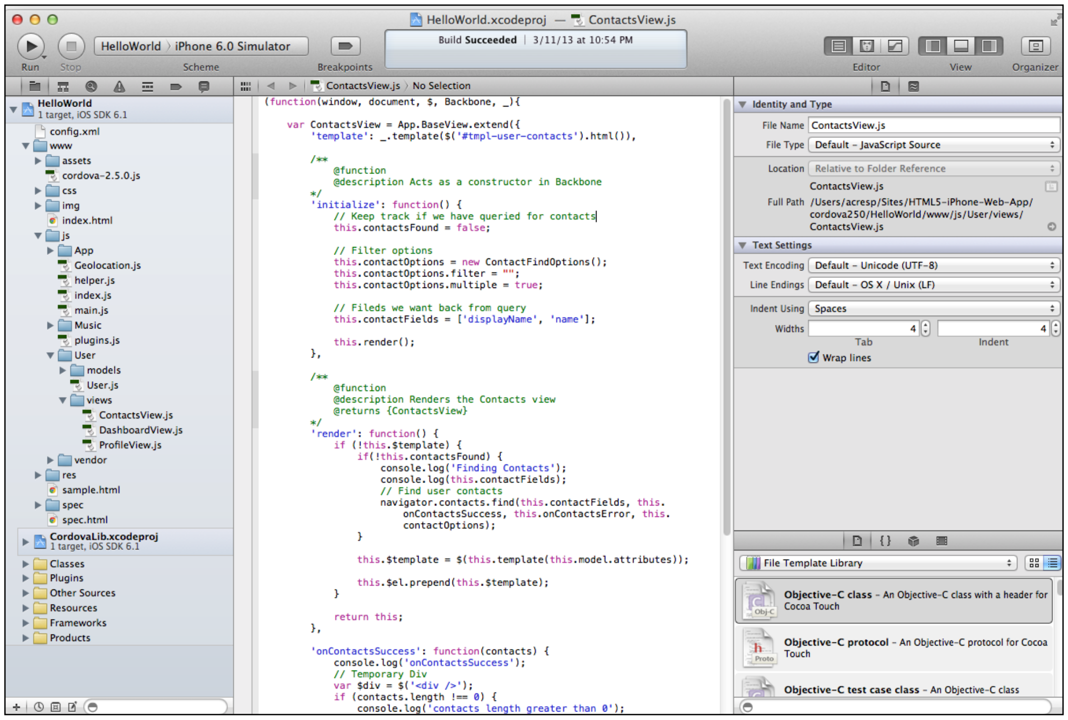
Moving on to the **View** toolbar section, we have three buttons. Each button reveals a certain part of the editor that is useful to us depending on the situation. The first button is selected by default because it shows us the Navigator Area to the left. The middle button reveals the debugging area as shown in the following screenshot:



```
(function(window, document, $, Backbone, _){

    var ContactsView = App.BaseView.extend({
        'template': _.template($('#tmpl-user-contacts').html()),

        /**
            @function
            @description Acts as a constructor in Backbone
        */
        'initialize': function() {
            // Keep track if we have queried for contacts
            this.contactsFound = false;

            // Filter options
            this.contactOptions = new ContactFindOptions();
            this.contactOptions.filter = "";
            this.contactOptions.multiple = true;

            // Fileds we want back from query
            this.contactFields = ['displayName', 'name'];

            this.render();
        },

        /**
            @function
            @description Renders the Contacts view
            @returns {ContactsView}
        */
        'render': function() {
            if (!this.$template) {
                if(!this.contactsFound) {
                    console.log('Finding Contacts');
                    console.log(this.contactFields);
                    // Find user contacts
                    navigator.contacts.find(this.contactFields, this.onContactsSuccess, this.onContactsError, this.contactOptions);
                }
```

The project debugging display

This is great because we can now debug our application as it runs and see the logs as the application is tested. Remember all those logs we use in our applications? They will show up here; they are extremely useful if we don't have a developer console that is very useful in our browser. The last button in the toolbar controls the utilities. These utilities help us control various settings of the current file in question; from name to source control, we can customize various details of our files.



```
(function(window, document, $, Backbone, _){

    var ContactsView = App.BaseView.extend({
        'template': _.template($('#tmpl-user-contacts').html()),

        /**
            @function
            @description Acts as a constructor in Backbone
        */
        'initialize': function() {
            // Keep track if we have queried for contacts
            this.contactsFound = false;

            // Filter options
            this.contactOptions = new ContactFindOptions();
            this.contactOptions.filter = "";
            this.contactOptions.multiple = true;

            // Fileds we want back from query
            this.contactFields = ['displayName', 'name'];

            this.render();
        },

        /**
            @function
            @description Renders the Contacts view
            @returns {ContactsView}
        */
        'render': function() {
            if (!this.$template) {
                if(!this.contactsFound) {
                    console.log('Finding Contacts');
                    console.log(this.contactFields);
                    // Find user contacts
                    navigator.contacts.find(this.contactFields, this.
                        onContactsSuccess, this.onContactsError, this.
                        contactOptions);
                }

                this.$template = $(this.template(this.model.attributes));

                this.$el.prepend(this.$template);
            }

            return this;
        },

        'onContactsSuccess': function(contacts) {
            console.log('onContactsSuccess');
            // Temporary Div
            var $div = $('<div />');
            if (contacts.length !== 0) {
                console.log('contacts length greater than 0');
```

Identity and Type
File Name: ContactsView.js
File Type: Default – JavaScript Source
Location: Relative to Folder Reference
ContactsView.js
Full Path: /Users/acresp/Sites/HTML5-iPhone-Web-App/cordova250/HelloWorld/www/js/User/views/ContactsView.js

Text Settings
Text Encoding: Default – Unicode (UTF-8)
Line Endings: Default – OS X / Unix (LF)
Indent Using: Spaces
Widths: 4 (Tab) 4 (Indent)
✓ Wrap lines

The project file configuration display

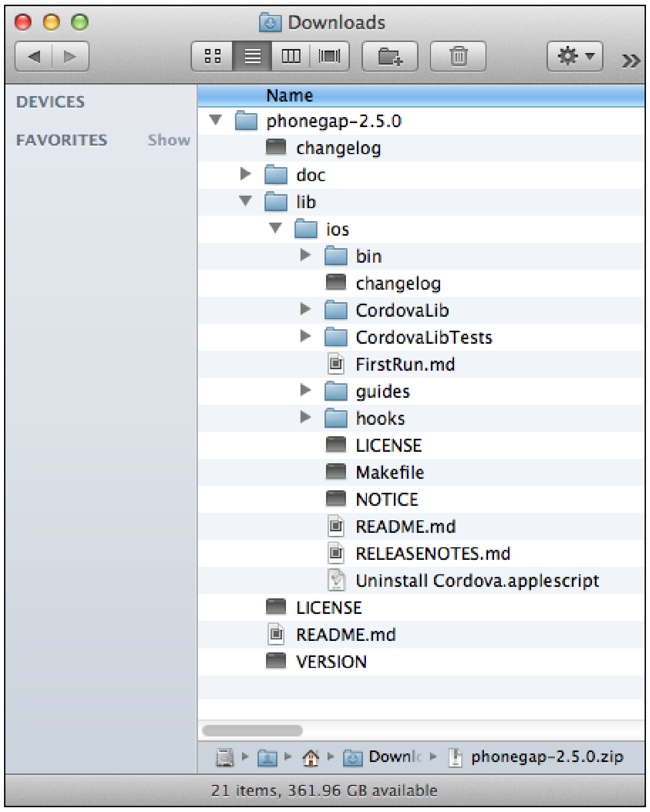
Okay, so we know the basic features of Xcode and that there is much to explore—and that it is both great and beneficial to us as developers. We can continue to cover all the extremely useful features of Xcode, but for our benefit let's move on to PhoneGap, after all we are more interested in learning to build a native application. The tools can always be used and customized for our needs.

