Quick answers to common problems

PhoneGap Mobile Application Development Cookbook

Over 40 recipes to create mobile applications using the PhoneGap API with examples and clear instructions

Foreword by Brian Leroux, Senior Product Manager, PhoneGap Lead and SPACELORD!1!! at Adobe Systems Ltd

Matt Gifford

PhoneGap Mobile Application Development Cookbook

Over 40 recipes to create mobile applications using the PhoneGap API with examples and clear instructions

Matt Gifford

[PACKT] open source*
PUBLISHING   community experience distilled

BIRMINGHAM - MUMBAI

PhoneGap Mobile Application Development Cookbook

Credits

Author
Matt Gifford

Reviewers
Raymond Camden
Shaun Dunne
Andrey Rebrov

Acquisition Editor
Usha Iyer

Lead Technical Editor
Unnati Shah

Technical Editor
Jalasha D'costa

Project Coordinator
Joel Goveya

Proofreader
Mario Cecere

Indexer
Hemangini Bari

Graphics
Valentina D'silva

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

Foreword

In the summer of 2007, Steve Jobs changed the world by releasing the iPhone and boldly declared the future was web-based applications. A short year later, the story changed, but the vision remained. At this time I was working as "acting CTO" for a very small web consultancy called Nitobi (we gave ourselves joke titles and mine was actually SPACELORD!1!!). The iPhone SDK, not yet called iOS SDK, was just released and a few of my colleagues found themselves at Adobe in San Francisco for the iPhone Dev Camp. They arrived with the ambiguous idea to discover if it actually was possible to realize web technology for app development. Rob Ellis, Brock Whitten, and Eric Osterly succeeded in bridging the UIWebView to make native calls, and the first implementation of PhoneGap was born. A very short time later, Joe Bowser built an Android implementation. Dave Johnson, Nitobi's "real CTO", followed quickly with the BlackBerry implementation. Herein, PhoneGap got real. And then, less than a year from the first commits, in the spring of 2009, I found myself giving one of the first PhoneGap presentations at the first JSConf, and despite me being terribly scared, people loved it.

Perhaps developers only loved the meme and the expletive-heavy presentation style I relied on to mask my terror. But perhaps developers really loved being treated like cohorts in a bigger plan, and respected by a technology instead of being locked into another proprietary ghetto.

We were, and still are, web developers with a strong bias for composing our own stacks from open source code. We want to be writing web apps, with technology of our choosing, and not paying for the right to do so. We didn't want a PhoneGap to exist, and so it is the goal of the project to see this thing through to obsolescence. This work continues under the stewardship of the Apache Software foundation under the name Apache Cordova. Defining our vision, and planning our execution to our end has been my primary role in the project since inception, in addition to meme, and expletive, heavy presentations.

Today PhoneGap is a robust, mature, well-tested, and a regularly released software project. There are 30 full-time core committers with us at Apache from a range of sponsoring organizations, and many hundreds more pitching in every day. All major operating systems are supported, our docs are comprehensive, the CLI tooling makes common mobile dev workflows trivial, the APIs cover all the common device capabilities, and we have a well documented plugin interface for extending beyond the browser.

Matt Gifford has been a long time supporter and hacker of PhoneGap, and his book brings his hard-won experience back to you. In this text you will find the specific areas you need to tackle, be it accessing the device sensors (such as geolocation) or the system data (such as the filesystem or perhaps the phone contacts). You will have a handy reference for dealing with rich media such as images, audio, and video.

Writing HTML, CSS, and JavaScript can be daunting and Matt has thankfully given you two great options to get started with, they are, XUI and jQuery Mobile. Finally, when you need to take your app beyond default PhoneGap and expose native capability you can learn all about the PhoneGap Plugin API.

Building applications for mobile devices is hard work but using PhoneGap makes that job a whole lot easier, and more portable to the inevitable future web. Matt's book will help you get there now. Have fun, and if you need any help at all, don't hesitate to find me (or Matt) online.

Brian Leroux,
Senior Product Manager, PhoneGap Lead
and SPACELORD!1!!, Adobe Systems Ltd

About the Author

Matt Gifford is an RIA developer from Cambridge, England, who specializes in ColdFusion, web application, and mobile development. With over ten years industry experience across various sectors, Matt is owner of Monkeh Works Ltd. (`www.monkehworks.com`).

A regular presenter at national and international conferences, he also contributes articles and tutorials in leading international industry magazines, as well as publishing on his blog (`www.mattgifford.co.uk`).

As an Adobe Community Professional for ColdFusion, Matt is an advocate of community resources and industry-wide knowledge sharing, with a focus on encouraging the next generation of industry professionals.

Matt is the author of *Object-Oriented Programming in ColdFusion* and numerous open source applications, including the popular *monkehTweets* twitter API wrapper.

> First and foremost, my thanks go to all the talented PhoneGap developers for their innovative and inspiring project. Without you this book would be a ream of blank pages.

About the Reviewers

Raymond Camden is a senior developer evangelist for Adobe. His work focuses on web standards, mobile development, and ColdFusion. He's a published author and presents at conferences and user groups on a variety of topics. Raymond can be reached at his blog (`www.raymondcamden.com`), `@cfjedimaster` on Twitter, or via e-mail at `raymondcamden@gmail.com`.

Shaun Dunne is a developer working for SapientNitro in London, UK and has been coding since 2008 with a passion for JavaScript and all the frontend goodness. Working for a large agency, over the past few years, Shaun has had the chance to use various web technologies to build large scale applications and found a passion for getting other people excited about the web.

Shaun has been hacking the mobile web for a couple of years, trying and testing all the tools available, and sharing his discoveries where he can to ensure that others are aware of what is available to use and in what situation.

When he's not working or spending some family time with his kids, he can usually be found on the web, tinkering, blogging, and building things. He's currently working on his own book, a self-published title about SASS and Friends called *UberCSS* due to be released in the Winter of 2012.

Andrey Rebrov started as a software developer in Magenta Technology – a big British software company specializing in enterprise java solutions and has worked with them for more than three years. Andrey now works as agile engineering coach in ScrumTrek, Russia.

As an engineering coach, he helps teams with learning and adopting XP practice, as TDD. A big part of his job is building the Russian Software Craftsmanship Community.

At work, he also uses innovation and agile games and at the moment, works on innovation games popularization in Russia and Russian communities.

Andrey has even worked on *PhoneGap: Beginner's Guide*.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- ▸ Fully searchable across every book published by Packt
- ▸ Copy and paste, print and bookmark content
- ▸ On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

*Dedicated to Arthur George Wood. Thank you for so many happy memories,
for a wonderful childhood and for showing me the man I want to be.
I love you, Granddad.*

Table of Contents

Table of Contents

Preface

We live in an ever-evolving technological landscape, and the transition from the *traditional* web for desktop machines to mobile devices is now of more importance than ever. With the constant advancement in mobile technology and device capabilities, as well as increasing user adoption and the preference to access content or interact with services through the mobile format, it is not surprising that more organizations and individual developers want to hook into this exciting format, whether it's for marketing purposes, the creation of an amazing new application to generate a revenue stream and financial income, or simply to experiment with the software and solutions available.

Which platform do you target? Which language do you write in? The implications of developing mobile applications can raise questions such as these. It may mean that you have to consider learning a new language such as Objective-C, Java, or C++ to create your applications for each platform. This alone potentially brings with it a number of costs: the financial cost of learning a new language, including time and resource material, and the cost of managing your development workflow effectively. If we then consider pushing the same application to a number of platforms and operating systems, these costs increase and the management of each codebase becomes harder to maintain and control.

PhoneGap aims at removing these complications and the worry of having to develop platform-specific applications using the supported native language for each operating system by letting developers build cross-platform applications using HTML, CSS, and JavaScript, existing web technologies that are familiar to most if not all of us.

This drastically opens the gateway to creating natively installed mobile applications to all web developers and designers, empowering them to use the language skills they already have to develop something specifically for mobile platforms.

We can then add into this the ability to tap into the device's native functionality such as geolocation and GPS, accelerometer, camera, video, and audio capture among other capabilities, implemented using the PhoneGap JavaScript API, and your HTML applications instantly become detailed apps with incredibly powerful features.

PhoneGap Mobile Application Development Cookbook will demonstrate a variety of examples to help you enhance your applications using the PhoneGap API. This book contains everything you need to get started with, to experience mobile application development using the PhoneGap library through the step-by-step examples and recipes found within.

PhoneGap or Cordova

Throughout this book you may find that the terms Cordova and PhoneGap are used interchangeably. Both refer to exactly the same open source platform and library to enable you to create native mobile applications built using HTML, JavaScript, and CSS.

In 2011, the PhoneGap codebase moved to an open source Apache Software Foundation project under the name Cordova. Adobe still distributes the library under the PhoneGap name. Although both of the project names are referenced in this publication, it is by design and not meant to cause confusion. Essentially, both the PhoneGap and Cordova projects are the same, and refer to the same free, open source library.

Brian Leroux has also written a blog post outlining the name change and the differences between the two projects and any impact if at all they may have on developers, project contributors, and the PhoneGap community in general.

```
http://phonegap.com/2012/03/19/phonegap-cordova-and-what%E2%80%99s-
in-a-name/
```

Help is at hand

While a lot of information is included in this book to cover the various methods and functions available through the PhoneGap library, there may be features that aren't covered here that you need answers for. During the writing process for this book, the project itself went through a number of version releases, the latest being version 2.0, and as new versions are released, inevitably some properties, minor functions, and details change.

If you do require assistance with any PhoneGap projects or code, there are a few extra resources available to you to ensure you constantly get the latest information.

First, the official PhoneGap documentation, available at `http://docs.phonegap.com/en/2.1.0/index.html`, covers available API methods, features, and properties. While the material may cover some of the same ground, if for any reason something within this publication is puzzling or causing confusion, check out the official documentation for a second explanation and some extra clarity.

Second, the PhoneGap Google group forum, available at `groups.google.com/group/phonegap`, provides an open discussion list for PhoneGap developers, contributors, and users covering a wide variety of topics. Here you can post questions and see what issues other community members may be facing. The PhoneGap community is passionate, helpful and considerate, and someone will be able to help you. You may even be able to help others with their issues. Knowledge and experience is better when it is shared.

What this book covers

Chapter 1, *Movement and Location: Using the Accelerometer and Geolocation Sensors*, demonstrates how we can create applications that have the ability to determine a user's geographic location, as well as detecting movement using the device's accelerometer.

Chapter 2, *File System, Storage, and Local Databases*, provides the reader with the details required to read and write files to the device storage, create and manage SQLite databases, upload and download files to and from remote servers, and store application content using local storage APIs.

Chapter 3, *Working with Audio, Images, and Video*, discusses how to create multimedia-rich applications using the device capabilities and hardware to capture audio, video and images, as well as audio playback and streaming.

Chapter 4, *Working with Your Contacts*, describes how to access and work with the contacts database on your device.

Chapter 5, *Hook into Native Events*, demonstrates how to employ and extend the native events on your device to manage pausing and resuming your application, as well as creating custom functions to detect connectivity changes and device battery levels.

Chapter 6, *Working with XUI*, explains the features and methods available to use from the lightweight XUI JavaScript library.

Chapter 7, *User Interface Development with jQuery Mobile*, guides the user through the processes of using the jQuery Mobile framework to create a simple mobile application, including page transitions and "near-native" user interface elements.

Chapter 8, *Extending PhoneGap with Plugins*, describes how to extend the PhoneGap API and available methods by creating custom plugins.

Chapter 9, *Development Tools and Testing*, demonstrates a number of options available to create your mobile application development environment and the tools available to help streamline your workflow.

What you need for this book

You will need a computer, a web browser, and a code editor of your choice. Some code editors include features and functions that have been designed to assist you specifically with PhoneGap mobile application development, and some of these are described in *Chapter 9*, *Development Tools and Testing*, of this book. Dreamweaver CS5.5 and CS6, for example, include support for PhoneGap and the PhoneGap Build service.

Ultimately, you can develop mobile applications using the Cordova/PhoneGap library for free. It costs nothing to download the library, and you can write HTML, CSS, and JavaScript using any text editor you have available or are comfortable with. Even running the applications on a device emulator won't cost you anything, as they are also freely available for download.

Who this book is for

This book is for anyone with prior exposure to HTML, CSS, and JavaScript development, regardless of skill set, and for anyone looking to enter the world of mobile application development, or those wishing to enhance their existing HTML applications with mobile-specific features and functions.

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: " Set the `id` attribute to `contactList`, the `data-role` to `listview`, and the `data-inset` attribute to `true`. "

A block of code is set as follows:

```
function alphabeticalSort(a, b) {
  if (a.name.formatted < b.name.formatted){
    return -1;
  }else if (a.name.formatted > b.name.formatted){
    return  1;
  }else{
    return 0;
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<div data-role="page" id="contact-info">

    <div id="contact_header" data-role="header">
      <a href="#contacts-home" id="back" data-icon="back"
        data-direction="reverse">Back</a>
        <h1></h1>
    </div>


    </div>
```

Any command-line input or output is written as follows:

```
power ac off
power status not-charging
power capacity 10
```

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: " select the **Settings** tab within the project view and click on the **enable hydration** 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 through 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

This book contains many code samples throughout the recipes. To make life a little easier for you, the complete code for each recipe is available to download from the public GitHub repository setup for this book: `https://github.com/coldfumonkeh/PhoneGap-Cookbook`. The GitHub repository may be updated as any possible typing mistakes are discovered in the book. As a result, it is a possibility that the code may not exactly match the text in the book.

If you are not familiar with GitHub, simply click on the **Downloads** tab and then either **Download as zip** or **Download as tar.gz** to get an archived collection of all of the files.

You can extract these files to any location on your local machine where you easily open them.

You can also 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/support`, 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 to our website, or added to any list of existing errata, under the Errata section of that title.

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

Movement and Location: Using the Accelerometer and Geolocation Sensors

In this chapter we will cover the following recipes:

- ▶ Detecting device movement using the accelerometer
- ▶ Adjusting the accelerometer sensor update interval
- ▶ Updating a display object position through accelerometer events
- ▶ Obtaining device geolocation sensor information
- ▶ Adjusting the geolocation sensor update interval
- ▶ Retrieving map data through geolocation coordinates
- ▶ Creating a visual compass to show the device direction

Introduction

Mobile devices are incredibly powerful tools that not only allow us to make calls and send messages, but can also help us navigate and find out where we are in the world, thanks to the accelerometer, geolocation, and other sensors.

This chapter will explore how we can access these sensors and make use of this exposed functionality in helpful applications that can be built in any IDE using the PhoneGap API.

Detecting device movement using the accelerometer

The accelerometer captures device motion in the *x, y,* and *z* -axis directions. The accelerometer is a motion sensor that detects the change (delta) in movement relative to the current device orientation.

How to do it...

We will use the accelerometer functionality from the PhoneGap API to monitor the feedback from the device:

1. First, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Accelerometer Data</title>

    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>

    <h1>Accelerometer Data</h1>

    <div id="accelerometerData">Obtaining data...</div>

  </body>
</html>
```

2. Below the `Cordova JavaScript` reference, write a new JavaScript tag block and define an event listener to ensure the device is ready and the native code has loaded before continuing:

```
<script type="text/javascript">

  // Set the event listener to run
// when the device is ready
   document.addEventListener(
      "deviceready", onDeviceReady, false);



</script>
```

3. We will now add in the `onDeviceReady` function which will run the `getCurrentAcceleration` method when the native code has fully loaded:

```
// The device is ready so let's
// obtain the current accelerometer data
function onDeviceReady() {
     navigator.accelerometer.getCurrentAcceleration(
       onSuccess, onError);
}
```

4. Include the `onSuccess` function to handle the returned information from the accelerometer.

5. We now define the `accelerometer div` element to the `accElement` variable to hold our generated accelerometer results.

6. Next, we assign the returned values from the `acceleration` object as the HTML within the `accelerometer div` element for display to the user. The available properties are accessed through the `acceleration` object and applied to the string variable:

```
// Run after successful transaction
// Let's display the accelerometer data
function onSuccess(acceleration) {
  var accElement =
    document.getElementById('accelerometerData');

  accElement.innerHTML  =
    'Acceleration X: ' + acceleration.x + '<br />' +
    'Acceleration Y: ' + acceleration.y + '<br />' +
    'Acceleration Z: ' + acceleration.z + '<br />' +
    'Timestamp: '      + acceleration.timestamp;
}
```
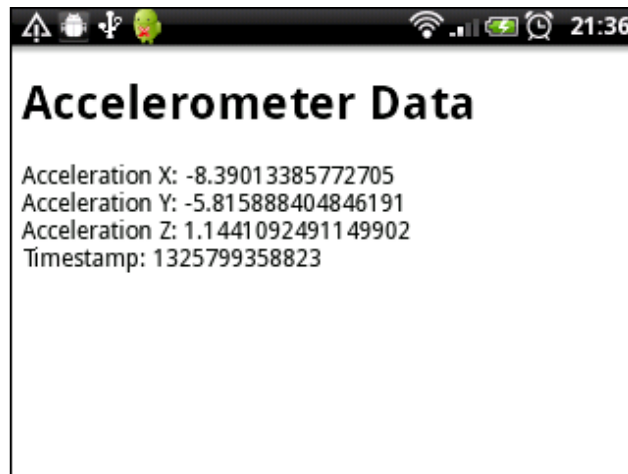
9

7. Finally, include the `onError` function to deal with any possible issues:

```
// Run if we face an error
// obtaining the accelerometer data
function onError(error) {
  // Handle any errors we may face
  alert('error');
}
```

8. When we run the application on a device, the output will look something like the following screenshot:

How it works...

By registering an event listener to the `deviceready` event we are ensuring that the JavaScript code does not run before the native PhoneGap code is executed. Once ready, the application will call the `getCurrentAcceleration` method from the accelerometer API, providing two methods to handle successful transactions and errors respectively.

The `onSuccess` function returns the obtained acceleration information in the form of the following four properties:

- `acceleration.x`: A `number` value, registered in *m/s^2*, that measures the device acceleration across the *X* axis. This is the movement from left to right when the device is placed with the screen facing an upright position. Positive acceleration is obtained as the device is moved to the right, whereas a negative movement is obtained when the device is moved to the left.

- `acceleration.y`: A `Number` value, registered in *m/s^2*, that measures the device acceleration across the *Y* axis. This is the movement from bottom to top when the device is placed with the screen facing an upright position. Positive acceleration is obtained as the device is moved upwards, whereas a negative movement is obtained when the device is moved downwards.

- `acceleration.z`: A `Number` value, registered in *m/s^2*, that measures the device acceleration across the *Z* axis. This is a perpendicular from the face of the device. Positive acceleration is obtained when the device is moved to face towards the sky, whereas a negative movement is obtained when the device is pointed towards the Earth.

- `acceleration.timestamp`: A `DOMTimeStamp` object that measures the amount of milliseconds from the point of the application's initialization. This could be used to store, update, and track changes over a period of time since the last accelerometer update.

The following figure shows the *X, Y*, and *Z* Axes in relation to the device:
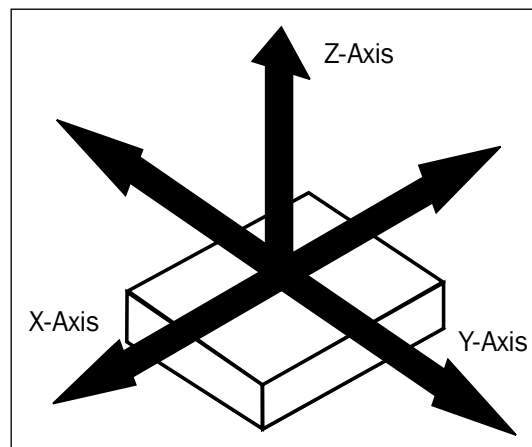
The `acceleration.x`, `acceleration.y` and `acceleration.z` values returned from the acceleration object previously mentioned include the effect of gravity, which is defined as precisely 9.81 meters per second squared (9.81 m/s^2).

There's more...

Accelerometer data obtained from the device has been used to a great effect in mobile handset games that require balance control and detection of movement including steering, control views, and tilting objects.

> You can check out the official Cordova documentation covering the `getCurrentAcceleration` method and obtaining accelerometer data at: `http://docs.phonegap.com/en/2.0.0/cordova_ accelerometer_accelerometer.md.html#accelerometer. getCurrentAcceleration`.

Adjusting the accelerometer sensor update interval

The `getCurrentAcceleration` method obtains the data from the accelerometer at the time it was called – a single call to obtain a single response object. In this recipe, we'll build an application that allows us to set an interval to obtain a constant update from the accelerometer to detect continual movement from the device.

How to do it...

We will provide additional parameters to a new method available through the PhoneGap API to set the update interval:

1. Firstly, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
      <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />

   <title>Accelerometer Data</title>

     <script type="text/javascript"
         src="cordova-2.0.0.js"></script>

   <!-- Add PhoneGap script here -->

   </head>
```

```
<body>

    <h1>Accelerometer Data</h1>

    <div id="accelerometerData">Obtaining data...</div>

  </body>
</html>
```

2. Below the Cordova JavaScript reference, write a new JavaScript tag block, within which we'll declare a variable called `watchID`.

3. Next, we'll define an event listener to ensure the device is ready and the native code has loaded before continuing:

```
<script type="text/javascript">


  // The watch id variable is set as a
  // reference to the current 'watchAcceleration'
  var watchID = null;


  // Set the event listener to run
// when the device is ready
  document.addEventListener(
      "deviceready", onDeviceReady, false);


</script>
```

4. We will now add in the `onDeviceReady` function which will run a method called `startWatch` once the native code has fully loaded:

```
// The device is ready so let's
// start watching the acceleration
function onDeviceReady() {
  startWatch();
}
```

5. We'll now write the `startWatch` function. Firstly, we'll create a variable called `options` to hold the optional `frequency` parameter, set to 3000 milliseconds (three seconds).

6. We will then set the initial `disabled` properties of two buttons that will allow the user to start and stop the acceleration detection.

7. Next we will assign the `watchAcceleration` to the previously defined `watchID` variable. This will allow us to check for a value or if it is still set to `null`.

8. As well as defining the success and error function names we are also sending the `options` variable into the method call, which contains the `frequency` value:

```
// Watch the acceleration at regular
// intervals as set by the frequency
function startWatch() {

  // Set the frequency of updates
  // from the acceleration
  var options = { frequency: 3000 };

  // Set attributes for control buttons
  document.getElementById('startBtn').disabled = true;
  document.getElementById('stopBtn').disabled = false;

  // Assign watchAcceleration to the watchID variable
  // and pass through the options array
  watchID = navigator.accelerometer.watchAcceleration(
          onSuccess, onError, options);
}
```

9. With the `startWatch` function written, we now need to provide a method to stop the detection of the acceleration. This firstly checks the value of the `watchID` variable. If this is not `null` it will stop watching the acceleration using the `clearWatch` method, passing in the `watchID` parameter before resetting this variable back to `null`.

10. We then reference the `accelerometer div` element and set its value to a user-friendly message.

11. Next, we reassign the `disabled` properties for both of the control buttons to allow the user to start watching again:

```
// Stop watching the acceleration
function stopWatch() {

  if (watchID) {
    navigator.accelerometer.clearWatch(watchID);
      watchID = null;

    var element =
      document.getElementById('accelerometerData');

    element.innerHTML =
      'No longer watching your acceleration.'

    // Set attributes for control buttons
    document.getElementById('startBtn').disabled = false;
```

```
        document.getElementById('stopBtn').disabled = true;


    }


}
```

12. Now we need to create the `onSuccess` method, which will be run after a successful update response. We assign the returned values from the `acceleration` object as the HTML within the `accelerometer div` element for display to the user. The available properties are accessed through the `acceleration` object and applied to the string variable:

```
// Run after successful transaction
// Let's display the accelerometer data
function onSuccess(acceleration) {
  var element = document.getElementById('accelerometerData');
  element.innerHTML =
    'Acceleration X: ' + acceleration.x + '<br />' +
    'Acceleration Y: ' + acceleration.y + '<br />' +
      'Acceleration Z: ' + acceleration.z + '<br />' +
      'Timestamp: '      + acceleration.timestamp + '<br />';
}
```

13. We also need to supply the `onError` method to catch any possible issues with the request. Here we will output a user-friendly message, setting it as the value of the `accelerometerData div` element:

```
// Run if we face an error
// obtaining the accelerometer data
function onError() {
  // Handle any errors we may face
  var element = document.getElementById('accelerometerData');
    element.innerHTML =
    'Sorry, I was unable to access the acceleration data.';
}
```

14. Finally, we will add in the two button elements, both of which will have an `onClick` attribute set to either start or stop watching the device acceleration:

```
<body>
    <h1>Accelerometer Data</h1>

    <button id="startBtn"
        onclick="startWatch()">start</button>

    <button id="stopBtn"
        onclick="stopWatch()">stop</button>
```
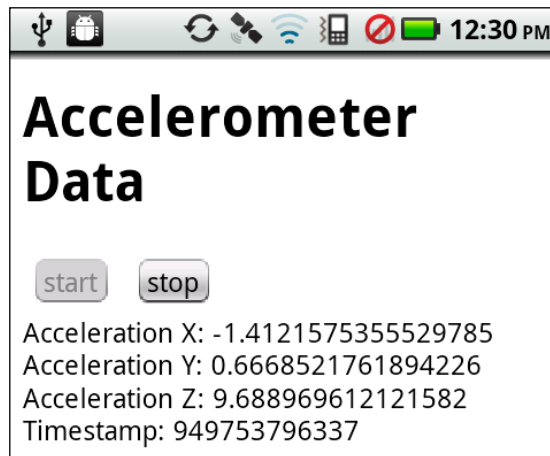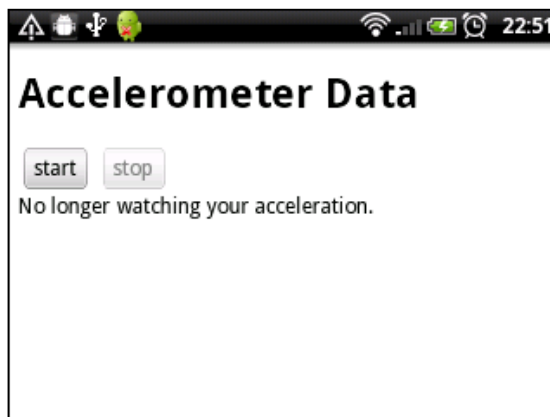
15

```
<div id="accelerometerData">Obtaining data...</div>

</body>
```

15. The results will appear similar to the following screenshot:

16. Stopping the acceleration watch will look something like the following screenshot:


How it works...

By registering an event listener to the `deviceready` event we are ensuring that the JavaScript code does not run before the native PhoneGap code is executed. Once ready, the application will call the `startWatch` function, within which the desired frequency interval for the acceleration updates is set.

16

The `watchAcceleration` method from the PhoneGap API retrieves the device's current acceleration data at the interval specified. If the interval is not passed through, it defaults to 10000 milliseconds (ten seconds). Each time an update has been obtained, the `onSuccess` method is run to handle the data as you wish, in this case displaying the results on the screen.

The `watchID` variable contains a reference to the watch interval and is used to stop the watching process by being passed in to the `clearWatch` method from the PhoneGap API.

There's more...

In this example the `frequency` value for the accelerometer update interval was set at 3000 milliseconds (three seconds). Consider writing a variation on this application that allows the user to manually change the interval value using a slider or by setting the desired value into an input box.
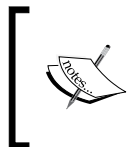
> You can find out more about the `watchAcceleration` method via the official Cordova documentation: `http://docs.phonegap.com/en/2.0.0/cordova_accelerometer_accelerometer.md.html#accelerometer.watchAcceleration`.

Updating a display object position through accelerometer events

Developers can make use of the accelerometer sensor and continual updates provided by it for many things including motion-detection games as well as updating the position of an object on the screen.

How to do it...

We will use the device's accelerometer sensor on continual update to move an element around the screen as a response to device movement. This is achieved through the following steps:

1. Let's start by creating our initial HTML layout. Include the Cordova JavaScript reference in the `head` tag to import the required library.

2. Within the `body` tag create two `div` elements. Set the first with the `id` attribute equal to dot. This will be the element we move around the screen of the device.

3. The second `div` element will have the ID of `accelerometerData` and will be the container into which our returned acceleration data will be output:

```
<!DOCTYPE html>
<html>
  <head>
```

```
        <meta name="viewport" content="user-scalable=no,
              initial-scale=1, maximum-scale=1,
              minimum-scale=1, width=device-width;" />

    <title>Accelerometer Movement</title>
    <script type="text/javascript"
         src="cordova-2.0.0.js"></script>

  </head>
  <body>
    <h1>Accelerometer Movement</h1>

    <div id="dot"></div>

    <div id="accelerometerData">Obtaining data...</div>

  </body>
</html>
```

4. We can now start with our custom scripting and PhoneGap implementation. Add a `script` tag block before the closing `head` tag to house our code:

```
<head>
    <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />

    <title>Accelerometer Movement</title>
  <script type="text/javascript"
       src="cordova-2.0.0.js"></script>

  <script type="text/javascript">

  </script>

</head>
```

5. Before we dive into the core code, we need to declare some variables. Here we are setting a default value for `watchID` as well as the radius for the circle display object we will be moving around the screen:

```
// The watch id variable is set as a
// reference to the current `watchAcceleration`
var watchID = null;

// The radius for our circle object
var radius   = 50;
```

18

6. We now need to declare the event listener for PhoneGap, as well as the `onDeviceReady` function, which will run once the native PhoneGap code has been loaded:

```
// Set the event listener to run when the device is ready
document.addEventListener("deviceready",
        onDeviceReady, false);

// The device is ready so let's
// start watching the acceleration
function onDeviceReady() {

  startWatch();

}
```

7. The `onDeviceReady` function will execute the `startWatch` method, which sets required the `frequency` value for accelerometer updates and makes the request to the device to obtain the information:

```
// Watch the acceleration at regular
// intervals as set by the frequency
function startWatch() {

  // Set the frequency of updates from the acceleration
   var options = { frequency: 100 };

  // Assign watchAcceleration to the watchID variable
  // and pass through the options array
  watchID =
    navigator.accelerometer.watchAcceleration(
      onSuccess, onError, options);
}
```

8. With the request made to the device we now need to create the success and error handling methods. The `onSuccess` function is first, and this will deal with the movement of our object around the screen.

9. To begin with we need to declare some variables that manage the positioning of our element on the device:

```
function onSuccess(acceleration) {

  // Initial X Y positions
  var x = 0;
  var y = 0;
```

```
        // Velocity / Speed
        var vx = 0;
        var vy = 0;

        // Acceleration
        var accelX = 0;
        var accelY = 0;

        // Multiplier to create proper pixel measurements
        var vMultiplier   =      100;

        // Create a reference to our div elements
        var dot = document.getElementById('dot');
        var accelElement =
              document.getElementById('accelerometerData');



        // The rest of the code will go here

    }
```

10. The returned `acceleration` object contains the information we need regarding the position on the *x* and *y* axes of the device. We can now set the acceleration values for these two axis into our variables and work out the velocity for movement.

11. To correctly interpret the acceleration results into pixels we can use the `vMultiplier` variable to convert the `x` and `y` into pixels:

```
accelX = acceleration.x;
accelY = acceleration.y;

vy = vy + -(accelY);
vx = vx + accelX;

y = parseInt(y + vy * vMultiplier);
x = parseInt(x + vx * vMultiplier);
```

12. We need to ensure that our display object doesn't move out of sight and to keep it within the bounds of the screen:

```
if (x<0) { x = 0; vx = 0; }
if (y<0) { y = 0; vy = 0; }

if (x>document.documentElement.clientWidth-radius) {
  x = document.documentElement.clientWidth-radius; vx = 0;
}
```

```
if (y>document.documentElement.clientHeight-radius) {
  y = document.documentElement.clientHeight-radius; vy = 0;
}
```

13. Now that we have the correct `x` and `y` coordinates we can apply them to the style of the `dot` element position. Let's also create a string message containing the properties returned from the `acceleration` object as well as the display coordinates that we have created:

```
// Apply the position to the dot element
dot.style.top  = y + "px";
dot.style.left = x + "px";

// Output the acceleration results to the screen
accelElement.innerHTML =
  'Acceleration X: '      + acceleration.x + '<br />' +
  'Acceleration Y: '      + acceleration.y + '<br />' +
  'Acceleration Z: '      + acceleration.z + '<br />' +
  'Timestamp: '           + acceleration.timestamp + '<br />' +
  'Move Top: '                   + y + 'px<br />' +
  'Move Left: '                  + x + 'px';
```

14. Our call to the accelerometer also requires the error handler, so let's write that now. We'll create a simple string message and insert it into the `div` element to inform the user that we encountered a problem:

```
// Run if we face an error
// obtaining the accelerometer data
function onError() {

  // Handle any errors we may face
  var accelElement =
      document.getElementById('accelerometerData');

  accelElement.innerHTML =
    'Sorry, I was unable to access the acceleration data.';
}
```

15. Finally, we'll add in some CSS to create the dot marker used to display the position on our device:
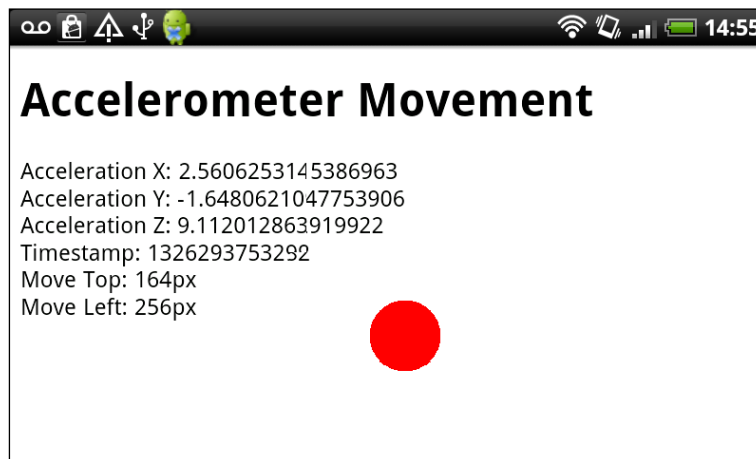
```
<style>
div#dot {
  border-radius: 14px;
  width: 25px;
  height: 25px;
  background: #ff0000;
```

21

```
    position: absolute;
    top: 0px;
    left: 0px;
}
</style>
```

16. When we run the application, we can move the element around the screen by titling
the device. This would look something like the following screenshot:

How it works...

By implementing a constant request to watch the acceleration and retrieve movement results
from the device, we can pick up changes from the accelerometer sensor. Through some simple
JavaScript we can respond to these changes and update the position of an element around the
screen based upon the returned sensor information.

In this recipe we are easily changing the position of the dot element by calculating the
correct *X* and *Y* axes to place it on the screen. We are also taking extra care to ensure that the
element stays within the bounds of the screen by using some conditional statements to check
the current position, the radius of the element, and the dimensions of the screen itself.

See also

▶ The *Detecting device movement using the accelerometer* recipe

Obtaining device geolocation sensor information

Geolocation and the use of **Global Positioning Satellites** (**GPS**) allow developers to create dynamic real-time mapping, positioning, and tracking applications. Using the available geolocation methods we can retrieve a detailed set of information and properties to create location-aware applications. We can obtain the user's location if they are connected via the mobile data network or Wi-Fi.

How to do it...

We will use the geolocation functionality from the PhoneGap API to monitor the feedback from the device and obtain the relevant location information:

1. Firstly, create the initial HTML layout and include the required script reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />

  <title>Geolocation Data</title>

    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

  <!-- Add PhoneGap script here -->

  </head>
  <body>

    <h1>Geolocation Data</h1>

    <div id="geolocationData">Obtaining data...</div>

  </body>
</html>
```

2. Write a new JavaScript tag block beneath the Cordova JavaScript reference. Within this let's define an event listener to ensure the device is ready:

```
<script type="text/javascript" charset="utf-8">


// Set the event listener to run when the device is ready
document.addEventListener(
        "deviceready", onDeviceReady, false);

</script>
```

3. Let's now add the `onDeviceReady` function. This will execute the `geolocation.getCurrentPosition` method from the PhoneGap API once the native code has fully loaded:

```
// The device is ready so let's
// obtain the current geolocation data
function onDeviceReady() {
  navigator.geolocation.getCurrentPosition(
        onSuccess, onError);
}
```

4. Include the `onSuccess` function to handle the returned `position` object from the geolocation request.

5. Let's then create a reference to the `geolocationData` div element and assign it to the variable `geoElement`, which will hold our generated position results.

6. Next we can assign the returned values as a formatted string, which we'll set as the HTML content within the `geolocationData` div element. The available properties are accessed through the `position` object:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {

  var geoElement =
        document.getElementById('geolocationData');

  geoElement .innerHTML =
    'Latitude: '  + position.coords.latitude + '<br />' +
      'Longitude: ' + position.coords.longitude + '<br />' +
      'Altitude: '  + position.coords.altitude + '<br />' +
      'Accuracy: '  + position.coords.accuracy + '<br />' +
      'Altitude Accuracy: ' +
        position.coords.altitudeAccuracy + '<br />' +
```

24

```
        'Heading: ' + position.coords.heading  + '<br />' +
        'Speed: '   + position.coords.speed + '<br />' +
        'Timestamp: ' + position.timestamp + '<br />';

}
```

7. Finally, let's include the `onError` function to handle any possible errors that may arise.

8. Depending on the existence of an error, we will use the value of the returned error code to determine which message to display to the user. This will be set as the HTML content of the `geolocationData` `div` element:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {

    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
      errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not ' +
          'be determined.';
      break;
      case 3: // TIMEOUT
        errString =
          'Unable to obtain the location within the ' +
          'specified time allocation.';
      break;
      default: // UNKOWN_ERROR
        errString =
          'Unable to obtain the location of the ' +
```
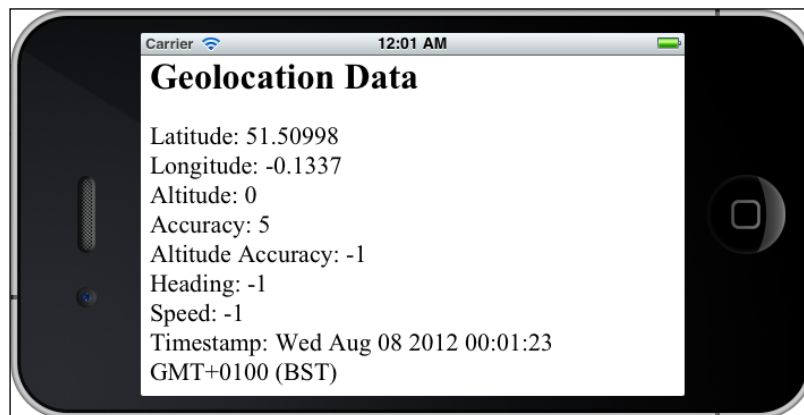
25

```
            'device due to an unknown error.';
        break;
    }

}

// Handle any errors we may face
var element = document.getElementById('geolocationData');
 element.innerHTML = errString;

}
```
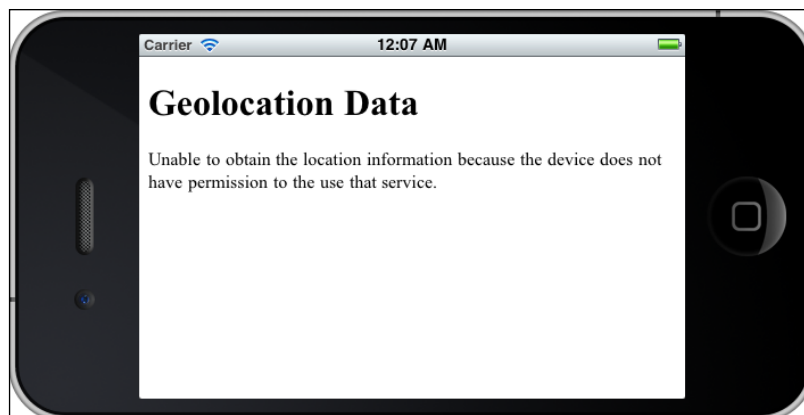
9. When we run the application on a device, the output will look something like the following screenshot:

10. If we face any errors, the resulting output will look something like the following screenshot:

How it works...

As soon as the device is ready and the native PhoneGap code has been initiated on the device, the application will execute the `getCurrentPosition` method from the `geolocation` API. We have defined an `onSuccess` method to manage the output and handling of a successful response, and have also specified an `onError` method to catch any errors and act accordingly.

The `onSuccess` method returns the obtained geolocation information in the form of the `position` object, which contains the following properties:

- `position.coords`: A `Coordinates` object that holds the geographic information returned from the request. This object contains the following properties:
    - `latitude`: A `Number` value ranging from -90.00 to +90.00 that specifies the latitude estimate in decimal degrees.
    - `longitude`: A `Number` value ranging from -180.00 to +180.00 that specifies the longitude estimate in decimal degrees.
    - `altitude`: A `Number` value that specifies the altitude estimate in meters above the **World Geodetic System** (**WGS**) 84 ellipsoid. Optional.
    - `accuracy`: A `Number` value that specifies the accuracy of the latitude and longitude accuracy in meters. Optional.
    - `altitudeAccuracy`: A `Number` value that specifies the accuracy of the altitude estimate in meters. Optional.
    - `heading`: A `Number` value that specifies the current direction of movement in degrees, counting clockwise in relation to true north. Optional.
    - `speed`: A `Number` value that specifies the current ground speed of the device in meters per second. Optional.

- `position.timestamp`: A `DOMTimeStamp` object that signifies the time that the geolocation information was received and the `Position` object was created.

    The properties available within the `position` object are quite comprehensive and detailed.

For those marked as 'optional', the value will be set and returned as `null` if the device cannot provide a value.

The `onError` method returns a `PositionError` object if an error is detected during the request. This object contains the following two properties:

- `code`: A `Number` value that contains a numeric code for the error.
- `message`: A `String` object that contains a human-readable description of the error.

The errors could relate to insufficient permissions needed to access the geolocation sensors on the device, the inability to locate the device due to issues with obtaining the necessary GPS information, a timeout on the request, or the occurrence of an unknown error.

27

There's more...

The exposed geolocation API accessible through PhoneGap is based on the *W3C Geolocation API Specification*. Many modern browsers and devices already have this functionality enabled. If any device your application runs on already implements this specification, it will use the built-in support for the API and not PhoneGap's implementation.
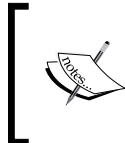
> You can find out more about Geolocation and the `getCurrentPosition` method via the official Cordova documentation at: `http://docs.phonegap.com/en/2.0.0/cordova_geolocation_geolocation.md.html#geolocation.getCurrentPosition`.

Adjusting the geolocation sensor update interval

Through the use of the `getCurrentPosition` method, we can retrieve a single reference to the device location using GPS coordinates. In this recipe, we'll create the functionality to obtain the current location based on a numeric interval to receive constant updated information.

How to do it...

We are able to pass through an optional parameter containing various arguments to set up an interval and improve accuracy:

1. Create the HTML layout for the application, including the required `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
      <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />
   <title>Geolocation Data</title>

    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

<!-- Add PhoneGap script here -->

</head>
```

```
<body>
  <h1>Geolocation Data</h1>

  <div id="geolocationData">Obtaining data...</div>

</body>
</html>
```

2. Below the Cordova JavaScript reference, write a new JavaScript tag block. Within this we'll declare a new variable called `watchID`.

3. Next, we'll write the event listener to continue once the device is ready:

```
<script type="text/javascript">

  // The watch id variable is set as a
  // reference to the current 'watchPosition'
  var watchID = null;

  // Set the event listener to run
// when the device is ready
  document.addEventListener(
      "deviceready", onDeviceReady, false);


</script>
```

4. Let's now add the `onDeviceReady` function which will execute a method called `startWatch`, written as follows:

```
// The device is ready so let's
// start watching the position
function onDeviceReady() {

  startWatch();

}
```

5. We can now create the `startWatch` function. Firstly, let's create the `options` variable to hold the optional parameters we can pass through to the method. Set the `frequency` value to 5000 milliseconds (five seconds) and set `enableHighAccuracy` to true.

6. Next we will assign the `watchPosition` method to the previously defined variable `watchID`. We'll use this variable to check if the location is currently being watched.

29

7. To pass through the extra parameters we have set, we send the `options` variable into the `watchPosition` method:

```
function startWatch() {

  // Create the options to send through
  var options = {
      enableHighAccuracy: true
  };

  // Watch the position and update
  // when a change has been detected
  watchID =
    navigator.geolocation.watchPosition(
      onSuccess, onError, options);

}
```

8. With the initial call methods created, we can now write the `onSuccess` function, which is executed after a successful response. The `position` object from the response is sent through as an argument to the function.

9. Declare some variables to store detailed information obtained from the response in the form of the `timestamp`, `latitude`, `longitude`, and `accuracy` variables. We'll also create the element variable to reference the `geolocationData div` element, within which our information will be displayed.

10. The returned information is then assigned to the relevant variables by accessing the properties from the `position` object.

11. Finally, we apply the populated variables to a concatenated string which we'll set as the HTML within the `div` element:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {

  var timestamp, latitude, longitude, accuracy;

  var element = document.getElementById('geolocationData');

  timestamp = new Date(position.timestamp);
  latitude = position.coords.latitude;
  longitude = position.coords.longitude;
  accuracy    = position.coords.accuracy;

  element.innerHTML +=
      '<hr />' +
```

```
            'Timestamp: ' + timestamp + '<br />' +
            'Latitude: ' + latitude   + '<br />' +
            'Longitude: ' + longitude + '<br />' +
            'Accuracy: ' + accuracy   + '<br />';
    }
```

12. With the `onSuccess` method created, let's now write the `onError` function to handle any errors that we may face following the response:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {
    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
        errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not be ' +
          'determined.';
      break;
      case 3: // TIMEOUT
        errString =
          'Unable to obtain the location within the ' +
          'specified time allocation.';
      break;
      default: // UNKOWN_ERROR
        errString =
          'Unable to obtain the location of the ' +
          'device to an unknown error.';
      break;
    }

  }
```

```
       // Handle any errors we may face
      var element = document.getElementById('geolocationData');
       element.innerHTML = errString;


   }
```

13. When we run the application, the output will be similar to the following screenshot:

The `watchPosition` method from the PhoneGap API runs as an asynchronous function, constantly checking for changes to the device's current position. Once a change in position has been detected, it will return the current geographic location information in the form of the `position` object.

With every successful request made on the continuous cycle, the `onSuccess` method is executed and formats the data for output onto the screen.

There's more...

There are three optional parameters that can be sent into either the `getCurrentPosition` or `watchPosition` method. They are as follows:

- `enableHighAccuracy`: A `Boolean` value that specifies whether or not you would like to obtain the best possible location results from the request. By default (false), the position will be retrieved using the mobile or cell network. If set to true, more accurate methods will be used to locate the device, for example, using satellite positioning.

- `timeout`: A `Number` value that defines the maximum length of time in milliseconds to obtain the successful response.

- `maximumAge`: A `Number` value that defines if a cached position younger than the specified time in milliseconds can be used.

> Android devices will not return a successful geolocation result unless `enableHighAccuracy` is set to `true`

Clearing the interval

The continual location requests can be stopped by the user or through the application using an interval timer by employing the use of the `clearWatch` method, available within the PhoneGap API. The method to clear the interval and stop watching location data is identical to the method used when clearing accelerometer data obtained from continual updates.

See also

- The *Adjusting the accelerometer sensor update interval* recipe

# Retrieving map data through geolocation coordinates

In this recipe, we will examine how to render a map on the screen and generate a marker based on latitude and longitude coordinates reported by the device geolocation sensors using the Google Maps API for JavaScript.

Getting ready

Before we can continue with coding the application in this recipe, we must first prepare the project and obtain access to the Google Maps services:

1. Firstly we need to sign up for a **Google Maps API key**. Visit `https://code.google.com/apis/console/` and log in with your Google account.

2. Select **Services** from the left-hand side menu and activate the **Google Maps API v3 service**.

3. Once the service has been activated you will be presented with your API key, available from the **API Access** page. You will find the key displayed in the **Simple API Access** section of the page, as shown in the following screenshot:
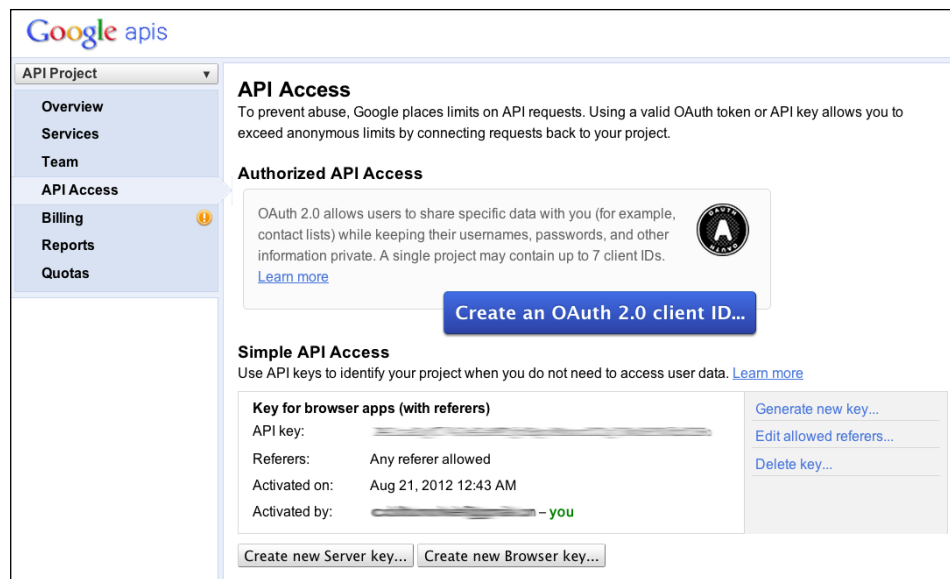
4. We can now proceed with the recipe.

> You can find out more about the Google Maps API from the official documentation: `https://developers.google.com/maps/documentation/javascript/`.

How to do it...

We'll use the device's GPS ability to obtain the geolocation coordinates, build and initialize the map canvas, and display the marker for our current position:

1. Create the basic HTML layout for our page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />
```

```
    <title>You are here...</title>
  </head>
  <body>


  </body>
</html>
```

2. Let's include the required JavaScript for the Google Maps API within the `head` tag. Append your API key into the query string in the script `src` attribute.

3. Next, add the `cordova-2.0.0.js` reference and create another JavaScript tag block to hold our custom code:

```
<head>
  <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

  <title>You are here...</title>

  <script type="text/javascript"
  src="http://maps.googleapis.com/maps/api/js?key=your_api_
key&sensor=true">
    </script>

  <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

  <script type="text/javascript">

  // Custom PhoneGap code goes here

  </script>

</head>
```

> When we included the Google Maps API JavaScript into our document, we set the sensor query parameter to true. If we were only allowing the user to manually input coordinates without automatic detection this could have been set to false. However, we are using the data obtained from the device's sensor to automatically retrieve our location.

4. Let's start creating our custom code within the JavaScript tag block. First, we'll create the event listener to ensure the device is ready and we'll also create the `onDeviceReady` method, which will run using the listener:

```
// Set the event listener to run when the device is ready
document.addEventListener(
        "deviceready", onDeviceReady, false);

// The device is ready, so let's
// obtain the current geolocation data
function onDeviceReady() {
  navigator.geolocation.getCurrentPosition(
    onSuccess,
    onError
  );
}
```

5. Next we can write the `onSuccess` method, which will give us access to the returned location data via the `position` object.

6. Let's take the `latitude` and `longitude` information obtained from the device geolocation sensor response and create a `latLng` object which we will send into the `Map` object when we initialize the component.

7. We will then set the options for our `Map`, setting the center of it to the coordinates we set into the `latLng` variable. Not all of the Google Map controls translate well to the small screen, especially in terms of usability. We can define which controls we would like to use. In this case we'll accept the `zoomControl` but not the `panControl`.

8. To define the `Map` object itself we reference a `div` element and pass through the `mapOptions` variable we have previously declared.

9. To close off this method, let's now create a `Marker` variable to display at the exact location
as set in the `latLng` variable:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {

  var latLng =
      new google.maps.LatLng(
          position.coords.latitude,
            position.coords.longitude);

  var mapOptions = {
        center: latLng,
        panControl: false,
```

```
        zoomControl: true,
        zoom: 16,
        mapTypeId: google.maps.MapTypeId.ROADMAP
    };

var map = new google.maps.Map(
            document.getElementById('map_holder'),
            mapOptions
        );

var marker = new google.maps.Marker({
        position: latLng,
        map: map
    });

}
```

10. To ensure we correctly handle any errors that may occur, let's now include the `onError` function which will display the specific string message according to the error within a `div` element:

```
// Run if we face an error
// obtaining the position data
function onError(error) {

  var errString = '';

  // Check to see if we have received an error code
  if(error.code) {
    // If we have, handle it by case
    switch(error.code)
    {
      case 1: // PERMISSION_DENIED
        errString =
          'Unable to obtain the location information ' +
          'because the device does not have permission '+
          'to the use that service.';
      break;
      case 2: // POSITION_UNAVAILABLE
        errString =
          'Unable to obtain the location information ' +
          'because the device location could not be ' +
          'determined.';
```

```
            break;
            case 3: // TIMEOUT
              errString =
                'Unable to obtain the location within the ' +
                'specified time allocation.';
            break;
            default: // UNKOWN_ERROR
              errString =
                'Unable to obtain the location of the ' +
                'device due to an unknown error.';
            break;
          }

        }

        // Handle any errors we may face
        var element = document.getElementById('map_holder');
        element.innerHTML = errString;
      }
```

11. With the `body` tag, let's include the `div` element into which the map will be displayed:

    ```
    <body>

        <div id="map_holder"></div>

    </body>
    ```

12. Finally, add a `style` block within the `head` tag to supply some essential formatting to the page and the `map` element:

    ```
    <style type="text/css">
      html { height: 100% }
       body { height: 100%; margin: 0; padding: 0 }
       #map_holder { height: 100%; width: 100%; }
    </style>
    ```

13. Running the application on the device, the result will look similar to this:
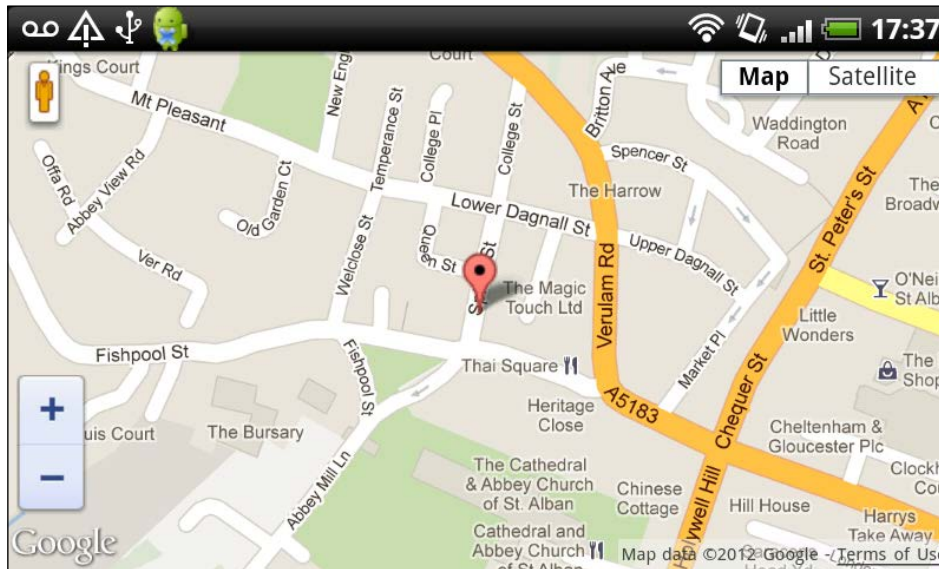
How it works...

Thanks to the use of exposed mapping services such as Google Maps we are able to perform geolocation updates from the device and use the obtained data to create rich, interactive visual mapping applications.

In this example, we centered the `Map` using the device coordinates and also created a `Marker` overlay to place upon the mark for easy visual reference.

