MOBILE PROGRAMMING SERIES

# APACHE CORDOVA 3 PROGRAMMING

COVERS PHONEGAP 3.X

JOHN M. WARGO

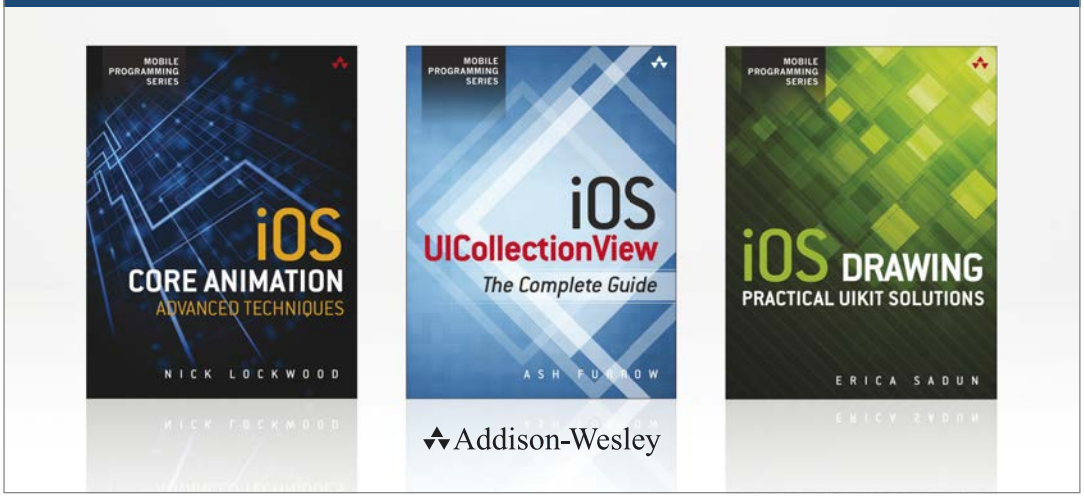
# Apache Cordova 3 Programming

John M. Wargo

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

# Addison-Wesley Mobile Programming Series




Visit **informit.com/mobile** for a complete list of available publications.

---

The **Addison-Wesley Mobile Programming Series** is a collection of digital-only programming guides that explore key mobile programming features and topics in-depth. The sample code in each title is downloadable and can be used in your own projects. Each topic is covered in as much detail as possible with plenty of visual examples, tips, and step-by-step instructions. When you complete one of these titles, you'll have all the information and code you will need to build that feature into your own mobile application.

Make sure to connect with us!
informit.com/socialconnect

informIT.com the trusted technology learning source | Addison-Wesley | Safari Books Online

ISBN-13: 978-0-321-95736-8
ISBN-10: 0-321-95736-9

First released, December 2013

*To my wife, Anna.*

*This work exists because of your outstanding support.*

*To my children,*

*who were relatively patient as I worked on yet another book.*

# *Contents*

# *Foreword*

It has been roughly twenty-five years now since the mobile technology era began. Can it be just a coincidence that it is almost exactly the same amount of time that I have known John Wargo, the author of this book? In 1988, the mobile technology landscape consisted of mobile phones the size of a carry-on suitcase and personal organizers that looked like glorified financial calculators. Wireless networks and widespread access to resources like the Internet were distant dreams. Unless you were a science fiction writer, it would have been hard to imagine the connected world that we take for granted today. If you knew John Wargo as I do, though, it would not have been very far-fetched at all to predict that he would turn out to be the author of four books.

Looking ahead another twenty-five years, I imagine it's safe to say that our ability to accurately predict the advance of technology will continue to fall short. Powerful development tools like Cordova and the combined creativity of millions of application developers around the world virtually guarantee that we won't be able to guess what indispensable capabilities will appear on our mobile devices tomorrow or the next day. Will tomorrow's devices interface directly to our human nervous systems? Will they assemble themselves from organic compounds and heal themselves if they become damaged? Who knows.

Whatever the future of mobile technology holds for us, as long as there is a means to program or control it in some way, I hope that John will put together a collection of words to point others in the right direction.

*—David M. Via*
*AT&T*

# *Preface*

This is a book about programming cross-platform mobile applications using Apache Cordova 3 (with some coverage of PhoneGap 3 as well). In Apache Cordova 3, the Cordova development team made some dramatic changes in the framework, and this book is what you need to understand what Cordova 3 is all about. The book can be considered as a sequel to my *PhoneGap Essentials*, updated for Cordova 3.

This is a book about Cordova 3. This book is targeted at mobile developers who want to learn about Cordova 3. If you're brand new to Cordova, then this book is just what you need to get started. If you're experienced with an older version of Cordova, this book will show you in detail how to use all of the new stuff that's in Cordova 3. You will, however, need to have at least some experience with mobile development to benefit from this book. The target audience could be existing web developers who want to get into mobile development, but much of the needed native mobile development background just isn't in here.

What you'll find in the book:

- Lots of detailed information about Apache Cordova: what it does and how it works
- Lots of examples and code

What you won't find in this book:

- Mobile web development and mobile development topics; this book is about Apache Cordova, not mobile development
- Expressions or phrases in languages other than English (I hate it when authors include phrases in Latin or French)
- Obscure references to pop-culture topics (although there is an overt reference to Douglas Adams's *Hitchhiker's Guide to the Galaxy* and one obscure reference to "Monty Python")
- Pictures of my children or my pets

This book is not a book for experienced Cordova 3 developers—if you consider yourself an experienced Cordova 3 developer, then you probably should not buy this book.

Herein I tried to provide complete coverage of Cordova 3; covering enough detail that readers will leave with a complete understanding of what Cordova is, what it does, how it works, and how to use it for their mobile application projects. There's a whole lot more to Cordova; many advance topics and more detailed coverage of the Cordova APIs can be found in the Cordova documentation and maybe in some future book I'll write.

## Cordova as a Moving Target

One of the challenges in writing a book about open source projects is that if the project is well staffed and busy, the project gets regular updates. In Cordova's case, it's one of the fastest moving open source projects on the planet, so with their monthly updates and yearly major release, it is definitely a moving target.

I've worked very hard to structure and craft this book so that it can survive the rapid pace of the project, but only time will tell. You may find that something I've written here has changed and the book doesn't align with reality. There's nothing I can do about this except to stay on top of it and post updates to the book's website (described shortly) when I find that something has changed enough that it breaks part of the book.

## A Comment on Source Code

One of the things you'll notice as you look at the source code included in the book is that I've paid special attention to the formatting of the code so that it can be easily read and understood. Rather than allowing the source code to wrap wherever necessary to fit the printed page, I've forced line breaks in the code in order to structure it in a way that should benefit the reader. Because of this, as you copy the source code over into your Cordova applications, you will likely find some extra line breaks that affect the functionality of the code. Sorry.

## Resources

I've created a website for the book at www.cordovaprogramming.com, shown in Figure P1, where I will post updates, errata, and the answers to questions I've received from readers.

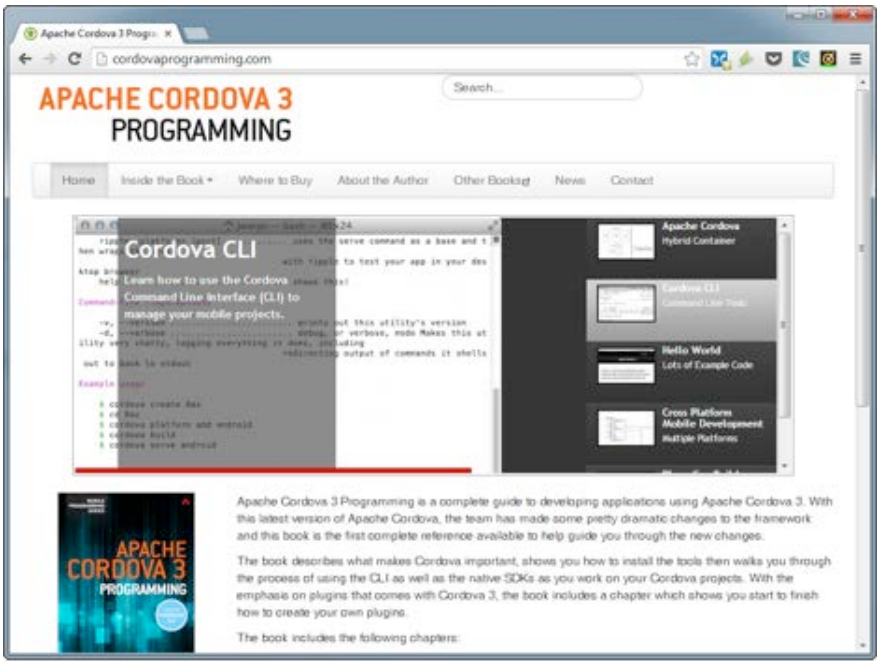



Figure P.1 Apache Cordova 3 Programming Web Site

Additionally, I've posted much of the book's sample code to my GitHub account located at https://github.com/johnwargo. At a minimum, I'll post all of the complete applications up there—potentially adding code snippets if readers ask for them.

Please feel free to use the contact form on the book's website to provide feedback and/or suggestions for the next release.

# *Acknowledgments*

I want to thank the following people for their help with this effort:

- The Cordova development team for answering every one of my silly questions as I prepared the manuscript
- Brian LeRoux from Adobe for his support
- Fil Maj and Hardeep Shoker from Adobe for reviewing and providing feedback on the Cordova CLI and PhoneGap Build chapters
- Brent Thornton, Bryan Higgins, Ken Wallis, and Jeffrey Heifetz from BlackBerry for helping me with the BlackBerry content
- Raman Sethi and the SAP Kapsel development team for teaching me all sorts of stuff I didn't know about Cordova
- My colleague Marcus Pridham for pointing out what was wrong with my Android plugin code and showing me how to create the iOS version of the plugin
- My colleague Istvan Nagy for showing me a better way to tell what's going on with a Cordova application when it fails (described in Chapter 14)
- Colleagues Damien Murphy, Scott Dillon, and Andrew Lunde for their review of the manuscript
- My managers and other colleagues at SAP for supporting me throughout this process
- Greg Doench and the staff at Addison-Wesley/Pearson Education for helping me write yet another book; may this one not be my last.

# 1

# The What, How, Why, and More of Apache Cordova

This chapter is your introduction to the Apache Cordova framework: what it is, how developers use it to develop mobile applications, how it differs from Adobe PhoneGap, why it was created, how it has changed over time, where it's going, and more. You could skip this chapter and dig into the more technical (and fun) stuff in the chapters that follow, but it's important to understand the heart of what you're working with before getting to all of the details.

As I read through support forum posts, it's clear from many of them that the developers just getting started with Apache Cordova don't really "get" what they're working with. This chapter should answer many of the initial what, how, and why questions related to Apache Cordova and Adobe PhoneGap.

## Introduction to Apache Cordova

Apache Cordova (http://cordova.apache.org) is a free, open source framework for building cross-platform native applications using HTML5. The creators of Apache Cordova wanted a simpler way of building cross-platform mobile applications and decided to implement it as a combination of native and web application technologies. This type of mobile application is called a *hybrid* application.

The initial benefit of Apache Cordova is the native capabilities above and beyond what is normally supported in the browser. At the time all of this started, the best way to build a mobile application that worked on multiple mobile devices was to build it using HTML. Unfortunately for mobile developers, though, many mobile applications needed to do more than HTML and web browsers could support, and building a web application that interacted with the device camera or the local contacts application simply wasn't possible. To get around this limitation, Cordova implements a suite of APIs that extend native device capabilities (such as the camera, accelerometer, Contacts application, and so on) to a web application running within the native container.

Apache Cordova consists of the following components:

- Source code for a native application container for each of the supported mobile device platforms. The container renders the HTML5 application on the device (more on what this means later, I promise).
- A suite of APIs that provide a web application running within the container access to native device capabilities (and APIs) not normally supported by a mobile web browser.

- A set of tools used to manage the process of creating application projects, managing plugins, building (using native SDKs) native applications, and testing applications on mobile device simulators and emulators.

To build a Cordova application, you create a web application, package the web application into the native container, test and debug the application, then distribute it to users (typically through an app store). That's all there is to it. The packaging process is illustrated in Figure 1.1; we talk more about how you package applications later in the chapter.

> **Note**
>
> When many developers first learn about this technology, they immediately assume that the web application is somehow translated into the native language for each supported mobile device platform—converted into Objective-C for iOS or Java for Android, for example—but that's not what's happening here. Some mobile application frameworks take that approach, but for Cordova, the web application simply runs unmodified within a native application shell.

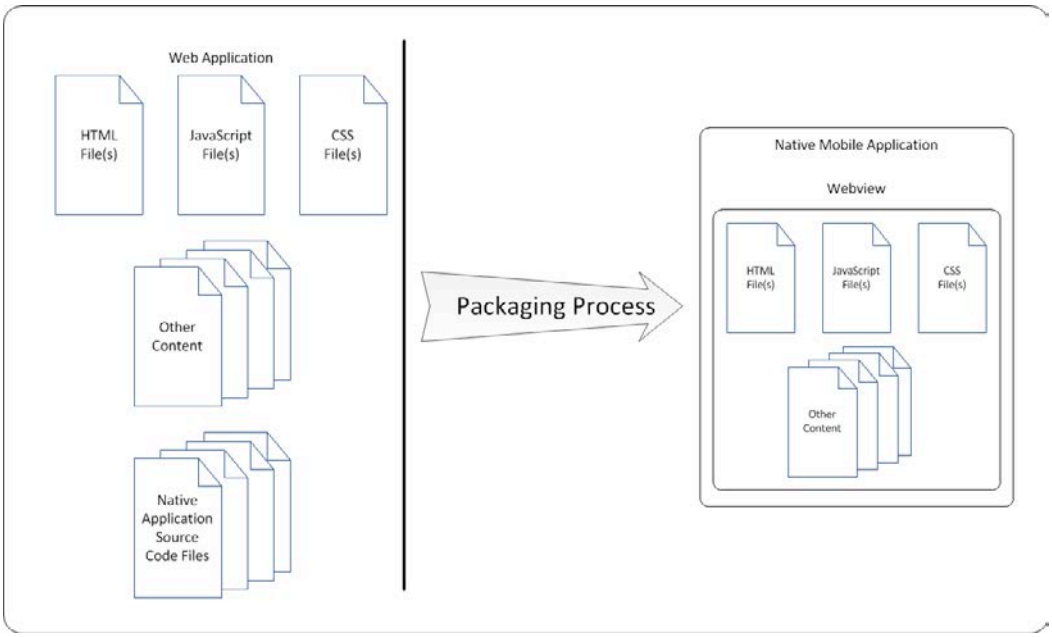



**Figure 1.1** Apache Cordova Application Packaging Process

Within the native application, the application's user interface consists of a single screen that contains nothing but a single web view that consumes the available screen space on the device. When the application launches, it loads the web application's startup page (typically index.html but easily changed by the developer to something else) into the web view, then passes control to the web view to allow the user to interact with the web application. As the user interacts with the application's content (the web application), links or JavaScript code within the application can load other content from within the resource files packaged with this application or can reach out to the network and pull content down from a web or application server.

For some mobile device platforms, such as bada, Symbian, and webOS, a native application is just a web application: there's no concept of a compiled native application that is deployed to devices. Instead, a specially packaged web application is what is executed as an application on the device.

### About Web Views

A web view is a native application component that is used to render web content (typically HTML pages) within a native application window or screen. It's essentially a programmatically accessible wrapper around the built-in web browser included with the mobile device.

For some examples, on the BlackBerry Java platform, it's implemented as a `BrowserField` object (using `net.rim.device.api.browser.field2`). On Android, it's implemented using a `WebView` view (using `android.webkit.WebView`), and on iOS, it's a `UIWebView` (using `System/Library/Frameworks/UIKit.framework`).

The web application running within the container is just like any other web application that would run within a mobile web browser. It can open other HTML pages (either locally or from a web server sitting somewhere on the network). JavaScript embedded within the application's source files implements needed application logic, hiding or unhiding content as needed within a page, playing media files, opening new pages, performing calculations, and retrieving content from or sending content to a server. The application's look and feel is determined by font settings, lines, spacing, coloring, or shading attributes added directly to HTML elements or implemented through Cascading Style Sheets (CSS). Graphical elements applied to pages can also help provide a theme for the application. Anything a developer can do in a web application hosted on a server can also be done within a Cordova application.

A mobile web browser application does not typically have access to device-side applications, hardware, and native APIs. For example, a web application typically cannot access the Contacts application or interact with the accelerometer, camera, compass, microphone, and other features, nor can it determine the status of the device's network connection. A native mobile application, on the other hand, makes frequent use of those capabilities. For a mobile application to be interesting (interesting to prospective application users, anyway), it will likely need access to those native device capabilities.

Cordova accommodates that need by providing a suite of JavaScript APIs that a developer can leverage to allow a web application running within the Cordova container to access device capabilities outside of the web context. These APIs were implemented in two parts: a JavaScript library that exposes the native capabilities to the web application and the corresponding native code running in the container that implements the native part of the API. The project team would essentially have one JavaScript library but separate native implementations on each supported mobile device platform.

Prior to Cordova 3.0, these APIs were implemented as shown in Figure 1.2—a single JavaScript interface that exposed the web application to all of the supported native APIs.

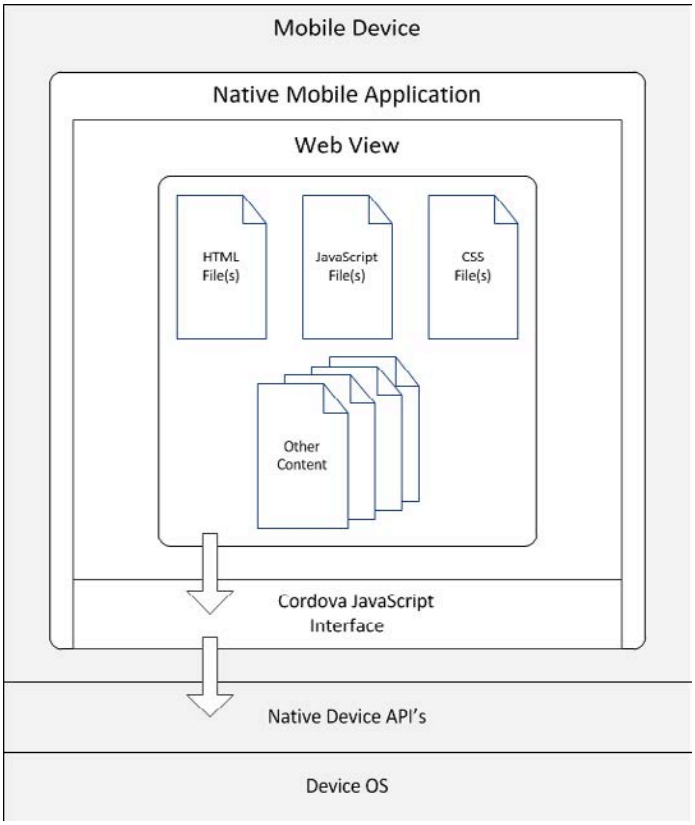



**Figure 1.2** Apache Cordova Native Application Architecture Pre-3.0

The result, however, was that your Cordova application included JavaScript and native code for every Cordova API. If your application didn't leverage all of the available APIs, you still had the code in your application for the APIs you weren't using. You could strip unused APIs from the JavaScript library and the native container, but that was not fun.

Beginning with Cordova 3.0, each of the Cordova APIs has been broken out into separate plugins; you can use the Cordova plugin manager (plugman) to add and remove plugins from your Cordova project. We talk more about plugins and the Cordova tools later in this chapter and throughout the book. This approach provides the architecture illustrated in Figure 1.3—an application with discrete code for each plugin and that is packaged with only the needed plugins.

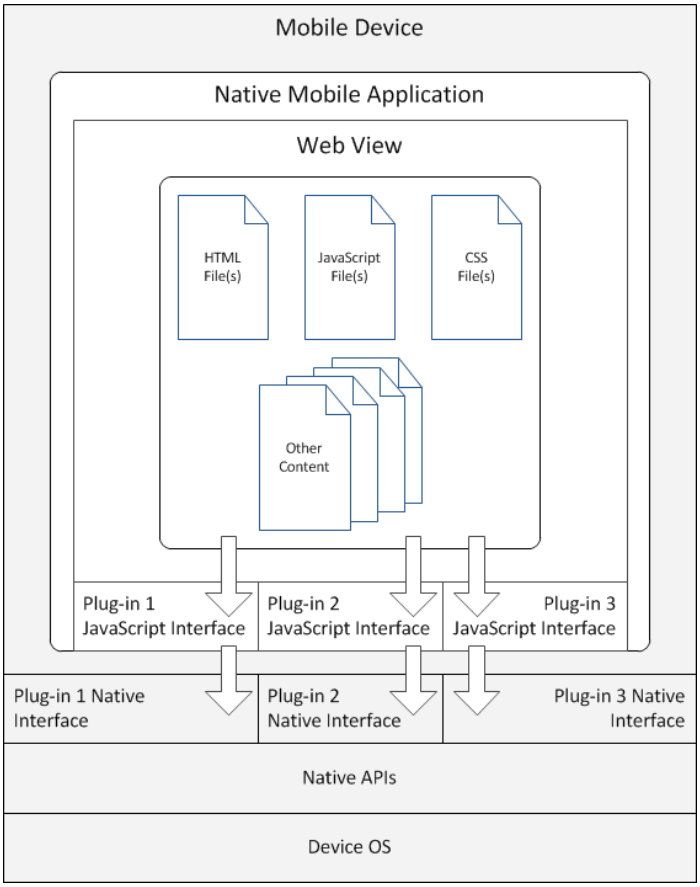



**Figure 1.3** Apache Cordova Native Application Architecture Post-3.0

Cordova currently provides the following APIs:

- Accelerometer
- Camera
- Capture
- Compass
- Connection
- Contacts
- Device
- Events
- File
- Geolocation
- Globalization

- InAppBrowser
- MediaNotification
- Splashscreen
- Storage

Most of these APIs are described in detail in Chapters 10 through 22 of *PhoneGap Essentials* (www.phonegapessentials.com), but only for PhoneGap and Cordova 2.x.

When a developer implements a feature in an application that uses one of the Cordova APIs, the application calls the API using JavaScript. A special layer within the application translates the Cordova API call into the appropriate native API for the particular feature. As an example, the way the camera is accessed on a BlackBerry is different from how it's done on Android, so this API common layer allows a developer to implement a single interface that is translated behind the scenes (within the container application) into the appropriate native API for each supported mobile platform. To take a picture in a mobile application using Cordova and the default options for the API, the JavaScript code would look like this:

```
navigator.camera.getPicture( onSuccess, onFail );
```

As parameters, the application passes in the names of two callback functions: `onSuccess` and `onFail`, which are called once a picture has been captured or if an error is encountered.

On BlackBerry, the code being executed behind the scenes might look like this:

```
Player player = Manager.createPlayer("capture://video");
player.realize();
player.start();
VideoControl vc = (VideoControl) player.getControl(
  "VideoControl");
viewFinder = (Field)vc.initDisplayMode(
  VideoControl.USE_GUI_PRIMITIVE,
  "net.rim.device.api.ui.Field");
scrnMain.add(viewFinder);
vc.setDisplayFullScreen(true);
String imageType =
  "encoding=jpeg&width=1024&height=768&quality=fine";
byte[] theImageBytes = vc.getSnapshot(imageType);
Bitmap image = Bitmap.createBitmapFromBytes(
  imageBytes, 0, imageBytes.length, 5);
BitmapField bitmapField = new BitmapField();
bitmapField.setBitmap(image);
scrnMain.add(bitmapField);
```

On Android, the code being executed by the function might look like this:

```
camera.takePicture( shutterCallback, rawCallback,
jpegCallback );
```

And on iOS, the code might look like this:

```
UIImagePickerController *imgPckr =
  [[UIImagePickerController alloc] init];
```

```
imgPckr.sourceType =
  UIImagePickerControllerSourceTypeCamera;
imgPckr.delegate = self;
imgPckr.allowsImageEditing = NO;
[self presentModalViewController:imgPckr
  animated:YES];
```

The code samples listed here don't cover all aspects of the process of taking a picture (such as dealing with errors or processing the resulting image), but they illustrate how Cordova simplifies cross-platform mobile development. A developer makes a single call to a common API available across all supported mobile platforms, and Cordova translates the call into the appropriate code for each target platform. Cordova eliminates the need for developers to have intimate knowledge of the underlying technologies, allowing them to focus on their application rather than on how to accomplish the same task on multiple devices.

# What Is Adobe PhoneGap?

Adobe PhoneGap is nothing more than an implementation of Apache Cordova with some extra stuff added to it. At its core is the Cordova container and API plugins described in the previous section. As Adobe's primary business is in selling tools and services, the PhoneGap implementation of Cordova more tightly integrates the framework with Adobe's other products.

The primary differences between Cordova and PhoneGap are the command-line tools and the PhoneGap Build service (described later in the chapter). The PhoneGap command-line tools provide a command-line interface into the PhoneGap Build service; the Adobe PhoneGap Build service is covered in Chapter 11, "Using PhoneGap Build."

Throughout the remainder of the book (except in the following "PhoneGap History" section), when I refer to *PhoneGap*, I'm talking about a specific capability that is available only in the PhoneGap version of Cordova. Both versions are free; PhoneGap simply adds some additional capabilities to Cordova.

# PhoneGap History

PhoneGap was started at the 2008 iPhoneDevCamp by Nitobi (www.nitobi.com) as a way to simplify cross-platform mobile development. The project began with a team of developers working through a weekend to create the skeleton of the framework; the core functionality plus the native application container needed to render web application content on the iPhone. After the initial build of the framework, the PhoneGap project team quickly added support for Android with BlackBerry following a short time thereafter.

In 2009, PhoneGap won the People's Choice award at the Web 2.0 Expo LaunchPad competition. Of course, being a project for geeks, the conference attendees voted for the winner by Short Message Service (SMS) from their mobile phones.

Over time, PhoneGap has added support for additional hardware platforms and worked to ensure parity of API features across platforms. During this period, IBM started contributing to the project, as did many other companies.

In late 2011, Nitobi announced that it was donating PhoneGap to the Apache Foundation. Very quickly thereafter, Adobe announced that it was acquiring Nitobi. PhoneGap joined the open source Apache project (www.apache.org) as an incubator project, first as Apache Callback, briefly as Apache

DeviceReady, and finally (beginning with version 1.4) as Apache Cordova (the name of the street where the Nitobi offices were located when PhoneGap was created).

The acquisition of Nitobi by Adobe (and Adobe's subsequent announcement that it would discontinue support for Adobe Flash on mobile devices) clearly indicates that Adobe saw PhoneGap as an important part of its product portfolio. The folks at Nitobi who were working on PhoneGap in their spare time as a labor of love are now in a position to work full time on the project. The result is that Cordova is one of, if not *the*, most frequently updated Apache projects. The Cordova team delivers new releases monthly, which is pretty amazing considering the complexity of the code involved.

## Cordova Going Forward

When you look at the project's description on its Apache project home page (http://cordova.apache.org/#about), you'll see that the project team describes itself almost entirely by the APIs Cordova implements.

The Cordova project's efforts around API implementation were initially guided by the World Wide Web Consortium (W3C) Device APIs and Policy (DAP) Working Group (www.w3.org/2009/dap/). This group is working to "create client-side APIs that enable the development of Web Applications and Web Widgets that interact with device services such as Calendar, Contacts, Camera, etc." The plan was for additional APIs to be added as the Cordova project team gets to them and as new standards evolve, but that's not what's happened lately.

From the middle of the Cordova 1.x code stream through the end of the 2.x releases, the project team started working on tightening up the framework. They focused primarily on fixing bugs and cleaning up the project's code. Where there were previously separate JavaScript libraries for each mobile platform, they worked toward consolidating them into a single file (cordova.js) and migrating everything from the PhoneGap to the Cordova namespace. For the 3.0 release, the project team focused on stripping the APIs out of the core container and migrating them into separate plugins, then creating some cool new command-line tools to use to manage application projects. The project team should start adding more new APIs to the framework soon after 3.0 is released.

In May 2012, Brian LeRoux (brian.io) from Adobe published "PhoneGap Beliefs, Goals, and Philosophy" (http://phonegap.com/2012/05/09/phonegap-beliefs-goals-and-philosophy) in which he talks about what was (then) driving the project's direction. At the time, as mobile device browsers implemented the DAP APIs in a consistent manner, the plan was for Cordova to obsolete itself. The expectation was that when mobile browsers all support these APIs, there would be no need for the capabilities Cordova provides and the project would just disappear. A good example of this is how modern browsers are starting to add support for the camera, as described in Raymond Camden's blog post "Capturing camera/picture data without PhoneGap" (www.raymondcamden.com/index.cfm/2013/5/20/Capturing-camerapicture-data-without-PhoneGap).

However, one of the things I noticed as I finished *PhoneGap Essentials* (www.phonegapessentials.com) was that plugins were gaining in prominence in the Cordova space. The APIs provided by Cordova were interesting and helpful to developers, but developers wanted more. Where there were first only a few Cordova plugins available, now there are many, and the core APIs are plugins as well. So, Cordova becomes at its core just a hybrid container, and everything else is done in plugins.

As the browsers implement additional APIs, the core Cordova APIs will become obsolete, but the Cordova container may live on. Cordova could still obsolete itself, but only in a time when the popular mobile browsers provide a standard interface to native APIs. As each platform's OS and API suite are different, I'm not sure how that would work out. It's the cross-platform development capabilities of

Cordova that make it most interesting to the market; there's limited chance the market will expose native APIs to the browser in a consistent enough way to keep cross-platform development viable.

# Supported Platforms

Apache Cordova currently supports the following mobile device operating system platforms:

- Android (Google)—http://developer.android.com/index.html
- bada (Samsung)—http://developer.bada.com
- BlackBerry 10 (BlackBerry)—https://developer.blackberry.com/
- iOS (Apple)—https://developer.apple.com/devcenter/ios/index.action
- Firefox OS—https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS
- Tizen (originally Samsung, now the Linux Foundation)—https://developer.tizen.org
- Windows Phone 7 and Windows Phone 8 (Microsoft)—http://developer.windowsphone.com/en-us
- Windows 8 (Microsoft)—http://msdn.microsoft.com

You can find the complete list of supported operating systems plus the specific capabilities that are supported at http://cordova.apache.org/docs/en/3.0.0/guide_overview_index.md.html#Overview. For this book, I cover what are considered the most popular smartphone platforms, Android and iOS, plus some others that I find interesting, such as BlackBerry 10 and Windows Phone 8: BlackBerry because I love BlackBerry and it still has a chance, and Windows Phone because, while it's not extremely popular, it's cool and different enough that I think it's interesting.

You'll notice that Windows 8 as well as Windows Phone 7 and Windows Phone 8 are all listed; as this book is focused on mobile development, desktop operating systems are not addressed here.

It is important to note that when the Cordova project started, support for different mobile platforms was added primarily by an independent developer who was interested in the framework and wanted it running on the devices he or she was working on. Over time, as Cordova became more popular and mainstream, the mobile device vendors got onboard and helped with the implementation of Cordova on their platforms. The folks at BlackBerry are heavily involved with the BlackBerry implementation of Cordova, Intel is involved in the Tizen implementation, and Microsoft has been involved in the Windows implementation as well. What this means for developers is that even though Cordova is an open source project, the device OS companies are heavily involved and have a vested interest in making this project successful. Sadly, only Apple has chosen not to support the framework directly.

# Cordova License

Apache Cordova has been released under the Apache License, version 2.0. You can find more information about the license at www.apache.org/licenses/LICENSE-2.0.

# Working with Cordova

Now that you know a little bit about Cordova, let's dig into how to build mobile applications using the framework. In this section, I describe how to design your web application so it will run in the container, then explain how to use the available tools to package your web application into a native mobile application.

## Designing for the Container

Cordova applications are web applications running inside of a client-side native application container. Therefore, web applications running within a Cordova application leverage an HTML5 application structure rather than that of a traditional server-based web application.

With old school, traditional web applications, a web server serves up either static HTML pages or dynamic pages to the requesting user agent (the browser). With dynamic pages, a server-side language or scripting language is used to retrieve dynamic content (from a database, for example) and format it all into HTML before sending it to the browser. When the browser makes a request, the server retrieves the containing page and content, massages it all into HTML (or some variant such as XHTML), and sends it to the browser to be displayed.

In this example, the browser doesn't need any intelligence with regard to the content; it merely requests a page and the server does most of the work to deliver the requested content. On the browser, the application can leverage client-side JavaScript code to allow the user to interact with the content on the page, but in general, most of the work is done by the server.

With the advent of Web 2.0, a reduced load is placed on the web server and instead, JavaScript code running within the browser manages the requesting and presentation of data. The web server delivers an HTML-based wrapper for the web application and JavaScript code delivered with the page dynamically manages the content areas of the page, moving data in and out of sections of the page as needed.

What allowed Web 2.0 applications to be successful was the addition of the XMLHTTPRequest (XHR) API in JavaScript. This API allows a web application to submit asynchronous requests to a server and process the data whenever it returns from the server, without interrupting the user's activity within the application.

This approach allows for much more interesting web applications, applications that can easily look and feel like native desktop applications. The web server is still involved, serving up the pages and the content to the browser, but it does less direct manipulation of the data. Google Maps (maps.google.com) and Google Gmail (mail.google.com) are good examples of Web 2.0 applications available today.

Mobile devices need a slightly different approach. Web 1.0 and 2.0 technologies work great on smartphones, but Web 1.0 apps caused a lot of data to be transmitted between server and device, and Web 2.0 apps were cooler but still required constant network connectivity to operate.

With HTML5, web applications can make use of new capabilities that allow an application to operate more efficiently on a mobile device (or devices with limited connectivity), and they can use a client-side database to store application data. This functionality makes it easier for mobile devices to operate as they go in and out of wireless coverage. Additionally, HTML5 supports the addition of a manifest file that lists all of the files that comprise the web application. When the web application's index file loads, the browser reads the manifest file, retrieves all of the files listed in the manifest, and downloads them to the client device. If a mobile device were to lose network connectivity, then as long as the files listed in the manifest were available on-device, the application could continue working—using any data that might be stored locally.

To leverage these HTML5 capabilities, though, a web application must be written so it is able to run completely within the browser container (or, in the case of Cordova applications, within the Cordova application container). The index.html file is typically the only HTML file in the application, and the application's different "screens" are actually just different  containers that are switched in and out as needed by the application. HTML5 applications will still reach out to a server for data as needed, using XHR to request data asynchronously and store it locally.

> **Note**
>
> Web applications coded in PHP, ASP.NET, JSP, and the like will not run unmodified in the Cordova container. Those applications are designed to run on a web server, and the application's pages are preprocessed by special software running on the web server before output (typically HTML) is sent to the browser.
>
> I see a lot of support forum posts where developers ask how to get a server-based web application built with one of those technologies to run in the Cordova container. There's no processor for PHP, ASP.NET, and the other files available in the container, so it simply won't work. Those applications must be rewritten so that a standalone web application is running in the Cordova container that then reaches back to the web server for the dynamic content generated by the server using one of those technologies (PHP, ASP.NET, and so on).

Web developers must rethink their approach to web development to leverage these capabilities. Instead of retrieving a web application from a web server, the HTML5 application running in the Cordova container must be self-contained, making sure it has the files and data it needs to at least launch and display a UI before optionally reaching out to a remote server for additional content and data. As mentioned earlier, when a Cordova application launches, it loads the web application's startup page (typically index.html) and associated content (CSS files, JavaScript files) before passing control to the web app. In order for this method to work, the resources the app needs to start have to be located within the container.

Some Cordova developers load as little as possible within the container and, immediately after startup, run off to a server to get the "real" content for the application; I see their questions on the support forums all the time. This approach works, but it's not the best experience for users and may cause you problems with app store submissions—some smartphone platforms (Apple iOS, for example) don't like it when your app doesn't contain content and is merely a shell for a web application being hosted by a web server.

For a great presentation on how to write a web application for Apache Cordova, see if you can track down Lyza Danger Gardner's presentation from PhoneGap Day 2013—it was amazing. The videos and presentations haven't been posted yet; otherwise I'd provide a link for you here. She did a really good job of describing the approach you must take to craft a mobile web application that "works" in the Cordova container.

## Coding Cordova Applications

As mentioned previously, Cordova applications are built using normal, everyday web technologies such as HTML, CSS, and JavaScript. Whatever you want your application to do, if you can make it work using standard web technologies, you can make it work in a Cordova application. Cordova applications can do more than standard web applications through the specialized JavaScript libraries provided with the framework discussed earlier.

The Cordova project doesn't currently offer or support any special editor for writing your Cordova applications; you simply need to dig out your web content editor of choice and start coding. To keep things simple, you could use default tools like Notepad on Microsoft Windows or TextEdit on a Macintosh. You could even use something more sophisticated, such as Adobe Dreamweaver (www.adobe.com/products/dreamweaver.html) or the Eclipse IDE (www.eclipse.org).

Adobe, however, offers a free, open source code editor called Brackets (http://brackets.io) that I've been playing around with. It provides a nice, clean interface for coding your web applications. As it's an Adobe product, I expect that you'll see Cordova and/or PhoneGap integration capabilities in it before long.

For this book, I primarily coded using the open source Aptana studio (www.aptana.com); it's an open source Eclipse-based IDE tailored for web development. It's lighter-weight than Eclipse and allowed me to easily format the project source code for importing into this manuscript (two spaces instead of tabs).

## Building Cordova Applications

Once you have a completed web application, whether it uses any of the Cordova APIs or not, it has to be packaged into a native application that will run on-device. Each of the mobile device platforms supported by the Cordova project has its own proprietary tools for packaging or building native applications. To build a Cordova application for each supported mobile platform, the application's web content (the HTML, CSS, JavaScript, and other files that comprise the application) must be added to an application project appropriate for each mobile platform, then built using the platform's proprietary tools. What's challenging about this process is that each mobile platform uses completely different tools, and application projects use different configuration files and most likely a different project folder structure.

Additionally, some of the supported mobile platform development tools will run only on certain desktop operating systems. For example:

- The Android SDK runs on Linux, Microsoft Windows, and Macintosh OS X.
- The BlackBerry SDKs (they have several) run on Microsoft Windows and Macintosh OS X.
- The iOS SDK runs only on Macintosh OS X (no surprise there).
- The Windows Phone SDK runs only on Microsoft Windows (no surprise there either).

What this means for developers is that to work with the most popular smartphones (you can argue with me later about whether BlackBerry or Windows Phone are popular), you have to have, at a minimum, development systems running both Windows and Macintosh OS X. For my Cordova development work, I use a loaded Macintosh Mini (Intel I7 quad core, 8GB memory, 500GB hard drive) running VMware Fusion (www.vmware.com/products/fusion/overview.html); this allows me to easily run both Windows-based and Macintosh-based SDKs and seamlessly share files between both OSs.

In the old days of Cordova development (back in the PhoneGap 1.0 timeframe—back when I started my previous book), you would use IDE plugins (on Android, iOS, and Windows Phone) and command-line tools (on Android and BlackBerry), or you would copy a sample application (on bada, Symbian and webOS) to create a new project. You would start with one of the supported platforms, write the appropriate web content, then package and test the application using the selected platform's SDK. Once you had it all working correctly, you would copy the web content over to a new project for one of the supported platforms and repeat the process. There was little consistency in project folder structure, framework JavaScript files (they had different file names on some platforms and were markedly different for each), and build process across mobile device platforms. This process is highlighted in Figure 1.4.

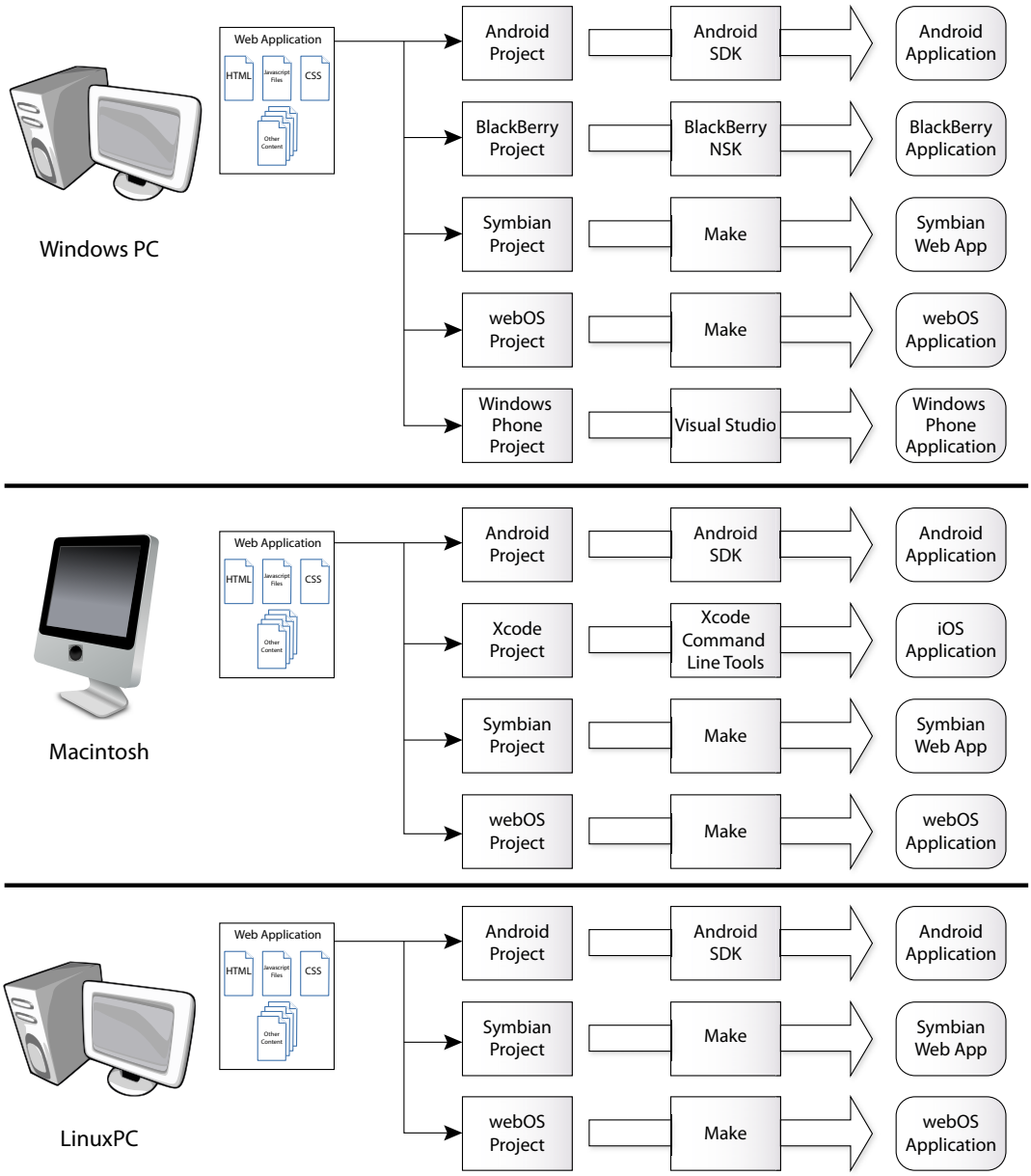



**Figure 1.4** Cordova Application Build Process

To make things easier, in later versions of the framework, the Cordova development team scrapped the IDE plugins and implemented a command-line interface for projects across a wider range of supported mobile device platforms. For some of the mobile platforms (Symbian and webOS, for example), you may still need to copy a sample project to get started, but the process is simpler and more consistent now. You use the command-line tools to create new projects, manage (add, remove, list, update) plugins, and build then test applications using the device emulators and simulators. The Cordova command-line tools are described in detail in Chapter 4, "Using the Cordova Command-Line Interface." You can still do it by hand, but the command-line tools make it much easier.

Adobe also offers a cloud-based packaging service for PhoneGap applications called PhoneGap Build. This service allows you to upload a web application to the Build service servers, and it packages the application into a PhoneGap container for the following mobile device platforms:

- Android
- BlackBerry
- iOS
- Symbian
- webOS
- Windows Phone

The PhoneGap Build service is a commercial offering from Adobe; it's free for open source projects, but paid subscription plans are available for developers building private applications. PhoneGap Build is covered in Chapter 11.

## Cordova Plugins

As with any developer tool, there are times when the base functionality provided by the solution just isn't enough for your particular needs. Every developer, it seems, wants that one thing that's not already in there. For those cases, the Cordova development team added the ability to extend Cordova applications via plugins. Initially, plugins were hacks inserted in the Cordova container by developers, but over time the Cordova project team solidified a specification for plugins. They even created tools to help developers manage the plugins used in a Cordova application.

With that in place, developers started creating all sorts of plugins. There's a Facebook plugin (https://github.com/phonegap/phonegap-facebook-plugin), Urban Airship plugin for push notifications (http://docs.urbanairship.com/build/phonegap.html), and one of the most popular plugins, ChildBrowser, actually became a part of the core Cordova API as inAppBrowser. The Cordova development team eventually even migrated all the core Cordova APIs into plugins. I'll show you how to create your own Cordova plugin in Chapter 13, "Creating Cordova Plugins," but for now you can browse a repository of Cordova plugins here: https://github.com/phonegap/phonegap-plugins.

## Putting Cordova to Best Use

There are people who try to categorize the types of apps you should or should not build with a hybrid container. I really don't approve of that approach. The hybrid application approach used by Cordova has strengths and weaknesses—and you have to assess your particular mobile application needs against them and decide on a case-by-case basis whether the Cordova approach is the right one for your application.