Setting up PhoneGap

Xcode is great to use in your arsenal of application development environments. However, PhoneGap is where all the magic happens. It's the framework that empowers us to create native applications that are based on the code we have already written with HTML, CSS, and JavaScript. So let's review how to install it, create a project, and briefly go over its support and license to prepare us to harness its abilities for our own applications.

Installing PhoneGap

PhoneGap is extremely easy to get started with; first let's install it from PhoneGap's website, here: `http://phonegap.com/download/`. When the ZIP file completely finishes downloading, we'll want to extract its contents. Now when you start examining the contents of the extraction, you'll notice that there's a lot going on, especially in the `lib` directory where multiple operating systems are listed. This is good, because PhoneGap supports multiple platforms, but what we want is specifically for iOS. Our focus should be on the following:



```
phonegap-2.5.0
    changelog
    doc
    lib
        ios
            bin
            changelog
            CordovaLib
            CordovaLibTests
            FirstRun.md
            guides
            hooks
            LICENSE
            Makefile
            NOTICE
            README.md
            RELEASENOTES.md
            Uninstall Cordova.applescript
    LICENSE
    README.md
    VERSION
```

PhoneGap directory structure

Notice that in the iOS directory we have multiple files, all of which are critical to the creation of our first PhoneGap project. In the next section, we'll create our first PhoneGap project using this cleaned up version of the PhoneGap framework.

Creating a PhoneGap project

Now that we have the PhoneGap framework downloaded and simplified for our purposes, we want to create our first ever project using this framework. In order to do that, we'll need the help of our trusted c**ommand-line interface** (**CLI**). By default, all Mac operating systems comes with **Terminal**, but we can also us iTerm (free) as well. Either way, launch that application, which is located in `/Applications/Utilities/`.

When you have Terminal open, we'll need to navigate to the directory your PhoneGap files are located at. This should be in our `Downloads` directory by default, depending on your browser settings. In this case, I would navigate to `/Users/acresp/Downloads` with the `cd` command, like so:

```
cd /Users/acresp/Downloads
```

Once we're in the directory PhoneGap was extracted to, we need to navigate to view the `bin` directory inside the `iOS` folder inside `phonegap`. To do this, we can type in the following:

```
cd phonegap-2.5.0/lib/ios/bin/
```

Now we can build our PhoneGap application using the `create` shell script inside the `bin` folder. The documentation for this script is as follows:

```
    #
    # create a Cordova/iOS project
    #
    # USAGE
    #   ./create <path_to_new_project> <package_name> <project_name>
    #
    # EXAMPLE
    #  ./create ~/Desktop/radness org.apache.cordova.radness Radness
    #
```

This is excellent for us because we know what we can do to create our application easily. But before we do that, let's make sure our application directory is created within our project. For the purpose of this chapter, I have created a `cordova250` directory that will contain our `HelloWorld` application, and may contain other PhoneGap projects as well.

Now that we have made sure that our application directory exists, we can run the following command to make sure our application gets created:

```
./create ~/Sites/HTML5-iPhone-Web-App/cordova250/HelloWorld
.org.apache.cordova.HelloWorld HelloWorld
```

This will produce a directory called `HelloWorld` inside the `cordova250` folder with all the necessary files we need to get started. We have now created our first PhoneGap project. There's not a whole lot happening just yet, but let's continue; we'll soon start developing a native application. First, let's review the support that's out there for this framework and the license backing it up.

The PhoneGap license

You're probably wondering about the PhoneGap license, especially since we have used many open source projects along the way to create our applications. PhoneGap is based on the Apache license (`http://phonegap.com/about/license/`). And what's even better for us is that the Apache foundation has provided us with clear and concise information about what is allowed, forbidden, and required. Straight from the *FAQ* section of the *What does it mean?* section (available at `http://www.apache.org/foundation/license-faq.html#WhatDoesItMEAN`), we are given all the details we need:

> *It* **allows** *you to:*
>
> *freely download and use Apache software, in whole or in part, for personal, company internal, or commercial purposes;*
>
> *use Apache software in packages or distributions that you create.*
>
> *It* **forbids** *you to:*
>
> *redistribute any piece of Apache-originated software without proper attribution;*
>
> *use any marks owned by The Apache Software Foundation in any way that might state or imply that the Foundation endorses your distribution;*
>
> *use any marks owned by The Apache Software Foundation in any way that might state or imply that you created the Apache software in question.*
>
> *It* **requires** *you to:*
>
> *include a copy of the license in any redistribution you may make that includes Apache software;*
>
> *provide clear attribution to The Apache Software Foundation for any distributions that include Apache software.*

> *It* **does not require** *you to:*
>
> *include the source of the Apache software itself, or of any modifications you may have made to it, in any redistribution you may assemble that includes it;*
>
> *submit changes that you make to the software back to the Apache Software Foundation (though such feedback is encouraged).*

Based on these parameters, we can continue to create open source software using PhoneGap as long as we include a copy of the license with every redistribution, so long as it has clear attribution to The Apache Software Foundation. If you do have any other questions related to PhoneGap's license or the Apache 2.0 license, there is more information on the preceding link and on PhoneGap's license page (`http://phonegap.com/about/license/`).

Configuring our project

Our project can be configured to fit our needs and concurrently, those for our users. This process is fairly simple and well documented at the PhoneGap API documentation site (`http://docs.phonegap.com/en/2.5.0/guide_project-settings_index.md.html#Project%20Settings`). Most of these settings are easily modified through the `config.xml` file located in our project directory `/cordovar250/HelloWorld/HelloWorld/config.xml`.

Here is a current list of items that can be customized:

Preference	Description
`UIWebViewBounce` (Boolean, defaults to **true**)	This sets the property of a rubber-band type interaction/bounce animation.
`TopActivityIndicator` (string, defaults to **gray**)	This sets the color of the spinning throbber in the status/battery bar, with valid values of **whiteLarge**, **white**, and **gray**.
`EnableLocation` (Boolean, defaults to **false**)	This determines whether to initialize the Geolocation plugin at start-up, making your location more accurate at startup.
`EnableViewportScale` (Boolean, defaults to **false**)	This enables/disables viewport scaling.

Preference	Description
`AutoHideSplashScreen` (Boolean, defaults to **true**)	This controls whether the splash screen is hidden through a JavaScript API.
`FadeSplashScreen` (Boolean, defaults to **true**)	This enables the splash screen to fade in and out.
`FadeSplashScreenDuration` (float, defaults to **2**)	This indicates the splash screen's fade duration in seconds.
`ShowSplashScreenSpinner` (Boolean, defaults to **true**)	This shows or hides the splash screen's loading spinner.
`MediaPlaybackRequiresUserAction` (Boolean, defaults to **false**)	This allows HTML5 to auto play.
`AllowInlineMediaPlayback` (Boolean, defaults to **false**)	This controls inline HTML5 media playback. The `video` element in the HTML document must also include the `webkit-playsinline` attribute
`BackupWebStorage` (string, defaults to **cloud**)	If set to **cloud**, storage data will be backed up to iCloud. If set to **local**, only local backups will be made. If set to **none**, no backups occur.
`KeyboardDisplayRequiresUserAction` (Boolean, defaults to **true**)	If this is set to **false**, the keyboard will open when `form` elements get called via JavaScript's `focus()`.
`SuppressesIncrementalRendering` (Boolean, defaults to **false**)	This allows content to be received before being rendered.

Transferring a web application

At this point, we have created a sample `HelloWorld` application using PhoneGap and Xcode. Now, we'll take it up a notch by reviewing transferring our single-page application from *Chapter 7, One-page Applications*. In this section, we'll go over transferring our assets, including our markup, styles, and scripts, and then learn how to debug our applications. Finally, we'll extend our single-page application by using native functionality that PhoneGap has allowed us to tap into using the code we've already written.