The available APIs for mapping services such as this are incredibly detailed and contain many functions and methods to assist you in creating your location-based tools and applications. Some services also set limits on the amount of requests made to the API, so make sure you are aware of any restrictions in place.

There's more...

We used the Google Maps API for JavaScript in this recipe. There are variations on the API level offered by Google, and other mapping systems are also available through other providers such as MapQuest, MultiMap, and Yahoo! Maps. Explore the alternatives and experiment to see if a particular solution suits your application better than the others.

Static maps

In this recipe, we used the dynamic Google Map API. We did this so that we could use the zoom controls and provide our user with a certain level of interaction by being able to drag the map.

As an alternative, you could use the Google Static Map service, which simplifies the code needed to generate a map, and will return a static image showing the location.

You can choose to use an API key with this service, but it is not required. You will still have to enable the API in the same way we enabled the API Access at the start of this recipe.

Consider the following code, which is an amendment to the `onSuccess` method, which runs after the geolocation data has been obtained:

```
// Run after successful transaction
// Let's display the position data
function onSuccess(position) {
  var mapOutput = '<img src="http://maps.googleapis.com/maps/api/
staticmap?center='+position.coords.latitude+','+position.coords.longit
ude+'&zoom=12&size=300x300&scale=2&sensor=true">';
  var element = document.getElementById('map_holder');
  element.innerHTML = mapOutput;
}
```

Here you can see that instead of creating the coordinates, the map and the markers as in the earlier code listing, we simply request an image source using the Static Maps API, and send in the coordinates, image size, and other data as parameters.

By using the Static Map API, you lose the interactivity offered through the dynamic map, but you gain an incredibly simple, easy-to-use service that requires very little code to achieve results.

> You can find out more about the Google Static Map API on the official documentation, available here: `https://developers.google.com/maps/documentation/staticmaps/`.

Creating a visual compass to show the devices direction

The PhoneGap API provides developers with the ability to receive coordinate and heading information from the device. We can use this information to build a custom compass tool that responds to the device movement.

How to do it...

1. Create the HTML layout for our page, including the `cordova-2.0.0.js` reference.

```
<!DOCTYPE html>
<html>
  <head>
     <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Compass</title>
    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>
  </head>
  <body>

  </body>
</html>
```

2. In this example, we will be referencing certain elements within the DOM by class name. For this we will use the **XUI** JavaScript library (`http://xuijs.com/`). Add the `script` reference within the `head` tag of the document to include this library.

3. Let's also create the `script` tag block that will hold our custom JavaScript for interaction with the PhoneGap API, as shown in the following code:

```
<head>
   <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />

    <title>Compass</title>
    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

    <script type="text/javascript"
        src="xui.js"></script>


    <script type="text/javascript">
    // PhoneGap code will go here

    </script>

</head>
```

41

4. Add a new `div` element within the `body` tag and give this the `class` attribute of `container`. This will hold our compass elements for display.

5. The compass itself will be made up of two images. Both images will have an individual `class` name assigned to them, which will allow us to easily reference each of them within the JavaScript. Add these two within the `container` element.

6. Next, write a new `div` element below the images with the `id` attribute set to `heading`. This will hold the text output from the compass response:

    ```
    <body>

      <div class="container">

          <img src="images/rose.png"
            class="rose" width="120" height="121"
            alt="rose" />

          <img src="images/compass.png"
            class="compass" width="200" height="200"
            alt="compass" />

        <div id="heading"></div>

      </div>

    </body>
    ```

7. With the initial layout complete, let's start writing our custom JavaScript code. First, let's define the `deviceready` event listener. As we are using `XUI`, this differs a little from other recipes within this chapter:

    ```
    Var headingDiv;

    x$(document).on("deviceready", function () {

    });
    ```

8. When we have a result to output to the user of the application, we want the data to be inserted into the `div` tag with the `heading id` attribute. `XUI` makes this a simple task, and so we'll update the `headingDiv` global variable to store this reference:

    ```
    x$(document).on("deviceready", function () {
      headingDiv = x$("#heading");

    });
    ```

9. Let's now include the requests to the PhoneGap compass methods. We'll actually call two requests within the same function. First we'll obtain the current heading of the device for instant data, then we'll make a request to watch the device heading, making the request every tenth of a second thanks to the use of the `frequency` parameter. This will provide use with continual updates to ensure the compass is correct:

```
navigator.compass.getCurrentHeading(onSuccess, onError);
navigator.compass.watchHeading(
                onSuccess, onError, {frequency: 100});
```

10. Both of these requests use the same `onSuccess` and `onError` method to handle output and data management. The `onSuccess` method will provide us with the returned data in the form of a `heading` object.

11. We can use this returned data to set the HTML content of the heading element with the generated message string, using the `headingDiv` variable we defined earlier.

12. Our visual compass also needs to respond to the heading information. Using the `CSS` method from `XUI`, we can alter the `transform` properties of the rose image to rotate using the returned `magneticHeading` property. Here we reference the image by calling its individual class name, `.rose`:

```
// Run after successful transaction
// Let's display the compass data
function onSuccess(heading) {
  headingDiv.html(
    'Heading: ' + heading.magneticHeading + '&#xb0; ' +
    convertToText(heading.magneticHeading) + '<br />' +
    'True Heading: ' + heading.trueHeading + '<br />' +
    'Accuracy: ' + heading.headingAccuracy
      );

  // Alter the CSS properties to rotate the rose image
  x$(".rose").css({
    "-webkit-transform":
    "rotate(-" + heading.magneticHeading + "deg)",
     "transform":
    "rotate(-" + heading.magneticHeading + "deg)"
  });

}
```

13. With the `onSuccess` handler in place, we now need to add our `onError` method to output a user-friendly message, should we encounter any problems obtaining the information, as shown in the following code:

```
// Run if we face an error
// obtaining the compass data
function onError() {
  headingDiv.html(
    'There was an error trying to ' +
    'locate your current bearing.'
  );
}
```

14. When creating our message string in the `onSuccess` function we made a call to a new function called `convertToText`. This accepts the `magneticHeading` value from the `heading` object and converts it into a text representation of the direction for display. Let's include this function outside of the `XUI deviceready` block:

```
// Accept the magneticHeading value
// and convert into a text representation
function convertToText(mh) {
  var textDirection;
  if (typeof mh !== "number") {
    textDirection = '';
  } else if (mh >= 337.5 || (mh >= 0 &&  mh <= 22.5)) {
    textDirection =  'N';
  } else if (mh >= 22.5 && mh <= 67.5) {
     textDirection =  'NE';
  } else if (mh >= 67.5 && mh <= 112.5) {
     textDirection =  'E';
  } else if (mh >= 112.5 && mh <= 157.5) {
     textDirection =  'SE';
  } else if (mh >= 157.5 && mh <= 202.5) {
     textDirection =  'S';
  } else if (mh >= 202.5 && mh <= 247.5) {
     textDirection =  'SW';
  } else if (mh >= 247.5 && mh <= 292.5) {
     textDirection =  'W';
  } else if (mh >= 292.5 && mh <= 337.5) {
     textDirection =  'NW';
  } else {
    textDirection =  textDirection;
  }
  return textDirection;
}
```

15. Let's provide some CSS to position our two images on the screen and ensure the rose image is overlaying the compass image. Create a new file called `compass_style. css` and insert the following styles into it:

```css
.container {
  position: relative;
  margin: 0 auto;
  width: 200px;
  overflow: hidden;
}

#heading {
  position: relative;
  font-size: 24px;
  font-weight: 200;
  text-shadow: 0 -1px 0 #eee;
  margin: 20px auto 20px auto;
  color: #111;
  text-align: center;
}
.compass {
  padding-top: 12px;
}
.rose {
  position: absolute;
  top: 53px;
  left: 40px;
  width: 120px;
  height: 121px;
}
```

16. Finally, include the reference to the `compass_style.css` file in the `head` tag of the HTML document:

```html
<title>Compass</title>
<link href="compass_style.css"
    rel="stylesheet" />
<script type="text/javascript"
    src="cordova-2.0.0.js"></script>
<script type="text/javascript"
    src="xui.js"></script>
```

45

17. Running the application on the device, the output will look something like the
     following screenshot:
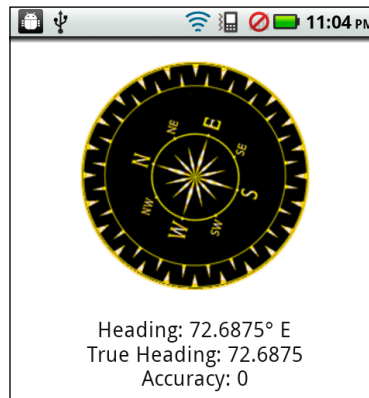
How it works...

The `watchHeading` method from the PhoneGap API compass functionality retrieves periodic
updates containing the current heading of the device at the interval specified as the value
of the `frequency` variable passed through. If no interval is declared, a default value of 100
milliseconds (one-tenth of a second) is used.

With every successful request made on the continuous cycle, the `onSuccess` method is
executed and formats the data for output onto the screen, as well as making a change to the
`transform` property of the graphical element to rotate in accordance with the heading.

The `onSuccess` method returns the obtained heading information in the form of the
`compassHeading` object, which contains the following properties:

▸ `magneticHeading`: A `Number` value ranging from 0 to 359.99 that specifies
   a heading in degrees at a single moment in time.

▸ `trueHeading`: A `Number` value ranging from 0 to 359.99 that specifies the
   heading relative to the geographic North Pole in degrees.

▸ `headingAccuracy`: A `Number` value that indicates any deviation in degrees
   between the reported heading and the true heading values.

▸ `timestamp`: The time in milliseconds at which the heading was determined.

See also

▸ *Chapter 6, Working with XUI*

2

File System, Storage, and Local Databases

In this chapter we will cover:

- ▸ Saving a file to device storage
- ▸ Opening a local file from device storage
- ▸ Displaying the contents of a directory
- ▸ Creating a local SQLite database
- ▸ Uploading a file to a remote server via a POST request
- ▸ Caching content using the web storage local storage API

Introduction

With the ever increasing storage capacities on offer with each mobile device, whether built-in storage or available as an expansion through a card, developers have the ability to interact with and manipulate files stored on the device as well as utilize API functionality to cache content.

This chapter will explore how we can save and open individual files on the device's local filesystem, creating and managing local SQLite databases, uploading a local file to a remote server, and caching content using the local storage API.

Saving a file to device storage

Thanks to the ability to traverse, read, and write to the device filesystem, an application can either write a file to a specific predefined location, or it can be written to a location chosen by the user within the application.

How to do it...

We will allow the user to enter a remote URL for a file into a textbox, and download and save that file to their mobile device. The steps to be performed are as follows:

1. Create the initial HTML layout for our application. We're going to use the `XUI` JavaScript library to easily access DOM elements, so we'll include the reference to the file within the `head` tag along with the `cordova-2.0.0.js` file.

2. Below the Cordova JavaScript reference let's also create a new JavaScript tag block to hold our custom code.

```
<!DOCTYPE html>
<html>
<head>
        <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />
    <title>File Download</title>
    <script type="text/javascript" src="xui.js"></script>
    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>
     <script type="text/javascript">


    </script>
</head>
<body>

</body>
</html>
```

3. Within the `body` tag create two `input` elements. Set the first element `type` attribute to `text` and set the `id` attribute to `file_url`.

4. Set the second `input` element `type` attribute to `button`, the `id` attribute to `download_btn` and the value to equal `Download`.

5. Finally include a new `div` element and set the `id` attribute to `message`. This will be the container into which our returned output is displayed. This is shown in the following code:

```
<body>

        <input type="text"
         id="file_url" value="" />
        <input type="button"
         id="download_btn" value="Download" />
```

48

```
<div id="message"></div>
```

```
</body>
```

6. Within the empty JavaScript tag block we need to define a global variable called `downloadDirectory` that will reference the location on the device to store the retrieved file. We'll also add this variable in our event listener for our application which will run once the native PhoneGap code has been loaded:

```
var downloadDirectory;

document.addEventListener("deviceready", onDeviceReady, true);
```

7. We can now write our `onDeviceReady` function. The first thing we need to do is to access the root filesystem on the device. Here we are requesting access to the persistent file system. Once a reference has been established we run the `onFileSystemSuccess` method to continue.

8. We are then binding a click function to the `download_btn` element using XUI, which will run the download function when clicked:

```
function onDeviceReady() {
window.requestFileSystem(
  LocalFileSystem.PERSISTENT,
    0,
    onFileSystemSuccess,
    null
  );

  x$('#download_btn').on( 'click', function(e) {
  download();
  });
}
```

9. With the connection made to the device storage, we can reference the root system using the `fileSystem` object provided by PhoneGap. Here we then call the `getDirectory` method, providing the name of the directory to gain access to. If it doesn't exist it will be created for us. After a successful response, the returned `DirectoryEntry` object is assigned to the `downloadDirectory` variable we set earlier.

```
function onFileSystemSuccess(fileSystem) {
fileSystem.root.getDirectory('my_downloads',
    {create:true},
    function(dir) {
      downloadDirectory = dir;
    },fail);
}
```

10. Our `download` function will be run when the user clicks the **Download** button. We first need to obtain the URL provided by the user from the text input box. We can pass the value through, to a new custom method called `getFileName`, which will split the string and return the filename and extension for use later in the function. We can now set a user-friendly message into the `message` container to inform them of our progress.

11. Next we instantiate a new `FileTransfer` object from the PhoneGap API to assist us in downloading the remote object. The download method accepts the remote URL to download, the directory on the device to save the file, and the success and error callback functions. After a successful operation, we will inform the user of the local path where the file was saved:

```
function download() {
var fileURL = document.getElementById('file_url').value;
  var localFileName = getFilename(fileURL);

  x$('#message').html('Downloading ' + localFileName);

  var fileTransfer = new FileTransfer();
  fileTransfer.download(
   fileURL,
   downloadDirectory.fullPath + '/' + localFileName,
    function(entry){
       x$('#message').html('Download complete. File saved to: ' +
entry.fullPath);
    },
    function(error){
       alert("Download error source " + JSON.stringify(error));
    }
);
}
```

12. Include the custom function to obtain the filename and extension of the remote file:

```
function getFilename(url) {
if (url) {
   var m = url.toString().match(/.*\/(.+?)\./);
    if (m && m.length > 1) {
    return m[1] + '.' + url.split('.').pop();
    }
  }
return "";
}
```

13. Finally, we supply the `fail` method, which is the generic error handler for all of our functions within the application.

```
function fail(error) {
$('#message').html(
    'We encountered a problem: ' + error.code);
}
```

14. When we run the application, we can specify a remote file to download to the local storage and provide the file's location on the device. The result would look something like the following screenshot:

How it works...

In this recipe we allowed a user to download an external file, publicly accessible on the Internet, and save it to a specified location on the device. First, we needed to create a reference to the `fileSystem` object on the device.

The `fileSystem` object returns the following properties:

 - **name**: A `DOMString` object that represents the name of the file system
 - **root**: A `DirectoryEntry` object that represents the root directory of the file system

Once obtained, we could then obtain the reference to the desired directory location into which our file would be saved using the `DirectoryEntry` object, which returns the following properties:

 - **isFile**: A `boolean` value that is always false as this is a directory
 - **isDirectory**: A `boolean` value that is always true
 - **name**: A `DOMString` object representing the name of the directory
 - **fullPath**: A `DOMString` object that represents the full absolute path of the directory from the root

The `DirectoryEntry` object contains a number of methods that allow you to interact with and manipulate the file system. For more information on the available methods, check out the official Cordova documentation, available here:

```
http://docs.phonegap.com/en/2.0.0/cordova_file_file.
md.html#DirectoryEntry
```

To download the file, we made use of PhoneGap's `fileTransfer` object, and called the object's `download` method to retrieve the remote file, saving it to the correct directory.

There's more...

For any Android application that needs to access or write to the device's local storage or filesystem, you would have to provide permission for the application to do so within the Android manifest file.

iOS applications will also need to have the relevant permissions added to the `Cordova.plist` file to allow access to interact with the device file system.

Domain whitelist

One issue you may encounter when running this example project is an error when trying to download the remote file. Access to remote sites and assets is heavily restricted, thanks to the security model in the Cordova project, whereby the default policy is set to block all remote network access.

This can be easily amended by the developer to allow access to specific domains, subdomains, or by setting a wildcard to allow access to every domain, granting full remote network access, by amending the whitelist access specifications.

To find out more about domain whitelists, check out the official documentation, available at:

```
http://docs.phonegap.com/en/2.0.0/guide_whitelist_index.
md.html#Domain%20Whitelist%20Guide
```

See also

▶   The *Opening a local file from device storage* recipe

Opening a local file from device storage

When developing your mobile application, you may need or want to read particular files from the storage system or from another location on the device.

How to do it...

In this recipe we will build an application that will create a text file on the phone's storage file system, write content into the file, and then open the file to display the content, as listed in the following steps.

1. Create the initial HTML layout for our application. We're going to use the `XUI` JavaScript library to easily access DOM elements, so we'll include the reference to the file within the `head` tag along with the `cordova-2.0.0.js` file.

2. Below the Cordova JavaScript reference let's also create a new JavaScript tag block to hold our custom code. This is shown in the following code:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <title>Open File</title>
    <script type="text/javascript" src="xui.js"></script>
    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>
    <script type="text/javascript">


    </script>
</head>
<body>

</body>
</html>
```

3. Within the body tag create two `input` elements. Set the first element `type` attribute to `text` and set the `id` attribute to `my_text`.

4. Set the second `input` element `type` attribute to `button`, the `id` attribute to `savefile_btn`, and the value to equal `Save`.

5. Finally include two new `div` elements. Set the first element `id` attribute to `message`. This will be the container into which our returned output is displayed. Set the second element `id` attribute to `contents`. This will display the contents of the file:

```
<body>

        <input type="text"  id="my_text" />
        <input type="button" id="saveFile_btn"
        value="Save" />
```

```
            <div id="message"></div>
            <div id="contents"></div>


        </body>
```

6. Within the empty JavaScript tag block we need to define a global variable called `fileObject` that will reference the file on the device. We'll also add in our event listener for our application that will run once the native PhoneGap code has been loaded. The `onDeviceReady` method requests access to the persistent filesystem root on the device. Once obtained, it will execute the `onFileSystemSuccess` method, shown as follows:

```
var fileObject;
document.addEventListener("deviceready", onDeviceReady, true);

function onDeviceReady() {
window.requestFileSystem(
    LocalFileSystem.PERSISTENT, 0,
    onFileSystemSuccess, fail);
}
```

> The `LocalFileSystem.PERSISTENT` constant is used here to ensure we access storage that cannot be removed by the user agent without permission from the application or the user. We could use the `LocalFileSystem.TEMPORARY` constant to access storage that has no guarantee of persistence.

7. With the connection made to the device storage, we can reference the root system using the `fileSystem` object provided by PhoneGap. Here we then call the `getFile` method, providing the name of the file we wish to open. If it doesn't exist it will be created for us.

```
function onFileSystemSuccess(fileSystem) {
fileSystem.root.getFile("readme.txt",
    {create: true, exclusive: false},
    gotFileEntry, fail);
}
```

8. After a successful response, the returned `FileEntry` object is assigned to the `fileObject` variable we created earlier. At this point we can also bind a click handler to our save button, which will run the `saveFileContent` function when clicked.

```
function gotFileEntry(fileEntry) {
fileObject = fileEntry;
```

```
  x$('#saveFile_btn').on('click', function() {
    saveFileContent();
});
}
```

9. As we're dealing with a local file that, as of yet, doesn't have any content, we can use methods within the PhoneGap API to write to the file. The `saveFileContent` method will access the `fileObject` and call the `createWriter` method to start this process.

```
function saveFileContent() {
fileObject.createWriter(gotFileWriter, fail);
}
```

10. Let's now create the `gotFileWriter` method called as a callback from the save function. We'll send the value from the `my_text` input field into the `writer.write()` method to populate the file content. After the writing has finished, we'll output a status message into the `message` `div` element and then instantiate the `FileReader` object to read the file contents.

11. We will then pass the `fileObject` into the `reader.readAsText()` method to return the text content of the file. After the read has completed, we will output the contents into the `div` element for display.

```
function gotFileWriter(writer) {
var myText = document.getElementById('my_text').value;
    writer.write(myText);

writer.onwriteend = function(evt) {
    x$('#message').html('<p>File contents have been written.<br
/><strong>File path:</strong> ' + fileObject.fullPath + '</p>');

    var reader = new FileReader();
  reader.readAsText(fileObject);
    reader.onload = function(evt) {
    x$('#contents').html('<strong>File contents:</strong> <br />'
+ evt.target.result);
    };

};
}
```

12. Finally, include the `fail` error handler method to catch any problems or errors:

```
function fail(error) {
alert(error.code);
}
```

13. When we run the application on our device, we should see output similar to the following screenshot:
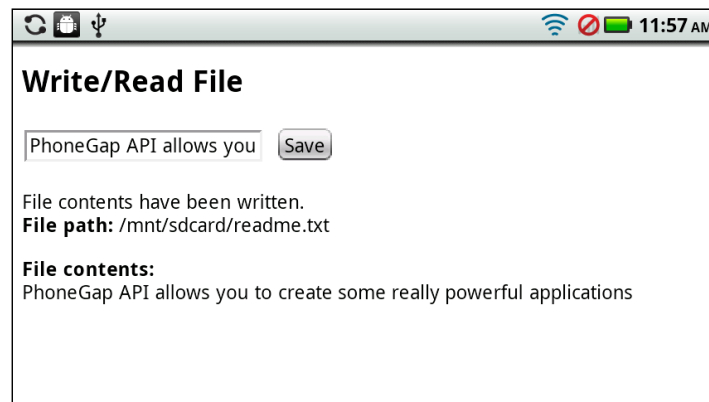
How it works...

To gain access to the filesystem on the device, we first request access to the persistent storage, which then provides us with access to the `fileSystem` object.

Navigating to the root of the filesystem, we then call the `getFile` method, which will look up through the requested file, or create it if it doesn't already exist in the specified location.

Once a user has typed content in to the input textbox, we can then instantiate a `FileWriter` object on the saved object containing the file reference and write the user-supplied content to the file. We can also make use of the `FileWriter` object's `onwriteend` method, called when the request has completed, to output a message to the user and then begin the request to read the contents of the file, achieved through the use of the `FileReader` object.
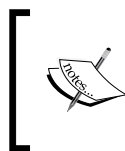
> For a comprehensive look at the File functions available within the PhoneGap API, please refer to the official documentation at `http://docs.phonegap.com/en/2.0.0/cordova_file_file.md.html#File`.

There's more...

In this example we were able to read the file as text, using the `readAsText` method. We are also able to read a file and return the contents as a base64 encoded data URL using the `readAsDataURL` method. While there are no limitations on what type of file can be read, depending on the choice of reading method as well as the size of the file, consideration must be placed on the speed and performance impact that may occur when trying to read large files which may take up quite a lot of processing power.

See also

▶ The *Saving a file to device storage* recipe

Displaying the contents of a directory

As devices may offer us a lot of storage space that we can potentially use, we can also make sure we have the ability to traverse the filesystem to ascertain the structure of the storage available.

How to do it...

In this recipe we will build an application that will read the contents of a directory from the device's root filesystem and display them in a list format. The following steps will help you to do the same:

1. Create the initial HTML layout for our application. For this recipe we will also be using the *jQuery Mobile* framework, so let's include the required JavaScript and CSS references, as well as the reference to the `cordova-2.0.0.js` file:

```
<!DOCTYPE html>
<html>
<head>
    <title>Directory Reader</title>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />

    <link rel="stylesheet"
            href="jquery/jquery.mobile-1.1.1.min.css" />
    <script src="jquery/jquery-1.8.0.min.js"></script>
    <script src="jquery/jquery.mobile-1.1.1.min.js"></script>
    <script type="text/javascript"
          src="cordova-2.0.0.js"></script>
      <script type="text/javascript">


    </script>
</head>
<body>


</body>
</html>
```

2. The jQuery Mobile framework will handle the formatting and layout of the body content. Include a `div` element with the `data-role` attribute set to `page`. Within this add a header to contain our application title. We'll then add a new `div` element with the `data-role` attribute set to `content`, inside of which we'll place a `ul` tag block to hold our directory listings. Set the `data-role` for the `ul` tag to `listview` and give it an `id` of `directoryList` so that we can reference it later:

```
<body>

<div data-role="page">

<div data-role="header">
    <h2>Directory Reader</h2>
  </div>

  <div data-role="content">
    <ul id="directoryList" data-role="listview"
      data-inset="true">

    </ul>
</div>

</div>

</body>
```

3. Next we need to add in the event listener and the `onDeviceReady` method to run once the native PhoneGap code is ready to be executed. In this method we will request access to the file root on the persistent storage, which will then run the `onFileSystemSuccess` callback method.

```
document.addEventListener("deviceready", onDeviceReady, false);

function onDeviceReady(){
window.requestFileSystem(
    LocalFileSystem.PERSISTENT,
    0, onFileSystemSuccess, fail
);
}
```

4. To ensure we have some extra content to list, we'll call the `getDirectory` and `getFile` methods respectively, which will then create the directory and file if they do not already exist. We can access the `DirectoryEntry` object at `fileSystem.root`, and call the `createReader()` method from it to instantiate the `DirectoryReader` object. Lastly, let's call the `readEntries()` method from this object to read the entries within the provided directory.

```
function onFileSystemSuccess(fileSystem) {
// Create some test files
  fileSystem.root.getDirectory("myDirectory",
   { create: true, exclusive: false },
   null,fail);
  fileSystem.root.getFile("readthis.txt",
   { create: true, exclusive: false },
   null,fail);

  var directoryReader = fileSystem.root.createReader();
  // Get a list of all the entries in the directory
  directoryReader.readEntries(success,fail);
}
```

5. The success callback method is passed an array of `FileEntry` and `DirectoryEntry` object. Here we'll loop over the array and create a list item for each returned entry, writing the name and URI path. We'll also check the type of the entry and display it if it's a directory or a file.

6. Each list item is appended to the `directoryList ul` element, and we then call a `listview` refresh method on the element to update the content for display:

```
function success(entries) {
var i;
  var objectType;
  for (i=0; i<entries.length; i++) {
    if(entries[i].isDirectory == true) {
      objectType = 'Directory';
    } else {
      objectType = 'File';
    }
    $('#directoryList').append('<li><h3>' + entries[i].name + '</
h3><p>' + entries[i].toURI() + '</p><p class="ui-li-aside">Type:
<strong>' + objectType + '</strong></p></li>');
  }
  $('#directoryList').listview("refresh");
}
```

7. Finally, let's include the `fail` error handler method to alert us of any issues:

```
function fail(error) {
alert("Failed to list directory contents: " + error.code);
}
```

8. When we run the application on our device, the output will look something like the following screenshot:

How it works...

When traversing through directories, the PhoneGap API provides the perfect solution in the form of the `DirectoryReader` object, which lists all directories and files within the chosen directory. This contains a single method called `readEntries`, and it's the success callback method from this that allows us to loop over the contents and output them as a visual representation.

See also

▶   The *Creating a jQuery Mobile layout* recipe in *Chapter 7*, *User Interface Development with jQuery Mobile*

Creating a local SQLite database

SQLite databases are a fantastic way to store structured information from a web context. SQLite is a self-contained transactional database that does not require any configuration. This is ideal for saving and querying dynamic information within a mobile application.

How to do it...

For this recipe we will create a mobile application that will allow us to store text entries into a local SQLite database and then query the database to retrieve all saved items.

1. Firstly, create the HTML layout for our page, including the reference to the `cordova-2.0.0.js` file:

```html
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <title>My ToDo List</title>
    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>

</head>
<body>
    <h1>My ToDo List</h1>


</body>
</html>
```

2. In this example we will be referencing certain elements within the DOM by class name. For this we will use the `XUI` JavaScript library. Add the `script` reference within the `head` tag of the document to include this library.

3. Below the PhoneGap JavaScript include, "write a new JavaScript tag block", and within this define an `onDeviceReady` event listener to ensure the device is ready and fully loaded before the application proceeds to execute the code.

```html
<title>My ToDo List</title>
<script type="text/javascript"
        src="cordova-2.0.0.js"></script>

<script type="text/javascript"
```

```
        src="xui.js"></script>
<script type="text/javascript">
// PhoneGap code here

</script>
```

4. Add an input element within the body tags with the `id` attribute set to `list_action`. This will allow the user to add entries into the database list.

5. Below the input element, add a button element with the `id` attribute set to `saveItem`.

6. Let's also add two `div` elements to hold any generated data. The first, with the `id` attribute set to `message`, will hold any database connection error messages if we have issues trying to connect. The second, with the `id` attribute set to `listItems`, will act as a container into which our generated list will be placed for display.

```
<body>
<h1>My ToDo List</h1>


  <input type="text"       id="list_action" />

  <input type="button" id="saveItem" value="Save" />


  <div id="message"></div>

  <div id="listItems"></div>




</body>
```

7. With the layout complete, let's move on to adding our custom JavaScript code. To begin with, we need to define the `deviceready` event listener. As we are using the `XUI` library, for this recipe we will write this function as follows:

```
x$(document).on("deviceready", function () {


});
```

8. As we want to set the inner HTML values for the `list` and `message` `div` containers, let's define the references to those particular elements. `XUI` makes this really easy. We'll also create a global variable called `db` that will eventually hold our database connection.

9. Let's also bind a click handler to the `saveItem` button element. When pressed, it will run the `insertItem` method to add a new record to the database.

10. We now need to create a reference to our SQLite database. The PhoneGap API includes a function called `openDatabase` that creates a new database instance or opens the database if it already exists. The returned object will allow us to perform transactions against the database.

```
var listElement = x$('#listItems');
var messageElement = x$('#message');
var db;
x$('#saveItem').on('click', function(e) {
insertItem();
});

// Create a reference to the database
function getDatabase() {
return window.openDatabase("todoListDB",
            "1.0", "ToDoList Database", 200000);
}
```

11. We can now include the call to and create the `onDeviceReady` method. Here we assign the database instance to the variable `db`, which will allow us to perform a transaction into the database. In this case, we'll execute a simple SQL script to create a table called `MYLIST` if it doesn't already exist.

```
// Run the onDeviceReady method
onDeviceReady();

// PhoneGap is ready
function onDeviceReady() {
db = getDatabase();
db.transaction(function(tx) {
tx.executeSql('CREATE TABLE IF NOT EXISTS MYLIST
    (id INTEGER PRIMARY KEY AUTOINCREMENT, list_action)');
}, databaseError, getItems);
}
```

12. Let's now define the `getItems` method, which is run on a successful callback from the database transaction in the previous method. Once more we reference the database object and perform another transaction, this time to select all items from the table.

```
// Run a select statement to pull out all records
function getItems() {
db.transaction(function(tx) {
tx.executeSql('SELECT * FROM MYLIST', [],
```

```
    querySuccess, databaseError);
}, databaseError);
}
```

13. Having received the results from the select query, we can loop over the results and create list item elements that we can then set within the list `div` container. Here we can reference the `id` and `list_action` columns from the results, drawn from the SQLite database table we created earlier. We'll also output a user-friendly message displaying the total number of records stored.

```
// Process the SQLResultSetList
function querySuccess(tx, results) {
var len = results.rows.length;
var output = '';
for (var i=0; i<len; i++){
    output = output +
        '<li id="' + results.rows.item(i).id + '">' +
        results.rows.item(i).list_action + '</li>';
}
messageElement.html('<p>There are ' + len +
    ' items in your list:</p>');
listElement.html('<ul>' + output + '</ul>');
}
```

14. We initially bound a click event handler to our `saveItem` button. Let's now create the `insertItem` method that the click handler would invoke. We want to take the value of the `list_action` text input box and pass that into the database transaction when we run an insert query. A successful insert will call our `getItems` method to query the database and populate the list with all updated information from our database.

```
// Insert a record into the database
function insertItem() {
var insertValue =
document.getElementById('list_action').value;
db.transaction(function(tx) {
tx.executeSql('INSERT INTO MYLIST
    (list_action) VALUES ("' + insertValue + '")');
}, databaseError, getItems);
// Clear the value from the input box
document.getElementById('list_action').value = '';
}
```

15. Finally, let's include our `databaseError` fault handler method to display any issues we may encounter from the database and display them in the `message div` element.

```
// Database error handler
function databaseError(error) {
```

```
messageElement.html("SQL Error: " + error.code);
}
```

16. When we run the application on the device, your output should look something like the following screenshot:
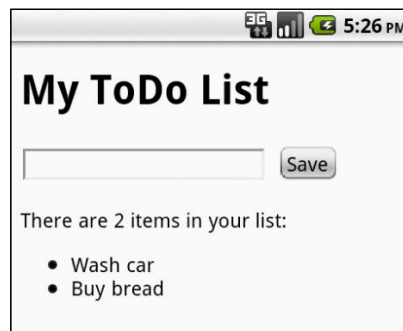
How it works...

In order to access the SQLite database and to perform any transactions, we first need to establish a connection to the `.db` file on the device using the `openDatabase` method. Once this connection is established, we can use the `SQLTransaction` object to perform `executeSql` methods such as table creation, selection and insertion queries, written using standard SQL syntax.
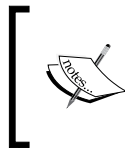
> For more details on the full methods available to use with the SQLite implementation, check out the official Cordova storage documentation at: `http://docs.phonegap.com/en/2.0.0/cordova_ storage_storage.md.html#Storage`.

There's more...

The Storage API accessible through Cordova is based on the *W3C Web SQL Database Specification*. Some devices already have an implementation of this specification. If any device your application runs on already provides this functionality, it will use its built-in support for the Storage specification and not use Cordova's implementation.

See also

► *Chapter 6, Working with XUI*

Uploading a file to a remote server

Sometimes working with only the local system for mobile applications is not enough. There are use cases for having the requirement to interact with a remote server; to share a file, for example.

How to do it...

In this recipe we will build an application that allows the user to take a photo and upload it to a remote server:

1. First let's create the initial HTML layout for our application. We will be using the `XUI` JavaScript library to assist us in easily referencing DOM elements. Include the `xui.js` and `cordova-2.0.0.js` references within the `head` tags and create an empty JavaScript tag block into which we will place our custom code.

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
  <title>Upload File</title>
   <script type="text/javascript"
         src="xui.js"></script>
   <script type="text/javascript"
         src="cordova-2.0.0.js"></script>
   <script type="text/javascript">


   </script>
</head>
<body>


</body>
</html>
```

2. We now need to create some elements within the body tag. Let's include a `button` element with the `id` attribute set to `selectorBtn`. Create a `div` element with the `id` attribute set to `message`, which we'll use to display status updates from the transfer, and finally we'll create an `img` tag with the `id` attribute set to `returnImage`.

```
<body>

<button id="selectorBtn">Take Photo</button>

<div id="message"></div>

<img id="returnImage" />

</body>
```

3. We can now include our event handler to ensure that the device is ready before proceeding, as well as creating our `onDeviceReady` method. In this method we will bind the `selectorBtn` element to a `touchstart` event. When pressed, this will call the `camera.getPicture` method from the PhoneGap API, which will open the device's default camera application to let our user take a photo to upload:

```
document.addEventListener("deviceready", onDeviceReady, true);

function onDeviceReady() {

x$("#selectorBtn").touchstart(function(e) {
  navigator.camera.getPicture(
    gotPicture,
      onError,
      {
       sourceType:Camera.PictureSourceType.CAMERA,
        destinationType:Camera.DestinationType.FILE_URI,
        quality:50
      }
    );
  });
}
```

> In this instance we have the `sourceType` property set to `Camera.PictureSourceType.CAMERA`. We could change this to be `Camera.PictureSourceType.PHOTOLIBRARY` or `Camera.PictureSourceType.SAVEDPHOTOALBUM` if we wanted the user to select an image to upload from their saved photos.

4. We now include the `gotPicture` method as the success callback having obtained an image. We have the location of the file as a provided parameter. Let's update the `message div` to display a friendly message to users, and then we'll create a new instance of the `FileUploadOptions` object, which we'll use to specify additional parameters for the upload script. The `options.fileKey` value sets the name for the form field that will contain the uploaded file.

5. We can now create a new instance of the `FileTransfer` object, from which we'll call the `upload` method. Here we can pass in the file's location on the device, the remote address to upload it to, and also send any additional parameters we may have included in the `options` object.

```
function gotPicture(fileLocation) {
x$("#message").html("<p>Uploading your image...</p>");

  var options = new FileUploadOptions();
  options.fileKey = "file";
  options.fileName =
   fileLocation.substr(fileLocation.lastIndexOf('/')+1);
  options.mimeType = "image/jpeg";
  options.chunkedMode = false;

  var fileTransfer = new FileTransfer();
  fileTransfer.upload(
    fileLocation,
    "http://address_to_remote_server_page/upload.cfm",
    fileUploaded,
    onError,
    options
  );
}
```
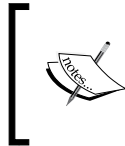
> In this example, I have used ColdFusion as the dynamic server-side language to process the upload. You can, of course, use any server-side language that you have access to or feel comfortable using to manage the upload.

6. Let's now create the callback handler following a successful upload. The `response` parameter is a `FileUploadResult` object, and from this we can obtain the total number of bytes sent, as well as the output response from the server. In this case, we are returning the image from the server-side script and we will set it as the `src` attribute for the `returnImage img` element:

```
function fileUploaded(result) {
x$("#message").html('<p>Upload complete!!<br />Bytes sent: '
    + result.bytesSent + '</p>');
  x$("#returnImage").attr("src", result.response);
}
```

7. Finally, let's create our `onError` fault handler to alert us of any possible issues:

```
function onError(error) {
alert("Error: " + JSON.stringify(error));
}
```

8. When we run the application on a device, the output should look something like the following screenshot:
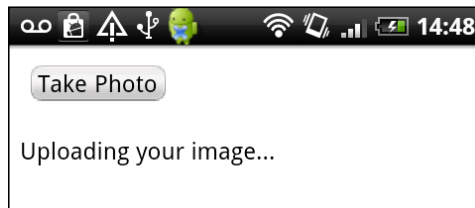
9. Following a successful response from the remote server, the application would then display something similar to the following screenshot:
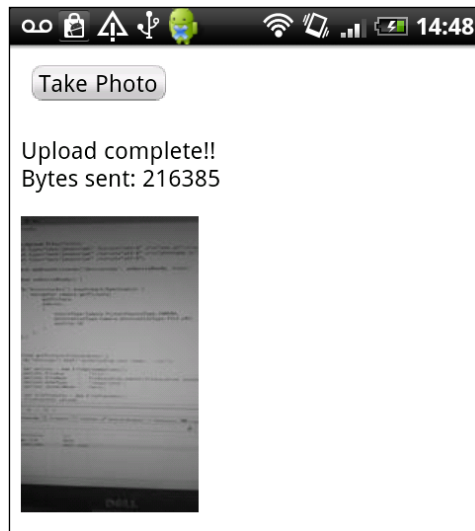
How it works...

Firstly, we need to define how we are going to retrieve our image from the device using the `camera.getPicture` method. Once we have obtained our picture, we can start to build our `FileTransfer` object, which will handle the transaction to the remote server for us. We can then create a `FileUploadOptions` object, which specifies any additional parameters to use on the server-side handling page. The properties available to use within the `FileUploadOptions` object are as follows:

▸ `fileKey`: A `DOMString` object that represents the name of the form element. The default value is `file`.

▸ `fileName`: A `DOMString` object that represents the name you wish the file to be saved as on the server. The default value is `image.jpg`.

- ► `mimeType`: A `DOMString` object that represents the mime type of the data you wish to upload. The default value is `image/jpeg`.

- ► `params`: An object that allows you to set optional key/value pairs to also be included in the HTTP request.

- ► `chunkedMode`: A `boolean` value that determines whether or not the data should be uploaded in a chunked streaming mode. The default value is `true`.

Finally, our success callback method will contain the `FileUploadResult` object returned from the transaction, which gives us access to the response from the server as well as the number of bytes sent in the upload, which we can then output, store, or use in any way we need to.
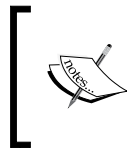
> For more details on the `FileTransfer` object, please check out the official Cordova documentation:
> `http://docs.phonegap.com/en/2.0.0/cordova_file_file.md.html#FileTransfer`.

See also

- ► The *Displaying network connection status* recipe in *Chapter 5*, *Hook into Native Events*

Caching content using the web storage local storage API

As mobile users access applications and pull remote data on the move, we need to be conscious and aware that our application may be using up limited data services. We can implement services and techniques to help reduce unnecessary remote calls to data.

How to do it...

In this recipe, we will build an application that allows the user to search Twitter using its open API. We'll store the results for the search in the `localStorage` so it's available when we re-open the application:

1. Let's start off by creating the initial format for our `index.html` page. We will be using the jQuery Mobile framework for our layout, so include the relevant CSS and JavaScript file references within the `head` tag.

2. Below these, include the JavaScript reference to the `cordova-2.0.0.js` file.

3. Within the `body` tag, create a new `div` element with the `data-role` attribute set to `page`, which will form the container for the jQuery Mobile layout:

   ```
   <!DOCTYPE html>
   ```

```
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />
    <title>Local Storage</title>
  <link rel="stylesheet"
        href="jquery/jquery.mobile-1.1.1.min.css" />
  <script src="jquery/jquery-1.8.0.min.js"></script>
  <script src="jquery/jquery.mobile-1.1.1.min.js"></script>
  <script type="text/javascript" src="cordova-2.0.0.js"></script>

</head>
<body>

  <div data-role="page">


  </div>

</body>
</html>
```

4. Let's place some more layout structures into our application, which are required by the jQuery Mobile framework. Create a new `div` element with the `id` attribute set to `header`. The `data-role` and `data-position` attributes must also be set as shown.

5. Within the header we'll display an `anchor` tag to exit the application. We are specifying the `id` attribute and are also setting a specific icon to display in the header, thanks to the `data-icon` attribute.

6. We'll also include an `h2` heading to add a title to the application, as well as a second button which we'll use to clear any cached content. This too has a specific icon set in the `data-icon` attribute, as well as the `id` attribute to allow us to reference it via our JavaScript:

```
<div id="header" data-role="header" data-position="inline">

<a id="exit_btn" data-inline="true"
   data-theme="b" data-icon="home">Exit</a>

<h2>Local Storage Search</h2>

<a id="clear_btn" data-inline="true"
   data-theme="b" data-icon="delete">Clear Storage</a>
</div>
```

7. Below this, create a new `div` element with the `data-role` attribute set to `content`. This will house a second `div` block, inside of which we'll place two `input` elements; the first to hold the user's search criteria, the second is the button to perform the search.

8. We'll now include a `ul` element with the `id` attribute set to `tweetResults`, and the `data-role` attribute set to `listview`, which will hold our returned data:

```
<div data-role="content">

<div data-role="fieldcontain">

<input type="search" name="search"
   id="searchTerm" data-inline="true" data-icon="search" />
<input type="button" id="search_btn"
   value="Search" data-theme="b" data-inline="true" />

</div>

<ul id="tweetResults" data-role="listview"
    data-inset="true">

</ul>

</div>
```

9. With the layout complete, we can start adding our custom code. Include a new JavaScript tag block before the closing `head` tag. Inside of this, let's write the event listener to ensure the PhoneGap native code has loaded before we proceed.

```
<script type="text/javascript">

document.addEventListener("deviceready", onDeviceReady, true);


</script>
```

10. Let's start adding the custom code that will be run when the device is ready. Create the `onDeviceReady` function. At the start we'll set some required variables; the first two will hold some messages to output to our users, while the third is a reference to the `localStorage` interface which we'll use to save our data in key/value pairs.

11. We need to run a check to see if any content from a previous request has been stored by calling the `getItem` method on the `localStorage` object. If it exists, we'll display a user-friendly message, set the search term from the previous search into the input box, display the clear button by calling the `showClearButton` method, then finally loop over the results to display them, by calling the `outputResults` method.

12. If we have no previous results saved in the `localStorage` object, we'll simply display a welcome message to the user and ensure the clear button is hidden.

```
function onDeviceReady() {
    // Create the friendly messages and define the variables
    var previousMessage = 'Here are your previous search
results..';
    var welcomeMessage = 'What would you like to search for?';
    var localStorage = window.localStorage;

    /* Firstly, check to see if localStorage
       has any cached content from a previous request. */
    if(localStorage.getItem('twitSearchResults')) {

      /* We have saved content,
         so display a nice message to the user */
      $('body h2').html(previousMessage);

      /* Set the value of the stored search
         term into the input box */
      $('#searchTerm').val(localStorage.getItem('searchTerm'));

      // Display the clear button
      showClearButton();

      // Send the stored data to be rendered as HTML
      outputResults(JSON.parse(localStorage.
getItem('twitSearchResults')));
    } else {

      /* There is nothing cached,
         so display a friendly message */
      $('body h2').html(welcomeMessage);
      hideClearButton();

    }

    // add click handlers here

  }
```

13. Within the `onDeviceReady` function, we now need to set up the click handlers for each of our buttons. The first is the `clear_btn` element, which when clicked will clear the values in the `tweetResults div` element, and remove and data we have cached by calling the clear method on the `localStorage` object.

73

14. The second handler is applied to the `exit_btn` element, which will gracefully close the application.

```
/* Add a click handler to the clear button
   which will be displayed is a user returns
      to the page with saved content */
$('#clear_btn').click(function() {
  // Clear the entire local storage object
  localStorage.clear();
  // Clear the content list
  $('#tweetResults').html('');
  $('#tweetResults').hide();

  /* There is nothing cached,
        so display a friendly message */
  $('body h2').html(welcomeMessage);

  // Remove the clear button
  hideClearButton();

  // Reset the search term input field
  $('#searchTerm').val('');

});

$('#exit_btn').click(function() {
  navigator.app.exitApp();
});
```

15. The third click handler will be placed on the `search_btn` element, which will take the search term provided by the user and pass it to a new function called `makeSearchRequest`:

```
/* Add a click handler to the search button
   which will make our AJAX requests for us */
  $('#search_btn').click(function() {

  /* Obtain the value of the search term and send it
     through to the request function */
  makeSearchRequest($('#searchTerm').val());

});
```

16. Before we make the call, we'll save the search term into the `localStorage` object by using the `setItem()` method. This will allow us to reference it at any time until we have cleared the cache.

74

17. To make the request to the remote API, we'll utilize jQuery's built-in `ajax()` method, here asking for five results per page on the search term provided by the user. To handle the returned data we'll also specify the `jsonpCallback` method, which in this case is a new function called `storeResults`.

```
function makeSearchRequest(searchTerm) {

// Display a user-friendly message
  $('body h2').html('Searching for: '+ searchTerm);

  /* Store the value we are searching
   for into the localStorage object */
  localStorage.setItem('searchTerm', searchTerm);

  // Make the request to the Twitter search API
  $.ajax({
    url:
   "http://search.twitter.com/search.json?q="+
      searchTerm+"&rpp=5",
    dataType: "jsonp",
    jsonpCallback: "storeResults"
  });

}
```

18. Once we have obtained a response from the request, we'll check to make sure we have access to the `localStorage` functionality, and if we do we'll save the entire response, converting the JSON data into a string before we output the results to the user:

```
function storeResults(data) {
  /* Save the latest search results,
     coercing the data from an object into a string */
  localStorage.setItem(
     'twitSearchResults',
     JSON.stringify(data));
  outputResults(data);
}
```

19. We have the raw JSON data with which to create our output. Here, we loop through the results to create the required HTML blocks for display. As Twitter information contains a lot of links to users, dates, and other links, we'll also ensure we have those converted for our user.

20. Within the loop we'll append each processed result and append it to the `tweetResults ul` tag block as an individual list item element.

21. Once the processing is complete, we then need to refresh the list to reload the contents ready for display.

```
function outputResults(data) {

    // Clear the content and hide the results element
    $('#tweetResults').html('');

    // Loop through the results in the JSON object
    $.each(data.results,
    function(i, tweet) {
    /* Replace and define any URLs
         for inclusion in the output */
        tweet.text = tweet.text.replace(/((https?|s?ftp|ssh)\:\/\/
[^"\s\<\>]*[^.,;'">\:\s\<\>\)\]\!])/g,
        function(url) {
            return '<a href="'+url+'">'+url+'</a>';
          }).replace(/\B@([_a-z0-9]+)/ig,
        function(reply) {
            return  reply.charAt(0)+'<a href="http://twitter.
com/'+reply.substring(1)+'">'+reply.substring(1)+'</a>';
        });

     $('#tweetResults').append('<li><img src="' + tweet.profile_
image_url + '" /><h3>@' + tweet.from_user_name + '</h3><p>'+
tweet.text + '</p><p class="ui-li-aside"></p></li>');
    });
    // Refresh the list view
    $('#tweetResults').listview("refresh");
}
```

22. Let's now create the functions to handle the visual display of our button to clear the `localStorage` cache. We can reference the button `id` attribute and then apply the jQuery `css()` method to alter its style.

```
function showClearButton() {
$("#clear_btn").css('display', 'block');
}

function hideClearButton() {
$("#clear_btn").css('display', 'none');
}
```

> Here we are directly changing the styles of the elements. You could amend this code to add and remove a CSS class to handle the display instead.

23. Finally, let's include our `onError` function which will be fired if we encounter any issues along the way.

```
function onError(error) {
alert("Error: " + JSON.stringify(error));
}
```

24. For any users running the application for the first time or with an empty cache, the application will look something like the following screenshot:
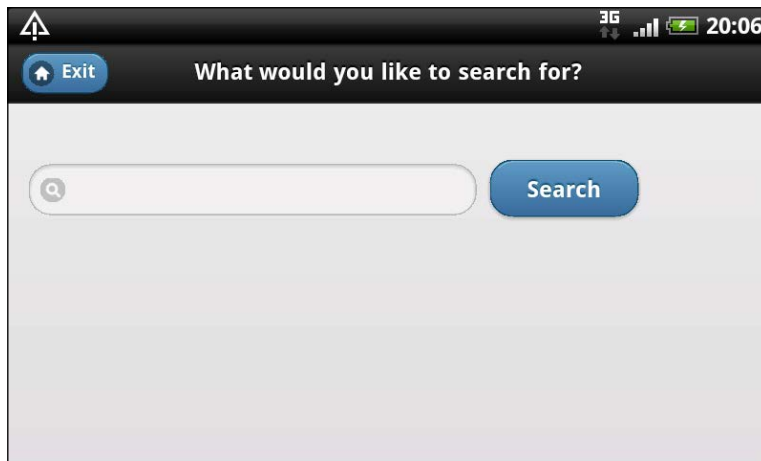
25. Once a request has been made, every time the user opens the application they will be presented with the details of their previous search request, until they clear the cache or make a new request, as shown in the following screenshot:
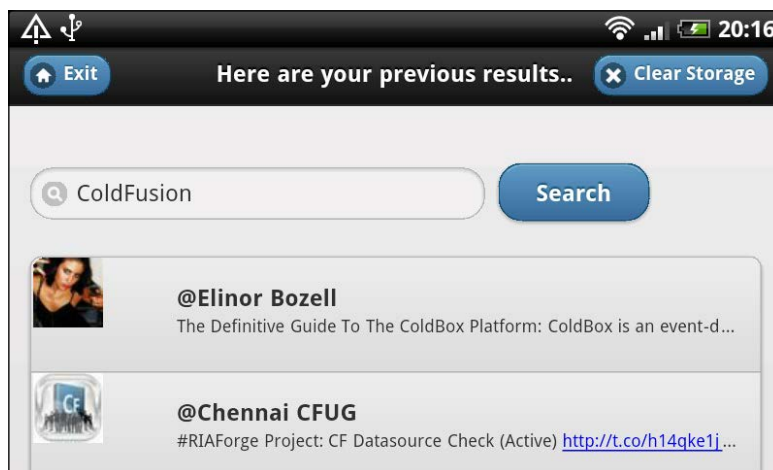
How it works...

To cache and store our data from the request we simply saved the values, assigning them to a key we could reference, using the `setItem` method into the `localStorage` object. We were then able to reference the storage to see if that particular key existed by calling the `getItem` method. If nothing was present, we were able to make a new remote call, and then save the data that was returned.

There's more...

The `localStorage` options provided with the Cordova API do a fantastic job of persisting and allowing us to easily access and retrieve saved data.

For those of you wishing to explore alternative storage options, check out **Lawnchair**, an open source project written by Brian Leroux. Built with mobile applications in mind, Lawnchair is a lightweight JavaScript file that is extensible and can be used with a number of adaptors to persist data, using key/value pairs, and it has an incredibly simple API.

> Find out more about Lawnchair here: `http://brian.io/lawnchair/`

3
Working with Audio, Images, and Video

In this chapter, we will cover:

- ▶ Capturing audio using the device audio recording application
- ▶ Recording audio within your application
- ▶ Playing audio files from the local filesystem or over HTTP
- ▶ Capturing video using the device video recording application
- ▶ Loading a photograph from the device camera roll/library
- ▶ Applying an effect to an image using canvas

Introduction

This chapter will include a number of recipes that outline the functionality required to capture audio, video, and camera data, as well as the playback of audio files from the local system and remote host. We will also have a look at how to use the HTML5 canvas element to edit an image on the fly.

Capturing audio using the devices audio recording application

The PhoneGap API allows developers the ability to interact with the audio recording application on the device and save the recorded audio file for later use.

How to do it...

We'll make use of the `Capture` object and the `captureAudio` method it contains to invoke the native device audio recording application to record our audio. The following steps will help you to do so:

1. Create the initial layout for the application, and include the JavaScript references to jQuery and Cordova. We'll also set a stylesheet reference pointing to `style.css`.

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <title>Audio Capture</title>
    <link rel="stylesheet" href="style.css" />
    <script src="jquery/jquery-1.8.0.min.js"></script>
    <script type="text/javascript"
        src="cordova-2.0.0.js"></script>
</head>
<body>


</body>
</html>
```

2. Create a new `button` element within the `body` tags of the document, and set the `id` attribute to `record`. We'll use this to bind a touch handler to it:

```
<button id="record">capture audio</button>
```

3. Create a new file called `style.css` and include some CSS to format the `button` element:

```
#record {
  display: block;
  padding: .4em .8em;
  text-decoration: none;
  text-shadow: 1px 1px 1px rgba(0,0,0,.3);
  -webkit-transition:.3s -webkit-box-shadow, .3s padding;
  transition:.3s box-shadow, .3s padding;
  border-radius: 200px;
  background: rgba(255,0,0,.6);
  width: 10em;
  height: 10em;
  color: white;
```

```
    position: absolute;
    top: 25%;
    left: 25%;
}
```

4. With the user interface added to the page and the styles applied, the application looks something like the following screenshot:
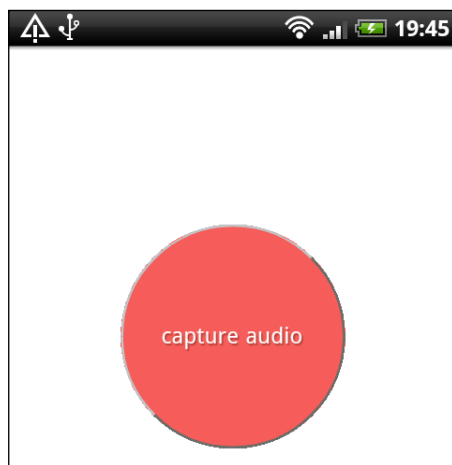
5. Now let's start adding our custom code. Create a new `script` tag block before the closing `head` tag. Within this we'll set up an event listener, which will call the `onDeviceReady` method once the PhoneGap code is ready to run.

6. We'll also create a global variable called `audioCapture`, which will hold our `capture` object:

```
<script type="text/javascript">

    document.addEventListener("deviceready",
     onDeviceReady, true);

    var audioCapture = '';

</script>
```

7. We now need to create the `onDeviceReady` method. This will firstly assign the `capture` object to the variable we defined earlier. We'll also bind a `touchstart` event to the `button` element, which when pressed will run the `getAudio` method to commence the capture process:

```
function onDeviceReady() {
    audioCapture = navigator.device.capture;
```

81

```
    $('#record').bind('touchstart', function() {
        getAudio();
    });
}
```

8. To begin the audio capture, we need to call the `captureAudio()` method from the global `capture` object. This function accepts three parameters. The first is the name of the method to run after a successful transaction. The second is the name of the error handler method to run if we encounter any problems trying to obtain audio. The third is an array of configuration options for the capture request.

9. In this example we are forcing the application to retrieve only one audio capture, which is also the default value:

```
function getAudio() {
audioCapture.captureAudio(
    onSuccess,
    onError,
    {limit: 1}
);
}
```

10. Following on from a successful transaction, we will receive an array of objects containing details for each audio file that was captured. We'll loop over that array and generate a string containing all of the properties for each file, which we'll insert in to the DOM before the `button` element.