Web applications are likely going to be slower than native applications, but an inexperienced developer can easily build a native application that performs poorly (without even trying very hard, it seems). You can read about Facebook ditching HTML5 and switching to native for performance reasons (http://techcrunch.com/2012/12/13/facebook-android-faster), but then you can read how Sencha was able to build a suitably fast web version of the Facebook application using its HTML5 framework (www.sencha.com/blog/the-making-of-fastbook-an-html5-love-story).

Hybrid applications may be slower than native mobile applications in most cases; it's just the nature of the technology being used. There are reports out there indicating that in more recent versions of iOS, Apple limited the performance of JavaScript running in a web view (instead of in the browser). On the other hand, many games have been created using Cordova, so if Cordova performs well enough for games, it should be okay for many of the applications you need to write.

A lot of commercial applications available today were built using Cordova; you can find a list of many of the applications on the PhoneGap website at www.phonegap.com/apps. The framework is used primarily for consumer applications (games, social media applications, utilities, productivity applications, and more) today, but more and more enterprises are looking at Cordova for their employee-facing applications as well. There are many commercial mobile development tools with Cordova inside and likely more in the works.

So where should you use Cordova? Use it anywhere you feel comfortable using it. Do some proof-of-concepts of your application's most critical and complicated features to make sure you can implement it in HTML and get the performance you need from the framework.

The issues I see are these:

- *Can you implement the app you want using HTML?*—There are so many JavaScript and CSS frameworks out there to help simplify mobile web development that most things a developer wants to do in an application can be done in HTML.
- *Can you get the performance you need from a hybrid application?*—That you'll have to prove with some testing.

So where does it fail?

Early on, Cordova would fail when you wanted to build an application that needed access to a native API that wasn't already exposed through the container. Nowadays, with all of the plugins available, you're likely to find one that suits your application's requirements—if not, write your own plugin and donate it to the community. Cordova doesn't (today) have access to the Calendar or email client running on the device, so if your application has requirements for those features, you may be out of luck—but don't forget about plugins.

If your application needs to interface with a particular piece of hardware (either inside the device or outside), Cordova might not be the best choice for you unless there's a plugin for the hardware.

If your application requires heavy-duty offline capabilities such as a large database and offline synchronization, you could run into issues. However, there are several HTML5-based sync engines that should work within a Cordova container, and there are SQLite plugins for Android (https://github.com/pgsqlite/PG-SQLitePlugin-Android) and iOS (https://github.com/pgsqlite/PG-SQLitePlugin-iOS).

Try it out with some common use cases for your application and see what happens. If you are building an application with a large database and a bunch of heavy data-entry forms, this might not be the best choice for Cordova, but you'll have to try it first and see.

# Getting Support

One of the things corporations worry about is getting support for the software products they use for their business applications. An open source project such as Linux wouldn't be as popular with companies if there weren't support options available to them. Since commercial support for Linux is provided by a wide range of companies, including Red Hat, Canonical, SUSE, and others,

organizations are much more willing to run their businesses on open source software products. Cordova is no different.

Although there is no Cordova support area, the best place I've found for getting support for the framework is the PhoneGap area in Google Groups (http://groups.google.com/group/phonegap). A lot of experienced developers monitor the list of questions, and you can usually get an answer there pretty quickly. You can even find me up there from time to time answering questions that I can.

There's also a support forum for PhoneGap Build located at http://community.phonegap.com /nitobi/products/nitobi_phonegap_build. The forums are supposed to be for questions related to the PhoneGap Build service, but a lot of developers incorrectly use the forum for general PhoneGap development questions as well. Please do me a favor and use the Build forums for PhoneGap Build–related questions and the Google Groups area for questions on any other PhoneGap-related questions. In my experience, you'll get a faster and sometimes better response to a PhoneGap development question in Google Groups.

In early 2011, Nitobi announced availability of commercial support options for PhoneGap, and Adobe has continued the offering after the acquisition. Support is offered at different levels (from Basic, currently at $249US per year, to Corporate, at $20,000US per year). There's even an Enterprise support option available, pricing for which is not publically available, so you will need to contact Adobe for pricing. Information on the available support options for PhoneGap can be found at www.phonegap.com/support.

# Resources

There are many places online where you can find information about how to work with the Cordova and PhoneGap frameworks. Table 1.1 lists the different websites where you can find information about Apache Cordova and Adobe PhoneGap.

**Table 1.1 Available Resources**

| Resource | Link(s) |
|---|---|
| Cordova Website | http://cordova.io or http://cordova.apache.org (they both point to the same site). |
| Cordova Documentation | http://docs.cordova.io |
| Cordova Wiki | http://wiki.cordova.io |
| Cordova Issue Tracker | https://issues.apache.org/jira/browse/CB |
| Cordova Mailing Lists | http://cordova.apache.org/#mailing-list |
| Cordova Twitter Account | http://twitter.com/apachecordova |
| PhoneGap Website | http://www.phonegap.com |
| PhoneGap Wiki | http://wiki.phonegap.com |
| PhoneGap Blog | http://www.phonegap.com/blog |
| PhoneGap Twitter Account | https://twitter.com/phonegap |

To stay informed about what's going on with the project, you can sign up for the mailing lists at http://cordova.apache.org/#mailing-list. If you have some extra time, it is fun to read through the emails as the development team discusses the implementation of a new feature or tracks down a bug. The dev

mailing list is used by the developers of the Cordova framework to discuss issues and make decisions about the Cordova implementation. The Commits mailing list is for tracking commit logs for the Cordova repositories when new or updated code is added to a version of the framework. The Issues mailing list is for conversations around bug and feature requests submitted to the Cordova Jira issue and bug tracking system at http://issues.apache.org/jira/browse/CB.

> **Caution**
>
> Please don't use the Dev list to ask questions about Cordova development—use Google Groups instead. The dev lists are for the developers building Cordova to discuss feature implementation and so on, and the Google Groups is set up specifically to provide Cordova developers with answers to their questions.

Where you'll spend the majority of your time will be on the Apache Cordova Documentation site, which is shown in Figure 1.5. The site contains a complete reference to all of the Cordova APIs plus additional guides you'll need as you work with the framework.

The documentation is far better than it was when I wrote *PhoneGap Essentials*, so it's probably a good place to start for anything Cordova related.

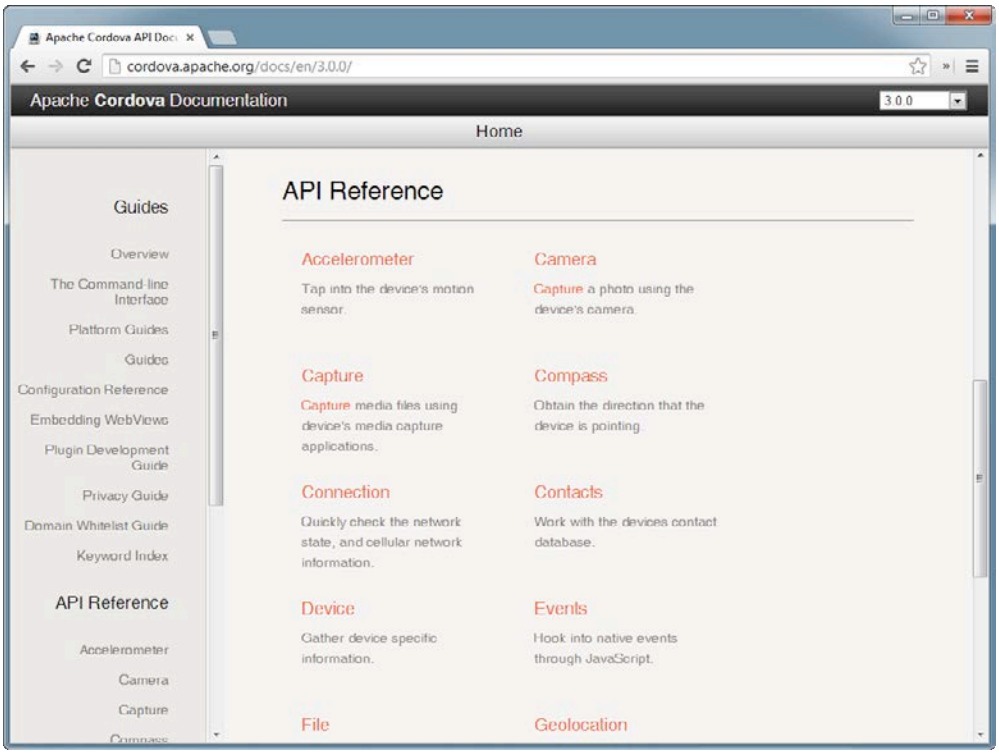



**Figure 1.5** Apache Cordova Documentation

The API reference shown in Figure 1.5 includes a complete reference for all of the methods, properties, and events for each of the Cordova APIs. On the API pages, you'll also find sample source code and additional information you will need to make use of the APIs.

While you're looking at the Documentation site, if you scroll down within either the left or right side of the page, you will see the list of guides shown in Figure 1.6. These guides contain a lot of useful information a developer needs to work with the framework, including how to create plugins, using the command-line tools, and most important, the getting-started guides for each of the supported mobile device platforms.

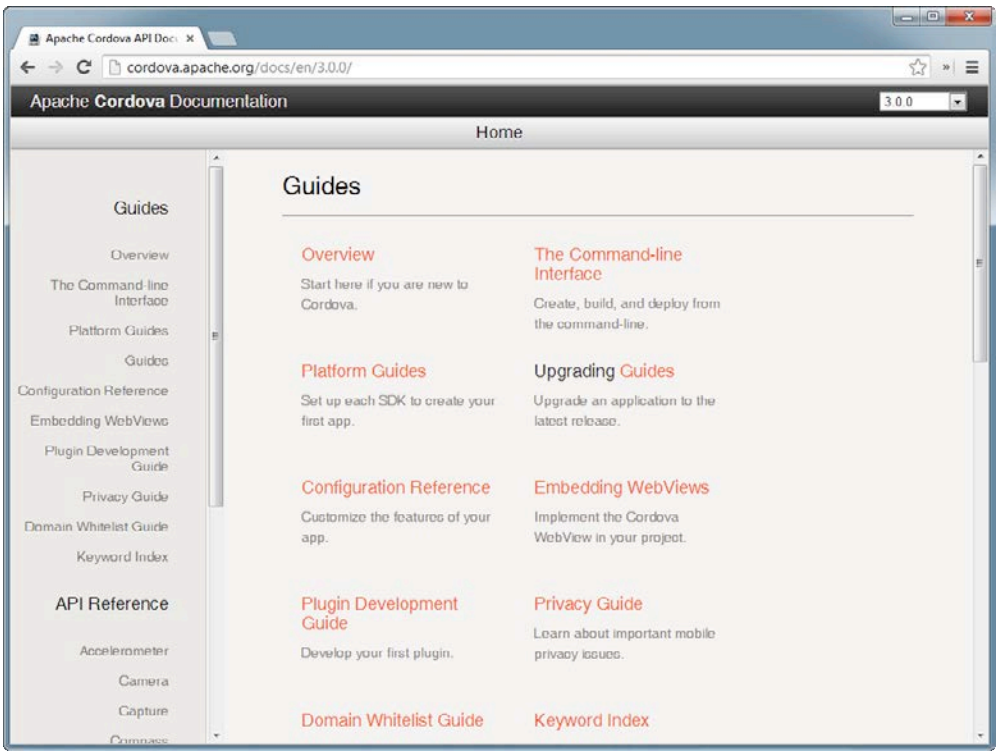



**Figure 1.6** Cordova Documentation: Guides Section

# Hybrid Application Frameworks

The hybrid application approach Cordova uses is not unique to the market. The Cordova project may have started the trend, but many other products on the market use a similar approach. Here is a list of some of the products that use a hybrid applications approach. Some are like Cordova, and others are built with Cordova inside. This is only a subset of the available options in the hybrid mobile application space:

- Appcelerator Titanium
- AT&T WorkBench and Antenna Software Volt
- BlackBerry WebWorks

- IBM Worklight
- Oracle Application Development Framework (ADF) Mobile
- Salesforce Touch
- SAP Mobile Platform (this is the product I work on)
- Strobe (formerly SproutCore and now part of Facebook)
- Tiggr

## Wrap-Up

In this chapter, I've given you a thorough (I think) overview of what Apache Cordova and Adobe PhoneGap are and how to work with the framework and the tools that are available. You now know a little bit of the history of the framework, where to get support, and what the future looks like for Cordova. Throughout the rest of the book, I'll start digging into the technical details of how all of this works—we'll dig into the tools and how to use them, talk about developing for some of the most popular mobile platforms, and even build an application or two.

# 2

# Installing the Cordova and PhoneGap Frameworks

The traditional way of installing Apache Cordova or Adobe PhoneGap is to download the framework's files and optionally (depending on which version of the framework you download) perform some additional configuration. Although Apache Cordova and Adobe PhoneGap are, at their core, very similar (the differences between the two are described in Chapter 1), the steps performed to get your development environment ready to work with the frameworks are different.

In this chapter, I walk you through the process of downloading and setting up the framework. Subsequent chapters illustrate how to install and use the command-line tools as well as how to set up and use development environments for several of the Cordova-supported mobile device platforms.

> **Note**
>
> You don't really need to do the stuff in this chapter if you plan to use the Cordova CLI or the PhoneGap Build service (described in Chapter 11, "Using PhoneGap Build") exclusively for your Cordova development projects. However, there's still value in having the framework installed so you can have a local copy of the documentation or source code.

## Installing Apache Cordova

Apache Cordova is an open source project, so the framework files are a free download from the Apache Cordova project website. Point your browser of choice to http://cordova.io. You will be presented with a page similar to the one shown in Figure 2.1. The big download link at the middle-right side of the page takes you directly to the download section of the page shown in Figure 2.2.

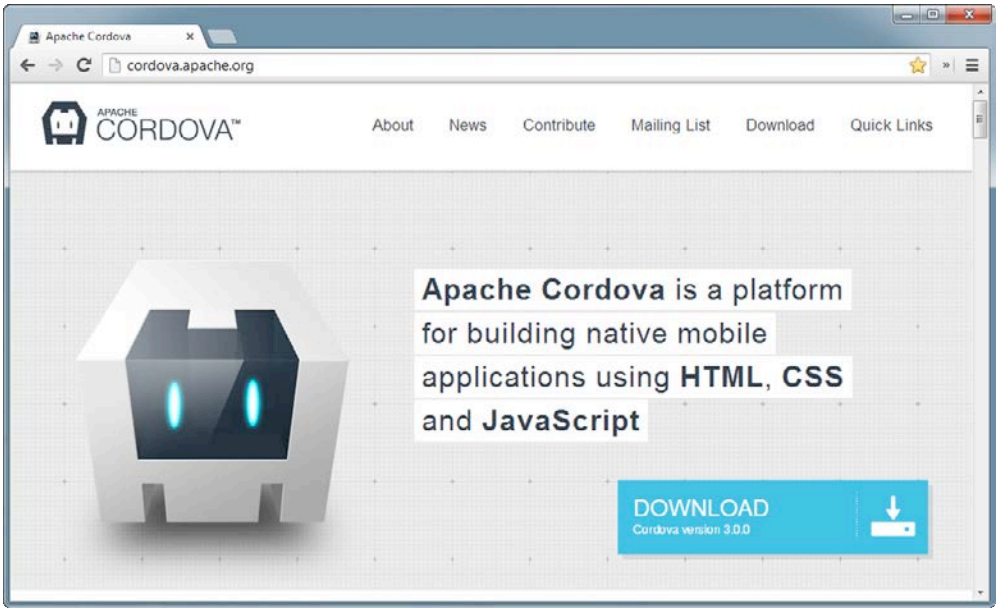



**Figure 2.1** Apache Cordova Project Home Page

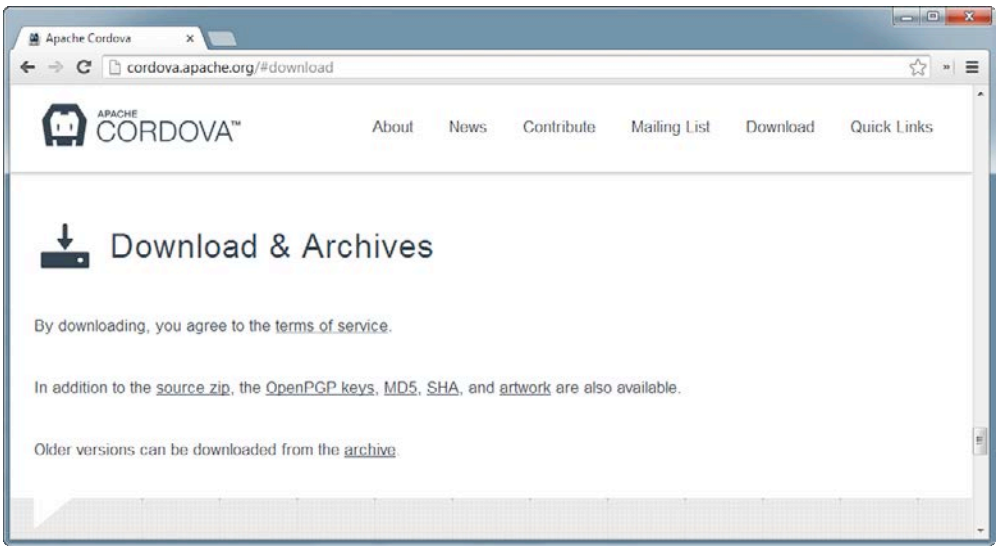



**Figure 2.2** Apache Cordova Current Version Download Folder

Click on the source.zip link to download the most recent version of the Cordova framework. You will be taken to an Apache download page, which will allow you to select the mirror from which to perform the software download. To download an earlier version of the framework, click the archive link shown at the bottom of Figure 2.2. The website will open the archives page shown in Figure 2.3.

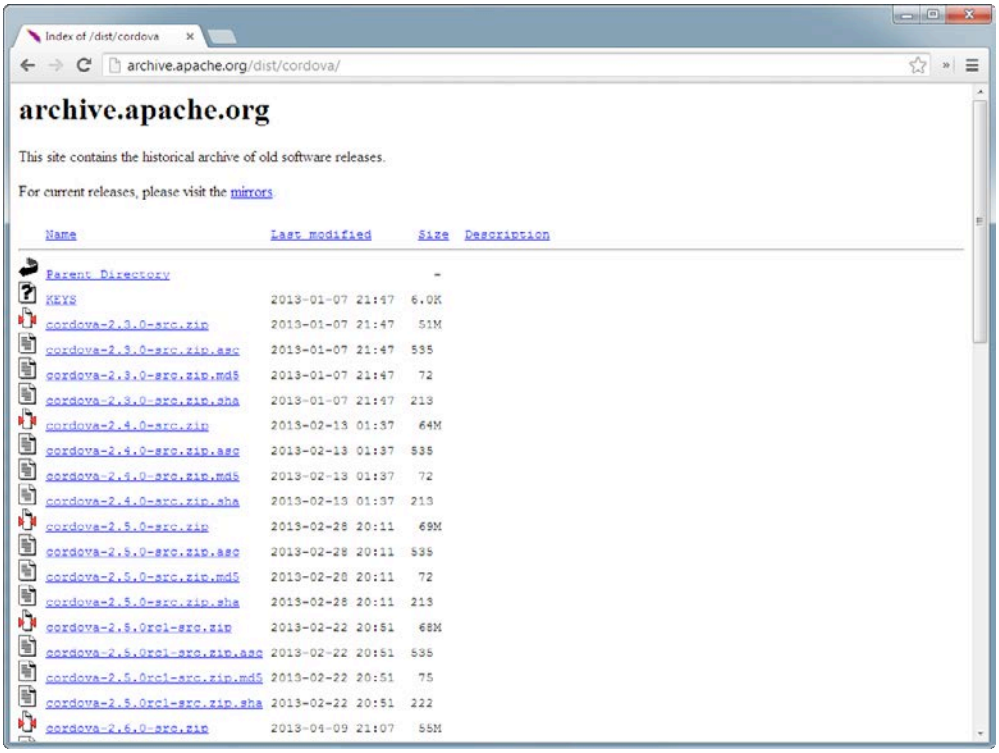



**Figure 2.3** Cordova Archives

The listing shows the framework download file (the .zip file shown in the figure) plus some output files from cryptographic hash functions used to help identify whether the .zip file has been modified. As you can see from the screenshot in Figure 2.2, the file access is provided over an HTTPS connection, so it is not likely that your download can be intercepted and a bogus file substituted. However, if you're appropriately paranoid, you can validate the source files using the keys provided. The .sha file is generated using the Secure Hash Algorithm (SHA), the .md5 file is generated using Message Digest Algorithm-5 (MD5), and the .asc file contains the PGP signature for the file.

### Why Would You Not Want the Latest Version of the Framework?

In most cases, you will likely want to start with the current version of Apache Cordova or Adobe PhoneGap. The current version is the most up to date and will have the most bug fixes and most recent documentation. You would want to go into the archives if you needed to support an obsolete mobile device platform or an older mobile device OS version that is no longer supported

In Apache Cordova 3.0, for example, the project team dropped support for BlackBerry 7 (and older) devices. The folks at BlackBerry did not have any plans for future updates to the BlackBerr WebWorks platform for BlackBerry 7 and older devices, and as Cordova leverages WebWorks fo its BlackBerry support, no further work would be done for that particular OS family. The BlackBer developers working on Cordova were focused on BlackBerry 10 and BlackBerry PlayBook support, so the older stuff was dropped from the framework.

If your development project is targeting BlackBerry 7 (or older) devices, grab an older version of the framework from the archives. Anything before Cordova 3.0 should include BlackBerry 7 support.

If downloading the file using Safari on later versions of Macintosh OS X, the files will be automatically extracted after they download. On Windows, or if you are using a different browser on your Macintosh, you'll have to manually extract the files.

> **Note**
>
> The Cordova framework .zip file no longer contains files for several of the supported mobile device platforms, such as bada, Tizen, Symbian, webOS, and even Windows Phone 7. Even though the platforms are still supported, the files for those particular OSs were removed after the Cordova 2.5 release. You will have to manually download the additional platforms if needed.
>
> A quick search on Google told me I could find the Cordova files for bada at https://github.com/apache/cordova-bada, for Tizen at https://github.com/apache/cordova-tizen, and for webOs at https://github.com/apache/cordova-webos.

There is no installer for the Apache Cordova files; once extracted, the Cordova download will expand to the file listing shown in Figure 2.4. As you can see, the download includes a separate .zip file for each of the mobile device platforms supported by Cordova plus some additional files, which are described in Table 2.1.

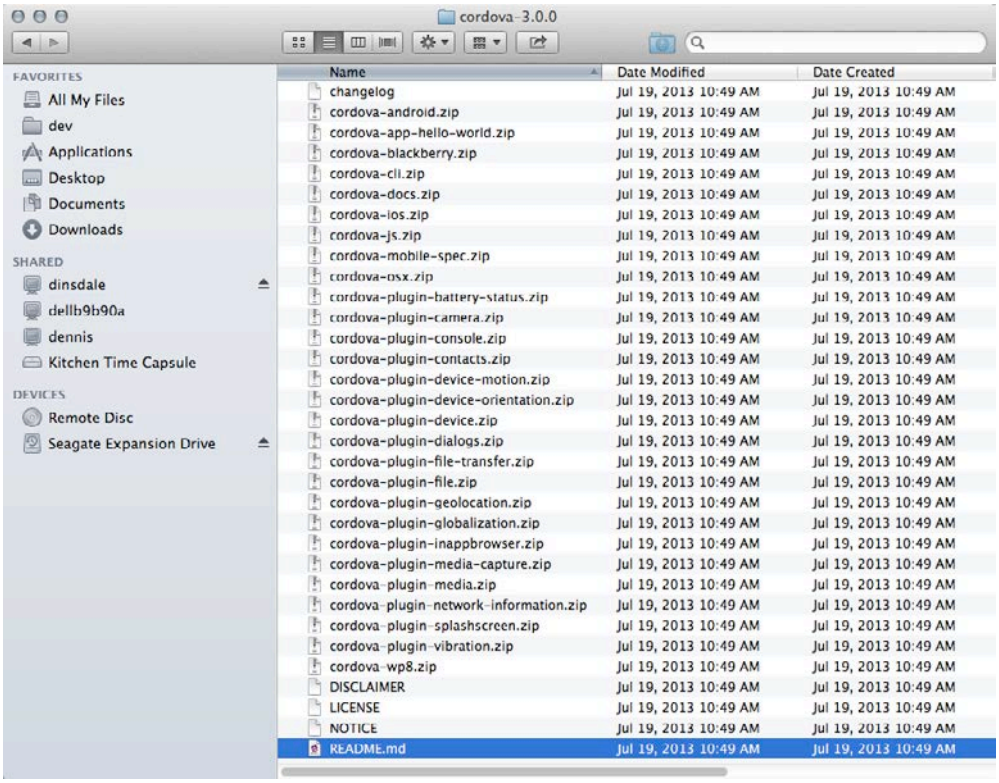



**Figure 2.4** Apache Cordova Download File Listing

Table 2.1 **Cordova Archive .zip File: List of Additional .zip Files**

| .zip File | Description |
|---|---|
| cordova-android.zip | Contains the files needed to build Cordova applications for the Android platform. |
| cordova-app-hello-world.zip | Contains a complete Cordova reference Hello World web application. The application is used as an illustration of how to set up a Cordova-compatible web application. The structure of a Cordova web application is discussed in Chapter 5, “Anatomy of a Cordova Application.” |
| cordova-blackberry.zip | Contains the files needed to build Cordova applications for the supported BlackBerry platforms: BlackBerry 10 and BlackBerry PlayBook. Remember, support for BlackBerry 7 and earlier devices was dropped in Cordova 3.0. |
| cordova-cli.zip | Contains the source files for the new Node.js-based command-line interface (CLI) for creating, managing and testing Cordova applications. These tools are described in detail in Chapter 3, “Installing the Cordova Command-Line Interface,” and Chapter 4, “Using the Cordova Command-Line Interface.” |
| cordova-docs.zip | Contains the complete Cordova documentation files (in HTML format) in English, Japanese, and Korean. For some reason, the project team chose to include all of the documentation back to Cordova 1.5 in this folder structure. |
| cordova-ios.zip | Contains the files needed to build Cordova applications for the iOS platform. |
| cordova-js.zip | Contains the source files for the Cordova JavaScript layer. |
| cordova-mobile-spec.zip | Contains the files associated with the Mobile Spec Suite—a suite of tests that are run to validate a Cordova version. |
| cordova-osx.zip | Contains the files needed to build Cordova applications for the Macintosh OS X platform. |
| cordova-plugin-* | Separate folders for each of the Cordova API plugins. When using the CLI and adding a plugin to a project, you can point the CLI to these file folders for the plugin files added to your project. |
| cordova-wp8.zip | Contains the files needed to build Cordova applications for the Windows Phone 8 platform. |

In order to build applications locally for each mobile device platform, you need to extract the .zip file(s) for each platform. Each .zip file contains a text file called readme.md; you can open the file in any standard text editor and quickly learn what the files in the folder are used for. In most cases, there are simply instructions for how to use the files, but in some cases, such as with the Android Cordova implementation, there are additional instructions you will need to follow to complete the setup.

> **Tip**
>
> On Macintosh, an .md file will open automatically in the TextEdit application; just double-click on it, and it opens. The default Windows text editor (notepad.exe) doesn't understand the .md file extension, so you can't just double-click on the file in Windows Explorer to open it. You must manually open the file from Notepad, or you may want to use a third-party text editor such as Notepad++ (http://notepad-plus-plus.org), which registers a right-click menu item that allows you to right-click on an .md file and edit it directly in Notepad++.
>
> An .md file is a text file created using one of the dialects of the Markdown language. Markdown is essentially a plain text format that can be easily converted to HTML.

To configure development environments for the different supported mobile device platforms, you need to install the platform's native software development kit (SDK) or integrated development environments (IDE). For some platforms, you may also need additional tools such as Java, Ant, make, and Cygwin. Except for the cost of joining the appropriate developer programs, pretty much all of the tools you need for local Cordova development are free.

# Ant-Based Command-Line Interface

As mentioned in Chapter 1, Cordova 3.0 added a new suite of command-line tools to the framework. When you expand all of the Cordova framework files for the different mobile device platforms, you may run across information in the readme.md about using Apache Ant (http://ant.apache.org) to create new Cordova projects and to debug Cordova applications in a device simulator or emulator.

In older versions of Cordova, you had IDE tools (like plugins for Xcode for iOS and Eclipse for Android) plus the CLI at your disposal. Unfortunately, the Ant-based CLI was implemented inconsistently across the different mobile platforms, so you couldn't use it everywhere, and each implementation was in a different place, so you didn't have a single tool you could use anywhere. The Node.js-based CLI (described in Chapter 3) replaces the Ant-based process and is implemented more consistently across platforms.

For that reason, I don't cover the Ant-based tools in this book and instead focus on the Node.js-based tools. Over time, all of the Ant-based tools should be migrated to Node.js.

# Installing Adobe PhoneGap

The installation process for Adobe PhoneGap is a little different. To access the installation instructions and archives, point your browser of choice to www.phonegap.com. You will be presented with a page similar to the one shown in Figure 2.5.

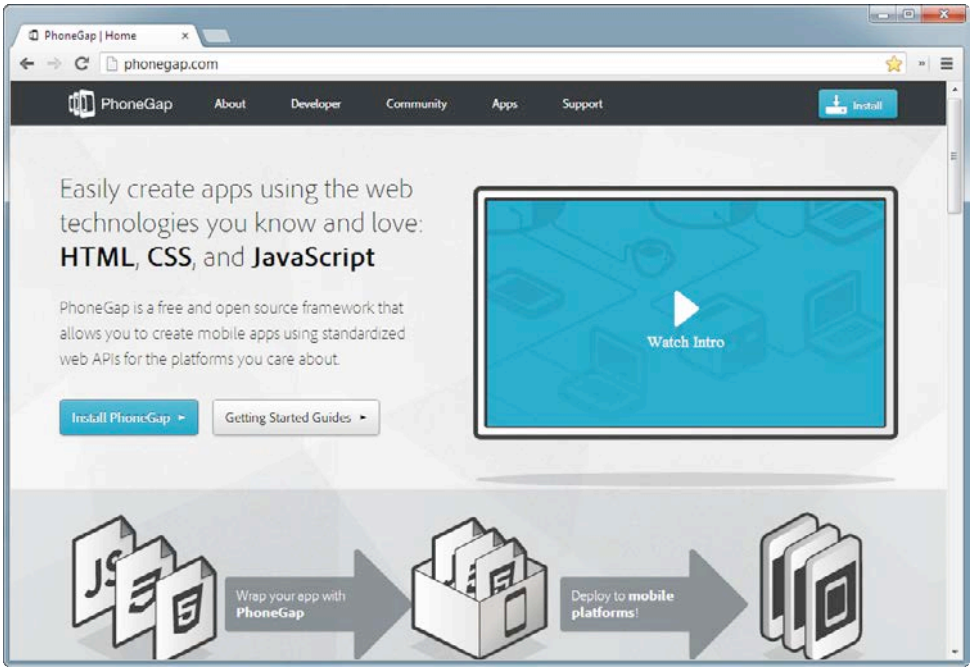



**Figure 2.5** PhoneGap Installation Instructions

Beginning with PhoneGap 3.0, Adobe stopped making a specific download package and instead does everything through a Node.js package. So, when you click the Download PhoneGap button shown in the figure, you will be taken to the installation instructions shown in Figure 2.6.

Installation is similar to what is described in greater detail in Chapter 3. To install PhoneGap, the PhoneGap website tells you that you simply have to install NodeJS (www.nodejs.org), then open a command prompt and type the following command:

```
npm install -g phonegap
```

Experience tells me that there's a lot more to it, so jump to the next chapter to read about all of the requirements and issues you will face during the installation and afterwards.

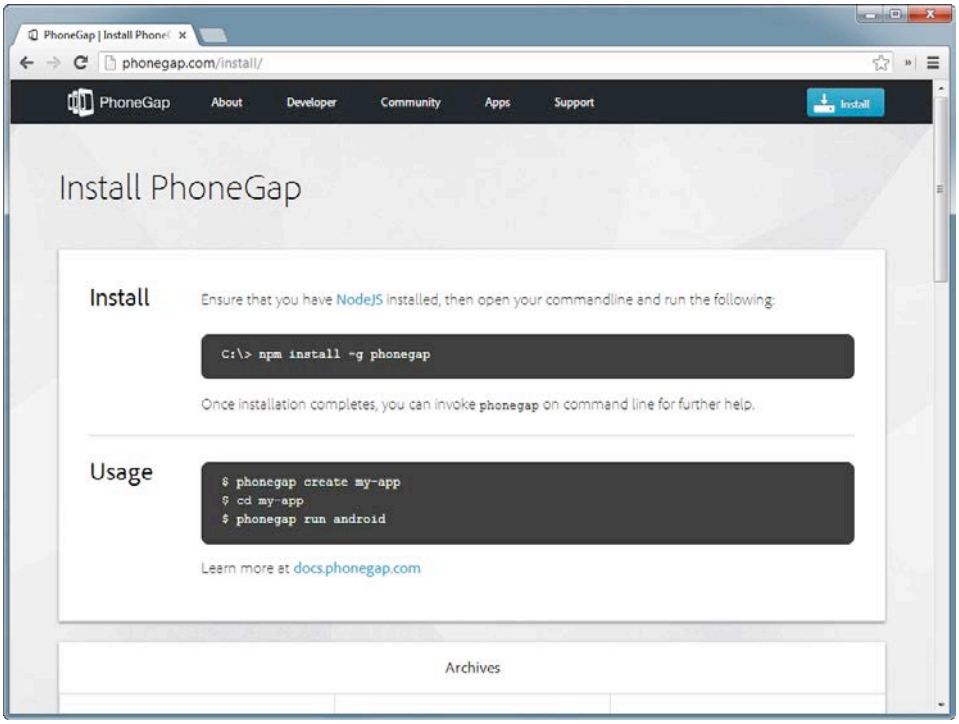



**Figure 2.6** PhoneGap Downloads Page

> **Tip**
>
> The PhoneGap documentation is essentially (most likely exactly—I didn't check) the same as the Cordova documentation, so there's little need to check both sites for information. I used the PhoneGap documentation only when I was looking for information about some feature that is unique to the PhoneGap implementation of Apache Cordova.

If you're looking for an older version of PhoneGap, you can scroll down on the page shown in Figure 2.6 and access the downloads for most earlier versions of the framework.

# Wrap-Up

In this chapter, I showed you how to download and install the Apache Cordova and Adobe PhoneGap frameworks. We can't really do anything yet with what we have installed, but in the chapters that follow, I show you how to develop and test your Cordova applications plus how to configure and use a local development environment for several of the most popular mobile device platforms. Immediately following those chapters, I show you how to use the PhoneGap Build service to help dramatically simplify all of this work.

# 3

# Installing the Cordova Command-Line Interface

To make it easier for developers to manage their projects, the Cordova project team built a single, unified command-line interface (CLI) that is expected to eventually work across all of the Cordova-supported mobile device platforms. This chapter illustrates how to install the Cordova CLI. The following chapter shows you how to use it to manage your Cordova projects.

The PhoneGap CLI is essentially the Cordova CLI with some additional commands added on. Almost everything you learn in this chapter easily applies to the PhoneGap CLI as well. You should refer to the PhoneGap documentation at http://docs.phonegap.com/en/3.0.0/index.html for information about the additional commands the PhoneGap CLI adds.

## Installing the CLI

The Cordova CLI is built using JavaScript code that is exposed through the Node JavaScript runtime engine (www.nodejs.org). Because it is built using JavaScript, it can essentially run anywhere that Node.js is available (Linux, Macintosh OS X, and Microsoft Windows). The instructions that follow will help you get the CLI installed on your development workstation; there are separate instructions for Macintosh OS X and Microsoft Windows; I won't be covering the Linux install.

> **Note**
>
> The CLI is a brand new feature for Cordova, so it's a little finicky about your development system's configuration. Be sure to follow all of the steps in the CLI readme.md. In the sections that follow, I show you all of the steps I had to complete to get this to work on my Macintosh and Windows systems. In your environment, you may or may not need to follow these steps.
>
> Be prepared to update all of your mobile SDKs and development tools, and be sure to follow all of the instructions. Be prepared to be patient because the CLI has a lot of required libraries that must be installed with it.

The CLI has to be able to create new Cordova projects, so it provides the necessary project files, plus it has to interface with the native SDKs to build projects and load applications in an emulator or simulator. Therefore, you have to make sure that the appropriate SDKs are installed (depending on which platforms you're working with) and that the SDKs are visible to the CLI. For BlackBerry, iOS, and Windows, for example, the SDKs are installed on the system, so the CLI can easily find what it

needs. For Android, there is no installer, so you have to manually configure your system so the CLI can find the SDK and associated tools.

In the sections that follow, I show you how to install the appropriate tools for Android, BlackBerry 10, iOS, and Windows Phone required to use the Cordova CLI on Microsoft Windows and Mac OS X. Chapters 7 through 10 show you in detail how to use the platform-specific capabilities for the different mobile OSs listed. So, if you're going to do some Cordova development for a particular mobile OS, read through the instructions here, then take a look at the appropriate chapter for your particular OS (Chapter 7 for Android, Chapter 8 for BlackBerry 10, Chapter 9 for iOS, and Chapter 10 for Windows Phone 8).

## Android Development Tools

One of the requirements for the Android SDK is the Java Development Kit (JDK). On Macintosh, you should have Java already available, although you may have to install it first. Open a terminal window and type `javac` at the command prompt, and press Enter. If you see a bunch of help text scroll by, you're all set. If not, you'll see instructions on how to download and install the appropriate Java files for your system.

On Windows, you need to point your browser of choice to http://java.oracle.com and download the JDK. On the Java home page, select Java SE, then download and install the latest version of the JDK.

At the conclusion of the JDK installation, there's still one more step to complete: you need to define the `JAVA_HOME` environment variable and point it to where the JDK is installed. To create the `JAVA_HOME` environment variable, assuming you're running Windows 7, right-click on the Computer icon (it used to be called My Computer in older versions of Windows), then select Properties. On the dialog that appears, select Advanced system settings, as highlighted on the left side of Figure 3.1.

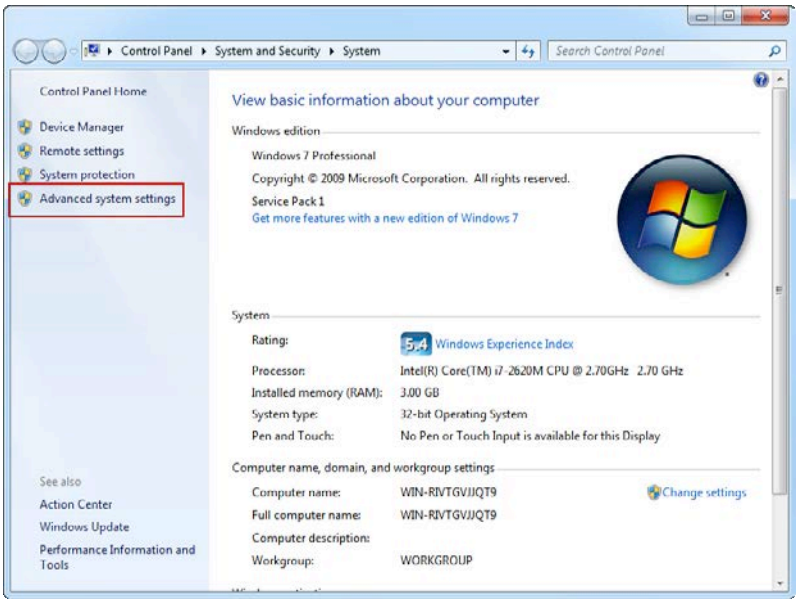



**Figure 3.1** Windows Control Panel: System Settings

In the dialog that appears, select the Advanced tab, as shown in Figure 3.2, and click the Environment Variables… button highlighted in the figure.

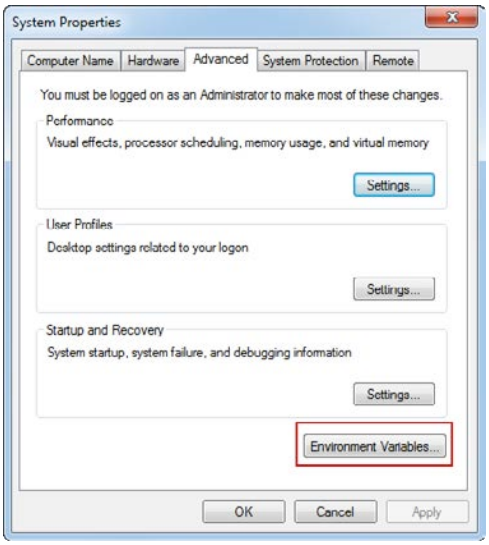



**Figure 3.2** Windows System Properties

In the dialog that appears, click the New button to add a user variable called `JAVA_HOME` that points to where the JDK software is located, as shown in Figure 3.3.

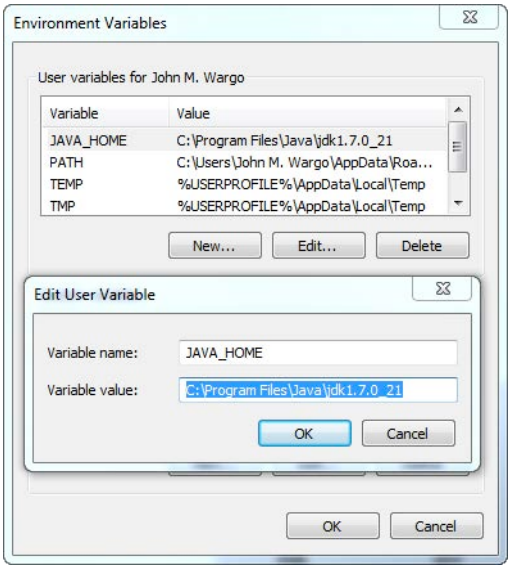



**Figure 3.3** Adding the JAVA_HOME User Variable

To test the configuration, save your changes, open a new terminal window and type `set`, then press Enter. You should see a long list of environment variables; look for an entry for your new `JAVA_HOME` variable.

With Java installed and configured, it's time to install the Android SDK. Open your browser of choice and point it to http://developers.android.com. On the Android developer home page, follow the links to download the Android SDK; on the current site, it is labeled "Get the SDK."

There is no installer for the SDK; it ships as a .zip file, so after it downloads, you have to extract the files to a location that makes sense for you. On Macintosh OS X, when you download .zip files using the Safari browser, the files are automatically extracted to the Downloads folder for you. Some developers might drag the extracted folder to the /Applications folder, but on my system I simply left everything in Downloads.

The SDK download includes the tools you need to build and emulate Android applications, including a preconfigured version of Eclipse, but does not contain all of the SDK libraries and device emulators you might need for your applications. There are additional steps you need to follow to download the SDK libraries and device emulators you will need—follow the instructions located at http://developer.android.com/sdk/installing/bundle.html to complete this step.

Because there's no installer for the Android SDK, the CLI won't know where to find Android SDK tools, so you have to add some folders to the system path. Specifically, the CLI needs to know where to find the SDK platform-tools and tools folders.

On Macintosh, there are several ways to update the system path; for my installation, I used TextMate to modify the paths file by issuing the following command:

```
mate /etc/paths
```

Figure 3.4 shows the file open in the editor. I left the existing file contents alone and the file paths for the platform-tools and tools folders to the file as separate lines.

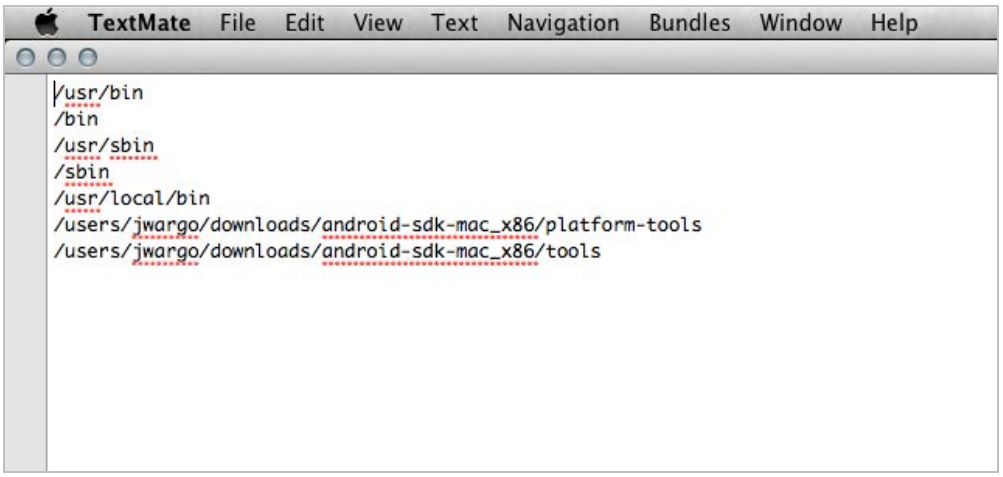



**Figure 3.4** Editing the Macintosh OS X paths file

On Windows, you can access the settings you need from the Control Panel. The easiest way to get there is to right-click on the Computer icon in the Start Menu then select Properties. In the window that appears, select Advanced system settings, as shown in Figure 3.1. In the Environment Variables dialog, select the PATH variable and click the Edit button, or simply double-click on the PATH entry. Append the folder paths for the Android SDK platform-tools and tools folders to the existing path environment variable, placing a semicolon between each entry, as shown in Figure 3.5.