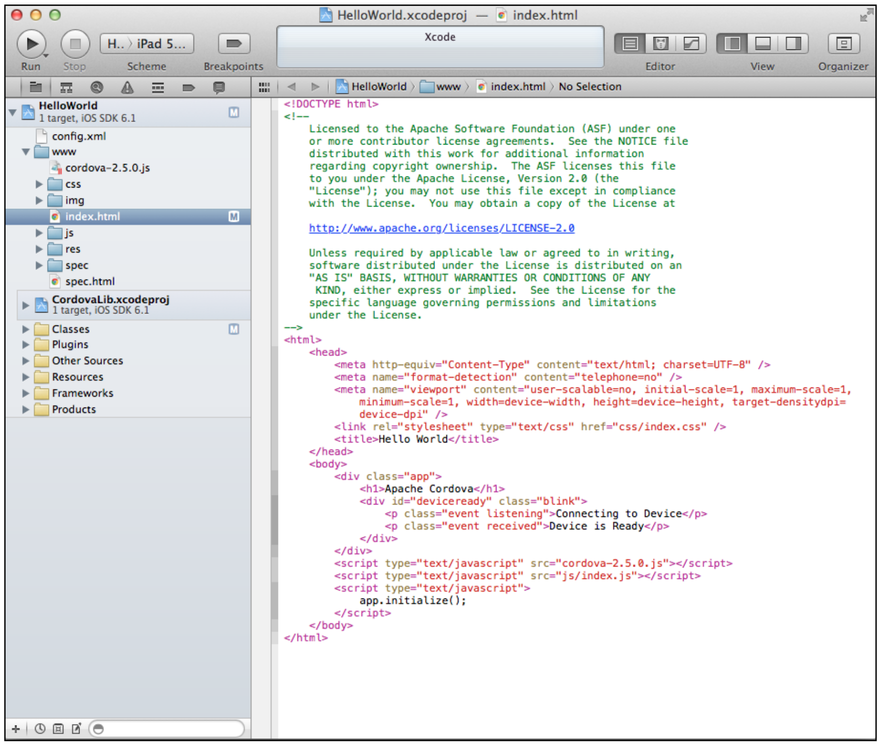
Transferring our assets

Let's start by transferring our assets. This section will briefly go over how to transfer what we have written with a minimal level of effort. The goal here is to basically have the same application we have running natively. We won't be using PhoneGap's built-in features just yet, but we'll have an app up and running quickly.

Including our markup

The first thing we want to do here is open up the Xcode project that was generated previously using PhoneGap. To do this, we first locate our project in Finder, in my case `~/Sites/HTML5-iPhone-Web-App/cordova250/HelloWorld/`. Once we've located our project, double-click on the `HelloWorld.xcodeproj` file; this will launch the project in Xcode.

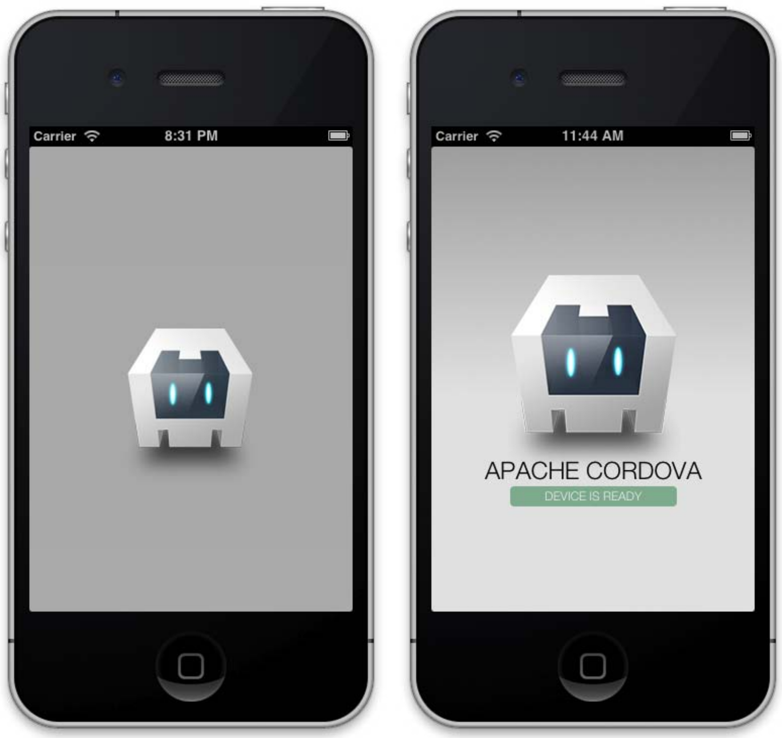
Once Xcode has launched with our project, we'll see it index our files. While it's indexing, it won't prevent you from interacting with your project, so you'll be able to start editing files. So, let's go ahead and check out our `index.html` file located in the `www` directory.



```
<!DOCTYPE html>
<!--
    Licensed to the Apache Software Foundation (ASF) under one
    or more contributor license agreements.  See the NOTICE file
    distributed with this work for additional information
    regarding copyright ownership.  The ASF licenses this file
    to you under the Apache License, Version 2.0 (the
    "License"); you may not use this file except in compliance
    with the License.  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing,
    software distributed under the License is distributed on an
    "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
     KIND, either express or implied.  See the License for the
    specific language governing permissions and limitations
    under the License.
-->
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
        <meta name="format-detection" content="telephone=no" />
        <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width, height=device-height, target-densitydpi=
            device-dpi" />
        <link rel="stylesheet" type="text/css" href="css/index.css" />
        <title>Hello World</title>
    </head>
    <body>
        <div class="app">
            <h1>Apache Cordova</h1>
            <div id="deviceready" class="blink">
                <p class="event listening">Connecting to Device</p>
                <p class="event received">Device is Ready</p>
            </div>
        </div>
        <script type="text/javascript" src="cordova-2.5.0.js"></script>
        <script type="text/javascript" src="js/index.js"></script>
        <script type="text/javascript">
            app.initialize();
        </script>
    </body>
</html>
```

Our project's preliminary HelloWorld markup

As you can see, we've got a basic template set up for us. Let's run this `HelloWorld` markup to see the results. The first item you should see is a splash screen with the default PhoneGap image and immediately after the device ready introduction. Here are screenshots displaying the results:



The splash screen and the device-ready screen

Now that we know that our application is running with the default markup and styles, we should be able to move pretty quickly. So, the first order of business is to bring in the completed markup from the single-page application screen seen in *Chapter 7, One-page Applications*. We won't review the code written for that chapter here, but here is the template:

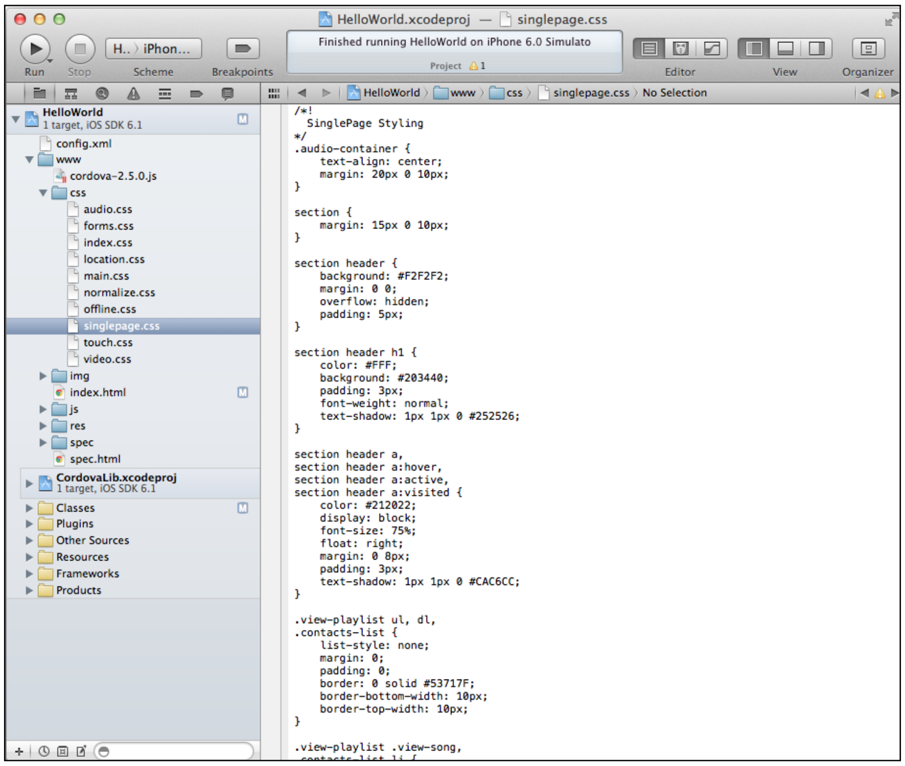
```
<!DOCTYPE html>
<html class="no-js">
<head>
    <meta charset="utf-8">
    <title></title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
/>
    <meta name="format-detection" content="telephone=no" />
    <meta name="viewport" content="user-scalable=no, initial-scale=1,
maximum-scale=1, minimum-scale=1, width=device-width, height=device-
height, target-densitydpi=device-dpi" />

    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/main.css">
    <link rel="stylesheet" href="css/singlepage.css">
    <script src="js/vendor/modernizr-2.6.1.min.js"></script>
</head>
    <body>
        <div class="app">
            <div id="deviceready" class="blink">
                <p class="event listening">Connecting to Device</p>
                <div class="event received site-wrapper">
                    <header>
                        <hgroup>
                            <h1>iPhone Web Application Development</
h1>
                            <h2>Single Page Applications</h2>
                        </hgroup>
                    </header>
                    <div class="content"></div>
                    <footer>
                        <p>iPhone Web Application Development &copy;
2013</p>
                    </footer>
                </div>
            </div>
        </div>
    </body>
</html>
```