```
function onSuccess(audioObject) {
var i, output = '';
for (i = 0; i < audioObject.length; i++) {
    output += 'Name: ' + audioObject[i].name + '<br />' +
            'Full Path: ' + audioObject[i].fullPath + '<br />' +
            'Type: ' + audioObject[i].type + '<br />' +
            'Created: '
+ new Date(audioObject[i].lastModifiedDate) + '<br />' +
            'Size: ' + audioObject[i].size + '<br />========';
    }
$('#record').before(output);
}
```

11. If we encountered an error during the process, the `onError` method will fire. The method will provide us with access to an error object, which contains the code for the error. We can use a switch statement here to customize the message that we will return to our user.

```
function onError(error) {
var errReason;
    switch(error.code) {
```

```
        case 0:
         errReason = 'The microphone failed to capture sound.';
     break;
        case 1:
            errReason = 'The audio capture application is currently
busy with another request.';
        break;
        case 2:
            errReason = 'An invalid parameter was sent to the
API.';
        break;
        case 3:
            errReason = 'You left the audio capture application
without recording anything.';
        break;
        case 4:
            errReason = 'Your device does not support the audio
capture request.';
        break;
    }
    alert('The following error occurred: ' + errReason);
}
```

12. If we run our application and press the button, the device's default audio recording
    application will open and we can record our audio.
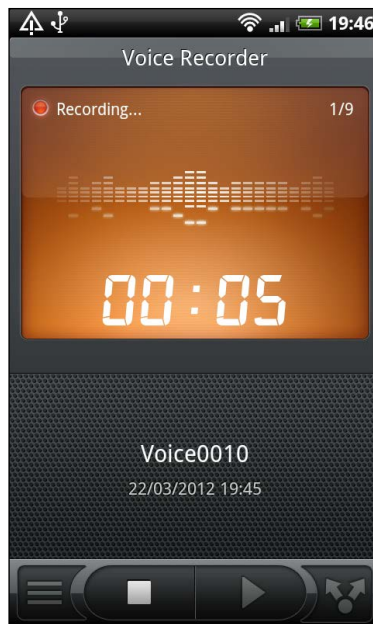
13. Once we have finished recording, our application will receive the audio data from the callback method and the output will look like the following screenshot:
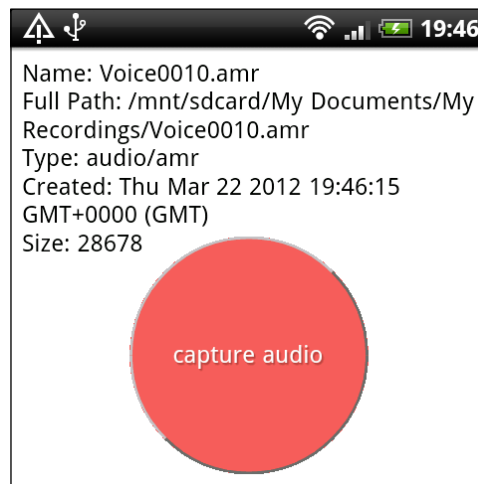
How it works...

The `Capture` object available through the PhoneGap API allows us to access the media capture capabilities of the device. By specifying the media type we wish to capture by calling the `captureAudio` method, an asynchronous call is made to the device's native audio recording application.

In this example we requested the capture of only one audio file. Setting the limit value within the optional configuration to a value greater than one can alter this.

The request is finished when one of two things happen:

 ▶ The maximum number of recordings has been created
 ▶ The user exits the native audio recording application

Following a successful callback from the request operation, we receive an array of objects that contains properties for each individual media file, which contains the following properties we can read:

 ▶ `name`: A `DOMString` object that contains the name of the file
 ▶ `fullPath`: A `DOMString` object that contains the full path of the file
 ▶ `type`: A `DOMString` object that includes the mime type of the returned media file
 ▶ `lastModifiedTime`: A `Date` object that contains the date and time that the file was last modified
 ▶ `size`: A `Number` value that contains the size of the file in bytes

84

> To find out more about the `captureAudio` capabilities offered by the PhoneGap API, check out the official documentation here: `http://docs.phonegap.com/en/2.0.0/cordova_media_capture_capture.md.html#capture.captureAudio`.

See also

► The *Playing audio files from the local filesystem or over HTTP* recipe

Recording audio within your application

The PhoneGap API provides us with the ability to record audio directly within our application, bypassing the native audio recording application.

How to do it...

We will use the `Media` object to create a reference to an audio file into which we'll record the audio data.

1. Create the initial layout for your HTML page. This will include the references to the jQuery and jQuery UI JavaScript libraries, the style sheets, and the Cordova JavaScript library. We'll also include an empty `script` tag block which will contain our custom code. This is shown in the following code block:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
         initial-scale=1, maximum-scale=1,
         minimum-scale=1, width=device-width;" />
    <title>Audio Recorder</title>
        <link rel="stylesheet" href="style.css" />
     <link type="text/css"
     href="jquery/css/smoothness/jquery-ui-1.8.23.custom.css"
     rel="stylesheet" />
        <script type="text/javascript"
        src="jquery/jquery-1.8.0.min.js"></script>
        <script type="text/javascript"
        src="jquery/jquery-ui-1.8.23.custom.min.js"></script>
        <script type="text/javascript"
                src="cordova-2.0.0.js"></script>
        <script type="text/javascript">
```

```
            </script>
    </head>
    <body>

    </body>
</html>
```

2. Include three elements within the `body` of our application. The first is a `div` with the `id` attribute set to `progressbar`. The second is a `div` with the `id` attribute set to `message`, and the third is a `button` element with the `id` attribute set to `record`.

```
<div id="progressbar"></div>
<div id="message"></div>
<button id="record"></button>
```

3. Now let's start adding our custom code within the empty `script` tag block. We'll begin by defining some global variables that we'll use in the application. We'll also create the event listener to ensure the device is ready before we proceed.

4. The `onDeviceReady` function will then run a new function called `recordPrepare`, as shown in the following code:

```
var maxTime = 10,
    countdownInt = 3,
    src,
    audioRecording,
    stopRecording;

document.addEventListener("deviceready",
 onDeviceReady, false);

function onDeviceReady() {
    recordPrepare();
}
```

5. The `recordPrepare` button will be used more than once in our application to reset the state of the button to record audio. Here, we unbind any actions applied to the button, set the HTML value and bind the `touchstart` handler to run a function called `recordAudio`:

```
function recordPrepare() {
    $('#record').unbind();
    $('#record').html('Start recording');
    $('#record').bind('touchstart', function() {
        recordAudio();
    });
}
```

86

6. Let's now create the `recordAudio()` function, which will create the audio file. We'll switch the value and bind events applied to our button to allow the user to manually end the recording. We also set the `Media` object to a variable, `audioRecording`, and pass in the destination for the file in the form of the `src` parameter, as well as the success and error callback methods.

7. A `setInterval` method is included, which will count down from three to zero to give the user some time to prepare for the recording. When the countdown is complete, we then invoke the `startRecord` method from the `Media` object and start another `setInterval` method. This will count to ten and will automatically stop the recording when the limit has been reached.

```
function recordAudio() {

    $('#record').unbind();
    $('#record').html('Stop recording');
    $('#record').bind('touchstart', function() {
        stopRecording();
    });

    src = 'recording_' + Math.round(new Date().getTime()/1000) +
'.mp3';

    audioRecording = new Media(src, onSuccess, onError);

    var startCountdown = setInterval(function() {

        $('#message').html('Recording will start in ' +
countdownInt + ' seconds...');
        countdownInt = countdownInt -1;

        if(countdownInt <= 0) {
            countdownInt = 3;
            clearInterval(startCountdown);
            audioRecording.startRecord();

            var recTime = 0;
                recInterval = setInterval(function() {
                recTime = recTime + 1;

                $('#message').html(Math.round(maxTime - recTime) +
' seconds remaining...');
```

```
                var progPerc = 100-((100/maxTime) * recTime);
                setProgress(progPerc);
                if (recTime >= maxTime) {
                    stopRecording();
                }
            }, 1000);
        }
    }, 1000);
}
```

8. As our recording is underway we can update the progress bar using the jQuery UI library and set it to the current value to show how much time is remaining.

```
function setProgress(progress) {
    $("#progressbar").progressbar({
            value: progress
    });
}
```

9. When a recording is stopped, we want to clear the interval timer and run the `stopRecord` method from the `Media` object. We'll also clear the value of the progress bar to zero, and reset the button bindings to prepare for the next recording.

```
function stopRecording() {
    clearInterval(recInterval);
    audioRecording.stopRecord();
    setProgress(0);
    recordPrepare();
}
```

Finally we can add in our success and error callback methods:

```
function onSuccess() {
    $('#message').html('Audio file successfully
    created:<br />' + src);
}
```

```
function onError(error) {
    $('#message').html('code: ' + error.code    + '\n' +
            'message: ' + error.message + '\n');
}
```

10. When we run the application to start recording, the output should look something like the following screenshot:
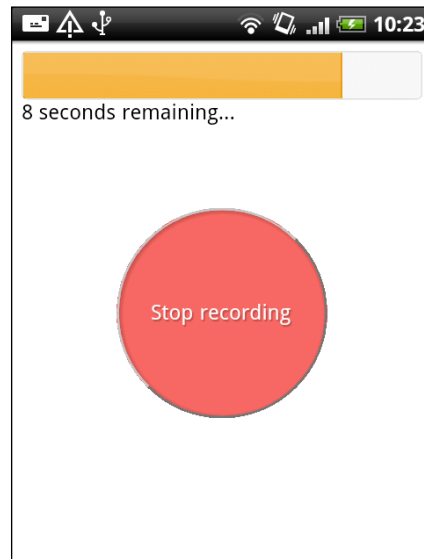
11. After a successful recording the user would be presented with the URI to the recorded file that we could use to access, upload, or play back the file, as shown in the following screenshot:
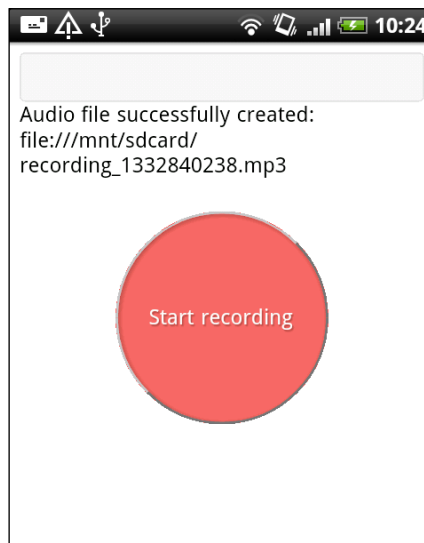
How it works...

The `Media` object has the ability to record and play back audio files. When choosing to utilize the object for recording, we need to provide the method with the URI for the destination file on the local device.

To start a recording we simply call the `startRecord` method from the `Media` object, and to stop the recording we need to call the `stopRecord` method.
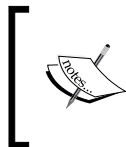
> To find out more about the available methods within the `Media` object, please refer to the official documentation, available here: `http://docs.phonegap.com/en/2.0.0/cordova_media_media.md.html#Media`.

See also

- The *Saving a file to device storage* recipe in *Chapter 2*, *File System, Storage, and Local Databases*
- The *Opening a local file from device storage* recipe in *Chapter 2*, *File System, Storage, and Local Databases*

Playing audio files from the local filesystem or over HTTP

The PhoneGap API provides us with a relatively straightforward process to play back audio files. These can be files stored within the application's local filesystem, bundled with the application, or over remote files accessible by a network connection. Wherever the files may be, the method of playback is achieved in exactly the same way.

How to do it...

We must create a new `Media` object and pass into it the location of the audio file we want to play back:

1. Create the initial layout for the HTML, and include the relevant references to the JavaScript and style sheets. In this example we are going to be using the jQuery Mobile framework (`http://jquerymobile.com/`):

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="width=screen.width; user-
```

```
scalable=no" />
    <title>Audio Player</title>
    <link rel="stylesheet" href="jquery/jquery.mobile-1.1.1.min.
css" type="text/css">
    <script type="text/javascript" src="jquery/jquery-1.8.0.min.
js"></script>
    <script type="text/javascript" src="jquery/jquery.mobile-
1.1.1.min.js"></script>
    <script type="text/javascript" src="cordova-2.0.0.js"></
script>

</head>
<body>

</body>
</html>
```

2. Create the layout for the application within the `body` tags. Here we are specifying a page for the jQuery Mobile framework, and four key `div` elements that have been assigned the role of buttons. We will reference their `id` attributes in our code, as follows:

```
<div data-role="page" id="page-home">
    <div data-role="header">
        <h1>PhoneGap Audio Player</h1>
    </div>

    <div data-role="content">

        <div data-role="button"
        id="playLocalAudio">Play Local Audio</div>
        <div data-role="button"
        id="playRemoteAudio">Play Remote Audio</div>
        <div data-role="button"
        id="pauseaudio">Pause Audio</div>
        <div data-role="button"
        id="stopaudio">Stop Audio</div>

        <div class="ui-grid-a">
            <div class="ui-block-a"> Current:
         <span id="audioPosition">0 sec</span></div>
            <div class="ui-block-b">Total:
         <span id="mediaDuration">0</span> sec</div>
        </div>
    </div>
</div>
```

91

3. Create a new `script` tag block within the `head` tag to contain our custom code, into which we'll add out an event listener to check that the device is ready to proceed, and also, some required global variables.

```
<script type="text/javascript">

document.addEventListener("deviceready", onDeviceReady, true);

var audioMedia = null,
  audioTimer = null,
  duration = -1,
  is_paused = false;


</script>
```

4. The `onDeviceReady` method binds a `touchstart` event to all four of our buttons in the main page content. For the local audio option, this example is set to read a file from the Android asset location. In both play functions, we pass the audio source to the `playAudio` method:

```
function onDeviceReady() {

  $("#playLocalAudio").bind('touchstart', function() {

    stopAudio();
    var srcLocal = '/android_asset/www/CFHour_Intro.mp3';
    playAudio(srcLocal);

  });

  $("#playRemoteAudio").bind('touchstart', function() {

    stopAudio();
    var srcRemote = 'http://traffic.libsyn.com/cfhour/Show_138_-_
ESAPI_StackOverflow_and_Community.mp3';
    playAudio(srcRemote);

  });

  $("#pauseaudio").bind('touchstart', function() {
      pauseAudio();
  });
```

```
    $("#stopaudio").bind('touchstart', function() {
        stopAudio();
    });

}
```

5. Now let's add in the `playAudio` method. This will firstly check to see if the `audioMedia` object has been assigned and we have an active audio file. If not, we will reset the duration and position values and create a new `Media` object reference, passing in the source of the audio file.

6. To update the duration and current position of the audio file, we will set a new interval timer which will check once every second and obtain these details from the `getCurrentPosition` and `getDuration` methods, available from the `Media` object:

```
function playAudio(src) {

    if (audioMedia === null) {
        $("#mediaDuration").html("0");
        $("#audioPosition").html("Loading...");
        audioMedia = new Media(src, onSuccess, onError);
        audioMedia.play();
    } else {
        if (is_paused) {
            is_paused = false;
            audioMedia.play();
        }
    }

    if (audioTimer === null) {
        audioTimer = setInterval(function() {
audioMedia.getCurrentPosition(
function(position) {
  if (position > -1) {

  setAudioPosition(Math.round(position));
    if (duration <= 0) {
 duration = audioMedia.getDuration();
        if (duration > 0) {
          duration = Math.round(duration);
        $("#mediaDuration").html(duration);
        }
    }
  }
 }
},
```

```
function(error) {
  console.log("Error getting position=" + error);
  setAudioPosition("Error: " + error);
}
      );
    }, 1000);
  }
}
```

7. The `setAudioPosition` method will update the content in the `audioPosition` element with the current details.

```
function setAudioPosition(position) {
    $("#audioPosition").html(position + " sec");
}
```

8. Now we can include the two remaining methods assigned to the touch handlers to control pausing and stopping the audio playback.

```
function pauseAudio() {
    if (is_paused) return;
    if (audioMedia) {
        is_paused = true;
        audioMedia.pause();
    }
}

function stopAudio() {
    if (audioMedia) {
        audioMedia.stop();
        audioMedia.release();
        audioMedia = null;
    }
    if (audioTimer) {
        clearInterval(audioTimer);
        audioTimer = null;
    }

    is_paused = false;
    duration = 0;
}
```

9. Lastly, let's write the success and error callback methods. In essence, they both reset the values to default positions in preparation for the next playback request.

```
function onSuccess() {
    setAudioPosition(duration);
```

```
            clearInterval(audioTimer);
            audioTimer = null;
            audioMedia = null;
            is_paused = false;
            duration = -1;
        }

        function onError(error) {
            alert('code: ' + error.code + '\n' +
                    'message: ' + error.message + '\n');
            clearInterval(audioTimer);
            audioTimer = null;
            audioMedia = null;
            is_paused = false;
            setAudioPosition("0");
        }
```

10. When we run the application on the device, the output will be similar to that shown in the following screenshot:

How it works...

The `Media` object has the ability to record and play back audio files. For media playback, we simply pass in the location of the audio file, remote or local, in to the Media instantiation call, along with the success and error handlers.

Playback is controlled by the `Media` objects' built-in methods available through the PhoneGap API.

> To find out more about all of the available methods within the Media object, please refer to the official documentation, available here: `http://docs.phonegap.com/en/2.0.0/cordova_media_media.md.html#Media`.

There's more...

In this example, we are assuming the developer is building for an Android device, and so we have referenced the location of the local file using the `android_asset` reference. To cater for other device operating systems you can use the `Device` object available in the PhoneGap API to determine which platform is running the application. Using the response from this check, you can then create a switch statement to provide the correct path to the local file.
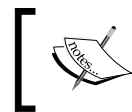
> To find out more about the Device object, please refer to the official documentation, available here: `http://docs.phonegap.com/en/2.0.0/cordova_device_device.md.html#Device`.

Capturing video using the devices video recording application

The PhoneGap API provides us with the ability to easily access the native video recording application on the mobile device and save the captured footage.

How to do it...

We will use the `Capture` object and the `captureVideo` method it contains to invoke the native video recording application.

1. Create the initial layout for the application, and include the JavaScript references to jQuery and PhoneGap. We'll also set a stylesheet reference pointing to `style.css`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Video Capture</title>
    <link rel="stylesheet" href="style.css" />
    <script type="text/javascript"
            src="jquery/jquery-1.8.0.min.js"></script>
    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>
```

```
        </head>
        <body>


        </body>
        </html>
```

2. Create a new `button` element within the `body` tags of the document, and set the `id` attribute to `record`. We'll use this to bind a touch handler to it.

   ```
   <button id="record">capture video</button>
   ```

3. Now let's start to add our custom code. Create a new `script` tag block before the closing `head` tag. Within this we'll set up our event listener, which will call the `onDeviceReady` method once the PhoneGap code is ready to run.

4. We'll also create a global variable called `videoCapture`, which will hold our `capture` object.

   ```
   <script type="text/javascript">

       document.addEventListener("deviceready",
        onDeviceReady, true);

       var videoCapture = '';

   </script>
   ```

5. We now need to create the `onDeviceReady` method. This will firstly assign the `capture` object to the variable we defined earlier. We'll also bind a `touchstart` event to the `button` element, which when pressed will run the `getVideo` method to commence the capture process:

   ```
   function onDeviceReady() {
      videoCapture = navigator.device.capture;

      $('#record').bind('touchstart', function() {
        getVideo();
   });
   }
   ```

6. To begin the video capture, we need to call the `captureVideo` method from the global `capture` object. This function accepts three parameters. The first is the name of the method to run after a successful transaction. The second is the name of the error handler method to run if we encounter any problems trying to obtain the video. The third is an array of configuration options for the capture request.

7. In this example we are requesting the application to retrieve two separate video captures, as shown in the following block of code:

```
function getVideo() {
videoCapture.captureVideo(
    onSuccess,
    onError,
    {limit: 2}
);
}
```

8. Following on from a successful transaction, we will receive an array of objects containing details for each video file that was captured. We'll loop over that array and generate a string containing all of the properties for each file, which we'll insert in to the DOM before the `button` element.

```
function onSuccess(videoObject) {
var i, output = '';
for (i = 0; i < videoObject.length; i += 1) {
    output += 'Name: ' + videoObject[i].name + '<br />' +
        'Full Path: ' + videoObject[i].fullPath + '<br />' +
            'Type: ' + videoObject[i].type + '<br />' +
            'Created: '
+ new Date(videoObject[i].lastModifiedDate) + '<br />' +
            'Size: ' + videoObject[i].size + '<br />========';
    }
$('#record').before(output);
}
```

9. If we encountered an error during the process, the `onError` method will fire. The method will provide us with access to an error object, which contains the code for the error. We can use a switch statement here to customize the message that we will return to our user, as follows:

```
function onError(error) {
var errReason;
    switch(error.code) {
        case 0:
          errReason = 'The camera failed to capture video.';
      break;
        case 1:
            errReason = 'The video capture application is currently
busy with another request.';
```

```
            break;
        case 2:
            errReason = 'An invalid parameter was sent to the
API.';
        break;
        case 3:
            errReason = 'You left the video capture application
without recording anything.';
        break;
        case 4:
            errReason = 'Your device does not support the video
capture request.';
        break;
    }
    alert('The following error occurred: ' + errReason);
}
```

10. If we run our application and press the button, the device's default video recording application will open and we can record our video, as shown in the following screenshot:
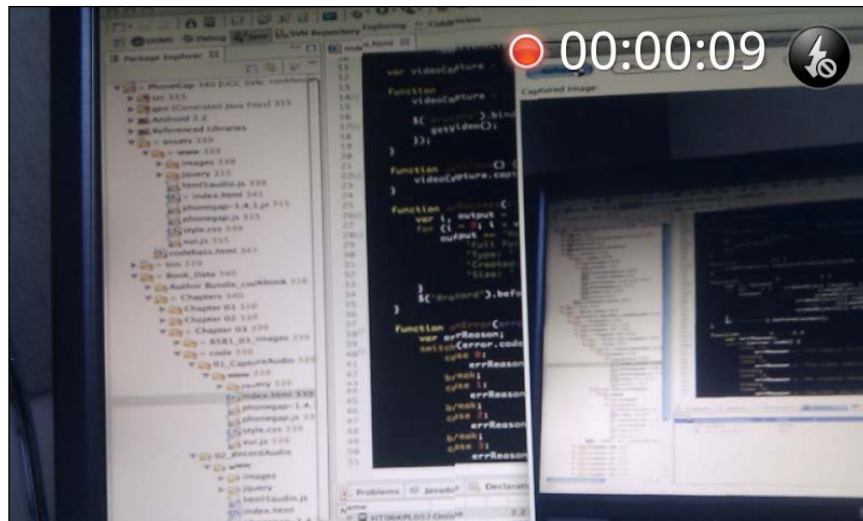
11. Once we have finished recording, our application will receive the video data from the callback method and the output will look like the following screenshot:
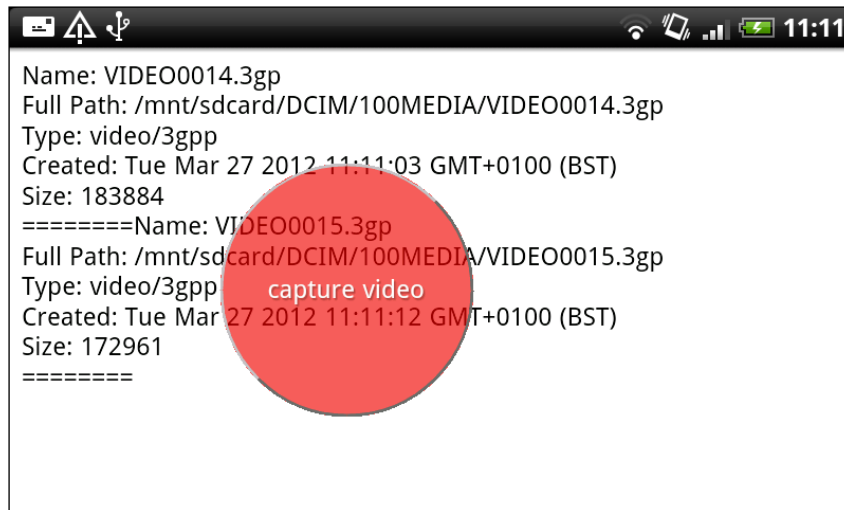
How it works...

The `Capture` object available through the PhoneGap API allows us access to the media capture capabilities of the device. By specifying the media type we wish to capture by calling the `captureVideo` method, an asynchronous call is made to the device's native video recording application.

In this example, we forced the method to request two video captures by setting the limit property in the optional configuration options – the default value for the limit is set to one.

The request is finished when one of the two things happen:

► The maximum number of recordings has been created
► The user exits the native video recording application

Following a successful callback from the request operation, we receive an array of objects that contains properties for each individual media file, which contains the following properties, as we can read:

► `name`: A `DOMString` object that contains the name of the file
► `fullPath`: A `DOMString` object that contains the full path of the file
► `type`: A `DOMString` object that includes the mime type of the returned media file

▸ `lastModifiedTime`: A `Date` object that contains the date and time that the file was last modified

▸ `size`: A `Number` value that contains the size of the file in bytes

> To find out more about the `captureVideo` capabilities offered by the PhoneGap API, check out the official documentation here: `http:// docs.phonegap.com/en/2.0.0/cordova_media_capture_ capture.md.html#capture.captureVideo`.

Loading a photograph from the devices camera roll/library

Different devices will store saved photographs in different locations, typically in either a photo library or saved photo album. The PhoneGap API gives developers the ability to select or specify from which location an image should be retrieved.

How to do it...

We must use the `getPicture` method available from the `Camera` object to select an image from either the device library or to capture a new image directly from the camera.

1. Create the initial layout for the HTML page, and include the required references to the jQuery and Cordova JavaScript libraries.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="width=screen.width; user-
scalable=no" />
    <title>Photo Finder</title>
    <script type="text/javascript"
            src="jquery/jquery-1.8.0.min.js"></script>
    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>
</head>
<body>

</body>
</html>
```

2. The `body` tag of our application will contain four elements. We'll need to provide two buttons, both with the `class` attribute set to `photo`, and each one with the `id` attribute set to `cameraPhoto` and `libraryPhoto` respectively.

3. Well also need to create a `div` element with `id` set to `message` and an `img` tag with `id` set to `image`, as shown in the following code block:

```html
<button class="photo"
id="cameraPhoto">Take New Photo</button><br />
<button class="photo"
id="libraryPhoto">Select From Library</button><br />
<div id="message"></div><br />
<img id="image" />
```

4. Create a new `script` tag block within the head of the document and include the event listener that will fire when the PhoneGap native code is compiled and ready. Below this, create the `onDeviceReady` function, within which we'll apply a bind handler to the buttons by using the jQuery class selector.

5. Depending on the value of the selected button's `id` attribute, the `switch` statement will run the particular method to obtain the image.

```javascript
<script type="text/javascript">

document.addEventListener("deviceready",onDeviceReady,false);

function onDeviceReady() {
$('.photo').bind('touchstart', function() {
  switch($(this).attr('id')) {
    case 'cameraPhoto':
      capturePhoto();
    break;
    case 'libraryPhoto':
      getPhoto();
    break;
  }
});
}


</script>
```

6. Let's now add the first of our image capture functions, `capturePhoto`. This calls the `getPicture` method from the `Camera` object. Here we are asking for the highest quality image returned and with a scaled image to match the provided sizes.

```
function capturePhoto() {
  navigator.camera.getPicture(onSuccess, onFail,
    {
   quality: 100,
      targetWidth: 250,
      targetHeight: 250
    }
  );
}
```

7. The second image capture method is `getPhoto`. In this method we once again call the `getPicture` method, but we now pass in the `sourceType` option value to request the image to be selected from the device photo library.

```
function getPhoto() {
  navigator.camera.getPicture(onSuccess, onFail,
    {
      quality: 100,
      destinationType: Camera.DestinationType.FILE_URI,
      sourceType: Camera.PictureSourceType.PHOTOLIBRARY,
      targetWidth: 250,
      targetHeight: 250
    }
  );
}
```

8. Finally, let's add in the success and error handlers, which both of our capture methods will use. The `onSuccess` method will display the returned image, setting it as the source for the `image` element:

```
function onSuccess(imageURI) {
  $('#image').attr('src', imageURI);
  $('#message').html('Image location: ' + imageURI);
}

function onFail(message) {
  $('#message').html(message);
}
```

9. When we run the application on the device, the output would look something like the following screenshot:
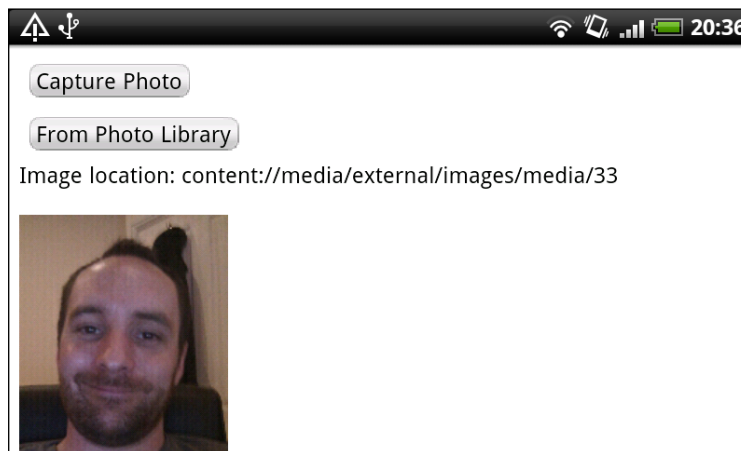
How it works...

The `Camera` object available through the PhoneGap API allows us to interact with the default camera application on the device. The `Camera` object itself contains just the one method, `getPicture`. Depending on the `sourceType` value being sent through in the capture request method, we can obtain the image from either the device camera or by selecting a saved image from the photo library or photo album.

In this example we retrieved the URI for the image to use as the source for an `img` tag. The method can also return the image as a Base64 encoded image, if requested.

> There are a number of optional parameters that we can send in to the method calls to customize the camera settings. For detailed information about each parameter, please refer to the official documentation available here: `http://docs.phonegap.com/en/2.0.0/ cordova_camera_camera.md.html#cameraOptions`.

There's more...

In this recipe, we requested that the images be scaled to match a certain dimension, while maintaining the aspect ratio. When selecting an image from the library or saved album, PhoneGap resizes the image and stores it in a temporary cache directory on the device. While this means resizing, it is as painless as we would want it to be, the image may not persist or will be overwritten when the next image is resized.

If you want to save the resized images in a permanent location after creating them, make sure you check out the recipes within this book on how to interact with the local file system and how to save files.

Now that we can easily obtain an image from the device, there are a number of things we can do with it. For an example, take a look at the next recipe in this book.

See also

▸ The *Uploading a file to a remote server via a POST request* recipe in *Chapter 2, File System, Storage, and Local Databases*

Applying an effect to an image using canvas

Capturing a photo on your device is fantastic, but what can we do with an image once we have it in our application? In this recipe, we'll create some simple functions to edit the color of an image without altering the original source.

How to do it...

We must create the use of the HTML5 canvas element to load and edit the values of a stored image, by performing the following steps:

1. Create the initial HTML layout and include the references to the jQuery and Cordova JavaScript files in the `head` tag of the document:

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="width=screen.width; user-
scalable=no" />
    <title>Image Effects</title>
    <script type="text/javascript"
            src="jquery/jquery-1.8.0.min.js"></script>
    <script type="text/javascript"
            src="cordova-2.0.0.js"></script>
</head>
<body>

</body>
</html>
```

2. Include a reference to the `rgb.js` file available in the downloadable code files of this book, below the Cordova JavaScript reference. This contains a required array of variables for one of our image manipulation functions.

```
<script type="text/javascript" src="cordova-2.0.0.js"></script>

<script type="text/javascript"
src="rgb.js"></script>


</head>
```

3. The `body` tag of our application will hold three `button` elements, each with a specific `id` attribute that we will reference within the custom code. We'll also need an `img` tag with the `id` attribute set to `sourceImage`, which will display the original image we want to manipulate.

4. Finally, we need to include a `canvas` element with the `id` attribute set to `myCanvas`, as shown in the following code block:

```
<button id="grayscale">Grayscale</button>
<button id="sepia">Sepia</button>
<button id="reset">Reset</button><br />

<img id="sourceImage" src="awesome.jpg"
alt="source image" height="150" width="150" />
<canvas id="myCanvas" width="300" height="300"></canvas>
```

5. Let's start to add our custom code. Create a new `script` tag block before the closing `head` tag, into which we'll add our event listener to ensure that PhoneGap is fully loaded before we proceed. We'll also create some required global variables.

```
<script type="text/javascript">

  document.addEventListener(
    "deviceready", onDeviceReady, true);

var canvas,
    context,
    image,
    imgObj,
    noise = 20;

</script>
```

6. Create the `onDeviceReady` method, which will run once the native code is ready. Here, we firstly want to run a method called `reset` which will restore our `canvas` to its default source. We'll also bind the `touchstart` handlers to our three buttons, each of which will run their own methods.

```
function onDeviceReady() {
  reset();

  $('#grayscale').bind('touchstart', function() {
    grayscaleImage();
  });

  $('#sepia').bind('touchstart', function() {
    processSepia();
  });

  $('#reset').bind('touchstart', function() {
    reset();
  });
}
```

7. The `reset` method creates the `canvas` reference and its context, and applies the source from our starting image into it.

```
function reset() {
  canvas = document.getElementById('myCanvas');
  context = canvas.getContext("2d");
  image = document.getElementById('sourceImage');
  context.drawImage(image, 0, 0);
}
```

8. Our first image manipulation function is called `grayscaleImage`. Let's include this now and within it we'll loop through the pixel data of our image, which we can retrieve from the `canvas` element using the `getImageData` method, as shown in the following code block:

```
function grayscaleImage() {
  var imageData = context.getImageData(0, 0, 300, 300);
  for (var i = 0, n = imageData.data.length; i < n; i += 4) {
    var grayscale = imageData.data[i] * .3 +
    imageData.data[i+1] * .59 + imageData.data[i+2] * .11;
   imageData.data[i] = grayscale;
   imageData.data[i+1] = grayscale;
   imageData.data[i+2] = grayscale;
  }
  context.putImageData(imageData, 0, 0);
}
```

107

9. Our second manipulation function is called `processSepia`. Once again, we obtain the image data from our `canvas` element and loop through each pixel applying the changes as we go.

```
function processSepia() {
var imageData =
  context.getImageData(0,0,canvas.width,canvas.height);
  for (var i=0; i < imageData.data.length; i+=4) {
    imageData.data[i] = r[imageData.data[i]];
    imageData.data[i+1] = g[imageData.data[i+1]];
    imageData.data[i+2] = b[imageData.data[i+2]];
    if (noise > 0) {
      var noise = Math.round(noise - Math.random() * noise);
      for(var j=0; j<3; j++){
        var iPN = noise + imageData.data[i+j];
        imageData.data[i+j] = (iPN > 255) ? 255 : iPN;
      }
    }
  }
  context.putImageData(imageData, 0, 0);
};
```

10. When we run the application on the device, after selecting a button to change our default image the output would look something like the following screenshot:

How it works...

When we start to process a change to the canvas image, we first obtain the data using the `getImageData` method available through the `canvas` context. We can easily access the information for each pixel within the returned image object and its `data` attribute.

With the `data` tag in an array, we can loop over each pixel object, and then over each value within each pixel object.

> Pixels contain four values: the red, green, blue, and alpha channels respectively

By looping over each specific color channel in each pixel, we can alter the values, thereby changing the image. We can then set the revised image as the source in our canvas by using the `putImageData` method to set it back in to the context of our `canvas`.

There's more...

Although this recipe does not involve any PhoneGap specific code with the exception of the `onDeviceReady` method, it was included here for the following three reasons:

- As an example to show you how you might like to work with images captured using the PhoneGap API
- To remind you of or introduce you to the power of HTML5 elements and how we can work with the `canvas` tag
- Because it's pretty cool

4
Working with Your Contacts

In this chapter, we will cover:

- ▶ Listing all available contacts
- ▶ Displaying contact information for a specific individual
- ▶ Creating and saving a new contact

Introduction

With the ever expanding and increasing technological resources and advancements, mobile devices now contain increasingly powerful processors and provide users and consumers with an impressive array of features.

Above all of the apps, widgets, and features your device can manage, let us not forget that the primary function of a mobile device (certainly a mobile phone) is to hold contact information for your friends, family, or favorite local takeaway restaurants.

All of the recipes in this chapter will focus on interacting with your device contact database and how we can list, display, and add new contacts into it.

Listing all available contacts

Developers can access, read from, and filter the contacts saved within the device contact database, allowing us to query and work with the address book on the device.

How to do it...

We will create an application to read all of the contacts from the device database and output them as a dynamic list.

1. Create a new HTML layout for this application. We will be using the jQuery Mobile framework in this example, and so we need to include the required `JavaScript` and `CSS` references within the `head` tag of our document.

2. We'll also need to include the `Cordova JavaScript` file to interact with the native features on the device.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <title>My Contacts</title>
    <link rel="stylesheet"
    href="jquery/jquery.mobile-1.1.1.min.css"
    type="text/css">
    <script type="text/javascript"
    src="jquery/jquery-1.8.0.min.js"></script>
    <script type="text/javascript"
    src="jquery/jquery.mobile-1.1.1.min.js"></script>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>

</head>
<body>

</body>
</html>
```

3. Let's add the initial page for our application within the body of the HTML document. Here we will create the page `div` element with the `id` attribute set to `contacts-home`, as shown in the following code block:

```html
<div data-role="page" id="contacts-home">

    <div data-role="header">
        <h1>My Contacts</h1>
    </div>
```

```
    <div data-role="content">
  </div>

</div>
```

4. Within the content `div` element, create a new unordered list block. Set the `id` attribute to `contactList`, the `data-role` attribute to `listview`, and the `data-inset` attribute to `true`.

```
<div data-role="content">

  <ul id="contactList" data-role="listview"
    data-inset="true">

  </ul>

</div>
```

5. With the HTML UI complete, let's now focus on creating the custom code to interact with the contacts. Create a new `script` tag block within the `head` of the document, and include the event listener to check whether the device is ready, as well as the callback method it will run, that is the `onDeviceReady` method.

```
<script type="text/javascript">

  document.addEventListener("deviceready",
    onDeviceReady, false);

  function onDeviceReady() {
    getAllContacts();
  }
</script>
```

6. The application will execute the `getAllContacts` method, which will read from the device contacts database. To achieve this, we'll set the optional parameter of the `contactFindOptions` object to return multiple contacts.

> The `multiple` parameter is set to false by default, which will only return one contact.

7. We then set the required `contactFields` parameter to specify which fields should be returned in each `Contact` object.

8. Finally, we call the `find()` method, passing in the fields, the options, and the success and error callback method names.

```
function getAllContacts() {
var options = new ContactFindOptions();
  options.filter = "";
  options.multiple = true;
  var fields = ["name", "phoneNumbers",
                "birthday", "emails"];
  navigator.contacts.find(fields,
    onAllSuccess, onError, options);
}
```

9. Following a successful response, the `onAllSuccess` method will return an array of `Contact` objects for us to work with. We will initially loop over the returned results and push each `Contact` object into a new array object, `arrContactDetails`, which allows us to sort the results alphabetically. If no results were returned, we'll output a user-friendly message.

```
function onAllSuccess(contacts) {

  if(contacts.length) {

    var arrContactDetails = new Array();
      for(var i=0; i<contacts.length; ++i){
        if(contacts[i].name){
          arrContactDetails.push(contacts[i]);
        }
      }

    arrContactDetails.sort(alphabeticalSort);

    // more code to go here

  } else {
    $('#contactList').append('<li><h3>Sorry,
    no contacts were found</h3></li>');
  }
  $('#contactList').listview("refresh");
}
```

10. Include the `alphabeticalSort` function, which will sort each contact in ascending order using the formatted version of the name.

```
function alphabeticalSort(a, b) {
  if (a.name.formatted < b.name.formatted){
    return -1;
```

```
    }else if (a.name.formatted > b.name.formatted){
      return  1;
    }else{
      return 0;
    }
}
```

11. To create our contact list, the following code will go directly beneath the `arrContactDetails.sort(alphabeticalSort);` call in the code. This will loop over the sorted array and create the list items for each contact, setting the `Contact` object `ID` and the formatted name into each list item. It will also create the list divider to differentiate each group of contacts by the first letter of the name.

```
var alphaHeader = arrContactDetails[0].name.formatted[0];
for(var i=0; i<arrContactDetails.length; ++i) {
  var contactObject = arrContactDetails[i];
    if( alphaHeader != contactObject.name.formatted[0] ) {
      alphaHeader = contactObject.name.formatted[0];
      $('#contactList').append('<li data-role="list-divider">'
       + alphaHeader + '</li>');
      $('#contactList').append(
       '<li class="contact_list_item" id="' +
       contactObject.id + '"><a href="#contact-info">' +
         contactObject.name.formatted + ' (' +
         contactObject.id + ')</a></li>'
    );
    } else {
      if( i == 0 ) {
        $('#contactList').append(
     '<li data-role="list-divider">'
     + alphaHeader + '</li>');
      }
      $('#contactList').append(
      '<li class="contact_list_item" id="'
      + contactObject.id + '"><a href="#contact-info">' +
      contactObject.name.formatted + ' (' +
      contactObject.id + ')</a></li>');
  }
}
```

12. Finally, include the `onError` callback method which will run if we encounter any issues obtaining data from the `find()` method:

```
function onError(error) {
alert('An error has occurred: ' + error.code);
}
```

115

13. When we run the application on a device, we will see the populated list looking something like the following screenshot:
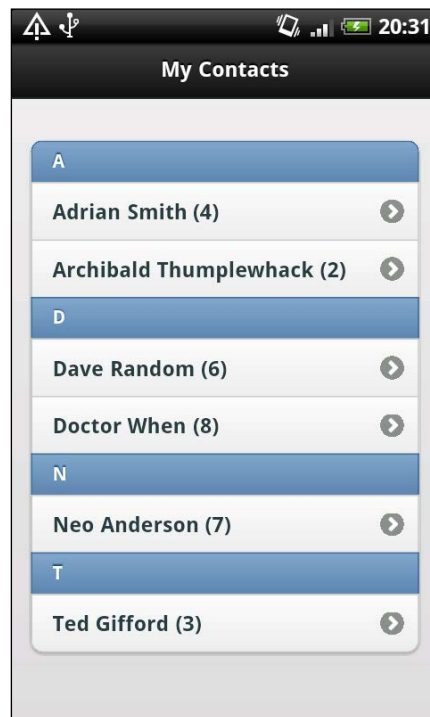
Next to each name in the list, we can see the ID of the contact as used in the device contact database. This value is also set in the `id` attribute of each list item.

How it works...

The `contacts.find()` method available from the PhoneGap API is designed to query the device contacts database to obtain and return an array of `Contact` objects. We set the contact fields into the required parameter of the function, which acts as a search qualifier for the transaction. Only the fields we set in the `contactFields` parameter will be included as properties of the returned `Contact` objects. Using this parameter, we can choose exactly what details for each contact we want to obtain from the request.

Following a successful result from the `find()` method, an array of `Contact` objects is passed to the success callback method. Once we have received this information, we then loop over the array to output the alphabetically sorted information into our unordered list, making use of the jQuery Mobile framework `listview` item for clear display.

For a comprehensive look at the methods and properties available to use through the `Contact` object, please refer to the official documentation, available at:

`http://docs.phonegap.com/en/2.0.0/cordova_contacts_`
`contacts.md.html#Contact.`

There's more...

In this example we specifically set values for the `contactFields` parameter to return in each `Contact` object. If this were left blank, we would receive only the `id` property of each contact. If we wanted to receive all available properties for each contact, we could set the value to a wildcard asterisk (`*`).

See also

▸ The *Creating a jQuery Mobile layout* recipe in *Chapter 7*, *User Interface Development with jQuery Mobile*

Displaying contact information for a specific individual

Working with the contact database, developers can easily obtain a full array of all `Contact` objects saved on the device. We want to be able to obtain and view the saved contact information for specific individuals if we choose to drill down and filter a certain contact from the database.

Getting ready

For this recipe, we'll build on the code created in the previous recipe, *Listing all available contacts*. This will give us a good head start to add additional functionality into the application. Therefore, if you haven't as of yet completed the *Listing all available contacts* recipe, it may help to go there first.

How to do it...

To manage the selected contact information, we will first make use of the `localStorage` API available and harness some more power from the jQuery Mobile framework.

1. When the user selects a contact from the list, we want to take them to a new page to show the details. Let's add a new page into `index.html` below the current one.

117

2. Set a new `div` element with the `data-role` attribute set to `page` and the `id` attribute set to `contact-info`. Within this we will add our page header with the `id` attribute set to `contact_header`. We will also include a **Back** button that will take the user back to the original page by referencing the `id` attribute in the link.

3. We will keep the `h1` tag empty as we'll populate it with the contact's name.

```
<div data-role="page" id="contact-info">


    <div id="contact_header" data-role="header">
      <a href="#contacts-home" id="back" data-icon="back"
        data-direction="reverse">Back</a>
        <h1></h1>
    </div>



</div>
```

4. Below the page header we will create a content `div` element with the `id` attribute set to `contact_content` that contains four form field items, which will display the given name, family name, phone number, and e-mail address for the chosen contact.

```
<div id="contact_content" data-role="content">

  <div data-role="fieldcontain">
    <label for="givenName">First Name:</label>
    <input type="text" name="givenName"
          id="givenName" disabled />
  </div>
  <div data-role="fieldcontain">
    <label for="familyName">Last Name:</label>
    <input type="text" name="familyName"
          id="familyName" disabled />
  </div>
  <div data-role="fieldcontain">
    <label for="phone">Phone:</label>
    <input type="text" name="phone"
          id="phone" disabled />
  </div>
<div data-role="fieldcontain">
  <label for="email">Email:</label>
    <input type="text" name="email"
          id="email" disabled />
</div>

</div>
```

5. At the top of the custom JavaScript code, create a global variable to reference the `localStorage` API. We'll also include a global variable called `contactInfo` which we will use to hold data later on.

```
<script type="text/javascript">

var localStorage    =       window.localStorage;
var contactInfo;

document.addEventListener("deviceready",
onDeviceReady, false);
```

6. Let's now amend the `onAllSuccess` method, which writes out the list of all contacts. Within the loop we'll add in a small portion of code that will add each item to the `localStorage`. Here we will store the entire contact object for each listing, and use the `ID` for each contact as the key which we can use to retrieve the information.

```
var alphaHeader = arrContactDetails[0].name.formatted[0];
for(var i=0; i<arrContactDetails.length; ++i) {
  var contactObject = arrContactDetails[i];
    if( alphaHeader != contactObject.name.formatted[0] ) {
      alphaHeader = contactObject.name.formatted[0];
      $('#contactList').append('<li data-role="list-divider">' +
alphaHeader + '</li>');
      $('#contactList').append('<li class="contact_list_item"
id="' + contactObject.id + '"><a href="#contact-info">' +
contactObject.name.formatted + ' (' + contactObject.id + ')</a></
li>');
    } else {
      if( i == 0 ) {
        $('#contactList').append('<li data-role="list-divider">' +
alphaHeader + '</li>');
      }
    $('#contactList').append('<li class="contact_list_item" id="'
+ contactObject.id + '"><a href="#contact-info">' + contactObject.
name.formatted + ' (' + contactObject.id + ')</a></li>');
  }

  localStorage.setItem(
      contactObject.id,JSON.stringify(contactObject)
    );
}
```

> The localStorage API saves data as a key/value pair, and can only contain strings. As such, we convert the object to a string before saving it. For more information, check out the *Caching content using the web storage local storage API recipe* in *Chapter 2*, *File System, Storage, and Local Databases*.

7. Each of our generated list items references a particular contact. We have stored the specific ID for each contact as an attribute in each list item. Create an event handler that will obtain the value of the contact ID from the selected list item and pass it through to the getContactByID method.

```
$(document).on('click', '#contactList li.contact_list_item',
function(){

var selectedID = $(this).attr('id');
  getContactByID(selectedID);

});
```

8. Let's now add the getContactByID function, which accepts the selected ID of the contact as a required parameter. This will obtain the selected contact information from the localStorage database and assign it to the contactInfo variable we set earlier. It will then send the user to a new page within the application.

```
function getContactByID(contactID) {
  contactInfo =
  JSON.parse(localStorage.getItem(contactID));
    $.mobile.changePage($('#contact-info'));
}
```

9. We now have the contact information stored, but we need to populate the form fields on the information page with the details. Let's add a new event handler to the code to detect a jQuery Mobile pagechange event, which will run a method called onPageChange.

```
$(document).bind("pagechange", onPageChange);
```

10. The onPageChange function will obtain the id of the page we have changed to. If it matches contact-info, we will first clear the values of all form fields, and then set each one with the details from the contactInfo object. We are also setting the h1 tag in the header with the contact name.
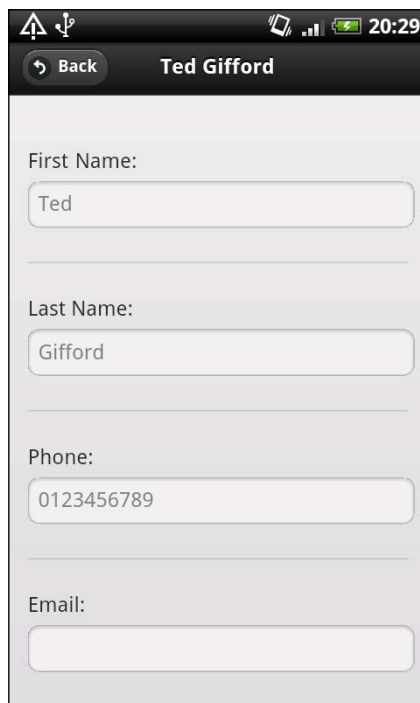
```
function onPageChange(event, data) {
  var toPageId = data.toPage.attr("id");

    switch (toPageId) {
      case 'contact-info':
      clearValues();
```

```
     $('#contact_header h1')
   .html(contactInfo.name.formatted);
    $('#givenName').val(contactInfo.name.givenName);
   $('#familyName').val(contactInfo.name.familyName);

  $('#phone').val(contactInfo.phoneNumbers[0].value);
   $('#email').val(contactInfo.emails[0].value);

   break;
  }
}
```

11. Finally, let's add the `clearValues` function, which will reset all form fields on the page where the input type is `text`.

```
function clearValues() {
  $('input[type=text]').each(function() {
    $('#' + this.id + '').val('');
  });
}
```

12. When we run the application on the device and select a contact, the resulting page would look like the following screenshot:

How it works...

In this recipe we extended the previous code to fulfill the desired functionality. We amended the initial code when we looped over the returned array of `Contact` objects and added in some code to set each `Contact` object into the `localStorage` database available on the device, using its `ID` property as the key for the storage entry.

When a contact was selected from the list, we were able to use that list item's `ID` attribute to obtain the saved `Contact` object from the `localStorage` database before taking the user to the next page by using the jQuery Mobile framework `mobile.changePage()` function.

With the `Contact` object stored in an accessible variable, we were then able to read the properties set within it and output them to the user.

See also

 ▸ The *Caching content using the web storage local storage API* recipe in *Chapter 2*, *File System, Storage, and Local Databases*

Creating and saving a new contact

Having an address book or an application that can read contact information from the database is fantastic, but wouldn't it be even better if we could add contacts to the database? The good news is that the PhoneGap API not only provides a way to read the information, but it also gives developers an incredibly powerful but easy way to add information too.

Getting ready

For this recipe, we'll build on the code created in the previous recipe, *Displaying contact information for a specific individual*. This will give us a good head start to add additional functionality into the application. Therefore, if you haven't yet completed the *Display contact information for a specific individual* recipe, it may help to go there first.

How to do it...

To store a new contact to the device database, we will create a form and save the new information into a `Contact` object:

1. Firstly, let's edit the default loading page for our application to include a button to take the user to a new page to add a contact.

   ```
   <div data-role="page" id="contacts-home">

       <div data-role="header">
   ```

```
    <h1>My Contacts</h1>
    <a href="#contact-add" id="back"
        data-icon="add">Add</a>
</div>
```

2. Now let's create the new page that will enable the user to input new contact information. Create a new jQuery Mobile page and set the `id` attribute to `contact-add`.

3. Within the header `div` element, add a new link that will take the user back to the home page, bypassing the save functionality which we will shortly be adding.

```
<div data-role="page" id="contact-add">

    <div data-role="header">
      <a href="#contacts-home" id="back"
        data-icon="back"
        data-direction="reverse">Back</a>
        <h1>Add Contact</h1>
    </div>

    <div data-role="content">

    </div>

</div>
```

4. Within the content `div` element block, we will add a new form element with the `id` attribute set to `new_contact_form` that contains a number of form field items. These will be used to enter the new information for the contact's given name, family name, phone number, and e-mail address.

5. The last form field block contains an input button with the `id` attribute set to `saveBtn`, which we'll reference via jQuery code to perform the save process.

6. Finally, we also include a hidden form item called `displayName`. We will populate this value after the form has been submitted, and will use it to store into the new `Contact` object:

```
<form id="new_contact_form">
<div data-role="fieldcontain">
  <label for="givenName">First Name:</label>
    <input type="text" name="givenName"
      id="givenName" />
  </div>
  <div data-role="fieldcontain">
    <label for="familyName">Last Name:</label>
    <input type="text" name="familyName"
```

```
        id="familyName" />
  </div>
  <div data-role="fieldcontain">
    <label for="phone">Phone:</label>
    <input type="tel" name="phone"
      id="phone" />
  </div>
  <div data-role="fieldcontain">
    <label for="email">Email:</label>
    <input type="email" name="email"
      id="email" />
  </div>
  <div data-role="fieldcontain">
  <input type="button" name="saveBtn"
      id="saveBtn" value="Save Contact" />
    <input type="hidden" name="displayName"
      id="displayName" />
</div>
</form>
```

7. With the layout and UI for the page complete, let's now focus on the JavaScript functionality to process the new contact information. Amend the `onPageChange` function to add a new case within the switch statement to check for the page `id` value `contact-add`. If it matches, everything within this case statement will be executable within that page context.

8. Firstly, we'll bind a `touchstart` event to the `saveBtn` button element which will commence the save process.

    ```
    case 'contact-add':

    $('#saveBtn').bind('touchstart',function(){

    });

    break;
    ```

9. We can now populate the value of the `displayName` hidden form field by concatenating the values from the `givenName` and `familyName` form fields provided.

    ```
    $('#saveBtn').bind('touchstart',function(){

      $('#displayName').val(
        $('#new_contact_form #givenName').val() +
    ```

```
        ' ' + $('#new_contact_form #familyName').val()
    );
});
```

10. Before we can create the new `Contact` object, we must first format the submitted information into the required format which we can the send through as a parameter. The information is accepted in the form of a structural object containing key/value pairs. We could call each form field individually to create this, but in doing so we would have a tightly coupled dependency on the specific form fields.

11. Here we can make use of the jQuery library and create a serialized array of all of the form fields within the form, which we have referenced by its `id` attribute.

12. We can then loop over the array to create the key/value pairs as expected to return the structure of information. Within the loop we have set a `switch` statement to check for the name of the submitted form value. PhoneGap manages e-mails and phone numbers in a separate way to the standard name contact fields. If they exist, we set them using a new `ContactField` object.