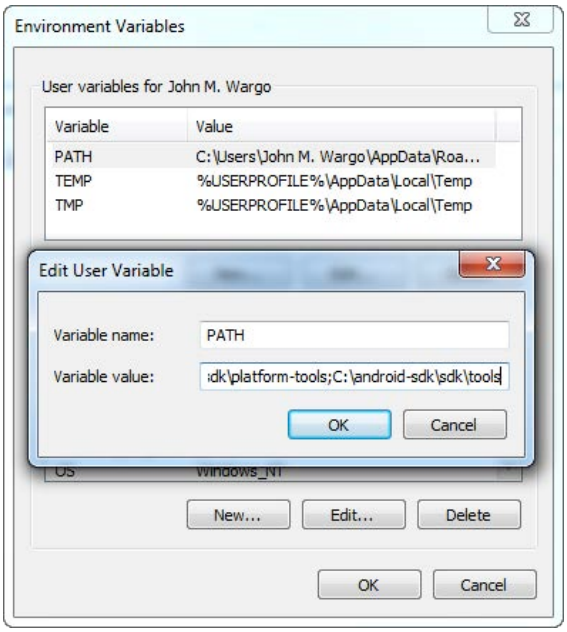



**Figure 3.5** Windows Environment Variables

After saving your changes, you should be able to open a terminal window and type `android` to run the Android SDK Manager, as shown in Figure 3.6. The SDK Manager should look the same regardless of whether you're running on Macintosh OS or Windows.

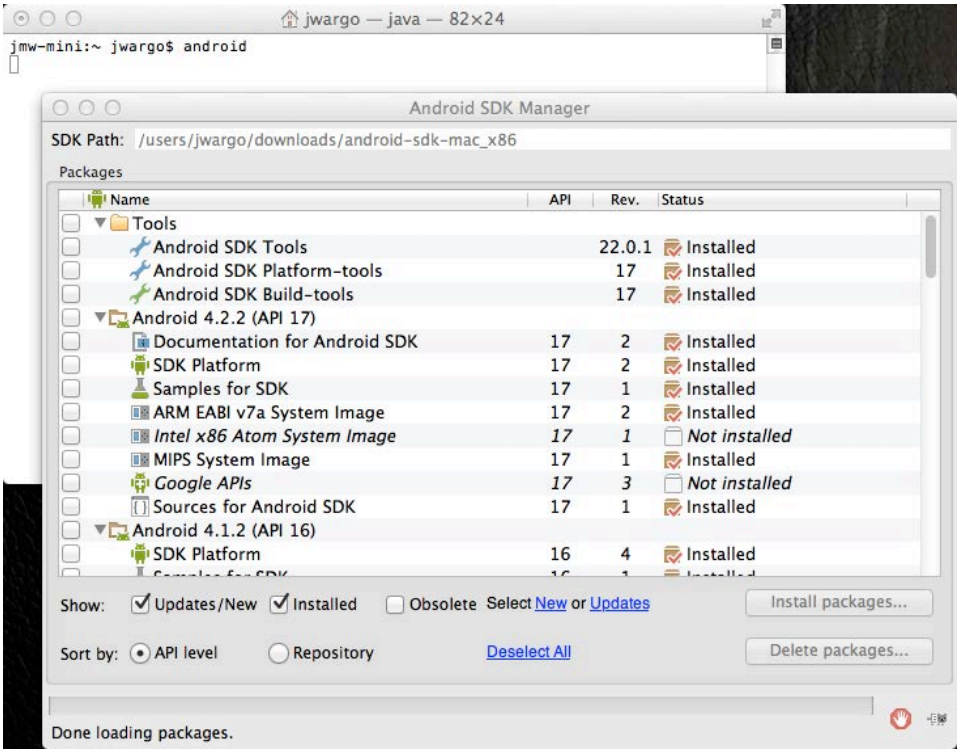



**Figure 3.6** Testing the Android Configuration

If you added the two required paths to the system path and the SDK Manager doesn't load when you type `android` at a command prompt, then the configuration is incomplete and the CLI will not be able to manipulate Android projects. You'll need to go back and resolve any issues before continuing.

If you know you will need to develop Cordova applications for Android OS versions beyond the default ones installed with the Android SDK Manager, now might be a good time to select the required additional platforms and install those as well.

You can read more about Android development for Apache Cordova in Chapter 7, "Android Development with Cordova."

## BlackBerry Development Tools

To build Cordova applications for BlackBerry, you must register for and install a set of encryption keys used to sign BlackBerry applications. A BlackBerry application will run in a BlackBerry simulator but must be signed before it will run on a physical device. To register for a set of keys, point your browser of choice to https://www.blackberry.com/SignedKeys/codesigning.html and follow the instructions. The good news is that signing keys are free and can be obtained in 1 to 2 hours. After you request the keys, you will receive several emails with instructions on how to configure your development environment for them. I describe how to register and use the signing keys in Chapter 8, "BlackBerry 10 Development with Cordova."

For Cordova 3, you will use the BlackBerry Native SDK (NSDK) to build Cordova applications. Older versions of Cordova used the BlackBerry WebWorks SDK, but they've switched over to the NSDK.

It's possible that by the time you read this that there will be a smaller, command-line tools installation you can use for Cordova development. Be sure to check the readme at https://github.com/apache/cordova-cli to make sure of the latest requirements before continuing.

> **Note**
>
> Cordova 3.0 and higher support development on BlackBerry 10 and the BlackBerry PlayBook device platforms. Beginning with Cordova 3.0, support for BlackBerry 7 devices has been dropped, so if you want to develop for older BlackBerry devices, you need to use Cordova 2.9 or earlier.

To install the NSDK, point your browser to http://developer.blackberry.com/native and follow the instructions to download and install the appropriate software depending on whether you will be developing for BlackBerry 10 or the BlackBerry PlayBook platform. The NSDK installation requires that you execute a script or batch file (depending on whether you're running on Macintosh OS X or Windows) to properly configure the system environment for command-line use. Be sure to follow the complete installation instructions on the BlackBerry Developers website. If you don't complete this step, you may see the following error during installation of the CLI:

```
WARNING: Your system does not meet requirements to create blackberry projects. See error
output below.
The BB10NDK environment variable QNX_HOST is missing. Make sure you run `source <path to
bb10ndk>/bbndk-env.sh`. Even better, add `source`ing that script to your .bash_profile or
equivalent so you don't have to do it manually every time.
SKIPPING blackberry bootstrap.
```

The BlackBerry device simulators for BlackBerry 10 and the PlayBook use a VMWare virtual machine for the simulators. For Macintosh installations, you need to acquire and install a VMWare Fusion (www.vmware.com/products/fusion/overview.html) license before you can install the simulators. For Windows, you can use the free VMWare player application, which is available at www.vmware.com/products/player. Simply install the player application then install the BlackBerry 10 device simulator software.

You can read more about BlackBerry 10 development for Apache Cordova in Chapter 8.

## iOS Development Tools

Before you can do much around iOS development, you must join the Apple iOS Developer program. You can find information about the program at https://developer.apple.com/programs/ios; there's a yearly fee and different capabilities available to you depending on which program you choose.

Next, you want to open the Macintosh App Store, search for "xcode," and install the Xcode IDE, as shown in Figure 3.7.


**Figure 3.7** Macintosh App Store Xcode Entry

Xcode is a huge install, so find something else to do for a while as it downloads. Once the installation has completed, you need to launch Xcode and install Xcode command-line tools. With Xcode open, open the Xcode menu and select Preferences. In the dialog that appears, select Downloads, then the Components tab, shown in Figure 3.8.

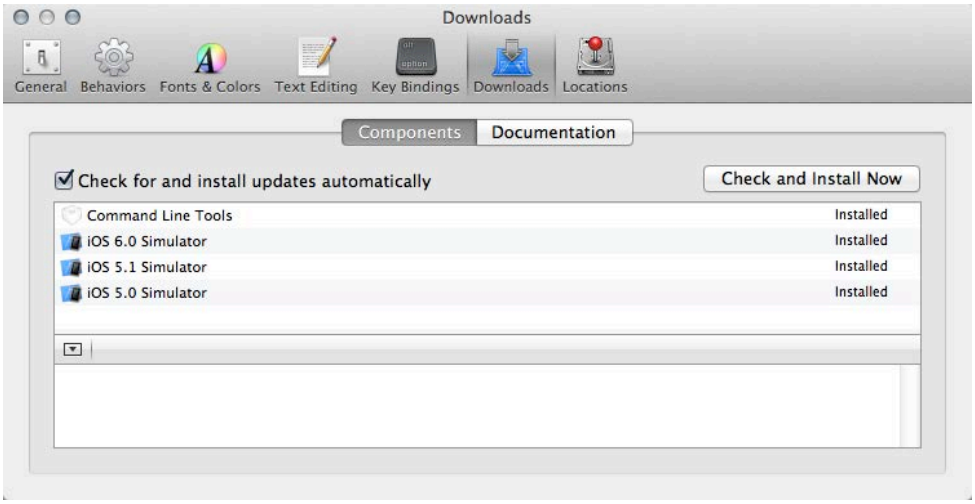



**Figure 3.8** Xcode Preferences Dialog

If you have the command-line tools installed as shown in Figure 3.8, you're all set. If not, there will be a button on the far right you can click to complete the installation. This is another large download, so be prepared for this to take a while. With that complete, you're still not done: there are a few more steps to complete for your Xcode configuration.

The Xcode command-line tools have their own license agreement that you must agree to before they can be used. You can't agree to the terms during installation; instead you must open a Macintosh Terminal window and execute the following command:

```
xcodebuild -license
```

If you receive an error, then the Xcode command line hasn't been installed correctly, and you'll have to go back and resolve the problem before continuing. If installed correctly, you should see a screen similar to the one shown in Figure 3.9.

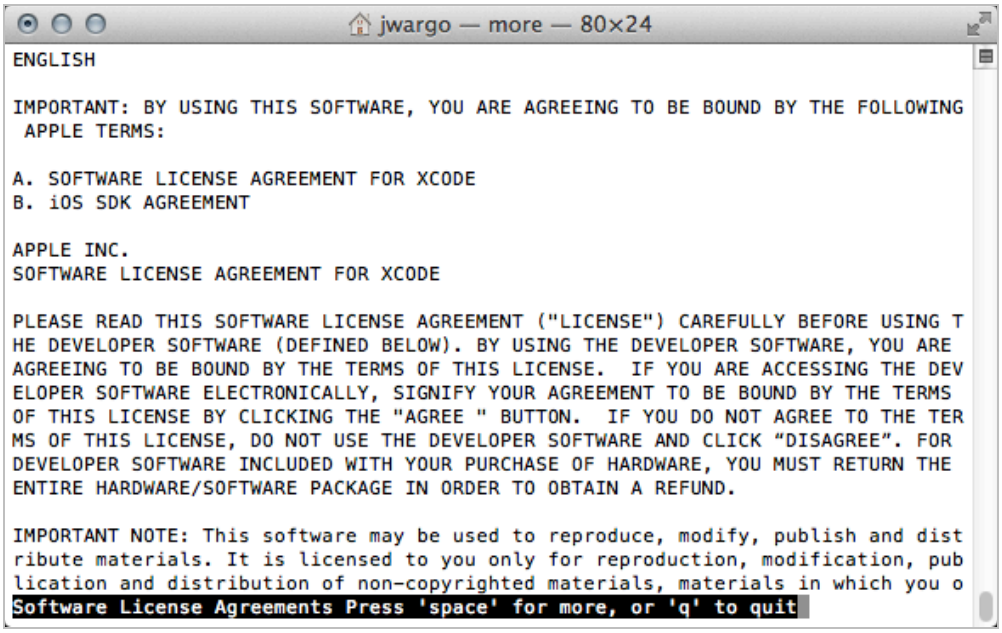



**Figure 3.9** Xcode License Agreement

Press the space bar repeatedly until you have read the entire agreement. When you reach the end of the agreement, you will see a screen similar to the one shown in Figure 3.10. If you agree with the terms of the agreement, type the word "agree" and press the Enter key. If you don't, then you will not be able to use the Xcode command-line tools, and the Cordova CLI will not work for iOS development on the system—sorry.

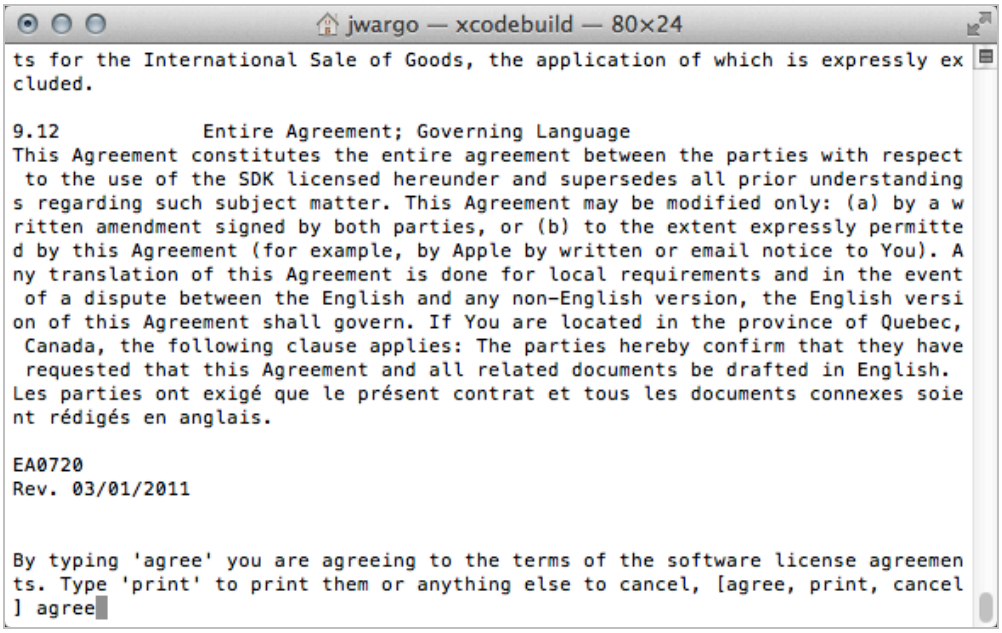



**Figure 3.10** Agreeing to the Xcode License Agreement

If you skip this process, during the installation of the Cordova CLI, you will receive the error message shown in Figure 3.11.

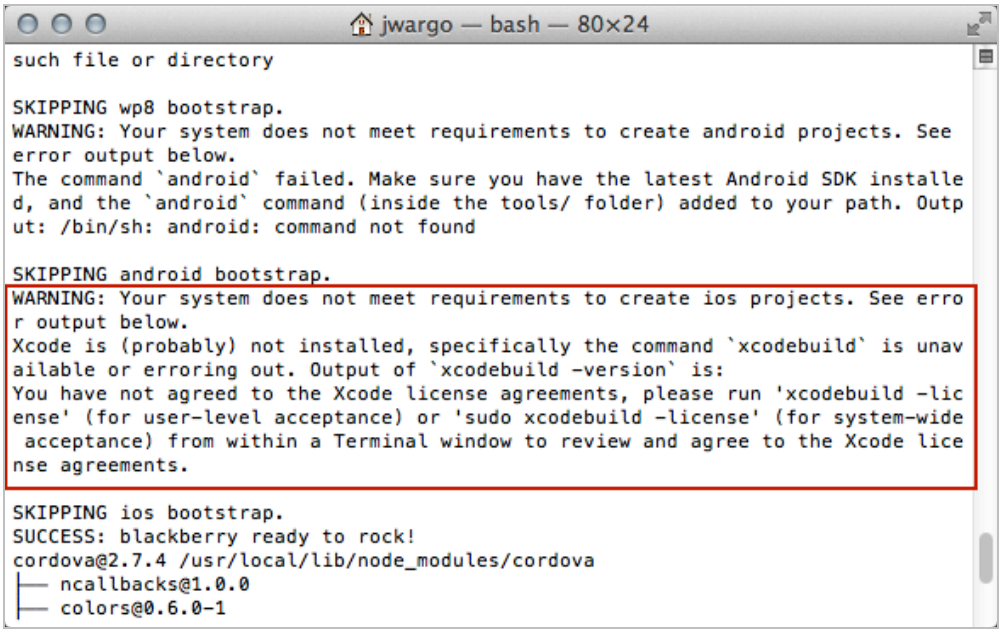



**Figure 3.11** Cordova CLI Installation Error

If you intend to use the CLI to build and test Cordova applications, you may have one last step to complete first. At this stage of the installation process, if you attempted to test a Cordova application on iOS using the CLI, you may get the following error:

```
[Error: An error occurred while emulating/deploying the ios project.Error: ios-sim was not
found. Please download, build and install version 1.5 or greater from
https://github.com/phonegap/ios-sim into your path. Or 'brew install ios-sim' using
homebrew: http://mxcl.github.com/homebrew/
```

This happens because the tools needed for the CLI to launch the iOS simulator aren't (currently) included with the Cordova download or the CLI, so you have to install them manually. The error message provides you with two options. I didn't have Homebrew installed, so I chose to install directly from Github. I downloaded the project files from https://github.com/phonegap/ios-sim, opened a Terminal window, navigated to the folder where I'd extracted the ios-sim download, then issued the following command in the terminal:

```
rake install prefix=/usr/local/
```

The command will build the tool and install it in a place where it can be easily accessed by the CLI.

You can read more about iOS development for Apache Cordova in Chapter 9, "iOS Development with Cordova."

## Windows Phone Development Tools

If you will be doing Windows Phone development for Cordova, you first need a system running Microsoft Windows (the development tools won't run on Linux or Macintosh OS X), plus you need to install the Microsoft Windows Phone SDK. You can download the SDK from the Windows Phone Dev Center located at http://developer.windowsphone.com. You can use version 8 of the SDK for both Windows Phone 8 and Windows Phone 7.5 applications, so if you're developing for those two OSs, you have all you need in that version. Windows Phone applications are built using Visual Studio, and there's a free Visual Studio Express download available in the dev center. For Windows Phone 8 applications, you need to install Visual Studio Express 2012, which is included with the SDK. After you install the SDK, you need to register with Microsoft for a free license key for Visual Studio Express.

The Windows Phone development tools have some hefty system requirements, so be sure to check out the requirements for the particular version of Visual Studio you will be using for your Cordova development and make sure your system exceeds the minimum requirements. Failure to do so will cause some stress, as the system will not be as responsive as you may expect.

> **Warning**
>
> The Windows Phone emulators place a pretty big load on the system running them. Microsoft has enhanced the tools to help reduce the system requirements, but when you launch an emulator for the first time, you'll get warnings for some system BIOS settings you need to change to enhance performance.
>
> When I wrote *PhoneGap Essentials*, I found that I couldn't even run the emulators in a VMWare virtual machine, even on the most modern and fastest system I had. For this book, I decided to upgrade one of my lab systems, install Windows 8, and dedicate it to Windows Phone development so I wouldn't have to deal with potential performance issues.

You can read more about Windows Phone 8 development for Apache Cordova in Chapter 10, "Windows Phone 8 Development with Cordova."

## CLI Installation

The CLI is written in JavaScript and therefore uses Node.js as a runtime execution engine. The CLI currently requires Node.js 0.10 and higher, so if you already have it installed, make sure you update it to the latest version. To install Node.js, point your browser of choice to www.nodejs.org, then download and install the software. Node.js will install in your system path, so you can open a terminal window and type `node` to launch the Node.js application.

During installation, the CLI retrieves some code from GitHub (www.github.com), so you will need to have Git installed on your development workstation. If you're running on Linux or Macintosh OS X, you can install Git using the Node.js package manager (npm) by opening a terminal window and typing:

```
npm install -g git
```

On Windows, you will have to download Git tools and install them on your development workstation from http://git-scm.com.

Once you've completed the installation, open a terminal window (Mac OS X) or command prompt (Windows), type `git`, and press Enter. A bunch of text should scroll by, and then you should see a screen similar to the one shown in Figure 3.12.

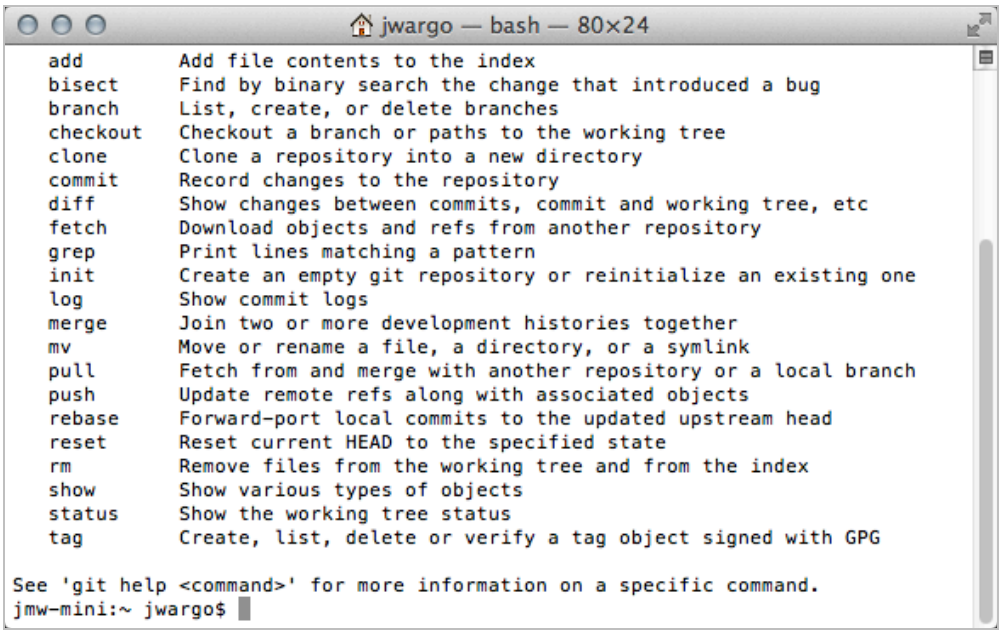



**Figure 3.12** Testing the Git installation

If you receive an error message indicating that the `git` command couldn't be found, then the Git installation didn't complete successfully, and you will need to resolve the error before continuing.

With Git and Node.js installed, you're ready to install the Cordova CLI; to install the CLI, you have two options.

If you downloaded the Cordova framework from the Cordova website (described in Chapter 2), you can navigate to the folder that contains the framework files and extract the contents of the cordova-cli.zip

file. Once the files are extracted, open a terminal window and navigate to the folder that contains the extracted files, then issue the following command:

```
npm install
```

The other option is to pull the latest version of the CLI from the Internet using npm. On Windows, open a command prompt and issue the following command:

```
npm install -g cordova
```

The `-g` in the command tells npm to install the Cordova CLI globally—if you did not use this parameter, the CLI would be available only from the current folder.

On Macintosh, you must do more to complete the installation. The previous command may work, but if you receive the EACCESS error shown in Figure 3.13, you will have to try a different approach.

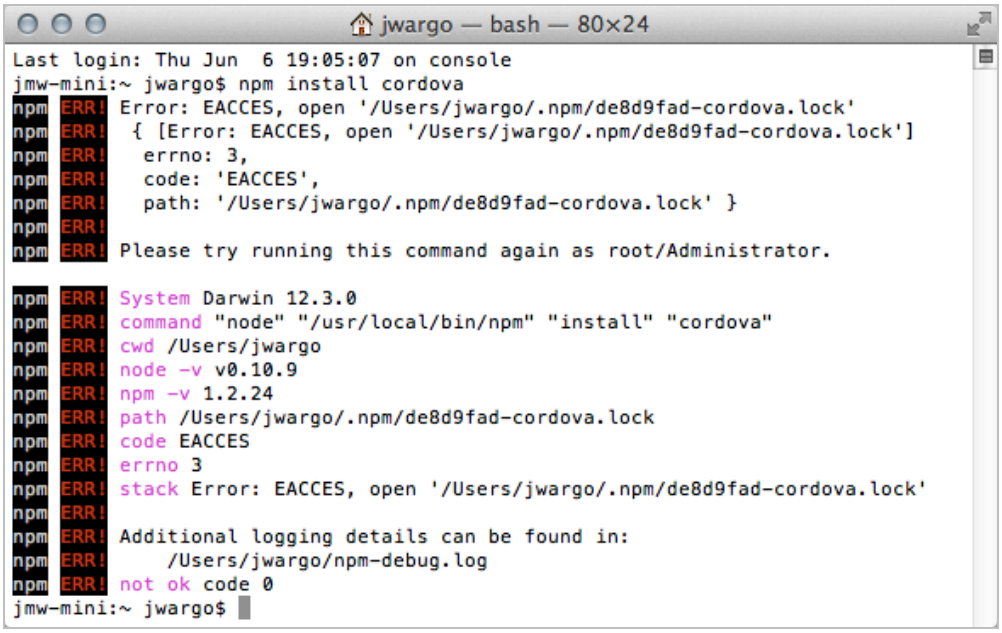



**Figure 3.13** Cordova CLI Installation Access Error

To get around this error, you need to install Cordova with `sudo` by using the following command:

```
sudo npm install -g cordova
```

`Sudo` allows you to execute programs with a higher access level, so when you issue the command, you should be prompted for your password before the installation begins.

The CLI installation installs a ton of stuff, so be prepared to watch a lot of information go scrolling by. When it is all done, you will see a message summarizing what was installed; if you scroll back up through the listing, you will see how much stuff was installed during the process.

> **Warning**
>
> In my experience with the CLI, a lot can go wrong with the installation, but you won't be able to tell because it will seem to have installed correctly. To remedy this, you should scroll back through the entire installation history shown in the terminal and look for errors.

That's it—that's all there is to installing the Cordova CLI. To test the installation, open a terminal window, type `cordova` at the prompt, and press Enter. If everything installed correctly, you should see a screen similar to the one shown in Figure 3.14.

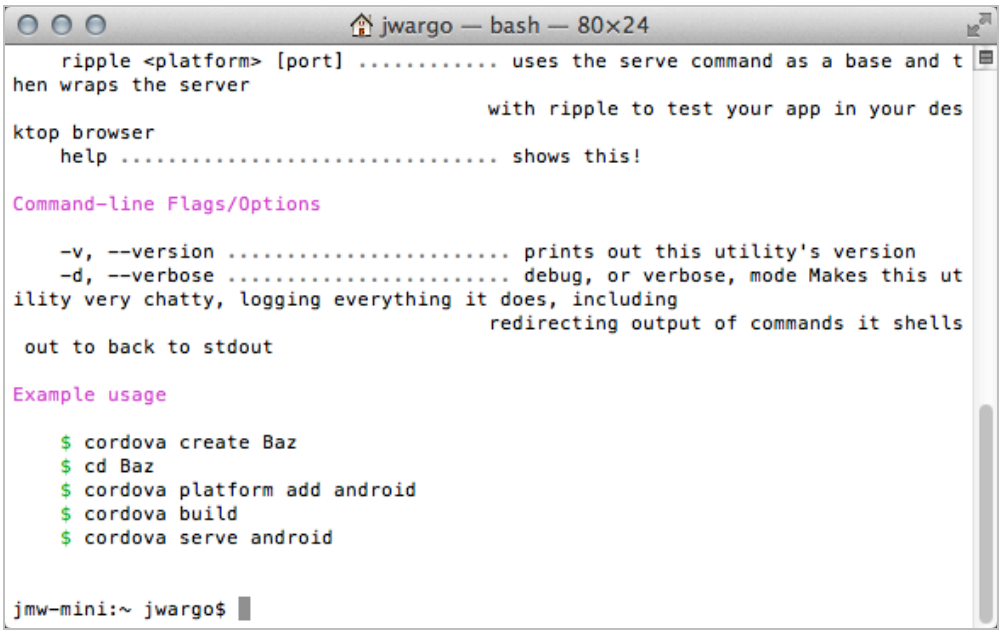



**Figure 3.14** Cordova Help Screen

If you're interested in using the PhoneGap CLI, it installs basically the same way; you can install from the Internet using the following command:

```
npm install -g phonegap
```

The PhoneGap CLI will likely be a bit behind the Cordova CLI, so don't expect the two to be at the same version number or to have all of the same features. Features are typically flushed out in the Cordova CLI, then migrated into the PhoneGap CLI later.

# Wrap-Up

In this chapter, I showed you how to install the Cordova CLI on Windows and Macintosh OS X. With these tools in place, you're ready to begin writing your own Cordova applications. In the next chapter, I show you how to use the CLI to manage your Cordova projects.

# 4

# Using the Cordova Command-Line Interface

As mentioned in Chapter 2, included with the framework files for most of the supported mobile device platforms are scripts that you can use to create new projects and test/debug applications. The problem with these scripts is that they're specific to each mobile device platform and therefore can't be used to manage a cross-platform project in its entirety.

To make it easier for developers to manage their projects, the Cordova project team built a single, unified command-line interface (CLI) that works across all of the Cordova-supported mobile device platforms. This chapter illustrates how to use the Cordova CLI to manage Apache Cordova application projects.

## About the CLI

Because Cordova applications are web applications and can be coded using any text editor, and the mobile device manufacturers use different integrated development environments (IDE) and project structures for building native applications, there wasn't a single, simple way to create and manage Cordova applications for multiple mobile device platforms. With earlier versions of Cordova, developers had to create multiple projects, one for each mobile device platform, and copy the web application code between the projects. In Cordova 3.0, the project also migrated all of the Cordova APIs to plugins, so managing your Cordova projects, web content, and installed plugins could be quite difficult.

Beginning with Cordova 3.0 (actually, the CLI was available in prerelease form in Cordova 2.7), the project added a CLI that provides a suite of commands a developer can use to

- Create cross-platform Cordova application projects
- Add support for each of the Cordova supported mobile device platform
- List supported mobile device platforms
- Remove support for a mobile device platform
- Add a plugin to a project; this can be a core Cordova plugin, a third-party plugin, or a plugin you've created yourself
- List plugins installed in a project

- Remove a plugin from a project
- Prepare, compile, and build projects
- Serve an application project's web content via a web server
- Launch an application in the appropriate mobile device simulator

With these commands in place, a developer can manage the complete lifecycle of a Cordova application. Each of these options is described throughout the remainder of the chapter.

# Troubleshooting the CLI

It's never good to start a chapter with troubleshooting information, but when working with the CLI, there will be times when you wish you could tell more about what's happening behind the scenes when a command is running. Right after Cordova 2.8 was released, the project team added a verbose mode to the CLI. To enable verbose mode, add a `-d` or `--verbose` to any CLI command.

For example, the following Cordova command returns nothing; it does its work and quits:

```
cordova build android
```

In general, as long as the command works before returning nothing to the terminal window, that's okay, but when working with a command that fails silently or fails with an unclear error message, it's useful to be able to see exactly what's happening with the command.

If you issue the following command instead:

```
cordova -d build android
```

the terminal window will fill with more than 100 lines of text (which I won't list here) that shows the status of every step performed by the CLI during processing of the command. Some of the information is provided by the CLI commands, but some is generated by any third-party tool called by the CLI.

> **Note**
>
> It's interesting to see how many different programs are called by the CLI as it does its work. The default terminal on Macintosh will change the contents of the terminal window title bar with every external program called by a script. For some longer running processes, it's fun to watch the title bar to see how much stuff is really going on.

# CLI Command Summary

Table 4.1 provides a summary of the available CLI commands; the sections that follow describe the operations in greater detail.

Table 4.1 **Cordova CLI Command Summary**

| CLI Command | Description |
|---|---|
| Help | Displays information about the available CLI commands |
| Create | Creates a Cordova project and associated project folders and files |
| Platform | Manages the mobile device platforms associated with a Cordova project |
| Plugin | Manages the installation and uninstallation of Cordova plugins |
| Prepare | Copies the web application content from the Cordova project's www folder into the project's mobile platform project folders |
| Compile | Packages web applications into native Cordova applications |
| Build | Prepares then packages web applications into native Cordova applications |
| Emulate | Runs a Cordova application in one or more mobile device platform's device simulators |
| Run | Runs a Cordova application in one or more mobile device platform's devices |
| Serve | Serves the web application content so it can be accessed via a web browser |

# Using the CLI

In this section, I show you how to use the commands exposed through the Cordova CLI. I cover the mechanics of the commands and what happens when each is executed. Look to Chapter 6, "The Mechanics of Cordova Development," for more information about the Cordova application project files.

To learn more about what commands are available in the CLI, you can open a terminal window (or Windows command prompt—I'm just going to call it a terminal window going forward) and get help by typing:

```
cordova help
```

You can also check the CLI version number by using the following command:

```
cordova -v
```

or

```
cordova -version
```

## Creating a Cordova Project

Before you can do any Cordova development, you must first create an application project. To help you, the Cordova CLI includes a `create` command, which is used to create new projects.

To create a new Cordova project, open a terminal window and navigate to the folder where you want the project folder created, then issue the following command:

```
cordova create project_folder
```

This creates a simple Cordova project structure and web content files you need for the application. You can also specify an application ID and application name using the following command:

```
cordova create project_folder app_id app_name
```

What this does is create the same project folder but also writes the application ID to the application's configuration file and allows you to have an application name, which is different than the application folder name. For example, to create a project in a folder called `myapp` but with an application name of `Hello`, you would issue the following command:

```
cordova create myapp com.cordovaprogramming.hello Hello
```

Let me show you an example. I'm going to create a sample application project called Test. To do so, I navigate to the location where I want to create the project and issue the command shown in Figure 4.1.

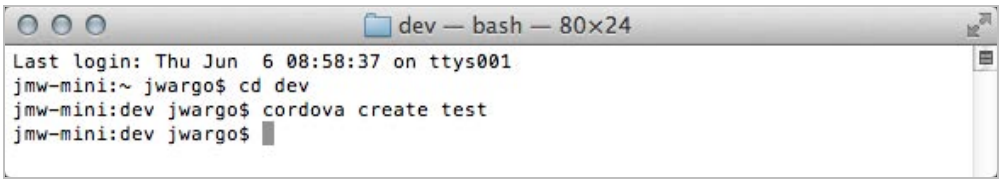



**Figure 4.1** Cordova CLI – Creating a New Project

Notice that the command doesn't tell you anything about what happened; it simply runs and returns to the command prompt.

> **Note**
>
> All of the Cordova CLI commands operate against the current folder. The `create` command creates a folder structure for your Cordova projects, and the remaining commands must be issued from within the project folder created by `create`.

The CLI will run and will create the folder structure shown in Figure 4.2. The folder structure organizes the Cordova project in a way that simplifies the process of using this application across multiple mobile device platforms. Notice that the www folder is separate; this allows you to create a single web application that's shared across multiple mobile device platforms. I talk more about the web project structure and files in the next chapter.

The merges folder contains web application assets (HTML, CSS, and JavaScript files plus images, videos, and more) that differ across mobile device platforms. If you had, for example, a CSS file with two versions, one for Android and another for iOS, you could use the folder structure in the merges folder to store each file in a mobile device platform–specific location and have the appropriate file copied into the project folder at build time. You would place your Android-specific files (and subfolders) in merges/android and your iOS-specific files in merges/ios.

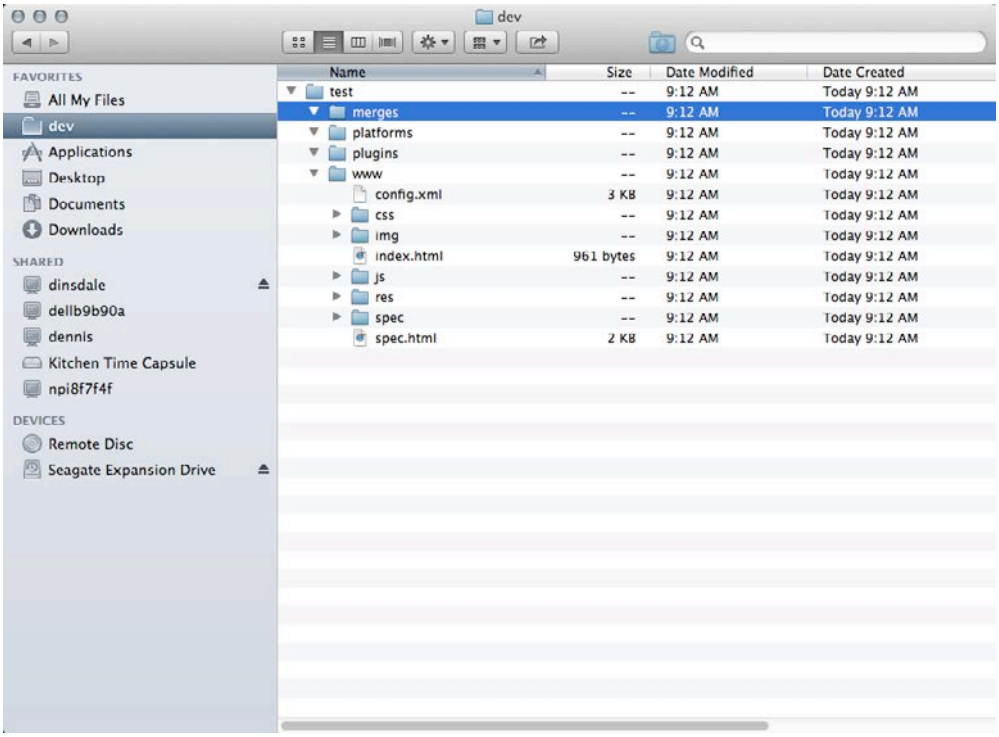



**Figure 4.2** New Cordova Project Folder

The platforms folder is used to store a separate folder for each mobile device platform being supported by this application. I talk more about this in the next section. The plugins folder is used to store the source code files for each plugin used by the application. I talk more about this later as well.

In the sections that follow, I describe the remaining CLI commands. The `create` command creates a new project in a subfolder of the folder from where the command was issued. The remaining commands must be issued from a terminal window pointing within the project folder. In this section, I created a project called test, and the CLI created a folder called test for the project. In order for me to issue CLI commands which manipulate the test project, I must use the `cd` command to change directory into the test folder before using any of the remaining commands.

### Cordova CLI Lazy Loading

Instead of preloading everything the Cordova CLI needs, the CLI uses a process called *lazy loading* to download the requisite project files to create Cordova projects and add platform support to a project. The files are downloaded only when you need them.

When you create a project, the CLI downloads the default HelloWorld project, extracts it, and copies it to the appropriate project folder. When you add a platform to a project, the CLI downloads a preconfigured project, extracts it, and copies it to the platform folder.

As interesting as this approach is, I encountered a lot of problems with this feature as I worked with the prerelease and even production versions of the CLI. In a lot of my testing, the CLI would attempt to download the files it needed, and the download or the extraction would fail—but the mechanism the CLI uses to determine if the download was successful would tell the CLI that the files it needed were actually there. The CLI checks for the existence of the downloaded files

folder and, if the folder exists, simply assumes everything it needs is in there.

I pushed on the CLI development team to modify the lazy load process to leverage the appropriate proxy settings that might be set on the system running the CLI. This fixed some of the problems with downloading the files on a corporate network. The check for the existence of the download files is still broken, but the development team has promised a fix for the next version.

The CLI uses a folder called .cordova for storing its lazy-loaded files. The folder is located immediately under your user's home folder (c:\users\user_name\ on Windows,

/users/user_name/.cordova on Macintosh). You can see an example of the folder structure in Figure 4.3. On Macintosh, the folder is hidden, so you can't view it in Finder without first reconfiguring the system to show hidden files and folders in Finder.

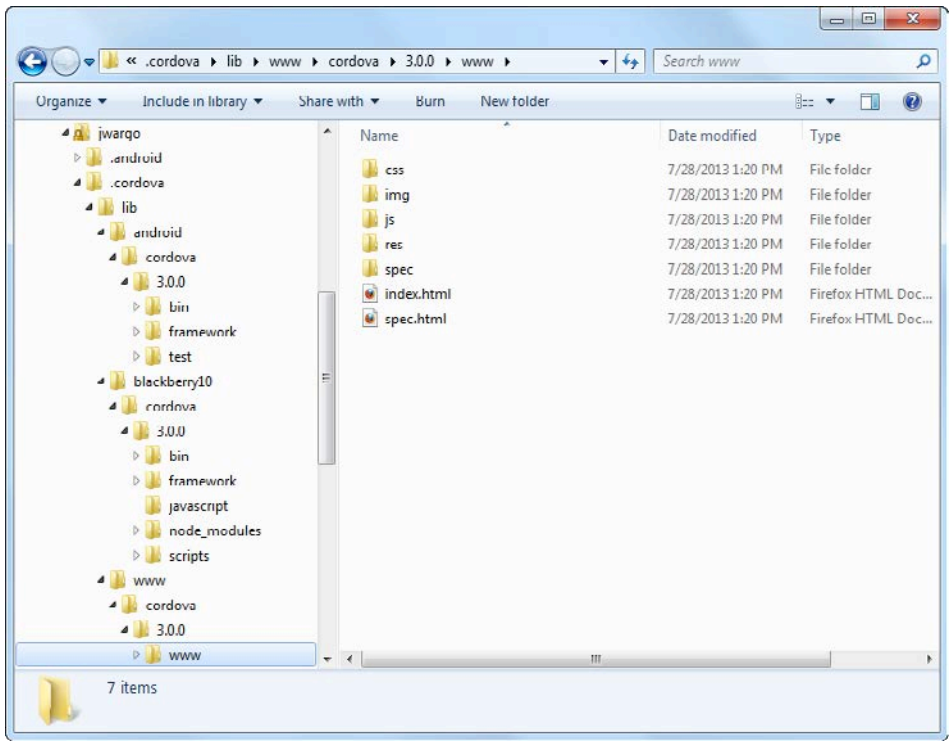



**Figure 4.3** CLI .cordova Folder Structure

If you get an error when creating a project or adding a platform, repeat the process using the `-d` debug flag and look for an indication of what is happening. If you see a message that says the CLI has the files it needs and is skipping the download, take a look in the folder name mentioned in the error message. If there are no files located in the specified location, you should whack the empty folder and try again.

## Platform Management

The project structure we've created so far has only a few empty folders plus a web application project. It doesn't know anything about the different mobile device platform it needs to support. The Cordova CLI provides a `platform` command that allows you to manage the project files for each of the mobile device platforms your application supports.

### Adding Platforms

To add support for a particular mobile device platform to your Cordova application, you must open a terminal window and navigate into the Cordova project folder. Then, to add a project file for a particular mobile device platform, you must issue the following command:

```
cordova platform add platform_name
```

> **Warning**
>
> The development tools used to create projects for the particular platform must be installed on the system and visible to the CLI before adding a platform. The CLI uses the platform's native tools to create a new project, so if the tools are not available to the CLI, the command will not work.

For example, if you wanted to add an Android project to your Cordova application, you would issue the following command from within the Cordova project folder:

```
cordova platform add android
```

You can also specify multiple target mobile device platforms in a single command, as shown in the following example:

```
cordova platform add android blackberry ios
```

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked. Let me show you an example. Using the test project we created in the previous section, I must first change to the test directory, then issue the `platform` command, as shown in Figure 4.4.

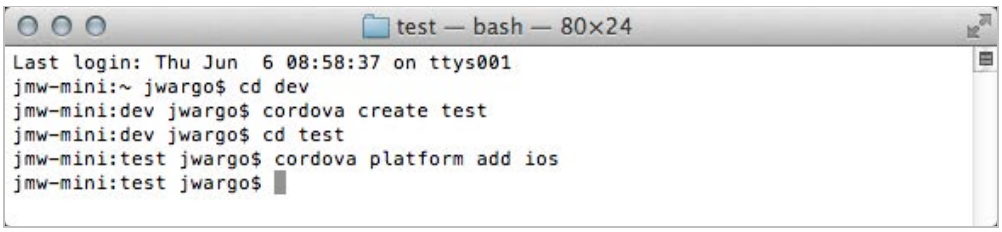



**Figure 4.4** Cordova CLI: Adding iOS Support to a Cordova Project

If you take a look at Figure 4.5, you'll see that my project has some new folders and files. In this case, the platforms folder now contains an iOS folder and a complete Xcode project called HelloCordova.xcodeproj. At this point, you could fire up Xcode, open the project, and run the application in the iOS simulator.

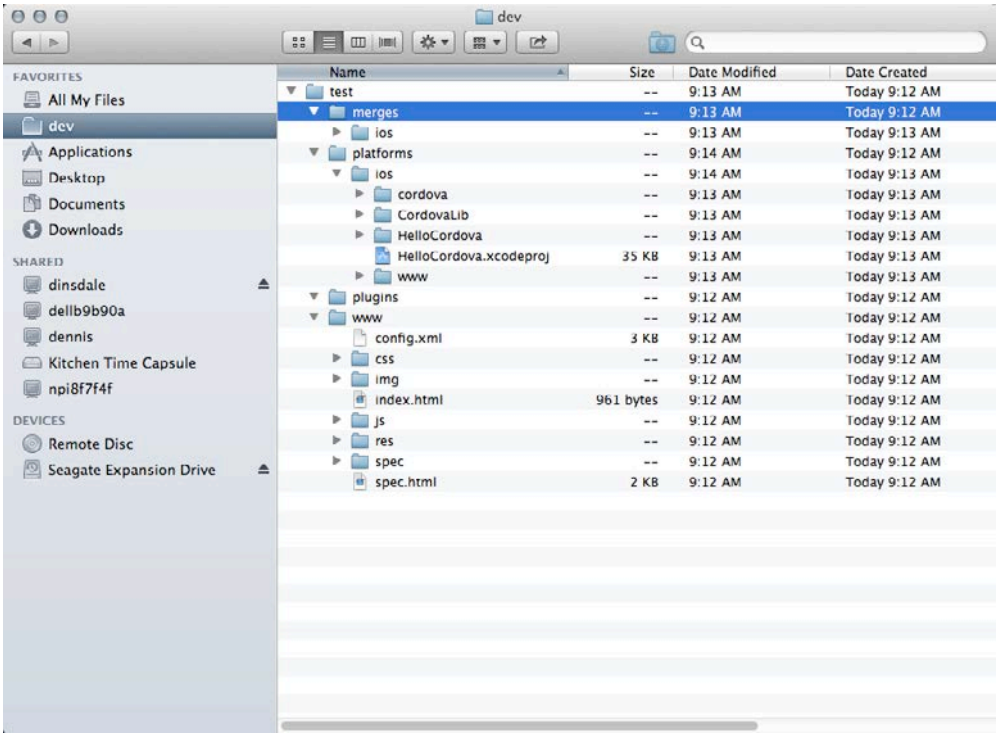



**Figure 4.5** Cordova Project Folder with iOS Platform Added

The Xcode application project is called HelloCordova because I didn't specify an app name when I created the project. If I'd created the project using the second form of `create` (described previously), like this:

```
cordova create test com.cordovaprogramming.hello Hello
```

the Xcode project would have been named Hello.xcodeproj.