Keep in mind that we have made some modifications in order to fit this directory structure. For example, instead of referencing our CSS files using `../css/somefile.css`, we use `css/somefile.css` and the same applies for any other assets that we'll be importing. You'll also notice that the preceding code template doesn't include the templates we created from *Chapter 7*, *One-page Applications*; this is to keep the preceding template short and simple in terms of how to import assets into your own PhoneGap project.

At this point, we won't test our application because we're not done bringing in our assets, including styles and scripts, but we should be good for the time being. What we want to take away here is that importing an existing static web application is as simple as copy and paste, but don't let this fool you; most applications are not as simple and this example is just to demonstrate how easy it is to get going.
Now let's move on to importing our styles.

Incorporating our styles

We now have our markup set up in our project `index.html` file. That was easy; this part will also be as easy. All we need to do is include our CSS files that are being used for this project. To make things easier, I've just included all of our previous stylesheets into the Xcode project's CSS directory. Your project should now look like this:



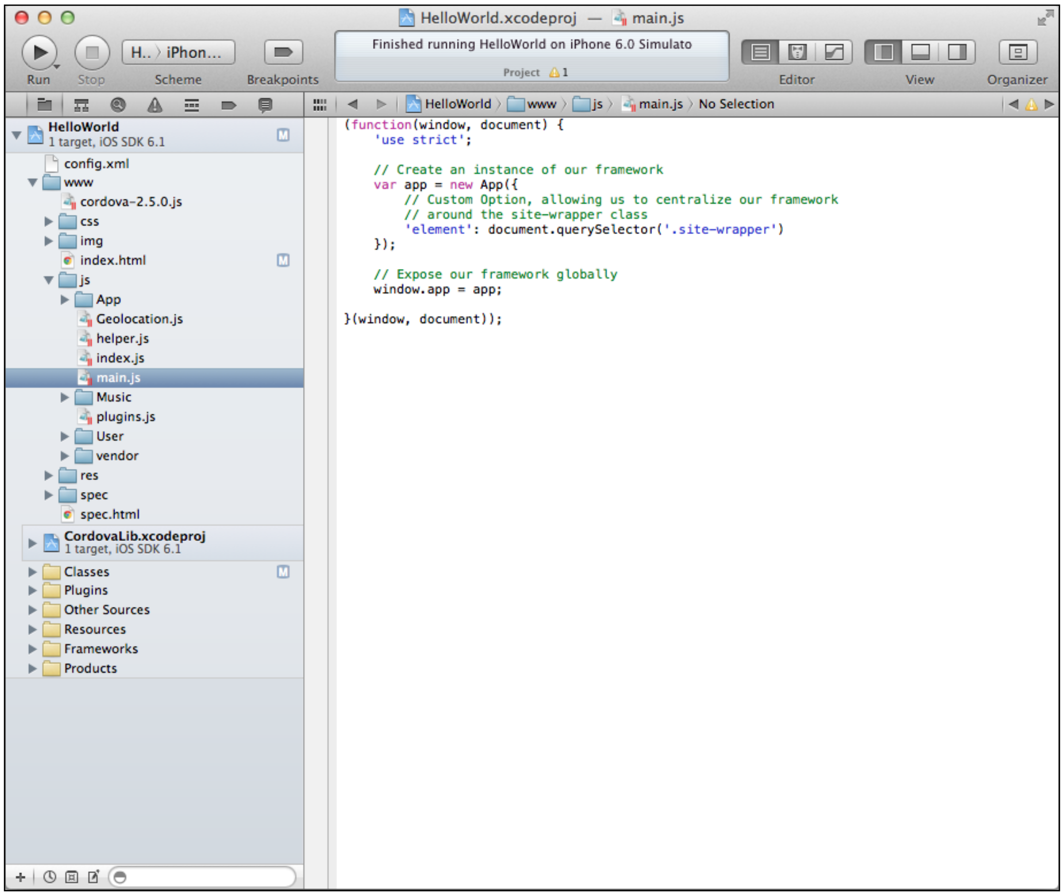
```css
/*!
  SinglePage Styling
*/
.audio-container {
    text-align: center;
    margin: 20px 0 10px;
}

section {
    margin: 15px 0 10px;
}

section header {
    background: #F2F2F2;
    margin: 0 0;
    overflow: hidden;
    padding: 5px;
}

section header h1 {
    color: #FFF;
    background: #203440;
    padding: 3px;
    font-weight: normal;
    text-shadow: 1px 1px 0 #252526;
}

section header a,
section header a:hover,
section header a:active,
section header a:visited {
    color: #212022;
    display: block;
    font-size: 75%;
    float: right;
    margin: 0 8px;
    padding: 3px;
    text-shadow: 1px 1px 0 #CAC6CC;
}

.view-playlist ul, dl,
.contacts-list {
    list-style: none;
    margin: 0;
    padding: 0;
    border: 0 solid #53717F;
    border-bottom-width: 10px;
    border-top-width: 10px;
}

.view-playlist .view-song,
```

Our imported stylesheets

Now that we have imported our stylesheets into our Xcode project, we're half way there. At this point we need to import our scripts. Again, do not test your application here since it probably won't work. This last bit will get us to where we need to be, so let's start importing our scripts.

Inserting our scripts

Okay, so we've imported our markup and stylesheets—this is great. But there's one last piece, our JavaScript. This last piece is essential to making our application run. So, let's start by doing the same thing we did for our styles; just import all your scripts into the `js` directory of the Xcode project. When you do this, the result will be the following:



Finished running HelloWorld on iPhone 6.0 Simulato

```
(function(window, document) {
    'use strict';

    // Create an instance of our framework
    var app = new App({
        // Custom Option, allowing us to centralize our framework
        // around the site-wrapper class
        'element': document.querySelector('.site-wrapper')
    });

    // Expose our framework globally
    window.app = app;

}(window, document));
```

Our imported scripts

We have our scripts inside the Xcode project. But we still need to do some configuration, including referencing our scripts correctly inside the `index.html` file, and making sure our application will launch accordingly. Let's start by referencing our scripts correctly in our `index.html` file.