```
var arrContactInfo = $('#new_contact_form').serializeArray();

var phoneNumbers = new Array();
var emails = new Array();

var contactInfo = '{';

  for(var i=0; i<arrContactInfo.length; i++) {
    switch (arrContactInfo[i].name) {
     case 'phone':
    if (arrContactInfo[i].value) {
      phoneNumbers[0] =
        new ContactField('mobile',
          arrContactInfo[i].value, true);
    }
    break;
    case 'email':
      if(arrContactInfo[i].value) {
      emails[0] =
        new ContactField('work',
          arrContactInfo[i].value, true);
      }
    break;
    default:
      contactInfo += '"' + arrContactInfo[i].name + '" : "'
     + arrContactInfo[i].value + '"';
```

```
        if(i < arrContactInfo.length-1) {
         contactInfo += ', '
        }
      }
    }
contactInfo += '}';
```

13. We can then create a new `Contact` object and pass in the `contactInfo` variable we have just created, before we save the contact to the device database.

```
var newContact =
  navigator.contacts.create(JSON.parse(contactInfo));

newContact.phoneNumbers = phoneNumbers;
newContact.emails = emails;

newContact.save(onSaveSuccess, onError);
```

> The `contacts.create` method generates and returns a new `Contact` object, populated with the information we have provided. At this point we have only created the new `Contact` object. To ensure it has been persisted and saved to the device database, we must remember to use the `save()` method.

14. When defining the `save()` method, we also included two callback methods to handle the successful save of any errors that may have arisen from the process. In the `onSaveSuccess` function, we will navigate the user back to the home page using the jQuery Mobile framework's built-in `changePage` method.

15. We will then refresh the `contactList` list element to show the new data stored within the device database.

```
function onSaveSuccess(contact) {
  $.mobile.changePage($('#contacts-home'));
  $('#contactList').listview("refresh");
}
```

16. When we run the application on the device, the home page would now look like the following screenshot with the new add button:
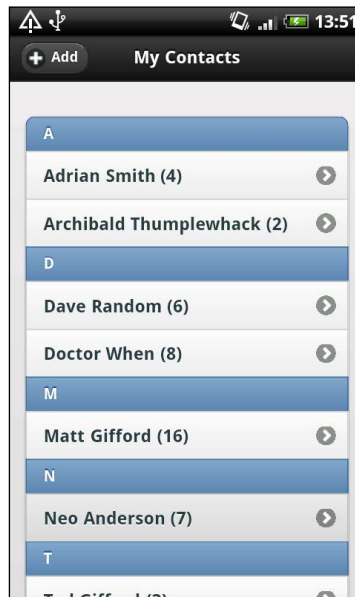
17. By selecting to add a new contact, the user would then be presented with the new form page, as shown in the following screenshot:

18. And finally, once the submission has been made, the user is then taken back to the home page listing all contacts, which would now show the new contact, created and saved into the device database. This is shown in the following screenshot:
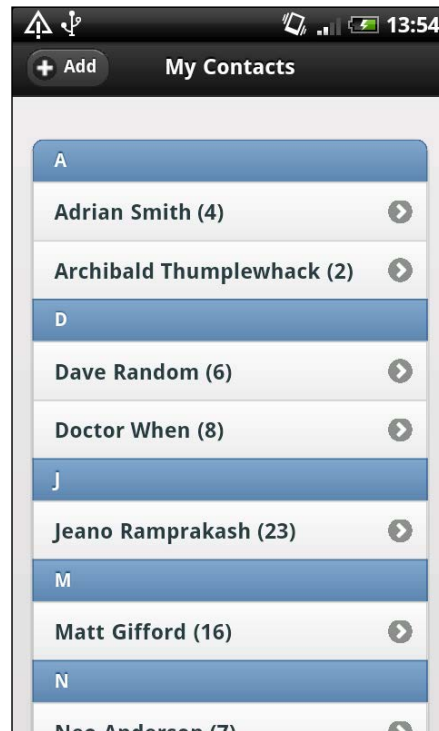
How it works...

When we create a new `Contact` object, we are simply creating a local variable populated with the provided information from the form submission. At this point, we could still add and amend properties within the `Contact` object, or simply fail to persist the object by not saving it.

To successfully save and store the contact within the device contact database, we then call the `save()` method available from the `Contact` object to do this.

The following methods are available to use from a `Contact` object:

▶ `save()`: This method will save a new contact into the device contacts database. If the `Contact` object has an `id` attribute that already matches that of a saved contact, it will update the saved contact with any revised information.

- ▶ `remove()`: Calling this method will remove the specified contact from the device contacts database.
- ▶ `clone()`: This method will create a deep copy of the provided `Contact` object, however, the `id` property will be set to `null`. This means that you can easily duplicate contact information and save as a new `Contact` object.

There's more...

When we saved the e-mail address and phone number to the new `Contact` object, we used the `ContactField` object to do so, instead of sending in the values as part of the contact information object. The `ContactField` object is provided to support generic fields within a `Contact` object, such as e-mail address, phone numbers, and URLs.

The `Contact` object itself stores values such as these in an array, which can contain multiple `ContactField` objects – for example, a contact can have more than one phone number assigned to them.

The `ContactField` object requires the following properties:

- ▶ `type`: A `DOMString` object that identifies which type of field this is. For example, a phone number type value can include "home", "work", "mobile", or any other value supported by the database on a particular device platform.
- ▶ `value`: A `DOMString` object that holds the value of the field itself – for example, the phone number or e-mail address.
- ▶ `pref`: A `Boolean` value that if set to true will set this specific field as the preferred value for the `ContactField` type.

Go a little further...

In our example application in this recipe, we stored only one phone number and one e-mail address per contact. We created a new array for each value in which to hold this information, but only set the first index in each array with a `ContactField` object.

Why not expand on this recipe to provide the user with a frontend UI to allow for more form fields to provide extra phone numbers, and then amend the code to create new `ContactField` objects for each new value submitted.

5
Hook into Native Events

In this chapter, we will cover:

- ▶ Pausing your application
- ▶ Resuming your application
- ▶ Displaying the status of the device battery levels
- ▶ Making use of the native search button
- ▶ Displaying network connection status
- ▶ Creating a custom submenu

Introduction

When developing for mobile devices, we can create feature-rich applications that harness the functionality of the native processes and systems.

The devices themselves provide us with built-in controls and user interface elements in the form of native buttons, to which we can apply methods and functions.

We can also make use of the hidden events and manage how our applications work when placed in the background on the device or alter states depending on network connectivity.

The recipes in this chapter will introduce you to some of the native events available through the PhoneGap API, and how we can implement them into applications.

Pausing your application

Although we want our users to spend their time solely on our applications, they will inevitably leave our application to open another one or do something else entirely. We need to be able to detect when a user has left our application but not closed it down entirely.

How to do it...

We can use the PhoneGap API to fire off a particular event when our application is put into the background on the device:

1. Create the initial HTML layout for the application, and include the reference to the `Cordova JavaScript` file in the `head` tag of the document.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html;>
    <title>Pausing an application</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>

  </head>
  <body>

  </body>
</html>
```

2. Before the closing `head` tag, create a new `script` tag block and add the event listener to check when the device is ready and the PhoneGap code is ready to run.

```
<script type="text/javascript">

  document.addEventListener("deviceready",
    onDeviceReady, false);

</script>
```

3. Create the `onDeviceReady` function, which will run when the event listener is fired. Inside this, we'll create a new event listener that will check for a `pause` event, and once received will fire the `onPause` method.

```
function onDeviceReady() {
  document.addEventListener("pause", onPause, false);
}
```

4. Let's create the `onPause` method. In this example application, we'll ask the device to notify the user that the application has moved into the background by playing an audio beep. The numeric parameter specifies how many times we want the audio notification to be played – in this case, just once.

```
function onPause() {
  navigator.notification.beep(1);
}
```

> Developing for iOS? There is no native beep API for iOS. The PhoneGap API will play an audio file using the media API, but the developer must provide the file, named `beep.wav` and under 30 seconds in length, in the `/www` directory of the application project files. iOS will also ignore the beep count argument and will play the audio once. If developing for Windows 7 mobile, the WP7 Cordova library contains a generic beep audio file that will be used.

5. When we run the application on the device, if you press the home button or navigate to another application, the device will play the notification audio.

How it works...

To correctly determine the flow of our lifecycle events, we first set up the `deviceready` event listener to ensure that the native code was properly loaded. At this point, we were then able to set the new event listener for the `pause` event.

As soon as the user navigated away from our application, the native code would set it into the background processes on the device and fire the `pause` event, at which point our listener would run the `onPause` method.

> To find out more about the `pause` event, please refer to the official documentation, available here:
> `http://docs.phonegap.com/en/2.0.0/cordova_events_events.md.html#pause`.

There's more...

In this recipe we applied the `pause` event in an incredibly simple manner. There is a possibility your application will want to do something specific other than sending an audio notification when the user pauses your application.

For example, you may want to save and persist any data currently in the view or in memory, such as any draft work (if dealing with form inputs) or saving responses from a remote API call.

We'll build an example that will persist data in the next recipe, as we'll be able to quantify its success when we resume the use of the application and bring it back into the foreground.

Resuming your application

Multi-tasking capabilities that are now available on mobile devices specify that the user has the ability to switch from one application to another at any time. We need to handle this possibility and ensure that we can save and restore any processes and data when the user returns to our application.

How to do it...

We can use the PhoneGap API to detect when our application is brought back into the foreground on the device. The following steps will help us to do so:

1. Create the initial layout for the HTML and include the JavaScript references to the Cordova and the `xui.js` files. We will also be setting the `deviceready` listener once the DOM has fully loaded, so let's apply an `onload` attribute to the `body` tag.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>Resuming an application</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
 <script type="text/javascript"
    src="xui.js"></script>

</head>
<body onload="onLoad()">

</body>
</html>
```

2. Create a new `script` tag block before the closing `head` tag and add the `deviceready` event listener within the `onLoad` method. We'll also set two global variables, `savedTime`, and `localStorage`, the latter of which will reference the `localStorage` API on the device:

```
<script type="text/javascript">

  var savedTime;
var localStorage = window.localStorage;

function onLoad() {

    document.addEventListener("deviceready",
        onDeviceReady, false);
}



</script>
```

3. Create the `onDeviceReady` function, within which we'll set the two event listeners to check for the `pause` and `resume` events, as follows:

```
function onDeviceReady() {
  document.addEventListener("pause", onPause, false);
  document.addEventListener("resume", onResume, false);
}
```

4. We can now add the first of the new callback functions for the added listeners. `onPause` will run when a `pause` event has been detected. In this method, we'll create a new date variable holding the current time, and store it into the global `savedTime` variable we created earlier.

5. If the user has entered something in to the text input field, we'll also take the value and set it into the `localStorage` API, before clearing out the input field.

```
function onPause() {
  savedTime = new Date();
  var strInput = x$('#userInput').attr('value');
  if(strInput) {
    localStorage.setItem('saved_input', strInput);
      x$('#userInput').attr('value', '');
  }
}
```

6. Define the `onResume` method, which will run when a `resume` event has been detected. In this function, we'll save a new date variable and we'll use it in conjunction with the `savedTime` variable created in the `onPause` method to generate the time difference between the two dates. We'll then create a string message to display the time details to the user.

7. We'll then check the `localStorage` for the existence of an item stored using the key `saved_input`. If this exists, we'll extend the message string and append the saved user input value before setting the message into the DOM to display.

```
function onResume() {
  var currentTime = new Date();
  var dateDiff = currentTime.getTime() - savedTime.getTime();
  var objDiff = new Object();
    objDiff.days = Math.floor(dateDiff/1000/60/60/24);
    dateDiff -= objDiff.days*1000*60*60*24;
    objDiff.hours = Math.floor(dateDiff/1000/60/60);
    dateDiff -= objDiff.hours*1000*60*60;
    objDiff.minutes = Math.floor(dateDiff/1000/60);
    dateDiff -= objDiff.minutes*1000*60;
    objDiff.seconds = Math.floor(dateDiff/1000);

  var strMessage = '<h2>You are back!</h2>'
    strMessage += '<p>You left me in the background for '
    strMessage += '<b>' + objDiff.days + '</b> days, '
    strMessage += '<b>' + objDiff.hours + '</b> hours, '
    strMessage += '<b>' + objDiff.minutes + '</b> minutes, '
    strMessage += '<b>' + objDiff.seconds + '</b> seconds.</p>';

  if(localStorage.getItem('saved_input')) {
    strMessage = strMessage + '<p>You had typed the following
before you left:<br /><br />'
    strMessage += '"<b>' + localStorage.getItem('saved_input') +
'</b>"</p>';
  }

  x$('#message').html(strMessage);

}
```

8. Finally, let's add the DOM elements to the application. Create a new `div` element with the `id` attribute set to `message`, and an `input` text element with the `id` set to `userInput`.

```
<body onload="onLoad()">
  <div id="message"></div>
  <input type="text" id="userInput" />

</body>
```

9. When we run the application on the device, the initial output would provide the user with an input box to enter text, should they wish to, as shown in the following screenshot:
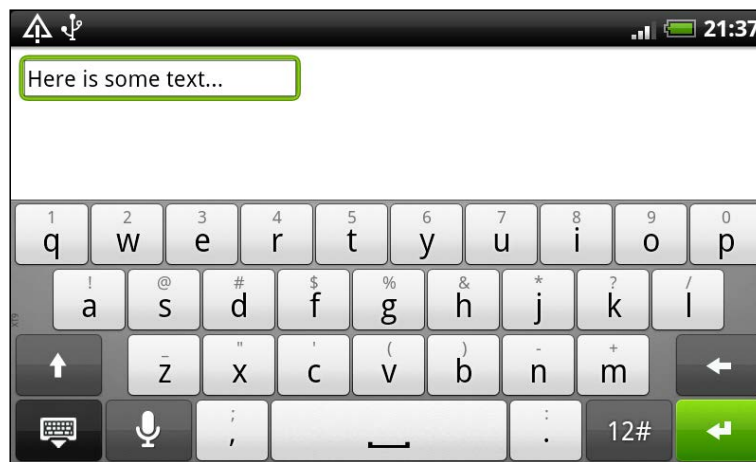
10. If we were to pause the application and then resume it after a period of time, the display would then update to look something like the following screenshot:

137
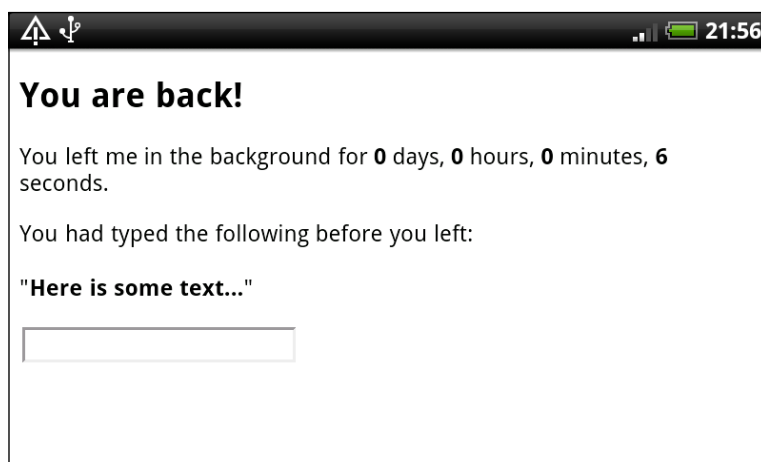
How it works...

We set up the `deviceready` event listener after the DOM was fully loaded, which would then run the `onDeviceReady` function. Within this method we then added two new event listeners to catch the `pause` and `resume` events respectively.

When the application is paused and placed into the background processes on the device, we saved the current date and time into a global variable. We also checked for the existence of any user-supplied input and if it was present we saved it using the `localStorage` capabilities on the device.

When the application was resumed and placed back into the foreground on the device, the `onResume` method was run, which obtained the time difference between the saved and current `datetime` values to output to the user. We also retrieved the saved user input from the `localStorage` if we had set it within the `onPause` method.
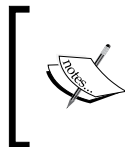
> To find out more about the `resume` event, please refer to the official documentation, available here:
> `http://docs.phonegap.com/en/2.0.0/cordova_events_events.md.html#resume`.

See also

▸ The *Caching content using the web storage local storage API* recipe in *Chapter 2*, *File System, Storage, and Local Databases*

Displaying the status of the device battery levels

Progressions in capabilities and processing power means we can do much more with our mobile devices including multi-tasking and background processes, this often means we end up using more battery power to fuel our applications.

How to do it...

In this recipe we will build an application to display the connection details and current power capacity of the device battery.

1. Create the initial HTML layout for the application and include the `Cordova JavaScript` file. We'll also be manipulating DOM elements, so include a reference to the `xui.js` file within the `head` tag.

2. We will be calling the `onDeviceReady` method to instantiate the PhoneGap functionality through an `onLoad()` function attached to the `body` tag.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
         initial-scale=1, maximum-scale=1,
         minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>Battery State</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript"
    src="xui.js"></script>

  </head>
  <body onload="onLoad()">

  </body>
</html>
```

3. Let's add the UI elements for the application. This will include a `div` element with the `id` attribute set to `statusMessage`, which will hold our returned information. We'll also build up some nested elements to create a visual representation of the device battery.

4. We will reference the `id` attributes of each element using the `XUI` library, so we need to make sure the three attribute values are set to `batteryIndicator`, `batteryLevel`, and `shade` respectively:

```
<h3>Battery Status</h3>

<div id="statusMessage"></div>

<div id="batteryIndicator">
<div id="batteryLevel">
    <div id="shade" />
</div>
</div>
```

5. With the DOM elements inserted, let's continue to the JavaScript code. Create a `script` tag block before the closing `head` tag, and add an `onLoad` method, run from the `body` tag, which will add the event listener to check that the native PhoneGap code has been loaded and is ready for use.

```
<script type="text/javascript">

  function onLoad() {
    document.addEventListener("deviceready",
      onDeviceReady, false);
  }



</script>
```

6. Add the `onDeviceReady` method, into which we will add three new event listeners that will respond to changes with the device's battery status. Each listener has a corresponding callback method, which we will define in the next few steps.

```
function onDeviceReady() {
  window.addEventListener("batterystatus",
  onBatteryStatus, false);
  window.addEventListener("batterylow", onBatteryLow, false);
  window.addEventListener("batterycritical",
  onBatteryCritical, false);
}
```

7. The first callback method is `onBatteryStatus`, which accepts the information object that contains properties on the device's battery. We will pass this information to a new function, `setBatteryInfo`.

```
function onBatteryStatus(battery_info) {
  setBatteryInfo(battery_info);
}
```

8. Let's write the `setBatteryInfo` method, called from the status change function. We can use the level property returned from the `battery_info` object to set the width of the `batteryLevel` element. We'll then create a message with the current capacity level and whether or not the device is plugged in, which we'll set into the `statusMessage` element.

9. If the battery level is below 21 percent, we'll change the background color of the battery to red, otherwise we'll set it at a healthy green.

```
function setBatteryInfo(battery_info) {
  x$('#batteryLevel').setStyle('width',
```

```
      battery_info.level + '%');
   var statusMessage = '<p>Percent: <span id="level">' +
        battery_info.level + '%</span></p>';
     statusMessage = statusMessage + '<p>A/C: ' +
chargingStatus(battery_info.isPlugged) + '</p>';
x$('#statusMessage').html(statusMessage);
     if(battery_info.level <= 20) {
       x$('#level').addClass('warning');
  x$('#batteryLevel').setStyle('backgroundColor',
     '#E74A4A');
 } else {
   x$('#batteryLevel').setStyle('backgroundColor',
       '#01A206');
 }
}
```

10. The return value of the `isPlugged` method from the `battery_info` object is a `Boolean` value, so we'll send it to a new function to return a string representation of the connection, as shown in the following block of code:

```
function chargingStatus(isPlugged) {
  if(isPlugged) { return 'Connected'; }
  return 'Disconnected';
}
```

11. The `batterylow` event handler will run a method called `onBatteryLow`. Inside of this, we'll include a notification alert to inform the user. This is shown in the following code block:

```
function onBatteryLow(battery_info) {
  navigator.notification.alert(
    'Time to charge it up!',
    function() {}, //alert dismissed
    'Low Battery',
    'OK'
  );
}
```

12. If the battery reaches critical levels, the `onBatteryCritical` callback method will run. Again, let's use this event to alert the user of their urgent need to charge the device.

```
function onBatteryCritical(battery_info) {
  navigator.notification.alert(
    'Seriously, plug your charger in!',
```

```
      function() {}, //alert dismissed
      'Critical Battery',
      'OK'
   );
}
```

13. Both of our notification alerts will execute an empty function specified as the `alertCallback` property in the previous snippets, which will run when the alert is dismissed. For this example, we don't need to perform any extra functionality at this point, hence the empty function.

14. Finally, include some CSS definitions to add a visual presence to our battery elements, as shown in the following code snippet:

```
<style>
#batteryIndicator {
  margin: 0 auto;
  width: 250px;
  height: 100px;
  border: 1px solid #ccc;
  background: #fff;
  border-radius: 10px;
  overflow: hidden;
}
#batteryLevel { height: 100%; }
#shade {
  width: 100%;
  height: 15px;
  background: -webkit-gradient(linear, 0% 0%, 0% 100%,
 from(#e5e5e5), to(#fff));
  opacity: 0.2;
  position: relative;
  top: 15px;
}
.warning { color: #ff0000; font-weight: bold; }
</style>
```

15. When we run the application on the device, assuming battery levels are within the healthy boundaries, the output will look similar to the following screenshot:
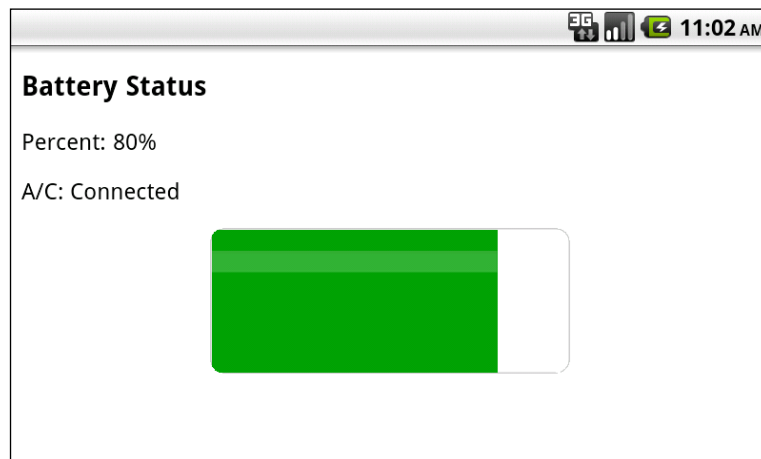
16. As soon as the device battery capacity goes below 21 percent, the UI will change to something similar to the following screenshot:
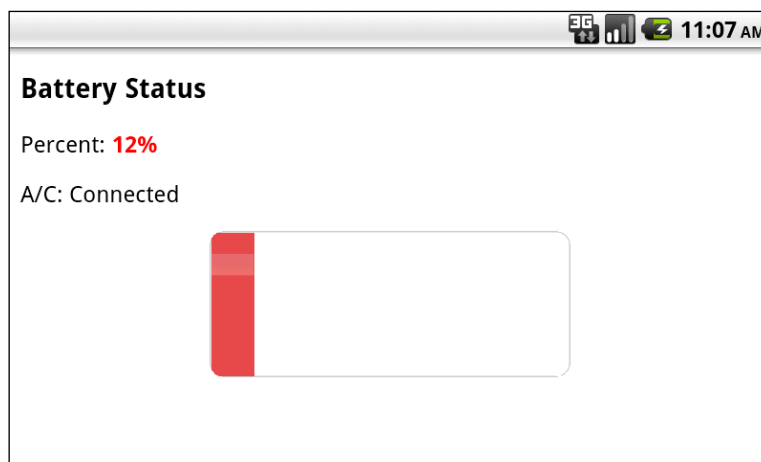
17. In the following screenshot, the device battery levels have hit the critical value of 5 percent. As a result, we notify the user with an alert message, as shown in the following screenshot:
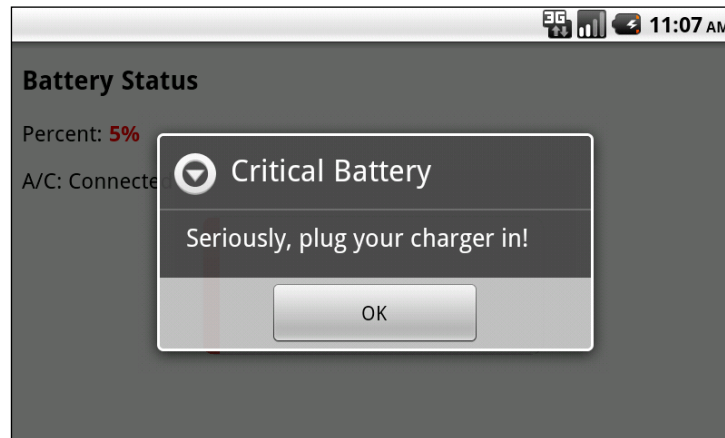
How it works...

Using the `onDeviceReady` method, we set up three new event listeners to check for the status of the device battery levels.

All three of the battery status handlers (`batterystatus`, `batterylow`, and `batterycritical`) return the same object with the following properties:

- ▸ `level`: A `Number` value that defines the percentage of the battery, between 0 and 100
- ▸ `isPlugged`: A `Boolean` value that returns a true or false value to represent if the device is connected to a charger or not

We were able to use the returned properties to update a visual representation of the device battery on screen, as well as use the level property to determine if we are below the critical threshold barrier.

Although at first glance the three events seem to do the same thing, there are important differences. The `batterystatus` event will detect changes in the battery capacity, and will fire its callback method with every percentage change. This allows us to keep a constant check on the status of the device battery levels. It will also fire if the device is connected or disconnected from the charger. From the `isPlugged` property, we can easily determine if the device is using the mains power or not.

The `batterylow` event will fire only when the battery has reached a specific percentage level, deemed as low by the device. The same applies to the `batterycritical` event, which will only fire once the battery level has reached a particular percentage.

The threshold levels for the `batterylow` and `batterycritical` events are specific to each device, so this is something to be aware of if you are hardcoding values within the application. As a reference, Android devices typically set the low threshold to 20 percent, and the critical threshold to 5 percent.
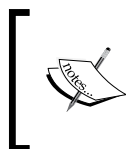
> To find out more about the `batterycritical`, `batterylow`, and `batterystatus` events, please refer to the official documentation, available here:
>
> `http://docs.phonegap.com/en/2.0.0/cordova_events_`
> `events.md.html#batterystatus`.

There's more...

In our sample application included in this recipe, we processed a simple alert notification when the low and critical thresholds were reached.

Depending on your mobile application and what its processes are, chances are you will want to action something specific at these points. For example, you may want to save any user input values into local memory, or shutdown/pause certain aspects of the application's functionality once these thresholds have been detected. If the user is unable to charge their device, you do not want them to lose data while using your application.

Making use of the native search button

The native functionality of a mobile device search button can be overridden using the PhoneGap API, which allows developers to create custom search commands for their applications, or use the button for something else entirely different from search operations.

How to do it...

In this recipe we will create a small application that will accept user input and transfer the query, opening the device's native browser to query a search engine.

1. Create the initial layout for the HTML, and include references to both the Cordova and XUI JavaScript libraries.

2. We will also apply an `onload` attribute to the `body` tag to run a method once the DOM has loaded.

   ```
   <!DOCTYPE HTML>
   <html>
     <head>
       <meta name="viewport" content="user-scalable=no,
   ```

145

```
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type" content="text/html;
    charset=utf-8">
    <title>Search Button</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript"
    src="xui.js"></script>

</head>
<body onload="onLoad()">



</body>
</html>
```

3. The user interface for this application is incredibly simple. Add a text input field within the `body` of the document, and set the `id` attribute of the element to `criteria`. We will reference this directly using XUI, as shown in the following code snippet:

```
<body onload="onLoad()">

  <h2>Goog Seeker</h2>

  <input type="text" id="criteria" />

</body>
```

4. Let's now add our custom code. Create a new `script` tag block before the closing `head` tag, into which we'll define the `onLoad` method. This will add the event listener to fire once the PhoneGap code is ready.

```
<script type="text/javascript">

  function onLoad() {
    document.addEventListener("deviceready",
    onDeviceReady, false);
  }



</script>
```

5. The `onDeviceReady` will then add a new event listener to detect the use of the device's search button.

```
function onDeviceReady() {
  document.addEventListener("searchbutton",
  onSearchPress, false);
}
```

6. The event listener will execute the `onSearchPress` method. This will obtain the value of the user-supplied input and append it to a URL string. We can then load the URL in the device browser.

```
function onSearchPress() {
  var userInput = x$('#criteria').attr('value');
  if(userInput) {
    var urlString = 'http://www.google.co.uk#q=' +
escape(userInput);
    navigator.app.loadUrl(urlString);
  }
}
```

7. When we run the application on a device, the initial page layout will look something like the following screenshot:
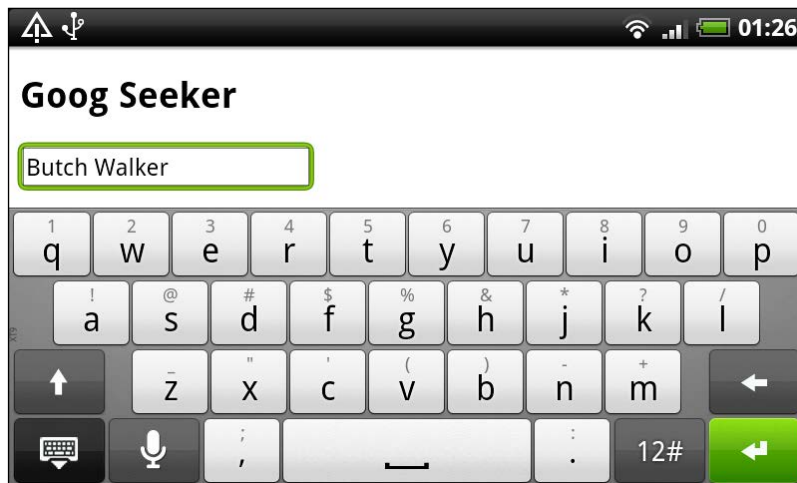
8. Once the user presses the search button on the device, the browser will open and run the search query, as shown in the following screenshot:
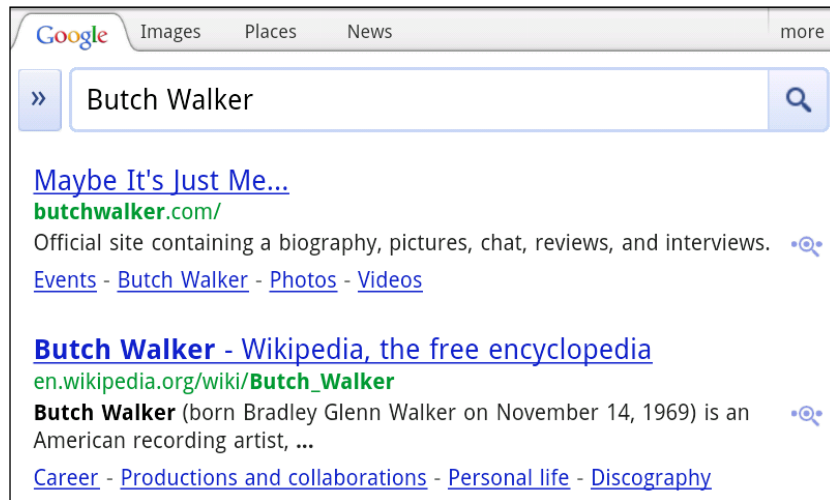
How it works...

The `onDeviceReady` method sets up the new event listener, which will fire when it detects the `searchbutton` event.

We then obtained the value from the text input field, appended it to the URL string, and loaded that URL in the native device browser.
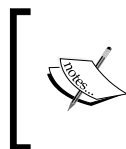
> To find out more about the `searchbutton` event, please refer to the official documentation, available here:
>
> http://docs.phonegap.com/en/2.0.0/cordova_events_
> events.md.html#searchbutton.

There's more...

The `searchbutton` event is only applicable to Android devices, but with such a prominent position on most Android phones, it really is a very useful button and event that can be used for many purposes.

Consider an application that lists contacts from the device database. Pressing the search button, you could open a dialog window to allow the user to enter search criteria to filter the contacts. We delved into the contacts database in *Chapter 4*, *Working with your contacts*. If you're feeling up to it, why not amend one of the example applications included in that chapter to add this search feature.

148

Displaying network connection status

Your application may require the user to be connected to a network. This may be for partial updates, remote data transfer, or streaming. Using the PhoneGap API, we can easily detect the status or existence of any network connectivity.

How to do it...

In this recipe, we will build an application to constantly check the network connection status of our device.

1. Create the initial HTML layout for the application. Include references to the Cordova and XUI JavaScript libraries within the `head` tag of the document.

2. We will also be setting the `deviceready` event listener after the DOM has fully loaded, so we'll also add the `onLoad()` function call to the `body` tag.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type" content="text/html;
    charset=utf-8">
    <title>Network Status</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript"
    src="xui.js"></script>

</head>
<body onload="onLoad()">


</body>
</html>
```

3. Let's add the UI elements to the body of our application. Create a `div` block to act as a container for our `statusMessage` and `count` elements, both of which we will be referencing directly using the XUI library. We will also be inserting content into the `speedMessage` element, so ensure the `id` attribute of those three elements matches those shown as follows:

```
<h3>Network Status</h3>
  <div id="holder">
```

```
    <div id="statusMessage"></div>

    <div id="count"></div>

  </div>

<div id="speedMessage">

</div>
```

4. Create a new `script` tag block before the closing `head` tag and define two global variables, which we will use within the custom code. We can also now define the `onLoad` method, which will set the `deviceready` event listener.

```
<script type="text/javascript">

  var intCheck = 0;
  var currentType;

  function onLoad() {
    document.addEventListener("deviceready",
    onDeviceReady, false);
  }



</script>
```

5. Let's now add the `onDeviceready` method, called from the `deviceready` event listener. Within this function we will add two new event listeners to check when the device is connected or disconnected from a network. Both of these listeners will run the same callback method, that is `checkConnection`.

6. We will then set up an interval timer to run the same `checkConnection` method every second to provide us with constant updates for the connection.

```
function onDeviceReady() {
  document.addEventListener("online", checkConnection,
false);
  document.addEventListener("offline", checkConnection,
false);
  var connCheck = setInterval(function() {
    checkConnection();
  }, 1000);
}
```

7. The `checkConnection` function sets up the `objConnection` variable to hold a representation of the device's connection. This object returns a value in the `type` property, from which we are able to determine the current connection type. We'll pass that value into another function called `getConnectionType`, which we'll use to return a user-friendly string representation of the connection type.

8. As this method runs every second, we want to be able to determine if the current connection type differs from the previous connection. We can do this by storing the connection type value in the `currentType` global variable and check if it matches the current value.

9. Depending on the returned value of the connection type, we can optionally choose to inform the user that to get the most out of our application they should have a better connection.

10. We will also increment an integer value, stored in the `intCheck` global variable, which we will use to count the number of seconds the current connection has been active for.

```
function checkConnection() {
  var objConnection = navigator.network.connection;
  var connectionInfo = getConnectionType(objConnection.type);
  var statusMessage = '<p>' + connectionInfo.message + '</p>';

  if(currentType != objConnection.type) {
    intCheck = 0;
    currentType = objConnection.type;

    if(connectionInfo.value <= 3) {
      x$('#speedMessage').html('<p>This application
    works better over a faster connection.</p>');
    } else {
      x$('#speedMessage').html('');
    }
  }
  intCheck = ++intCheck;

  x$('#statusMessage').html(statusMessage);
  x$('#count').html('<p>Checked ' + intCheck +
' seconds ago</p>');
}
```

151

11. The `getConnectionType` method mentioned previously will return a `message` and `value` property depending on the `type` value sent as the parameter. The `value` properties have been assigned manually to allow us to control what level of connection we deem best for our application and for the experience of our users.

```
function getConnectionType(type) {
  var connTypes = {};
      connTypes[Connection.NONE]     = {
        message: 'No network connection',
      value: 0
      };
    connTypes[Connection.UNKNOWN]     = {
      message: 'Unknown connection',
      value: 1
    };
    connTypes[Connection.ETHERNET]      = {
      message: 'Ethernet connection',
      value: 2
    };
    connTypes[Connection.CELL_2G]     = {
      message: 'Cell 2G connection',
      value: 3
    };
    connTypes[Connection.CELL_3G]     = {
      message: 'Cell 3G connection',
      value: 4
    };
    connTypes[Connection.CELL_4G]     = {
      message: 'Cell 4G connection',
      value: 5
    };
    connTypes[Connection.WIFI]     = {
      message: 'WiFi connection',
      value: 6
    };
  return connTypes[type];
}
```

12. Finally, let's add some CSS definitions to the bottom of our application to add some style to the UI.

```
<style>
div#holder {
```

```
    width: 250px;
    min-height: 60px;
    margin: 0 auto;
    position: relative;
    border: 1px solid #ff0080;
    border-radius: 10px;
    background: #ff0080;
  }
  div#holder p {
    margin: 20px auto;
    text-align: center;
    color: #ffffff;
    font-weight: bold;
  }
  div#speedMessage {
    width: 250px;
    margin: 0 auto;
    position: relative;
  }
</style>
```

13. When we run the application on our device, the output will look something like the following screenshot:
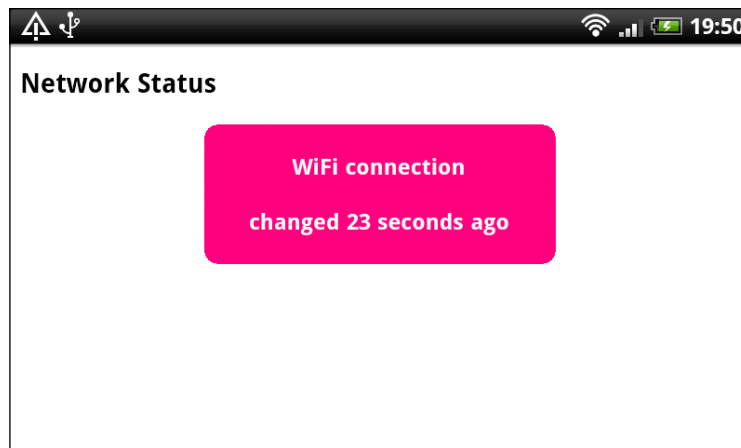
14. If our user changes their connection method or disconnects completely, the interval timer will detect the change and update the interface accordingly, and the timer will restart. This is depicted in the following screenshot:
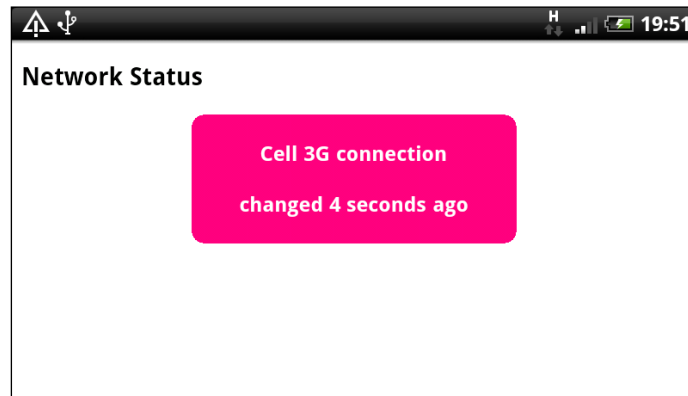
How it works...

We set up the `onDeviceReady` method to create two new event listeners to check for the `online` and `offline` events respectively. The `online` event will fire when the device's network connection is started, and the `offline` event will fire when the network connection is turned off or lost.

These events will only fire once, and so in this recipe we added in the `setInterval` timer function to constantly call the `checkConnection` method to allow us to obtain changes made to the network. The addition of this functionality helps greatly and means we can tell when a user switches from a 3G to a WiFi connection, for example. If this happens, they would not go offline, but simply change the connection type.
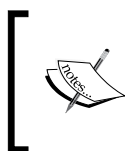
> To find out more about the `online` and `offline` events, please refer to the official documentation, available here:
>
> http://docs.phonegap.com/en/2.0.0/cordova_events_events.md.html#online.

There's more...

Your application may involve streaming data, remote connections or another process that requires a certain level of connectivity to a network. By constantly checking the status and `type` of connection, we can determine if it falls below an optimal level or a recommended type for your application. At this point, you could inform the user, or restrict access to certain remote calls or data streams to avoid latency in your application's response and possible extra financial costs incurred to the user from their mobile provider.

Creating a custom submenu

Your application may include an option for users to update or change settings, or perhaps the ability to truly exit the application gracefully, closing down all services and storing state or data.

How to do it...

In this recipe, we will create a simple application that interacts with the device's native menu button to create a sliding submenu:

1. Create the initial layout of the HTML for our application. Include the Cordova and XUI JavaScript library references in the `head` of the document, and include an `onLoad` method call within the `body` tag, which will set the `deviceready` event listener once the DOM is fully loaded.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type" content="text/html;
    charset=utf-8">
    <title>Sub Menu</title>
    <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript"
    src="xui.js"></script>

</head>
<body onload="onLoad()">

</body>
</html>
```

2. Let's now add the UI to the `body` of the document. Create a new `button` element with the `id` attribute set to `menuToggle`, and an unordered list within a `div` element. In this recipe, each `anchor` tag has a specific `id` attribute, which we'll use shortly to assign touch handlers to each link.

```html
<h2>Menu Display</h2>

  <button id="menuToggle">Toggle Menu</button>
```

155

```
<div id="subMenu">
  <a id="closeMenu"><span>Close Menu</span></a>
  <a id="hello"><span>Hello</span></a>
  <a id="exit"><span>Exit Application</span></a>
</div>
```

3. Add a new `script` tag block before the closing `head` tag in the document, and include the `onLoad` function which will add the `deviceready` event listener, as follows:

```
<script type="text/javascript">

function onLoad() {
  document.addEventListener("deviceready",
  onDeviceReady, false);
}



</script>
```

4. Create the `onDeviceReady` method, within which we will set a new event listener to check for the `menubutton` event. This will run the `onMenuPress` function once detected.

5. We'll also include the `setMenuHandlers` method, which will apply the touch handlers to the menu items.

```
function onDeviceReady() {
  document.addEventListener("menubutton", onMenuPress, false);
    setMenuHandlers();
}
```

6. The `onMenuPress` function, which is the callback method from the event listener, will handle the transition of our menu element and links. We will use XUI library to determine the current value of the `subMenu` element position and react accordingly to either open or close the menu.

```
function onMenuPress() {
  var menuPosition = x$('#subMenu').getStyle('bottom');
  if(menuPosition == '-100px') {
    x$('#subMenu').tween({bottom: '0px' });
  } else {
    x$('#subMenu').tween({bottom: '-100px' });
  }
}
```

7. The `setMenuHandlers` method will apply the touch handlers to our individual menu items. We can reference each element by `id` attribute and set the listener with the specific action we want it to run. To exit the application, we can call the `exitApp` method to gracefully close our application and not leave it running in the background on the device.

8. The `menuToggle button` element and the `closeMenu` link item both provide the user with the ability to close the menu themselves by calling the previously created `onMenuPress` method.

```
function setMenuHandlers() {
  x$('#exit').on('touchstart', function() {
    navigator.app.exitApp();
  });
  x$('#hello').on('touchend', function() {
    alert('Hello!');
  });
  x$('#closeMenu').on('touchend', function() {
    onMenuPress();
  });
  x$('#menuToggle').on('touchend', function() {
    onMenuPress();
  });
}
```

9. Finally, we will include some CSS definitions to set the menu into position and apply colour styles and required attributes:

```
<style>
#menuToggle {
  width: 100%;
  height: 40px;
  position: relative;
  margin: 0 auto;
  color: #ffffff;
  background: #ff0080;
  font-size: 20px;
}
#subMenu {
  position: fixed;
  bottom: -100px;
  left: 0px;
  border-top: 1px solid #555;
  height: 100px;
  width: 100%;
  background: #e5e5e5;
}
```

```
#subMenu span {
  position: relative;
  margin: 0 auto;
  top: 40%;
}
#subMenu a {
  width: 29%;
  height: 100px;
  display: block;
  float: left;
  margin: 0 2% 0 2%;
  border-left: 1px solid #ccc;
  border-right: 1px solid #ccc;
  text-align: center;
}
</style>
```

10. When we run the application on the device, the initial view will look like the following screenshot:
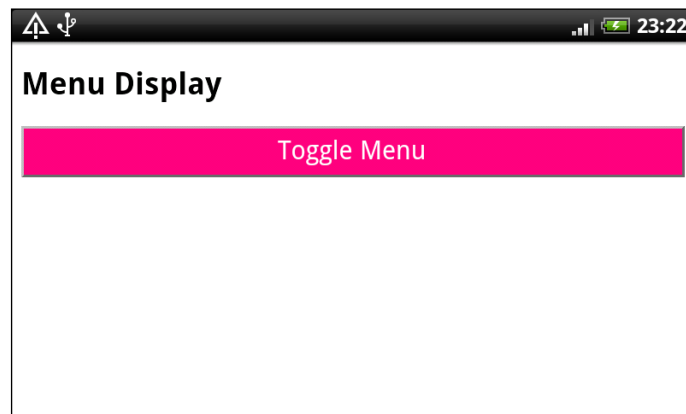
11. And with the menu open, the user will be presented with our links, as shown in the following screenshot:
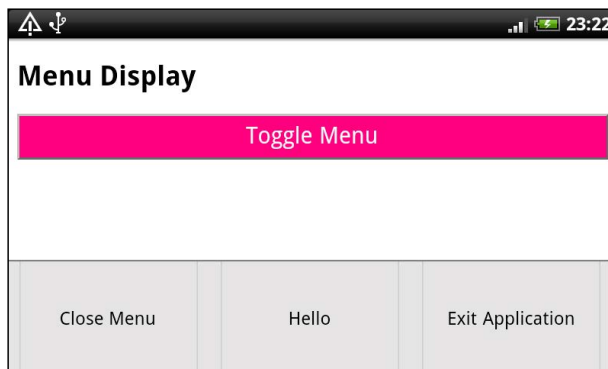
How it works...

The `onDeviceReady` method set up the new event listener to listen for the `menubutton` event. At this point, the `onMenuPress` function is run, which either opens or closes the menu depending on the current position of the `subMenu` element.

This is an ideal way to incorporate menu options and hidden gems of functionality within your application without overcrowding your user interface

To create the transition of the menu, we used the tween capabilities provided by the XUI JavaScript library. We'll cover the library in more detail in the next chapter.
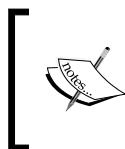
> To find out more about the `menubutton` event, please refer to the official documentation, available here: `http://docs.phonegap.com/en/2.0.0/cordova_events_events.md.html#menubutton`.

There's more...

The `menubutton` event provided by the PhoneGap API is not cross-device or cross-platform compatible. At present the supported device platforms are Android and BlackBerry WebWorks (OS 5.0 or higher versions).

There are other ways to include custom menus in your applications thanks to one of the many PhoneGap plugins created by the community developers and users.

The Native Menu plugin (`https://github.com/mwbrooks/cordova-plugin-menu`) allows you to add native toolbars, tab bars, and menus to your application, and is supported on Android, BlackBerry WebWorks, and iOS platforms.

The community and open source nature of the PhoneGap API and the Cordova product means that developers can freely extend and enhance the functionality of their applications and dig a little deeper into native processes offered by devices by creating custom plugins.

See also

▸   *Chapter 8*, *Extending PhoneGap with plugins*

6
Working with XUI

In this chapter, we will cover:

- ▶ Learning the basics of the XUI library
- ▶ DOM manipulation
- ▶ Working with touch and gesture events
- ▶ Updating element styles
- ▶ Working with remote data and AJAX requests
- ▶ Animating an element

Introduction

There are a number of common, widely used JavaScript (JS) frameworks that web professionals use and implement, some of which translate very well into the mobile landscape, such as jQuery.

There are a number of considerations when selecting a JS framework to use in your mobile applications. One is the size of the library, which would inevitably add size to your final packaged application.

While having the full product you may be used to using on your web applications also in use within your mobile apps is serendipitous, options exist for smaller libraries that contain many of the features you need, such as CSS selectors, filtering, style detection, and AJAX requests using XmlHttpRequests.

In this chapter we will look at the XUI JavaScript library and how we can use it. XUI was written and maintained by core members of the PhoneGap development team specifically for use in mobile applications, and removes much of the unrequired tools and hidden engines that do not apply to the modern browsers available on mobile devices.

Built with mobile devices in mind, XUI is incredibly small and lightweight and works across all of the devices in the mobile landscape. Unlike some other mobile JavaScript libraries available, XUI does not enforce any page structure or layout. It simply works with the DOM created by the developer to manipulate the layout and work with content.

Getting ready

Before we can continue with the recipes in this chapter, we must download a copy of the XUI library.

How to do it...

Perform the following steps to get started with the recipes:

1. Visit `http://xuijs.com/downloads`.

2. There are currently three variations of the library available for download:

   - **webkit / firefox / opera**
   - **blackberry mobile**
   - **ie & ie mobile**
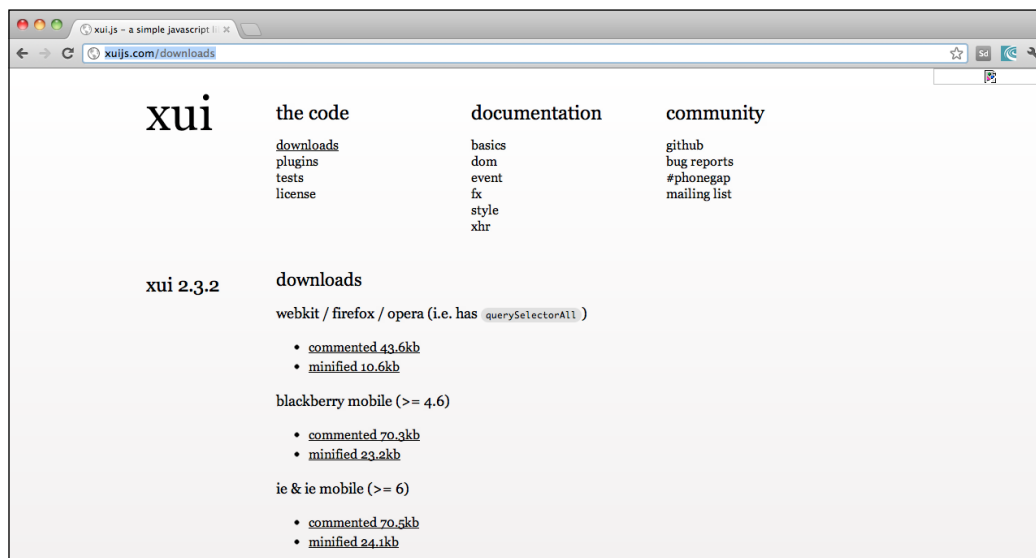
3. Select the version of the library you wish to implement into your application. Each option contains a full commented and a minified version. Feel free to download both options. The commented version may be an exciting read, but always remember to use the minified version on your final application.

We can now proceed with the recipes.

Learning the basics of the XUI library

When creating applications that focus as heavily on user interactions as mobile apps do, we want to be able to easily update and manage the underlying HTML and data collections.

How to do it...

We'll make use of XUI's simple but powerful DOM traversal methods and the ability to extend the library functionality into custom code.

1. Create the basic layout for the HTML page.

2. Create a `div` element within the page with some text. In this case we're going for the globally recognized "Hello World" sample text. Set the `id` attribute for this element to `content`.

3. Include a new `script` tag into the `head` of your document, and reference the XUI library within your project directory.

    ```
    <!DOCTYPE HTML>
    <html>
      <head>
        <meta name="viewport" content="user-scalable=no,
            initial-scale=1, maximum-scale=1,
            minimum-scale=1, width=device-width;" />
        <meta http-equiv="Content-type"
        content="text/html; charset=utf-8">
        <title>XUI</title>
        <script type="text/javascript"
        src="xui.js"></script>

    </head>
    <body>

      <div id="content">Hello World</div>

    </body>
    </html>
    ```

4. Before the closing `head` tag, include a new `script` tag block, inside of which we'll place the `onLoad` method, and an XUI `on` event to execute the method once the DOM has fully loaded, as shown in the following code snippet:

```
<script type="text/javascript">

  x$(window).on('load', onLoad);
  function onLoad() {


  }

</script>
```

5. Let's use this method to obtain the value of the content element on our page. We can access the XUI library and it's methods using the global `x$()` function.

   Create a new variable called `contentDiv`, and we'll pass the `id` of the `div` into the XUI global function to obtain the object reference, as shown in the following code snippet:

```
function onLoad() {

  var contentDiv = x$('#content');
    console.log(contentDiv);
    console.log(contentDiv.length);
    console.log(contentDiv[0].outerText);

}
```

6. If we run the page in a browser and open up the console view, we can see the returned data written to the console log, as shown in the following screenshot:
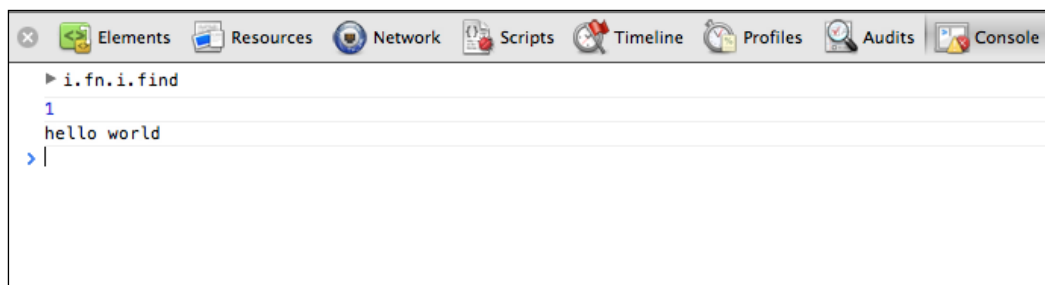
7. The first result is a large structure containing a lot of detailed information about the content `div` element, a sample of which is shown in the following screenshot:
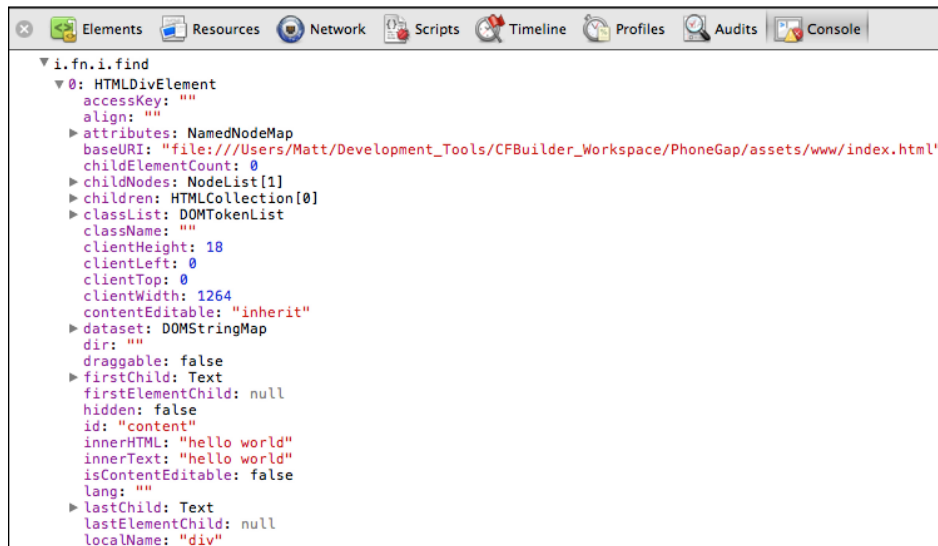
8. The level of detail about the returned object is quite extensive, and we can use some of this to programmatically obtain details about our selected elements.

9. Amend the `div` element to add some more attributes, such as a `style` and `class` attribute, as shown in the following code snippet:

```
<div id="content"
  style="border: 2px solid #e5e5e5;"
  class="sampleText">
  Hello World
</div>
```

10. Amend the `onLoad` method to include code to obtain the array node containing all attributes within the selected document element.

11. We'll use this array and loop over the contents to build an HTML string containing the `id` and `value` of each attribute within the `content div` element.