To add support for Android, I executed the command shown in Figure 4.6.

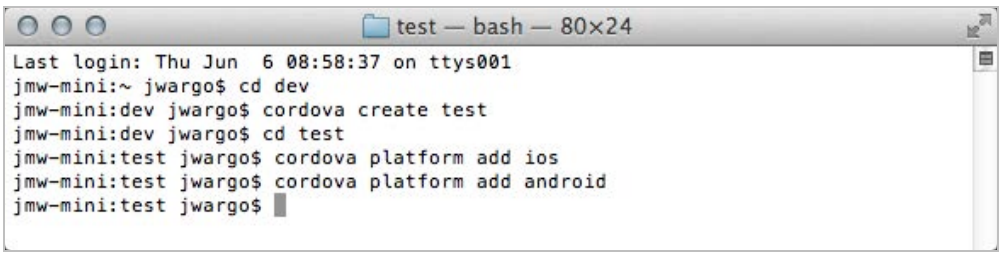



**Figure 4.6** Cordova CLI: Adding Android Support to a Cordova Project

If you now take a look at the test folder we've been working with, you will see that the project's platforms folder now contains an android folder with a complete Android project inside, as shown in Figure 4.7.

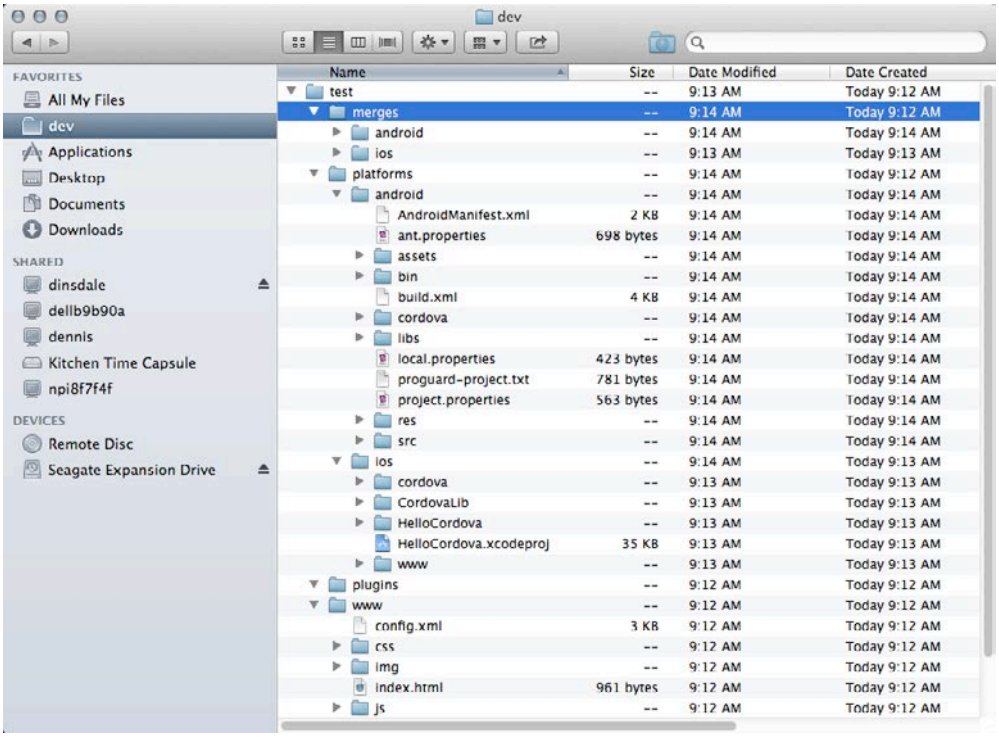



**Figure 4.7** Cordova Project Folder with Android Platform Added

You can open the Android project in a properly configured version of Eclipse (an Eclipse instance with Android Development Tools [ADT] installed) and run and debug the application in the Android emulator. I show you how to do this in Chapter 7, "Android Development with Cordova."

### Listing Platforms

The Cordova CLI provides a mechanism for listing the platforms that are defined within a Cordova project. Simply issue any of the following commands in a terminal window inside of the root folder of a Cordova project:

```
cordova platforms
cordova platform ls
cordova platform list
```

The CLI will return a JavaScript Object Notation (JSON) array containing the names of each of the platforms defined within the project:

```
[ 'android', 'ios', 'blackberry10' ]
```

You can see an example of this output in Figure 4.8.


**Figure 4.8** Cordova CLI: Listing Project Platforms

This feature isn't that useful for developers to use directly, since all they have to do is open the project folder and see what subfolders exist within the project's platforms folder. However, for automated processes, this function is more useful. Since the result is returned as a JSON array, it would be easy for a Node.js application to parse the resulting array, determine if an expected platform were missing and add it.

### Removing Platforms

If you decide that you no longer need to support a particular mobile platform for your application, you can remove it by issuing the following command:

```
cordova platform remove platform_name
```

You can also use the shortcut `rm` instead of `remove` to remove platforms:

```
cordova platform rm platform_name
```

So, for my test project, if I want to remove the project files for the iOS project, I would issue the following command:

```
cordova platform remove ios
```

You can see the results of this operation in Figure 4.9.


**Figure 4.9** Cordova CLI: Removing Platforms

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

## Plugin Management

The ability to manage a Cordova project's plugin configuration is one of the biggest features of the CLI. Instead of manually copying around plugin files and manually editing configuration files, the CLI does it all for you.

### Adding Plugins

To use the CLI to add a plugin to an existing project, open a terminal window, navigate to the Cordova project folder, and issue the following command:

```
cordova plugin add path_to_plugin_files
```

As an example, to add the Camera plugin to a Cordova application, you could use the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-camera.git
```

This connects to the Apache Git repository and pulls down the latest version of the plugin. You can find a list of the core Cordova plugin locations at http://cordova.apache.org/docs/en/3.0.0/guide_cli_index.md.html#The%20Command-line%20Interface.

In general, the plugin git location shown in the previous example uses the following format:

```
https://git-wip-us.apache.org/repos/asf/cordova-plugin-<plugin-name>.git
```

where `<plugin-name>` refers to the short name for the plugin.

The CLI can pull plugin code from most any location; if the system can access the location where the plugin files are located, you can install the plugin using the CLI. If you have plugins installed locally, say, for example, if you were working with a third-party plugin or one you created yourself and the files were stored on the local PC, you could use the following command:

```
cordova plugin add c:\dev\plugins\my_plugin_name
```

> **Warning**
>
> Installing plugins using the CLI is the only supported way to add plugins to a Cordova project. With previous versions of the framework, plugins were added by the developer manually copying files around and modifying configuration files. If you try to use that approach with Cordova 3.0 and beyond, you run the risk of corrupting your Cordova project.

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

Beginning with Cordova 3.1, plugins can be installed directly from the plugin repository just by specifying the plugin name. So, to add the console plugin to your application, you would issue the following command from a terminal window pointing at your Cordova project folder:

```
cordova plugin add org.apache.cordova.console
```

The CLI checks the repository for a plugin matching the name specified, pulls down the plugin code, and installs it into your project.

Figure 4.10 shows a Cordova project folder structure after a plugin has been added using the CLI. Notice that each plugin has a unique namespace built using the Reverse-DNS name convention plus the plugin name. All of the Apache Cordova core plugins are referred to by the plugin name added to the

end of `org.apache.cordova.core`. So, for the Camera plugin we just added to the project, you can refer to it via the CLI as `org.apache.cordova.core.camera`. This approach to plugin naming helps reduce potential for name conflicts across similar plugins created by different organizations.

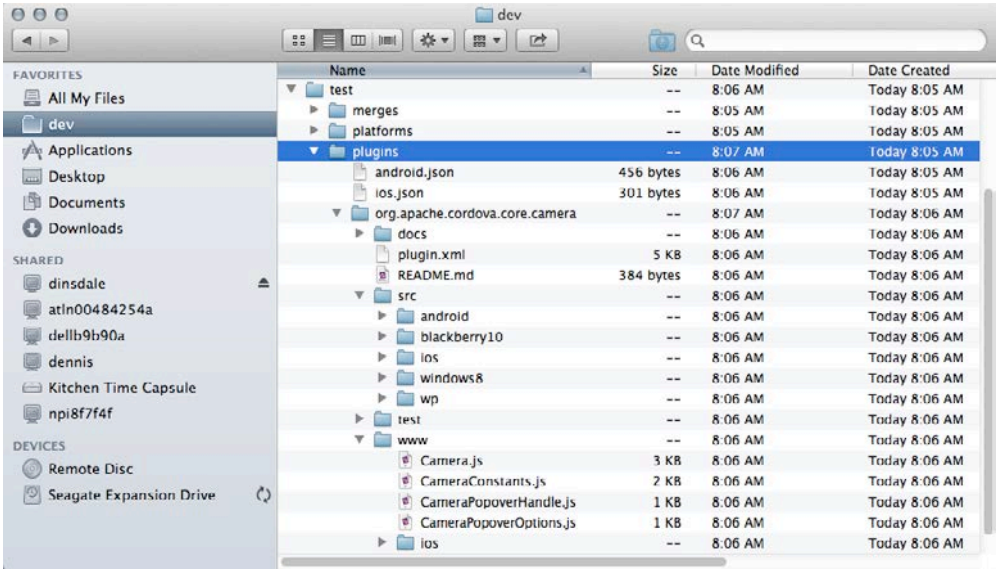



**Figure 4.10** Cordova Project Plugins Folder Content

Plugins have separate folders for the source code for the plugin for each supported mobile device platform. Additionally, some JavaScript files are consumed by Cordova applications in the www folder. There is a lot more to know about plugins; I dig deeper into the plugin project folder structure when I cover plugin development in Chapter 13, "Creating Cordova Plugins."

### Listing Plugins

To view a list of the plugins installed in a Cordova project, open a terminal window, navigate to a Cordova project folder, then issue the following command:

```
cordova plugins
```

The CLI will return the list as a JSON array, as shown here:

```
[ 'org.apache.cordova.core.camera',
  'org.apache.cordova.core.device-motion',
  'org.apache.cordova.core.file' ]
```

In this case, the Cordova project has the Cordova core Camera, Accelerator (device-motion), and File plugins installed.

### Removing Plugins

To remove a plugin from a Cordova project, open a terminal window, navigate to a Cordova project folder. then issue the following command:

```
cordova plugin remove plugin_name
```

You can also use the shortcut `rm` instead of `remove` to remove plugins:

```
cordova plugin rm plugin_name
```

So, for a project that has the Cordova core File plugin installed, you would remove it from a project by issuing the following command:

```
cordova plugin remove org.apache.cordova.core.file
```

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

The CLI will essentially reverse the plugin installation process by removing configuration file entries that point to the plugin plus removing the plugin's folder from the project's plugins folder.

## Build Management

The CLI has built-in integration with mobile device platform SDKs, so you can use the CLI to manage the application build process.

### Prepare

The CLI `prepare` command copies a Cordova project's web application content from the www and merges folders into the appropriate platforms folders for the project. This process is described in detail in Chapter 6. You will use this command whenever you make changes to a Cordova web application's content (in the www or merges folder). The `prepare` command is called automatically before many of the operations described throughout the remainder of this chapter.

To use the `prepare` command, open a terminal window, navigate to a Cordova project folder, then issue the following command:

```
cordova prepare
```

This copies the web application content into the appropriate folders for each of the mobile device platforms that have been added to the project.

To prepare a specific platform's files, use the following command:

```
cordova prepare platform_name
```

So, to prepare the Android platform folder, use the following command:

```
cordova prepare android
```

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

### Compile

In Figure 4.7, you can see that there's a cordova folder in each platform's folder structure. Within that folder are platform-specific build scripts used to compile a native application for that platform. The `compile` command initiates a compilation process by calling the build script for one or more mobile platforms.

To use the `compile` command, open a terminal window, navigate to a Cordova project folder, then issue the following command:

```
cordova compile
```

To compile a specific platform's native application, use the following command:

```
cordova compile platform_name
```

So, to compile the Android version of an application, use the following command:

```
cordova compile android
```

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

### Build

The `build` command is similar to `compile` except that it first calls `prepare` before calling compile.

To use the `build` command, open a terminal window, navigate to a Cordova project folder, then issue the following command:

```
cordova build
```

To build a specific platform's native application, use the following command:

```
cordova build platform_name
```

So, to build the Android version of an application, use the following command:

```
cordova build android
```

The command doesn't return anything to the terminal window on success, so unless you see an error reported, the command worked.

## Running Cordova Applications

The CLI has built-in integration with mobile device platform simulators, so you can launch Cordova applications directly onto simulators or physical devices. Chapters 7 through 10 provide more detailed information about the simulators, how to configure and launch them, as well as what is required to work with physical devices from the CLI. The following sections provide a high-level overview of the relevant commands.

### Emulate

The CLI `emulate` command automates the process of building an application and deploying it onto a mobile device simulator. The command first prepares the application, executes the build process, then deploys the resulting native application package to the simulator.

To run a Cordova application on the default simulator for each of the platforms configured in the project, issue the following command:

```
cordova emulate
```

To emulate the application on a single device platform emulator, Android for example, you would issue the following command:

```
cordova emulate android
```

The CLI is supposed to launch the simulator automatically for you; this works well for iOS but doesn't work for BlackBerry, and I had mixed results trying this on Android.

For the BlackBerry platform, additional steps must be followed to define simulator targets for the `emulate` command; refer to Chapter 8, "BlackBerry 10 Development with Cordova," for additional information.

### Run

The CLI `run` command automates the process of building an application and deploying it onto a physical device. The command first prepares the application, executes the build process, then deploys the resulting native application package to a connected device.

To run a Cordova application on a physical device for each of the platforms configured in the project, issue the following command:

```
cordova run
```

To run the application on a single device, Android for example, you would issue the following command:

```
cordova run android
```

For the BlackBerry and Windows Phone 8 platforms, additional steps must be followed before you can run an application on a device. Refer to Chapter 8 for additional information on configuring a BlackBerry device and Chapter 10, "Windows Phone 8 Development with Cordova," for Windows Phone 8.

### Serve

When working with a mobile web application, many developers find that it's better to test the application in a desktop browser before switching to the mobile device. This is especially important when it comes to Cordova applications because an extra packaging process has to happen before the application can be run on a device or simulator.

The CLI includes a `serve` command, which a developer can use to launch a local web server that hosts a particular platform's web content. It doesn't expose any of the Cordova APIs to the browser, so all you can really test using this option is your web application's UI. You must issue the CLI `prepare` command before launching the server to make sure your web content is up to date.

To use the `serve` command, open a terminal window, navigate to a Cordova project folder, then issue the following command:

```
cordova serve platform_name
```

To serve up a Cordova project's Android web application content, you would issue the following command:

```
cordova serve android
```

Figure 4.11 shows the `serve` command in action. Once the command loads, it will show you what URL you must use to access the web server. Simply open your browser of choice and navigate to the specified URL to access the application content.

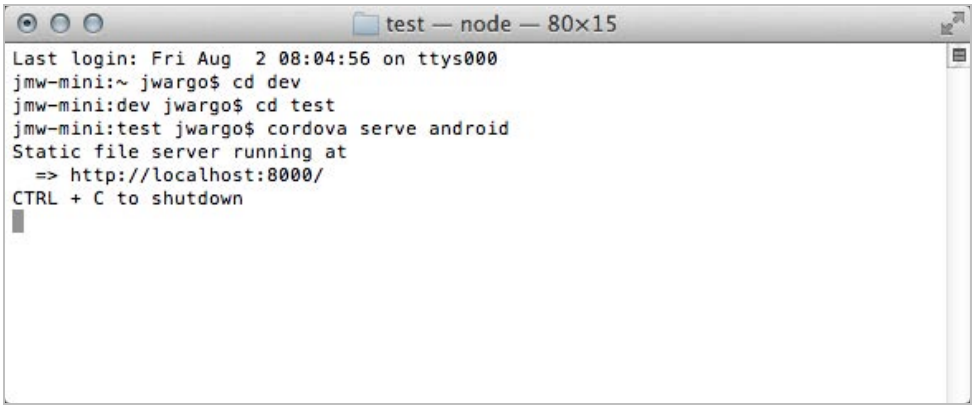



**Figure 4.11** Launching a Web Server using the CLI

When running this process on Microsoft Windows, you may receive a security warning similar to the one shown in Figure 4.12. You need to click the Allow access button to allow the web server to start.

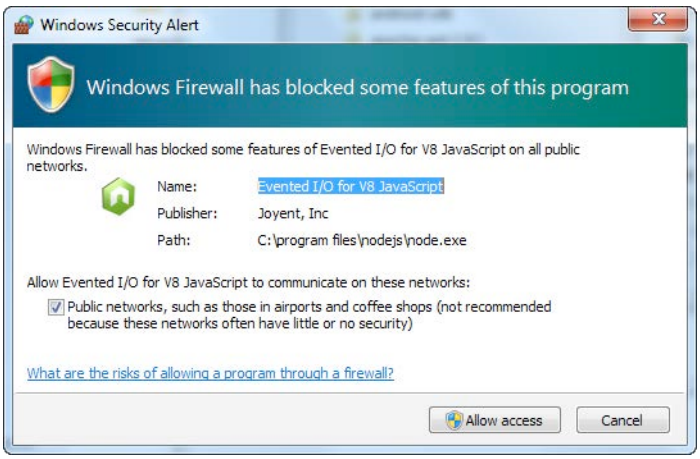



**Figure 4.12** Windows Security Alert

By default, the server will respond to port 8000, as shown in Figure 4.11. If you want to use a different port, you can pass the port number to the CLI, as shown in the following example:

```
cordova serve platform_name port_number
```

So, for the Android example shown previously, to serve the Android platform's content on port 8080, you would use the following command:

```
cordova serve android 8080
```

## Wrap-Up

In this chapter, I showed you how to utilize the Cordova CLI to manage your Cordova application projects. The capabilities provided by the CLI dramatically simplify the cross-platform development process and managing your Cordova application's plugin configuration.

In the next chapter, I discuss the structure of a Cordova application. In Chapter 5, I describe the process of building and debugging your Cordova applications.

# 5

# Anatomy of a Cordova Application

Now that we have Cordova and/or the command-line interface installed, it's time to talk about Cordova applications and Cordova application development. In this chapter, I show you what makes a web application a Cordova application and give you a tour of the sample application the Cordova team provides.

As mentioned at the beginning of the book, a Cordova application can do anything that can be coded in standard, everyday HTML, CSS, and JavaScript. There are web applications and Cordova applications, and the distinction between them can be minor or considerable.

The sections in this chapter highlight different versions of the requisite HelloWorld application found in almost any developer book, article, or training class. For the purpose of highlighting aspects of the applications' web content, rather than how they were created, the steps required to create the applications are omitted. Refer to the chapters that follow for specific information on how to create and test and debug Cordova application projects for several of the supported mobile platforms.

## Hello World!

As with any self-respecting developer book, we're going to start with the default HelloWorld application, then build on it to highlight different aspects of a Cordova application. The following HTML content (Listing 5.1) describes a simple web page that displays some text on a page.

**Listing 5.1 HelloWorld1 Application**

```
<!DOCTYPE HTML>
<html>
<head>
  <title>HelloWorld1</title>
</head>
<body>
  <h1>Hello World</h1>
  <p>This is a sample Cordova application</p>
</body>
</html>
```

If you package that page into a Cordova application and run it on a smartphone device or device emulator (in this case, an Android emulator), the app will display a simple splash screen, and then you will see something similar to what is shown in Figure 5.1.


**Figure 5.1** HelloWorld1 Application Running on an Android Emulator

This is technically a Cordova application because it's a web application running within the Cordova native application container. If I hadn't cropped the image, you would see that the web application consumes the entire screen of the emulated Android device. Even though I'm running a web application, because it's running within a native application, there's no browser UI being displayed and no access to browser features. It's simply a native application rendering web content.

There is, however, nothing Cordova-ish about this application. It's running in the Cordova native container, but it isn't leveraging any of the APIs provided with the Cordova framework. Therefore, any web application can be packaged into a Cordova application; there's nothing forcing you to use the Cordova APIs. If you have a simple web application that just needs a way to be deployed through a smartphone's native app store, then using Cordova is one way to accomplish that goal.

## Cordova Initialization

Now let's take the previous example application and add some Cordova-specific stuff to it. The HellowWorld2 application, shown in Listing 5.2, has been updated to include code that recognizes when the Cordova application has completed initialization and displays an alert dialog letting you know Cordova is ready.

**Listing 5.2 HelloWorld2 application**

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>
    <script type="text/javascript" charset="utf-8">
      function onBodyLoad() {
        document.addEventListener("deviceready",onDeviceReady,
          false);
      }
      function onDeviceReady() {
        navigator.notification.alert("Cordova is ready!");
      }
    </script>
  </head>
  <body onload="onBodyLoad()">
    <h1>HelloWorld2</h1>
    <p>This is a sample Cordova application.</p>
  </body>
</html>
```

### Warning

If you copy the code from any of the listings in this chapter and try it in your own Cordova applications, you may notice that there are some extra carriage returns in the middle of some of the HTML. This was done to make the code render cleanly in the printed edition of the book. To download clean versions of all of the projects in this book, access the Code section of the book's website at www.cordovaprogramming.com or download it from Github at https://github.com/johnwargo/cordova-programming-code.

On the iPhone simulator, the application will display the screen shown in Figure 5.2.


**Figure 5.2** HelloWorld2 Application Running on an iOS Simulator

Within the `<Head>` section of the web page are two new entries: meta tags that describe the content type for the application and viewport settings. The content type setting is a standard HTML setting and should look the same as it would for any other HTML5 application.

The following viewport settings tell the embedded web browser rendering the content how much of the available screen real estate should be used for the application and how to scale the content on the screen:

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

In this case, the HTML page is configured to use the maximum height and width of the screen (through the `width=device-width` and `height=device-height` attributes) and to scale the content at 100% and not allow the user to change that setting in any way (through the `initial-scale=1`, `maximum-scale=1`, and `user-scalable=no` attributes).

> **Note**
>
> The viewport and associated attributes are not required. If they're omitted, the browser will revert to its default behavior, which may (or may not, who knows?) result in the application's content either consuming less of the screen area available to it or else zooming beyond it. Because there's not much content in the HelloWorld2 application, it could, for example, consume only the upper half of the screen on some devices. You may also find that on some platforms, the settings have no effect—all the more reason to test your Cordova applications on a variety of mobile devices before release.

There's also a new script tag in the code that loads the Cordova JavaScript library:

```
<script type="text/javascript" charset="utf-8" src="cordova.js"></script>
```

This script tag loads the core Cordova API library and makes any core Cordova capabilities available to the program. Remember, though, that with Cordova 3, the core APIs have all been migrated to plugins. The cordova.js file contains some basic housekeeping functions.

The JavaScript code in a Cordova application does not have immediate access to the Cordova APIs after the web application has loaded. The native Cordova application container must complete its initialization process before it can respond to calls JavaScript made using the Cordova APIs. To accommodate this delay in API availability, a web developer building Cordova applications must instruct the container to notify the web application when the Cordova APIs have completed initialization. Any application processing that requires the use of the APIs should be executed by the application only after it has received its notification that the APIs are available.

In the HelloWorld2 application, this notification is accomplished through the addition of an `onload` event defined in the page's body section:

```
<body onload="onBodyLoad()">
```

Within the `onBodyLoad` function, the code registers an event listener that instructs the application to call the `onDeviceReady` function when the device is ready, when the Cordova application container has finished its initialization routines and fired its `deviceready` event:

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

In this example application, the `onDeviceReady` function simply displays a Cordova alert dialog (which is different than a JavaScript alert dialog; I show you the difference in Chapter 12.) letting the user know everything's okay:

```
navigator.notification.alert("Cordova is ready!")
```

In production applications, this function could update the UI with content created through API calls or do whatever other processing is required by the application. You'll see an example in Listing 5.3.

Remember, most of the Cordova APIs have been removed from the container and implemented as plugins. So, to utilize the Cordova alert method, you must add the dialogs plugin to your application by opening a terminal window to your Cordova project folder and issuing the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-dialogs.git
```

### The Cordova Navigator

Many of the APIs implemented by Cordova are instantiated from the Navigator object. Unfortunately, it's not consistent: some do and some do not. Be sure to check the API documentation before calling an API.

# Leveraging Cordova APIs

Now that we know how to configure an application to wait until the Cordova APIs are available, let's build an application that actually uses the Cordova APIs, as illustrated in the HelloWorld3 application shown in Listing 5.3.

**Listing 5.3 HelloWorld3 application**

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>
    <script type="text/javascript" charset="utf-8">
      function onBodyLoad() {
        document.addEventListener("deviceready", onDeviceReady,
        false);
      }
      function onDeviceReady() {
        br = "<br />";
        //Get the appInfo DOM element
        var element = document.getElementById("appInfo");
        //Replace it with specific information about the device
        //running the application
        element.innerHTML = 'Cordova Version: ' +
          device.cordova + br +
          'Platform: ' + device.platform + br +
         'Model: ' + device.model + br +
         'OS Version: ' + device.version;
       }
    </script>

  </head>
  <body onload="onBodyLoad()">
    <h1>HelloWorld3</h1>
    <p>This is a Cordova application that makes calls to the
    Cordova APIs.</p>
```

```
    <p id="appInfo">Waiting for Cordova Initialization to
    complete</p>
  </body>
</html>
```

Figure 5.3 shows the HelloWorld3 application running on the BlackBerry Q10 simulator.

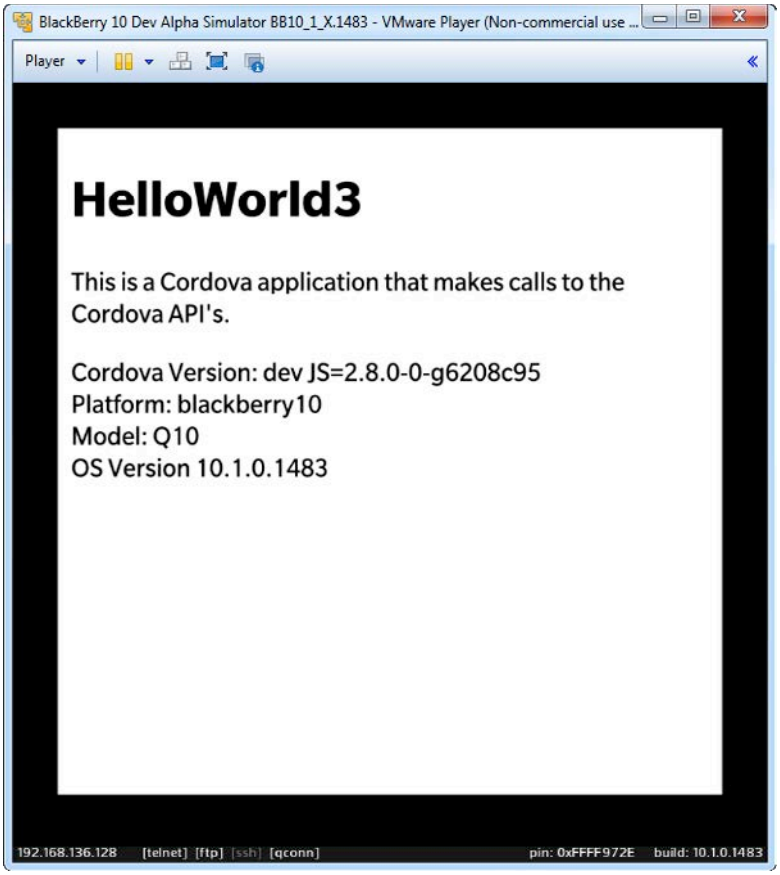



**Figure 5.3** HelloWorld3 Application Running on a BlackBerry Q10 Simulator

In this version of the HelloWorld application, the code in the `onDeviceReady` function has been updated so the program updates a portion of the application's content with an ID of `appInfo` with information about the device running the application and the version of Cordova used to build the application. Device-specific information is available via the Cordova device API (http://cordova.apache.org/docs/en/3.0.0/cordova_device_device.md.html#Device), and this sample application uses a subset of the available methods in this API.

In order for me to be able to call the Device API, I had to add the Device API plugin to the project using the CLI command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device.git
```

# Enhancing the User Interface of a Cordova Application

As you can see from the application examples highlighted so far, the Cordova framework doesn't do anything to enhance the user interface (UI) of a Cordova application. The framework provides access to device-specific features and applications and leaves it up to developers to theme their applications however they see fit. Web developers should use the capabilities provided by HTML, CSS, and even JavaScript to enhance the UI of their Cordova applications as needed. I'm not going to cover mobile web UI design anywhere in this book.

As Android- and iOS-based smartphones became more popular, web developers found themselves needing to be able to build web applications that mimic the look and feel of native applications on these mobile platforms. To accommodate this need, many open source and commercial JavaScript mobile frameworks were created to simplify this task, such as jQuery Mobile (www.jquerymobile.com), Dojo Mobile (www.dojotoolkit.org/features/mobile), and Sencha Touch (www.sencha.com/products/touch).

### Adobe Topcoat

To make it easy for web developers to create good-looking Cordova applications, some developers at Adobe created a small, fast CSS library called Topcoat (http://topcoat.io), which you can use to apply a simple and clean UI to your Cordova applications. Topcoat is a clean wrapper around some common web application UI elements with a library of images and button styles you can use in your application. It's not a full HTML5 framework like Sencha Touch or jQuery Mobile, but it's very useful for creating clean, simple, fast UIs for your web applications.

Although not directly related to Cordova development, the use of these frameworks is very common for Cordova applications, so it's useful to highlight them here. In this section, I discuss how to enhance the UI of a Cordova application using jQuery Mobile (jQM), an offshoot of the popular jQuery project. The jQuery and jQM libraries work together to provide some pretty useful UI elements and theming for any mobile web application.

In the HelloWorld4 application shown in Listing 5.4, I took the HelloWorld3 application and applied an enhanced UI to the application using the jQuery Mobile framework. When running, the application should have a common look across each supported device.

**Listing 5.4 HelloWorld4 application**

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <link rel="stylesheet" href="css/jquery.mobile-1.3.1.css" />
    <script type="text/javascript" charset="utf-8"
      src="js/jquery-2.0.2.js"></script>
    <script type="text/javascript" charset="utf-8"
      src="js/jquery.mobile-1.3.1.js"></script>
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>
```

```
    <script type="text/javascript" charset="utf-8">
      function onBodyLoad() {
        document.addEventListener("deviceready", onDeviceReady,
          false);
      }
      function onDeviceReady() {
        br = "<br />";
        //Get the appInfo DOM element
        var element = document.getElementById("appInfo");
        //Replace it with specific information about the device
        //running the application
        element.innerHTML = 'Cordova Version: ' +
          device.cordova + br +
          'Platform: ' + device.platform + br +
         'Model: ' + device.model + br +
         'OS Version: ' + device.version;
      }
    </script>

  </head>
  <body onload="onBodyLoad()">
    <div data-role="page">
      <div data-role="header" data-position="fixed">
        <h1>Hello World 4</h1>
      </div>
      <div data-role="content">
        <p>This is a Cordova application that makes calls to
        the Cordova APIs and uses the jQuery Mobile
        framework.</p>
        <p id="appInfo">Waiting for Cordova Initialization to
        complete</p>
      </div>
      <div data-role="footer" data-position="fixed">
        <h1>Cordova Programming</h1>
      </div>
    </div>
  </body>
</html>
```

Figure 5.4 shows the HelloWorld4 application running on the Android emulator.

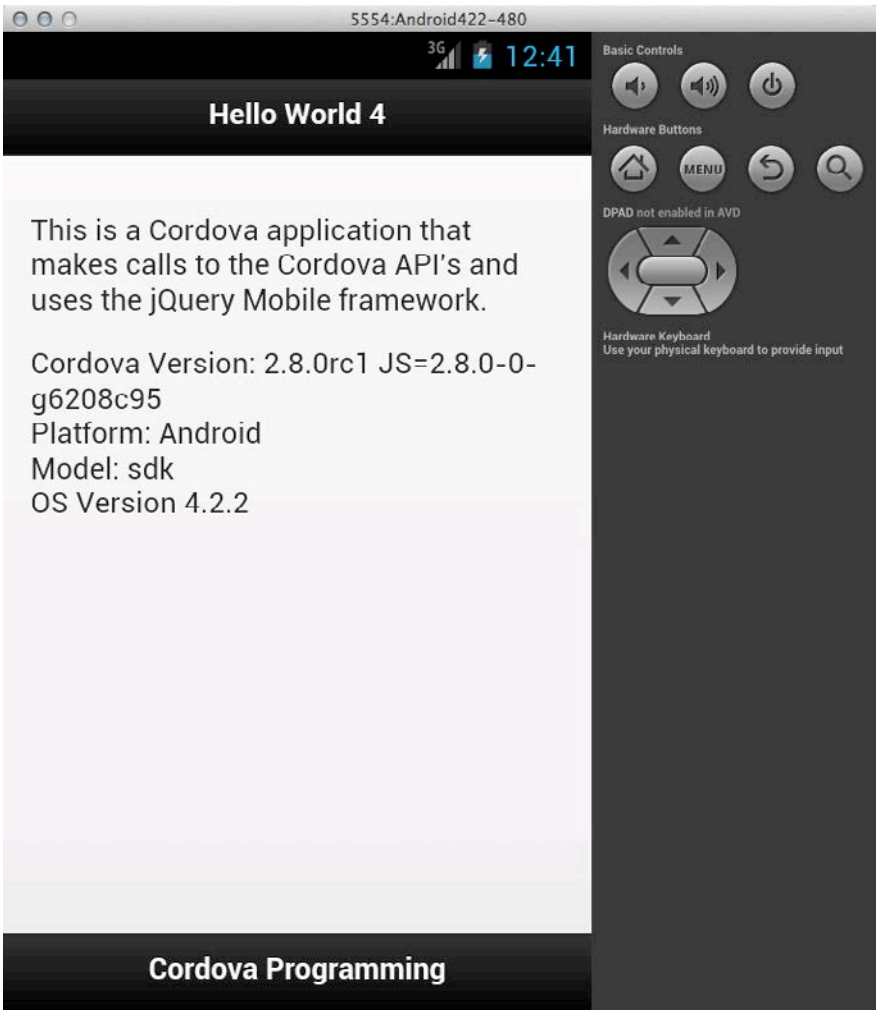



**Figure 5.4** HelloWorld4 Application Running on an Android Emulator

Figure 5.5 shows the same application running on an iOS simulator.

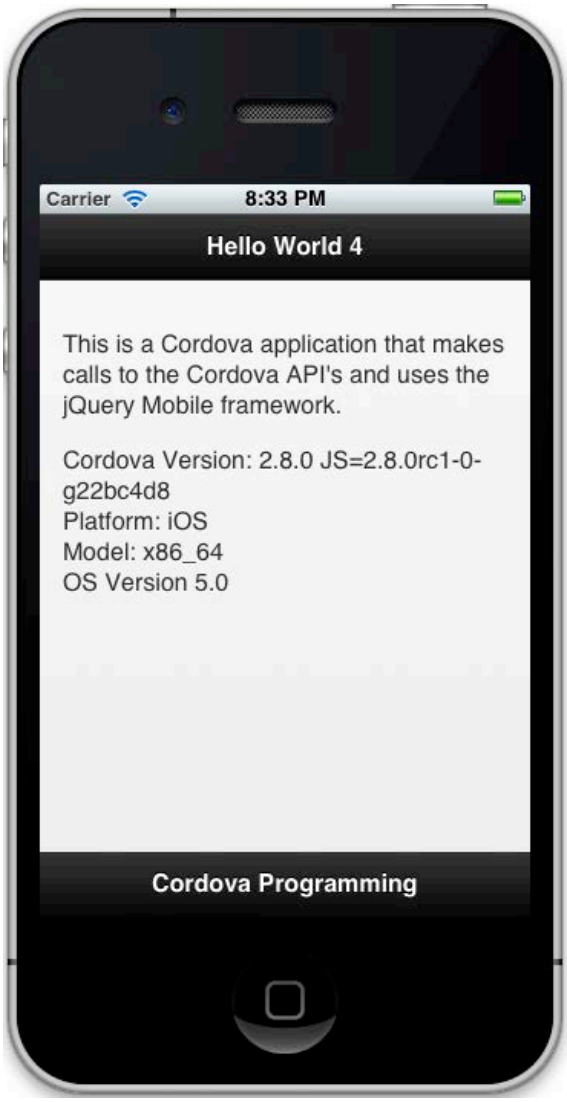



**Figure 5.5** HelloWorld4 Application Running on an iOS Simulator

Figure 5.6 shows the same application running on a BlackBerry 10 simulator.

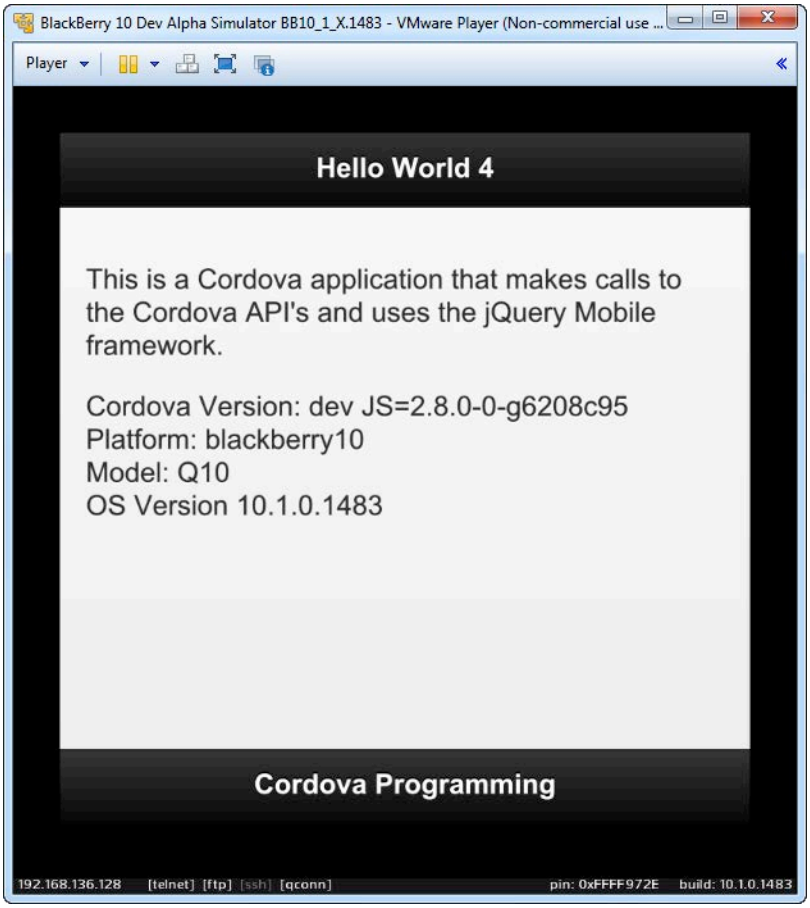



**Figure 5.6** HelloWorld4 Application Running on a BlackBerry 10 Simulator

Notice that the application looks similar across platforms. In a more complicated application, as you navigate deeper into the application, the jQM framework will automatically add mobile device platform features to the application, such as a back button on iOS and support for the escape key and menu on Android and BlackBerry applications.

In this version of the application, some additional resources have been added to the page's header:

```
<link rel="stylesheet" href="css/jquery.mobile-1.3.1.css" />
<script type="text/javascript" charset="utf-8"
  src="js/jquery-2.0.2.js"></script>
<script type="text/javascript" charset="utf-8"
  src="js/jquery.mobile-1.3.1.js"></script>
```

The first line points to a CSS file provided by the jQM framework. It contains the style information used to render the iPhone-ish UI shown in Figure 5.6. Next come references to the jQuery and jQuery Mobile JavaScript libraries that are used to provide the customized UIs plus additional capabilities to the application. The files referenced in the example application are the full versions of the CSS and JavaScript files. These files are used during testing of the application and should be replaced with the min versions of the files, as shown in the following code snippet, before rolling the application into production.

```
<link rel="stylesheet" href="css/jquery.mobile-1.3.1.min.css" />
<script type="text/javascript" charset="utf-8"
  src="js/jquery-2.0.2.min.js"></script>
<script type="text/javascript" charset="utf-8"
  src="js/jquery.mobile-1.3.1.min.js"></script>
```

The min versions are compressed, so comments, white space, line breaks, and so on are removed from the files. Compression allows the files to take up less space within the packaged application, helping to reduce the overall file size for the application, and enables these resources to load more quickly when the user launches the application.

The body of the HTML page has been updated to include several HTML `div` tags wrapped around the content for the application. These divs include a `data-role` attribute that is used by jQM to define specific areas of the content page, which are then styled appropriately depending on which role is assigned.

In Listing 5.4, the content in the section of the page given the `header data-role` is styled with a gradient background and forced to the top of the page by the `data-position="fixed"` attribute. Similarly, the content in the section of the page given the `footer data-role` is styled with a gradient background and forced to the bottom of the page by the `data-position="fixed"` attribute. The page content defined within the `data-role="content"` div will be rendered between the header and footer, with the middle section scrollable as needed to display all of the content within the section. All of this is illustrated in Figures 5.4, 5.5, and 5.6.

These examples only lightly cover the capabilities of jQM; there's so much more you can do with this framework to enhance the user experience within your Cordova applications. Refer to the jQM online documentation or several of the new books on jQM, such as *Sams Teach Yourself jQuery Mobile in 24 Hours* (ISBN: 0672335948), for additional information about the capabilities provided by the framework.

# The Generated Web Application Files

Now that I've shown you how a Cordova application is crafted, let's take a look at the Cordova application generated by the Cordova CLI. As shown in Chapter 3, "Installing the Cordova Command-Line Interface," when the CLI generates a new Cordova application project, it creates a simple HelloCordova web application and places it in the project's www folder. You can see an example of the generated application files in Figure 5.7.

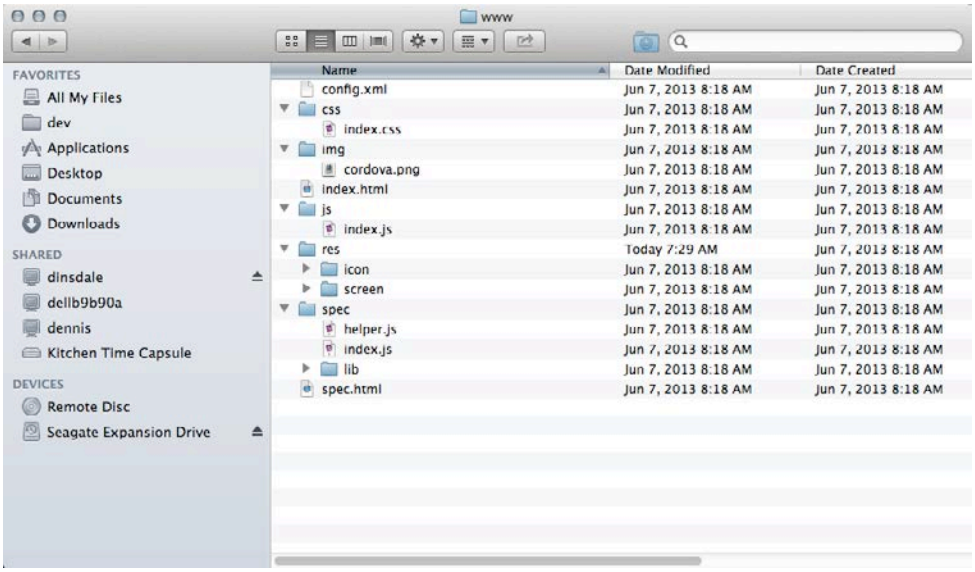



**Figure 5.7** CLI-Generated Web Application Files

The project folder contains a web application folder structure that is designed to separate the different types of files into separate folders. For example, the web application's CSS files should be placed in the css folder, JavaScript files in the js folder, and so on.

The application's index.html file is shown in Listing 5.5; it contains many of the same HTML elements and attributes as the other examples shown throughout the chapter. One difference is that the application loads the cordova.js and other resources at the end of the file rather than at the beginning, as I have shown previously. What this approach does is allow the application to complete loading its HTML content and display the content on the screen before loading the JavaScript files. If you're loading a bunch of JavaScript in your application, it may take some time, so this approach at least allows something to be displayed on the screen as the application's supporting JavaScript code is loaded.