Remember the markup we transferred over two sections ago, with a default template that was showcased? We're going to take a step back and look at that template again, except we're only going to look at the bottom of the markup right before the `body` tag closes. This is where our applications previously included the JavaScript; so there's nothing new here, we just want to make sure the files are being referenced correctly. Just make sure that in your `index.html` file, your scripts look like this:

```
            <!-- BEGIN: LIBRARIES / UTILITIES-->
            <script src="cordova-2.5.0.js"></script>
            <script src="js/vendor/zepto.min.js"></script>
            <script src="js/vendor/underscore-1.4.3.js"></script>
            <script src="js/vendor/backbone-0.9.10.js"></script>
            <script src="js/helper.js"></script>
            <!-- END: LIBRARIES / UTILITIES-->
            <!-- BEGIN: FRAMEWORK -->
            <script src="js/App/App.js"></script>
            <script src="js/App/App.Nav.js"></script>
            <script src="js/App/views/BaseView.js"></script>
            <!-- END: FRAMEWORK -->
            <!-- BEGIN: MUSIC PLAYLIST APPLICATION -->
            <script src="js/Music/Music.js"></script>
            <script src="js/Music/models/SongModel.js"></script>
            <script src="js/Music/collections/SongCollection.js"></script>
            <script src="js/Music/views/SongView.js"></script>
            <script src="js/Music/views/PlayListView.js"></script>
            <script src="js/Music/views/AudioPlayerView.js"></script>
            <!-- END: MUSIC PLAYLIST APPLICATION -->
            <!-- BEGIN: USER APPLICATION -->
            <script src="js/User/User.js"></script>
            <script src="js/User/models/UserModel.js"></script>
            <script src="js/User/views/DashboardView.js"></script>
            <script src="js/User/views/ProfileView.js"></script>
      <!-- END: USER APPLICATION -->
            <script src="js/main.js"></script>
            <!-- END: BACKBONE APPLICATION -->
        </body>
    </html>
```

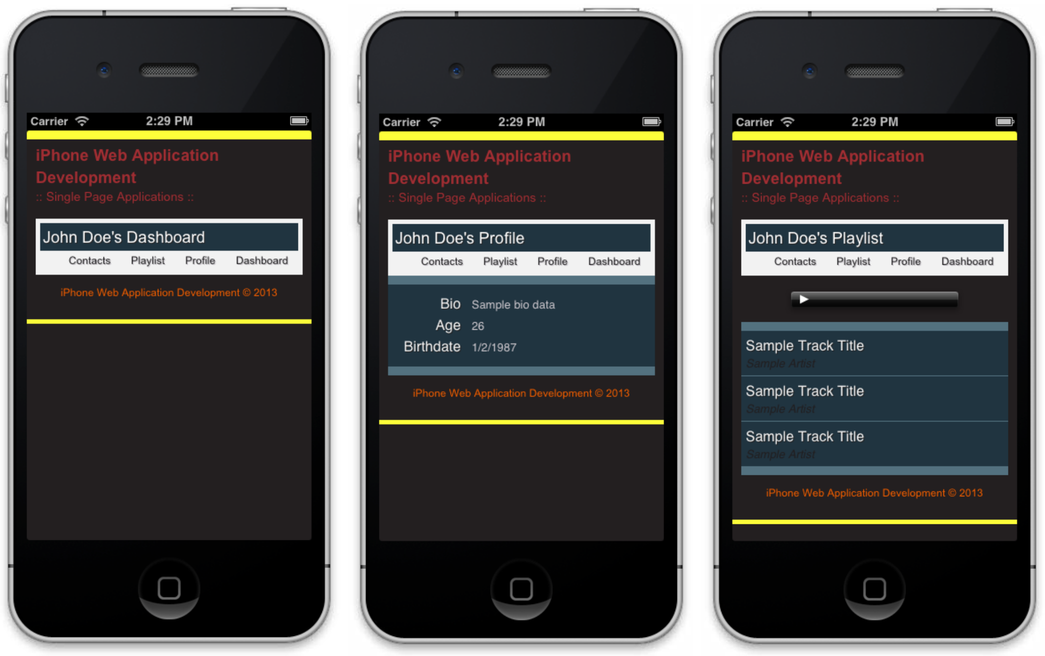
Note a couple of things that are going on here. First, we include the `cordova` library that comes with PhoneGap at the very top; this will be essential in a moment or two when we try to detect a `deviceready` event. Next, we reference all our JavaScript source files to the `js` directory in our Xcode project, not `../js`. Now, the last thing we need to do here is make sure our code runs when the device is ready, which means we need to modify how our single-page application starts.

To make sure our application starts, we need to listen to the `deviceready` event that is provided by the PhoneGap event (`http://docs.phonegap.com/en/2.5.0/cordova_events_events.md.html#deviceready`). This event is triggered once Cordova has been fully loaded. This is essential because the DOM is not loaded while native code is loading, and the splash screen is displayed. So we can run into problems when a Cordova function is required before the DOM loads. So for our purpose, we'll listen to the `deviceready` event and then start our application. This can be done with the following code:

```
<script>
    (function(){
     document.addEventListener('deviceready', onDeviceReady, false);

     function onDeviceReady(){
        console.log("onDeviceReady");
        var parentElement,
            listeningElement,
            receivedElement;

        parentElement = document.getElementById('deviceready');
        listeningElement = parentElement.querySelector('.listening');
        receivedElement = parentElement.querySelector('.received');

        listeningElement.setAttribute('style', 'display:none;');
        receivedElement.setAttribute('style', 'display:block;');

        // Start our application
        Backbone.history.start();
     }
    }());
</script>
```

Let's examine this code line by line. First, we create a closure that executes immediately. Within this scope, we listen to the `deviceready` event and assign the `onDeviceReady` callback function. We then define the `onDeviceReady` callback that shows and hides our application. This method creates three variables, `parentElement`, `listeningElement`, and `receivedElement`. We cache the `deviceready` DOM element and assign it to `parentElement`, and we do the same thing for `listeningElement` and `receivedElement`. Next, we set the `style` attribute on the proper elements, showing the application and hiding the splash screen. Finally, we start our Backbone-based single-page application.

Let's place the preceding script after all our scripts in the `index.html` file. Now, we should be able to run our application successfully and navigate the dashboard, profile, and playlist views. If everything as discussed previously was done correctly, you should be able to use your single-page application natively, like so:



Native single-page application

Note that in the preceding screenshots, we have a **Contacts** navigation item. This has not been built yet and will be part of the last section of this chapter.

At this point we have created a native application that showcases the simplicity of transferring a current web application using PhoneGap. Yes, we haven't used PhoneGap or Xcode to its fullest extent, but we now understand that getting the process started is pretty easy. We'll sidestep for a moment to understand debugging our application(s), and then ultimately use PhoneGap's API to build a native component into our application.

Debugging our application

Debugging an application is critical to any workflow or process; for this reason we need to know what it's like debugging a native application built off web technologies. It's not as complex or easy as you may think. But it's still doable and at the current time of writing, this is one of the best ways to debug your application. So let's get to it.

Logging out our code

We're all familiar with the console object available to us via JavaScript. This is still available to us but is somewhat confusing when trying to find where the log has been output when creating a native app. Traditionally, we had a console tool that was available on our simulator or actual device to debug errors; however, this is no longer the case.

First, let's look at how logging takes place in Xcode. Remember the debug view discussed earlier on in the chapter? Well, this is where we want to use it. So first, let's enable the debug view. Now, let's run the application we currently have.