12. Finally, we'll display the generated string after the original `div` element on the page.

```
function onLoad() {

  var contentDiv  = x$('#content');
  var attributes = contentDiv[0].attributes;
```

```
if(attributes.length) {

  var attrMessage = '';

    attrMessage = '<h3>' + contentDiv[0].id + ' '
      + contentDiv[0].localName + ' has '
      + attributes.length + ' attributes:</h3>';

    attrMessage += '<ul>';

    for(i=0; i<attributes.length;i++) {
      attrMessage += '<li>' + attributes[i].localName
        + ' "' + attributes[i].value + '"</li>';
    }

    attrMessage += '</ul>';

    contentDiv.after(attrMessage);

  }

}
```

13. When we run the amended code in the browser, you will see something similar to that shown in the following screenshot:
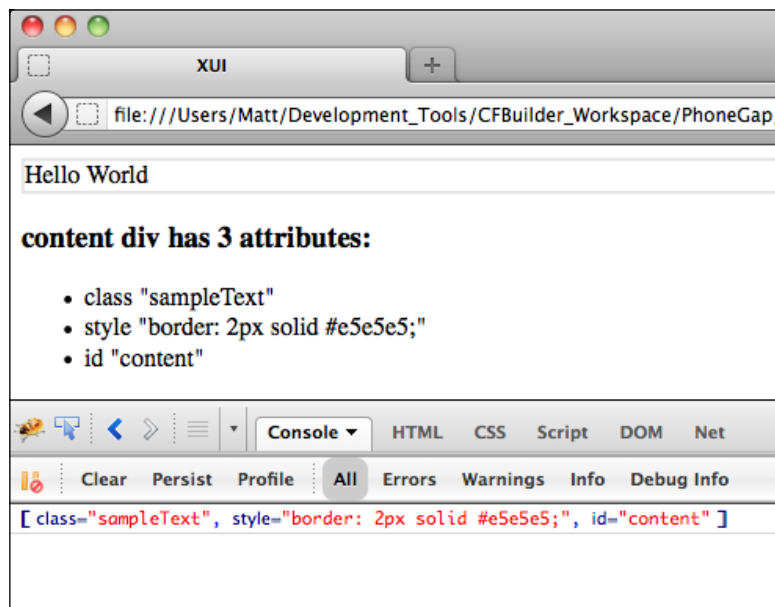
Let's now take a look at some of the other useful methods available with the XUI library, which also help to form a solid understanding of the basic functionality available:

1. Create a new HTML file, including a reference to the XUI JavaScript library, and an empty `script` tag block before the closing `head` tag.

2. Within the empty script block, define a new `onLoad` method.

3. Add an XUI `on` event handler to run the `onLoad` JavaScript method when the DOM is ready.

4. The `body` of the document will contain two unordered list elements, each with their own `id` attributes set to `family` and `friends` respectively. Each list item has a specific `class` attribute that relates to the gender of the individual within the list. Feel free to list your own family and friends.

5. Finally, we'll include a `div` element before the closing `body` tag with the `id` attribute set to `output`.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>XUI</title>
    <script type="text/javascript"
    src="xui.js"></script>
        <script type="text/javascript">

    x$(window).on('load', onLoad);

    function onLoad() {

    }

    </script>
</head>
<body>

  <ul id="family">
    <li class="male">Ted</li>
    <li class="female">Molly</li>
    <li class="female">Cate</li>
    <li class="female">Jean</li>
   <li class="male">George</li>
  </ul>
```

```
<ul id="friends">
  <li class="male">Steve</li>
  <li class="female">Pip</li>
  <li class="male">Dave</li>
  <li class="male">Scott</li>
</ul>

<div id="output"></div>

</body>
</html>
```

6. We'll add some code into the `onLoad` method to filter the various list elements. Firstly, create an empty `strMessage` variable into which we'll build our string for output.

7. We can access all of the names within the document list elements, which will provide us with an array of the results. We'll store that into the `allNames` variable.

```
function onLoad() {

  var strMessage = '';
  var allNames = x$('li');

  strMessage += '<p>There are ' + allNames.length
      + ' names in total</p>';


}
```

8. Let's now find the elements within specific lists. We can select all of the names within our family list by providing a specific element to look into – in this case we just want to select from the unordered list where the `id` attribute equals `family`.
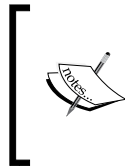
> Although in this instance we specifically called the `find()` method to locate the list items, the `x$` namespace is an alias for the `find()` method. As such, we didn't call it in the previous example to obtain all of the names in our lists, but we were performing the same function.

9. With can now dig a little deeper and find list items that match a given CSS selector. Here we will first look for all items within the `allFamily` array that first has the `male` class and then those with the `female` class. We assign both results to new array variables and add details to our output string, as shown in the following code snippet:

```
var allFamily = x$('#family').find('li');
var maleFamily = allFamily.has('.male');
var femaleFamily = allFamily.has('.female');
```

```
strMessage += '<p>' + allFamily.length
  + ' family members are listed:';
strMessage += '<ul><li id="maleFamily">' + maleFamily.length
  + ' male</li><li id="femaleFamily">' + femaleFamily.length
  + ' female</li></ul></p>';
```

10. Let's do the same to access all names within our `friends` list.

11. When accessing the class selectors to see if an element `has` a particular class, we can also check to see if an element does `not` have a matching CSS selector, as we are using here for our female friends.

```
var allFriends = x$('#friends').find('li');
var maleFriends = allFriends.has('.male');
var femaleFriends = allFriends.not('.male');

strMessage += '<p>' + allFriends.length
  + ' friends are listed:';

strMessage += '<ul><li id="maleFriends">' + maleFriends.length
  + ' male</li><li id="femaleFriends">' + femaleFriends.length
  + ' female</li></ul></p>';
```

12. Finally, we can set the generated string variable into the `div` element for display:

```
x$('#output').html(strMessage);
```

13. Another of XUI's built-in methods allows us to iterate over a collection of elements. Let's combine this into a generic function, which we will use to extend the native XUI library.

14. Create a new variable called `nameFunctions` which will be an object containing the various functions we wish to use to extend the library.

15. Name the first reference `generateList` and write the function code to iterate over each element of the provided array and set the resulting string variable into the parent element, as shown in the following code snippet:

```
var nameFunctions = {
  generateList: function(array) {
    var list = '<ul>';
    array.each(function(element, index, xui) {
      list+= element.outerHTML;
    });

    list+= '</ul>';

    this.bottom(list);
  }
}
```

16. Finally, revise the `onLoad` method and add four calls to the `getNameList` function, passing into it each array and list `id` attribute:

```
x$('#output').html(strMessage);

xui.extend(nameFunctions);

x$('#maleFamily').generateList(maleFamily);
x$('#femaleFamily').generateList(femaleFamily);
x$('#maleFriends').generateList(maleFriends);
x$('#femaleFriends').generateList(femaleFriends);
```

17. When we run the page in the browser, the output will look something like the following screenshot:
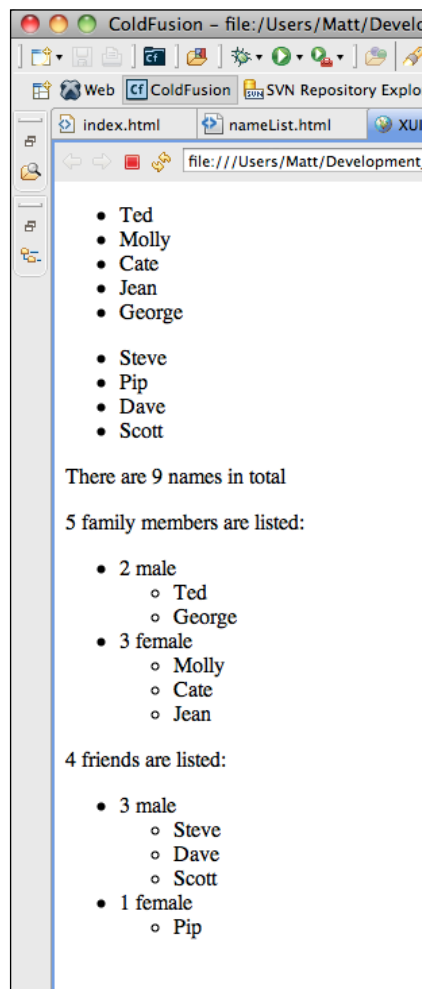
How it works...

In this recipe we used the find method available with XUI to locate specific elements within our document. We also made use of the `has` and `not` methods to access elements using the CSS selector.

Finally, we extended the functionality offered through XUI by setting our own function into the namespace for easy access to the elements and method calls.

> For more information on how to extend XUI and use the search or filter methods, make sure you check out the official documentation, available at: `http://xuijs.com/docs/basics`.

DOM manipulation

Although we may be building our applications in HTML, we still have the ability to alter and manipulate the elements within the document to create more of a dynamic application.

How to do it...

We will use the DOM manipulation methods available with the XUI library to read and write content directly within our application. The following steps will help us to do so:

1. Create a new HTML file, including a reference to the XUI JavaScript library, the `Cordova JavaScript` file, and an empty `script` tag block before the closing `head` tag.

2. Within the empty script block, define a new `onLoad` method.

3. Add an `on` event handler, which will run the `onLoad` JavaScript method when the DOM has fully loaded.

4. The `body` tag of the document will contain one `div` element with the `id` attribute set to content.

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
         initial-scale=1, maximum-scale=1,
         minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
   content="text/html; charset=utf-8">
    <title>XUI</title>
```

171

```
    <script type="text/javascript"
    src="xui.js"></script>
   <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
 <script type="text/javascript">

    x$(window).on('load', onLoad);

    function onLoad() {

 }


    </script>
</head>
<body onload="onLoad()">

  <div id="content"></div>

</body>
</html>
```

5. Before the `onLoad` method, let's define a new JavaScript object variable. This will act as a container to hold our various HTML strings. We'll access these from `button` click events.

```
var textContent = {
  top: '<div id="top">Top Context</div>',
  inner: '<p>Inner context</p>',
  bottom: '<div id="bottom">Bottom Context</div>',
  after: '<p>DOM Manipulation is easy with XUI!</p>'
};
```

6. Let's amend the `onLoad` method now to include an event listener to check that the device is ready, at which point the `onDeviceReady` method will fire.

```
function onLoad() {

document.addEventListener("deviceready",
    onDeviceReady, false);
}
```

7. Include the `onDeviceReady` function. This will make use of the DOM manipulation features and set a `h1` tag before the `content div` element.

8. We'll also set up an event listener to manage button clicks. This will take the `id` attribute of the `button`, select the relevant value from our `textContent` object and insert it into the relevant position in the DOM.

```
function onDeviceReady() {

  x$('#content').before('<h1>XUI DOM Manipulation</h2>');

  x$('button').on('click', function(e) {
    x$('#content').html(this.id, textContent[this.id]);
  });

}
```

9. Back into the HTML, add the following `button` elements at the top of the `body` content. Notice how each button has a specific `id` attribute to reference the object values, as shown in the following code snippet:

```
<button id="inner">Inner</button>
<button id="top">Top</button>
<button id="bottom">Bottom</button>
<button id="after">After</button>

<div id="content"></div>
```

10. Finally, let's add some basic styles to the document elements to easily see when they have been added.

```
<style>
  div#content { border: 2px solid #e5e5e5; }
  div#top { background: #ff0000; color: #fff; }
  div#bottom { background: #ffff00; color: #000; }
</style>
```

173

11. When we run the application on a device, the output should look something like the following screenshot:
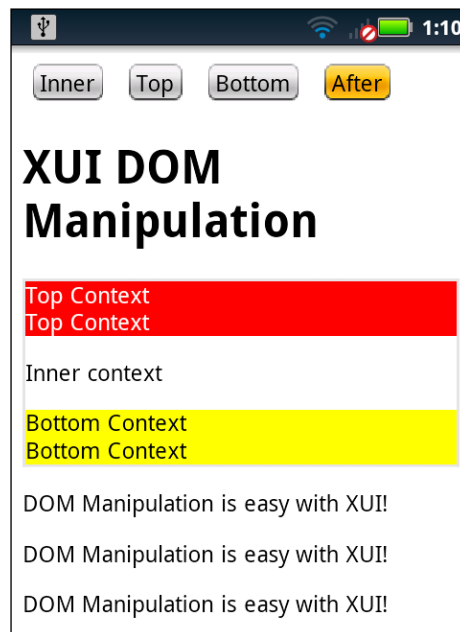
How it works...

We created a simple mobile application to see how the XUI library can manipulate the DOM. To set a value within an element we can use the `html()` function, appended to the XUI element collection.

The `html` function takes two arguments:

- ▸ `location`: A `String` value that determines the location surrounding the selected element where the DOM manipulation should take place.
- ▸ `html`: A `String` value that contains the HTML markup or elements to be placed into the DOM.

The location can take one of the following values: **inner**, **outer**, **top**, **bottom**, **remove**, **before,** or **after**.

In our button click handler function, we used the `html` method and passed in the location string, which we accessed from the `id` attribute of the clicked button.

You can also access the locations directly using the shorthand version and simply pass in the HTML element or markup, which we did for our `h1` tag entry.

For example:

```
x$('#content').before('<h1>XUI DOM Manipulation</h2>');
```

To access the current HTML within an element, we would simply need to call the methods without passing in an HTML value to set.

For example:

```
// Get the value of the content element
x$('#content').html();
```

> For more information on the DOM manipulation methods available, make sure you check out the official documentation: `http://xuijs.com/docs/dom`.

Working with touch and gesture events

When working with user interfaces that demand a high level of user interaction or certain processes to be run at certain moments, we need to start looking at using events and detecting them to execute methods. In this recipe we will create some functionality that will demonstrate not only setting an event, but removing it as well.

How to do it...

We will use the methods available in the XUI library to control the delegation of event handlers.

1. Create the HTML layout for your application. Include the XUI and Cordova JavaScript libraries within the `head` tag of your document.

2. Add an empty `script` tag block before the closing `head` tag. This will hold our custom PhoneGap code. Within this, define a new `onLoad` function.

3. Add an `on` event handler, which will run the `onLoad` JavaScript method when the DOM has fully loaded.

4. Finally, add a new `button` element within the `body` of the document. Set the `id` attribute to `touchme`.

    ```html
    <!DOCTYPE HTML>
    <html>
      <head>
        <meta name="viewport"
        content="width=screen.width; user-scalable=no" />
        <meta http-equiv="Content-type"
        content="text/html; charset=utf-8">
    ```

```
      <title>XUI</title>
      <script type="text/javascript"
      src="xui.js"></script>
   <script type="text/javascript"
      src="cordova-2.0.0.js"></script>
      <script type="text/javascript">

      x$(window).on('load', onLoad);

   function onLoad() {

      }


   </script>
</head>
<body>

   <button id="touchme">Touch gestures</button>

</body>
</html>
```

5. Let's add the first of our XUI events. We want to be really sure that the DOM has fully loaded before we manipulate it. The `ready()` method will run once this is the case, so place this within the `onLoad` method.

6. Inside of the event, we'll set up our event listener to execute the `onDeviceready` method once the PhoneGap code is ready to run.

```
function onLoad() {
  x$.ready(function() {
    console.log('The DOM is ready to go!');
    document.addEventListener("deviceready",
    onDeviceReady, false);
  });
}
```

7. Create the `onDeviceReady` function. This will register a new `touchstart` event to the `button` element.

```
function onDeviceReady() {
  x$('#touchme').on('touchstart', touchConfirmation);
}
```

8. The `touchConfirmation` function will run when the button event has been fired, that is, when it has been touched.

9. Here, we'll display a confirmation notification event from the PhoneGap API to give our users the choice to either keep touching our button or to leave it alone.

```
var touchConfirmation = function(){
  navigator.notification.confirm(
    'Do you want to apply a touch gesture again?',
    touchConfirmAction,
    'Touch gesture detected..',
    'Yes,No'
  )
};
```

The following screenshot shows the confirmation notification event triggered after the touch gesture is detected:
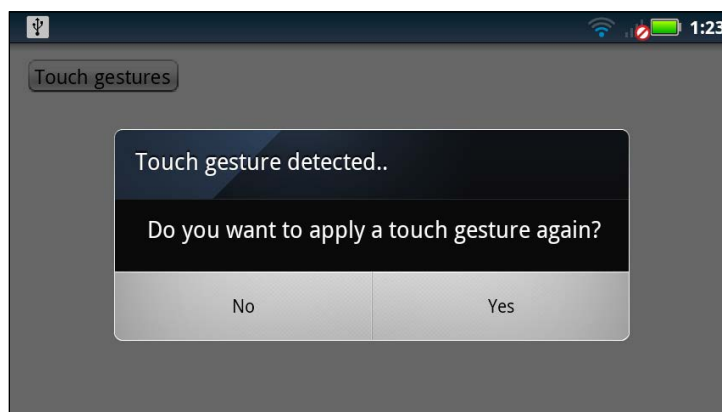
10. Now we need to add the callback function to deal with the selected user option from the confirmation box.

11. If the user selected **Yes** they can continue to press the button and receive the notification window.

12. If they select **No**, we will unregister the event applied to the button, removing the ability to perform any actions when touched.

```
function touchConfirmAction(buttonIndex) {
  if(buttonIndex === 2) {
    x$('#touchme').un('touchstart');
    x$('#touchme').html('Touch gestures are closed');
  } else {
    x$('#touchme').html('Touch gestures are active');
  }
}
```

13. When we run the application on the browser and select **No**, the button value would change and the output would look something like the following screenshot:

How it works...

In this recipe, we used our first event to detect when the DOM would be ready for us that is the `ready()` method.

We also specifically applied a `touchstart` event to a button element, which when pressed would alert the user with a notification window.

We then added in functionality to unregister the specific `touchstart` callback, which would remove the functionality to display the notification window.

> For more information on using events within XUI, make sure you check out the official documentation, available at: `http://xuijs.com/docs/event`.

Updating element styles

As we build our applications and apply a number of styles and properties to define the layout and visual representation of elements, we also need to be able to update and alter the aesthetics and styles to reflect changes or events within the application.

How to do it...

In this recipe we will read, write, and detect the style properties and class attributes assigned to selected elements using XUI's built-in style functions.

1. Create the initial HTML layout for the application. Include the Cordova and XUI JavaScript libraries within the `head` of the document.

2. Write a new `script` tag block before the closing `head` tag, which will hold our custom PhoneGap application code. Define an empty `onLoad` function inside the `script` block.

3. Add three `button` elements to the `body` of the document, each with its own individual `id` attribute.

4. Finally, include the `on` event listener to run the `onLoad` function.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>XUI</title>
    <script type="text/javascript"
    src="xui.js"></script>
  <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript">

    x$(window).on('load', onLoad);

  function onLoad() {

    }

  </script>
</head>
<body>

  <button id="one">Button One</button>
  <button id="two">Button Two</button>
  <button id="three">Button Three</button>

</body>
</html>
```

5. The `onLoad` method will set up the XUI `ready` event handler, which in turn will add the event listener to run the `onDeviceReady` method when the PhoneGap code has been compiled and is ready to run:

```javascript
function onLoad() {

  x$.ready(function() {
    document.addEventListener("deviceready",
    onDeviceReady, false);
  });
}
```

179

6. When the `onDeviceReady` method executes, the first thing we want to do is register a click handler to all of the buttons. We can determine which button was clicked by reading the `this.id` value following a click event.

7. We'll check to see if the selected button has a specific CSS class called `active` applied to it. If it does not, we'll add the class to the element.

8. We'll also obtain the `font-size` style attribute applied to the element and send that value to a new method called `resizeFont`. This will increase the font size value by the integer defined in the parameters; in this case it will add an extra five pixels to the value.

9. If the selected button already has the `active` class applied to it, we'll remove the class and reduce the `font-size` style property on the `button` element.

```
function onDeviceReady() {

    x$('button').on('click', function(e) {

      var selectedButton = x$('#' + this.id + '');

      if(!selectedButton.hasClass('active')) {


          selectedButton.addClass('active');
        selectedButton.getStyle('font-size',
              function(property) {
          resizeFont(selectedButton, property ,'+', 5);
        });


      } else {

          selectedButton.removeClass('active');
        selectedButton.getStyle('font-size',
              function(property) {
          resizeFont(selectedButton, property ,'-', 5);
        });


      }
    });

  }
```

10. Now let's write the `resizeFont` function. This will accept four parameters: the button element, the value of the `getStyle` property response, whether to increase or decrease the font size, and the amount by which to alter the original property.

11. Once we have calculated the new font size, we'll use the `setStyle` method from XUI to change the style value for the element.

```
function resizeFont(element, property, direction, alterBy) {
  var sizeType = property.replace(/[0-9]+/, "");
  var fontSize = property.replace(/[^-\d\.]/g, "");
  if('-' === direction) {
    fontSize = parseInt(fontSize) - alterBy;
  } else {
    fontSize = parseInt(fontSize) + alterBy;
  }
  element.setStyle('font-size', fontSize + sizeType);
}
```

12. To finish off, include some CSS to the bottom of the HTML page to define the two states for the buttons; the normal state and the active/selected state.

```
<style>

button {
  background: rgba(100, 100, 100, 0.6);
  color: #fff;
  padding: 5px 10px;
  float: left;
  margin: 5px 5px 0 0;
  border-radius: 2px;
  font-size: 11px;
  font-weight: 600;
  cursor: pointer;
  border: none;
}

button.active {
  background: #EEA839;
  color: #fff;
  padding: 5px 10px;
  float: left;
  margin: 5px 5px 0 0;
  border-radius: 3px;
  font-size: 11px;
  font-weight: 600;
  cursor: pointer;
  border: none;
}

</style>
```

181

13. If we run the application on a device the initial screen would look like the following screenshot, with all three buttons in their normal state:
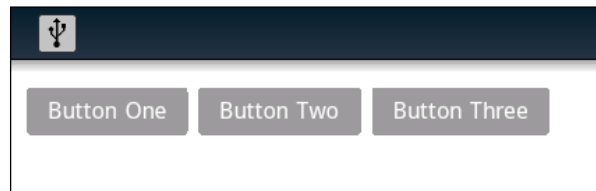
14. If we select a button element, you can see that XUI has applied the active style and altered the size of the font. Selecting an active button will remove the styles and revert it back to its normal state, as shown in the following screenshot:
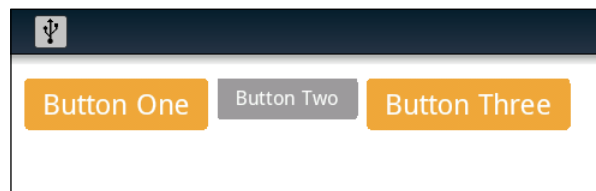
How it works...

In this recipe we made use of the style manipulation and selector functions in XUI.

After detecting a button `click` event, we were able to check if the particular button had a specific class, and were able to easily add it to the element if it did not by using the `hasClass` and `addClass` methods respectively. Similarly, if the class was already applied to the button, we removed it using the `removeClass` method.

We were also able to alter the style properties of our CSS by first accessing it using the `getStyle` method before setting the amended property value using `setStyle`.

There's more...

We took the slightly longer route in this example to detect and change the button class attribute. This was intentional so that we could see the `addClass` and `removeClass` methods in action.

We could have removed them both entirely, as well as the `hasClass` method, by using the `toggleClass` method also available in the XUI library.

This method will add the specified class it if exists on the selected element, or remove it if it does not.

For example:

```
x$('#myButton').toggleClass('active');
```

It's always good to know you have options!

> For more information on how to gather and alter style information, make sure you check out the official XUI documentation, available at: `http://xuijs.com/docs/style`.

Working with remote data and AJAX requests

With a vast array of remote servers exposing their services as accessible APIs you can create some truly dynamic applications by pulling and pushing data to and from external applications and providers.

How to do it...

We will use the `xhr()` method within the XUI library to request data from a remote server, making an asynchronous call to obtain the results from a search.

1. Create the initial HTML layout for the application. Include both the XUI and Cordova JavaScript libraries within the `head` tag of the document.

2. Create a new `script` tag block before the closing `body` tag with an empty `onLoad` function, which we'll populate shortly.

3. Add an XUI `on` event handler to run the `onLoad` method once the DOM has fully loaded:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
        initial-scale=1, maximum-scale=1,
        minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>XUI</title>
    <script type="text/javascript"
    src="xui.js"></script>
      <script type="text/javascript"
    src="cordova-2.0.0.js"></script>
    <script type="text/javascript">
```

183

```
    x$(window).on('load', onLoad);

  function onLoad() {

}

  </script>
</head>
<body>


</body>
</html>
```

4. Within the `body` tag of the document, add a text `input` element with the `id` attribute set to `criteria`. Below this, include a new `button` element with the `id` attribute set to `search`.

5. Finally, create a new `div` block with the `id` attribute set to `response`. This will hold our returned data from the remote calls:

```
<body>

  <input type="text" id="criteria" />
  <button id="search">search</button>

  <div id="response"></div>

</body>
```

6. Amend the `onLoad` method to include a call to the XUI ready `event`, which will execute when the DOM is ready.

7. Within this event handler, add a new event listener to run the `onDeviceReady` method once the native PhoneGap code is ready.

```
function onLoad() {

  x$.ready(function() {
    document.addEventListener("deviceready",
     onDeviceReady, false);
  });

}
```

8. Now let's create the `onDeviceReady` method. We will apply a `touchstart` event handler to the `button` element. This will clear any content currently in the response `div` element.

9. It will then obtain the value of the `criteria` input field as specified by the user, and pass that value to a new function.

```
function onDeviceReady() {

  x$('#search').on('touchstart', function(e) {
    x$('#response').html(' ');
    var criteria = x$('#criteria')[0].value;
    searchTwitter(criteria);
  });

}
```

10. Create the new function called `searchTwitter`, which accepts the user input. This will make a simple AJAX call to the Twitter search API to return the last five entries that match the criteria provided.

11. Set the inline callback option to execute a function called `processResults`, which will also contain the response from the request.

```
function searchTwitter(criteria) {

  x$('#response').xhr(
  'http://search.twitter.com/search.json?q='+
    criteria+'&rpp=5', {
    async: true,
   callback: function() {
     processResults(this.responseText);
   },
   headers:{
     'Mobile':'true'
   }
  });

}
```

12. The `processResults` method will parse the JSON response from the request. The function will loop over the results so that we can access each entry.

185

13. We'll use this data to build a simple element collection containing the content of the tweet, which we'll then add to the bottom of the `response div` container.

```
function processResults(response) {
  var response = JSON.parse(response);
  var results = response.results;
  for(var i=0; i<results.length; i++) {
    var thisTweet = results[i].text;
    var tweetDiv = '<div class="tweet">' +
        thisTweet + '</div>';
    x$('#response').bottom(tweetDiv);
  }
}
```

14. Finally, add some style definitions to the bottom of the document to add some formatting to the returned tweet elements.

```
<style>
  .tweet {
    border: 1px solid #999;
    background: #e5e5e5;
    clear: both;
    margin: 5px;
    padding: 5px;
  }
</style>
```

15. If we run the application on a device, the initial display would like something like the following screenshot:
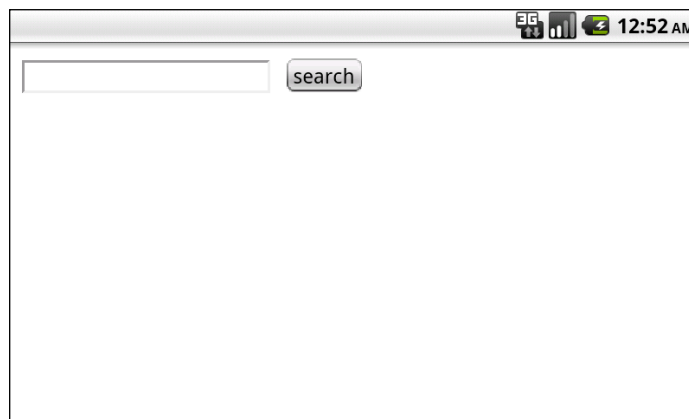
16. Once a user has entered in search criteria and pressed the button, the remote call will fire and the response will look like the following screenshot:
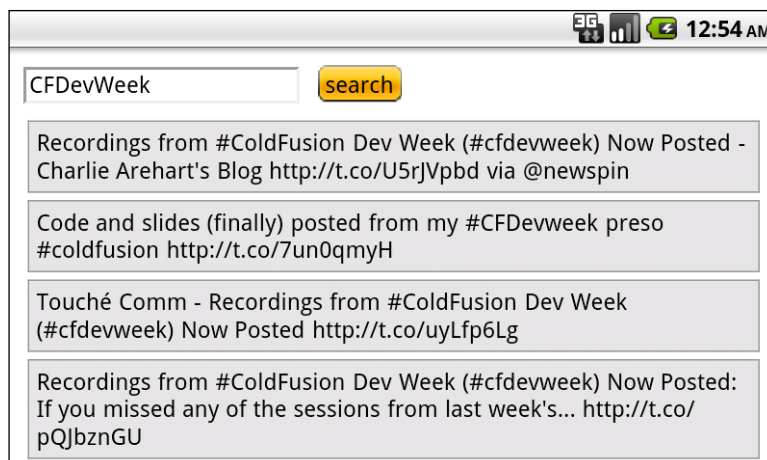
How it works...

In this recipe we used the `xhr()` method available within the XUI JavaScript library to make a request to a remote web service, in this case the Twitter search API.

Using `xhr()`, we made an **XmlHttpRequest** to the provided URL, which ran an asynchronous call. We passed in a callback method, `processResults`, which looped over the returned entries and built an element collection containing the tweet message before adding each one to the bottom of the `response` container.

> For more information on the `xhr()` method, make sure you check out the official XUI documentation , available at: `http://xuijs.com/docs/xhr`.

Animating an element

Although altering elements in our application's pages is relatively easy, we can further enhance the user's experience by adding some animation to these elements as we change them.

How to do it...

We will use the `tween()` method within the XUI library to alter a property of an image and tween the element to its new position on the screen.

1. Create the initial HTML layout for the application. Include the XUI JavaScript library within the `head` tag of the document.

187

2. Add an XUI `ready` event handler to run our code once the DOM has fully loaded.

```html
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport" content="user-scalable=no,
          initial-scale=1, maximum-scale=1,
          minimum-scale=1, width=device-width;" />
    <meta http-equiv="Content-type"
    content="text/html; charset=utf-8">
    <title>XUI</title>
    <script type="text/javascript"
    src="xui.js"></script>
    <script type="text/javascript">

    x$.ready(function() {


});

  </script>
</head>
<body>


</body>
</html>
```

3. Within the `body` tag of the document, add two `button` elements, one with the `id` attribute set to `up`, the other with the `id` attribute set to `down`. Below these, add an empty `div` tag block with the `id` attribute set to `details`.

4. Add a new `div` tag block with the `id` attribute set to `rocket`, inside of which add an image tag with the `src` attribute referencing the `rocket.gif` image file.

```html
<body>

    <button id="up">UP</button>
  <button id="down">DOWN</button>

  <div id="details"></div>

  <div id="rocket">
    <img src="rocket.gif" width="31" height="72" />
  </div>
</body>
```

5. Now let's start adding our custom code to the `ready()` method. Firstly, we'll create two variables that reference the `rocket` and `details div` elements.

188

6. We'll then create a click handler event, applied to the `up` button. This will obtain the value of the `rocket` element's `bottom` CSS property. We can then increment the value by 100 to create the new position for the element.

7. We will then send the property through to the `tween` method to animate the `rocket` element.

```
x$.ready(function() {

  var rocket = x$('#rocket');
  var details = x$('#details');

    x$('#up').on('click', function(e) {

      rocket.getStyle('bottom', function(property) {
       sizeType = property.replace(/[0-9]+/, "");
      topPosition = property.replace(/[^-\d\.]/g, "");
      topPosition = parseInt(topPosition) + 100;
        newPosition = topPosition + sizeType;
      rocket.tween({ bottom:newPosition, duration:1000 },
          function() {
           details.html('bottom: ' + newPosition);
            }
);
    });

  });

});
```

8. We now need to create the click handler event for the `down` button directly below this, within the `ready()` method, which has the same functionality, but decreases the value of the `bottom` property by 100.

```
x$('#down').on('click', function(e) {

  rocket.getStyle('bottom', function(property) {
    sizeType = property.replace(/[0-9]+/, "");
    topPosition = property.replace(/[^-\d\.]/g, "");
    topPosition = parseInt(topPosition) - 100;
    newPosition = topPosition + sizeType;
    rocket.tween({ bottom:newPosition, duration:1000 },
      function() {
          details.html('bottom: ' + newPosition);
      }
    );
  });
});
```

9. Finally, add some style definitions to the bottom of the document to apply the default positions and format for the elements, as shown in the following code snippet:

```
<style>
   body { background: #0c0440 url(outerspace.jpg) repeat; }
   div#rocket { bottom: 0px; position: absolute; width: 31px;
height: 72px; left: 100px; }
   div#details { color: #ffff00; height: 20px; position: inherit;
top: 20px; right: 20px; float: right; }
</style>
```

10. If we run the application on a device, the initial display would look something like the following screenshot:
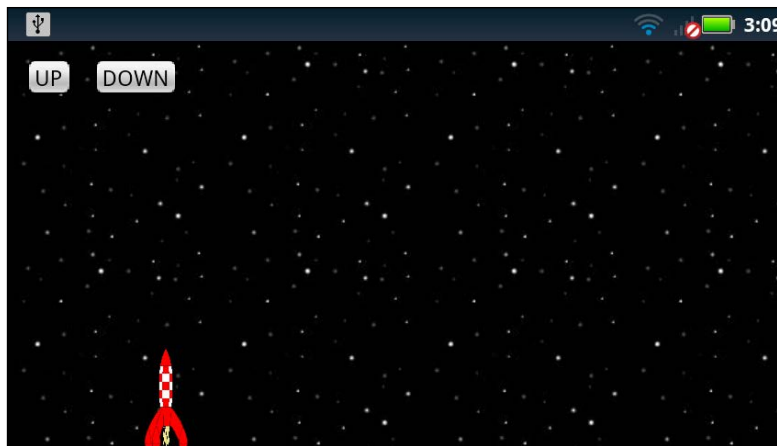
11. If we choose to move the rocket element up, the display would then look something like the following screenshot:
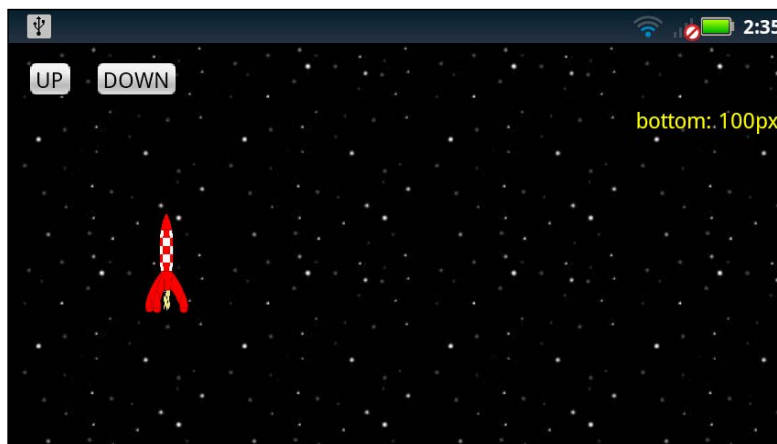
How it works...

The tween() method transforms a CSS property value, in our case the bottom property of the rocket element. Before we could change the value, we first obtained its current value using XUI's getStyle() method. Once we had altered the value of the property, we sent it through to the tween() method. We also sent through an optional property to set duration of the animation in milliseconds.

The tween() method accepts the following two arguments:

 ▶ properties: An Object or Array of CSS properties to tween.
 ▶ callback: An optional Function that is called when the animation is complete.

In our example, we used the optional callback argument to update the content of the details element with the new bottom property for the rocket element.

> For more information on the tween() method, make sure you check out the official XUI documentation, available at: http://xuijs.com/docs/fx.

191

7
User Interface Development with jQuery Mobile

In this chapter, we will cover:

- ▸ Creating a jQuery Mobile layout
- ▸ Persisting data between jQuery Mobile pages
- ▸ Using jQuery Mobile ThemeRoller

Introduction

When we develop for mobile devices, we are extremely limited in terms of space and we have to think differently about creating a user interface or frontend for our application than we would if we were creating for the big screen.

Users on mobile devices need the information clearly represented and given to them in a format and layout that is easily understandable and recognizable on the smaller devices.

In this chapter, we will briefly look into creating a layout for a podcast application, and we will be using the jQuery Mobile framework to do this. With its big, easily identifiable buttons, elements, and interface, it gives us everything we need to create a layout that almost matches the designs of native apps.

Creating a jQuery Mobile layout

In this recipe we will be creating a simple podcast application. It will obtain available shows from a remote XML feed and display these in a list on the main page.

We want the interface and elements to be easily recognizable and simple to use, which jQuery Mobile can easily help us achieve.

Getting ready

Before we start to build our application, we need to ensure we have the jQuery Mobile framework.

Head over to `http://jquerymobile.com/download/` to download the code as a `.zip` file.
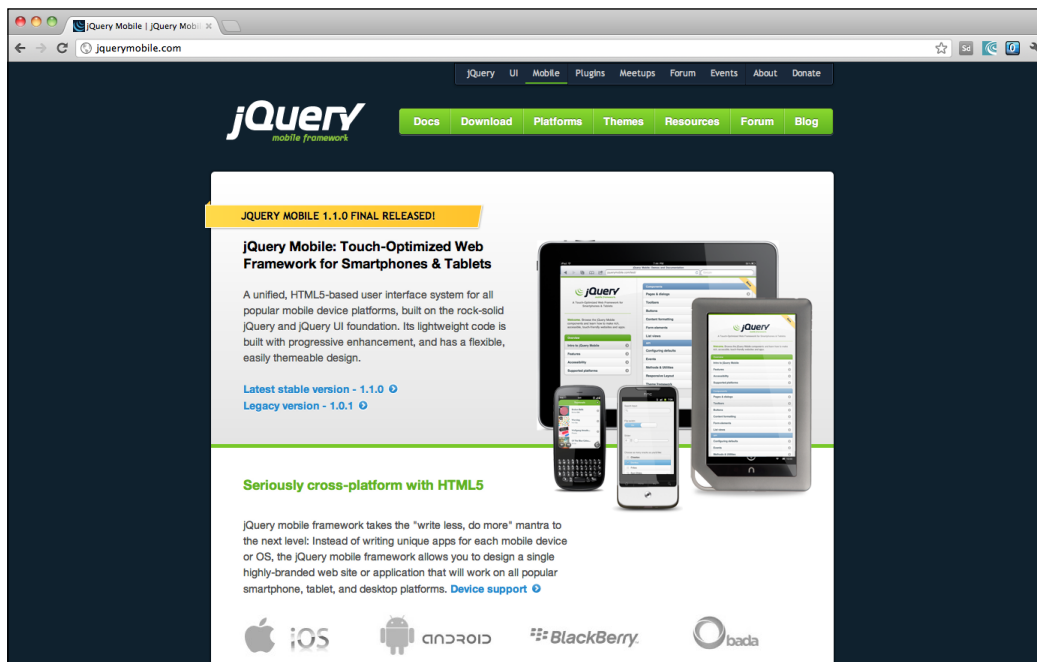
We have options available to include the code stored on a **Content Distribution Network** (**CDN**). This means that the files are hosted on a remote server with a fairly high guarantee of their availability at all times without any server downtime.

Using this method would help to reduce the overall size of our compiled application as it would mean that our app would not contain the files – they are not gargantuan in size, but when dealing with mobile networks and data use, we want to consider the user by reducing the amount of external requests that they may have to make.

We also need to think about connectivity. The users of our application may not have a constant connection to their network, and so we can't rely on remotely stored files. For this reason alone it is best that we include them in our application.

Once you have downloaded the archive, extract the files to a directory within your application project folder. With that done, we can now get started creating our jQuery Mobile application!

How to do it...

We will use the jQuery Mobile framework to create the layout and user interface for our mobile application.

1. Create a new `index.html` file in the project folder, which will be our main application page. The `head` tag of the document will include a `meta` tag that defines the viewport to assist in page definitions for use on mobile devices.

2. Include the stylesheet reference to the jQuery Mobile CSS file, and two JavaScript references to the jQuery Mobile and jQuery core framework `.js` files. Let's also include the `Cordova JavaScript` file within the `head` tag.

    ```
    <!DOCTYPE HTML>
    <html>
      <head>
        <meta name="viewport"
        content="width=screen.width; user-scalable=no" />
      <meta http-equiv="Content-type"
        content="text/html; charset=utf-8">
      <title>CFHour Mobile</title>
      <link href="jquerymobile/jquery.mobile-1.1.1.min.css"
        rel="stylesheet" type="text/css" />
        <script src="jquery-1.8.0.min.js"
        type="text/javascript"></script>
        <script src="jquerymobile/jquery.mobile-1.1.1.min.js"
        type="text/javascript"></script>
        <script src="cordova-2.0.0.js"
        type="text/javascript"></script>

      </head>
    <body>



      </body>
      </html>
    ```

3. Let's start to add jQuery Mobile-specific code. We'll begin by adding a new page to our application. jQuery Mobile recognizes that certain aspects of the code should be defined as a page by the `data-role` attribute specified within the `div` tag.

4. Each of our pages will have a unique `id` attribute. Here we'll call our page `home`.

195

5. Create a `div` tag block with the `data-role` set to `header`, and a second with the `data-role` set to `footer`.

6. We'll also create a new `div` tag with the `data-role` attribute set to `content`, which declares the code within as the content for our page, between the header and footer sections. This is shown in the following code snippet:

```html
<body>

  <div data-role="page" id="home">

    <div data-role="header">
      <h1>CFHour Mobile</h1>
    </div>

    <div data-role="content">
      <p>CFHour is the number #1
    ColdFusion podcast.</p>
    </div>

    <div data-role="footer">
      <h4>&copy; cfhour.com</h4>
    </div>

  </div>

</body>
```

7. With very little code, we have created a simple application layout. When we run the application on a device, it would look something like the following screenshot:

8. As you can see, we have made extensive use of the HTML5 data attributes in our code. jQuery Mobile uses these to define the layout, markup and behavior to our code.

9. Let's add a second page into our application. Below the current page definition, we'll create a new `div` tag block with the `data-role` attribute set to `page`. Set the `id` attribute for the page to `about` so that jQuery Mobile can differentiate this page from the first.

10. We've also included a header, content, and footer section, as these need to be defined for each page.

```
<div data-role="page" id="about">

  <div data-role="header">
    <h1>About CFHour</h1>
  </div>

  <div data-role="content">
    <p>CFHour is a weekly podcast primarily focused on
      ColdFusion development, but brings you news and
      updates about all things 'web'.</p>
    <p>Join your hosts Dave Ferguson, Scott Stroz and their
      producer Matt Gifford for the latest information,
      live shows and guest interviews.</p>
    <p>
      <a href="http://www.cfhour.com"
      data-role="button">Visit www.cfhour.com</a>
    </p>
  </div>

  <div data-role="footer">
    <h4>&copy; cfhour.com</h4>
  </div>

</div>
```

> If you create an application with multiple pages in one file, jQuery Mobile will display the first page it encounters, in this case the home page. It is important to remember that the order of the content in your application will have an effect on what is rendered.

11. We now have our second page created, but we haven't yet created a way for the user to navigate to it. Add an `anchor` tag within the header of the first page, `home`, and set the `href` attribute to point to the second page by referencing its specific `id` attribute, `about`. This creates an internal link, and jQuery will know exactly what to do.

12. We can also make use of the mobile framework to turn the standard link into a button, and we'll add an icon from those included in the library to enhance the user interface, as shown in the following code snippet:

```
<div data-role="header">

    <h1>CFHour Mobile</h1>

    <a href="#about" data-role="button"
    data-icon="info">About</a>

</div>
```

This is shown in the following screenshot:

13. We also need to give the user the option to navigate back to the home page. If their mobile device has a back button, they can use that to switch back but to enhance the user experience. It's best that we provide them with the ability to do so from within the application. Let's amend the about page code to include the following:

```
<div data-role="page" id="about"
    data-add-back-btn="true">

    <div data-role="header">
    <h1>About CFHour</h1>
    </div>
```

14. Instead of adding a button within the header as we did on the first page, we can use jQuery Mobile's built-in function to automatically include a back button for us, as shown in the following screenshot. To do so, we simply added a new `data` attribute to the opening page `div` tag, `data-add-back-btn`, and set it to `true`.
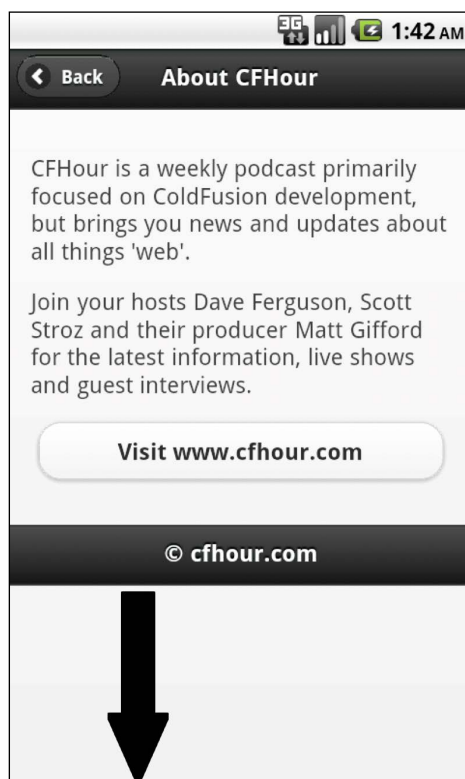
15. We now have our second page complete with automatically generated back button, but as you can see on the screenshot of the application we also have a lot of empty space below the footer. Let's resolve that and set the footer to sit at the bottom of the screen. Amend the footer `div` tag within each page by adding a new `data` attribute, `data-position`, and set the value to `fixed`, as shown in the following code snippet:

```
<div data-role="footer"
data-position="fixed">
```

16. We now have the header and footer fixed to their positions at the top and bottom of the screen. This is cleaner for the user, provides a more consistent layout, and also allows the user to scroll the main content of the page and keep the header and footer locked into position.

17. When we run the application on our device now, the layout will look something like the following screenshot:

With the core of the application's layout now created, let's start to add some dynamic functionality:

1. When the user loads up the application we want to provide them with a list of available podcast episodes to listen to. We'll start off by revising the code in the initial home page to include an unordered list.

```
<div data-role="content">

    <p>CFHour is the number #1 ColdFusion podcast.</p>

    <p>Select a show to listen to:</p>

    <ul id="showList" data-role="listview"
    data-inset="true"></ul>

</div>
```

2. Create a new `script` tag block at the bottom of the file to hold our custom code. To begin with, we'll bind a jQuery Mobile `pagecreate` event to the `home` page, which will run a new method called `getRemoteFeed`.

```
$("#home").bind("pagecreate", function(e) {

getRemoteFeed();


})
```

3. The `getRemoteFeed` method will run a jQuery `get()` method call to obtain the contents of the podcast feed. When the XML data has been retrieved, we'll loop over each item element (which represents an individual show), and generate a new list item for each one.

4. To obtain the subtitle/description for the show, we send the item node into a separate method called `XMLtoString`, which handles the specific namespacing for us and converts the entire node into a string variable. This method isn't included in this code for brevity, but is available in the completed project code.

5. When building our list, you can see we are adding elements from the feed as data attributes to the list item, as well as the text for the link.

6. After looping through all available items, we append the complete string containing our list items to the `shortList` unordered list element we created earlier. Finally, we need to refresh the list to allow jQuery Mobile to render the list elements with the correct styles and formatting.

```
var getRemoteFeed = function() {

  $.get(
"http://feeds2.feedburner.com/CfhourColdfusionPodcast",
  {},
  function(data) {

   var listItem = '';

   $(data).find('item').each(function(){
   var strXML = XMLtoString($(this));
   var showDescription =

    strXML.substring(
    strXML.indexOf('<itunes:subtitle>') + 17,
    strXML.indexOf('</itunes:subtitle>')
    );
```

```
listItem += '<li data-show-description="'
 + showDescription + '" data-show-title="'
 + $(this).find('title').text()
  + '" data-enclosure="'
 + $(this).find("enclosure").attr("url")
  +'" id="'+ $(this).find("guid")
  +'"><h3><a>'
 + $(this).find('title').text()
  + '</a></h3><p>Released: '
 + $(this).find('pubDate').text() +'</p></li>';
});

$('#showList').append(listItem);
$("#showList").listview("refresh");
}
);
};
```

7. Our main application page will now look something like the following screenshot:
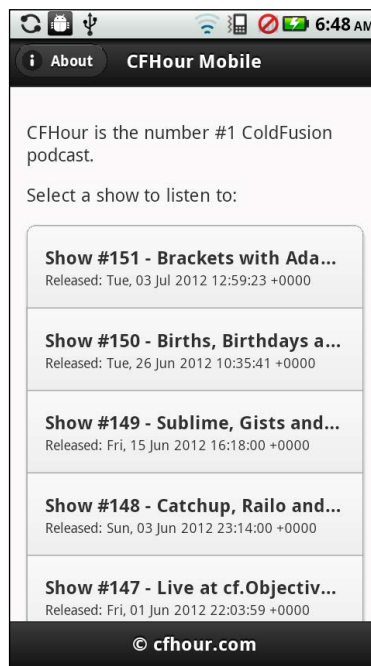
We defined the list to be inset. Had we left this attribute as default,
the list would have covered the entire width of the screen.

How it works...

The jQuery Mobile framework gives developers the ability to create application layouts that are responsive to the size of the device screen and provide an entire library of user interface elements that aesthetically match a native application.

We were able to create a layout and define various sections and content simply by using data attributes within the HTML. jQuery Mobile has been built to apply styles and events based upon the existence of these data attributes. This means that we need only write well-structured HTML code – we do not need to worry about diving too deeply into a new language or any development framework that uses the model view architecture.

Persisting data between jQuery Mobile pages

In this recipe we will build upon the podcast application built in the previous recipe, extending the functionality and features available to the user.

So far, our podcast application consists of a few simple pages that are independent of each other; that is to say, any content consumed by each page is used by that page only.

How to do it...

We will use the `localStorage` capabilities to save and retrieve information, persisting it across pages:

1. With the list now populated, we now need to revise the home method to include a function to capture the tap events on each list item. This will obtain the `title`, `enclosure`, and `description` attribute values from the selected item and set them into the `localStorage` on the device so that we can persist them to the next screen. We'll then force a page change to a new page called `itemdetail`.

    ```
    $("#home").bind("pagecreate", function(e) {

        getRemoteFeed();

        $('#showList').on('tap', 'li', function(){

        localStorage.clear();
        localStorage.setItem("enclosureURL",
          $(this).attr('data-enclosure'));
        localStorage.setItem("showTitle",
          $(this).attr('data-show-title'));
        localStorage.setItem("showDescription",
    ```

```
            $(this).attr('data-show-description'));

            $.mobile.changePage('#itemdetail');

      });

  })
```

2. Let's now create the new `itemdetail` page, which must sit in the code below the initial home page. Remember that the order of page content in your jQuery mobile application is important. We will leave the `h1` tag within the header empty, as we'll populate the title with the name of the show from the `localStorage`. We have also added another link button inside the header to open up a new page, `showdescription`. This link differs from the others we have implemented because it will open the page in a dialog window as we have specified the `data-rel` attribute to `dialog`.

3. In the content of the page, we'll define the layout for the audio controls, which will manage the playback of the remote audio file. The code is as follows:

```html
<div data-role="page"
  id="itemdetail" data-add-back-btn="true">

  <div data-role="header" data-position="fixed">
    <h1></h1>
    <a href="#showdescription" data-role="button"
    data-icon="info" data-rel="dialog"
    class="ui-btn-right">Description</a>
  </div>

  <div data-role="content">

  <h2 id="showTitle"></h2>

  <div data-role="button" id="playaudio">Play</div>
  <div data-role="button" id="pauseaudio">Pause</div>
  <div data-role="button" id="stopaudio">Stop</div>

  <div class="ui-grid-a">
    <div class="ui-block-a">
  Current: <span id="audio_position">0 sec</span></div>
    <div class="ui-block-b">
  Total: <span id="audio_duration">0</span> sec</div>
    </div>

  </div>
```

```
<div data-role="footer" data-position="fixed">
  <h4>&copy; cfhour.com</h4>
</div>

</div>
```

4. Let's now add the JavaScript controls for the audio into the `.js` file. We'll begin by including a `pagebeforeshow` event bound to the `itemdetail` page. This will obtain the MP3 file's remote URL and the show title from the `localStorage` and define the playback controls for the audio player.

```
$("#itemdetail").bind("pagebeforeshow", function(e) {

  var mp3URL = localStorage.getItem("enclosureURL");
    var showTitle = localStorage.getItem("showTitle");

  var audioMedia = null,
    audioTimer = null,
    duration = -1,
    is_paused = false;
```

5. We can now set the show title as the title for the page, ad start to apply the audio controls:

```
  $('#showTitle').html(showTitle);

  $("#playaudio").live('tap', function() {
    if (audioMedia === null) {
    $("#audio_duration").html("0");
    $("#audio_position").html("Loading...");
    audioMedia = new Media(mp3URL, onSuccess, onError);
    audioMedia.play();
    } else {
    if (is_paused) {
    is_paused = false;
     audioMedia.play();
    }
  }
}

  if (audioTimer === null) {
    audioTimer = setInterval(function() {
   audioMedia.getCurrentPosition(
    function(position) {
      if (position > -1) {
          setAudioPosition(Math.round(position));
```

```
            if (duration <= 0) {
              duration = audioMedia.getDuration();
            if (duration > 0) {
              duration = Math.round(duration);

          $("#audio_duration").html(duration);
        }
      }
    }
  },
      function(error) {
      setAudioPosition("Error: " + error);
    }
   );
  }, 1000);
 }
});

  function setAudioPosition(position) {
    $("#audio_position").html(position + " sec");
}
```

6. Include the success and error callback methods, as shown in the following code:

```
function onSuccess() {
    setAudioPosition(duration);
    clearInterval(audioTimer);
    audioTimer = null;
    audioMedia = null;
    is_paused = false;
    duration = -1;
}

function onError(error) {
    alert('code: '    + error.code    + '\n' +
            'message: ' + error.message + '\n');
    clearInterval(audioTimer);
    audioTimer = null;
    audioMedia = null;
    is_paused = false;
    setAudioPosition("0");
}
```

7. We'll now add the methods to pause and stop the audio, as well as the event handlers to detect the tap action on the relevant control buttons to run these functions.

```
function pauseAudio() {
    if (is_paused) return;
    if (audioMedia) {
        is_paused = true;
        audioMedia.pause();
    }
}

function stopAudio() {
    if (audioMedia) {
        audioMedia.stop();
        audioMedia.release();
        audioMedia = null;
    }
    if (audioTimer) {
        clearInterval(audioTimer);
        audioTimer = null;
    }

    is_paused = false;
    duration = 0;
}

    $("#pauseaudio").live('tap', function() {
pauseAudio();
    });

    $("#stopaudio").live('tap', function() {
stopAudio();
    });

});
```
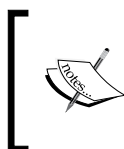
> We won't go into detail about the audio functions here, as they are covered in detail in the *Playing audio files from the local filesystem or over HTTP* recipe in *Chapter 3*, *Working with Audio, Images, and Videos*.

8. When we run the application on the device and select an episode from the list, the show detail page will look something like the following screenshot:
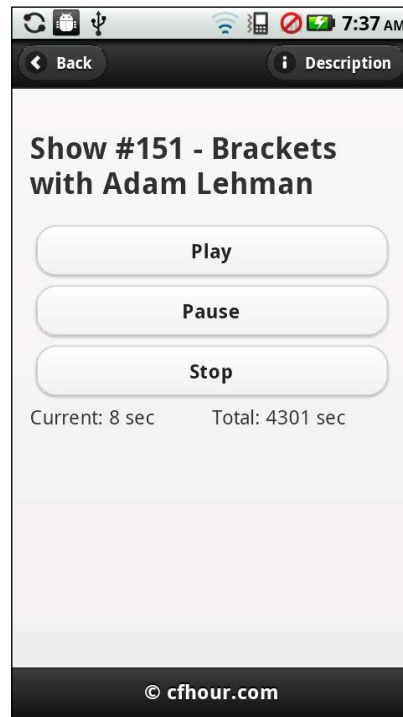
9. Add a new page into the document and set the id attribute to showDescription. We'll also set the id attribute of the content div block to descriptioncontent and include an empty paragraph tag block, into which we'll insert the content dynamically.