### Listing 5.5 Contents of the HelloCordova index.html file

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html;
      charset=UTF-8" />
    <meta name = "format-detection" content = "telephone=no"/>
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <link rel="stylesheet" type="text/css"
      href="css/index.css" />
    <title>Hello Cordova</title>
  </head>
  <body>
    <div class="app">
```

```
      <h1>Apache Cordova</h1>
      <div id="deviceready">
        <p class="status pending blink">Connecting to Device</p>
        <p class="status complete blink hide">Device is Ready
        </p>
      </div>
    </div>
    <script type="text/javascript" src="cordova.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
    <script type="text/javascript">
      app.initialize();
    </script>
  </body>
</html>
```

What the application does is display a simple page with the Cordova logo and some blinking text, "Connecting to Device," centered beneath the logo. The JavaScript code, in the index.js file shown in Listing 5.6, is called by the application through the call to `app.initialize()` at the very end of the index.html.

### Listing 5.6 Contents of the index.js file

```
var app = {
    initialize: function() {
        this.bind();
    },
    bind: function() {
        document.addEventListener('deviceready',
          this.deviceready, false);
    },
    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');
    },
    report: function(id) {
        console.log("report:" + id);
        //Hide the .pending <p> and show the .complete <p>
        document.querySelector('#' + id + ' .pending').className
          += ' hide';
        var completeElem =
          document.querySelector('#' + id + ' .complete');
        completeElem.className =
          completeElem.className.split('hide').join('');
    }
};
```

The JavaScript code registers the `deviceready` listener you've seen in the HelloWorld3 and HelloWorld4 applications earlier in the chapter. When the `deviceReady` function executes, it writes some information to the console (this is described in Chapter 6, "The Mechanics of Cordova Development"), then updates the page content to indicate that the Cordova container is ready.

This application is much more complicated than it needs to be; as you can see from my previous examples, you can easily do this with much less code. However, it's apparently the way the Cordova team wants to highlight how to build Cordova applications.

> **Note**
>
> In the examples I provide throughout the chapter, I deliberately simplified the application code to make it easier to teach you what a Cordova application looks like. The sample application generated by the CLI is structured more like modern HTML5 applications.
>
> The approach you take when building your web applications is up to you: there's no right or wrong approach. I think the CLI-generated application is more complicated than it needs to be, but as features are added to an application, it may be easier to use the approach highlighted in this section.

Figure 5.8 shows the sample HelloCordova application running on an Android emulator. When building your Cordova applications, you can start with this sample application and add in your custom code, or you can rip out the HTML and CSS files and start from scratch.

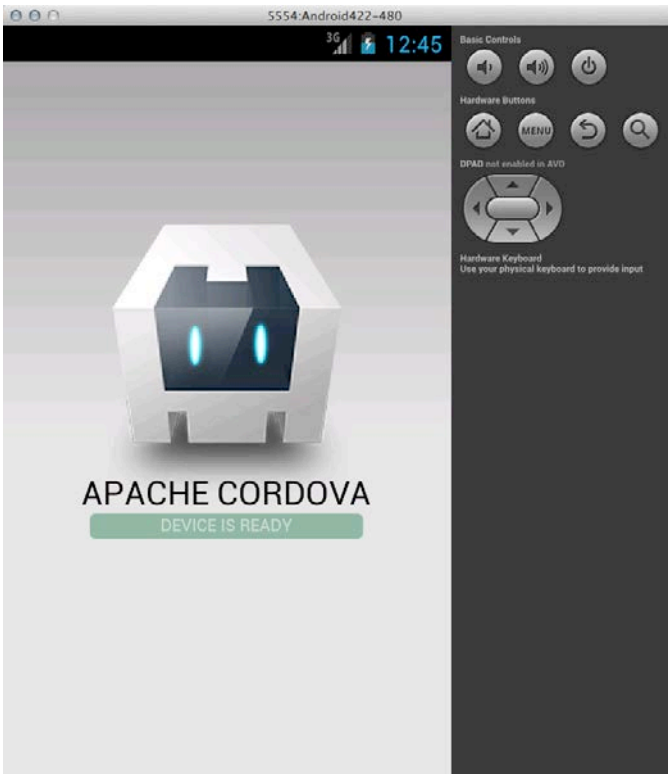



**Figure 5.8** HelloCordova Application Running on an Android Emulator

## Wrap-Up

In this chapter, I showed you what makes an application a Cordova application plus showed you the sample application that the Cordova CLI creates for you. You now have the building blocks necessary to start building your own Cordova applications.

In the following chapters, I show you how to develop, test, and debug your Cordova applications. Chapter 6 provides general guidance on how to work with Cordova applications, and the chapters that follow provide specific guidance for several of the more popular smartphone platforms.

# 6

# The Mechanics of Cordova Development

Each of the mobile platforms supported by Cordova has a process and tools you can use to test and, in the unlikely event your code has bugs, debug Cordova applications. In general, you can load a Cordova application into a device simulator or emulator, provided as part of the mobile platform's SDK, or you can load an application onto a physical device. There are also third-party solutions you can use to test your Cordova applications within a desktop browser interface.

Some processes and capabilities apply across all supported mobile device platforms. In this chapter, I address the mechanics of Apache Cordova development. I begin the chapter by addressing some of the issues a Cordova developer must deal with, then cover the development process and some of the tools you can use to test and debug your Cordova applications.

## Cordova Development Issues

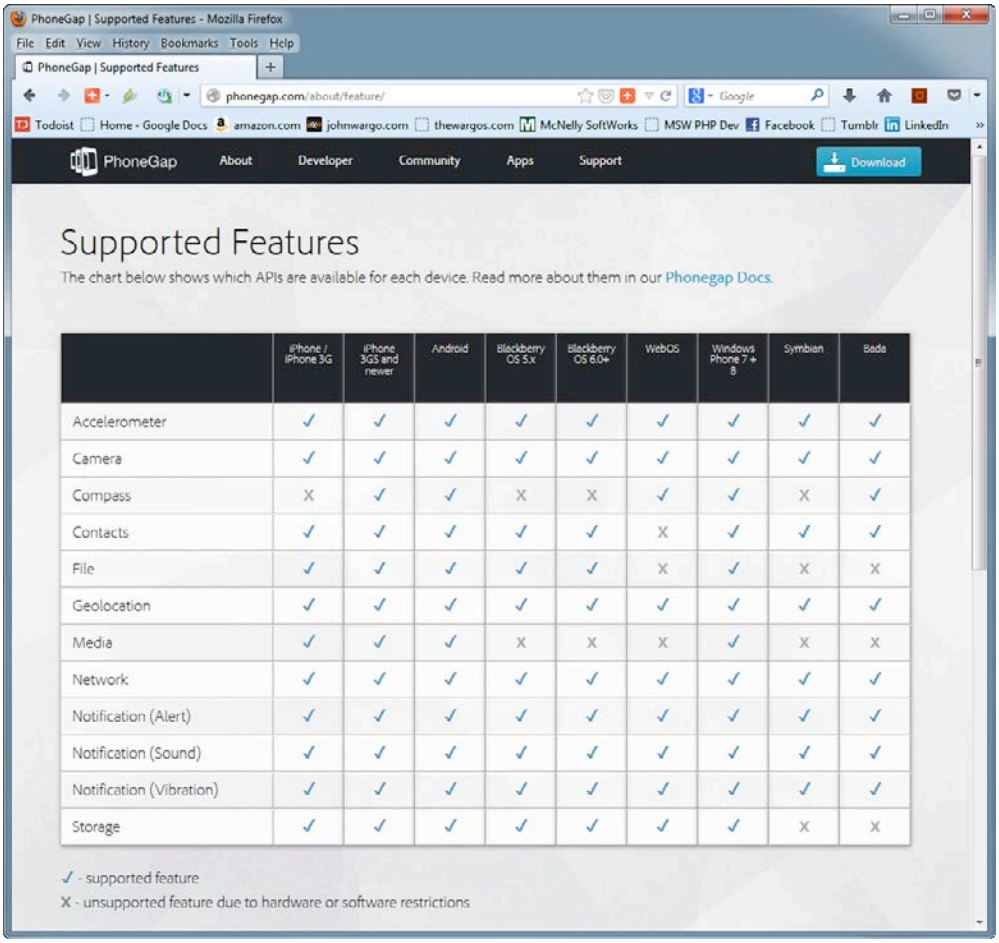
Before we start discussing how to develop Cordova applications, let's address some of the issues that you will face as you work with the framework. The Cordova project is supported by developers from all over the world, developers who may have experience with only one or a small number of mobile platforms, developers who have a strong opinion about how something should be done. The problem is that when you take development projects written by different people and try to collect them into a single framework, you will likely bump up against some inconsistencies. Add the fact that every mobile platform supported by Cordova is different and has different ways of doing things, and you have a difficult task to make everything work cleanly and seamlessly.

In the predecessor to this book, I used this section of the chapter to highlight all of the issues I'd encountered while learning (at the time) PhoneGap and later writing the book. Back then, there were a bunch of issues, and they created some interesting problems for developers. The good news is that over time, the Cordova development team has done an amazing job in eliminating most of them. All that's left are two, and they're not that complicated.

### Dealing with API Inconsistency

Figure 6.1 shows the supported feature matrix from the PhoneGap website (the Cordova team doesn't seem to publish a matrix); you can find the page at http://phonegap.com/about/feature/. As you can see, the table is pretty complete; there are some gaps, but it's more full than empty. If a particular feature you want to use in your application is supported only on some mobile platforms, then you'll

have to make special accommodation within your application for platforms that do not support the particular API.




| | iPhone / iPhone 3G | iPhone 3GS and newer | Android | Blackberry OS 5.x | Blackberry OS 6.0+ | WebOS | Windows Phone 7 + 8 | Symbian | Bada |
|---|---|---|---|---|---|---|---|---|---|
| Accelerometer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Camera | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Compass | X | ✓ | ✓ | X | X | ✓ | ✓ | X | ✓ |
| Contacts | ✓ | ✓ | ✓ | ✓ | ✓ | X | ✓ | ✓ | ✓ |
| File | ✓ | ✓ | ✓ | ✓ | ✓ | X | ✓ | X | X |
| Geolocation | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Media | ✓ | ✓ | ✓ | X | X | X | ✓ | X | X |
| Network | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Notification (Alert) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Notification (Sound) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Notification (Vibration) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Storage | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | X | X |

**Figure 6.1** Cordova-Supported Feature Matrix

> **Caution**
>
> Keep in mind that the table is not updated as often as the API is, so you may want to validate through the API documentation or through actual testing of an API whether or not it works on a platform where there's an *X* in Figure 6.1.

If your application uses an API that isn't supported on all of the mobile devices that your application will target, then your application's code can use the Device API discussed in Chapter 5. Your application should use `device.platform` and, as necessary, `device.version` to determine which platform and OS the application is running on and disable any unsupported feature if the application is running on a device that doesn't support the API. Another option is to simply wrap the call to a particular API with a JavaScript try/catch block and deal directly with any failures that occur.

### Application Graphics

Each mobile platform and, often, different versions of a particular device OS have different requirements for application icons and splash screens. Developers building Cordova applications for multiple device platforms must be prepared to create a suite of graphics for their application that addresses the specific requirements for each target device platform and/or device OS. For application icons, the PhoneGap project maintains a wiki page listing the icon requirements for the different supported operating systems here: https://github.com/phonegap/phonegap/wiki/App-Icon-Sizes.

Additionally, for some devices on some carriers (older BlackBerry devices, for example), mobile carriers apply a specific theme to the OS to help distinguish themselves in the market. Any application icon designed for one of these devices will need to accommodate, as best as possible, rendering pleasantly within different themes. Fortunately, with the merges capabilities described later in this chapter, you have the ability to easily merge the appropriate graphics files (and other content as needed) into your project depending on which mobile platform you are building for.

## Developing Cordova Applications

Now it's time to start working through the process of how to create Cordova applications. In this section I describe the process for coding a Cordova application. In later sections, I show you how to test and debug applications.

### Working with a Single Mobile Device Platform

It's possible that some developers will work with only a single mobile platform. If you are such a developer, all you have to do is open up a terminal window and issue the following commands (which are described in Chapter 4, "Using the Cordova Command-Line Interface"):

```
cordova create app_name
cd app_name
cordova platform add platform_name
cordova prepare platform_name
```

> **Warning**
>
> This isn't necessarily the right way to do single-platform development, as I'll describe later—I'm just trying to describe a potential process here.

In this example, `app_name` refers to the name of the application you are creating and `platform_name` refers to the mobile device platform you will be working with. So, if you were creating a BlackBerry application called lunch_menu, you would issue the following commands:

```
cordova create lunch_menu
cd lunch_menu
cordova platform add blackberry
cordova prepare blackberry
```

You can also specify more information about your application by using the following:

```
cordova create lunch_menu com.cordovaprogramming.lunchmenu "Lunch Menu"
cd lunch_menu
```

```
cordova platform add blackberry
cordova prepare blackberry
```

At this point, the command-line interface (CLI) would create the Cordova project folder shown in Figure 6.2, and all you need to do at this point is open your code editor of choice and start coding and testing your new Cordova application.

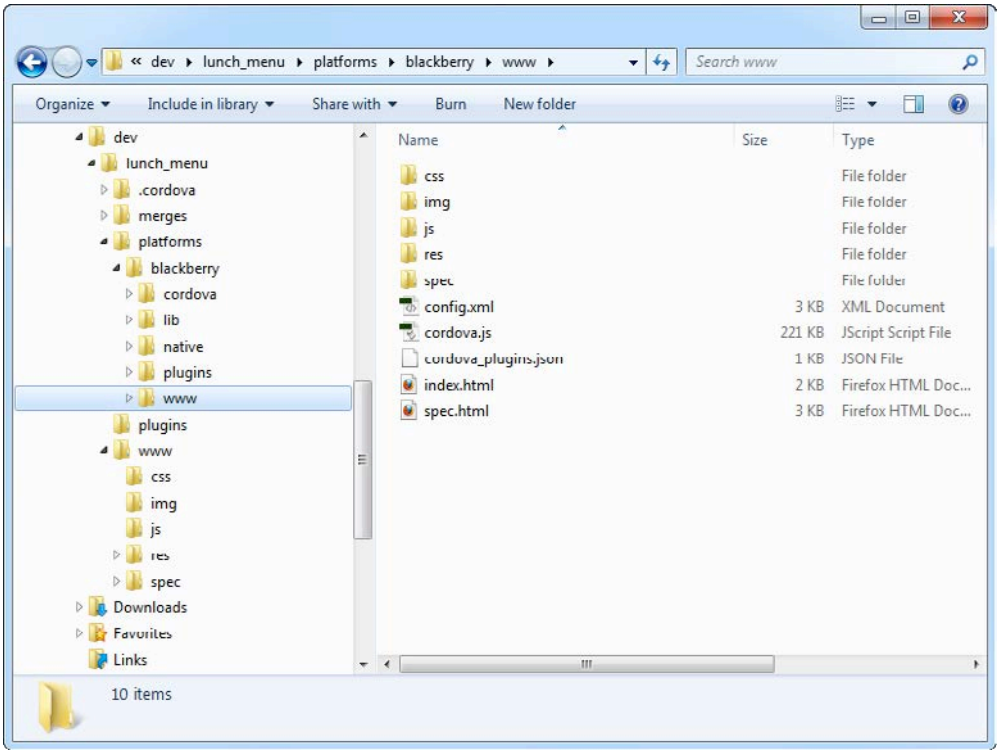



**Figure 6.2** Cordova Application Project Folder Structure: BlackBerry Application

The BlackBerry platform project folder contains a copy of the web application files you need to work with.

> **Note**
>
> If you later decide to add additional mobile device platforms to your project, you need to manually copy the application's web content files from the BlackBerry project's www folder, highlighted in Figure 6.2, over to the www folder within the overall Cordova (not BlackBerry) project shown in the figure. The content that follows describes how multiplatform Cordova projects differ and why this is important.

## Working with Multiple Mobile Device Platforms

Because Cordova is all about cross-platform mobile development, you're probably going to want to target multiple mobile device platforms. In that case, if you were building an app for Android and iOS, for example, you would open a terminal window and do something like the following:

```
cordova create lunch_menu
cd lunch_menu
cordova platform add android ios
```

At this point, you'd have a new Cordova project structure with projects for both Android and iOS, as shown in Figure 6.3. As discussed in Chapter 4, there's a separate folder called www contains the application's core web content files, the content files that are shared across both the Android and iOS projects.

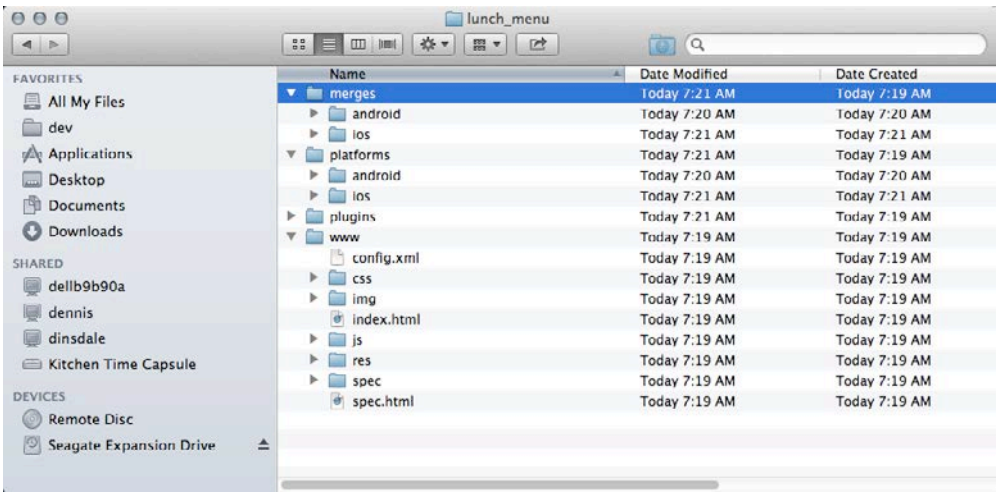



**Figure 6.3** Cordova Application Project Folder Structure

In this scenario, you will work with the web content stored in the www folder, shown at the bottom of the folder structure in Figure 6.3. When you have the web application content in that folder ready for testing, you use the CLI to copy the code into the platforms sub-folders (android and ios), shown in the figure.

What I do while working on a Cordova project is keep my web content files open in an HTML editor such as Adobe Brackets (www.brackets.io) or Aptana Studio (www.aptana.com), then use the CLI to manage my mobile device platform projects for me. As I edit the files, I add the web content to the .html file and my application's code to the application's .js files. When I'm ready to test (and debug) the applications, I switch over to a terminal window that I keep open and pointed to the Cordova project's root folder (the lunch_menu folder I created a while back) and issue some commands. If I want to switch to the Android IDE and test the Android application, I issue the following command:

```
cordova prepare android
```

Or, if I will be testing and debugging both the Android and iOS versions of the application, I issue the following command:

```
cordova prepare android ios
```

What this command does is copy all of the project files from the www folder into the appropriate folder for each specified mobile platform project, as shown in Figure 6.4. In this example, it copies the content files to the Android project's assets/www folder and the iOS project's www folder. The contents of the config.xml file should be applied to the platform-specific config.xml file located in the target directory.

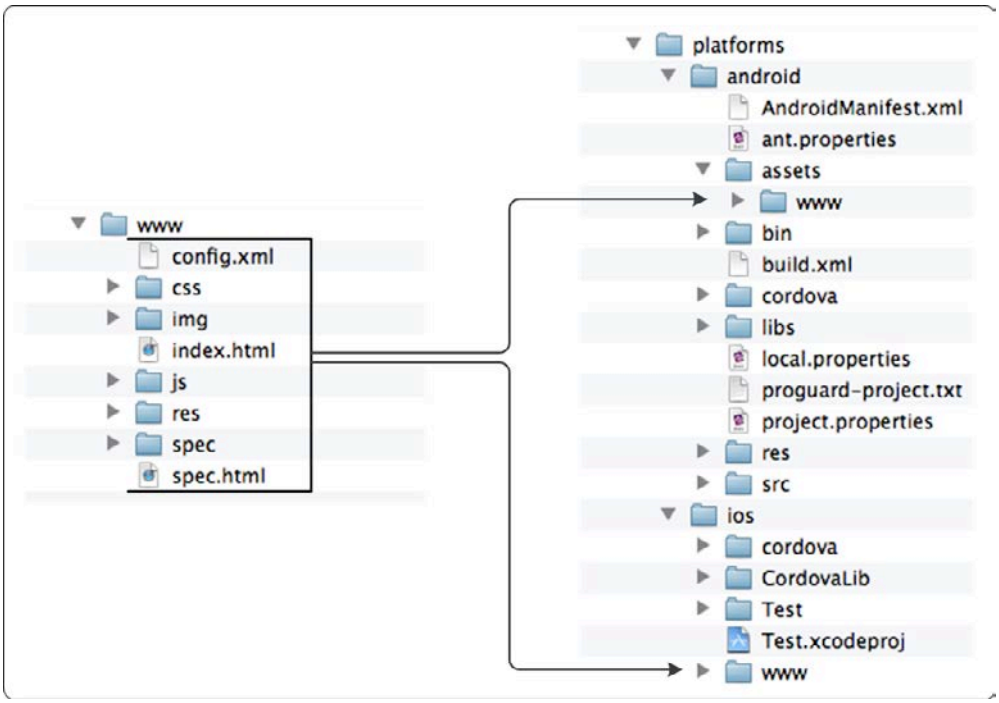



**Figure 6.4** Copying Web Content to the Platform Projects Folders

Now, any self-respecting mobile web project may have some icons, screen graphics, CSS, and/or JavaScript files that are unique to each target platform. Since each mobile device has its own theme and icon requirements, it's likely that at a minimum of those will be required. In older versions of Cordova, you had to manage all of that manually; with the CLI, that's all taken care of for you.

Notice the merges folder shown in Figure 6.3; Cordova uses that folder structure to provide you with a place to store the web application resources that are unique to each target platform. When you issue the Cordova `prepare` commands shown earlier, the CLI copies the custom content for each of the platforms into the appropriate web content folder for each platform's project folder, as shown in Figure 6.5.

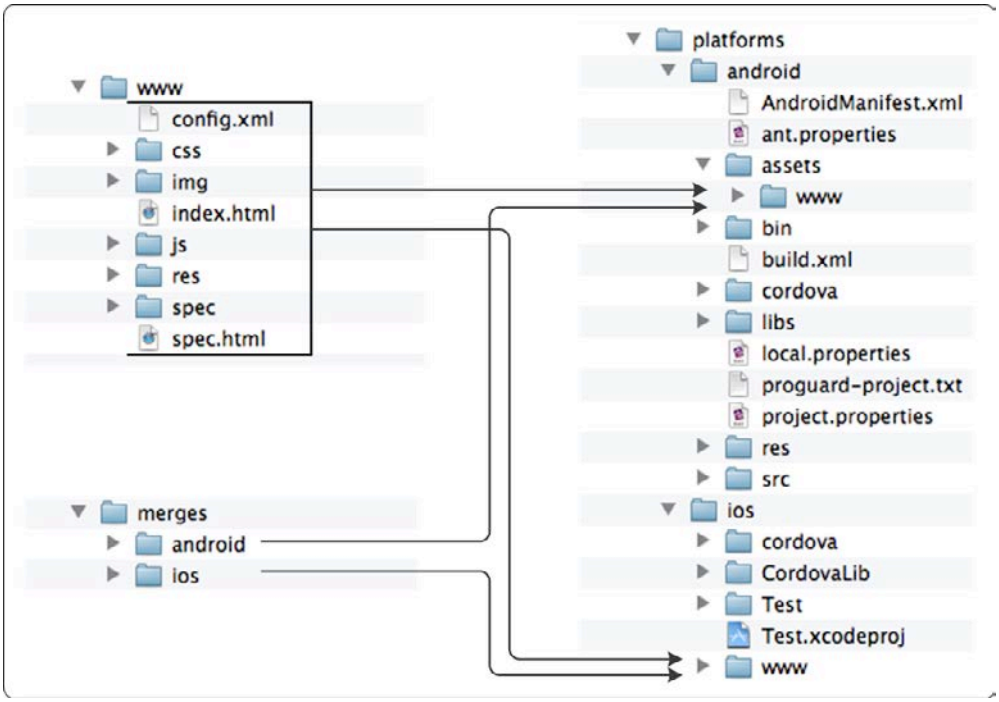



**Figure 6.5** Copying Web Content and Platform-Specific Content to the Platform Projects Folders

As shown in the figure, custom content for the Android platform stored in the merges\android folder is copied into the Android platform project's assets\www folder. Custom content for iOS applications is copied from merges\ios to the iOS project's www folder.

With all of the application's content copied into the appropriate project folders, you open the appropriate IDE (Eclipse for Android and Xcode for iOS) and begin the testing process. For information on how to import the Cordova projects to each IDE and use the platform's debugging tools, refer to Chapters 7 through 10.

# Testing Cordova Applications

You can also skip the IDEs entirely and test the applications directly from the command line; I show you how in the following sections.

## Run a Cordova Application on a Device Simulator

Most mobile device manufacturers provide a software program that emulates or simulates a mobile device. This allows developers to easily test their mobile applications when they don't have a physical device. Performance isn't usually the same, but it looks and acts like a real device much as it can. In some cases, what's provided is generic and simply mimics the capabilities of the specific OS version, while for other mobile platforms it might mimic specific devices. Either way, there's a software-only solution available that developers can use to test Cordova applications in an almost

real-world scenario (I'll explain "almost real-world" in the following section). Google, for example, provides Android emulators, and Apple and BlackBerry provide simulators of their devices.

> **Simulator vs. Emulator**
>
> There is a technical difference between the two that I'm not going to get into here. In order to make things simpler for me (and you), I'm going to dispense with calling out whether I'm referring to an emulator or simulator for the remainder of the book and simply refer to either as simulators. If you see that word going forward, know that I mean either *emulator* or *simulator*.

To run a Cordova application using a device simulator, you would use the following command:

```
cordova emulate device_platform
```

Replace the value for `device_platform` with the name of the mobile device platform you wish to emulate (android, blackberry10, ios, wp8, and so on). For example, to run the application on a BlackBerry 10 simulator, you would issue the following command:

```
cordova emulate blackberry10
```

In this example, the CLI will prepare the files, build the application using the platform's command-line tools, then launch the appropriate simulator and run the application. You saw examples of the device simulators and emulators in the screenshots found in Chapter 5, "Anatomy of a Cordova Application."

## Run a Cordova Application on a Physical Device

Before you deploy your application to mobile users, you should perform final testing on a physical device. As good as these options are, there is always something that doesn't work quite right on a simulator. To test an application on a physical device, connect the device to your development system using a USB cable, then issue the following command:

```
cordova run device_platform
```

For example, to run the application on an Android device, issue the following command:

```
cordova run android
```

Behind the scenes, the CLI will execute the `prepare` command described earlier, then call the particular platform's command-line tools to package the application and deploy it to the device that is connected to the system. Within seconds (or as much as a few minutes in the case of some platforms), the application will appear on the device's screen.

> **Warning**
>
> For many mobile device platforms, applications will not run on physical devices without first being registered with the manufacturer (Windows Phone 8) or signed by an appropriate signing authority (BlackBerry 10, iOS). I'm deliberately omitting the details of this process from this chapter, as it differs across the different supported mobile device platforms and would add some bulk to this manuscript. I cover this topic a little bit in the chapters that deal with each mobile device platform separately (Chapters 7 through 10).
>
> Before testing Cordova applications on a physical device, make sure you have followed the manufacturer's instructions for configuring the appropriate environment to do so.

# Leveraging Cordova Debugging Capabilities

As you test your Cordova applications, you're likely to run into issues that you must resolve. The purpose of this section is to highlight some of the debugging capabilities that are available to you outside of an IDE.

## Using `Alert()`

One of the simplest, and most annoying, ways to debug a Cordova application is to use the JavaScript `alert()` function to let you know what part of the code you're running or to quickly display the contents of a variable. I've always called this approach the "poor man's debugger," but it works quite well for certain types of application debugging tasks. If you see an event that's not firing within your application or some variable that's not being set or read correctly, you can simply insert an `alert()` that displays a relevant message and use that to see what's going on.

As I started working with PhoneGap and PhoneGap Build, I noticed that there were many times when the `deviceready` event wasn't firing in my applications. I would write my application and start testing it only to find that none of the PhoneGap APIs were working. In some cases, it was because the PhoneGap Build service wasn't packaging the phonegap.js file with the application (that's what happens when you use a beta product). In other cases, it was simply because I had some stupid typo in the application that I couldn't see.

> **Warning**
>
> Cordova fails silently when it encounters a JavaScript error, so if you have a typo in your code, the code will simply not run.

What I started doing in my then PhoneGap, now Cordova, applications was to add a call to `alert()` in the `onBodyLoad` and `onDeviceReady` functions of all of my applications during development. In Chapter 5, I provided a listing for the HelloWorld3 application, but the real HelloWorld3 application code is shown in Listing 6.1. In this version of the application, you can see the calls to alert in the `onBodyLoad` and `onDeviceReady`. Once I was certain that the application worked correctly, I would remove the alerts.

**Listing 6.1 The "Real" HelloWorld3 application**

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <script type="text/javascript" charset="utf-8"
      src="Cordova.js"></script>
    <script type="text/javascript" charset="utf-8">
      function onBodyLoad() {
        alert("onBodyLoad!");
        document.addEventListener("deviceready", onDeviceReady,
```

```
        false);
      }
      function onDeviceReady() {
        alert("onDeviceReady!");
        br = "<br />";
        //Get the appInfo DOM element
        var element = document.getElementById("appInfo");
        //Replace it with specific information about the device
        //running the application
        element.innerHTML = 'Cordova Version: ' +
          device.cordova + br +
          'Platform: ' + device.platform + br +
         'Model: ' + device.model + br +
         'OS Version: ' + device.version;
       }
    </script>

  </head>
  <body onload="onBodyLoad()">
    <h1>HelloWorld3</h1>
    <p>This is a Cordova application that makes calls to the
    Cordova APIs.</p>
    <p id="appInfo">Waiting for Cordova Initialization to
    complete</p>
  </body>
</html>
```

When I was writing all of the sample applications for *PhoneGap Essentials*, I even go as far as to put an alert at the beginning of every function in the application. As I learned how and when each event fired, I used the alerts to help me tell what was going on. Now, there are easier ways to do that, which I show you in the next section, but this was just a simple approach to help me as I got started with each API.

> **Warning**
>
> Those of you who know a little bit about the Cordova APIs might be asking, Why did he use `alert()` rather than the Cordova `navigator.notification.alert()` function?
>
> Well, in the `onBodyLoad()` function, it is highly likely that cordova.js hasn't loaded yet, so I can't be sure that the Cordova `navigator.notification.alert()` will even be available. I could have used `navigator.notification.alert()` in the `onDeviceReady()` function because the only time that function runs is when the Cordova `deviceready` event has fired, but for some reason I just kept the two alerts consistent.

## Writing to the Console

The problem with using the approach described in the previous section is that when you fill your buggy code with alerts, you're constantly interrupting the application flow to dismiss the alerts as they come up. For a simple problem, this approach works pretty well, but when debugging more troublesome errors, you need an approach that allows you to let the application run then analyze what

is happening in real time or after the application or a process within the application has completed, without interrupting the application. Cordova applications can do this through the JavaScript `console` object implemented by the WebKit browser-rendering engine.

Using the console object, developers can write messages to the browser's console that can be viewed outside of the running program through capabilities provided by the native SDKs or device simulators. The `console` object has scope at the window level, so it's essentially a global object accessible by any JavaScript code within the application. WebKit supports several options; the most common ones used are listed here:

- `console.log("message");`
- `console.warn("message");`
- `console.error("message");`

Beginning with Cordova 3.0, the console has been removed from the core Cordova APIs and is instead available as a plugin. To add console capabilities to your Cordova project, you must open a terminal window, navigate to the project folder, and issue the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-console.git
```

Now, let's take a look at a sample application that illustrates the use of this feature, as shown in Listing 6.2

### Listing 6.2 Example Application That Writes to the Console

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width,
      height=device-height, initial-scale=1.0,
      maximum-scale=1.0, user-scalable=no;" />
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>

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

      function onBodyLoad() {
        document.addEventListener("deviceready", onDeviceReady,
          false);
      }
      function onDeviceReady() {
        //Just writing some console messages
        console.warn("This is a warning message!");
        console.log("This is a log message!");
        console.error("And this is an error message!");
      }

    </script>
  </head>
  <body onload="onBodyLoad()">
```

```
    <h1>Debug Example</h1>
    <p>Look at the console to see the messages the application
    has outputted</p>
  </body>
</html>
```

As you can see from the code, all the application has to do is call the appropriate method and pass in the text of the message that is supposed to be written to the console.

Figure 6.6 shows the messages highlighted in the Xcode console window. This window is accessible while the program is running on an iOS simulator, so you can debug applications in real time.

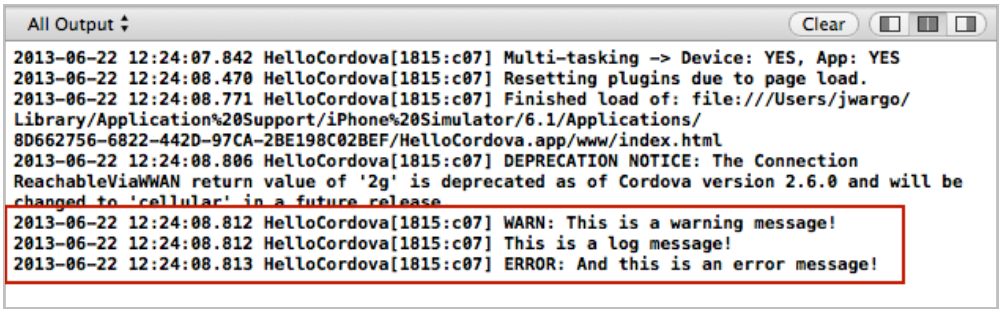



**Figure 6.6** Cordova iOS Application Output Log in Xcode

On some platforms, the console will display `log`, `warning`, or `error` messages differently, making it easier for developers to identify a warning versus an error message. To illustrate this, Figure 6.7 shows the contents of the Android LogCat (described in Chapter 7, "Android Development with Cordova"). Notice that the different console message types are color coded, making it easier for you to spot a particular type of message.

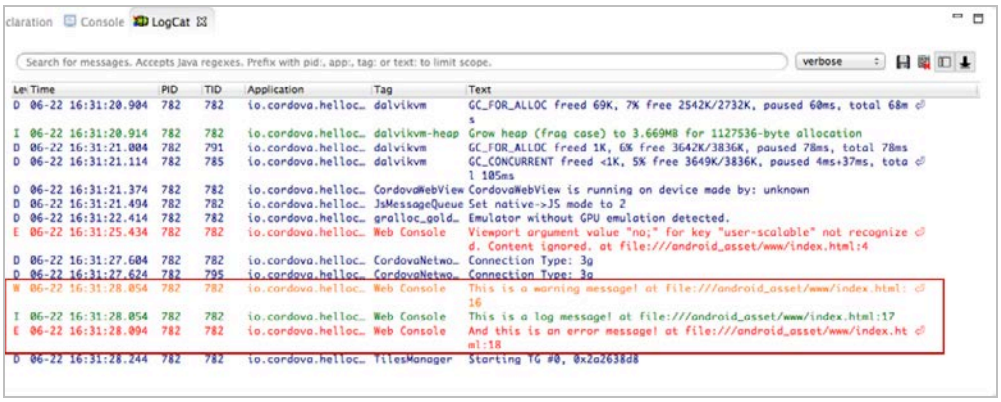



**Figure 6.7** Cordova Android Application LogCat Output in Eclipse

Remember I mentioned in the previous section that the JavaScript code in a Cordova application fails silently? Well, you can also wrap the code in a try/catch block so your application will at least have the chance to write its error to the console, as shown in the following example:

```
try {
  console.log("Validating the meaning of life");
  someBogusFunction("42");
} catch (e) {
  console.error("Hmmm, not sure why this happened here: " +
    e.message);
}
```

Notice that in Figure 6.7, the Android LogCat shows you the line number where the console message was generated. This helps you identify information about where the application is failing. You could also use an alert here, but that's slightly less elegant.

# Debugging and Testing Using External Tools

There's a very active partner community supporting Cordova with additional tools for Cordova developers. In this section, I introduce a couple of the more popular tools that help developers test and debug Cordova applications. This is by no means a complete list of options; refer to the PhoneGap Tools page (http://phonegap.com/tool) for information on additional tools that might be available. Some built-in debugging tools are also available with several of the mobile SDKs. These tools are covered in the individual chapters for each mobile OS (Chapters 7 through 10).

## Debugging Applications with Weinre

Web Inspector Remote (weinre) is a community-built remote debugger for web pages. It was donated to the PhoneGap project and is currently implemented as part of the PhoneGap Build service. You can find the download files and instructions at http://people.apache.org/~pmuellr/weinre/docs/latest.

For Cordova development, it allows you to remotely debug a web application running in a Cordova container on a physical device or a device simulator. Weinre consists of a debug server, debug client, and debug target. The debug server runs on Macintosh or Windows, and the debug client runs in any compatible desktop browser.

To configure weinre, you need to perform a series of steps. The process begins with the server installation. Weinre is Node.js-based, and since we already have Node installed for the Cordova CLI, you can install the server using the following command:

```
npm install -g weinre
```

Unfortunately, on Macintosh weinre may not like your security configuration, so even though it's not recommended, you may have to install weinre using `sudo` using the following command:

```
sudo npm install -g weinre
```

After the installation completes, you should see a message similar to the following:

```
weinre@2.0.0-pre-HH0SN197 /usr/local/lib/node_modules/weinre
├── underscore@1.3.3
├── coffee-script@1.3.3
├── nopt@1.0.10 (abbrev@1.0.4)
└── express@2.5.11 (qs@0.4.2, mime@1.2.4, mkdirp@0.3.0, connect@1.9.2)
```

With the installation completed, you can start weinre by issuing the following command in the terminal window:

```
weinre
```

When the server starts, it will indicate that it is running by displaying a message in the terminal window similar to the following:

```
2013-06-22T17:00:50.564Z weinre: starting server at http://localhost:8080
```

> **Note**
> There are some command-line options you can pass to the weinre server at startup. I chose not to cover them here, but you can find detailed information on the weinre website at http://people.apache.org/~pmuellr/weinre/docs/latest/Running.html.

With the weinre server started, you use a browser-based client application to interact with the server and Cordova client application. Open your browser of choice (I recommend using Safari or Chrome) and point it to the URL shown on the server console when the weinre server started. For my development environment, I simply use:

```
http://localhost:8080
```

The browser will connect to the weinre server and open the weinre debug client, which will display a page similar to the one shown in Figure 6.8.

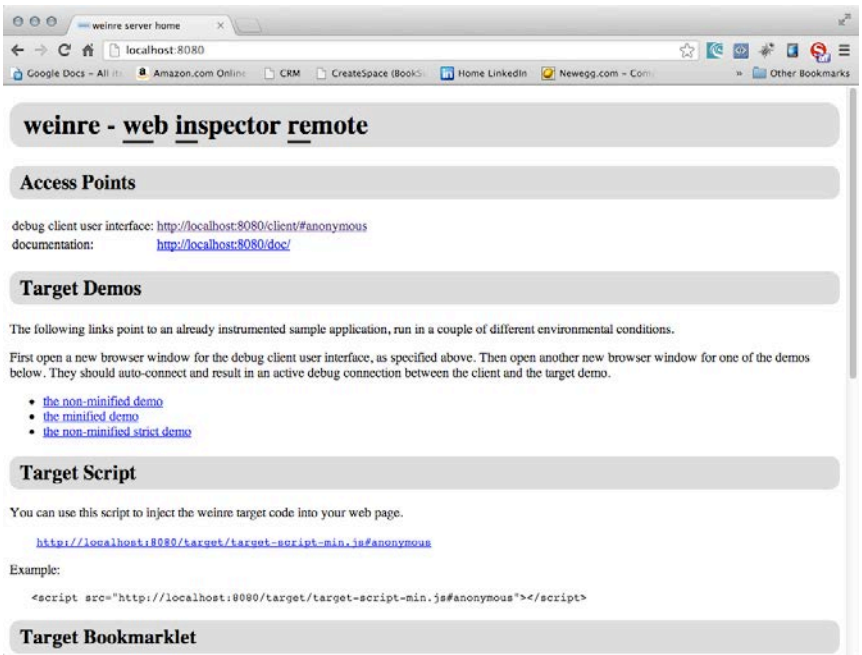



**Figure 6.8** Weinre Debug Client Startup Page

With the server and client running, you can now connect a Cordova application to the debug server by adding the following script tag to `body` section of the Cordova application's index.html file:

```
<script src="http://debug_server:8080/target/target-script-min.js"></script>
```

You need to replace the `debug_server` portion of the URL with the correct host name or IP address for the debug server (the system running the weinre server). This makes the application into a weinre debug target and provides the Cordova application with the code needed to upload information to the weinre server as the application runs.

When using weinre with a device simulator, you can usually point the Cordova application to the local weinre server instance using

```
<script src="http://localhost:8080/target/target-script-min.js"></script>
```

The Android emulator, however, does not have the ability to connect to host-side resources using localhost, so for the Android emulator you must use the host address http://10.0.2.2, as shown in the following example:

```
<script src="http://10.0.2.2:8080/target/target-script-min.js"></script>
```

When using weinre to debug a Cordova application running on a physical device, the device must be able to connect to your debug server. That means that the device must be able to "see" the server on the local network (most likely over a Wi-Fi connection), or the system running the weinre server must have a public facing IP address. Using a server host name of localhost will not work on a physical device; you must use an actual host name or IP address that is visible to the device.

> **Warning**
>
> Be sure to remove the weinre script tag from your Cordova application before releasing it into production. The application will likely hang if attempting to connect to debug server that isn't available.

After you have added the script tag to the Cordova application's index.html file, run the application in the simulator or on a device. Nothing special will appear on the device screen—you can't tell that the weinre debug client is running. However, if you switch to the browser running the weinre debug client and click the first link, shown in Figure 6.8 (the one labeled "debug client user interface"), you will initially see a page similar to the one shown in Figure 6.9.

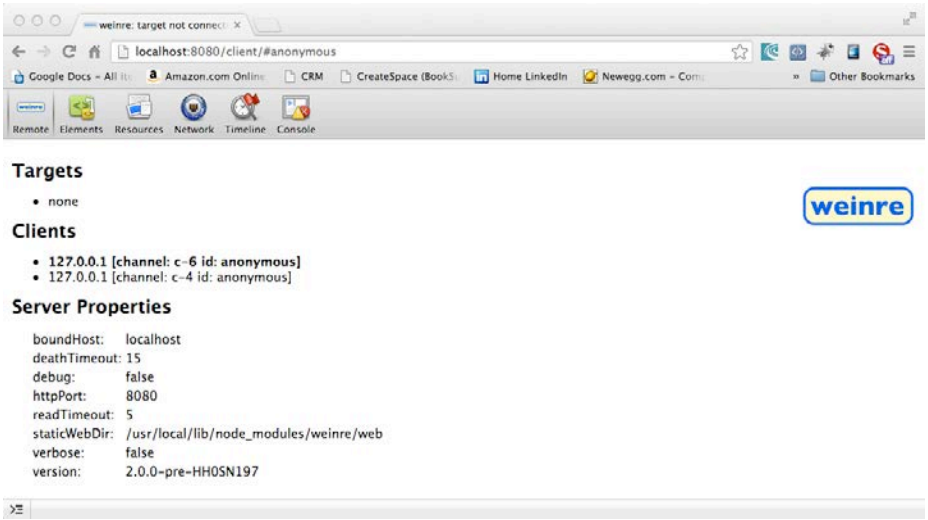



**Figure 6.9** Weinre Debug Client

In this example, the figure is indicating that no targets have connected yet, but as soon as I start my Cordova application, as long as it can connect to the weinre server, the debug client page will update and display the content shown in Figure 6.10.

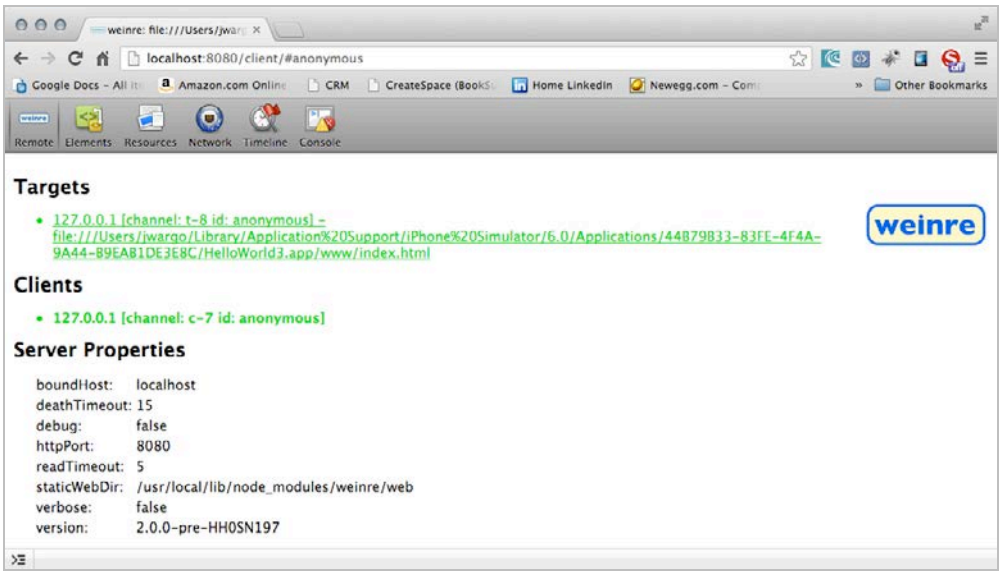



**Figure 6.10** Weinre Debug Client with an Application Connected

The debug client provides the means to view and optionally manipulate many of the page elements and other aspects of your application's web content.