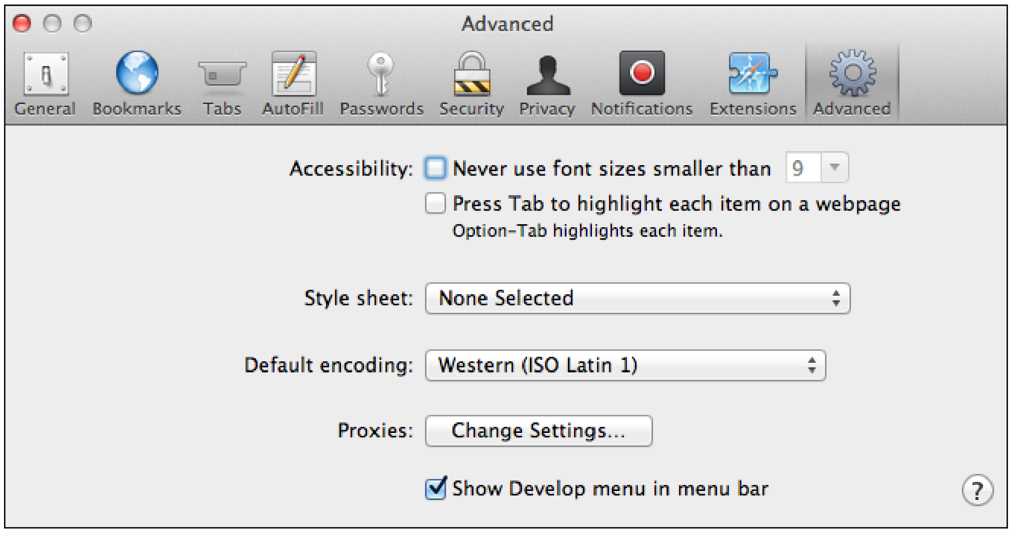
When we run your application, we should see the following in the debugger area:

```
2013-03-16 14:24:43.732 HelloWorld[2322:c07] Multi-tasking -> Device:
YES, App: YES

2013-03-16 14:24:44.624 HelloWorld[2322:c07] Resetting plugins due to
page load.

2013-03-16 14:24:45.196 HelloWorld[2322:c07] Finished load of: file:///
Users/acresp/Library/Application%20Support/iPhone%20Simulator/6.0/
Applications/DEEABC2E-C2D6-40F3-A19E-43E4F7F5EB47/HelloWorld.app/www/
index.html

2013-03-16 14:24:45.243 HelloWorld[2322:c07] [LOG] onDeviceReady
```

We should focus on the last line where `[LOG]` takes place. This is the output generated using `console.log()`, which is currently in our `onDeviceReady` callback. This is great for us because we can actively see the logs we created. The negative aspect to this is that we don't have typical developer tools that we can find in other browsers. But with recent developments, we can now debug our applications using Safari's built-in developer tools to debug an iOS app running in the simulator.

Using the Safari developer tools

As I've mentioned, we are now able to debug web applications built off PhoneGap using Safari's developer tools. So let's take a crack at this real quick, by opening Safari on our computer. If you don't already have the developer tools enabled, do so by going into Safari's preferences and selecting the checkbox for **Show Develop menu in menu bar** under the **Advanced** tab.



Accessibility: ☐ Never use font sizes smaller than 9
☐ Press Tab to highlight each item on a webpage
Option-Tab highlights each item.

Style sheet: None Selected

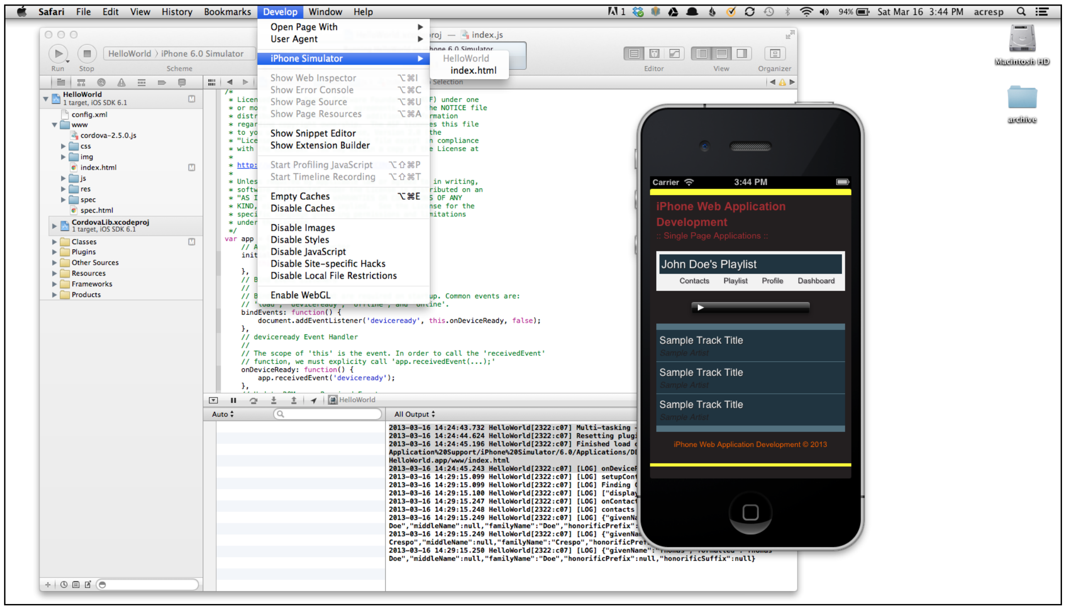
Default encoding: Western (ISO Latin 1)

Proxies: Change Settings...

☑ Show Develop menu in menu bar

The Advanced tab for Safari preferences

Once we have the developer tools enabled, we can access them from the **Develop** menu for Safari. If we have our application running in the iOS simulator, then we should be able to debug our application by selecting `index.html` from the iPhone Simulator submenu. This will then launch the native developer tools in Safari.



```
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
    // deviceready Event Handler
    //
    // The scope of 'this' is the event. In order to call the 'receivedEvent'
    // function, we must explicity call 'app.receivedEvent(...);'
    onDeviceReady: function() {
        app.receivedEvent('deviceready');
    },
```

```
2013-03-16 14:24:43.732 HelloWorld[2322:c07] Multi-tasking -
2013-03-16 14:24:44.624 HelloWorld[2322:c07] Resetting plugi
2013-03-16 14:24:45.196 HelloWorld[2322:c07] Finished load o
Application%20Support/iPhone%20Simulator/6.0/Applications/DE
HelloWorld.app/www/index.html
2013-03-16 14:24:45.243 HelloWorld[2322:c07] [LOG] onDeviceR
2013-03-16 14:29:15.099 HelloWorld[2322:c07] [LOG] setupCont
2013-03-16 14:29:15.099 HelloWorld[2322:c07] [LOG] Finding C
2013-03-16 14:29:15.100 HelloWorld[2322:c07] [LOG] ["display
2013-03-16 14:29:15.247 HelloWorld[2322:c07] [LOG] onContact
2013-03-16 14:29:15.248 HelloWorld[2322:c07] [LOG] contacts
2013-03-16 14:29:15.249 HelloWorld[2322:c07] [LOG] {"givenNa
Doe","middleName":null,"familyName":"Doe","honorificPrefix":
2013-03-16 14:29:15.249 HelloWorld[2322:c07] [LOG] {"givenNam
Crespo","middleName":null,"familyName":"Crespo","honorificPref
2013-03-16 14:29:15.250 HelloWorld[2322:c07] [LOG] {"givenName":"Thomas","formatted":"Thomas
Doe","middleName":null,"familyName":"Doe","honorificPrefix":null,"honorificSuffix":null}
```

Debugging environment

Now we are able to fully debug an application using the Safari developer tools for our native application. It is truly that easy to have a fully integrated development environment with simulation and debugging all part of the process. Although we can go into further detail about debugging, it is beyond the scope of this book. However, let's move on to the final section of this book, where we will learn to utilize PhoneGap's built-in APIs to tie into native functionality for our single-page application.