10. Finally, we'll also include a link button, which will close the dialog window for the user.

```
<div data-role="page" id="showDescription">

  <div data-role="header">
    <h1>Notes</h1>
  </div>

   <div id="descriptionContent" data-role="content">
    <p></p>
    <a href="#" data-rel="back" data-role="button">Close</a>
  </div>

</div>
```

11. Amend the `.js` file once more and include a new `pagebeforeshow` event binding to the `showDescription` page. This will obtain the `showDescription` key value from the `localStorage` and set it into the empty paragraph tags.

```
$("#showdescription").bind("pagebeforeshow", function(e) {
var description = localStorage.getItem("showDescription");
$('#descriptionContent p:first').html(description);
});
```

The result will be as shown in the following screenshot:
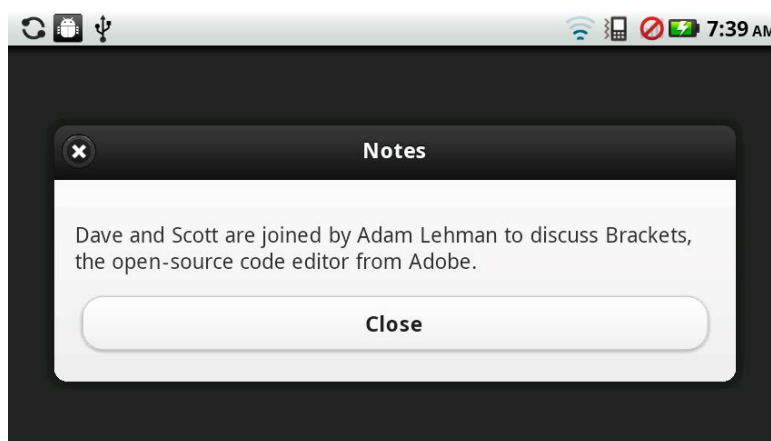
How it works...

We were able to add functionality to the home page to set the touch interaction for each list item. This set the values of the selected individual feed items into the device `localStorage`, which allowed us to access them using the `localStoragegetItem()` method on a new page, independent from the data feed.

We also altered the link to open the `itemDetail` page in a dialog window overlay by setting the `data-rel` attribute for the link itself.

There's more...

You can find out much more about what the jQuery Mobile framework has to offer and how to implement the many events, UI elements and much more in *jQuery Mobile Web Development Essentials*, written by *Raymond Camden* and *Andy Matthews*, published by *Packt Publishing*. Available at:

```
http://www.packtpub.com/jquery-mobile-web-development-
essentials/book
```

See also

▸ The *Caching content using the web storage local storage API* recipe in *Chapter 2, File System, Storage, and Local Databases*

▸ The *Playing audio files from the local filesystem or over HTTP* recipe in *Chapter 3, Working with Audio, Images, and Videos*

Using jQuery Mobile ThemeRoller

The jQuery Mobile framework not only provides with near-native aesthetics and functionality for page transitions, user interface elements, and page layouts, it also gives us the ability to customize the visual theme of our application. This is managed by changing the specifying values for the data-theme attribute on various elements and containers.

The framework itself ships with five themes, or swatches, built-in, alphabetized from A through to E.
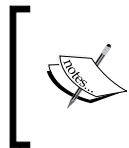
> For more information on theming your application, please read the official jQuery Mobile documentation, available at: `http://jquerymobile.com/demos/1.1.1/docs/api/themes.html`.

Although the provided themes work beautifully, and care and consideration have gone into them by the jQuery Mobile team in regards to readability and accessibility, the themes should be considered a starting point and not a definitive design for our applications. If you have spent the time developing a custom-made native application that interacts with your data and brand, you also want to make sure that visually it stands out from the crowd and doesn't look like an *off-the-shelf* theme. Make an impact and make it individual.

How to do it...

In this recipe we will explore the features offered by jQuery Mobile **ThemeRoller**, an online application that allows us to generate our own themes or swatches for our mobile application, using the drag-and-drop interactions.

1. Head over to `http://jquerymobile.com/themeroller/` to begin the creation and customization of your swatches:
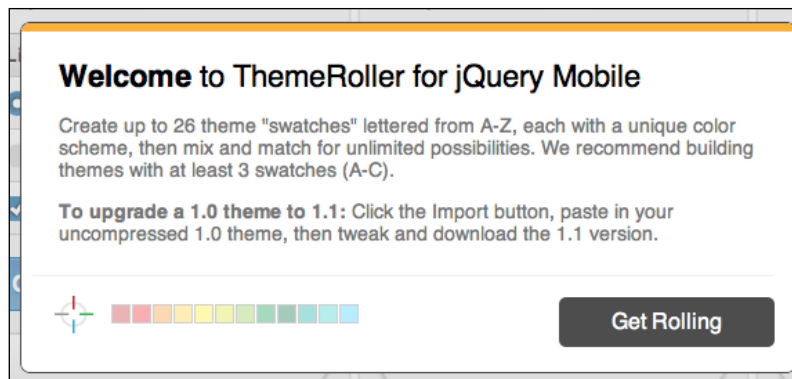
2. The homepage will load with a welcome message overlay box, which also reminds you that you have the ability make 26 swatches in total. This should give you plenty of scope to unleash your creative side and explore the many color matches and possibilities. Click on the **Get Rolling** button to dismiss the message.

3. The welcome message recommends that you create a minimum of three swatch variations for your theme. As such, it has rendered three jQuery Mobile page layouts for you to get started with. These are fully interactive so that you can see how your theme looks and feels on a true application layout, as shown in the following screenshot:
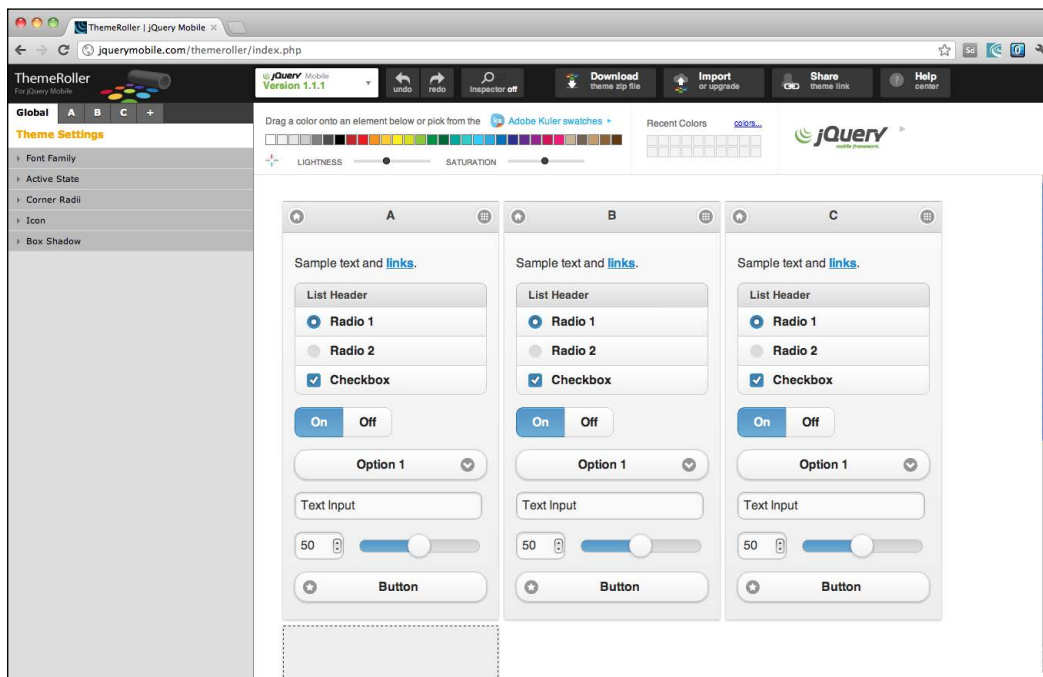
4. To add more than three swatches to your theme, click on the empty layout holder in the main preview panel. This will generate the next swatch and generate the name in the next alphabetical order.

5. The left-hand side of the interface is home to the inspector panel. From here you have control over the global theme settings such as **Font Family**, icon color, and **Corner Radii**. Any changes made to this section will be applied to all of the swatches selected in the preview section.
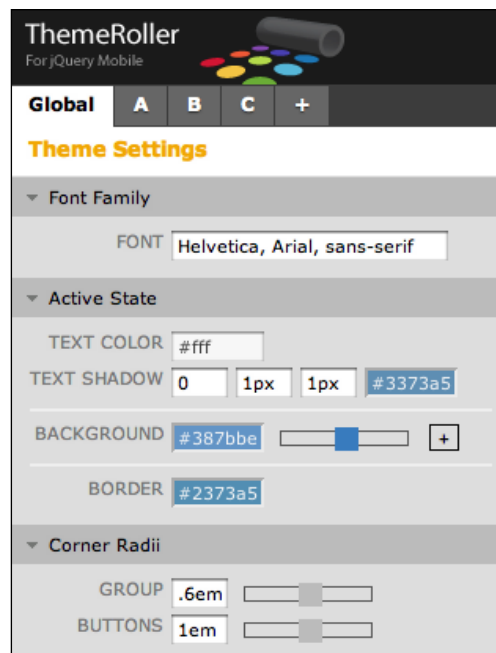
6. Next to the **Global** tab, you also have access to change theme settings for each individual swatch, accessed by the alphabet character assigned to each swatch. You can also add a new swatch from here, which will automatically be placed into the preview section, using the **+** button.

7. You can also delete or duplicate a specific swatch from these tabs, using the **Delete** or **Duplicate** links within each swatch tab, as shown in the following screenshot:
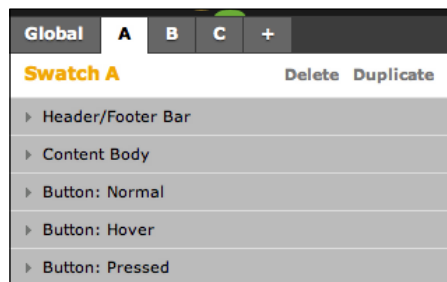
8. Adding colors to your theme is incredibly easy. Above the main preview content, you will see a panel containing a number of color blocks and empty squares. To apply a color to a section of the jQuery Mobile layout, simply drag the selected color block from the panel and drop it onto the UI element on a swatch you wish to update. The change will be applied instantly, and the selected color will be placed into the **Recent Colors** palette for quick reference should you wish to apply that color elsewhere within the same swatch.

9. You can also adjust the lightness and saturation of each color before applying it to your theme by adjusting the **LIGHTNESS** and **SATURATION** sliders beneath the main palette display, as shown in the following screenshot:
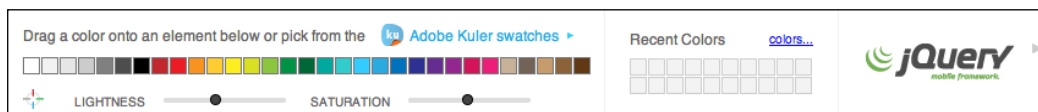
10. If you need some color inspiration or want to see what color schemes other creative professionals have developed, you can access the **Adobe kuler** service by clicking on the **Adobe Kuler swatches** link, which will display visual representations of the latest swatches generated through the service. You can also filter for the most popular or selected random themes, as shown in the following screenshot:

213
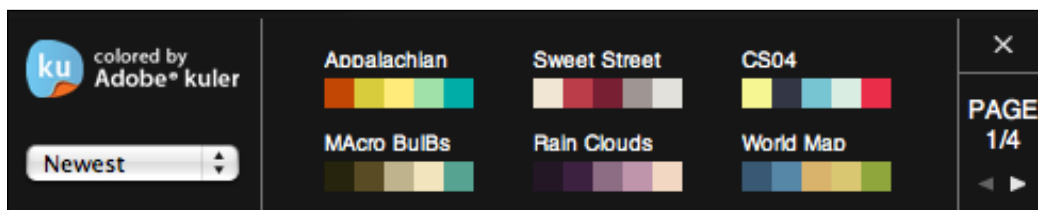
11. You can also create your own free account with Adobe kuler to start generating and sharing your own swatches. To find out more, visit `http://kuler.adobe.com/`.
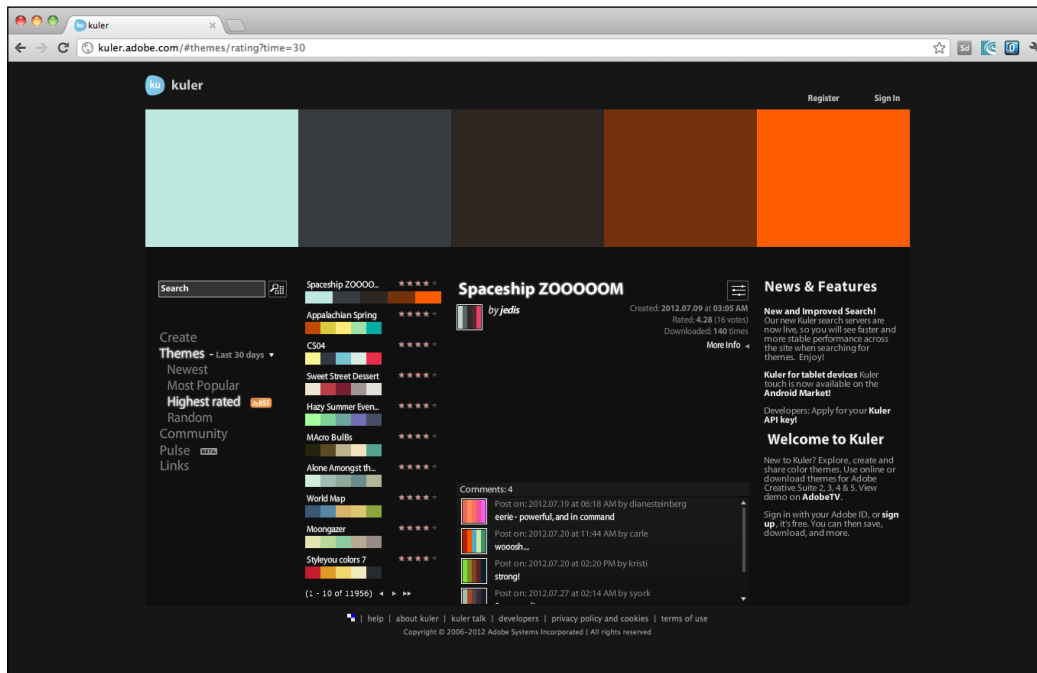
12. The toolbar at the top of the interface gives you instant access to some useful tasks and features, including the undo and redo options:
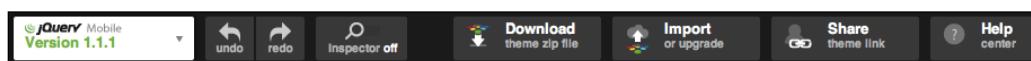
13. If you have an existing theme that you would like to amend or revise, you can import it directly into the interface by using the **Import** option. This will generate the correct number of layouts for each swatch in the theme, and apply the styles to each one so that you have an immediate visual representation of the theme.

14. A very useful tool is the **Inspector** tool. Click on this to turn it on or off. When turned on, you simply need to hover over a specific part of the layout and click on the selection. This will open up the specific panel and tab in the **Inspector** panel on the left-hand side of the interface. This is great for being able to access the portion of the layout that you wish to edit instantly.

15. Once you are happy with your color choices, you can choose to download your completed theme. This will package the relevant files in a `.zip` file in both compressed and uncompressed formats. Simply import the CSS files into your project location and include the reference to the stylesheet within the `head` tag of your document. This is shown in the following screenshot:
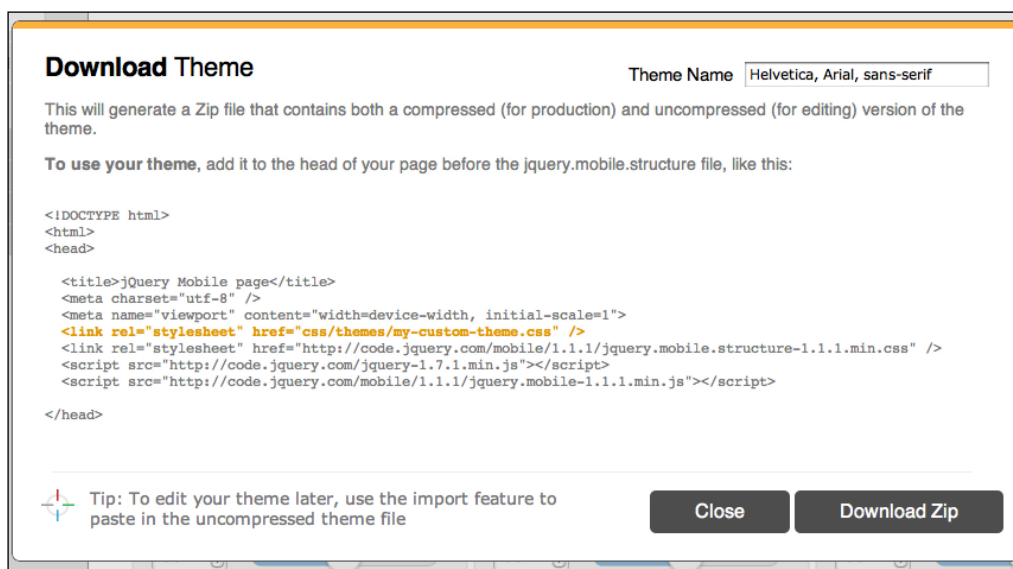
How it works...

Using a simple online interface, we have the ability to create color swatches and themes that better suit our application's brand while still using the power offered by the jQuery Mobile framework.

There's more...

A framework is a framework, and offers a standard approach to fulfill common design and development tasks. Themes generated by the ThemeRoller application (and indeed the themes that come with the jQuery Mobile framework) still look like every other jQuery Mobile theme – they do share common interface elements after all.

One thing to remember is that these themes are nothing more than CSS, and as such you can create truly custom-made layouts and designs with the right amount of skill, time, and patience. Your mobile application need not look like a clone of every other application on the market place using the same framework, while still retaining the common user interface elements that help define it as a mobile application.

215

For inspiration on what you can achieve with some CSS, have a look at the JQM Gallery site, which showcases some wonderful designs created using the jQuery Mobile framework. This is available at `http://www.jqmgallery.com/`.
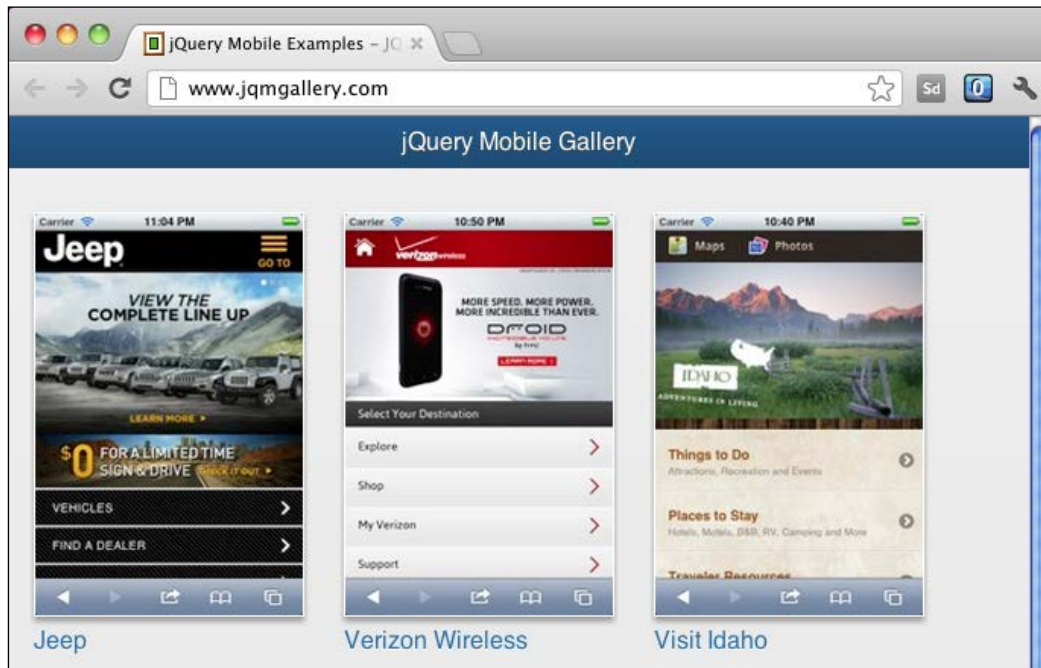
8
Extending PhoneGap with Plugins

In this chapter, we will cover:

- ▸ Extending your Cordova Android application with a native plugin
- ▸ Extending your Cordova iOS application with a native plugin
- ▸ The plugin repository

Introduction

Along with providing developers with an incredibly easy yet powerful way to build native mobile applications using HTML, CSS, and JavaScript, the PhoneGap framework also gives us the ability to further extend the functionality available by creating native plugins that can interact in a more detailed manner with device features and functions that are not already exposed through the PhoneGap API.

By creating native plugins, we can enhance the already vast list of methods available through the API or build totally unique features, all of which will be made available to call and process responses from JavaScript methods.

For some, the thought of writing code that is native to the device platform, you wish to enhance, may be a little daunting. If you haven't had any prior exposure or experience in these languages they may be perceived as some form of unwelcome paradigm shift.

Luckily, the PhoneGap API simplifies this process as much as possible and essentially breaks it down into two core related parts: some JavaScript code to use within your HTML applications, and a corresponding native class to perform actions and processes in the native code. You will also have to edit and amend some XML to inform Cordova of your plugin and to grant permissions for its use, which is also incredibly simple. PhoneGap will manage the communication between the two parts for you, so all you have to worry about is building your awesome applications.

Extending your Cordova application with a native plugin

Android devices have proven to be *developer-friendly* in so far as they easily allow us to test and deploy unsigned applications throughout the development process.

Let's see how we can extend the PhoneGap API features for our Android applications.

Getting ready

Before you can create your native Android plugins, you must have a working Android development environment on your local machine.

For details on how to set this up, read the recipe *Using Eclipse to develop Android Cordova applications* in *Chapter 9, Development Tools and Testing*.

How to do it...

In this recipe we will create a native Android plugin for our Cordova application, using Eclipse as our development environment.

1. The new Cordova project created in Eclipse can be named `MyPlugin`.

2. Create the initial HTML layout in `index.html`, including a reference in the `head` tag of the document to the `Cordova JavaScript` file.

3. Below this, add a new `script` tag block, inside of which place the `deviceready` event listener to handle the initialization of our Cordova code. The code to be added is given as follows:

```
<!DOCTYPE HTML>
<html>
  <head>
    <meta name="viewport"
      content="width=320; user-scalable=no" />
    <meta http-equiv="Content-type"
      content="text/html; charset=utf-8">
    <title>My Plugin Application</title>
    <script type="text/javascript"
      src="cordova-2.0.0.js"></script>
  <script type="text/javascript"
```

```
            var onDeviceReady = function() {
                    document.getElementById("devready").innerHTML
                    = "Device ready.";
            };
            function init() {
                    document.addEventListener("deviceready",
                    onDeviceReady, true);
            }
    </script>

    </head>
    <body onload="init();">
      <h2>My Plugin App</h2>

      <p>Native Android Plugins.</p>
      <p>
      <span id="devready">Device not ready.</span>
      </p>

    </body>
    </html>
```

4. Create a new JavaScript file in the `assets/www` directory called `MyPlugin.js` and place the following code at the top of the new file:

```
    varMyPlugin = function(){};

    MyPlugin.prototype.greeting = function(
    message, successCallback, errorCallback){
       cordova.exec(
        successCallback,
        errorCallback,
        'MyPlugin',
        'greeting',
        [message]
      );
    };
```

5. Firstly, we define the class name for our JavaScript plugin, and then create the first method we will invoke called `greeting`.

6. Add the following code to the bottom of the file to load the `MyPlugin` class into the `window.plugins` object:

```
if(!window.plugins) {
    window.plugins = {};
}
if (!window.plugins.MyPlugin) {
    window.plugins.MyPlugin = newMyPlugin();
}
```

7. Create a new Java class file in your project by accessing **File|New|Class** from the main Eclipse menu. You can also access the wizard by selecting **File|New|Other** from the main menu and finding the Java class option in the list, as shown in the following screenshot:
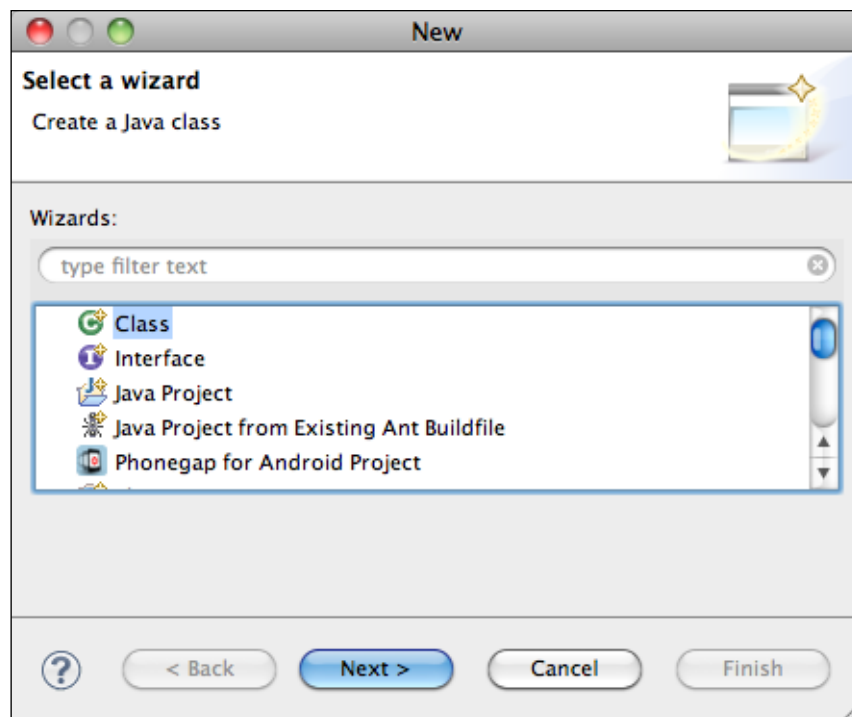
8. The **New Java Class** wizard will open to help you complete this task. Provide the reverse domain format package name, and set the class name to `MyPlugin`. As we want to hook into the Cordova framework, we will also make sure it extends the `org.apache.cordova.api.Plugin` class, which is part of the Cordova core API.
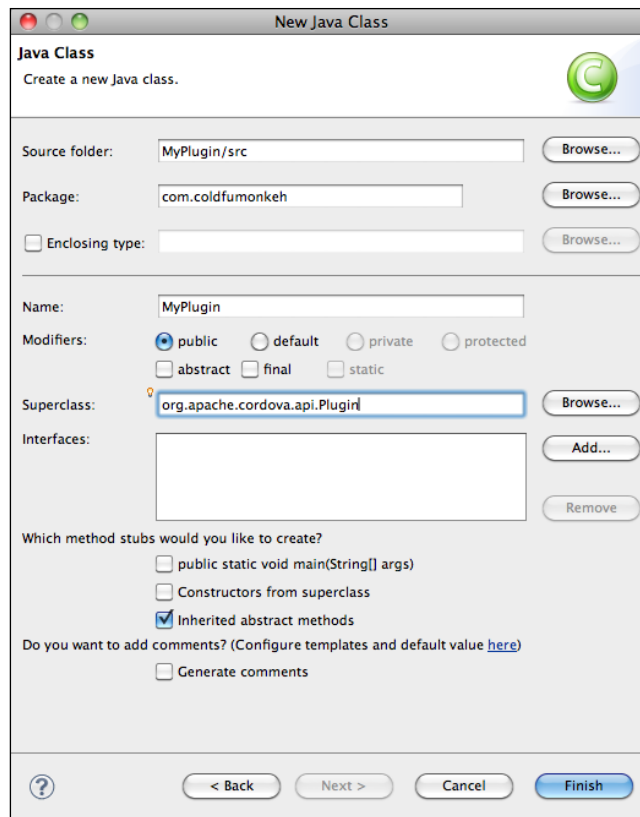
> All custom Cordova Android plugins must extend the `org.apache.cordova.api.Plugin` class as this encapsulates all of the logic required to communicate between the native plugin and the JavaScript implementation, using the Cordova API as the bridge.

9. Click on the **Finish** button to complete the wizard. The new Java class file will now be open in your Eclipse editor window. Let's rename the parameters included in the `execute` method to make them easier to work with and read. Rename `arg0` to `action`, `arg1` to `data`, and `arg2` to `callbackId`.

10. We'll also remove the automatically generated `return` statement and add in a default `PluginResult` status response and `result` value, as shown in the following code snippet:

```
public PluginResult execute(
String action,
JSONArray data,
String callbackId
```

```
     ) {
       PluginResult.Status status = PluginResult.Status.OK;
       String result = "";

     }
```

11. Let's now add the native code for our plugin. Add the following code within the `execute` method after the `result` variable definition. We'll begin by wrapping our code in a `try/catch` block to trap and handle any errors.

12. We'll qualify the `action` value sent through to the plugin to ensure we have a valid method to run. In this case, we'll check to make sure that it matches `greeting`. We'll also assign the first item from the `data` JSON array to our `result` variable, which we'll return to the JavaScript implementation of the plugin as a `PluginResult`, also sending through the `status` value we defined earlier.

13. If we receive empty data, we'll respond with another `PluginResult` object, setting the status to `PluginResult.Status.ERROR`.

14. Finally, if the `action` sent through does not match any value we have explicitly defined, we will respond with an `INVALID_ACTION` status:

```
    try {
      if ("greeting".equals(action)) {

        result = data.getString(0);

        if (result != null && result.length() > 0) {
          PluginResult pluginResult =
        new PluginResult(status, result);
        return pluginResult;
         } else {
        return new PluginResult(PluginResult.Status.ERROR);
         }

      } else {

        status = PluginResult.Status.INVALID_ACTION;

      }

      return new PluginResult(status, result);

    } catch (JSONException e) {

      return new
      PluginResult(PluginResult.Status.JSON_EXCEPTION);

    }
```

15. Add a new import at the beginning of the class file to make sure we have included the `JSONException` type, which we use in the final `catch` response.

```
package com.coldfumonkeh;

import org.apache.cordova.api.Plugin;
import org.apache.cordova.api.PluginResult;
import org.json.JSONArray;
import org.json.JSONException;

public class MyPlugin extends Plugin {
```

16. Save the file and open up `index.html` once more. Add a reference to the `MyPlugin.js` file in the `head` tag of the document, below the Cordova JavaScript reference, as follows:

```
<script type="text/javascript"
src="cordova-2.0.0.js"></script>

<script type="text/javascript"
src="MyPlugin.js"></script>
```

17. Let's now amend the `onDeviceReady` function to include the request to our plugin method. Here we will call the plugin from the `window.plugin` object and execute the `greeting` method, passing in the message to display and the `successCallback` function.

```
var onDeviceReady = function() {

document.getElementById("devready").innerHTML
  = "OnDeviceReady fired.";

window.plugins.MyPlugin.greeting(
   "My First Plugin",
   function(echoValue) {
     alert(echoValue);
   }
 );

};
```

18. Before we can run the application, we need to add a mapping to our plugin within the Cordova `config.xml` file. Right-click on the file and select **Open With|Text Editor**. Locate the `plugins` section and add a reference to our plugin within the following node:

```
<plugin name="MyPlugin" value="com.coldfumonkeh.MyPlugin"/>
```

This is shown in the following screenshot:

```
34 <plugins>
35     <plugin name="App" value="org.apache.cordova.App"/>
36     <plugin name="Geolocation" value="org.apache.cordova.GeoBroker"/>
37     <plugin name="Device" value="org.apache.cordova.Device"/>
38     <plugin name="Accelerometer" value="org.apache.cordova.AccelListener"/>
39     <plugin name="Compass" value="org.apache.cordova.CompassListener"/>
40     <plugin name="Media" value="org.apache.cordova.AudioHandler"/>
41     <plugin name="Camera" value="org.apache.cordova.CameraLauncher"/>
42     <plugin name="Contacts" value="org.apache.cordova.ContactManager"/>
43     <plugin name="File" value="org.apache.cordova.FileUtils"/>
44     <plugin name="NetworkStatus" value="org.apache.cordova.NetworkManager"/>
45     <plugin name="Notification" value="org.apache.cordova.Notification"/>
46     <plugin name="Storage" value="org.apache.cordova.Storage"/>
47     <plugin name="Temperature" value="org.apache.cordova.TempListener"/>
48     <plugin name="FileTransfer" value="org.apache.cordova.FileTransfer"/>
49     <plugin name="Capture" value="org.apache.cordova.Capture"/>
50     <plugin name="Battery" value="org.apache.cordova.BatteryListener"/>
51     <plugin name="SplashScreen" value="org.apache.cordova.SplashScreen"/>
52     <plugin name="MyPlugin" value="com.coldfumonkeh.MyPlugin"/>
53 </plugins>
54 </cordova>
```

19. Let's test our application and make sure we get a response from the plugin. Right-click on the main project folder and select **Run As|Android Application** from the context menu to launch the application on the emulator. This is shown in the following screenshot:
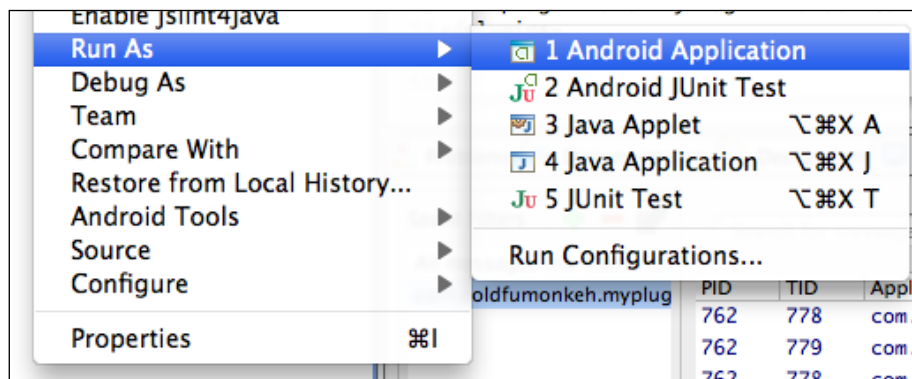
20. After the application has launched on the virtual device, it should look something like the following screenshot, and you'll see the message displayed using an alert notification:
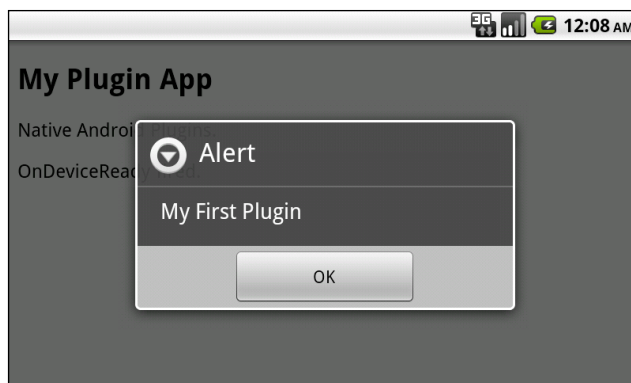
How it works...

We created a JavaScript file to hold our client-side method calls and to handle success or error callbacks. We then created the native portion of the plugin, which extended the `org.apache.cordova.api.Plugin` class, thereby providing us with access and the ability to communicate between the JavaScript and native code.

Our JavaScript code called the `cordova.exec` method to invoke methods within our custom class:

```
cordova.exec(
        successCallback,
        errorCallback,
        'MyPlugin',
        'greeting',
        [message]
    );
```

The `exec()` method accepts the following parameters:

- ▶ `success`: The success callback function, which will run following a successful response from the plugin.
- ▶ `error`: The error callback function, which will run following an unsuccessful operation and response from the plugin.
- ▶ `service`: This is mapped to the name of the native plugin class.
- ▶ `action`: The name of the action within the custom class to call and invoke.
- ▶ `arguments`: This is an optional parameter that allows you to send through data in an array, which can then be processed and used by the custom class.

The `cordova.exec` JavaScript method then invokes the `execute` function on the corresponding native plugin class.

Within our native class, we first checked that the action being sent through matched the predefined value. We then checked to make sure that we had sent through a valid message value in the data JSON array, and that it wasn't a null or a zero-length string.

We defined the `PluginResult` object and set the response status according to the success or failure of our data validation checks.

Android WebView renders HTML content in Cordova applications, and uses the **WebView API** to allow communication between the native and JavaScript class definitions.
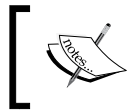
> You can find out more about the WebView API on the Android Developer documentation pages, available at: `http://developer.android.com/reference/android/webkit/WebView.html`.

Following the communication from the native class, our JavaScript method was then able to process the response with either the success or error callback functions, depending on the status.

> For more information on developing native Android Cordova plugins, check out the official documentation, available at: `http://docs.phonegap.com/en/2.0.0/guide_plugin-development_android_index.md.html#Developing%20a%20Plugin%20on%20Android`.

Extending your Cordova iOS application with a native plugin

If you are building mobile applications for the iOS platform, you can add extra functionality to your project with the help of custom native plugins and with the help of some Objective-C code and the Cordova framework.

How to do it...

In this recipe we will create a native iOS plugin for our Cordova application, using Xcode as our development environment.

1. Create a new Cordova project using the command line tools available with version 2.0.0. We explore the command line features available in more detail in the *Use the command line to create a new iOS Cordova application* recipe in *Chapter 9, Development Tools and Testing.* In this command we are creating a new project called `MyPlugin` within a `Development_Tools` directory on my machine:

    ```
    phonegap-2.0.0/lib/ios/bin/create ~/Development_Tools/
    myPlugin_ios com.coldfumonkeh.myplugin MyPlugin
    ```

2. Open the project in Xcode by clicking **File| Open** in the main menu and selecting the `MyPlugin.xcodeproj` file.

3. Select **File| New File** from the main menu and select the option to create an empty file, as shown in the following screenshot:
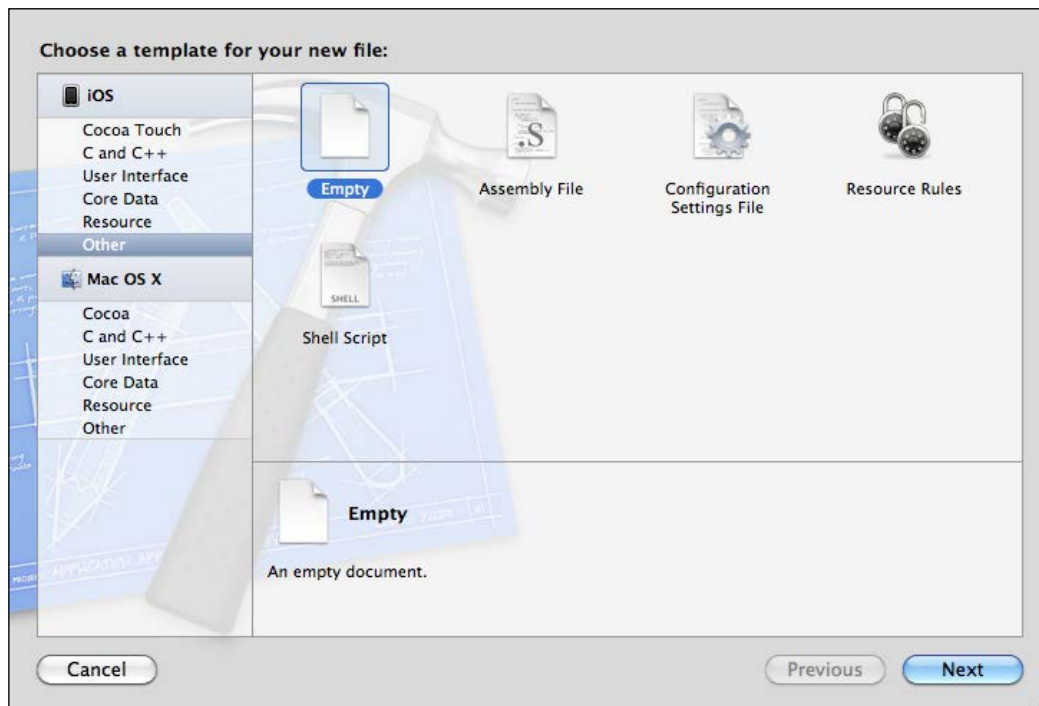
4. Call the file `MyPlugin.js` and save it in the `www/js` directory within the current project, as shown in the following screenshot:
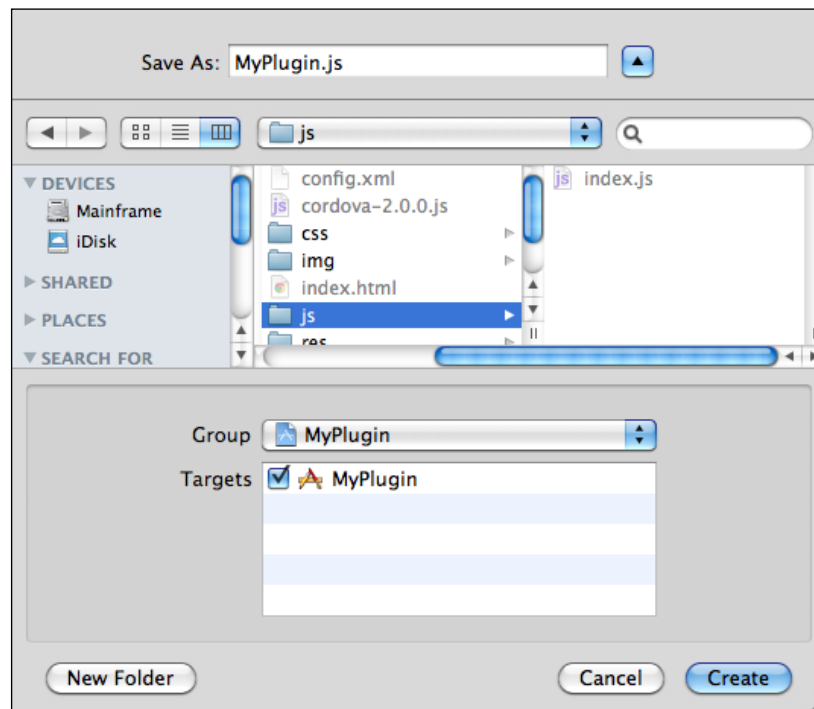
5. Place the following code at the top of the new JavaScript file:

```
varMyPlugin = function(){};

MyPlugin.prototype.greeting = function(
message, successCallback, errorCallback){
   cordova.exec(
      successCallback,
      errorCallback,
      'MyPlugin',
      'greeting',
      [message]
   );
};
```

6. Here, we define the class name for our JavaScript plugin, and then create the first method we will invoke called `greeting`. It's worth noting that the previous JavaScript code is identical to that used in the earlier recipe to create a custom plugin for an Android application.

7. Add the following code to the bottom of the file to load the `MyPlugin` class into the `window.plugins` object:

```
if(!window.plugins) {
    window.plugins = {};
}
if (!window.plugins.MyPlugin) {
    window.plugins.MyPlugin = newMyPlugin();
}
```

8. Right-click on the `Plugins` directory in the left-hand side Xcode panel and select **New File** from the context menu. Select **Cocoa Touch|Objective-C class** from the file template selection and click on **Next** to proceed. This is shown in the following screenshot:
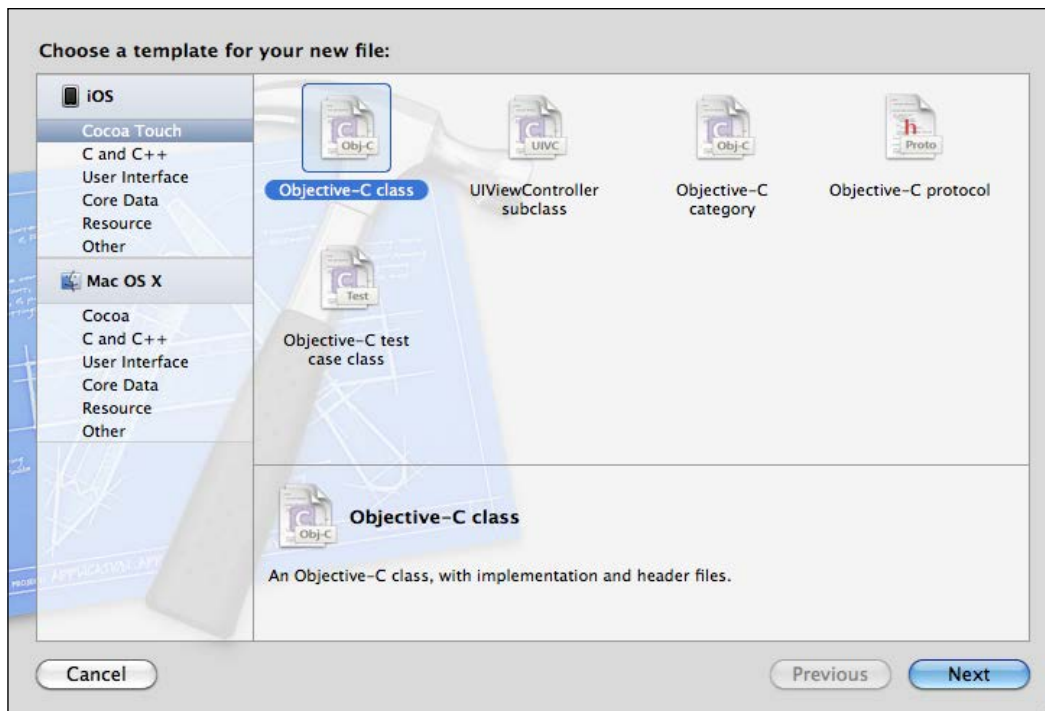
9. Enter `MyPlugin` as the class name, and set the subclass value to `CDVPlugin`, as shown in the following screenshot:
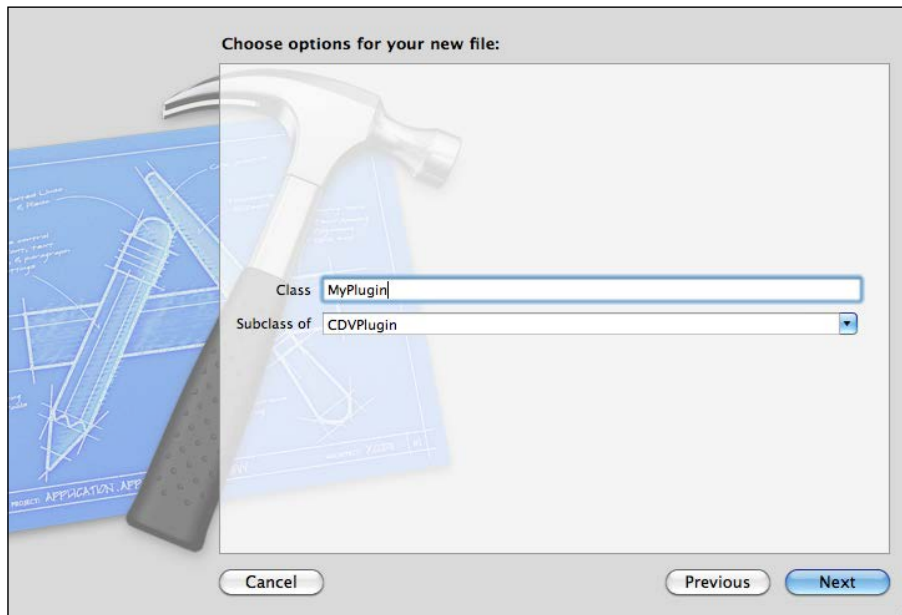
> All custom Cordova iOS plugins must extend the `CDVPlugin` class, as this encapsulates all of the logic required to handle communication between the native code and the JavaScript implementation, using the PhoneGap API as the bridge.

10. Save the files within the `Plugins` directory and click on **Create** to complete this step. The following screenshot illustrates the same:
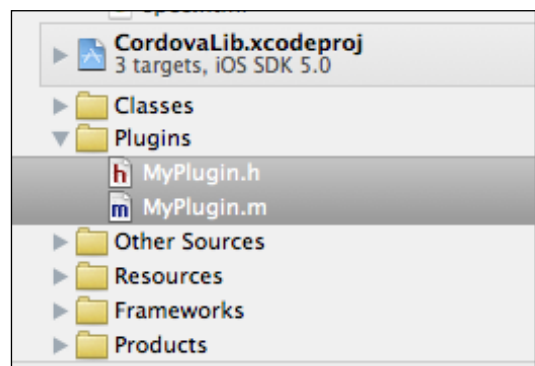
11. Open `MyPlugin.h` into the main editor window and replace the default content with the following code. Here we define the `greeting` method that consists of two parameters. The first is an `NSMutableArray` of arguments that will be passed from the JavaScript implementation. The second is an `NSMutableDictionary` class to allow us to map key and value pairs that we may send through.

```
#import <Cordova/CDV.h>

@interface MyPlugin : CDVPlugin

 (void) greeting:(NSMutableArray*)arguments
withDict:(NSMutableDictionary*)options;

@end
```

12. Open `MyPlugin.m` and replace the default content with the following code, in which we define the `greeting()` method and import the relevant header files.

```
#import "MyPlugin.h"
#import <Cordova/CDVPluginResult.h>

@implementation MyPlugin

- (void) greeting:(NSMutableArray*)arguments
withDict:(NSMutableDictionary*)options
{


}

@end
```

13. Place the following code within the `greeting()` function definition. We are setting the `callbackId` variable, using the first item in the `argument` array. We'll also set default values for our `pluginResult` and `javascript` variables, as shown in the following code snippet:

```
NSString* callbackId = [arguments objectAtIndex:0];

CDVPluginResult* pluginResult = nil;
NSString* javaScript = nil;
```

14. We want to obtain the string variable, that we are sending through, from the JavaScript call, which we can retrieve from the `arguments` array, referencing it as the second index item in the collection.

15. We'll then check the validity of the `result` value to make sure that we have not sent through a nil or zero-length string. If our check passes, we'll return a `pluginResult` object with the status set to `OK`, and return the `result` string back to our JavaScript success callback function. If the string is not valid, we will return a `pluginResult` object with an `ERROR` status.

16. We'll also wrap the entire logic within a `try/catch` block to handle any errors that we may encounter. If this happens, we will return a `pluginResult` object, setting the status to `JSON_EXCEPTION` and an error message.

17. Finally, we write the JavaScript back to the application WebView, which will process the response using either success or error callback function we define, based upon the status.

```
@try {

  NSString* result = [arguments objectAtIndex:1];

  if (result != nil && [result length] > 0) {

   pluginResult = [CDVPluginResultresultWithStatus:CDVCommand
   Status_OK messageAsString:result];

   javaScript = [pluginResult toSuccessCallbackString:
callback    Id];

    } else {

   pluginResult = [CDVPluginResult resultWithStatus:CDVCommand
   Status_ERROR];

   javaScript = [pluginResult toErrorCallbackString:callbackId];

  }

} @catch (NSException* exception) {

  pluginResult = [CDVPluginResult resultWithStatus:CDVCommandStat
  us_JSON_EXCEPTION messageAsString:[exception reason]];

  javaScript = [pluginResult toErrorCallbackString:callbackId];

}

[self writeJavascript:javaScript];
```

18. Open `index.html` in Xcode and add the following reference to the plugin JavaScript file below the `index.js` script reference:

```
<script type="text/javascript"
src="cordova-2.0.0.js"></script>
<script type="text/javascript" src="js/index.js"></script>

<script type="text/javascript"
src="js/MyPlugin.js"></script>

<script type="text/javascript">
app.initialize();
</script>
```

19. Let's add the call to our plugin's JavaScript method. Open `index.js`, which initializes the `deviceready` event listener. Amend the `deviceready` function to include a request to our `greeting` method. We'll send through a message to return, and specify the success callback function, which will display the returned value in an alert notification window.

```
deviceready: function() {

    // note that this is an event handler so the scope is that
    of the event
    // so we need to call app.report(), and not this.report()

    app.report('deviceready');

window.plugins.MyPlugin.greeting(
  "My First iOS Cordova Plugin",
  function(echoValue) {
   alert(echoValue);
  }
 );

}
```

20. Finally, we need to map a reference to the plugin so that Cordova can identify our custom plugin. Expand the `Resources` folder in the left-hand side project navigator panel in Xcode and right-click on `Cordova.plist`. Select **Open As|Source Code** to open the file in the main editor window, shown in the following screenshot:
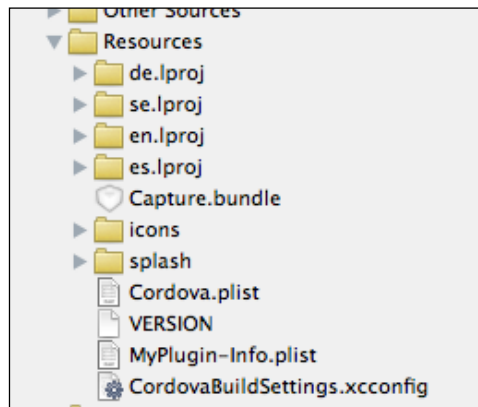
21. Locate the `Plugins` section, and add a map to your custom plugin. Set both the `key` and `string` node values to `MyPlugin` and save the file.

```
<key>Plugins</key>
  <dict>
    <key>MyPlugin</key>
    <string>MyPlugin</string>
    <key>Device</key>
    <string>CDVDevice</string>
    <key>Logger</key>
    <string>CDVLogger</string>
```

22. Click on the **Run** button in the top-left section of the Xcode window to build and launch the application on the device simulator. You should see something similar to following screenshot:
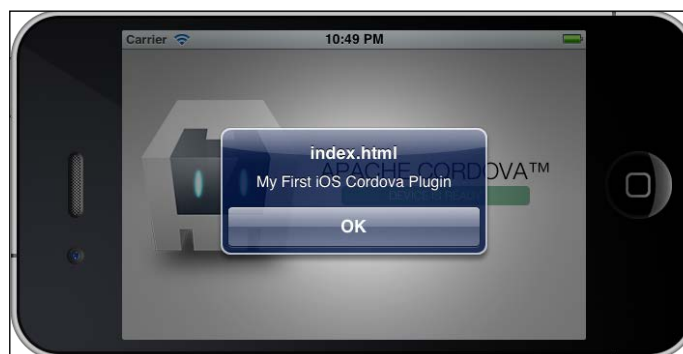
How it works...

We created a JavaScript file to hold our client-side method calls and to handle success or error callbacks. We then created the native portion of the plugin, which implemented the `CDVPlugin` subclass, thereby providing us with access and the ability to communicate between the JavaScript and native code.

Our JavaScript code called the `cordova.exec` method to invoke methods within our custom class:

```
cordova.exec(
        successCallback,
        errorCallback,
        'MyPlugin',
        'greeting',
        [message]
    );
```

The `exec()` method accepts the following parameters:

- ▸ `success`: The success callback function, which will run following a successful response from the plugin.
- ▸ `error`: The error callback function, which will run following an unsuccessful operation and response from the plugin.
- ▸ `service`: This is mapped to the name of the native plugin class.
- ▸ `action`: The action/method name within the custom class to call and invoke.
- ▸ `arguments`: This is an optional parameter that allows you to send through data in an array, which can then be processed and used by the custom class.

Within our native class, we processed the request to ensure we had sent through a value in the `arguments` parameter, and handled the response accordingly using the `pluginResult` object and setting the status.

Finally, we communicated the response back to the JavaScript implementation using the `writeJavascript` function. Our JavaScript method was then able to process the response with either the success or error callback functions, depending on the status.
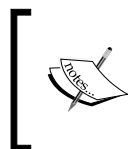
> For more information on developing native iOS Cordova plugins, check out the official documentation available at: `http://docs.phonegap.com/en/2.0.0/guide_plugin-development_ios_index.md.html#Developing%20a%20Plugin%20on%20iOS.`

The plugin repository

Building native custom plugins involves writing a little native code that relates to your chosen device's platform and operating system.

How to do it...

In this recipe we will have a brief look at the PhoneGap plugin repository.