At this point, the different buttons across the top of the debug client are available to provide you with information about the debug target. For example, in Figure 6.11 you see the contents of the Elements page; it shows you the current HTML5 content running within the debug target.

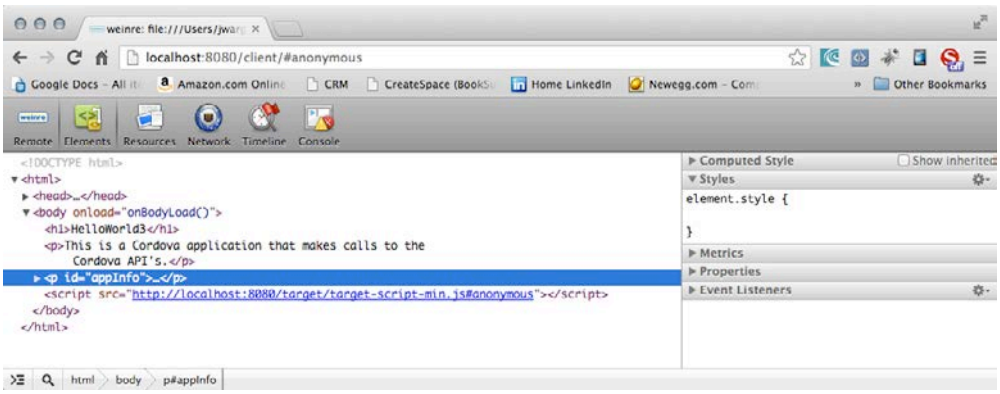



**Figure 6.11** Weinre Debug Client Resources Area

One of the cool features of weinre is that as you highlight the different code sections shown in Figure 6.11, weinre will highlight the corresponding content within the web application. So, for the HelloWorld3 application shown in Figure 6.11, highlighting the paragraph tag `<p id="appInfo">_</p>` reveals, in the debug target, the section of the page shown in Figure 6.12. In this example, I kept the content of the paragraph tag collapsed in the debug client. You can click the black triangle to the left of the `<p>` element to see the complete HTML content.

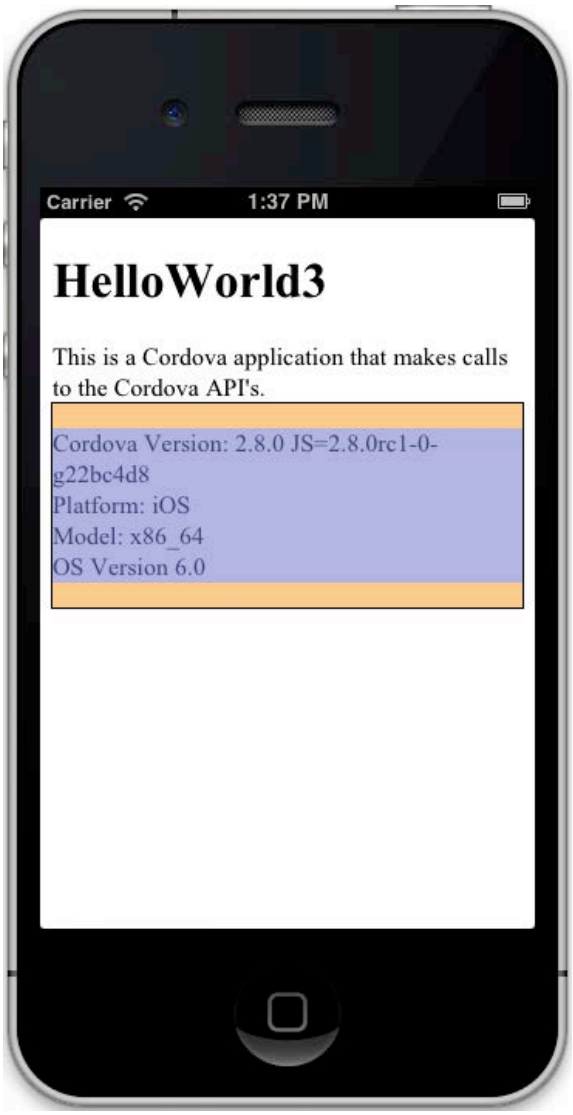



**Figure 6.12** Weinre Target Highlighting HTML Content

Using the debug client, you can access the following content areas:

- Elements: The HTML, CSS, and JavaScript code for the application
- Resources: Local resources used by the application, such as databases, local storage, and session storage
- Network: Information about requests made using the XMLHTTPRequests (XHR)
- Timeline: Events that occur within the target application
- Console: Information written to the console using the `console` object described earlier in the chapter

The available documentation for Weinre is pretty light, but since the project's capabilities are based on the Google Chrome Developer Tools, you can find additional information on the Google Code website at http://code.google.com/chrome/devtools/docs/overview.html.

## Testing Applications Using the Ripple Emulator

The Ripple Emulator is a tool you can use to help with the initial testing of your Cordova application. Ripple is a browser-based emulator that can be used to emulate several different systems. Originally created by Tiny Hippos, which was then acquired by Research In Motion (now called BlackBerry), Ripple is now an incubator project at Apache. The problem with Ripple is that it's been in beta for a very long time (almost two years by my counting), and the emulator is way behind on its Cordova support (supporting Cordova 2.0 when Cordova 2.8 was just released). Because of those limitations, I don't go into too much detail about how Ripple works.

Ripple emulates the execution of the Cordova APIs within the browser container. You can use Ripple for quick testing of Cordova application features and UI during development, then switch to packaging/building Cordova applications and testing them on actual devices or device simulators for more thorough testing. Ripple is not designed to replace testing on real devices or simulators.

Since Ripple was a BlackBerry project for a while, it has a lot of features that help BlackBerry developers. You can, for example, test your BlackBerry WebWorks applications using Ripple, then package them into WebWorks applications directly from the browser. You can learn more about Ripple's capabilities at https://developer.blackberry.com/html5/documentation/getting_started_with_ripple_1866966_11.html.

The emulator installs as a Google Chrome plugin, so you will need to install Chrome from www.google.com/chrome before you begin. Because Ripple is an incubator project and may become a full Apache project at any time, any URL I give you now may be invalid by the time you read this. So, to install Ripple, you should point your browser to http://emulate.phonagep.com. Follow the links on that page to download and install Ripple in your instance of the Chrome browser.

Once you have Ripple installed, you must enable file access for Ripple. In Chrome, open the settings page, then select the Extensions section. In the list of plugins that appears, enable the "Allow access to file URLs" option, shown in Figure 6.13.


**Figure 6.13** Enabling Ripple File Access in the Chrome

Once the browser is configured, open your application's index.html file in the browser. You can press Ctrl-O on Windows or Command-O on Macintosh to open the File Open dialog. Once the page has loaded, you need to enable Ripple for the selected page. To do this, click the Ripple icon to the right of the browser's address bar to open a window allowing you to enable Ripple for the loaded page. You can also append `?enableripple=true` to the end of any URL to enable Ripple emulation for that page.

With Ripple enabled, the browser will display a page that prompts you to identify which type of emulation you wish to enable, as shown in Figure 6.14. As you can see, Ripple can emulate Apache Cordova plus several other platforms and frameworks. Click the Cordova 2.0 button to continue.

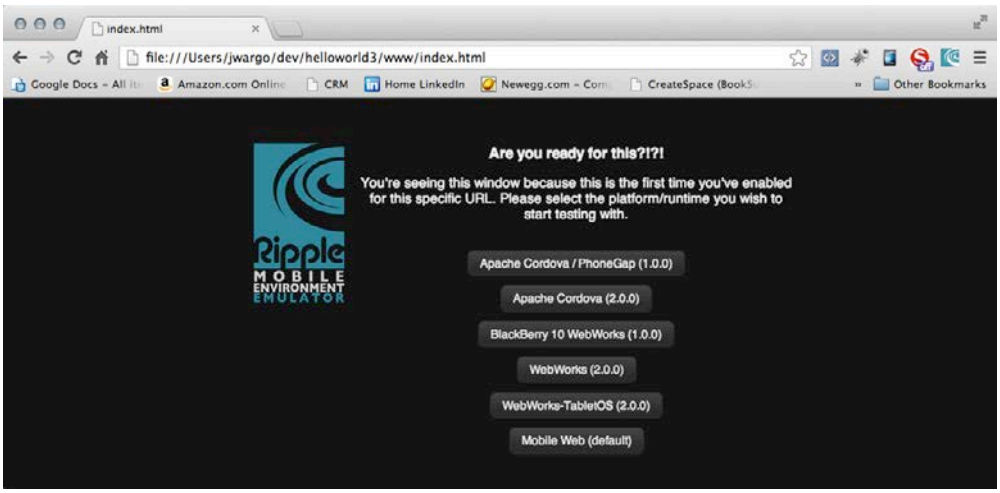



**Figure 6.14** Ripple Emulation Platform Selection Page

At this point, Ripple will display a page with the content from the index.html file rendered within the boundaries of a simulated smartphone screen, as shown in Figure 6.15. Wrapped around the simulated smartphone are properties panes that can be used to configure options and status for the simulated smartphone, such as simulated device screen resolution, accelerometer, network, geolocation, and more.

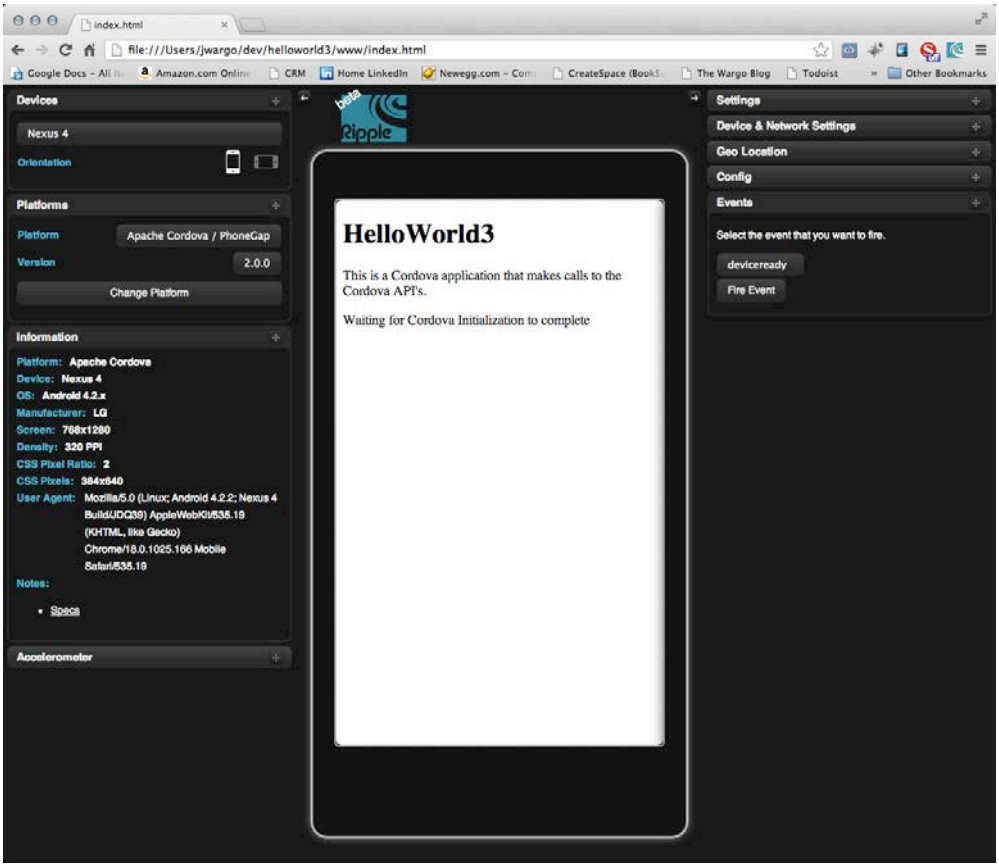



**Figure 6.15** Ripple Emulator Running a Cordova Application

You can click on each of the tabs to expand the options for the tab and make changes to the simulated device's configuration. At this point, you would simply click around within the simulated smartphone screen and interact with the options presented within your application. When you find a problem or a change you want to make within the Cordova application, simply return to your HTML editor, make the necessary changes, write the changes to disk, then reload the page in the Chrome browser to continue with testing.

# Wrap-Up

Hopefully by now, you have an inkling of how to build and debug Cordova applications. In the four chapters that follow, I show you how to use the platform-specific development tools and debugging capabilities.

# 7

# Android Development with Cordova

Google offers developers a robust suite of tools they can use to develop applications for the Android platform. Even though the Cordova CLI takes care of most of the process of creating, managing, and testing applications, there will be times when you want to have more control over the process. Even though the CLI can launch a Cordova application in the Android platform emulator, when you encounter problems with an application, and you want to know more about what's going on, you'll need to use the development tools that come with the Android Development Tools (ADT).

Instructions for using the CLI to create and manage Android projects can be found in Chapter 4, "Using the Cordova Command-Line Interface." In this chapter, I show you how to configure ADT and use it to help you test and debug your Cordova applications.

If you're not developing for Android or you plan on using only the CLI and/or PhoneGap Build to build and test your applications, then you can likely skip this chapter. However, there are some cool tools included with ADT, so what you'll learn in this chapter should simplify your Android application testing and debugging efforts.

## Working with the Android Development Tools

In Chapter 3, I briefly showed how to install the Android Development Tools, but I really didn't cover the topic too deeply or show you how to use them. By default, ADT ships with a preconfigured version of the open-source Eclipse IDE. You can use this IDE to edit, compile, run, and debug Android Java applications; you can learn more about and download ADT at the Android Developer web site at http://developer.android.com/sdk/index.html.

If you're already using Eclipse for other development work, you can add ADT to an existing Eclipse installation using instructions found at http://developer.android.com/sdk/installing/installing-adt.html. The process is pretty easy and doesn't take a lot of time. Be sure to check the system requirements to make sure the version of Eclipse you are using is compatible before attempting to add ADT. Google is really good about keeping up with Eclipse and supports a wide range of Eclipse versions.

ADT includes command-line tools a developer can use to interact with a device or emulator from the command line. This mechanism is one of the tools the Cordova CLI uses to interact with the Android platform. I'm not going to spend any time on the command-line tools here, though.

# Using the ADT IDE

To start the ADT IDE, navigate to the folder where you extracted the Android SDK and launch the Eclipse application located in the eclipse folder. The application will be called eclipse on Macintosh OS and Eclipse.exe on Microsoft Windows. When you first start ADT, you will see a screen similar to the one shown in Figure 7.1.

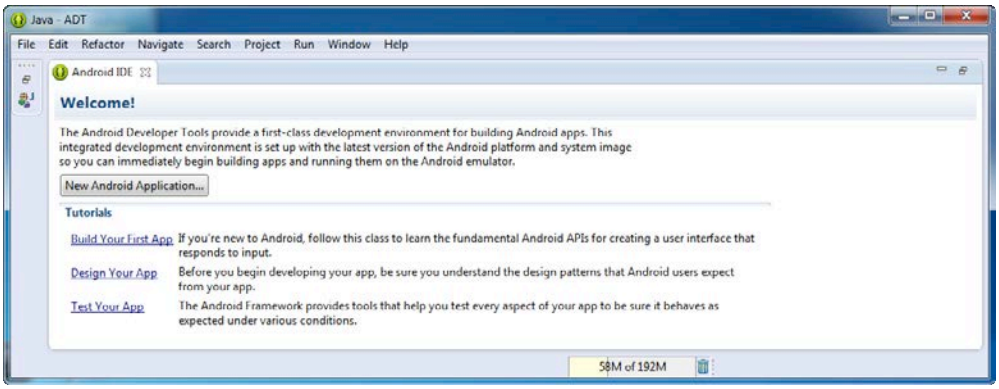



**Figure 7.1** Android Development Tools (ADT) IDE

## Dealing with ADT IDE Memory Problems

When I first started working with ADT on my Windows laptop, I ran into a lot of problems with the IDE crashing whenever I tried to open Cordova project files or add plugins to Eclipse. After many hours of troubleshooting and searching on the web, I finally located the problem. By default, the Eclipse instance in ADT is configured to utilize a limited amount of system memory; ADT was crashing simply because it didn't have the memory it needed to do the things it wanted to do.

There are several ways to increase the amount of memory consumed by different parts of Eclipse. You can pass memory configuration settings as command-line options to Eclipse when you start it, or you can make some simple updates to ADT's eclipse.ini file. I chose to use the latter approach. I'm not sure what the exact settings need to be; your mileage will vary, but I simply opened up the eclipse.ini file and cranked up the memory settings highlighted in bold in Listing 7.1. If you have problems with random crashes in ADT, increase those memory settings and see if they go away.

**Listing 7.1 Contents of the Android Development Tools eclipse.ini File**

```
-startup
plugins/org.eclipse.equinox.launcher_1.3.0.v20120522-1813.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.200.v20120913-144807
-product
com.android.ide.eclipse.adt.package.product
--launcher.XXMaxPermSize
512M
-showsplash
com.android.ide.eclipse.adt.package.product
```

```
--launcher.XXMaxPermSize
512m
--launcher.defaultAction
openFile
-vmargs
-XX:MaxPermSize=512m
-Dosgi.requiredJavaVersion=1.6
-Xms512m
-Xmx1024m
-Declipse.buildId=v22.0.1-685705
```

## Editing Cordova Application Content Files

Android applications are built using Java, but developers can also code portions of their applications in C or C++. Since Android supports only those limited options for application development, ADT does not include tools specifically designed to help with debugging web applications such as those that run within the Cordova container. So, you won't be able to step through an application's JavaScript code, set breakpoints, configure watch expressions, and so on, within the ADT IDE. To debug Android applications, you need to use the debugging approaches highlighted in Chapter 6, "The Mechanics of Cordova Development." However, these debugging approaches can be enhanced using some of the tools described later in this chapter.

> **Note**
>
> During the book's technical review process, Damien pointed out that Google offers a Chrome plugin that enables remote debugging of Android Cordova applications. Unfortunately, there was no time available to add content about this feature to the manuscript, so I will read up on this capability and post updated content to my personal blog at www.johnwargo.com and the book's website at www.cordovaprogramming.com.

Although the CLI is optimized for editing a Cordova application's web application source code in the www folder rather than within one of Cordova project's platform folders, you will likely use an external web content editor such as Adobe Dreamweaver, Adobe Brackets, or some other tool. Since ADT is configured for editing Java and C applications, it doesn't include editors to help with editing the HTML, CSS, and JavaScript files your Cordova application will use. Because of this limitation, if you want to use Eclipse to edit your Cordova application content, you need to install web content editing capabilities into ADT. The default tools for web content editing in Eclipse are from the Eclipse Web Developer Tools Project, and you can install them in ADT in a few minutes.

Within ADT, open the Help menu and select Install New Software. ADT displays the Install wizard, shown in Figure 7.2. In the Work With dropdown field shown in the figure, select the Juno option. Juno is the version of Eclipse that ADT is currently using—you may find yourself using a newer version, so look for the entry that points to http://download.eclipse.org/releases/ instead of http://download.eclipse.org/eclipse/updates/. After you make the selection, the list of options will populate: scroll down and select Eclipse Web Developer Tools, as highlighted in the figure, and step through the wizard to complete the installation.

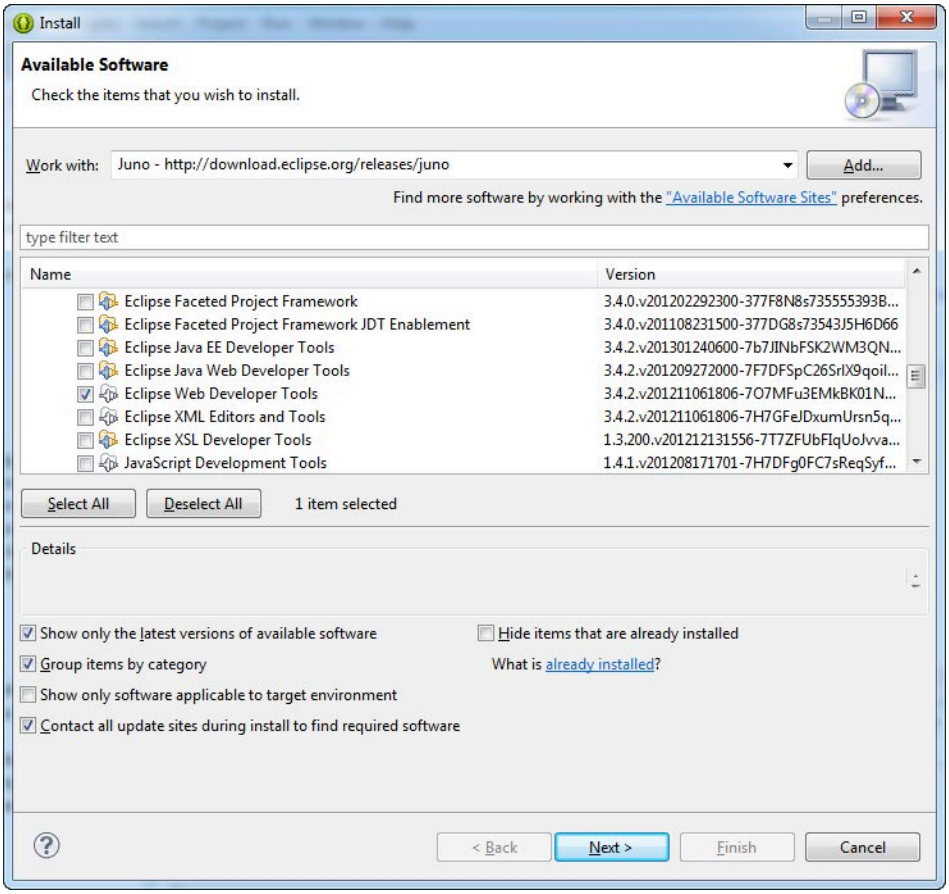



**Figure 7.2** Eclipse Available Software Wizard

## Importing the Cordova Project

Assuming you followed the instructions in Chapter 4 to create an Android project, you should have a Cordova application project in a folder structure similar to the one shown in Figure 7.3. If you haven't already copied over the project's web content from the www folder to the Android project's folder, be sure to issue the Cordova `prepare` command now to complete that process:

```
cordova prepare android
```

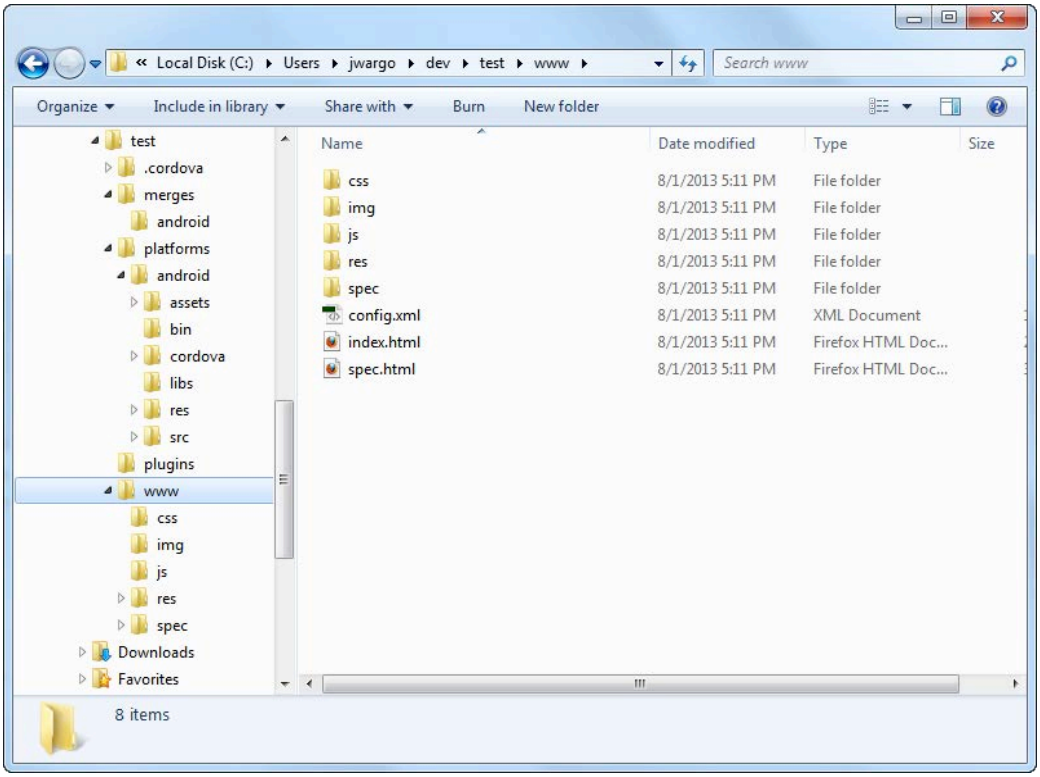



**Figure 7.3** Cordova Project Folder Structure

With the project folder in place, you must import the project into ADT before you can work with it. To start the import process, in the ADT IDE open the File menu, then select Import. The ADT IDE displays the Import wizard, shown in Figure 7.4. Expand the Android option and select Existing Android Code Into Workspace, as shown in the figure, then click the Next button.

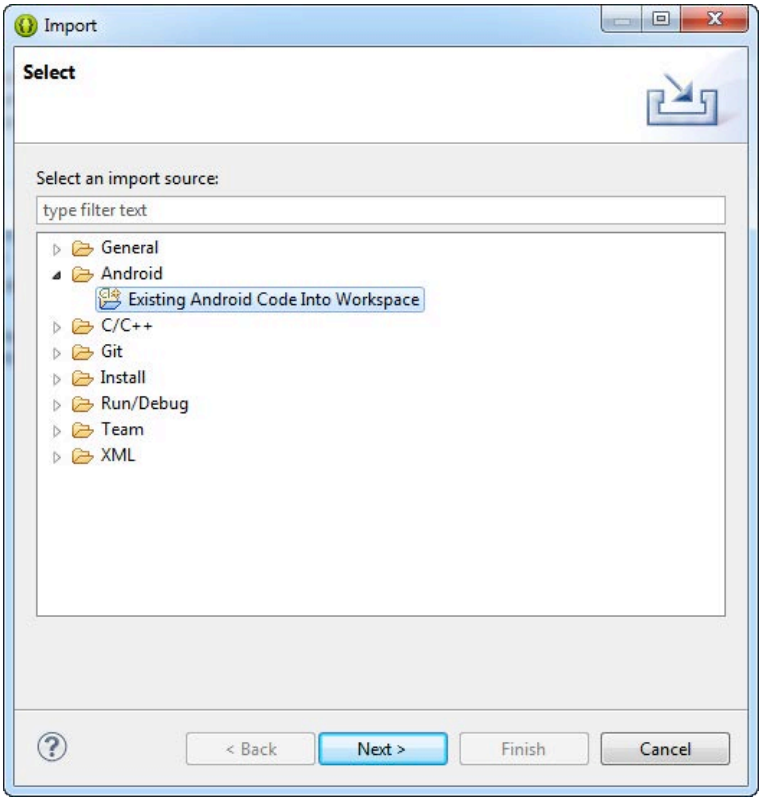



**Figure 7.4** ADT IDE Import Wizard

ADT displays the next page in the Wizard, shown in Figure 7.5. In this dialog, you need to populate the Root Directory field with the location where the Cordova project's Android project files are located. Click the Browse button and navigate to the Android folder, highlighted in Figure 7.3, then click the OK button. The Wizard should automatically add the HelloCordova project to the list of available projects, as shown in Figure 7.5.

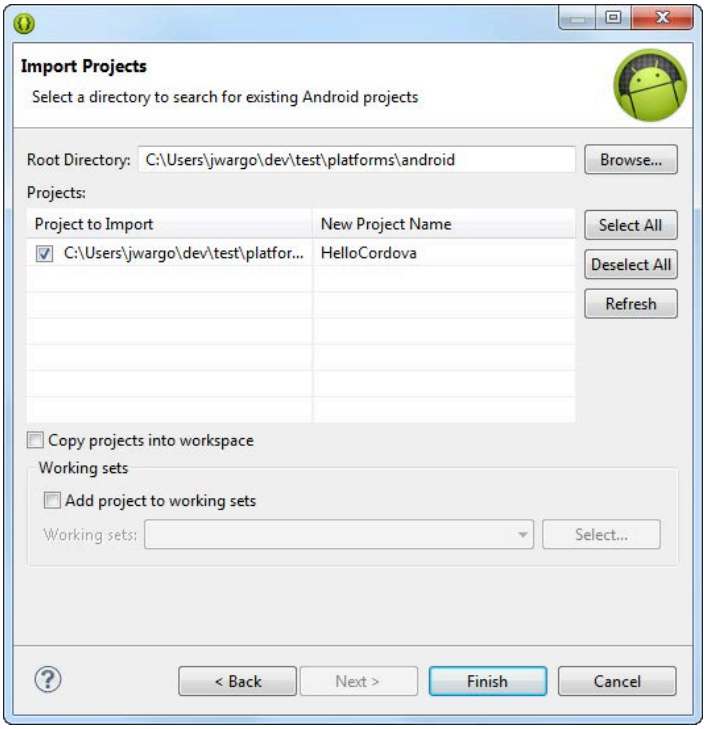



**Figure 7.5** ADT IDE Import Projects

When you click the Finish button, ADT should complete the import process and open the imported project into the IDE, as shown in Figure 7.6. Since the Cordova project consists of both a native Java Android application plus the web content that executes within the application, you will see a blending of a default Android project structure plus the Cordova www folder discussed in previous chapters.

### Warning

Any changes you make to the web application content stored in the Android project's www folder will not be propagated back to the Cordova project's main www folder. The CLI doesn't currently have a mechanism to do so (who knows, perhaps there will be one by the time you read this). If, during your debugging of the application, you make changes to the web application in ADT, you need to manually copy those changes back to the Cordova project's root www folder.

To make this easier, you could add an External Tool to ADT that triggers a copy of the web content files back to the Cordova project's www folder.

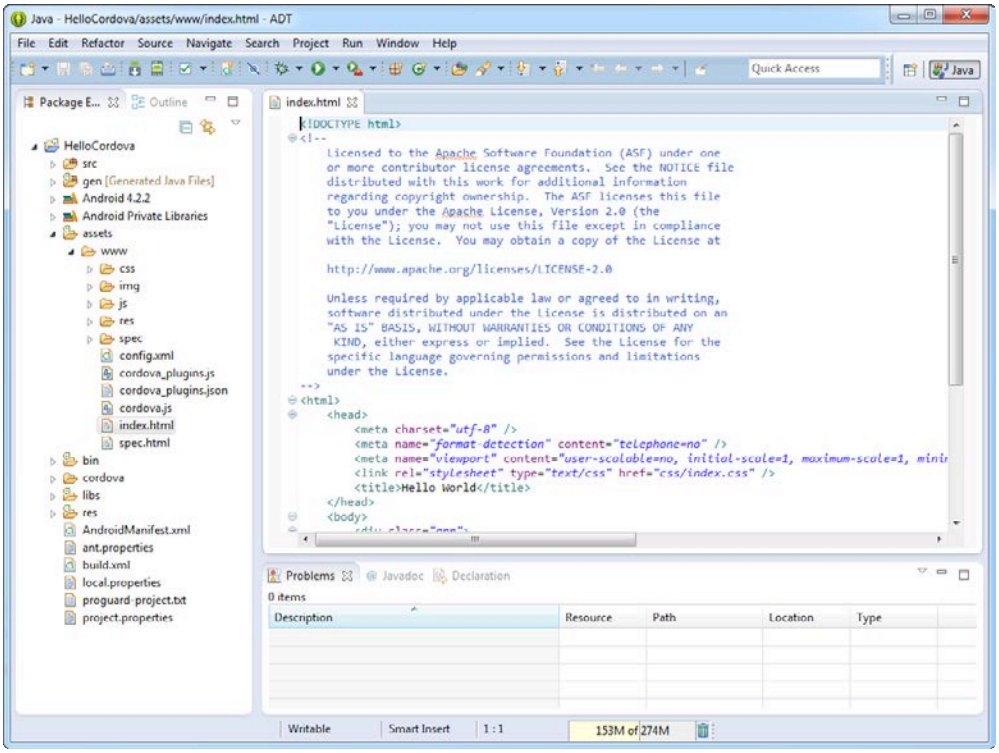



**Figure 7.6** ADT IDE Imported Project

## Running Your Cordova Application

Now that the Cordova project has been imported, you probably want to run it to see how it works. The ADT IDE doesn't have any Android run configurations defined by default, so you have to add at least one before you begin. In the ADT IDE, open the Run menu and select Run Configurations. The IDE displays a dialog similar to the one shown in Figure 7.7.

In the dialog, select Android Applications, then click the New button, highlighted in the figure. Define the settings for the Run Configuration, assigning an application to it, and determine parameters around how it launches one or more Android device emulators when it runs. If you don't have any emulator devices defined (by default, you won't), click the Manage button to open the Android Virtual Device (AVD) Manager to create and manage the system's emulator definitions.

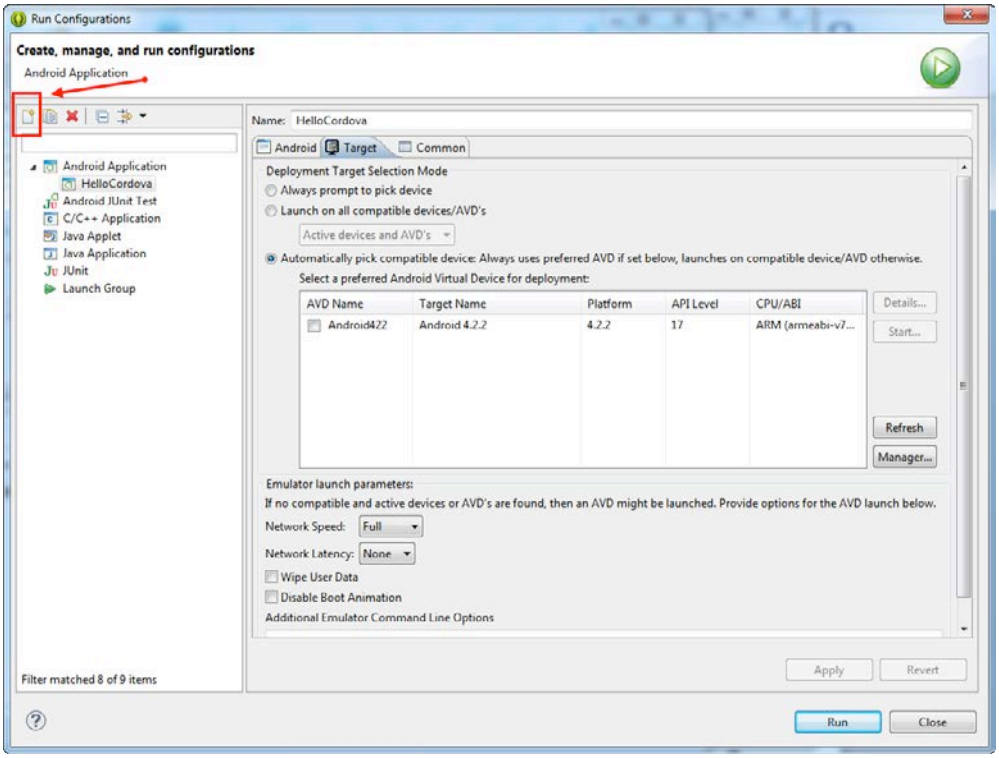



**Figure 7.7** ADT IDE Run Configurations

With one or more Run Configurations defined, you can click the Run button, shown at the bottom of the figure, to launch the emulator and run your application. You can also close this dialog, then run the application on the selected Android emulator by making the appropriate selection from the Run menu or by right-clicking on the application project in the Project Explorer (shown on the left side of Figure 7.6), selecting Run As, then selecting Android Application.

ADT will launch the selected Android emulator (this will take a very long time—be patient, the Android emulators are not known for being fast), then compile, package, and deploy the Cordova application to the emulator.

## ADT Debugging Tools

As the ADT IDE processed everything and launched your Cordova application, you may have noticed a lot of information scrolling by on certain parts of the screen. ADT includes utilities that allow a developer to monitor activity on an emulator or physical device. This functionality is provided by the Console and LogCat views in the IDE. LogCat is part of the IDE but can also run through the command line or as a standalone utility.

ADT opens the LogCat window automatically, but if it gets closed for any reason, you can open it by opening the Window menu, selecting Show View, then selecting Other. The ADT IDE will display a dialog similar to the one shown in Figure 7.8. Expand the Android option to see the list of views available to you. Select LogCat (not the deprecated one shown in the figure) and click the OK button to open the view.

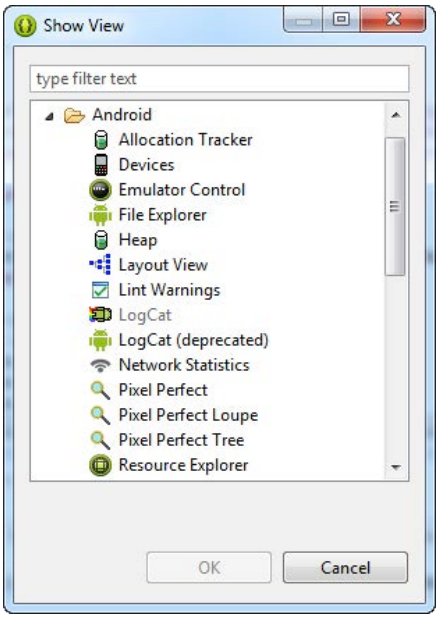



**Figure 7.8** ADT IDE Show View Dialog

The console window shows messages generated by the ADT IDE as it prepares the Cordova application and deploys it to the emulator. You can see an example of the console's output in Figure 7.9. If the ADT IDE has trouble building, packaging, or deploying your application, it lets you know in the console window. If you've launched the app in the emulator, but nothing seems to be happening, take a look at the console.

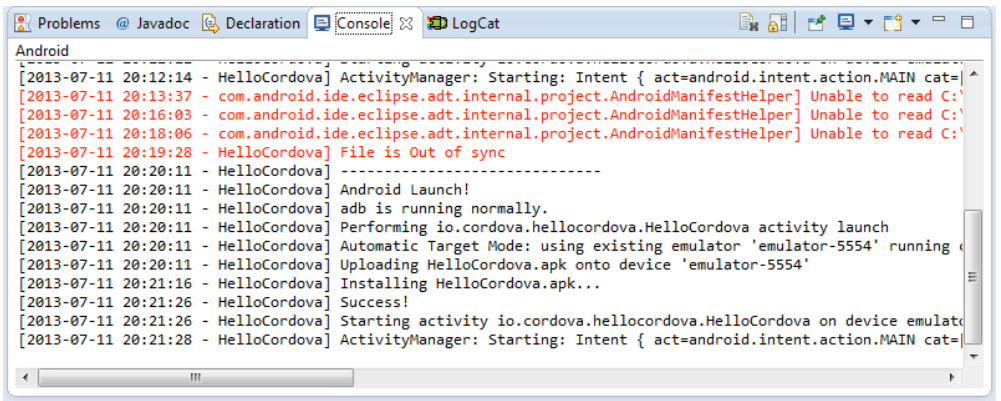



**Figure 7.9** ADT IDE Console Window

LogCat, on the other hand, interacts directly with an Android emulator or physical device (I show you how to debug on a physical device a little later). The LogCat window shows a multitude of messages as the emulator completes its startup process and as you interact with any part of the Android OS. Look to

LogCat for any error messages when you're having trouble with an emulator or device or a running an Android application.

Also, if you remember from Chapter 6, I mentioned you could have your Cordova applications write information to the `console` object as your application runs; on Android, those `console` object messages are displayed within the LogCat window. If you take a look at Figure 7.10, you'll see LogCat displaying the output from the sample application highlighted in Listing 6.2.

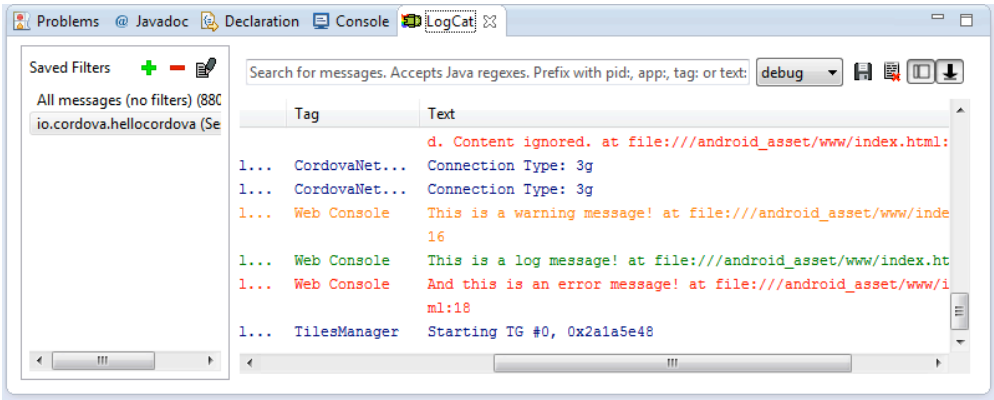



**Figure 7.10** ADT IDE LogCat Window

In the LogCat window, you can filter messages based on their type. Notice the debug button shown on the upper-right corner of Figure 7.10; when you click the button, a drop-down appears, allowing you to select which level of message (verbose, debug, info, warning, and error) are displayed in the view.

# Debugging Outside of the ADT IDE

If you don't want to use the ADT IDE, you can run the ADT debugging tools outside of the IDE. The standalone version of LogCat and associated tools can be found in the Android Debug Monitor (ADM) utility, which is started by navigating to the Android SDK's Tools folder and executing the monitor.bat (on Windows) or monitor (on Macintosh OS) application.

When the ADM launches, it displays a window similar to the one shown in Figure 7.11. I'm not going to cover all of the options available here, but you can see that there's a lot of information available. The lower half of the application's window displays the same LogCat window from the ADT IDE. In the figure, you can see that I've connected the ADM to an Android device emulator; I show you how to connect to a physical device a little later.

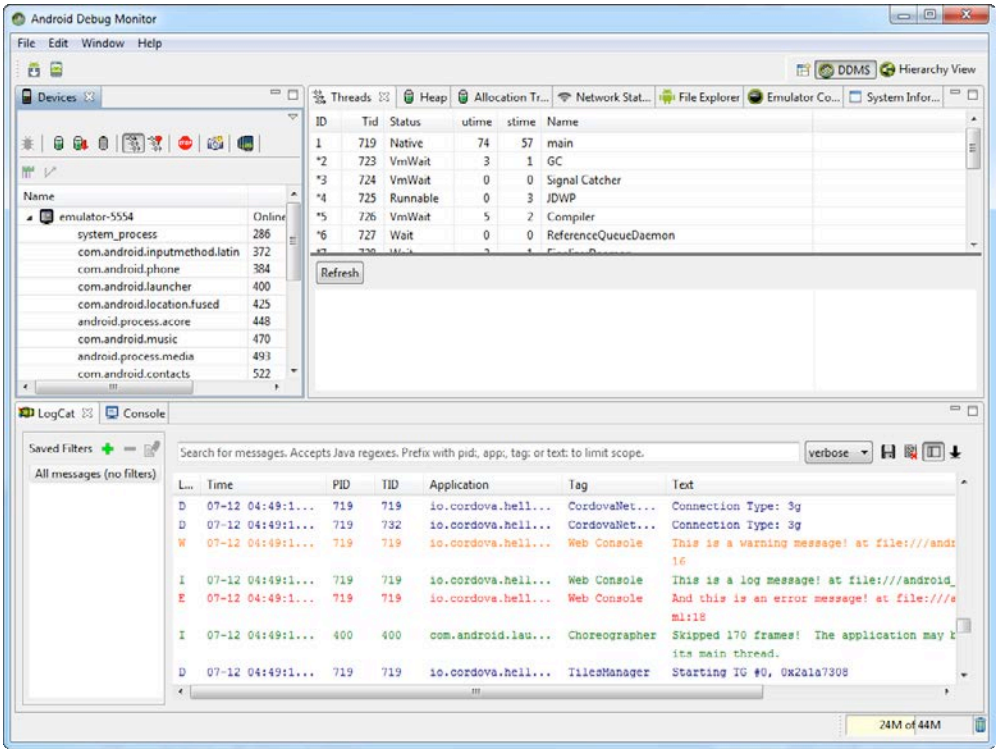



**Figure 7.11** Android Debug Monitor Application

You should spend some time poking around within the ADM—there's a lot of power in what it provides developers.

# Grabbing a Screenshot

While not critical for developers, it's sometimes useful to be able to pull a screenshot off of an emulator or physical device. If you take a look at Figure 7.11, within the Device area in the upper-left corner of the ADM, you will see a little camera icon. With a device connected, you can click that button and grab a screen capture from the connected device. When you click the button, the ADM displays a screen similar to the one shown in Figure 7.12 (I've cropped the image to reduce the amount of space used for the figure). In this example, you're seeing a portion of the screen from my Google Nexus 7 tablet.

You can use the buttons along the top of the screen to refresh, rotate, save and copy the image. Using this feature is a quick and simple way to grab a screen image for documentation or support purposes and frees you from having to perform the screen capture on the device and transfer it to your PC.