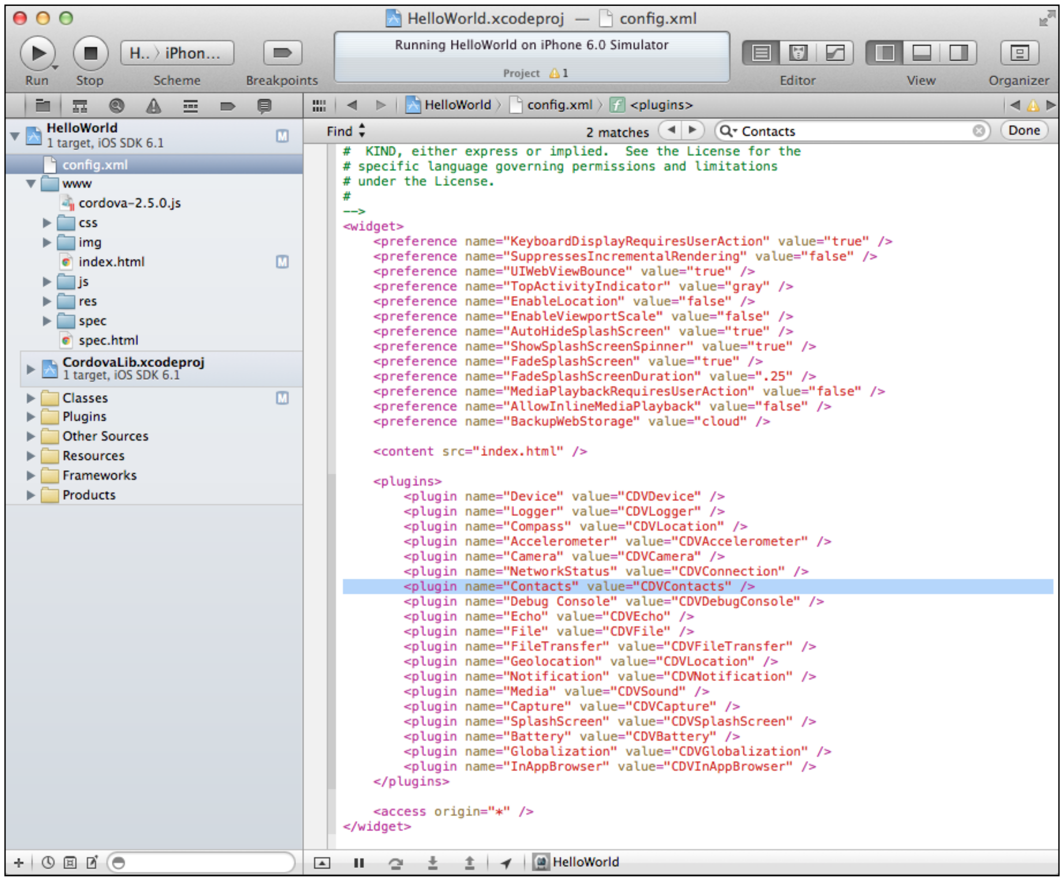
Extending our application with native functionality

Congratulations! We have been able to create our first native application using what we've already created with HTML5, CSS, and JavaScript. This is exciting stuff, but we're not done yet. Let's now leverage one of PhoneGap's APIs to tap into native functionality.

From a higher level we want our application to display the contacts we have on our phone. We want to be able to access this information when we click on the **Contacts** button in our application navigation. For this example, we just want to display the full name of our contacts. To achieve these goals we'll be using PhoneGap's Contacts API (`http://docs.phonegap.com/en/2.5.0/cordova_contacts_contacts.md.html#Contacts`). To do this, we'll ensure that this is configured in our application, and then write the appropriate code to handle this within the already existing framework of our application. Let's begin with the configuration.

Configuring our application

We've already gone over the basics of configuring our application previously, but let's take a look at this again to ensure its full understanding. First, let's open up the `config.xml` file located at the top of our project. Then enable the Contacts API by setting its value to `CDVContacts`. When you're done, you should have the following in your `config.xml` file:



```
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
#
-->
<widget>
    <preference name="KeyboardDisplayRequiresUserAction" value="true" />
    <preference name="SuppressesIncrementalRendering" value="false" />
    <preference name="UIWebViewBounce" value="true" />
    <preference name="TopActivityIndicator" value="gray" />
    <preference name="EnableLocation" value="false" />
    <preference name="EnableViewportScale" value="false" />
    <preference name="AutoHideSplashScreen" value="true" />
    <preference name="ShowSplashScreenSpinner" value="true" />
    <preference name="FadeSplashScreen" value="true" />
    <preference name="FadeSplashScreenDuration" value=".25" />
    <preference name="MediaPlaybackRequiresUserAction" value="false" />
    <preference name="AllowInlineMediaPlayback" value="false" />
    <preference name="BackupWebStorage" value="cloud" />

    <content src="index.html" />

    <plugins>
        <plugin name="Device" value="CDVDevice" />
        <plugin name="Logger" value="CDVLogger" />
        <plugin name="Compass" value="CDVLocation" />
        <plugin name="Accelerometer" value="CDVAccelerometer" />
        <plugin name="Camera" value="CDVCamera" />
        <plugin name="NetworkStatus" value="CDVConnection" />
        <plugin name="Contacts" value="CDVContacts" />
        <plugin name="Debug Console" value="CDVDebugConsole" />
        <plugin name="Echo" value="CDVEcho" />
        <plugin name="File" value="CDVFile" />
        <plugin name="FileTransfer" value="CDVFileTransfer" />
        <plugin name="Geolocation" value="CDVLocation" />
        <plugin name="Notification" value="CDVNotification" />
        <plugin name="Media" value="CDVSound" />
        <plugin name="Capture" value="CDVCapture" />
        <plugin name="SplashScreen" value="CDVSplashScreen" />
        <plugin name="Battery" value="CDVBattery" />
        <plugin name="Globalization" value="CDVGlobalization" />
        <plugin name="InAppBrowser" value="CDVInAppBrowser" />
    </plugins>

    <access origin="*" />
</widget>
```

The project configuration

Setting up our contacts functionality

In this section of the chapter, we'll look at hooking into our contacts information to display in our native application. First we'll create the view, then the template, and finally the actual API that comes with PhoneGap. When we're done, we should have a good idea on how we can tap into these APIs to create native web applications for iOS.

Creating the ContactsView class

Once we have set up our configuration for this application, we need to set up everything else in order to get the contacts view to work. To begin, let's create a contacts view to be added to our user directory. We'll extend its functionality later, but for now here is the template we'll be using:

```
(function(window, document, $, Backbone, _){

  var ContactsView = App.BaseView.extend({
    'template': _.template($('#tmpl-user-contacts').html()),

    'initialize': function() {

      this.render();
    },

    'render': function() {

      return this;
    }
  });

  window.User.ContactsView = ContactsView;

}(window, document, Zepto, Backbone, _));
```

The preceding code isn't anything new. We're basically creating a `ContactsView` class that follows the conventions we've set previously with no bit of additional functionality. Do note that we have set a template for this view that does not exist yet. Let's include this file with the scripts we've included in `index.html` add it to the last view being included. Now, let's create the template associated with this view.

Implementing the ContactsView template

Using what we've already built for our playlist, we'll just duplicate the template for the playlist view and change its header. While we're at it, we'll just change the class of the unordered list to `contacts-list` as well. When we're done, our template will look like this:

```
<script type="tmpl/User" id="tmpl-user-contacts">
    <section class="view-contacts">
    <header>
    <h1><%= name + "'s" %> Contacts</h1>
    <% print(_.template($('#tmpl-user-nav').html(), {})); %>
    </header>
    <ul class="contacts-list"></ul>
    </section>
</script>
```

Include this template after the rest of the other templates we've created. At this point, we should be 50 percent there. Now, you may run into some styling issues, but just make sure to add the `contacts-list` class to the same styles that the playlist uses. We won't go over that here, since it's pretty brief; so we'll continue by writing the contacts implementation.