1. Visit `https://github.com/phonegap/phonegap-plugins` in your browser, as shown in the following screenshot:
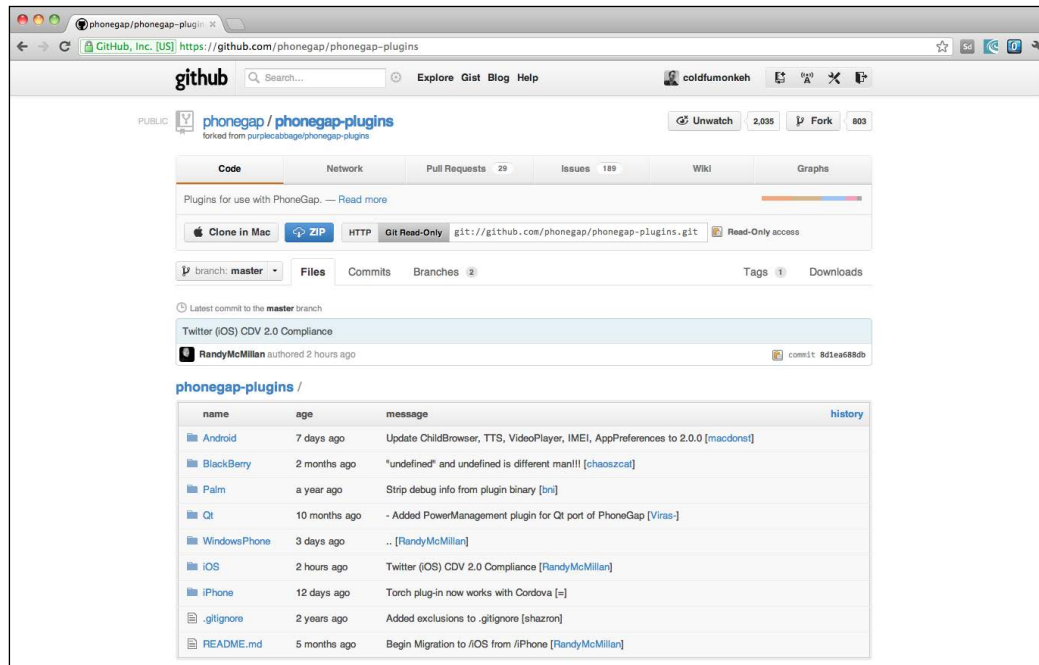
2. Each plugin has been separated into platform-specific categories. Have a look through the repository structure and the different platform categories to see what native custom plugins are available to use for each platform.

3. Browse through some source code to see how other developers have built their plugins.

4. Download individual plugins or clone the entire repository onto your local development machine.

5. Get inspired!

How it works...

This recipe is really a testament to the passion of the Cordova/PhoneGap development community. All of the plugins available in the repository are open source and available for you to learn from, download, and implement into your own projects to enhance your applications and add some amazing features.

The essential element to take away from this recipe is that the possibilities for creating plugins are almost endless with regards to what you can achieve and what features you can access as an extension of the PhoneGap API.

And of course the most important point here is to have *fun* building your Cordova applications!

9
Development Tools and Testing

In this chapter, we will cover:

- ▶ Downloading Cordova
- ▶ Using the command line to create a new iOS Cordova application
- ▶ Using Xcode templates for iOS to develop Cordova applications
- ▶ Using Eclipse to develop Android Cordova applications
- ▶ Controlling your Android Virtual Device
- ▶ Using Adobe Dreamweaver to develop Cordova applications
- ▶ Using the PhoneGap Build service

Introduction

To successfully create your Cordova applications, it is really important to set up the correct development environment; one that suits the requirements of your application, your personal development style, the tools and features that you may need to use, and one that is compatible with your local development machine's operating system.

In this chapter, we will investigate some of the options available to set up your local environment with development tools to assist you and help you to make development of your mobile applications even easier.

Downloading Cordova

Before we develop any Cordova applications, we first need to download a copy of the framework.

How to do it...

In this recipe we will download the Cordova framework to make sure we have the framework available to start local development:

1. First, head over to `http://phonegap.com/download`, which contains the latest release of the project as well as archived versions going back to version 1.2.0:

2. Download the latest version of the Cordova framework, currently 2.0.0. It's always best to use the latest stable version wherever possible. We'll also download version 1.9.0, which we'll use for the recipe, *Using Xcode templates for iOS to develop Cordova applications*, in this chapter.

3. The download will be in the `.zip` format. Extract the files to a preferred location on your local machine. Feel free to rename the folders to help you to easily identify which versions of the framework you have:
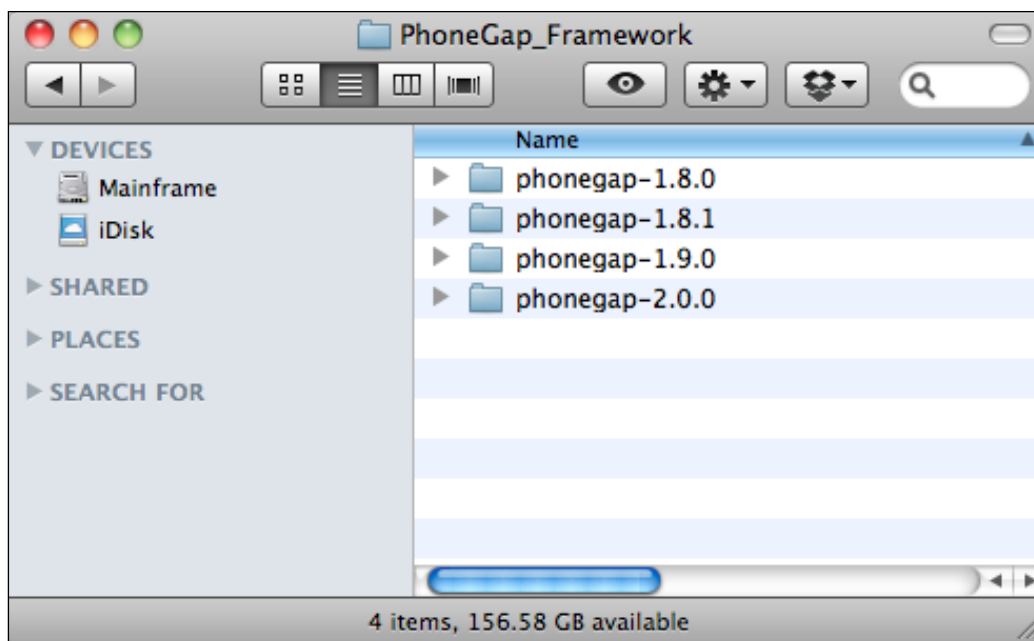
How it works...

We visited the official PhoneGap site to download the framework.

That's it! With the framework downloaded, let's start setting up our development environment.

Using the command line to create a new iOS Cordova project

A streamlined workflow is something that can benefit us all greatly by speeding up our processes and reducing the amount of manual work needed to complete a task.

How to do it...

In this recipe we will explore the command line tools available in Cordova 2.0.0 to create and run iOS applications from the Terminal application:

1. Navigate to the directory where you extracted the downloaded Cordova 2.0.0 framework files, and browse to the `libs/ios/` folder. Double-click on the `Cordova-2.0.0.dmg` package installer:
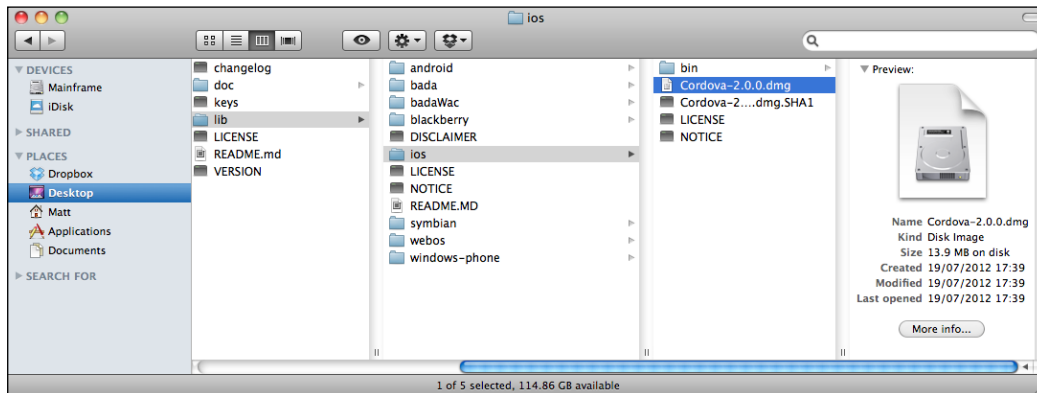
2. The installer will launch the installation wizard. Follow the short steps to install Cordova for iOS:
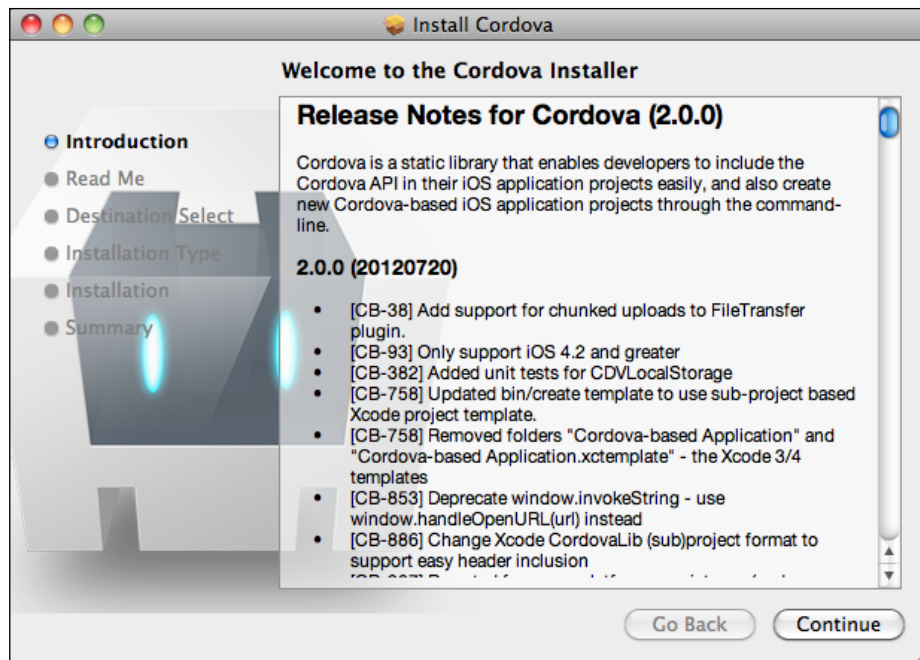
3. Once the installation process has completed, you now need to copy the `bin` directory to a location on your hard drive. This directory can be copied from the `.dmg` archive window, or from the downloaded directory.

4. Open a new Terminal application window. You can either type the path to the `bin` directory yourself, or you can drag the directory into the Terminal window to have the path populated for you.

5. With the path defined, we can now use the command line to create our new application. This is easily achieved using the `create` command:

    `$ ./create ~/Development_Tools/ios_commandline_app com. coldfumonkeh.commandme CommandMe`

243
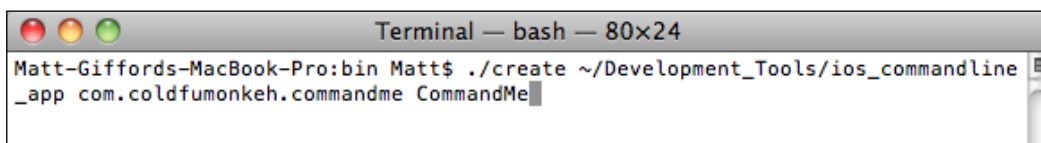
6. Once the command has been run, browse to the location of the project as specified in the previous step. Inside the directory you will now find that the Xcode project has been generated for you, as well as the `www` directory containing the sample `index.html` file and assets:
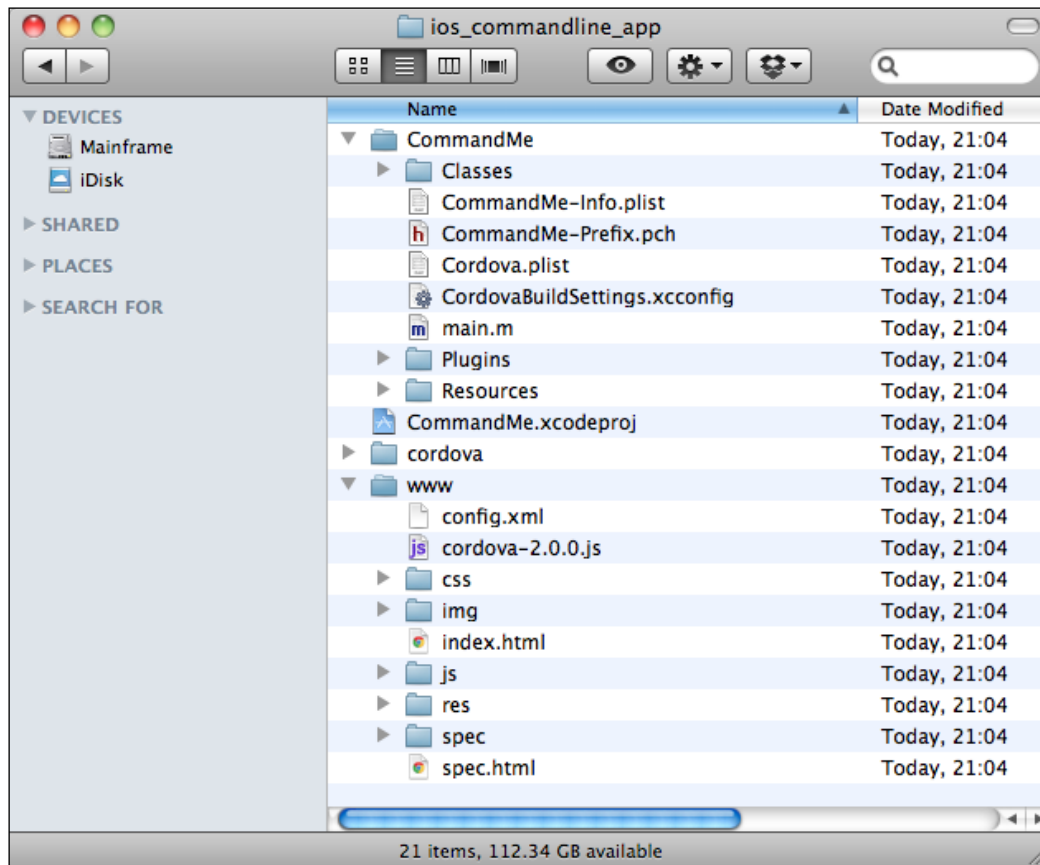
7. Open your Xcode application and select either the **Open Other...** button from the welcome screen, or **File | Open** from the menu and browse to the freshly-created project directory. Select the `.xcodeproj` file associated with the project and confirm to open the entire project into Xcode:
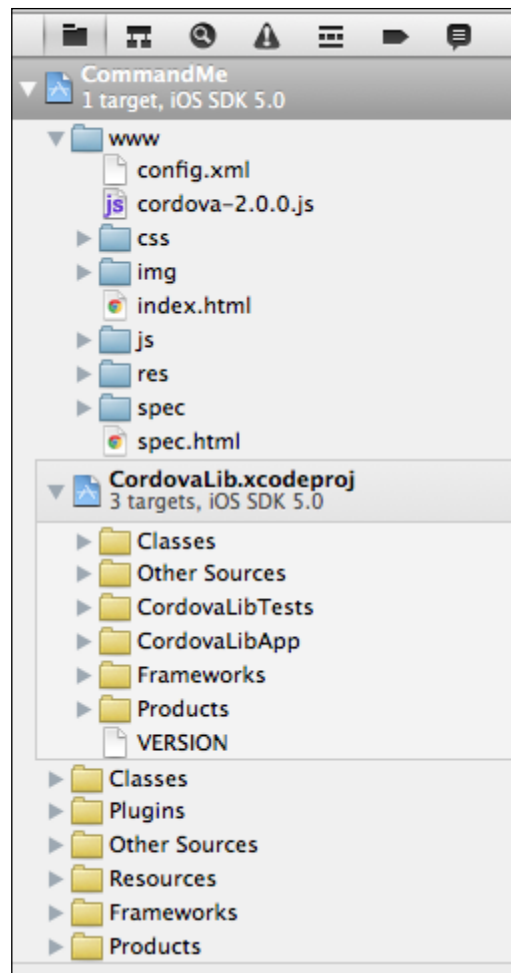
The generated project also includes a sub-project which links in `CordovaLib`. This means that you have access to the Cordova source code directly within your current project, which can be very useful for debugging.

8. If we choose to run the imported project via Xcode onto a simulated device, the output would look something like the following screenshot:

How it works...

The command line tools available in Cordova 2.0.0 greatly simplify the tasks involved in creating a project. With a simple command, we can generate a complete project, complete with the `www` directory linked at a project level, and ready to open up in Xcode to continue our development.

The `create` command accepts the following three parameters:

- ▸ The path to your iOS project
- ▸ The name of the package
- ▸ The project name

There's more...

The use of the command line is incredibly powerful and can allow developers to automate the creation of a project by running a very simple script. However, that's not the only tool available to use through the command line.

As a result of generating the project through the command line, Cordova 2.0.0 also creates another directory within the specified location, called `cordova`. This file contains some scripts that can simplify your workflow.

Running the application on the iOS Simulator

We can run the emulate script from the command line via the Terminal application to launch our application onto a simulated device.

For the emulation to successfully work on OS X, you will need to install a command-line utility tool called **ios-sim**, an open source project that launches the built application on the iOS Simulator. You can download ios-sim from the GitHub repository `https://github.com/phonegap/ios-sim`.

The `readme` file in the repository has short, detailed instructions explaining how to install ios-sim onto your machine.

Once installed, simply run the emulate script to load the application onto the iOS simulator. To do so, simply type in the path to the `cordova` directory within your project folder and run the `emulate` script:

```
$ cordova/emulate
```

When you run this command for the first time, the script will ascertain whether or not you currently have a successfully built version of the application. If not, it will ask you whether or not you would like the build process to happen.

Debugging your application

We can run the application on the simulator and gather debugging information in the Terminal window thanks to the `debug` script:

```
$ cordova/debug
```

This script will build your application and deploy it to the iOS Simulator, where the Terminal window will remain open to capture and display any console information as you test the application.

Using Xcode templates for iOS to develop Cordova applications

If you want to develop Cordova applications for the iOS platform, you have the option to set up your Xcode application to include PhoneGap templates. These offer quick, easy access to settings including the icons and package name, as well as device rotation options. It will even generate a sample application to help get you started.

The Xcode templates are not available in Cordova-2.0.0, so we'll use the previous version, 1.9.0 for this recipe.

How to do it...

To begin with, let's install the latest PhoneGap template for Xcode and iOS development:

1. Navigate to the directory where you extracted the PhoneGap files, and browse to the `libs/ios/` folder. Double-click on the `Cordova-1.9.0.dmg` package installer:
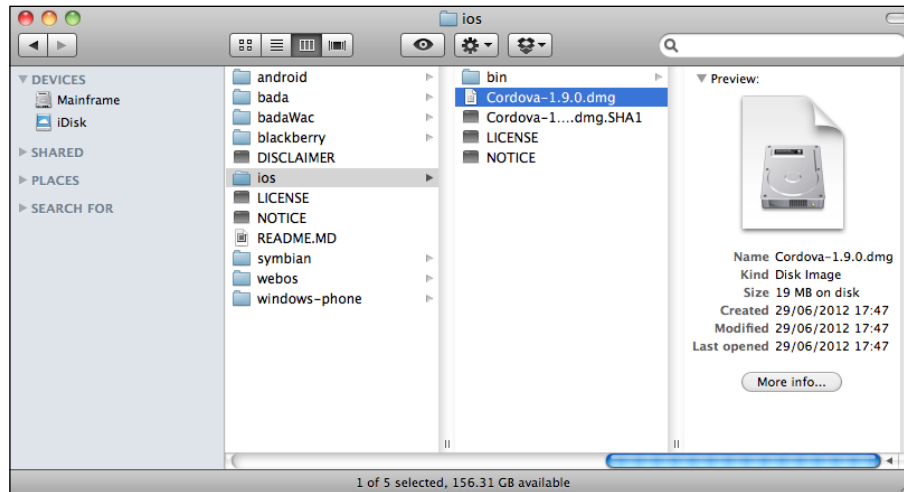
2. The installer will launch the installation wizard. Follow the short steps to install PhoneGap for iOS:
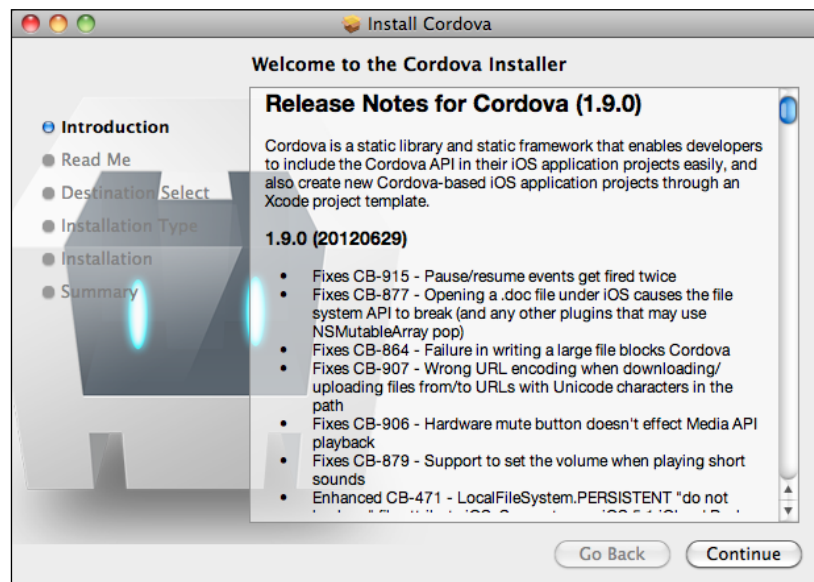
3. Once the installation process has completed, you will have a PhoneGap project template available to use in Xcode, which really helps to streamline your application development.

4. Let's now create a new project in Xcode. Click on **Create a new Xcode project** from the startup screen, or under the **File** | **New** | **New Project** menu item.

5. Select **Cordova-based application** from the available project templates, available under **iOS** | **Application** and click on **Next** to proceed, as shown in the following screenshot:
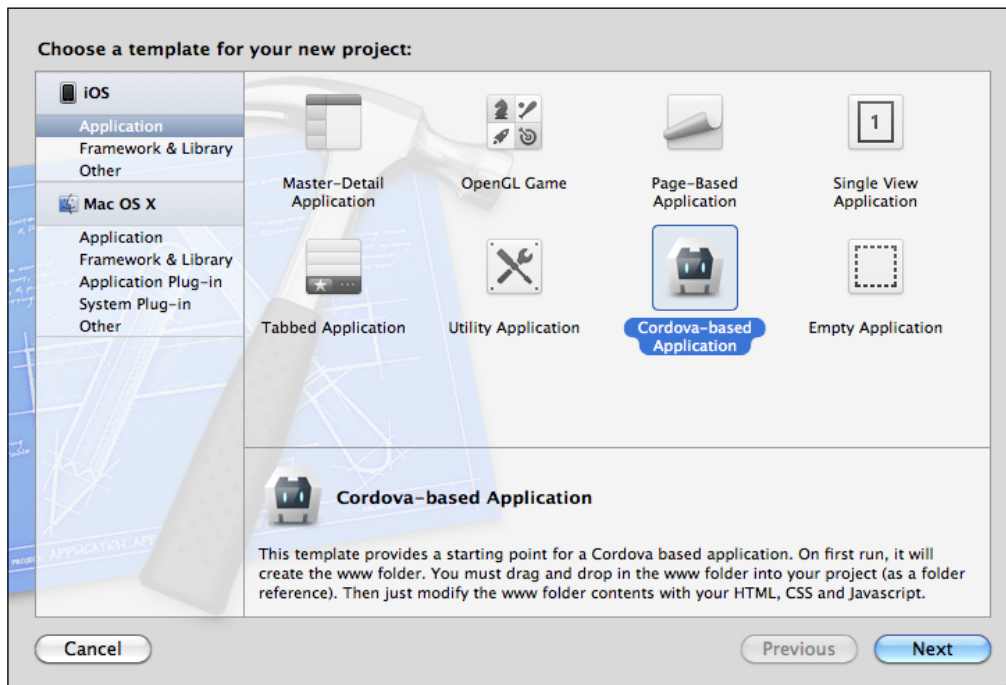
6. You will then be asked to enter in a name for the project and the **Company Identifier**. Make the name a unique descriptive of the application. The company identifier will create a unique **Bundle Identifier**.

> The generated **Bundle Identifier** should match the **App ID** value that you create for a unique application within the Apple Provisioning Portal.

7. Uncheck the **Use Automatic Reference Counting** option and click on **Next** to proceed:
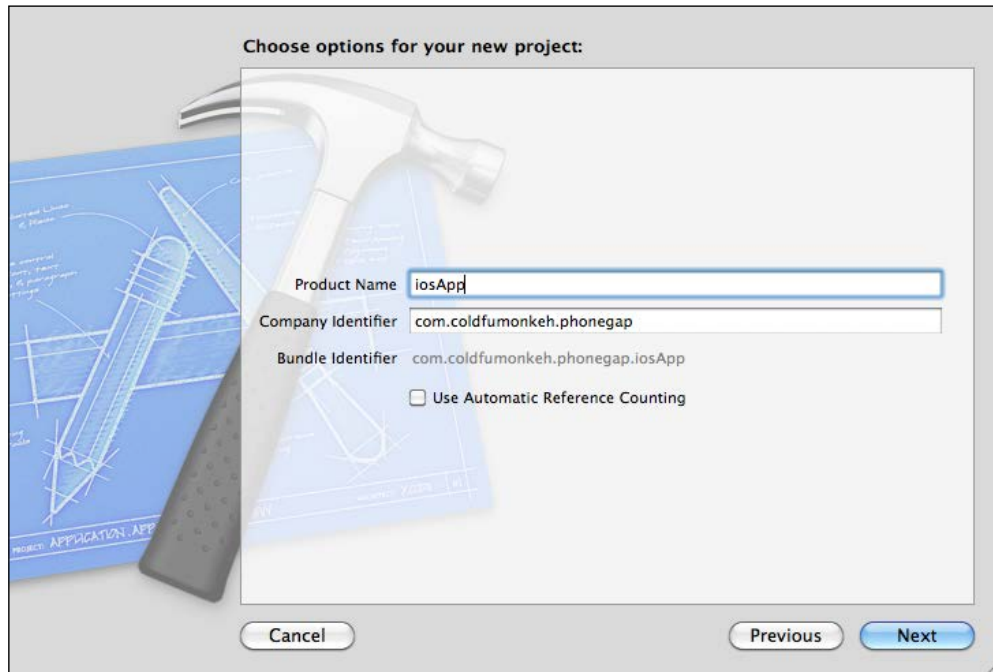
8. Select a directory on your filesystem into which the PhoneGap project will live. You also have the option for Xcode to create a Git source control repository for the project on your behalf. Click on **Create** to complete the project wizard:
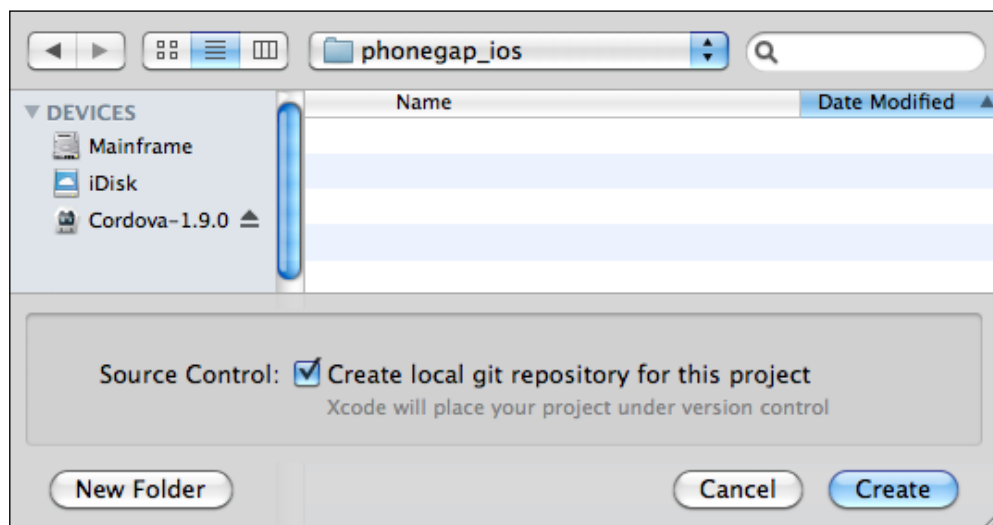
Having reached this step, the new project has been created using the PhoneGap template and should now be visible in Xcode. However, the project does not yet contain the www folder that will hold our JavaScript, CSS, and HTML files. This can be automatically generated for you, by Xcode, the first time you attempt to run the project.

9. Click on the **Run** button situated at the top left of Xcode to build and run the application. Alternatively, you can select **Product** | **Run** from the main menu or use the keyboard shortcut combination of *command + R*.

10. At this point, you may receive an error in Xcode that reads as follows: **_NSURLIsExcludedFromBackupKey, referenced from: -[CDVFile setMetadata:withDict:] in Cordova**.

11. This is a missing reference in Cordova 1.9.0, and is an easy issue to resolve. To remove this error, open up the `AppDelegate.m` file in the project and add the following highlighted code:

```
@implementation AppDelegate

@synthesize window, viewController;

NSString * const NSURLIsExcludedFromBackupKey
=@"NSURLIsExcludedFromBackupKey";

- (id) init
{
  /** If you need to do any extra app-specific initialization,
you can do it here
    *   -jm
    **/
    NSHTTPCookieStorage *cookieStorage = [NSHTTPCookieStorage
sharedHTTPCookieStorage];
     [cookieStorage setCookieAcceptPolicy:NSHTTPCookieAcceptPolicy
Always];

     [CDVURLProtocol registerURLProtocol];

     return [super init];
}
```

12. Save the file and run the project build again.

13. You will receive an error message stating **ERROR: Start Page at 'www/index.html' was not found**. This is expected, and has actually helped us out. The project has created the required `index.html` and the `www` directory for using a default template. We now just need to include the directory into the project.

14. Highlight the root of the project and right-click to access the context menu (or *Ctrl* + click) and select **Add Files To "__your project name__"**:
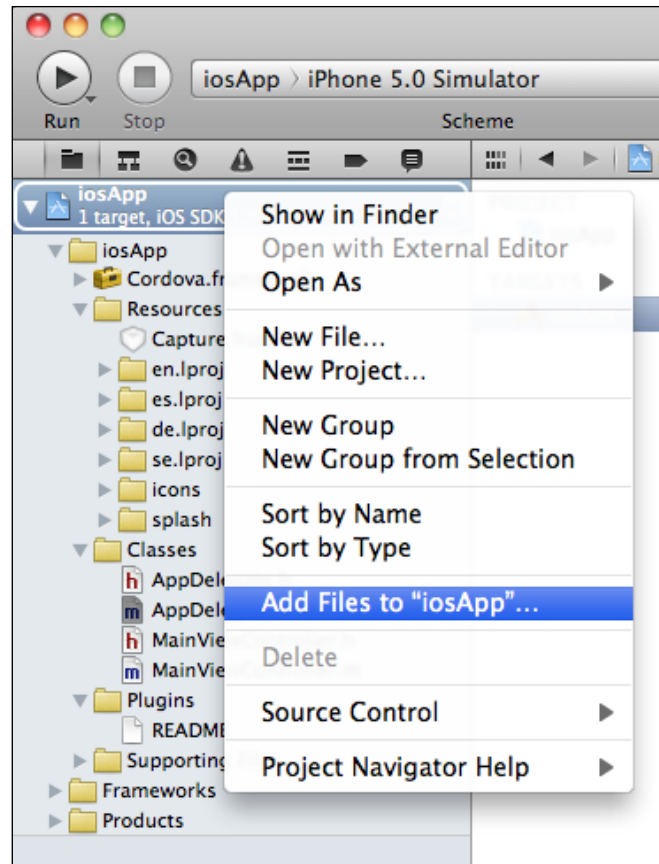
15. A dialog window will open. Expand the project directory to find the generated www folder. Select this folder directly as we only want to import that. Select **Create folder references for any added folders** and click on **Add** to continue:
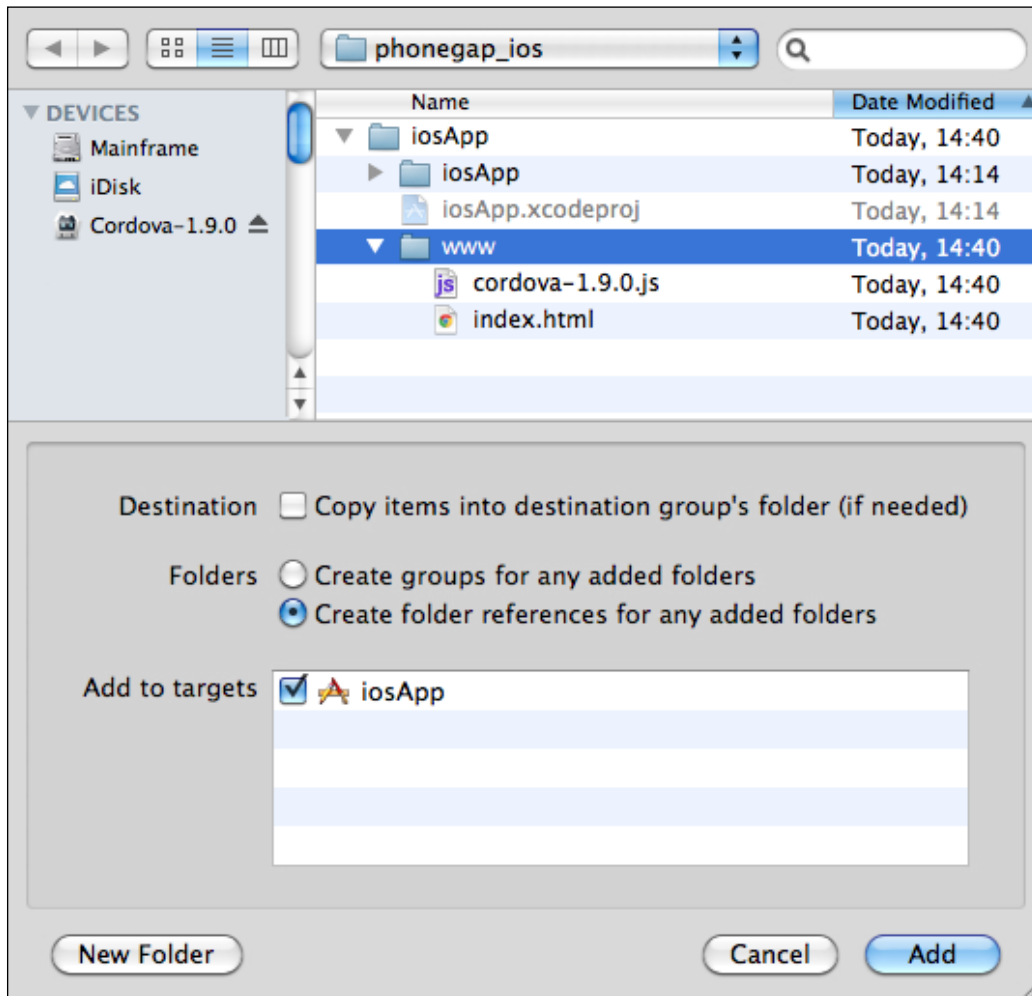
16. You can now see the referenced `www` directory in the project navigation window on the left-hand side in Xcode, as shown in the following screenshot:
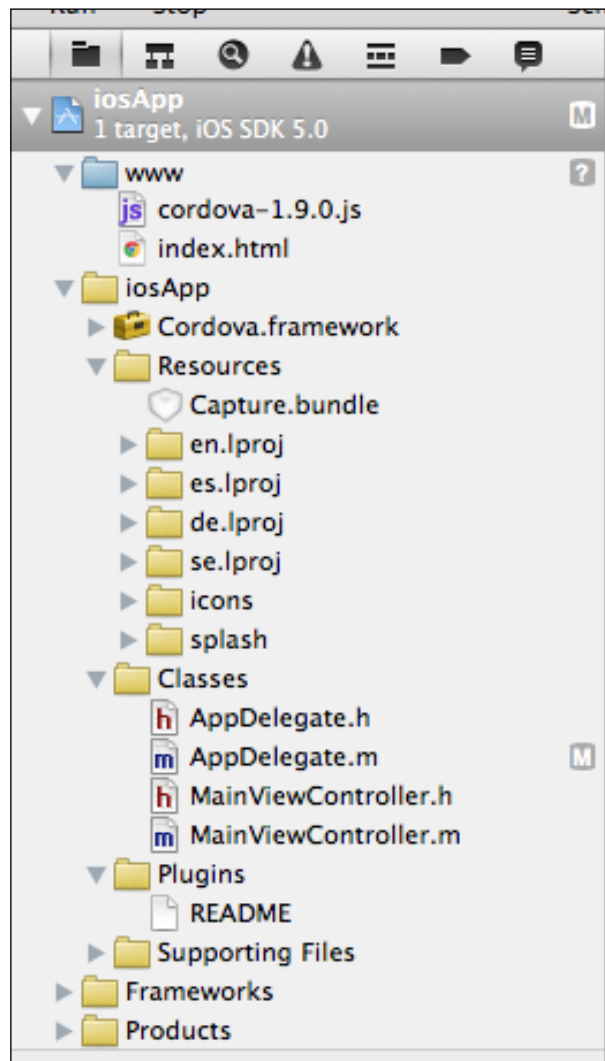
17. Select **Run** to launch the application once more, which should now load the application using the default `index.html` file generated by Cordova in the iOS Simulator:

If you can see the previous output running in the simulator, you have successfully created your first Cordova application in Xcode for iOS platforms using the templates!

How it works...

Running the Cordova-1.9.0 installation will add the template files to your Xcode project library and enhance the development application for PhoneGap applications.

There's more...

If you haven't seen or worked with Xcode before, it could be a little off-putting as there is a lot going on in the workspace. We'll have a brief introduction to the layout of the development environment and what tools we have at our disposal for Cordova application development.

Interface layout

The left-hand side of the window displays the project file navigation. From here you can easily access and view the files within your project. A single click on a file will open it up in the center of the Xcode window, whereas if you double-click on the file, it will open the file in a new window:
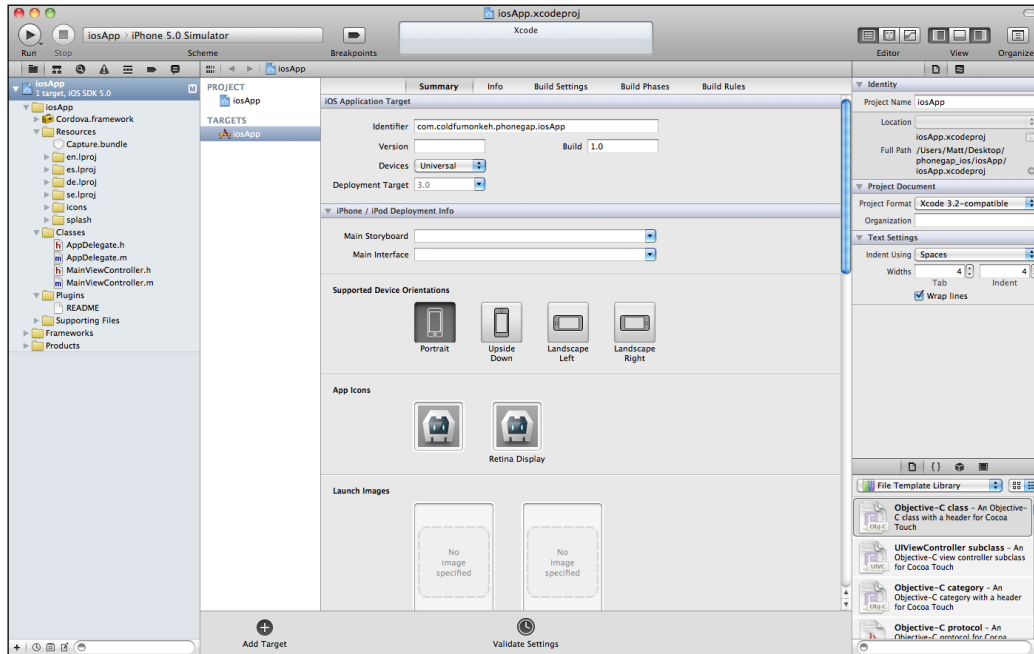
The center of the Xcode window shows the main content area, where any active files will be displayed for editing. By default it will display the project summary screen. This allows you to quickly and easily update various options for your application including the identifier, build and version numbers, and supported devices. You can choose if the application should support only iPad or iPhone devices, or if you want it to work on both.

You can also quickly set the preferences for application icons, splash screens, and supported device orientation.
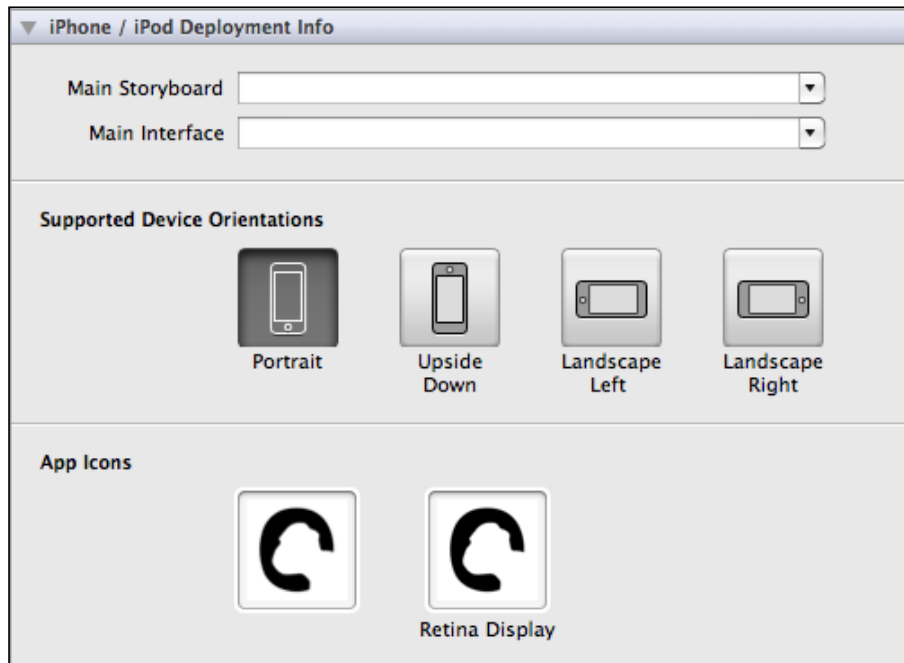
Schemes and target devices

The top of the Xcode application window contains a **Run** button, next to which is a **Scheme** menu. This menu allows you to change the target device or simulator on which to run the application. The **Scheme** menu is shown in the following screenshot:
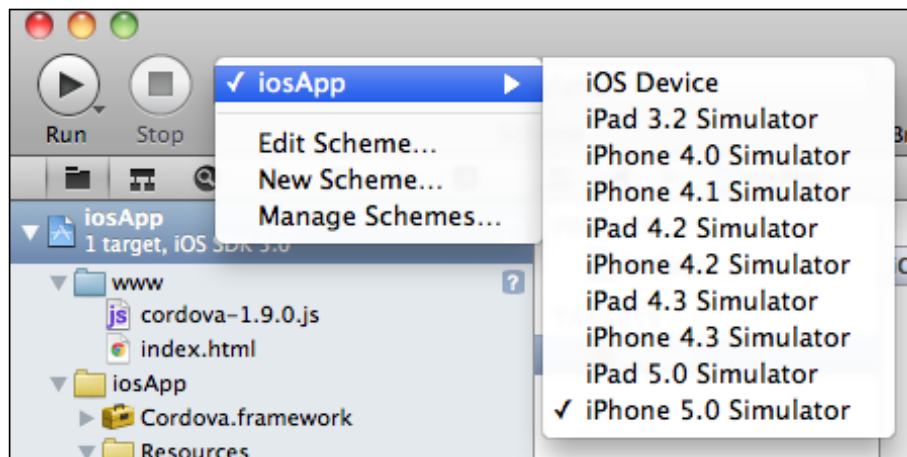
You can also edit the scheme to set specific options and arguments for the build process, including location simulation – perfect for developing geolocation applications:
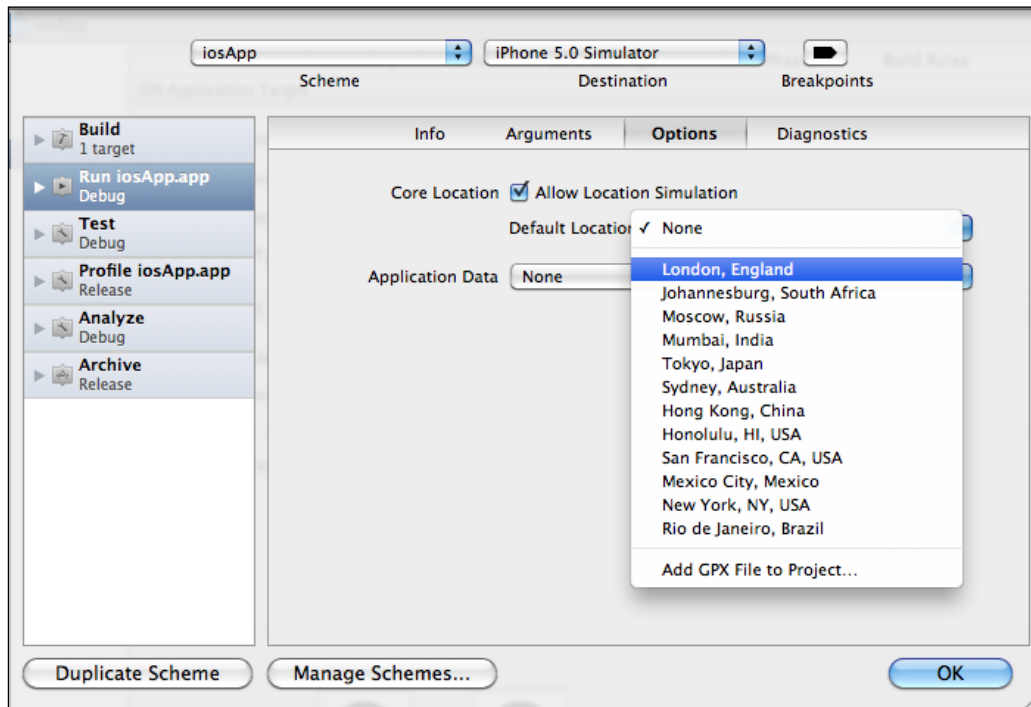
Using Eclipse to develop Android Cordova applications

The Eclipse IDE (that is, **Integrated Development Environment**) is highly extensible with a wide range of plugins available (many of them free) to help you create a development environment that suits your workflow and project needs.

Getting ready

Before we can start building our Android applications, we need to download the Eclipse IDE and ensure that we have a few required plugins installed:

▶ **Download Eclipse IDE**: Head over to `http://www.eclipse.org/downloads/` and download a copy of Eclipse Classic. The latest version at time of writing is 4.2. The minimum version you can use is Eclipse 3.4. The Eclipse downloads are as shown in the following screenshot:
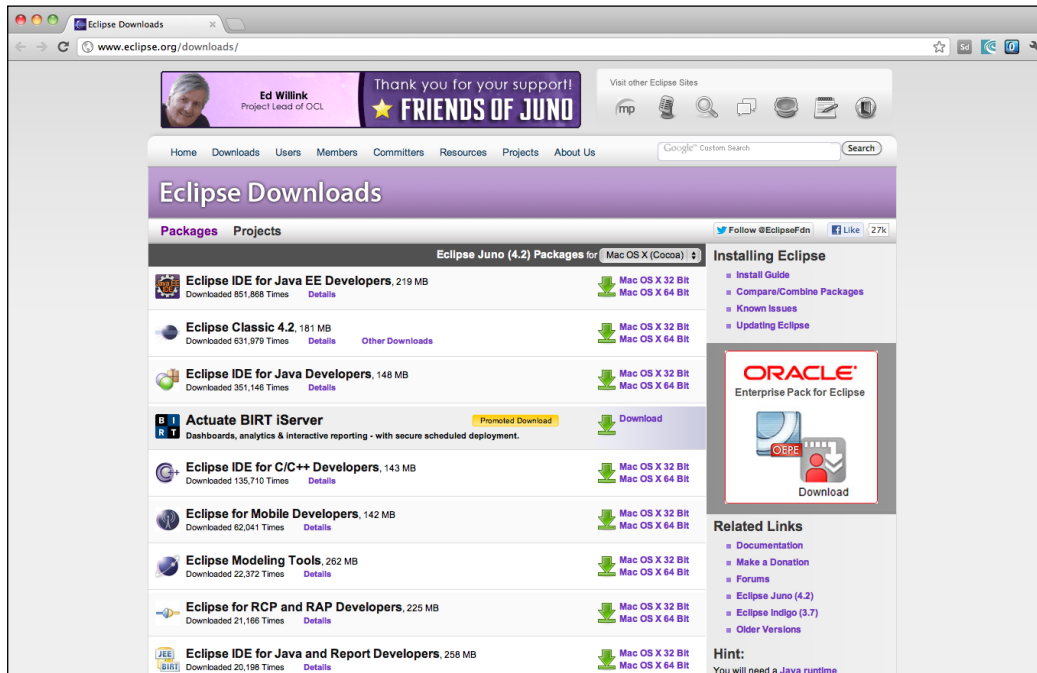
▶ **Download the Android SDK**: To successfully develop, debug, and test Android applications you will need to download and install a copy of the Android SDK to your local development machine.

    ❑ The SDK itself is not a complete development environment. It merely provides you with the core SDK tools, but we will use these to obtain the rest of the SDK packages to set up our development environment.

❑ Head over to `http://developer.android.com/sdk/index.html` and click on the download option to begin the process, as shown in the following screenshot:
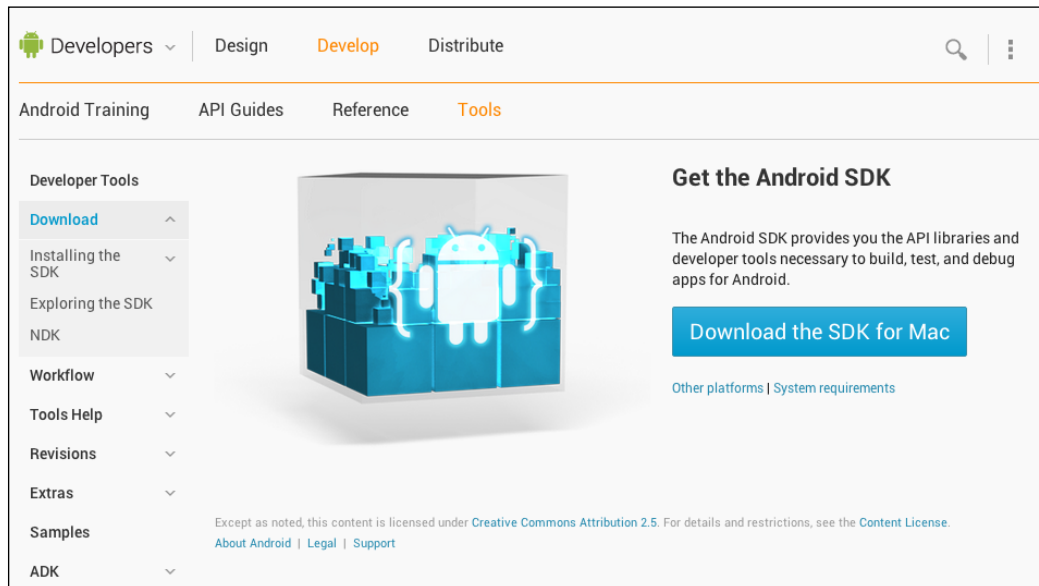
❑ The SDK will download as a `.zip` archive file. Extract this to the desired location on your local machine. The folder name will typically be android-sdk-<platform_version>, for example, `android-sdk-mac_x86`. You may want to rename it to `android-sdk` for simplicity sake, but this is not a requirement.

❑ The contents of the extracted archive will look something like the following screenshot:
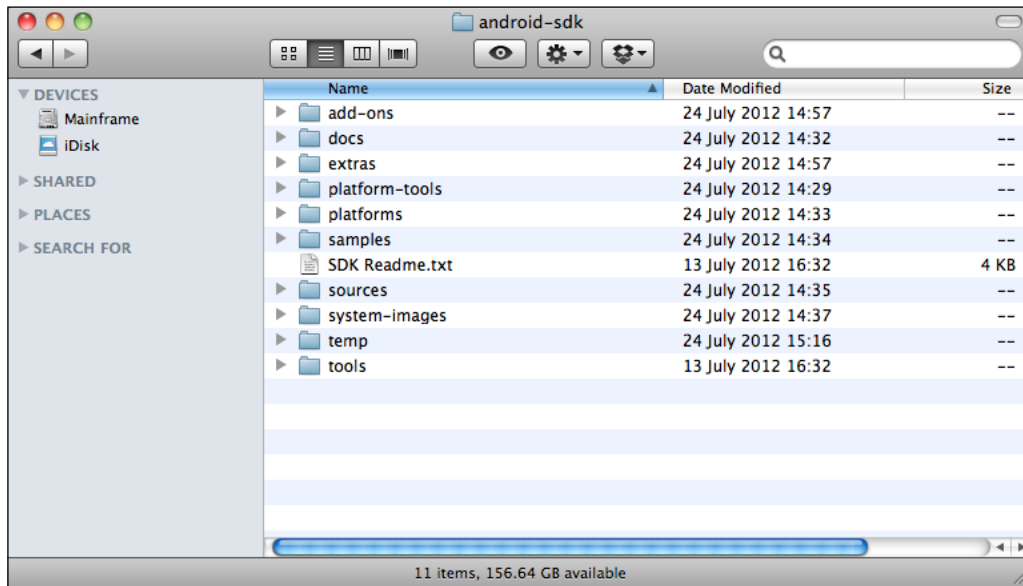
▶ **Install Eclipse plugin**: For Android mobile application development, and PhoneGap Android development, we can use the Eclipse Android Development Tools (ADT) plugin, which provides you with integrated tools to help you easily build an Android application project and much more.

❑ Open up the Eclipse IDE and select **Help | Install New Software...** from the main menu, as shown in the following screenshot:
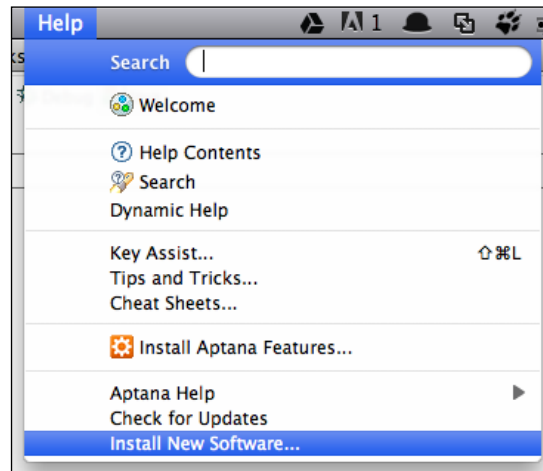
- ❑ Click on the **Add** button at the top-right corner to open up the install wizard window.

- ❑ Type `ADT Plugin` as the name for the new repository, and add the following URL for the plugin location: **https://dl-ssl.google.com/ android/eclipse/**.

- ❑ Click on **OK** to add the repository to the available software sites.
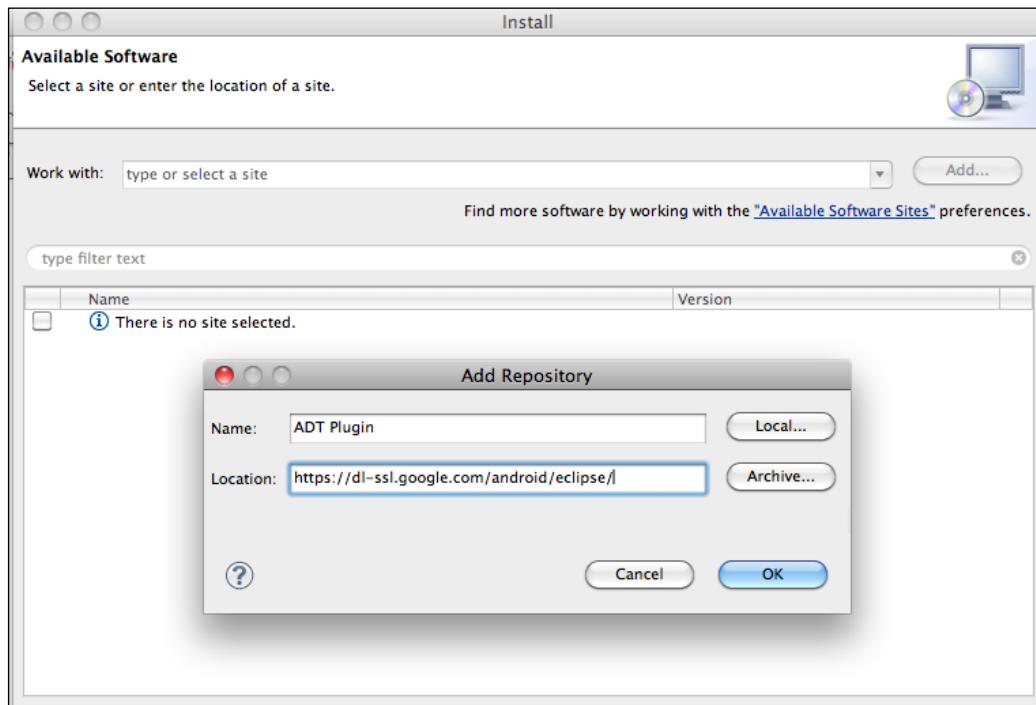
- ❑ With the **ADT Plugin** selected in the **Work with:** drop-down list, select the **Developer Tools** checkbox and click on **Next** to proceed. You will be presented with a small list of tools available to download. Click on **Next** once more to continue.

- ❑ Accept the license agreements that accompany the tools to be downloaded. If you're feeling adventurous you may wish to read them first. Click on **Finish** to begin the installation procedure.

- ❑ When the installation has finished, restart Eclipse to complete the process.

For more detailed instructions on installing the ADT Eclipse plugin, check out the official documentation available at: `http://developer.android.com/sdk/installing/installing-adt.html`.

❏ Having restarted Eclipse, select the **Preferences** menu and select the Android option. Use the **Browse** button to set the location of the Android SDK files downloaded earlier:

	Preferences		
	Android		⇦▾ ⇨▾ ▾

Android Preferences

SDK Location: `/Users/Matt/Development_Tools/android-sdk`   Browse...

Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.

Target Name	Vendor	Platform	AP...vel
Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8
DTS Add-On	KYOCERA Corporation	2.2	8
Real3D Add-On	LGE	2.2	8
GALAXY Tab Addon	Samsung Electronics Co., Ltd.	2.2	8
Android 4.0.3	Android Open Source Project	4.0.3	15
Google APIs	Google Inc.	4.0.3	15
HTC OpenSense SDK	HTC	4.0.3	15
Android 4.1	Android Open Source Project	4.1	16
Google APIs	Google Inc.	4.1	16

Restore Defaults   Apply

Cancel   OK

Sidebar list: General, Android, Ant, Aptana, Help, Install/Update, Java, Plug-in Development, Run/Debug, Team, XML

❏ Apply any changes made to the preferences to complete this step.

How to do it...

We will install a new Eclipse plugin from AppLaud to help simplify the creation of Android Cordova projects:

1. Select **Help | Install New Software...** from the main menu in Eclipse and add a new plugin repository. Set the name to `AppLaud Plugin` and the location to: `http://svn.codespot.com/a/eclipselabs.org/mobile-web-development-with-phonegap/tags/r1.2/download`.

2. Click on **OK** to add the repository to the available software sites, as shown in the following screenshot:
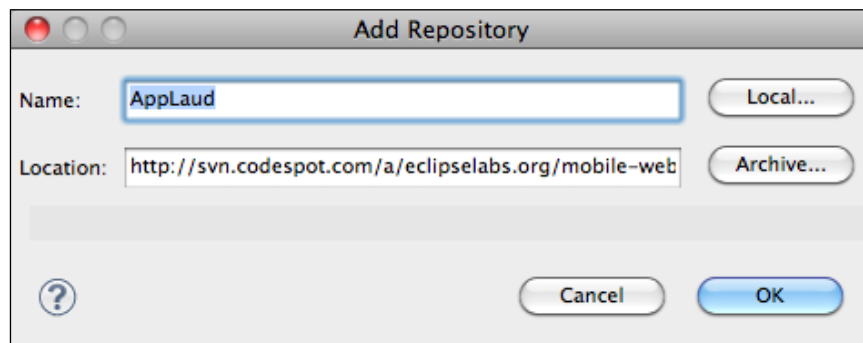
3. Select the entire **MDS AppLaud** library and click on **Next** to proceed:
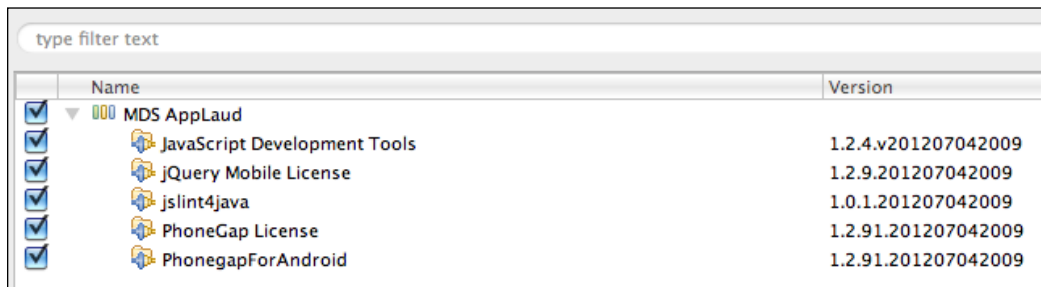
4. Agree to the plugin license and click on **Finish** to begin the installation. You will need to restart Eclipse to complete the procedure.

5. Once Eclipse has restarted, select **File** | **New Project** | **Project** from the main menu, and select **PhoneGap for Android Project**. You can also find this option within the **Phonegap Wizards** folder:
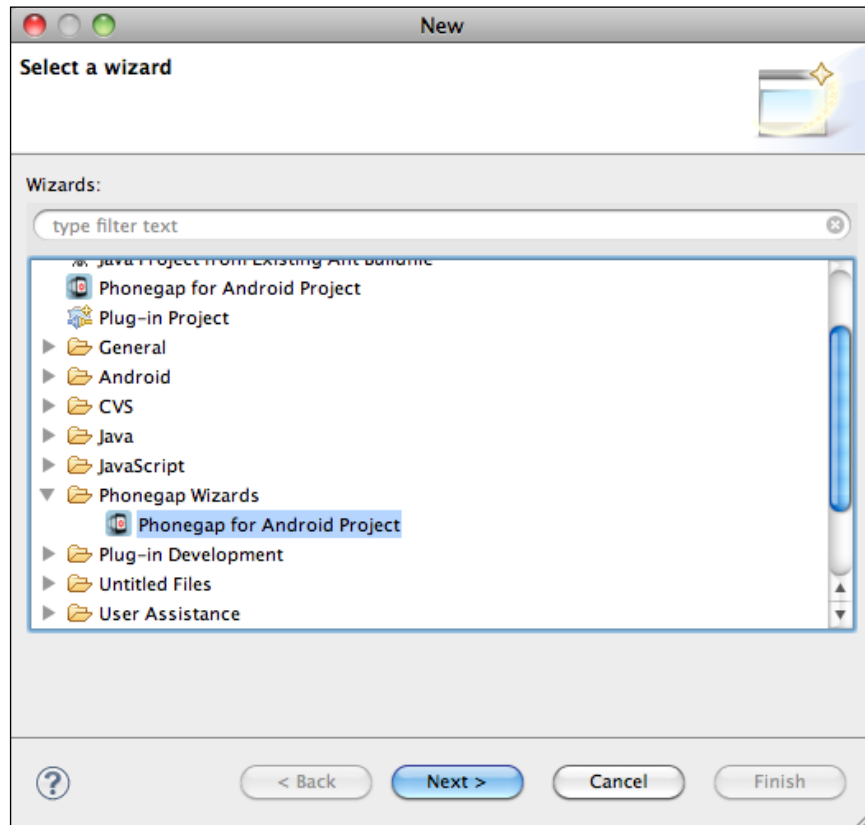
6. The project creation wizard will open a new window, and provide you with a series of options to make project creation easier. You can choose to use a version of the Cordova framework included within the plugin, which currently includes 1.4.1, 1.5.0, 1.6.0, and 1.9.0 respectively.

7. Let's use the latest version of the framework. Select the **Enter path to installed PhoneGap** option, and browse to the location of your local copy of Cordova 2.0.0:

```
PhoneGap Configuration
  ○ Use Built-in PhoneGap



  ● Enter path to installed PhoneGap

  s/Matt/Development_Tools/PhoneGap_Framework/phonegap-2.0.0    ( Browse... )
```
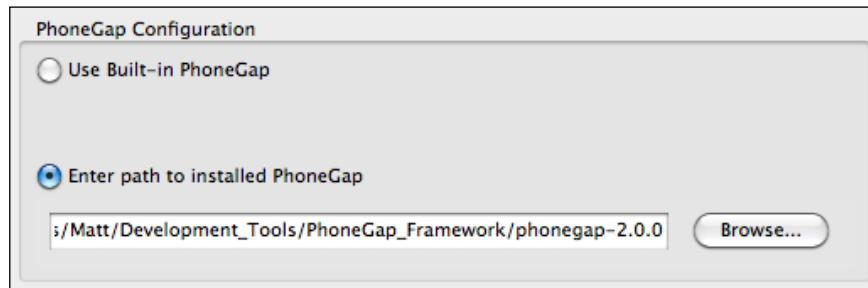
8. The AppLaud plugin also gives you the option to include a UI framework in the new project. Currently, the options are jQuery Mobile or Sencha Touch. Select the checkbox to **Include jQuery Mobile libraries in project** and accept the default values to use the built-in version of the library, as shown in the following screenshot:

```
jQuery Mobile
  ☑ Include jQuery Mobile libraries in project
  ● Use Built-in jQuery Mobile – version 1.1.0
  ○ Use separately installed version of jQuery Mobile



  ☐ Create project with jQuery Mobile UI demo
```
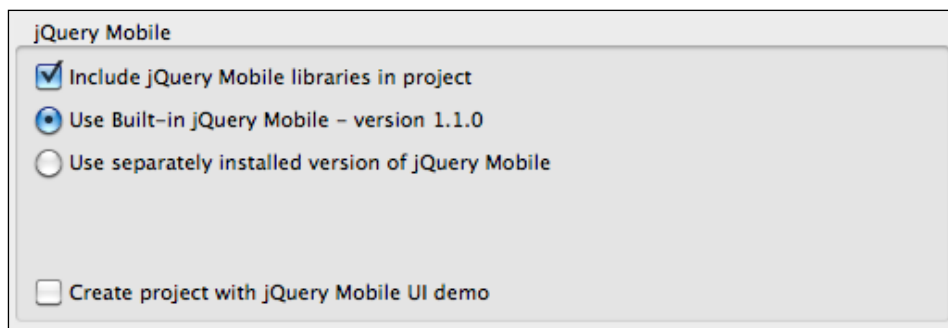
9. You have a few options to choose any predefined contents for your application, provided by the plugin, or to include code from a particular directory of your choosing. In this example, let's select the **PhoneGap API Example** option:

```
Project Contents
  ● PhoneGap API Example              with jQuery Mobile
  ○ Minimal PhoneGap project
  ○ Create project from specified source directory
```
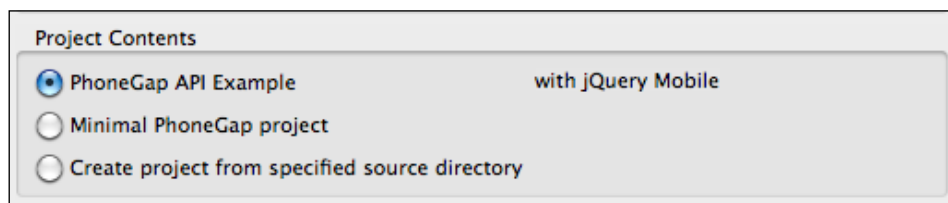
10. Click on **Next** to proceed. Enter a name for your project and select the desired project location, or accept the default location provided by Eclipse:
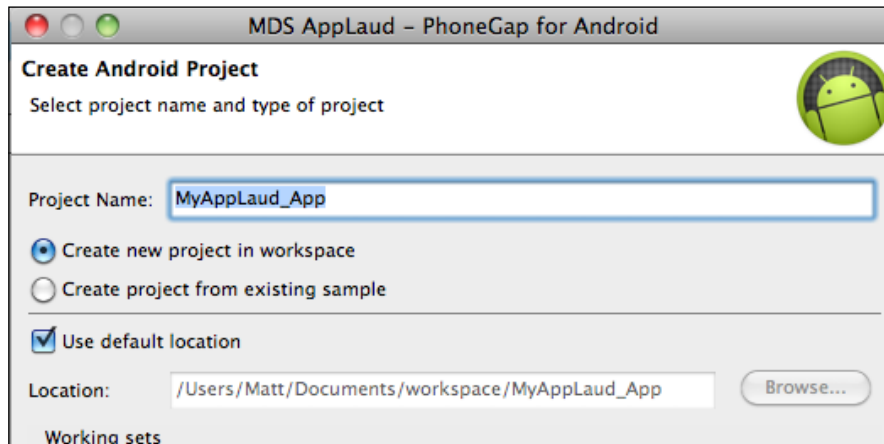
11. Choose an SDK build target and click on **Next** to proceed.

12. Enter a unique reverse-domain package name for your application and choose the minimum SDK version. Click on **Finish** to complete the project wizard:
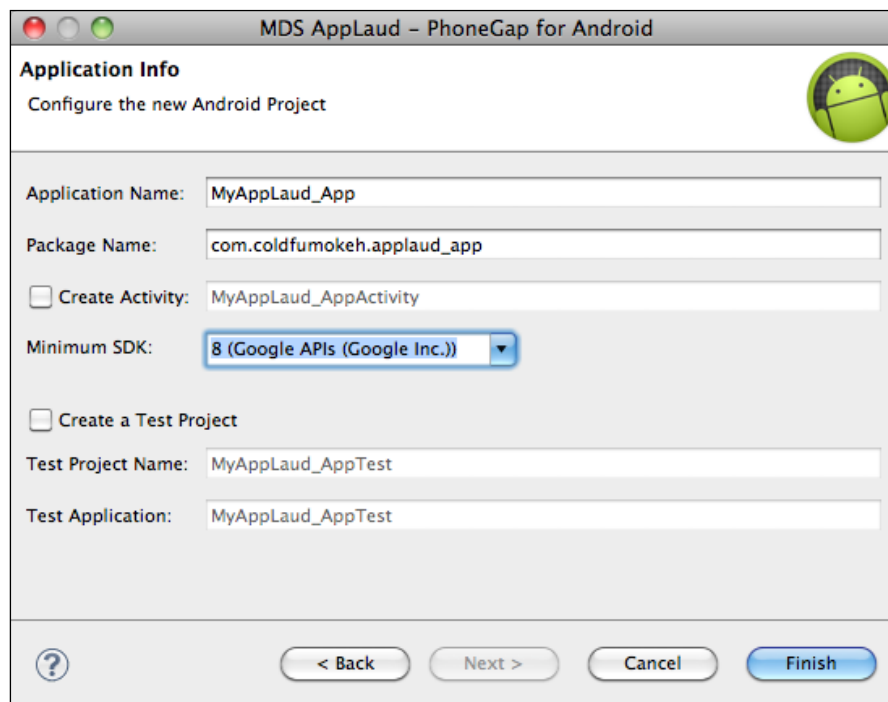
13. The new PhoneGap project will now be created and available in Eclipse. The project structure will include all required files, the Cordova library, and any framework files if selected – in this case, jQuery Mobile:
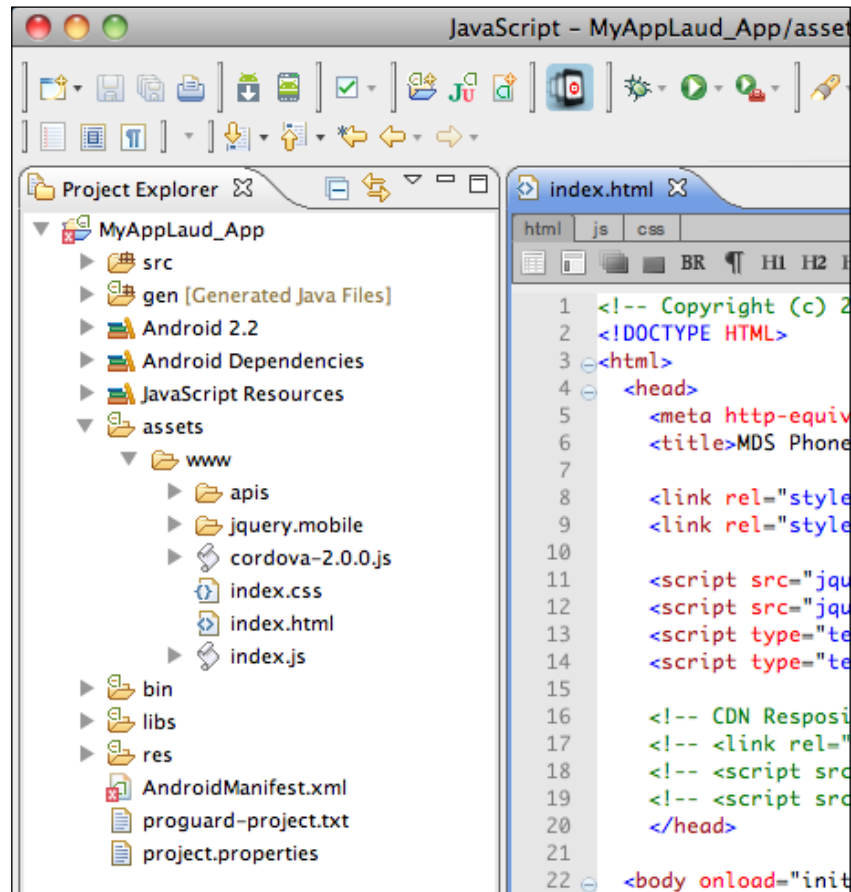
At this point, you may see an error in the **Problems** view in Eclipse, or you may experience this issue when attempting to run your application:

"No resource identifier found for attribute 'xlargeScreens' in package 'android'"

14. To resolve this issue, open up the `AndroidManifest.xml` file in Eclipse. Right-click on the file and choose **Open With | Text Editor** from the context menu. Find the `supports-screens` node at approximately line number nine, and remove the line that relates to the `xlargeScreens`, highlighted as follows:

```
<supports-screens
        android:largeScreens="true"
```

```
android:normalScreens="true"
android:smallScreens="true"
android:xlargeScreens="true"
android:resizeable="true"
android:anyDensity="true"
/>
```

15. Right-click on the main project folder in the left-hand side window and select **Run As | Android Application** from the context menu to launch the application on a device:
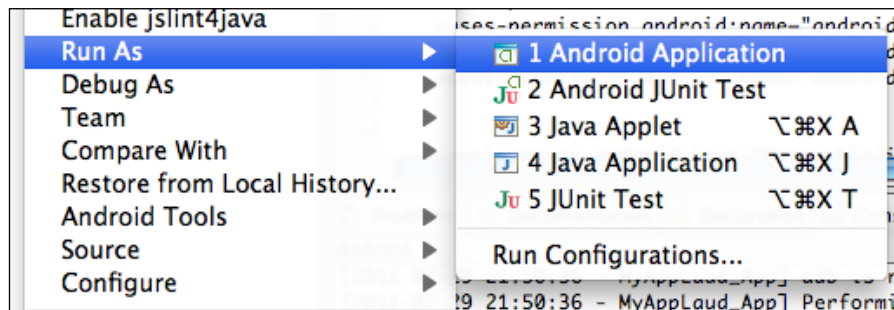
16. When you run the application on a device, the output should look something like the following screenshot:

17. Selecting an item from the menu within the application will take you to a page that demonstrates the specific Cordova API features.

269

How it works...

Installing the AppLaud plugin into Eclipse enhances the code editor and adds extra functionality into your development environment.

The plugin provides a number of options that can help you generate projects that may or may not contain framework libraries, and can generate application code to help you or leave you to create your own code from the very beginning.

> You can find out more about the AppLaud plugin from the official documentation available at: `http://www.mobiledevelopersolutions.com/`.

There's more...

The official Cordova documentation contains a section dedicated to getting started with the library. This includes a step-by-step guide to setting up your first Android application using the Eclipse IDE.

The AppLaud plugin handled many of the steps covered in the Cordova documentation automatically for you, but if you do not want to use a plugin or you'd be interested in reading an alternative method of setting up your Eclipse environment, check out the *Getting Started with Android* documentation using the link `http://docs.phonegap.com/en/2.0.0/guide_getting-started_android_index.md.html#Getting%20Started%20with%20Android`.

Controlling your Android Virtual Device

Depending on the functionality included in your Cordova application, if you are testing on an Android Virtual Device, you may wish to make sure that your application successfully picks up changes to the network or the battery level of the device, for example.

How to do it...

In this recipe, we will access the Android Virtual Device using the command line to manipulate and change the settings to allow us to test effectively:

1. With your application running on an Android Virtual Device, look at the top title bar of the emulator to see what port it is running on, in this case, it is port 5554, as shown in the following screenshot:
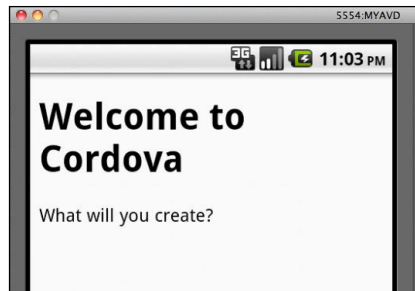
2. Open up your command line tool, or the Terminal if you are working on an OS X machine, and use the `telnet` command to connect to the localhost running on that port number:

   telnet localhost 5554

3. Following a successful connection, type `help` into the console to retrieve a list of commands that we can run against the emulator:

4. Let's adjust the current battery status of the device. Firstly, we'll check what we can do, so type the following in to the console window:

help power

The console will provide us with all available commands to manage the power of the device:

```
allows to change battery and AC power status

available sub-commands:
    power display           display battery and charger state
    power ac                set AC charging state
    power status            set battery status
    power present           set battery present state
    power health            set battery health state
    power capacity          set battery capacity state

OK
```

At the moment, your virtual device may appear as though it is charging. We want to adjust the charging state and set the battery capacity to 10 percent, which we can do by running the following three commands:

power ac off
power status not-charging
power capacity 10

5. Instantly, our virtual device will change from appearing as though it is charging, to something like the following screenshot:
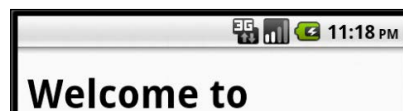
6. It will now look like it is not charging and has very low battery levels. We will even receive the notification alert to remind us to charge our device, as shown in the following screenshot:
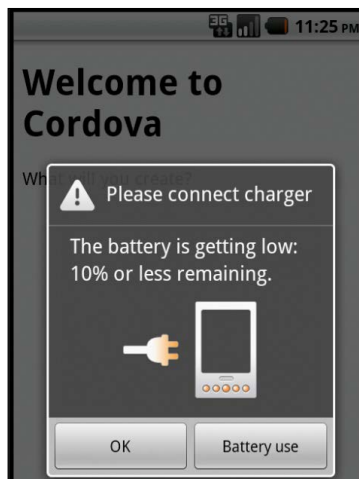
Let's now change the network connection for our device. We may want to make sure that our application successfully manages to save data or perform an action if the connection speed falls below a certain level:

```
network speed edge
```

7. Before the change, our device was running on full network speed:

8. The adjustments to the network speed have taken immediate effect:

How it works...

By accessing the Android Virtual Device running in the emulator, we are able to control a large number of device parameters, which also include SMS-related commands, geolocation commands, and control of the emulator itself.

These provide us with invaluable tools to test how our application responds to various device states and events.

> For more information on how you can interact with the Android emulator via the command line prompt, check out the official Android developer documentation available at: `http://developer.android.com/tools/help/emulator.html`.

273

Using Adobe Dreamweaver to develop Cordova applications
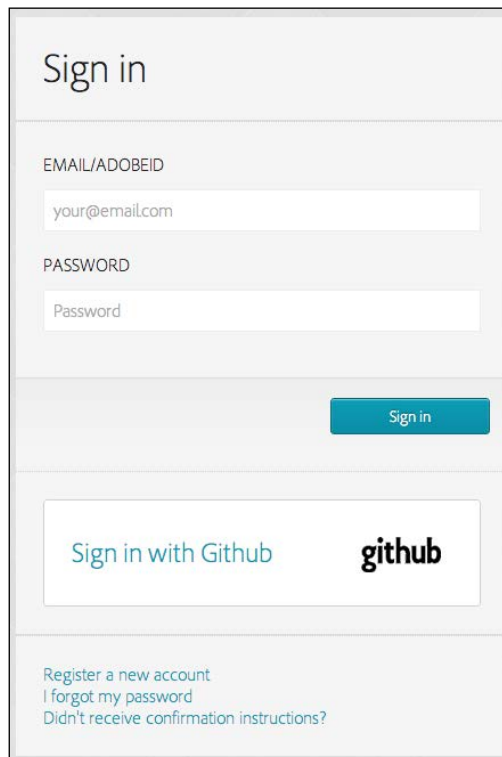
Adobe Dreamweaver has long been a favorite tool for web designers and developers. Dreamweaver CS5 included features to assist building mobile applications and the ability to simulate and package PhoneGap applications without having to worry about the underlying framework library.

Dreamweaver CS6 took this one step further and integrated an automated build service in the form of PhoneGap Build.

Getting ready

To use the PhoneGap Build service, you will first need to have an active PhoneGap Build service account, which is free and incredibly easy to set up.

Head over to `https://build.phonegap.com/people/sign_in` to begin the registration process, and you can sign in using either your Adobe ID or GitHub account details:

That's it! Is is time to start building an application using Dreamweaver CS6.

> If you do not have a copy of Dreamweaver CS6, you can download a full-featured trial version available here: `http://www.adobe.com/uk/products/dreamweaver.html`.

How to do it...

In this recipe we will use Dreamweaver CS6 to create a simple application that we will pass to the PhoneGap Build service to build for us:

1. To manage our Cordova application in Dreamweaver CS6, we need to create a new Dreamweaver site definition, pointing the project directory to your preferred location on your development machine.

2. We'll simplify the creation of our application structure and download a project template, which contains all of the core files we will need. Head over to `https://github.com/phonegap/phonegap-start` and click on the **ZIP** button to download a compressed version of the code. Alternatively, you can clone the GitHub project directly to your local machine using the command line:

3. Move or copy the extracted files from the download into the location of your Dreamweaver project. The resulting project structure should look something like the following screenshot:
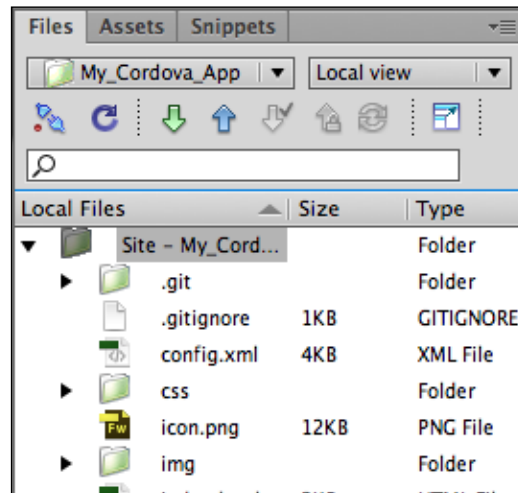
4. Open up `index.html` in the editor window. There are some JavaScript assets included in the `head` tag of the document, one of which is called `cordova-2.0.0.js`. However, you may notice that file does not actually exist in the project itself:

```html
20  <html>
21      <head>
22          <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
23          <meta name="format-detection" content="telephone=no" />
24          <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" />
25          <link rel="stylesheet" type="text/css" href="css/index.css" />
26          <title>Hello World</title>
27      </head>
28      <body>
29          <div class="app">
30              <h1>PhoneGap</h1>
31              <div id="deviceready">
32                  <p class="status pending blink">Connecting to Device</p>
33                  <p class="status complete blink hide">Device is Ready</p>
34              </div>
35          </div>
36          <script type="text/javascript" src="cordova-2.0.0.js"></script>
37          <script type="text/javascript" src="js/index.js"></script>
38          <script type="text/javascript">
39              app.initialize();
40          </script>
41      </body>
42  </html>
43
```

> When dealing with the PhoneGap Build service you do not need to have a local copy of the PhoneGap/Cordova JavaScript file, but you do need to reference it in the document as the service needs it to successfully build your application.

5. Open up the `config.xml` file in the editor window, which holds the configuration details for our application. Change the `id` attribute value to a reverse-domain value specific to you.

6. We'll also change the `name` value to that of our application and change the author node to set the developer information with our URL, e-mail, and name:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<widget
xmlns=http://www.w3.org/ns/widgets
xmlns:gap=http://phonegap.com/ns/1.0
id="com.coldfumonkeh.mycordovaapp"
version="1.0.0">

<name>My Cordova Application</name>
<author href="http://www.monkeh.me" email="me@monkeh.me">Matt
Gifford</author>
```

7. We can request that our application is built using a particular version of the PhoneGap library. Although the build service will use the current default version (currently 2.0.0), you can also enforce this by setting a `preference` tag to confirm that we want to use that version:

```xml
<preference name="phonegap-version" value="2.0.0" />
```

8. There is one `feature` node in the `config.xml` file, which specifies the features you want your application to use and have access to. This sample application is requesting access to the device API. Let's add two more feature nodes to request access to the geolocation and network connectivity APIs:

```xml
<feature name="http://api.phonegap.com/1.0/geolocation"/>
<feature name="http://api.phonegap.com/1.0/network"/>
```

> For detailed information on all of the parameters and values available to use in the `config.xml` file, please visit the PhoneGap Build documentation available at:
>
> `https://build.phonegap.com/docs/config-xml.`

9. Save the `config.xml` file, and open the PhoneGap Build service window by selecting **Site** | **PhoneGap Build Service** | **PhoneGap Build Service** from the main application menu, which will open the build service window in Dreamweaver. If this is the first time that you are running this process within Dreamweaver, you will be asked to enter your PhoneGap Build account credentials to log in and use the remote service:
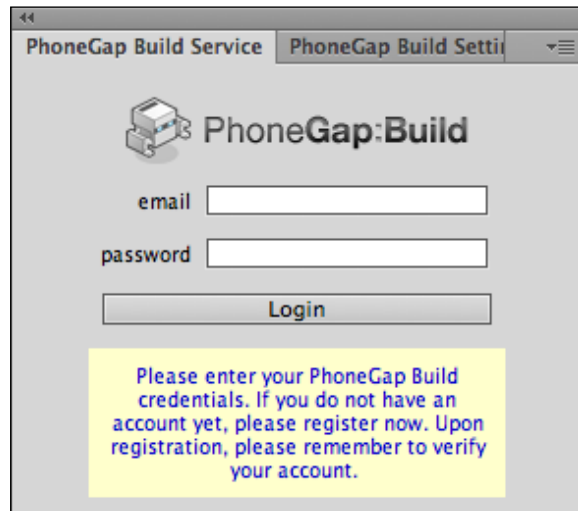
10. Once logged in, you will be presented with the build service panel, which will inform you that a project settings file needs to be built for this application. Ensure **Create as a new project** is selected in the drop-down box, and click on **Continue** to start the build process:
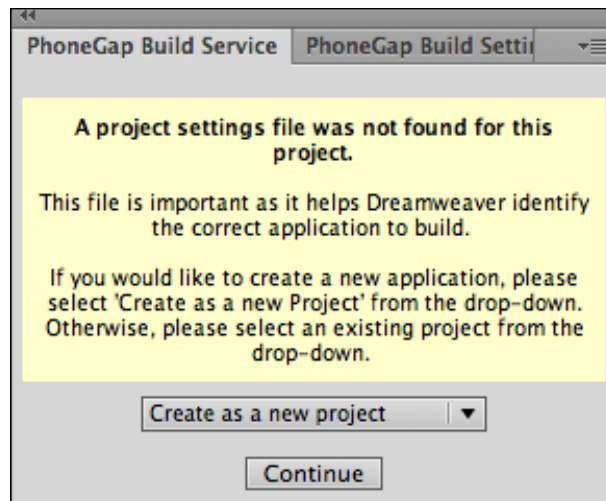
11. Dreamweaver will now submit your application to the remote PhoneGap Build service. You can view the process of each build from the panel in the editor.
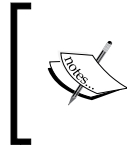
> If you receive any build errors, make sure that the `id` value in the `config.xml` file does not contain any underscores or invalid characters, and that there are no spaces within the XML node names and values.

12. Once the build process is complete, you will be shown the results for each mobile platform, as shown in the following screenshot:
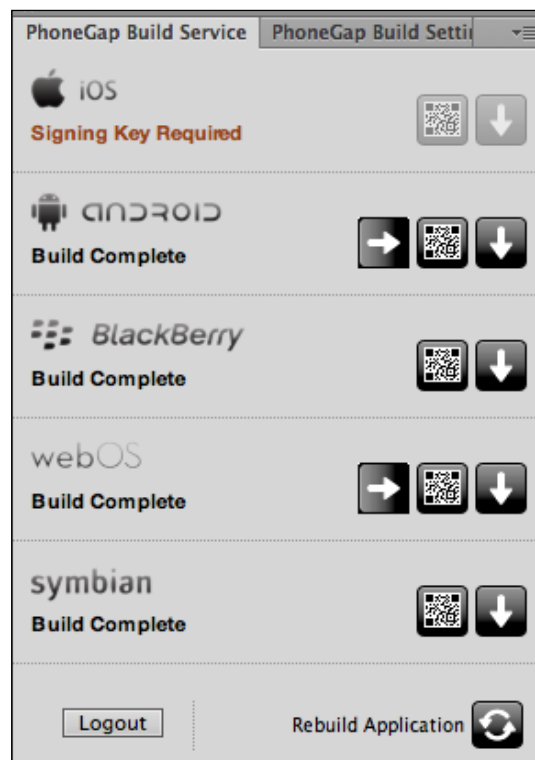
13. You will be presented with the option to download the packaged application in the respective native format by clicking on the downward-facing arrow. You can then install the application onto your device for testing.

14. If your mobile device has a **Quick Response Code** (**QR Code**) reader, you can select the barcode button and scan the resulting image shown to you. This will download the application directly onto the device without the need of any USB connectivity or transfers:

15. In this example, the iOS application was not built as no signing key was provided in the `config.xml` file.
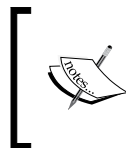
> For more details on iOS code signing, check out the official documentation from the iOS Developer Library available at: `http://developer.apple.com/library/ios/#technotes/tn2250/_index.html`.

16. You may also notice that the Android and webOS platforms have an extra button, the right-facing arrow. Clicking on the button will load the application onto a local emulator for testing purposes.

17. To use the emulators on your local machine, Dreamweaver needs to know the location of the SDK libraries. To set these paths, open up the **PhoneGap Build Settings** panel, which you can reach via **Site** | **PhoneGap Build Service** | **PhoneGap Build Settings** from the main menu, as shown in the following screenshot:
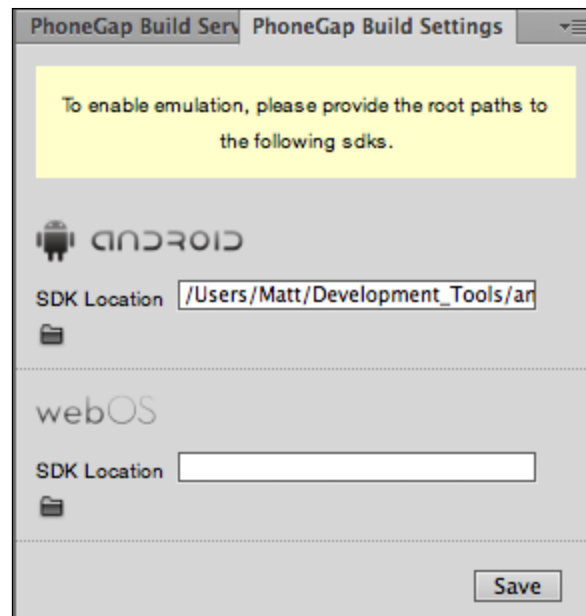
18. Include the path to the SDKs you have installed and wish to use, and click on **Save** to store them.

How it works...

The integration of the PhoneGap Build automated service directly into Dreamweaver CS6 manages the packaging and building of your application across a number of device platforms. The build process also takes care of the device permissions and the inclusion of the correct Cordova JavaScript file for each platform.

This means that you can spend more time developing your feature-rich native application and perfecting your layouts and visuals without having to handle the various build processes and differences between each platform.

There's more...

To install the Android SDK, you can follow the *Getting ready* section in the *Using Eclipse to develop Android Cordova applications* recipe in this chapter or check out the official Android Developer documentation, available at: `http://developer.android.com/sdk/index.html`.

To install the webOS SDK, please check out the official documentation, available at: `https://developer.palm.com/content/resources/develop/sdk_pdk_download.html`.
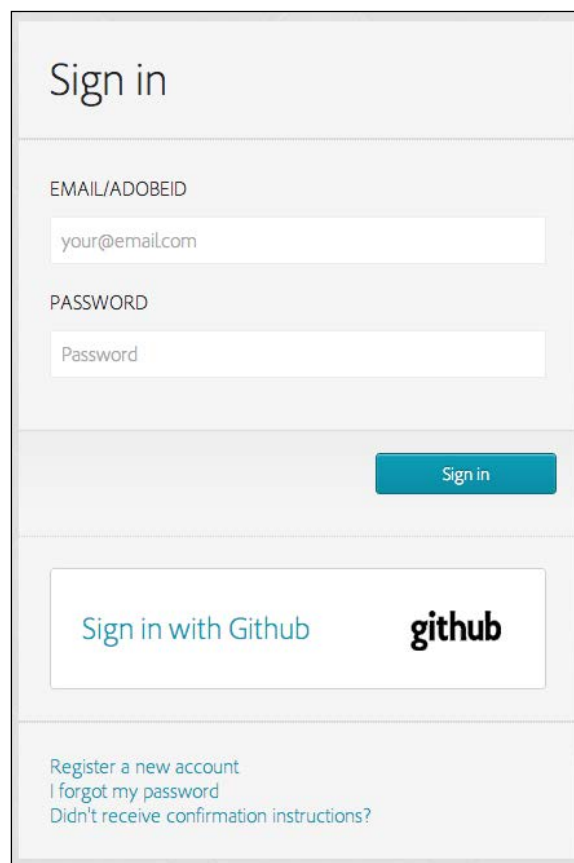
Using the PhoneGap Build service

The benefit of the Cordova project is the ability to create a native application from an HTML file and a little JavaScript, at it's very minimum. Adobe Dreamweaver CS6 has the built-in capabilities to interact with and upload your mobile project directly to the remote PhoneGap Build service on your behalf. You can, however, build your application using the service directly via the web.

Getting ready

To use the PhoneGap Build service, you will first need to have an active PhoneGap Build service account, which is free and incredibly easy to set up.

Head over to `https://build.phonegap.com/people/sign_up` to begin the registration process, and you can sign in using either your Adobe ID or GitHub account details:

How to do it...

In this recipe we will create a very simple, single-file application and upload it to the build service using the web interface:

1. Create a new `index.html` file, and includes the following code:

```html
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="user-scalable=no,
    initial-scale=1, maximum-scale=1,
    minimum-scale=1, width=device-width;" />
  <title>Cordova Application</title>

  <script type="text/javascript"
      src="cordova.js"></script>

  <script type="text/javascript">
    function onLoad() {
      document.addEventListener("deviceready", onDeviceReady,
false);
    }

    function onDeviceReady() {
      alert('Cordova has loaded');
    }
  </script>

</head>
<body onload="onLoad()">

  <h1>Welcome to Cordova</h1>

  <p>What will you create?</p>

</body>
</html>
```
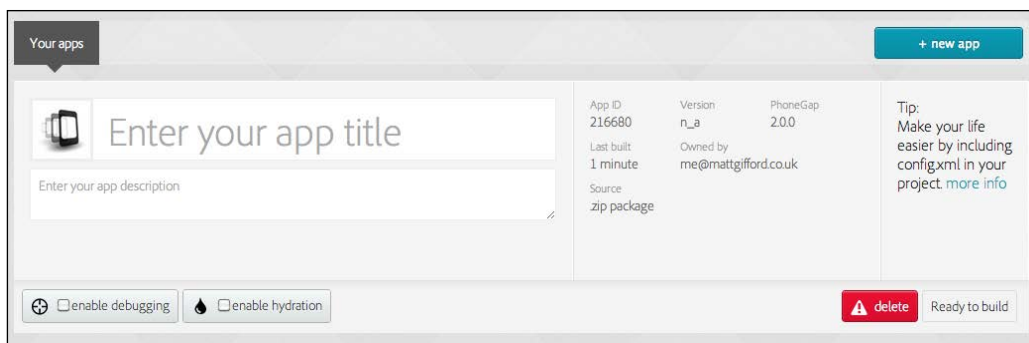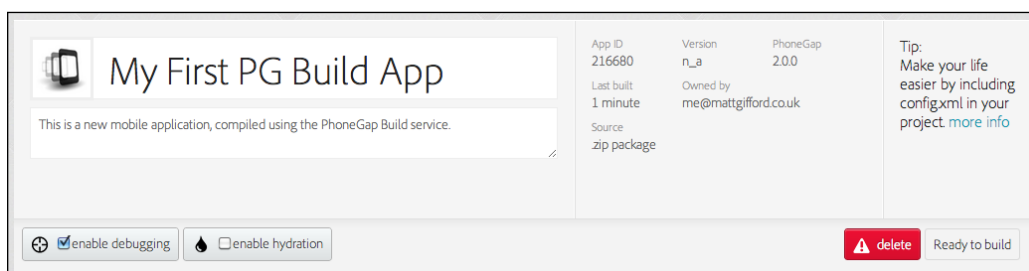
2. In this file we have included a reference to `cordova.js` in the `head` tag of the document. This file does not exist locally, but the build service requires this reference to successfully build the application. We have created a very simple function that will generate an alert notification window once the device is ready.

3. Create a new `.zip` file that contains the `index.html` file we have just created. We will use this archive file to send to the PhoneGap Build service to create our application.

4. Head over to `https://build.phonegap.com/` and sign in with your account details if you haven't done so already. If this is your first visit, or you have not yet submitted any applications to the service, you will be greeted instantly with a form to help you generate your first build.

5. You have the option to create either an open source or private application. Both options give the ability to provide code as either a link to a Git repository or as an archived `.zip` file. For the purpose of this recipe, we will select the **Upload a .zip file** button. Navigate to and select the archived file we created earlier and click on **Open** to proceed. The PhoneGap Build service will instantly upload your code and create a new project.

6. As we have not included a `config.xml` file in the upload, the application title and description are placeholders. Click on the **Enter your app title** text and enter the title or name of your application in the input box, making it easily identifiable so you can distinguish your application easily from any other applications you may upload. You can also add a description for your application in the same manner.

7. Let's also select the ability to debug your application:

8. Click on the **Ready to build** button at the bottom-right corner of the screen layout. The build process will now begin, and you will be presented with a visual reference to the available platforms and the status of each build:

9. In this example, four of the six builds were successful. The iOS and BlackBerry builds were not, as iOS needed a signing key and BlackBerry was unable to verify application passwords.

10. Click on the title of the application, which will take you to a new page with the options to download the successfully-built packages, as shown in the following screenshot:
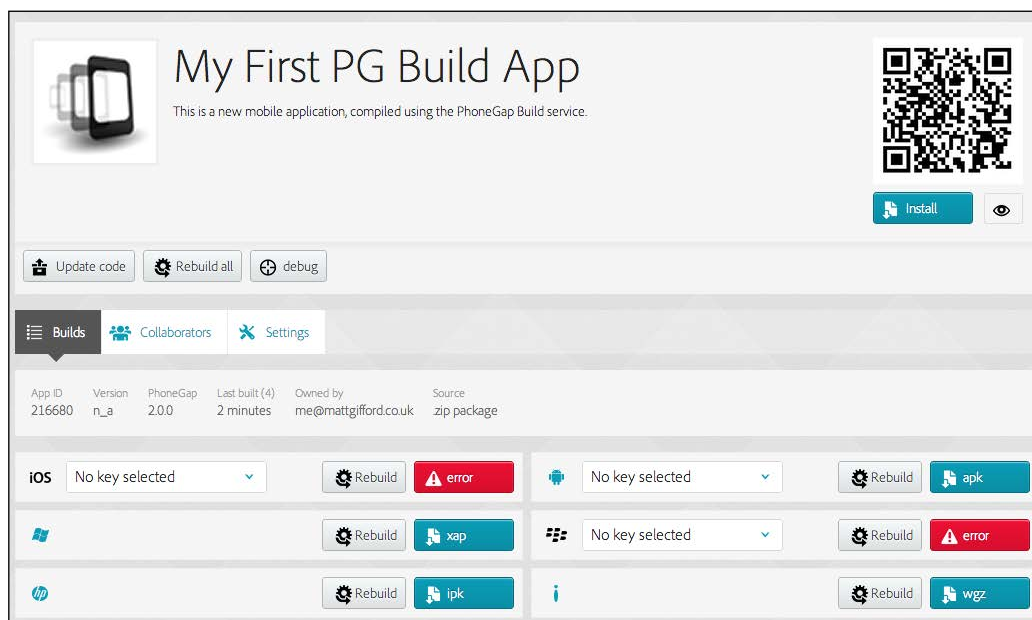
11. You can download the applications using the direct download link next to each successful build listing, or if your device has a QR Code reader, you can scan the barcode to download and install the application directly onto the device.

How it works...

The PhoneGap Build service automates the packaging process for your mobile application across the available device platforms, which greatly simplifies your workflow and the task of preparing configuration files for each individual platform.

There's more...

When we created this sample application using the online build service, we chose to enable debugging. This is another online service offered by the PhoneGap team, and makes use of an open source debugging application called **weinre**, which stands for **Web Inspector Remote**.

To see this in action, you need to have the application packaged by the PhoneGap Build service running on a physical device attached to your local development machine or on a device emulator. Once you have the application running, click on the **debug** button at the top of your project page, as shown in the following screenshot:
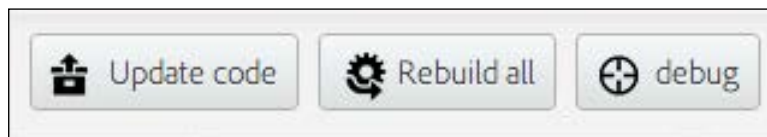
This will open up a new window or tab in your browser using the **debug.phonegap.com** subdomain and you should see your connected device in the list:

For any developers who have used development tools such as Firebug, the tools on offer with weinre should look very familiar. It provides similar functionality to test and debug HTML applications inline, but is designed to work remotely and is exceedingly good at testing apps and pages on mobile devices.

> To find out more about the features and functions available in weinre, check out the official documentation, available at: `http://people.apache.org/~pmuellr/weinre/`.

The PhoneGap Build documentation also provides an introduction to the service with an emphasis on using it with the online build service. This is available at: `https://build.phonegap.com/docs/phonegap-debug`.

Hydrating your application

PhoneGap Build also provides a tool to enhance the workflow and deployment of compiled applications to your devices for testing, called **Hydration**. This reduces the time taken to compile an application and automatically pushes out a new build of a hydrated app to the device. This can be enabled when creating a new project, or the settings updated and applied to an existing project.

To apply hydration to our current application, select the **Settings** tab within the project view and click on the **enable hydration** button. Click on the **Save** button to retain the changes. At this point, the application will automatically rebuild as a hydrated app.
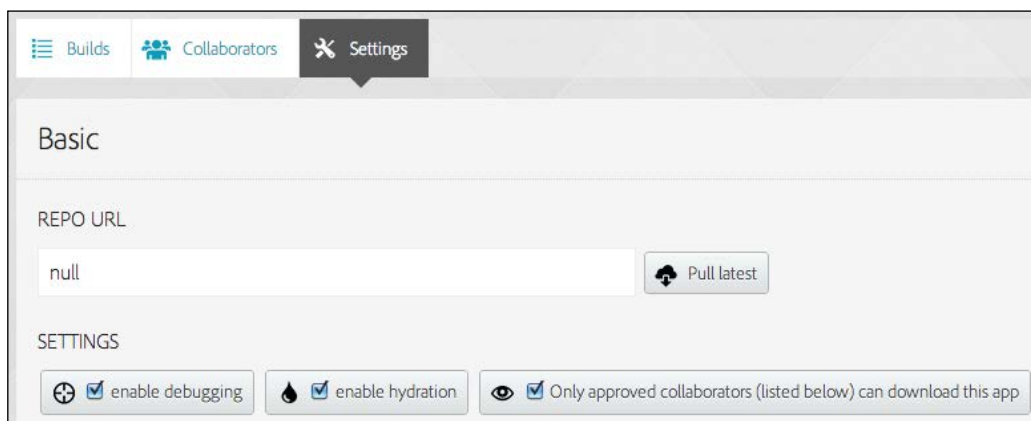
Install or deploy the freshly compiled application onto your device, replacing any previous non-hydrated version you may already have.

The benefit of a hydrated application is the ease and simplicity of deploying further updates. Once any future code has been updated, deployed, and compiled in the build service, you can easily update your installed version by restarting the application on the device. The hydrated application will check on every startup to see if an updated version of the app exists, and if so, it will prompt the user with a dialog, offering the chance to update the application.
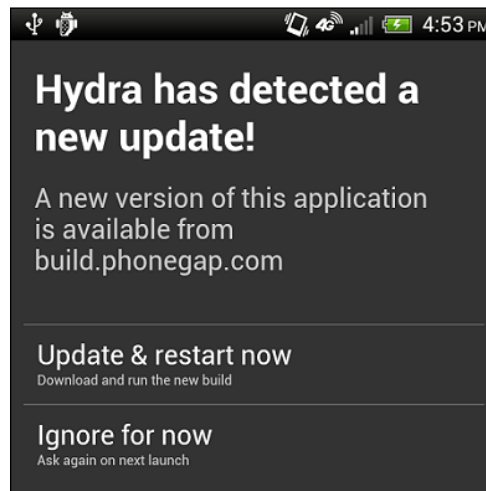
If accepted, the new build will be downloaded and installed directly onto the device.
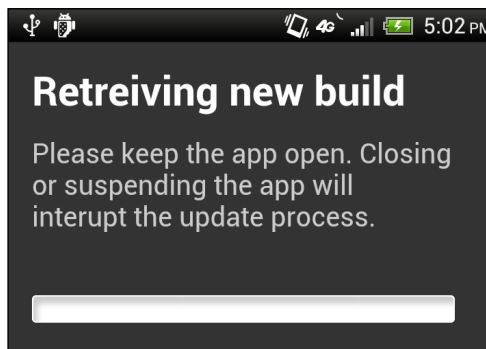
To find out more about the Hydration service, check out the official documentation available at: `https://build.phonegap.com/docs/hydration`.

PhoneGap Build API

The online PhoneGap Build service is also available via an exposed public **Application Programming Interface** (**API**).

By opening the build service as an API, developers have the ability to create, build, update, and download PhoneGap applications from shell scripts, the command line, from a separate automated build process, or the source control post commit hook process.

Both read and write access to the service is available using authentication protocols.

> To find out more about the PhoneGap Build API, check out the official documentation available at: `https://build.phonegap.com/docs/api`.

Index

C

W

X

[PACKT] open source✳
PUBLISHING   community experience distilled

Thank you for buying
PhoneGap Mobile Application
Development Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
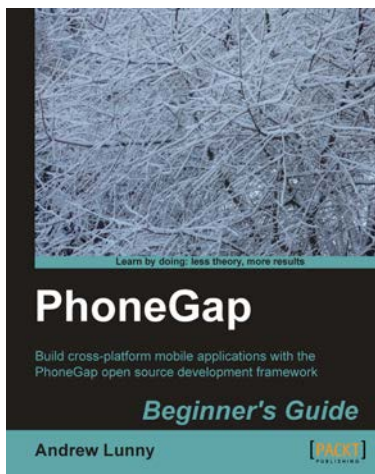
PhoneGap Beginner's Guide

ISBN: 978-1-849515-36-8   Paperback: 328 pages

Build cross-platform mobile applications with the PhoneGap open source development framework

1. Rich interactions using HTML5 and CSS3 APIs

2. Mobile JavaScript expertise: write code that travels anywhere

3. Working offline: use web development skills with native code to create installable web-apps that sync with remote servers

4. Enhancing application experiences with real-time sensor data
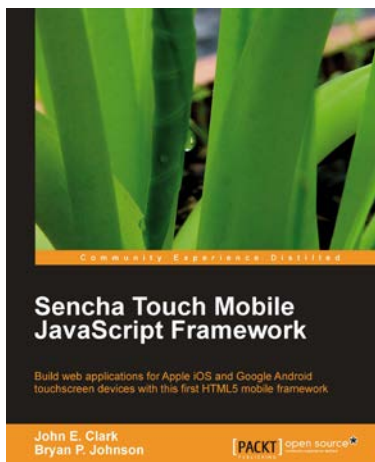
Sencha Touch Mobile JavaScript Framework

ISBN: 978-1-849515-10-8   Paperback: 316 pages

Build web applications for Apple iOS and Google Android touchscreen devices with this first HTML5 mobile framework

1. Learn to develop web applications that look and feel native on Apple iOS and Google Android touchscreen devices using Sencha Touch through examples

2. Design resolution-independent and graphical representations like buttons, icons, and tabs of unparalleled flexibility

3. Add custom events like tap, double tap, swipe, tap and hold, pinch, and rotate

Please check **www.PacktPub.com** for information on our titles
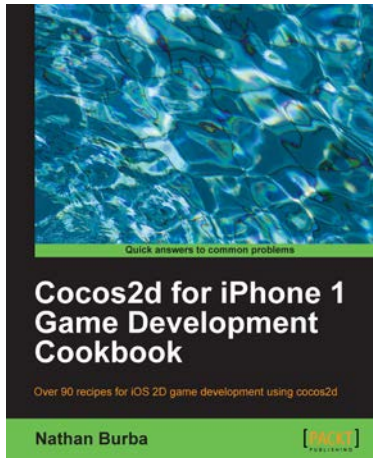
Cocos2d for iPhone 1 Game Development Cookbook

ISBN: 978-1-849514-00-2　　　　Paperback: 446 pages

Over 90 recipes for iOS 2D game development using cocos2d

1. Discover advanced Cocos2d, OpenGL ES, and iOS techniques spanning all areas of the game development process

2. Learn how to create top-down isometric games, side-scrolling platformers, and games with realistic lighting

3. Full of fun and engaging recipes with modular libraries that can be plugged into your project
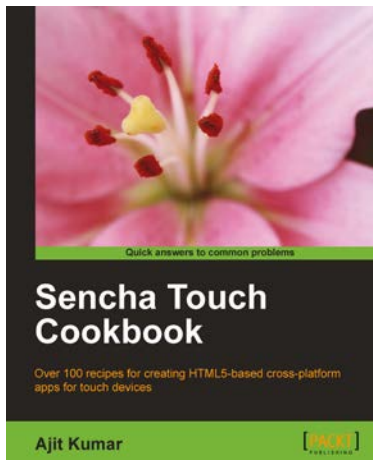
Sencha Touch Cookbook

ISBN: 978-1-849515-44-3　　　　Paperback: 350 pages

Over 100 recipes for creating HTML5-based cross-platform apps for touch devices

1. Master cross platform application development

2. Incorporate geo location into your apps

3. Develop native looking web apps

Please check **www.PacktPub.com** for information on our titles