Integrating the Contacts API

To look up a user's contacts is pretty simple using the PhoneGap API. In fact, our example will be based on the documented `Navigator` object, `contacts`. But first, we need to create a new instance of `ContactFindOptions` (`http://docs.phonegap.com/en/2.5.0/cordova_contacts_contacts.md.html#ContactFindOptions`), which will hold our filtering options when finding contacts.

```
'initialize': function() {

  // Filter options
  this.contactOptions = new ContactFindOptions();
  this.contactOptions.filter = "";
  this.contactOptions.multiple = true;

  this.render();
},
```

The preceding code sets the `filter` and `multiple` properties on the instance of `ContactFindOptions`. By default `filter` is empty, meaning there is no limitation, and `multiple` is set to `true`, allowing for multiple contacts to come through. Next, we want to find two fields when we get the contacts, their `displayName`, and `name`. These fields will be in an array, which we'll use shortly.

```
'initialize': function() {

  // Filter options
  this.contactOptions = new ContactFindOptions();
  this.contactOptions.filter = "";
  this.contactOptions.multiple = true;

  // Fileds we want back from query
  this.contactFields = ['displayName', 'name'];

  this.render();
},
```

Next, we want to find the contacts when the view renders. So inside our render view, we want to pass in the preceding options.

```
'render': function() {
    // Find user contacts
    navigator.contacts.find(this.contactFields, this.
onContactsSuccess, this.onContactsError, this.contactOptions);

    this.$template = $(this.template(this.model.attributes));

    this.$el.prepend(this.$template);
  }

  return this;
},
```

Note that we have not yet created our `onContactsError` or `onContactsSuccess` methods. Also, you'll see that we create the template and attach it to the DOM the same way we did for all the other views. There's not much left to do with this method, so let's look at our callbacks, beginning with `onContactSuccess`.

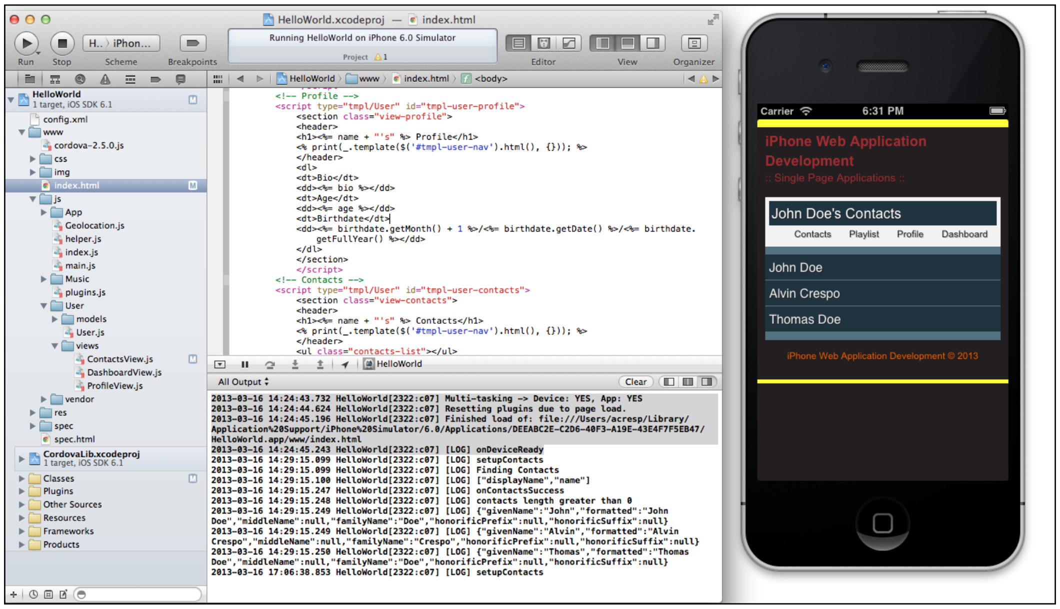
The `onContactSuccess` callback is where all our magic happens. We'll create a `div` element in memory and then loop through the results, adding each element to `div` as a list item. Once everything is completed, we'll take the contents of that the `div` element and add it to our `contacts-list` unordered list.

```
'onContactsSuccess': function(contacts) {
  console.log('onContactsSuccess');
  // Temporary Div
  var $div = $('<div />');
  if (contacts.length !== 0) {
    console.log('contacts length greater than 0');
    _.each(contacts, function(contact){
      console.log(contact.name);
      $div.append($('<li>' + contact.name.formatted + '</li>'));
    });
  } else {
    alert("No contacts found!");
  }

  $('.contacts-list').append($div.html());
},
```

As you can see here, we use the **underscore** method `each` to loop through the results. And as we mentioned before, we create a list item containing the user's name as its text content. Pretty simple behavior here and nothing too complicated. Now, let's check out our `onContactsError` callback:

```
'onContactsError': function(contactsError) {
  alert('onContactsError!');
}
```

In this callback, we just alert that there has been error. Of course, in our real-world application we would create something a bit more comprehensive, but for our purposes this is good enough. If we run our application now, we should get the following:



```
<!-- Profile -->
<script type="tmpl/User" id="tmpl-user-profile">
    <section class="view-profile">
    <header>
    <h1><%= name + "'s" %> Profile</h1>
    <% print(_.template($('#tmpl-user-nav').html(), {})); %>
    </header>
    <dl>
    <dt>Bio</dt>
    <dd><%= bio %></dd>
    <dt>Age</dt>
    <dd><%= age %></dd>
    <dt>Birthdate</dt>
    <dd><%= birthdate.getMonth() + 1 %>/<%= birthdate.getDate() %>/<%= birthdate.
        getFullYear() %></dd>
    </dl>
    </section>
    </script>
<!-- Contacts -->
<script type="tmpl/User" id="tmpl-user-contacts">
    <section class="view-contacts">
    <header>
    <h1><%= name + "'s" %> Contacts</h1>
    <% print(_.template($('#tmpl-user-nav').html(), {})); %>
    </header>
    <ul class="contacts-list"></ul>
```

```
2013-03-16 14:24:43.732 HelloWorld[2322:c07] Multi-tasking -> Device: YES, App: YES
2013-03-16 14:24:44.624 HelloWorld[2322:c07] Resetting plugins due to page load.
2013-03-16 14:24:45.196 HelloWorld[2322:c07] Finished load of: file:///Users/acresp/Library/
Application%20Support/iPhone%20Simulator/6.0/Applications/DEEABC2E-C2D6-40F3-A19E-43E4F7F5EB47/
HelloWorld.app/www/index.html
2013-03-16 14:24:45.243 HelloWorld[2322:c07] [LOG] onDeviceReady
2013-03-16 14:29:15.099 HelloWorld[2322:c07] [LOG] setupContacts
2013-03-16 14:29:15.099 HelloWorld[2322:c07] [LOG] Finding Contacts
2013-03-16 14:29:15.100 HelloWorld[2322:c07] [LOG] ["displayName","name"]
2013-03-16 14:29:15.247 HelloWorld[2322:c07] [LOG] onContactsSuccess
2013-03-16 14:29:15.248 HelloWorld[2322:c07] [LOG] contacts length greater than 0
2013-03-16 14:29:15.249 HelloWorld[2322:c07] [LOG] {"givenName":"John","formatted":"John
Doe","middleName":null,"familyName":"Doe","honorificPrefix":null,"honorificSuffix":null}
2013-03-16 14:29:15.249 HelloWorld[2322:c07] [LOG] {"givenName":"Alvin","formatted":"Alvin
Crespo","middleName":null,"familyName":"Crespo","honorificPrefix":null,"honorificSuffix":null}
2013-03-16 14:29:15.250 HelloWorld[2322:c07] [LOG] {"givenName":"Thomas","formatted":"Thomas
Doe","middleName":null,"familyName":"Doe","honorificPrefix":null,"honorificSuffix":null}
2013-03-16 17:06:38.853 HelloWorld[2322:c07] [LOG] setupContacts
```

The Contacts API implementation

Give yourself a pat on the back! You have reached the end of this section, have now successfully integrated with the PhoneGap API, and tapped into native functionality. Very cool, isn't it?

Please note that the source code provided with this book comes with a few checks, making sure that the same contacts aren't added each time the user visits the **Contacts** view. This was done in order to save some time and really focus on the meat of the solution.

Summary

In this chapter, we were introduced to native application development using the same programming languages we use for our web applications. Using the popular open source PhoneGap framework, we achieved the ability to create the single-page application, built in *Chapter 7*, *One-page Applications*, as a native application for iOS. We went over extending the single-page application by tying into the native functionality using the Contacts API in PhoneGap, listing out our contacts and some information. We should now have a foundation for creating native applications that allow us to use web technologies to distribute web applications for iOS Safari and the iOS operating system.

Index