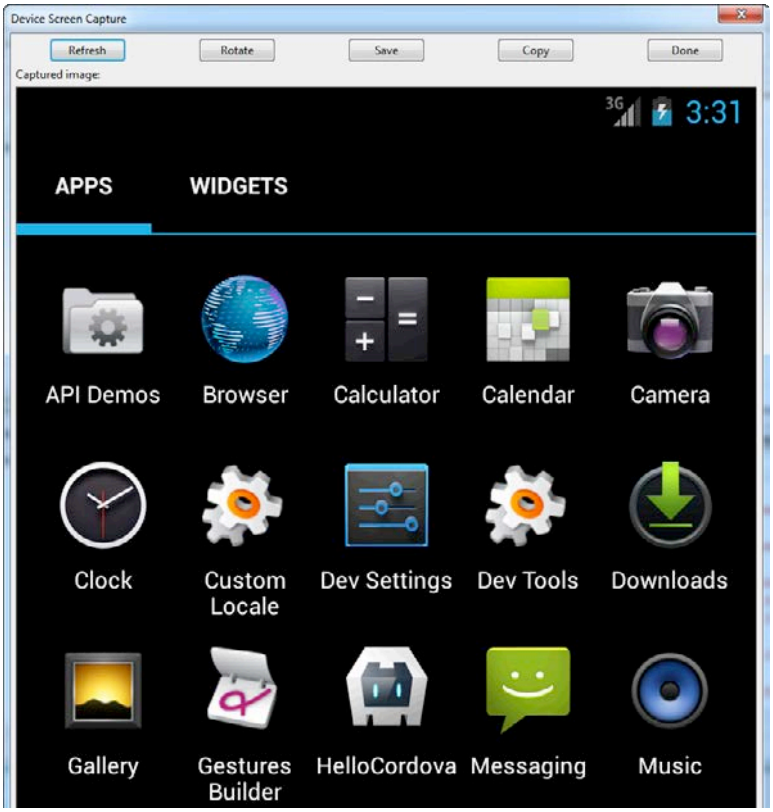



**Figure 7.12** Android Emulator Screenshot

## Debugging on a Physical Device

The ADM can also interact with a physical device just as it can with a device emulator. Initial testing of an application can be done on an emulator, but before any application is released, it should be tested on a representative sample of the physical devices the application is expected to run on.

To use a physical device with ADM, you must connect the device to the computer system running ADM using a USB cable. Before you can do that, you must enable USB debugging on the device. The way you enable this setting varies depending on which type of Android device you are using. On some devices, enabling developer options is accomplished by simply opening the on-device Android settings application and making some selections. On other devices, it's a little harder. You're just going to have to figure this one out on your own, sorry.

### Warning

In order to be able to test on a live device, your computer system must be able to recognize an Android device when it's connected to the system via a USB cable. If you connect a device and it's not recognized by the system, you must resolve any connectivity issues before continuing.

In my testing on Windows, I had to manually install the Google USB driver from the SDK Manager and then, in Windows Device Manager, force the installation of the right driver. You can read about the steps at www.johnwargo.com/index.php/mobile-development/nexus-7-setup-on-windows1.html.

For Samsung devices, you may need to download and install the USB drivers separately before the device is picked up correctly by Windows. See http://developer.samsung.com/android/tools-sdks/Samsung-Andorid-USB-Driver-for-Windows for additional information.

For a Google Nexus 7 tablet, for example, you have to take some special steps before you can even access the developer settings. To enable developer mode on the Nexus 7, you have to open the settings application and select About Tablet (it is the very last item on the list of options on the Settings page). On the About Tablet page, scroll down until you see the Build number item in the list. Tap on the Build number item seven times to enable the developer options on the device. As you tap, Android will pop up a little window indicating how many more times you need to tap to enable developer mode.

With that process completed, when you open the Settings application, you should be able to scroll down to the bottom of the list of options and see a Developer options option available, as shown in Figure 7.13.

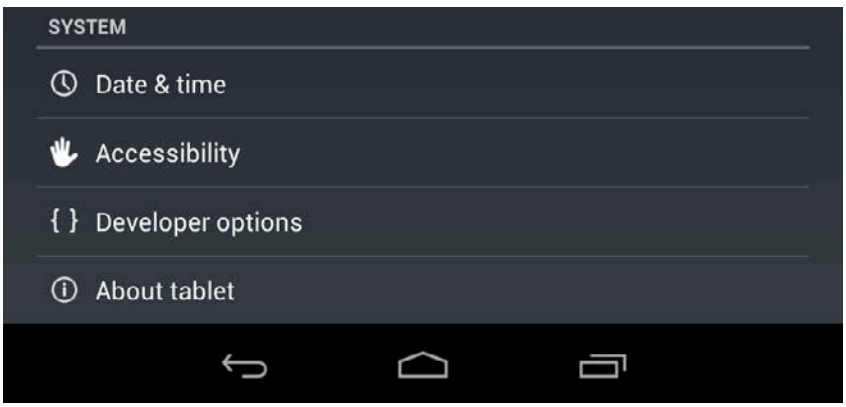



**Figure 7.13** Developer Options Enabled in the Android Settings Application

Click on Developer options and you will see a screen similar to the one shown in Figure 7.14; from here you can enable USB debugging, as shown at the bottom of the figure.

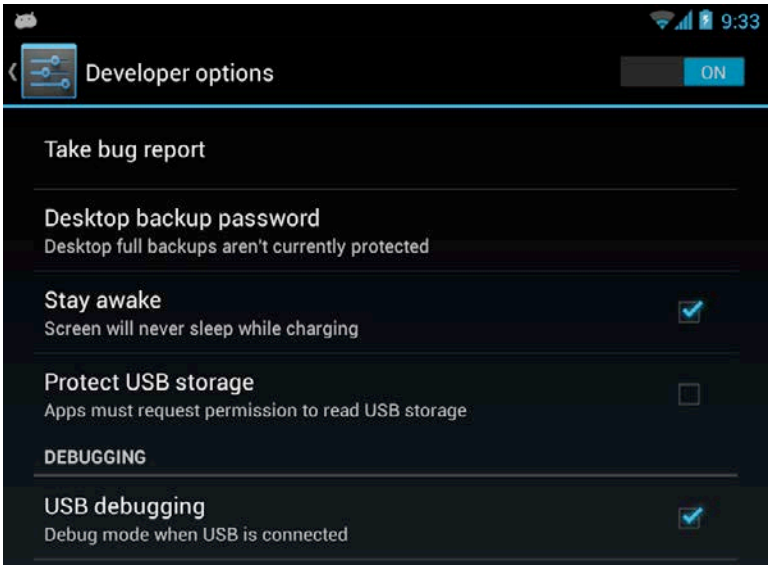



**Figure 7.14** Enabling USB Debugging

With USB debugging enabled, launch the ADM, then connect the Android device to the computer system using a USB cable. After the necessary device drivers initialize (see the preceding warning for information about how to deal with driver issues), ADM should connect to your device and show the connected device in its list of connected devices, as shown in Figure 7.15.

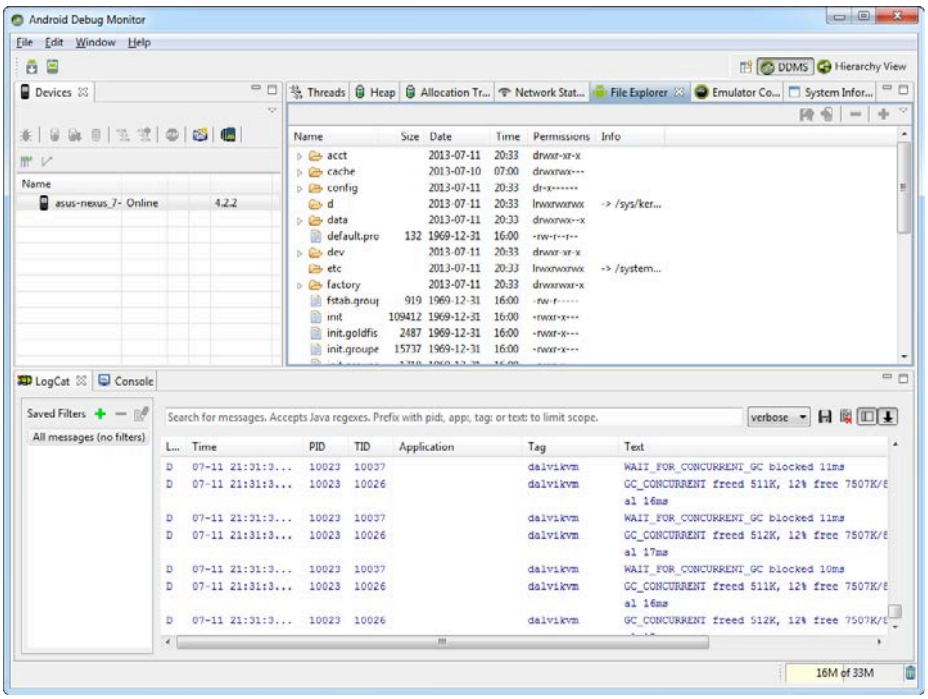



**Figure 7.15** Android Debug Monitor

In ADM, you have access to many of the same capabilities with a physical device as you do with a device emulator, but there are differences. Refer to the Android documentation for more information about the capabilities of ADM with a physical device. ADM is a good way to pull files from or push files to a device for testing purposes. However, much of the file structure is protected, so you won't be able to access all file locations.

## Wrap-Up

In this chapter, I showed you how to use the free tools available from Google that help simplify Android development for Cordova. Although the Android tools do not directly assist with debugging Cordova applications, by using these tools, you can more easily determine what's going on within your Cordova applications.

# 8

# BlackBerry 10 Development with Cordova

Of all of the platforms that Cordova supports, the BlackBerry 10 platform most closely aligns with the Cordova approach. The hybrid application approach has been a core application type for BlackBerry for quite some time, and for BlackBerry devices, Cordova is built on top of it. In this chapter, I show you how to use the BlackBerry command-line tools to test and debug BlackBerry Cordova applications.

One of the cool things about BlackBerry development is that the BlackBerry SDK automatically integrates a web inspector like weinre into the simulator; I show you how to use it here.

The Cordova team announced that it is dropping support for legacy BlackBerry devices (devices running BlackBerry Device Software 7 and earlier), so I cover only BlackBerry 10 here. To work with legacy BlackBerry devices, you need to use Cordova version 2.9 or earlier.

## Configuring Your Environment for BlackBerry Development

Before you can get started with BlackBerry development for Cordova, you must first configure your development environment. I showed you how to install the BlackBerry Native SDK (NDK) in Chapter 3, but I omitted some BlackBerry-specific configuration details from that chapter; I describe them here.

As a reminder, before you can build BlackBerry applications that will run on BlackBerry devices, you must first obtain and install a set of free signing keys from BlackBerry. Open your browser of choice and point it to https://www.blackberry.com/SignedKeys to request a set of keys. You eventually will receive a series of emails from BlackBerry, extract the key files to a location on your system's hard drive, and follow the instructions in the emails to activate your keys.

You need to register the keys using the instructions found at https://developer.blackberry.com/html5/documentation/signing_setup_bb10_apps_2008396_11.html. You can register the keys using the Momentics IDE included with the SDK installation, or you can use the command line by issuing the following command:

```
blackberry-signer -register -csjpin <csj pin>
  -storepass <KeystorePassword> <client-RDK-xxxxxx.csj file>
  <client-PBDT-xxxxx.csj file>
```

In this example, you replace the items in brackets with the following information:

- `<csjpin>`: The PIN you provided BlackBerry when you requested the signing keys.
- `<KeystorePassword>`: The password you want used to secure the keystore created during this process. This is the password you will use to unlock your signing keys whenever you sign a BlackBerry application.
- `<client-RDK-xxxxxx.csj file>`: The path pointing to the client-RDK-xxxxxx.csj file you received from BlackBerry.
- `<client-PBDT-xxxxx.csj file>`: The path pointing to the client-PBDT-xxxxx.csj file you received from BlackBerry.

For the NDK, the files you need are located in the BlackBerry NDK installation folder. For my development system, I installed the NDK in c:\bbndk, so the blackberry-signer file is located in C:\bbndk\host_10_1_0_132\win32\x86\usr\bin. So, I opened a terminal window and navigated to that folder, then issued the following command:

```
blackberry-signer -register -csjpin 987456321
  -storepass my_keystore_password c:\dev\keys\client-RDK-123456.csj
  client-PBDT-12345.csj
```

If this works correctly, the system should respond with the following:

```
Info: CSK file created.
Info: Developer certificate created in
  C:\Users\jwargo\AppData\Local\Research In Motion\author.p12
Info: Successfully registered with server.
Info: Successfully registered with server.
```

Additional steps must be completed before you can test Cordova applications on a physical BlackBerry 10 device, but I cover those later in the chapter.

As shown in Chapter 3, “Installing the Cordova Command-Line Interface,” you can add support for a particular mobile device platform to a Cordova project using the following command:

```
cordova platform add platform_name
```

When adding support for BlackBerry to an existing project, you would use the following:

```
cordova platform add blackberry10
```

However, when you run the command, you may receive an error message similar to the following:

```
blackberry-nativepackager cannot be found on the path. Aborting.
```

This happens because one or more of the tools you need to add BlackBerry support to a Cordova project or to use the BlackBerry native SDK cannot be found on the system path. Before you can use any of the BlackBerry 10 NDK command-line tools, you must first add some BlackBerry 10 NDK configuration information to your environment’s configuration. The easiest way to do this is to execute the `bbndk-env` command that’s included with the NDK. The command updates your system path with some additional folders and sets several environment variables the BlackBerry 10 NDK needs to find its files, as shown in Figure 8.1.

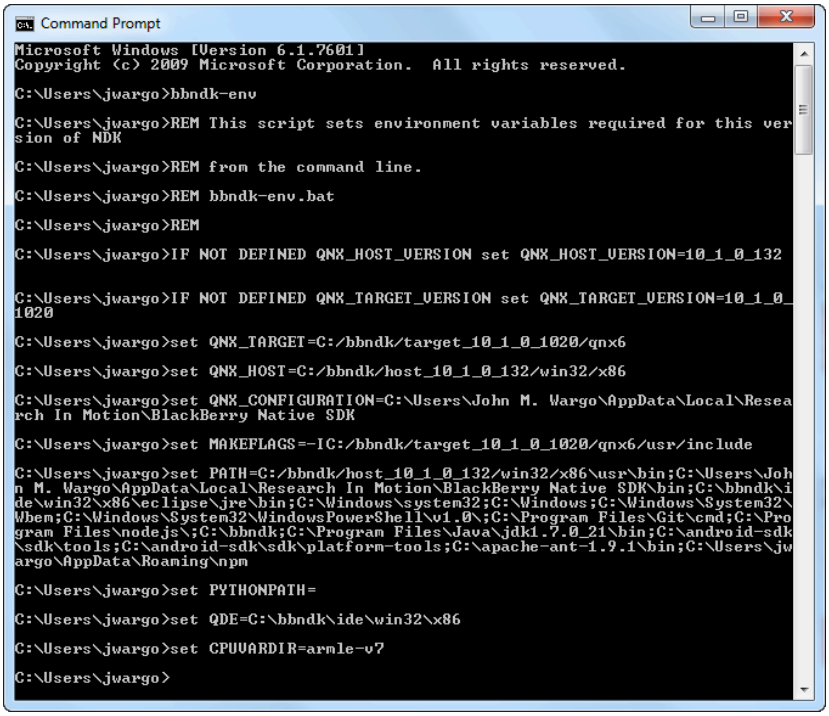



**Figure 8.1** Setting Up the BlackBerry 10 NDK Environment

On Windows, I simply added the BlackBerry 10 NDK installation folder (c:\bbndk\) to the Windows system path. Then, whenever I open a terminal window to work on a BlackBerry project, I simply execute the bbndk-env.bat file before issuing any commands that affect the BlackBerry 10 portion of a project. You could automate this by adding the call to the batch file to the system's autoexec.bat file, but I didn't take the time to do this.

On Macintosh OS, you need to run a shell script called bbndk-env.sh to set up the BlackBerry 10 development environment. You can add the BlackBerry 10 NDK installation path to the system path, then execute the script manually as needed, or you can configure your system to execute the script when opening new terminal windows.

Now when you add the BlackBerry 10 platform to your Cordova project, the process should run properly and not return any errors to the terminal window.

# Configuring a BlackBerry Cordova Project

A BlackBerry Cordova project is configured through the BlackBerry 10 project's config.xml file, shown in Listing 8.1. A BlackBerry hybrid application conforms to the WWC Widget Specification (www.w3.org/TR/widgets), and the config.xml is the standard file used to configure a widget. To access a BlackBerry project's config.xml file, navigate to the Cordova project's platforms\blackberry\www folder and open the file with your XML or text editor of choice.

I'm not going to describe all of the possible settings for the BlackBerry config.xml; you can find detailed documentation from BlackBerry at https://developer.blackberry.com/html5/documentation/config_doc_elements.html. I do, however, describe some of the options, but first take a look at the file in Listing 8.1.

**Listing 8.1 BlackBerry 10 Project Config.XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Licensed to the Apache Software Foundation (ASF) under one
  or more contributor license agreements.  See the NOTICE file
  distributed with this work for additional information
  regarding copyright ownership.  The ASF licenses this file
  to you under the Apache License, Version 2.0 (the
  "License"); you may not use this file except in compliance
  with the License.  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

  Unless required by applicable law or agreed to in writing,
  software distributed under the License is distributed on an
  "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
  KIND, either express or implied.  See the License for the
  specific language governing permissions and limitations
  under the License.
-->
<!--
  Widget Configuration Reference:
    http://docs.blackberry.com/en/developers/deliverables/15274/
-->

<widget xmlns="http://www.w3.org/ns/widgets"
  xmlns:rim=http://www.blackberry.com/ns/widgets
  version="1.0.0.1" id="default.app.id">

  <name>WebWorks Application</name>
  <author>John M. Wargo</author>
  <description>
       A sample Apache Cordova application that responds to the
       deviceready event.
  </description>
  <license href="http://opensource.org/licenses/alphabetical">
  </license>

  <!-- Expose access to all URIs, including the file and http protocols -->
  <access subdomains="true" uri="file:///store/home" />
  <access subdomains="true" uri="file:///SDCard" />
  <access subdomains="true" uri="*" />

  <icon src="res/icon/blackberry/icon-80.png" />
  <rim:splash src="res/screen/blackberry/splash-1280x768.png" />
  <rim:splash src="res/screen/blackberry/splash-720x720.png" />
  <rim:splash src="res/screen/blackberry/splash-768x1280.png" />

  <content src="index.html" />

  <rim:permissions>
```

```
    <rim:permit>use_camera</rim:permit>
    <rim:permit>read_device_identifying_information</rim:permit>
    <rim:permit>access_shared</rim:permit>
    <rim:permit>read_geolocation</rim:permit>
    <rim:permit>record_audio</rim:permit>
    <rim:permit>access_pimdomain_contacts</rim:permit>
  </rim:permissions>

</widget>
```

The config.xml `name`, `author`, and `description` elements describe the application to the BlackBerry user. The `name` element defines the name that appears on the device home screen and the application list. `Author` and `description` are not critical but are useful to set. The `author` is displayed in application security prompts (you can see an example in Figure 8.4), and the `description` should appear in the application details screen within device options.

The `content` element defines the name of the web content file that is launched when the container starts. This value is set to index.html by default, but you can change it to any valid `src` reference. If your web application's main page is in a file called rabbit.html, you would set the `src` value for content to rabbit.html, as in the following example:

```
<content src="rabbit.html" />
```

The most important aspect of the config.xml is the `rim:permissions` element; it is used to list the different application permissions the application requires. If your application uses a restricted capability and you don't have it listed as a `rim:permit` element, then the part of the application that leverages that capability will not function. Your application's not going to tell you what's wrong—it just won't work correctly.

There's a lot you can do with the settings in the config.xml, but it's way beyond the scope of this book to cover all options. Refer to the BlackBerry developer documentation for more information on this topic.

# Defining BlackBerry 10 Targets

Before you can test and debug a BlackBerry 10 Cordova application using a device simulator or a physical device, you must first configure your Cordova development environment for each of the devices you will use (whether they are a simulator or a physical device); each one is referred to as a target. Fortunately, the targets you define are global to your Cordova configuration, so you only have to create them once, and you can use them for any Cordova project you work with.

If you take a look at the .cordova folder in your default user's folder, you will see a file called blackberry10.json. This file contains the global list of targets defined for your Cordova development environment. Initially, it points to an empty list of target objects:

```
{
  "targets": {}
}
```

As you'll see in the sections that follow, as you add targets to the system, this file gets updated with the appropriate settings for each. The Cordova CLI `emulate` and `run` commands use this information to locate the appropriate targets when executing Cordova applications.

The files you need to create BlackBerry targets are installed into your Cordova project when you add support for BlackBerry 10. The sections that follow illustrate how to define targets for the BlackBerry 10 simulator as well as a physical BlackBerry 10 device.

## Defining a BlackBerry 10 Simulator Target

To define a BlackBerry 10 simulator target, first launch the appropriate device simulator, then navigate to the project's `platforms/blackberry10/cordova` folder and execute the following command:

```
target add target_name ip_addr -t simulator
```

As you can see, the process needs to know the IP address of the simulator; fortunately, the simulator displays its IP address at the bottom of the simulator window, as shown in Figure 8.2.

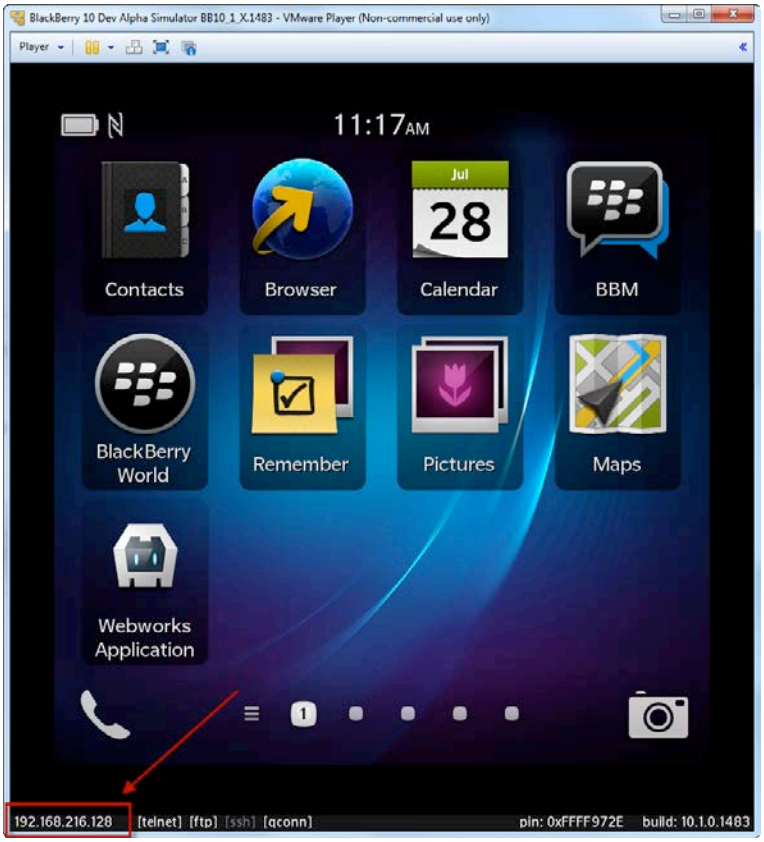



**Figure 8.2** BlackBerry 10 Device Simulator

Assuming the simulator's IP address is 192.168.215.128 and the Cordova project was created on Windows in c:\users\jwargo\dev\test\, you would open a terminal window, then navigate to c:\users\jwargo\dev\test\platforms\blackberry10\cordova\ and issue the following command:

```
target add bb10-sim 192.168.216.128 -t simulator
```

After the completion of that command, the blackberry10.json file is updated to include the settings for the target:

```
{
  "targets": {
    "z10-sim": {
      "ip": "192.168.216.128",
      "type": "simulator"
    }
  },
  "defaultTarget": "z10-sim"
}
```

When you start the BlackBerry 10 device simulator, you can select which device type the simulator simulates. You can define separate targets for each simulator type and define the default simulator to use via the `defaultTarget` variable defined in the blackberry10.json file.

## Defining a BlackBerry 10 Device Target

To define a BlackBerry 10 target using a physical device, the process is similar to what was shown in the previous section, but additional setup steps must be completed before you can create the target.

To test and debug your BlackBerry 10 Cordova applications, you must first enable debug Development Mode on the device. To do this, open the device's Settings application, then open the Security and Privacy section of the application. At the bottom of the list of options displayed, select the Development Mode option. The device will open a screen similar to the one shown in Figure 8.3. At the top of the screen, set the Use Development Mode option to On, as shown in the figure. You may also want to set the IP address for the device. If you will be testing on multiple devices, you need to make sure that the IP addresses are unique across devices and define a separate target for each device.

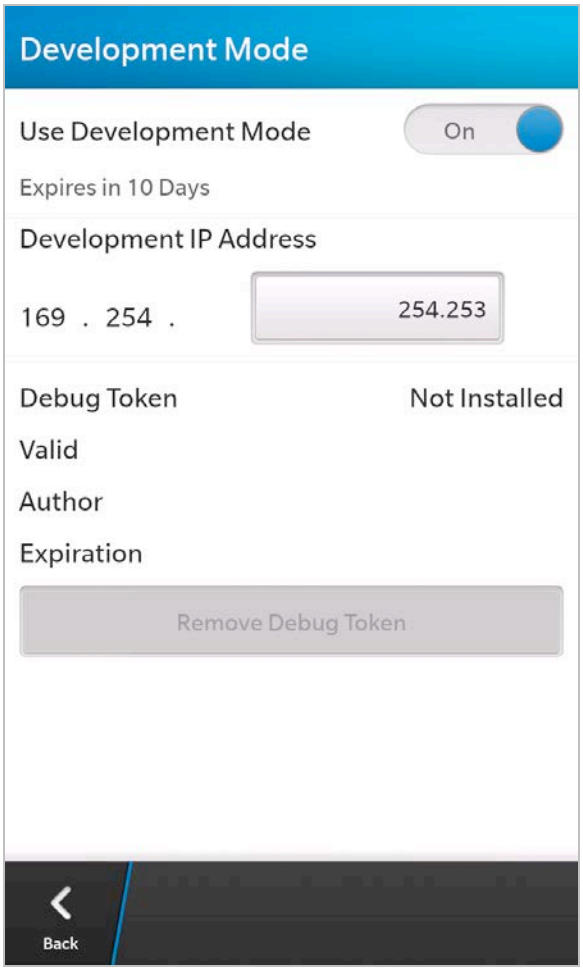



**Figure 8.3** Enabling Development Mode Options on a BlackBerry Device

With those settings in place and the device physically connected to the computer system using a USB cable, you can now create the target. The settings for a physical device are a little different as the current version of the BlackBerry command line tools requires the device password as well as the device PIN. In Cordova 3.1, the BlackBerry development team will be implementing PIN detection to their command line tools.

Assuming the Cordova project was created on Windows in `c:\users\jwargo\dev\test\`, you would open a terminal window then navigate to `c:\users\jwargo\dev\test\platforms\blackberry10\cordova\` and issue the following command:

```
target add target_name ip_address -t device -p device_password -pin device_pin
```

For my particular configuration, the command is

```
target add z10-device 169.254.254.253 -t device -p abcdef -pin 2AB09876
```

which adds settings for the device to the system's blackberry10.json file, as shown here:

```
{
  "targets": {
    "z10-sim": {
      "ip": "192.168.216.128",
      "type": "simulator"
    },
    "z10-device": {
      "ip": "169.254.254.253",
      "type": "device",
      "password": "abcdef",
      "pin": "2AB09876"
    }
  },
  "defaultTarget": "z10-sim"
}
```

To change the configuration so the physical device is the default target, issue the following command in the terminal window:

```
target default z10-device
```

To see a list of the defined targets on the system, issue the following command:

```
target
```

The system will respond with the list of targets, as shown here:

```
  z10-sim
* z10-device
```

The asterisk marks the default target.

# Debugging on a Device Simulator

Cordova applications for BlackBerry don't use the BlackBerry Momentics IDE for managing applications; instead, everything is done via the Cordova CLI. To run a Cordova application in the BlackBerry 10 simulator, first start the simulator, then navigate to the Cordova project folder and issue the following command:

```
cordova emulate blackberry10
```

The CLI will build the BlackBerry 10 project, load it into the simulator, then launch the application. You can also navigate to the project's platforms/blackberry10/cordova folder and run the application using the run command found in that folder. The simulator must be running before you issue the command—the CLI won't launch it for you. The process of building and deploying the Cordova application takes quite some time, so be patient.

The default Cordova project used in this example enables security settings for all of the available plugins, although none are enabled in this project. Because BlackBerry is a secure platform, when the application first launches, you are prompted to allow access to the different Cordova APIs, as shown in Figure 8.4. These settings are described in "Configuring a BlackBerry Cordova Project," earlier in the chapter. Just click the OK button to continue.

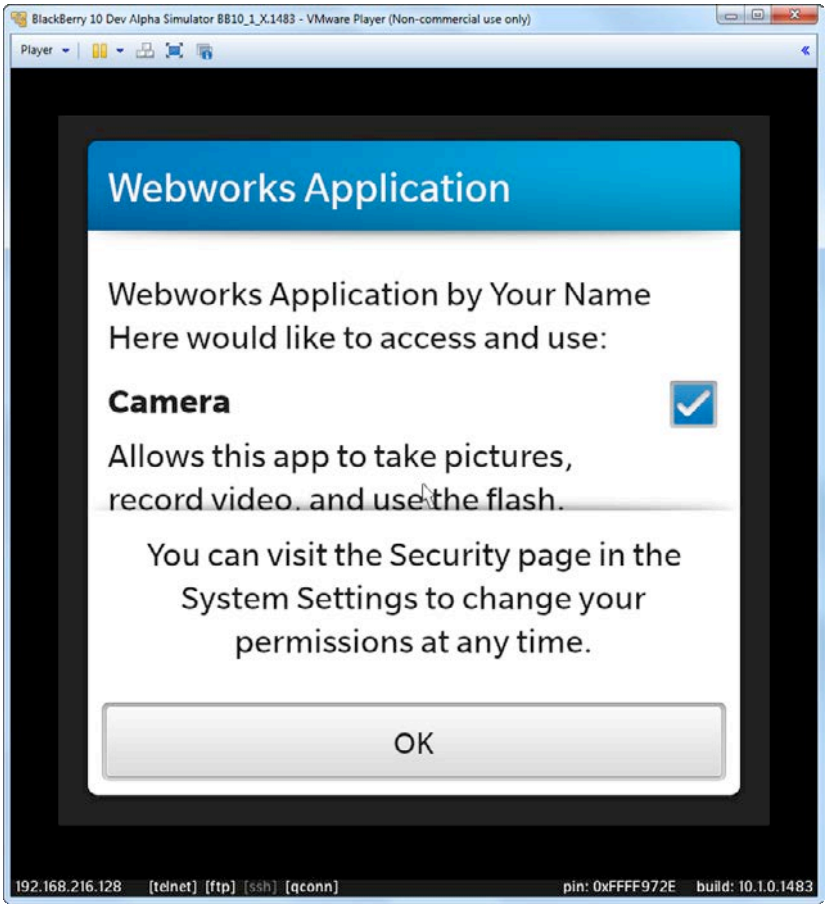



**Figure 8.4** Cordova Application Security Prompt

BlackBerry 10 Cordova projects enable the BlackBerry Web Inspector by default. When a Cordova application launches in the simulator, it lets you know if Web Inspector is enabled, as shown in Figure 8.5.

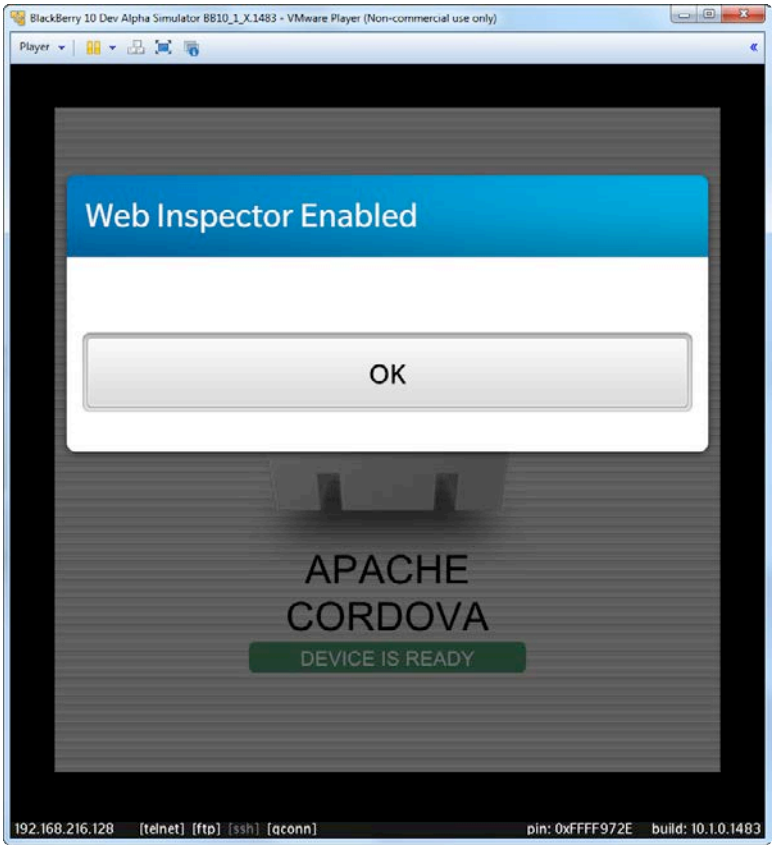



**Figure 8.5** BlackBerry 10 Simulator Web Inspector Warning

To disable the Web Inspector, you must build the application in release mode. You can do this by opening a terminal window, navigating to the Cordova project's platforms\blackberry10\cordova\ folder, and issuing the following command:

```
build --release --keystorepass your_keystore_password
```

I show you how to use the Web Inspector a little later. When you click OK, the application opens and displays its web content, as shown in Figure 8.6.

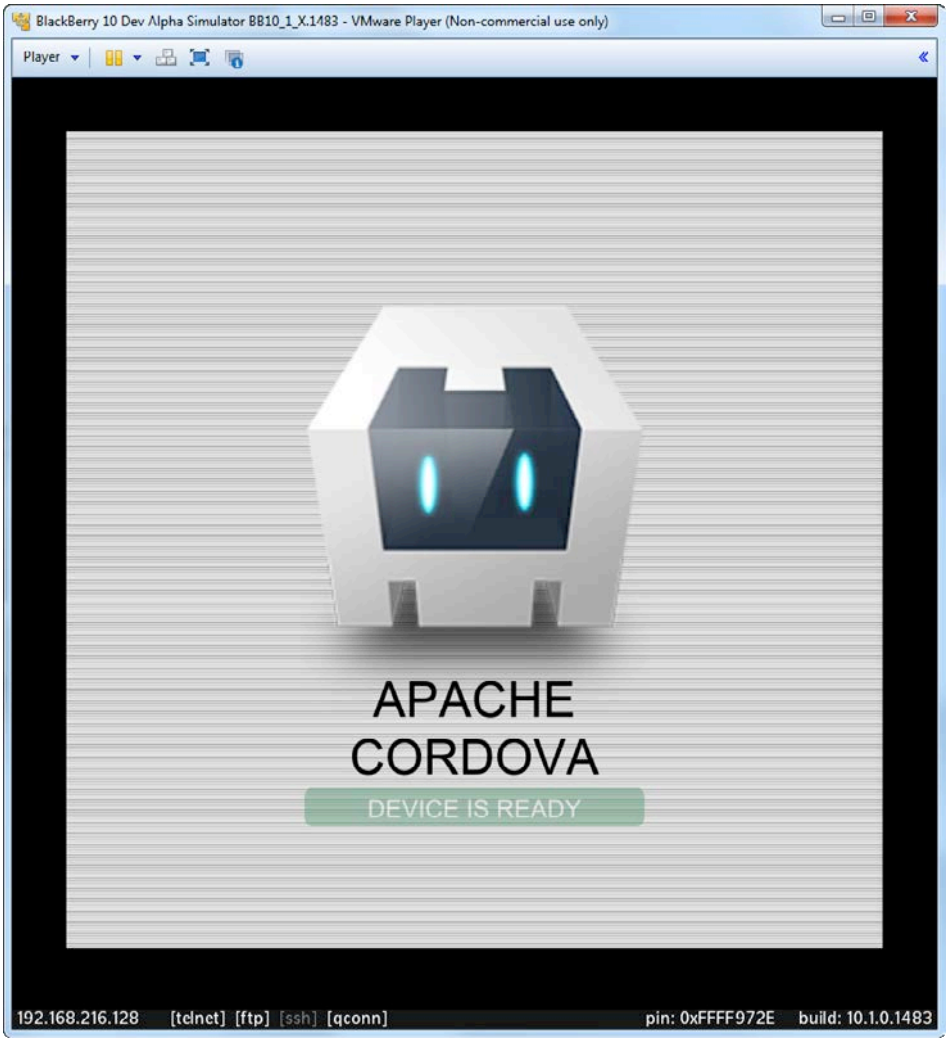



**Figure 8.6** Cordova HelloCordova Application Running in the BlackBerry 10 Simulator

At this point, the application is running on the simulator, and you can interact with the application as you would on a physical device.

> **Note**
>
> For some reason, the BlackBerry 10 device simulator has difficulty rendering the background image in the default Cordova HelloWorld application. I experienced this on two different computers but didn't experience the same issue on a physical device. All the more reason to use the simulator for initial testing, but do all final testing on physical devices.

## Using the BlackBerry Simulator Controller

Some things happen on a physical device that you can't easily map to a simulator; examples are determining the device location, responding to phone calls, orientation changes, and so on. To help developers manage such issues while working with a simulator, the BlackBerry simulator includes a separate controller program you can use to manipulate the simulator as its running. The controller is installed alongside the device simulator as a separate icon; double-click the icon to the program, and it opens in a window similar to the one shown in Figure 8.7.

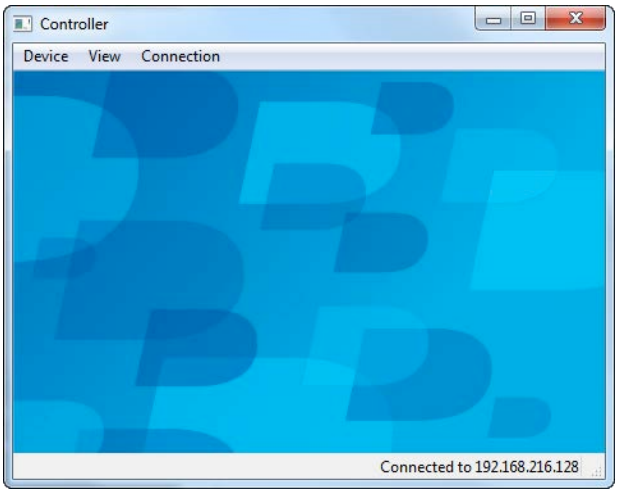



**Figure 8.7** BlackBerry Simulator Controller

The controller automatically connects to a running simulator; otherwise, you can use options in the Connection menu to disconnect and connect manually. Use the Device menu to access the available options for manipulating the simulator. For example, there's an option for setting GPS position; when you select that option, a new window appears, as shown in Figure 8.8.

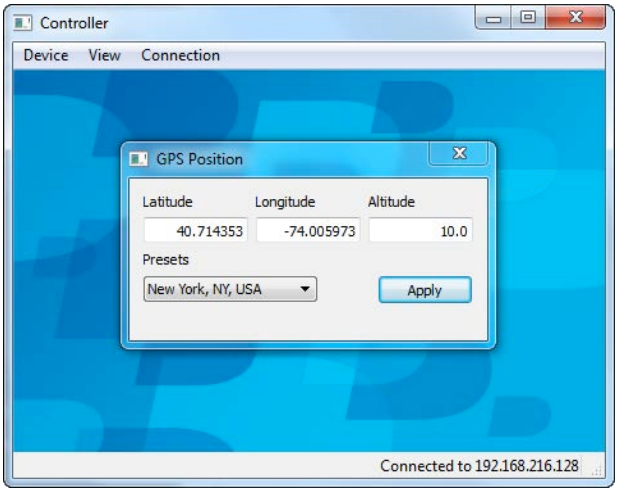



**Figure 8.8** BlackBerry Simulator Controller: GPS Settings

Set the appropriate location information and click the Apply button to apply the settings to the simulator. If your Cordova application that's running in the simulator were watching the simulated device's location, it would automatically register the change and react as you've coded within your application.

The Controller can do a lot more for you as you work with your BlackBerry Cordova applications. Take a few moments and poke around in the Device menu to see what other options are available to you.

## Using the BlackBerry Web Inspector

Since the hybrid application approach is a standard option for BlackBerry development, debugging Cordova applications is much easier than on some other platforms. The built-in tools allow a Cordova developer to more easily understand what's going on within the web application running in the Cordova native container. The Web Inspector provides critical capabilities in this area and works with the device simulator as well as physical devices.

As described previously, by default Web Inspector is enabled for Cordova applications built using the Cordova CLI. When a Cordova application with Web Inspector enabled runs in the BlackBerry 10 Simulator, you can fire up a browser and watch the application as it runs within the container, highlighting HTML elements, evaluating expressions, stepping through code, and more. To demonstrate, I created a simple Cordova application with a little bit of JavaScript code and launched it within the simulator. When the application runs in the simulator, it renders the screen shown in Figure 8.9 (cropped for brevity's sake).

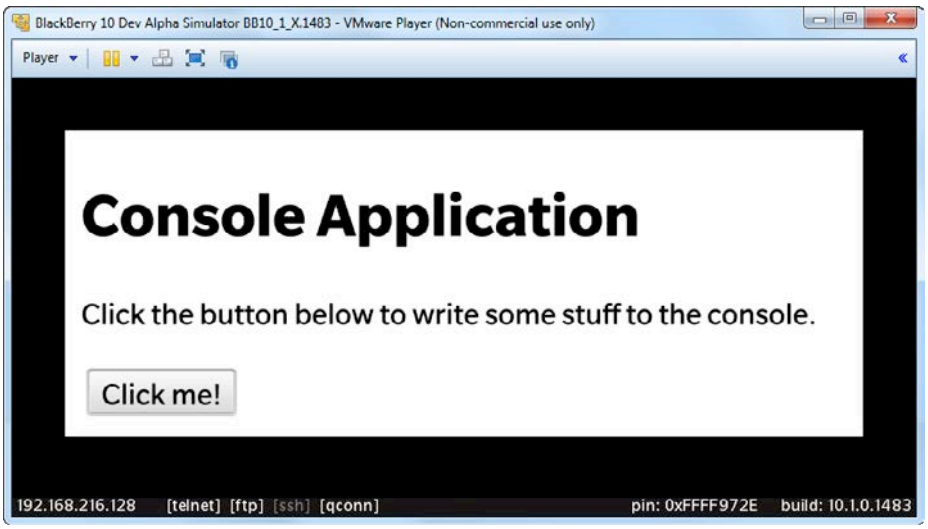



**Figure 8.9** Console Application Used to Demonstrate the BlackBerry Web Inspector

To use the Web Inspector to debug this application, open the Google Chrome browser and enter the following URL:

```
http://device_IP_Address:1337
```

The `device_IP_Address` refers to the IP address of the device simulator or physical device running the application. So, for the simulator shown in the figure, use the following:

```
http://192.168.216.128:1337
```

When you press enter, the browser connects to the browser running on the device and opens a screen similar to the one shown in Figure 8.10. To debug a Cordova application, select the option highlighted in the figure; the web resource at local:///index.html is the start page for the web application running within the Cordova container. The other options shown in the figure are for the browser and browser chrome running on the device.

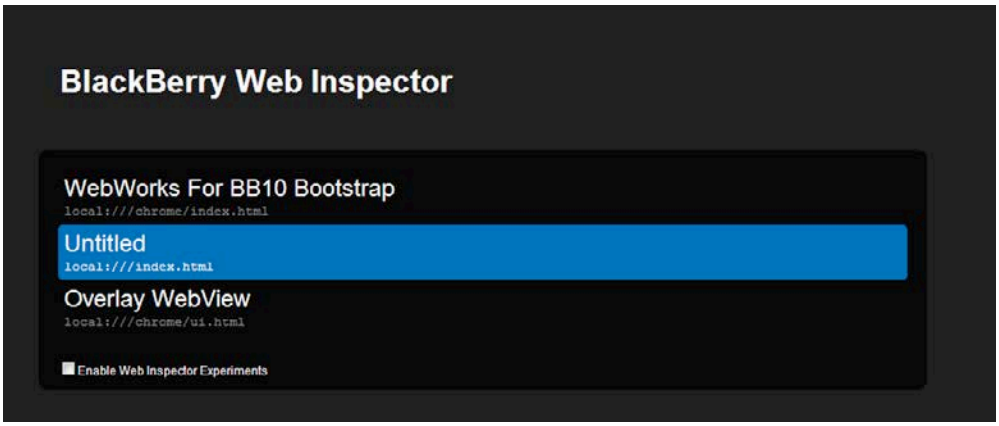



**Figure 8.10** BlackBerry Web Inspector Start Page

After you make that selection, the browser should open a page similar to the one shown in Figure 8.11.

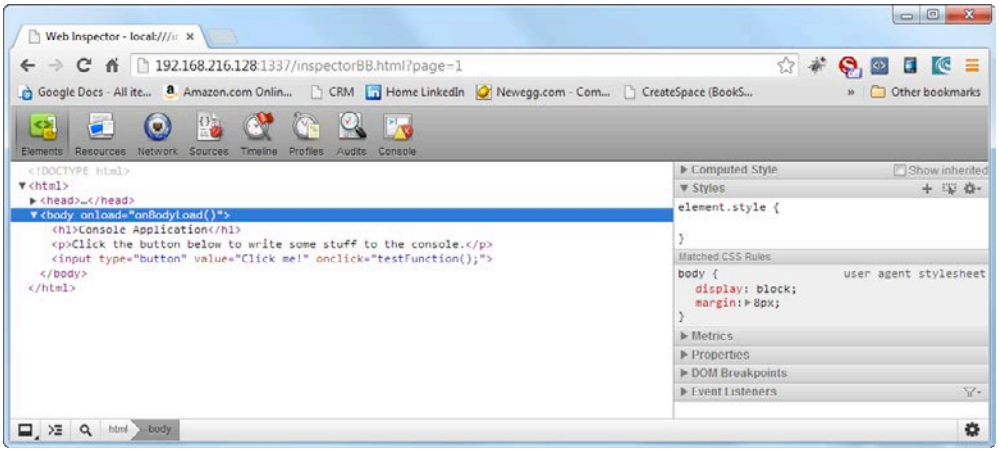



**Figure 8.11** BlackBerry Web Inspector: Elements Page

Using the different options shown across the top of the browser page shown in Figure 8.11, you have access to a tremendous amount of information about the application. For Cordova application debugging, there are a couple of features here that are quite useful.

If you click on the Sources icon (located in the list of icons across the top of the page shown in Figure 8.11), you are presented with the page shown in Figure 8.12. The Web Inspector disables debugging by default, but you can turn it on here, as shown in the figure. Enable the appropriate radio button on the page and click the Enable Debugging button to continue.

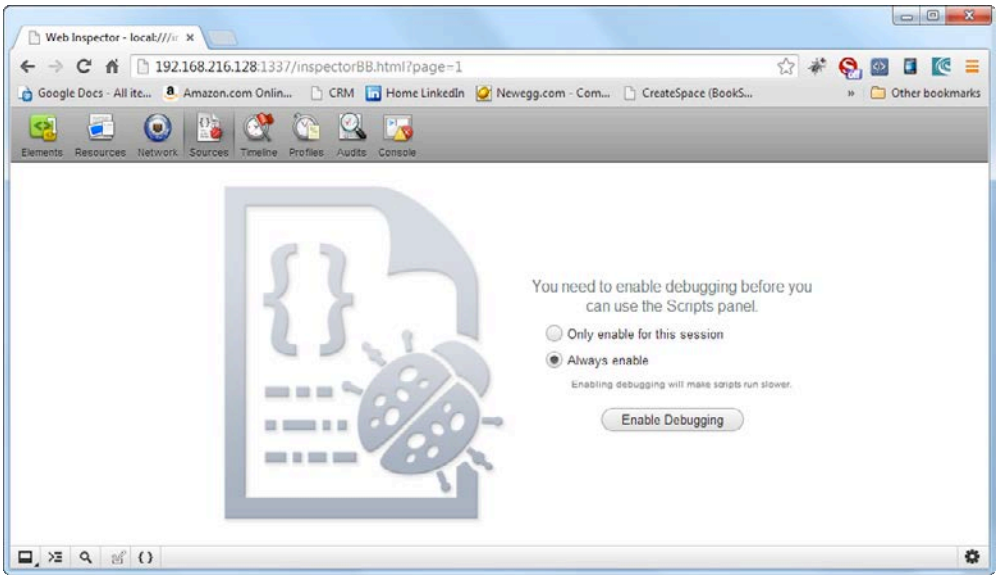



**Figure 8.12** BlackBerry Web Inspector: Enabling JavaScript Debugging

Next, the Web Inspector displays a list of source files for your application; you can select any of them to open a debug session using that source file. In Figure 8.13, I selected the Cordova application's index.html file because, for this simple example, that's where all of my code is located.

In this interface, I can set breakpoints (as I have in line 18, shown in Figure 8.13), define watch expressions, view the call stack, inspect variables, and more.

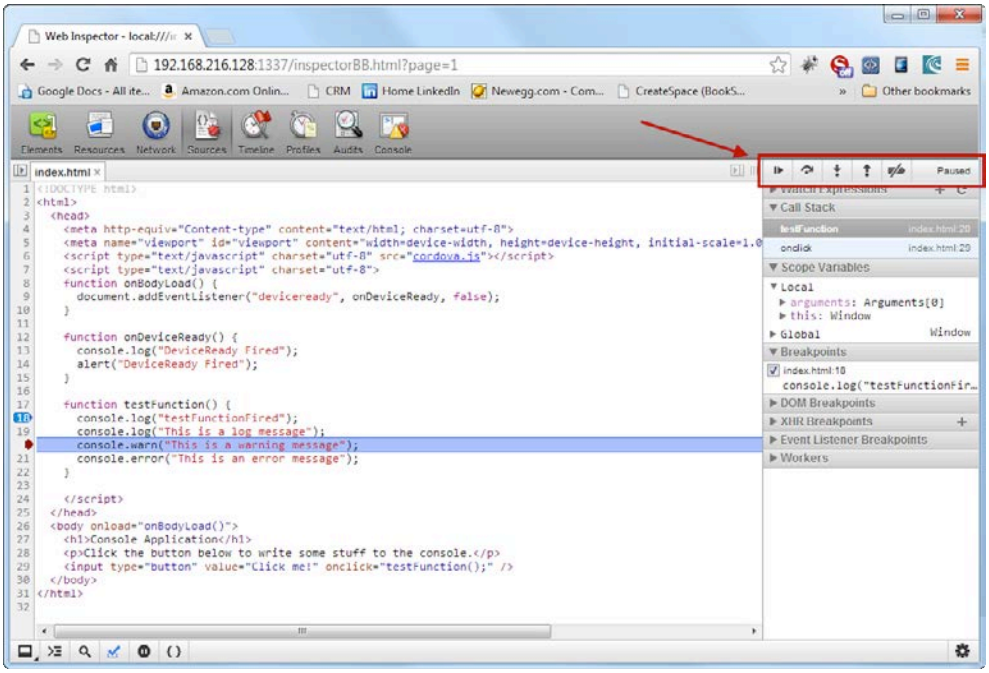



**Figure 8.13** BlackBerry Web Inspector: JavaScript Debugging

In this example, I clicked the Click Me button shown in Figure 8.9 and stepped through a few lines of code. When the Web Inspector hits a breakpoint, you can step through your code in different ways using the buttons highlighted at the top right of Figure 8.13. At this point, the Web Inspector works like any other debugger you've worked with in the past.

To round out this example, Figure 8.14 shows the console page, which displays the output of the code. Here I can view the output of calls to the `console` object. Unlike the Apple debugger, the Web Inspector uses colored icons for warning and error log entries and even color-codes the error output so it can be more easily located during a debugging session.

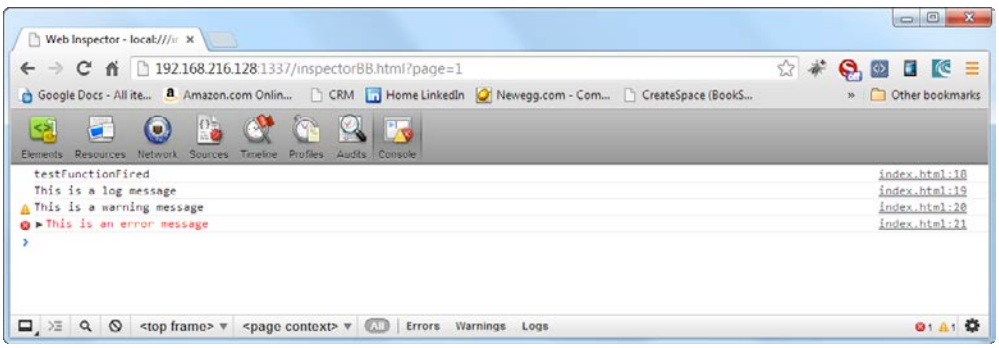



**Figure 8.14** BlackBerry Web Inspector Console Window

As you can see, the BlackBerry Web Inspector is a pretty powerful tool for debugging Cordova applications.

## Debugging on a Physical Device

When working with a physical device, the Web Inspector capabilities are still supported, so the information in the previous section still applies. In general, to run the application on a device, you must connect the device to the computer, then navigate to the Cordova project folder and issue a CLI command. You can connect the device via USB cable, or if the device is on the same network as your development computer, you can connect over Wi-Fi as well.

The first time you run the application on a device, you have to generate a debug token and deploy it to the device. To learn more about Debug Tokens on BlackBerry, refer to https://developer.blackberry.com/html5/documentation/runnning_unsigned_apps_using_a_debug_token_1866987_11.html. So, the first time you run an application on a BlackBerry device, you should use the following command:

```
cordova run blackberry10 -k keystore_password
```

This command instructs the CLI to create the debug token, deploy it to the device, then package the application and deploy it to the device as well. If everything works correctly, after quite a long wait (it really is quite a long wait), you should see the Cordova application running on the device.

On subsequent runs, after the debug token has been installed on the device, you can run the application on a device using following command:

```
cordova run blackberry10
```

If you have multiple targets defined, you can launch the application on a specific target by issuing the following command:

```
cordova run blackberry10 --target target_name
```

To run the application on whichever device is currently connected, issue the following command:

```
cordova run blackberry10 --device
```

## Wrap-Up

This chapter has demonstrated how to use the command-line tools provided by BlackBerry to run and debug Cordova applications on both a device simulator and physical device. The broad spectrum of available tools provided by BlackBerry provides the Cordova developer with some additional capabilities not available on other platforms.

# 9

# iOS Development with Cordova

For developers building Cordova applications for the iOS platform, Apple provides a suite of tools used to design, package, and deploy iOS applications. Even though the Cordova CLI takes care of most of the process of creating, managing, and testing iOS applications, there will be times when you want to have more control over the process. The CLI can launch a Cordova application in the iOS simulator, but when you encounter problems with an application and you want to know more about what's going on, you need to use the development tools that Apple provides.

In this chapter, I show you how to use Xcode, Apple's tool for iOS development, to run and debug Cordova applications for iOS devices. I also show you how to use the Safari browser to debug Cordova applications.

## Working with Xcode

Assuming you've used the Cordova CLI to create a project and added the iOS platform to it, working with a Cordova application in Xcode is pretty straightforward. When you have your web application's content all ready, issue the following CLI command to copy the web application source code over to the iOS project folder:

```
cordova prepare ios
```

Instead of having to import the Cordova project into the IDE as you do for Android and BlackBerry, the Cordova CLI creates a project that can be opened directly in Xcode. Simply start Xcode, open the project, and go to work. When the project opens, you should see a screen similar to the one shown in Figure 9.1. From here, you can run the application in any of the iOS simulators as you would for any other iOS application.

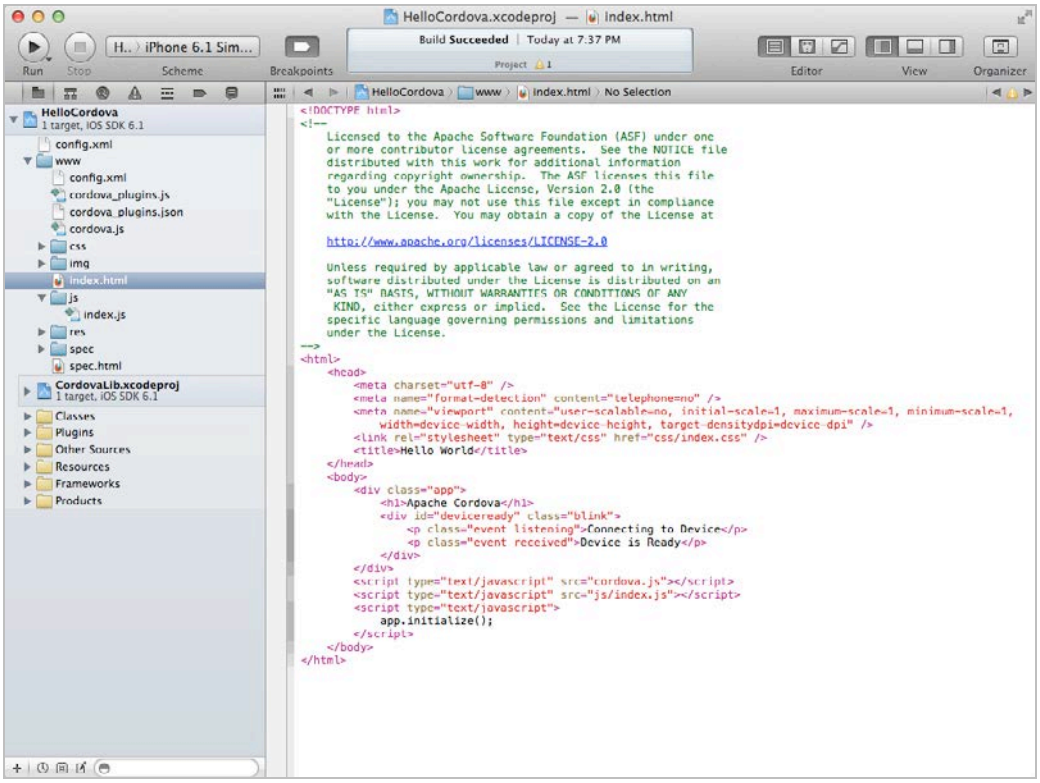



**Figure 9.1** Cordova Project Open in Xcode

# Debugging iOS Applications

iOS applications are written in Objective-C, a variant of C that was used by NeXT to develop applications for the NeXTSTEP operating system back in the 1980s. Xcode lets you edit HTML5 source files, but because iOS applications are written in C, Xcode doesn't have the ability to debug web applications running in the Cordova container. You can use weinre or the `console` object to help debug applications, as described in Chapter 6, "The Mechanics of Cordova Development," to debug your iOS applications.

When you run an iOS application in Xcode, the IDE opens a console window at the bottom of the screen where you can see messages generated by the simulator as well as the Cordova application. Using the application from Listing 6.2, you can see the output from the code that uses the `console` object written to the IDE console screen in Figure 9.2.

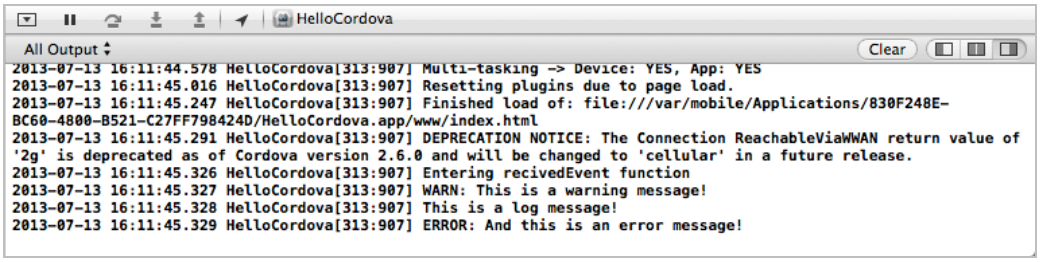



**Figure 9.2** Xcode Console Output Window

Notice that the Warning and Error messages are tagged with additional text so you can identify them more clearly. Unfortunately, Xcode doesn't color-code them as the Android IDE does; perhaps Apple will add more robust capabilities in the future.

Greater debugging capabilities exist when working with a physical device; refer to the next section for additional information.

# Debugging on a Physical Device

There are a few differences between debugging on a simulator and debugging on a physical device, so at some point in your development process, you are going to want to run your app on a device. Apple has some pretty sophisticated processes you must follow before you can run an application on a physical device. As there are hundreds of books and websites dedicated to iOS development, I'm not going to dig into the details here. You can find detailed instructions on the Apple Developer's website at http://tinyurl.com/op9lvv3. You have to follow the instructions provided there before you can work with a physical device.

Once you've completed the setup of your device, you can connect it to the computer running Xcode with a USB cable and run the application on the device directly from Xcode. In this environment, Xcode provides the same debugging capabilities for a physical device as it does for the simulators.

# Using the Safari Web Inspector

Beginning with iOS 6, Apple added some capabilities to Safari (on both the desktop and in iOS) that allow a web application to be remotely debugged in the desktop Safari browser. The cool thing about this capability is that it works with the `UIWebView` used within a Cordova application. You can find detailed information about this remote debugging capability on Apple's web site at http://tinyurl.com/c9yqebs.

The first thing you must do is enable the developer menu in the desktop version of Safari for Macintosh OS (this process doesn't work on Windows and probably never will). Open Safari application preferences and select the Advanced tab, as shown in Figure 9.3. Enable the checkbox at the bottom of the figure labeled "Show Develop menu in menu bar."


**Figure 9.3** Safari Preferences on Macintosh OS X

With that change in place, close preferences, and you should see a new Develop menu option in Safari.

Next, you need to open Settings on your iOS device and select the Safari option, then click the Advanced option at the bottom of the Settings screen. On the screen that appears, enable the Web Inspector option, shown in Figure 9.4.


**Figure 9.4** Safari Advanced Settings on iOS

With those settings in place, launch the simulator or connect the device to the computer with a USB cable, then launch the Cordova application. You can run the Cordova application from Xcode or preload the application on the device and run it manually from the device's home screen.

With the Cordova application running, switch to your desktop computer and fire up the Safari browser (remember, this works only on Macintosh OS X) and open the Develop menu. You should see your simulator or iOS device listed in the menu that appears. Figure 9.5 shows an example: my iPhone, appropriately named "John's iPhone," appears as an option in the menu. You can also see the HelloCordova application running; select the index.html file, as shown in the figure.

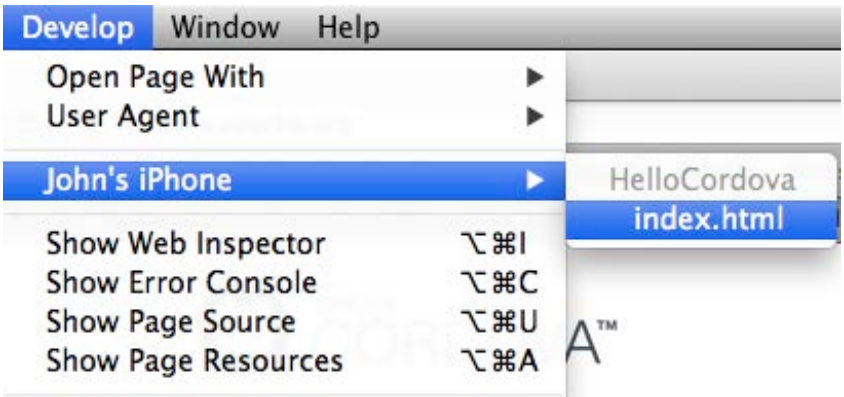



**Figure 9.5** Connecting the Remote Web Inspector to the Cordova Application

Safari connects to the remote application, then opens a new window, shown in Figure 9.6. From this new window, you have access to the code running within the Cordova container and the ability to interact with different parts of the application.

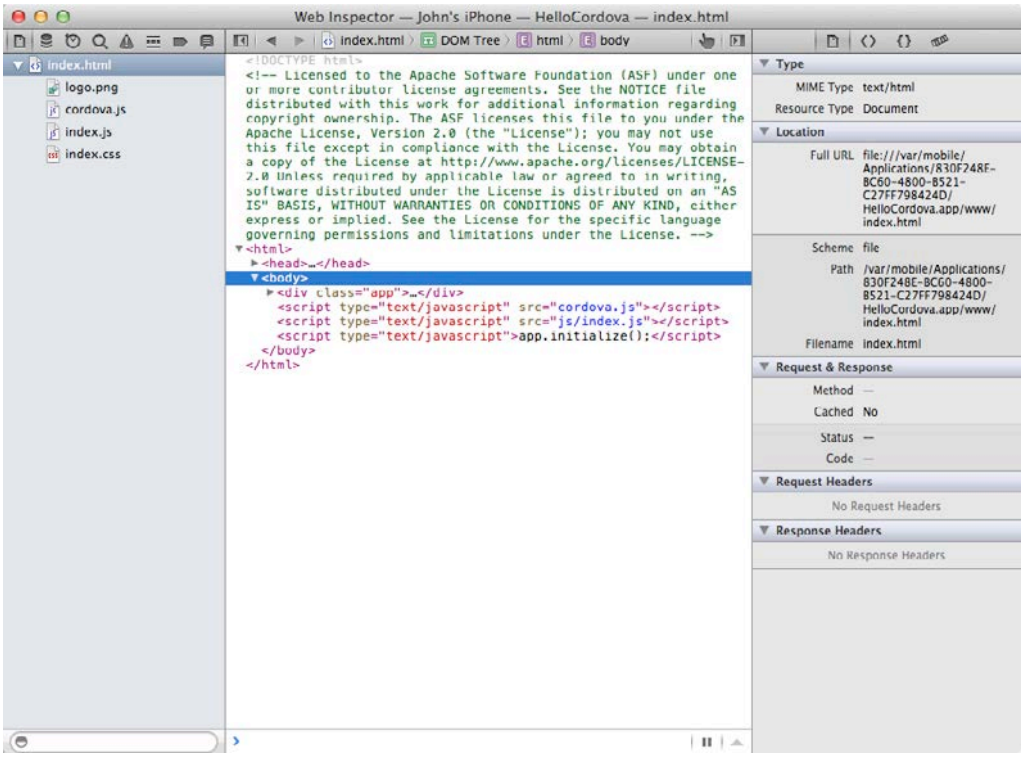



**Figure 9.6** Safari Remote Web Inspector Window

Figure 9.6 shows the contents of the index.html file. You can expand the different parts of the page and even edit the contents on the fly. Right-click on any of the divs within the HTML file, and you can select an option that allows you to edit the content. In Figure 9.7, you can see that I've inserted a new paragraph tag into the file while the application is running.

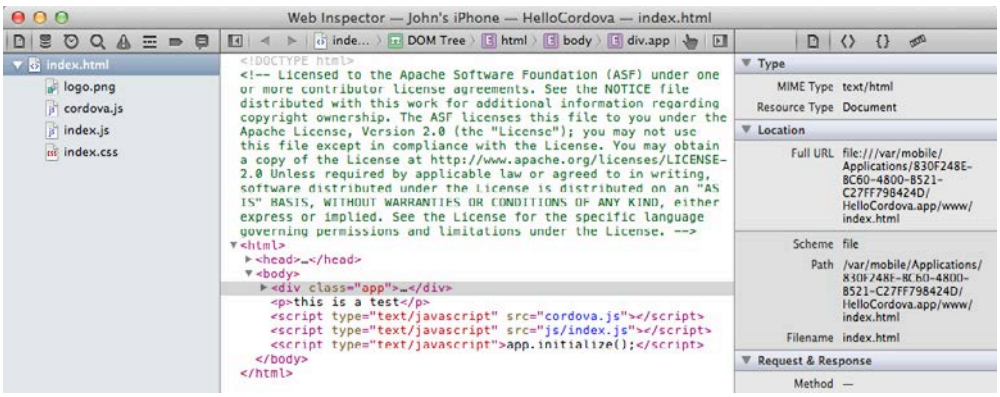



**Figure 9.7** Updating HTML Content in the Web Inspector

In real time, on the device, the Cordova application will update to show the new content, as shown in Figure 9.8.


**Figure 9.8** Modified Cordova Application

Very important to Cordova developers and something that's not available on most other platforms is the ability to set breakpoints and step through the JavaScript code in a Cordova application. Figure 9.9 shows the application's index.js with some breakpoints set.

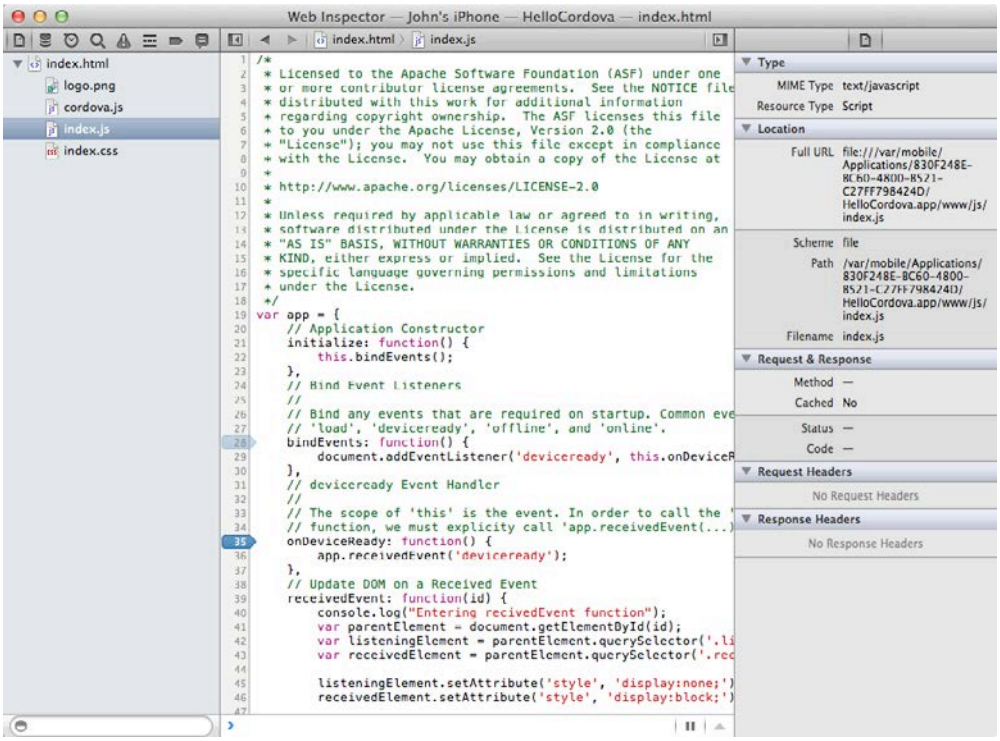



**Figure 9.9** Web Inspector JavaScript Debugger

Figure 9.10 shows an evaluation window you can use to evaluate expressions. When you type any object value or expression and press Enter, the log evaluates the expression and displays the results. In this example, I'm retrieving the value of the `document.title` variable.


**Figure 9.10** Web Inspector Log

Although I'm not quite sure why you would need this ability, you can also execute the JavaScript `alert()` function remotely, which displays the alert on the connected device's screen, as shown in Figure 9.11.

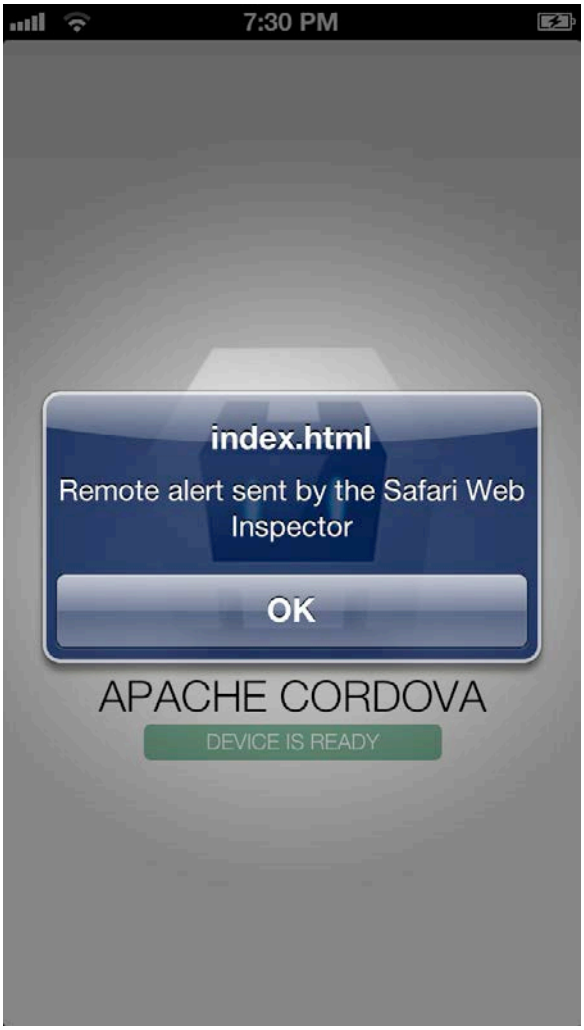



**Figure 9.11** Remote JavaScript Alert Displayed in a Cordova Application

I've only skimmed the surface of what you can do with the Web Inspector. Before you go too far with iOS development for Cordova, spend some time with this tool to understand all it can do for you. You'll likely save time and frustration by using a tool like this for your testing and troubleshooting.

## Wrap-Up

In this chapter I've shown you how to test and debug your iOS Cordova applications in an iOS simulator and on a physical device. Using these tools should help simplify the development process and allow you to more quickly identify issues with your applications.

# 10

# Windows Phone 8 Development with Cordova

Cordova provides good support for the Windows platform, supporting Windows desktop, Windows Phone 7, and Windows Phone 8. In this chapter, I show you how to use the Microsoft development tools to build and test Cordova applications for Windows Phone 8. The developer tools for Windows Phone are free and pretty easy to use; I found that for the different platforms highlighted in the book, developing for Windows Phone 8 was the easiest.

## Getting Started with Windows Phone Development

To develop, test, and deploy Windows Phone 8 applications, you need to download and install a copy of Visual Studio 2012. There's a free version for Windows Phone development called Visual Studio Express 2012 for Windows Phone, which can be downloaded from the Microsoft website at http://developer.windowsphone.com.

You can use the Windows Phone SDK version 8 for Windows Phone 8 and Windows Phone 7.5 development. The Windows 8 SDK requires a 64-bit version of Windows 8 Professional and at least 4 GB of RAM; I wouldn't try to do Windows Phone development in a virtual machine (VM). Older versions of the SDK can be used for Windows Phone 7.5 and 7.1 applications; you'll be able to run those development tools on a Windows 7 or Windows 8 system, and they don't have a 64-bit requirement.

> **Warning**
>
> Pay special attention to the hardware and software requirements for Visual Studio, as the use of the device emulators puts quite a strain on your system, and if you don't have a beefy enough system, you may find that the emulators won't work or will be too slow to be useful.
>
> When I worked on *PhoneGap Essentials*, I found that I couldn't run any of the Windows Phone development tools in a Windows VM, and only one of my lab machines had the horsepower to run the emulators. For this manuscript, I upgraded one of my lab systems to a brand new quad core processor and a lot of memory to make sure it would perform for me. Even with that, when I first started the emulator on that system, I received specific instructions on how to configure my system's BIOS for optimal performance of the emulator.

Next, you must follow some administrative steps to be able to deploy Windows Phone applications into the Windows Phone Store or onto a physical device. To help you get started with the process, take a look at "How to deploy and run a Windows Phone app" on Microsoft's website at http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff402565(v=vs.105).aspx.

First, you need to create a Microsoft Account. Accounts are free and can be obtained at https://signup.live.com. Next, you need to register as a member of the Windows Phone Dev Center at https://dev.windowsphone.com/join. Joining the program isn't free, but it doesn't cost that much. With your registration, you get the ability to deploy your applications to a physical device and into the Windows App Store.

# Configuring a Windows 8 Device for Application Testing

The Visual Studio development environment allows you to easily deploy and test Windows Phone applications onto a Windows Phone emulator. However, before you can test your applications on a physical device, you must register the device with Microsoft. You can find specific instructions for the process at http://msdn.microsoft.com/en-us/library/windowsphone/develop /ff769508(v=vs.105).aspx.

First, you need to power on the device and connect it to your Windows 8 desktop via a USB cable. Since each registered device has to have a unique name, open Windows Explorer on your desktop system and change the name of the device to something you know is unique, perhaps using your initials in the device name.

In Windows 8, bring up the All Apps view and, under Windows Phone 8 SDK, open t Windows Phone Developer Registration. You will see a screen similar to the one shown in Figure 10.1.


**Figure 10.1** Windows Phone Developer Registration: Locked Device

In this example, my Windows Phone device is locked, so I have to unlock the device and click the Retry button to allow it to connect to the device. After you unlock your device, you should see a screen similar to the one shown in Figure 10.2.


**Figure 10.2** Windows Phone Developer Registration: Connected Device

When you click the Register button, you are prompted to log in with your Microsoft account. Once you are logged in, you are taken to a screen similar to the one shown in Figure 10.3.


**Figure 10.3** Windows Phone Developer Registration Completed

At this point, the device is registered with Microsoft and can be used to test your Cordova applications.

# Running a Cordova Application Using Visual Studio

To work with a Cordova application on Windows Phone 8, you must first create Cordova project using the steps outlined in "Creating a Cordova Project" in Chapter 4, "Using the Cordova Command-Line Interface." Once you have the application created, in a terminal window navigate to the project folder and issue the following command:

```
cordova platform add wp8
```

This command downloads a Windows Phone 8 project and extracts it to the project's platforms\wp8\ folder. Make whatever changes you want to make to the web application's content in the Cordova project's www folder, then issue the following command:

```
cordova prepare wp8
```

The CLI copies the web content over into the platforms/wp8/www folder. You can see an example of the files that are created in Figure 10.4.

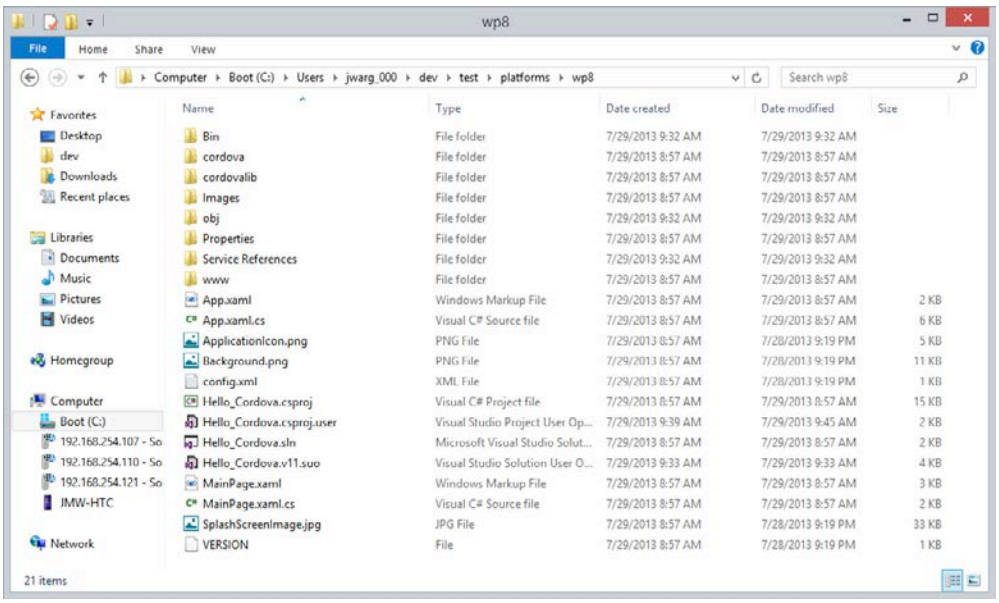



**Figure 10.4** Cordova Windows Phone 8 Platform Folder

Notice from the figure that the Visual Studio project is a Visual C# project (hello_cordova.csproj). Even though the default project type for Windows Phone 8 is a hybrid project, for Cordova applications it's a C# project. Because of this, developers of Cordova applications cannot leverage the web application debugging capabilities of Visual Studio; instead, they must use the `console` object and weinre (described in Chapter 6) to debug their Cordova applications for Windows Phone 8. There is also a third-party remote JavaScript debugger called Aardwolf (http://lexandera.com/aardwolf), which may prove useful. I've not tested it, but it looks like it's one solution for debugging Cordova applications on Windows Phone.

Now that you have created a Cordova project for Windows Phone 8, open Visual Studio 2012 Express for Windows Phone. When the development environment opens, it displays a screen similar to the one shown in Figure 10.5.

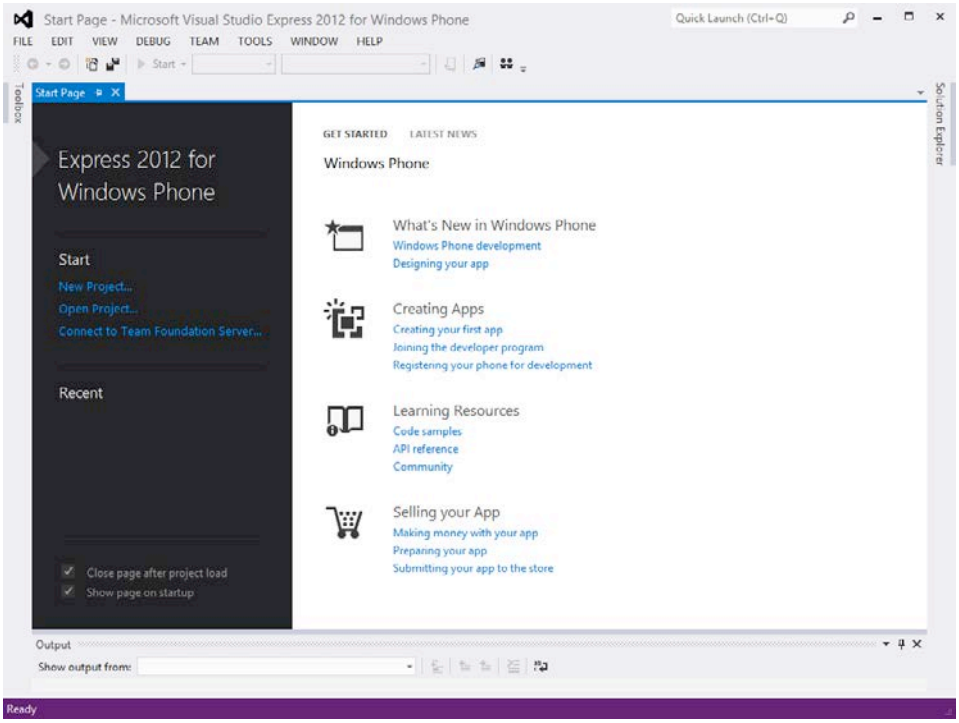



**Figure 10.5** Visual Studio Express 2012 for Windows Phone

To open the Cordova project, click the Open Project link shown on the left of Figure 10.5, or open the File menu and select Open Project. Visual Studio displays the Open Project dialog shown in Figure 10.6. Navigate to the project folder, shown in Figure 10.4, and open the Microsoft Visual Studio Solution file, as shown in Figure 10.6.

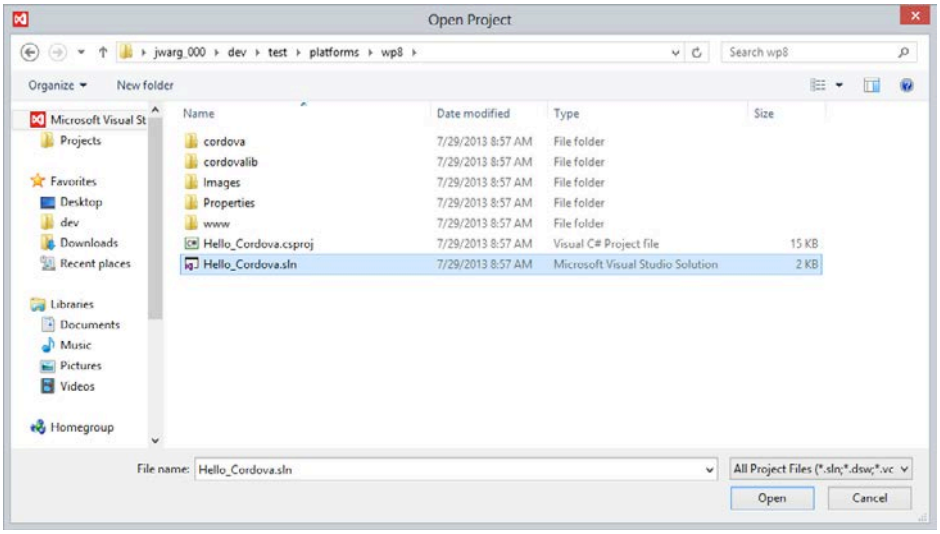



**Figure 10.6** Visual Studio: Open Project Dialog

When the project opens, the Solution Explorer will show you all of the files in the project, as shown in Figure 10.7. From here you can open the different web content files generated by the CLI; keep in mind, though, that the files shown here are copied from the Cordova project's www folder. Any changes you make to the web application content here must be copied back to the Cordova project's www folder before they can be applied to other platform projects.

To test applications in Visual Studio, use the options highlighted at the top of Figure 10.7. Click the green play symbol to run the application. The drop-down list to the right of the Play icon allows you to select the emulator or a physical device where the application will run.

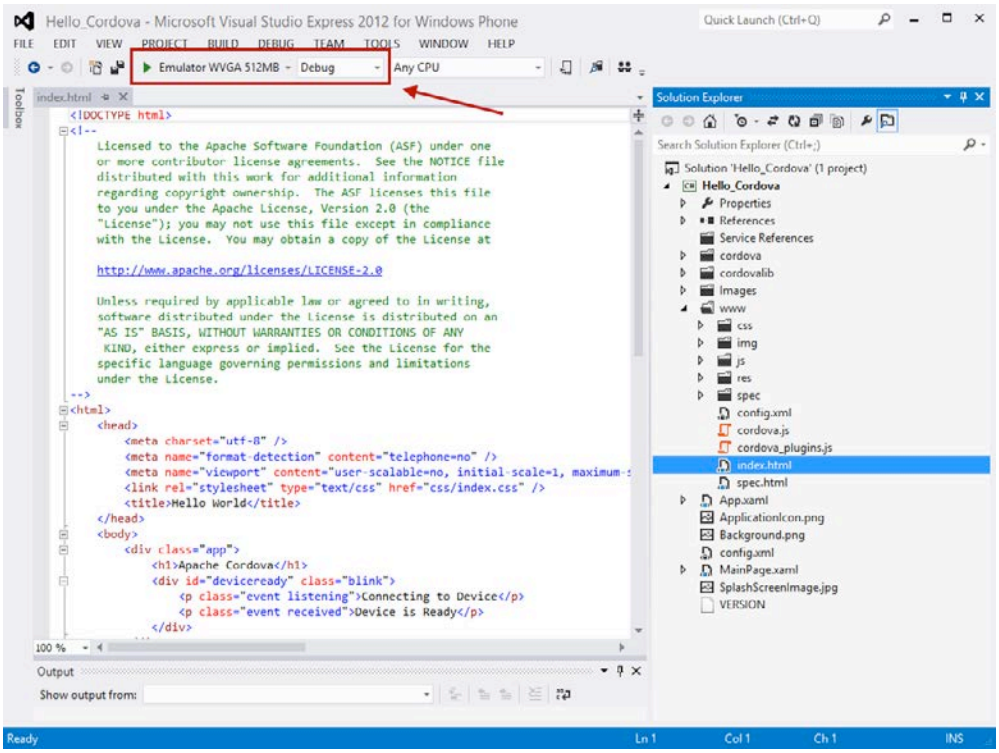



**Figure 10.7** Cordova Application Project Open in Visual Studio

To run the application on a physical device, make sure the device is powered on, unlocked, and connected to the development system using a USB cable. When running on an emulator, the emulator window will open, start the emulated OS, and run the application, as shown in Figure 10.8. The emulator looks and works like a regular device. You can swipe, click, and work with the application just as you would on a physical device.

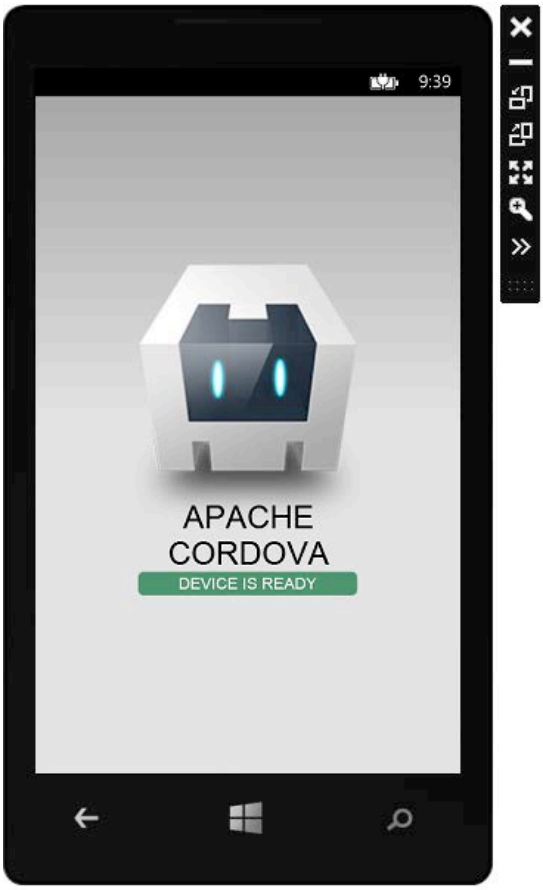



**Figure 10.8** Windows Phone 8 Emulator

The icons to the right of the figure provide developers with some additional control over the emulator. You can change the orientation of the device, expand the size of the emulated device, or change the zoom level. The double brackets (>>) icon at the bottom of the list is used to open an Additional Tools window, which provides some additional capabilities for manipulating the emulator.

The Accelerometer pane of the Additional Tools window is shown in Figure 10.9; it allows you to manipulate the orientation of the emulator along three axes. You can hold the primary mouse button down over the orange dot in the middle of the device image and move it around to position the emulated device. You can also select preset orientations or play back recorded orientation changes. This is a simple way to test mobile applications that leverage the accelerometer.

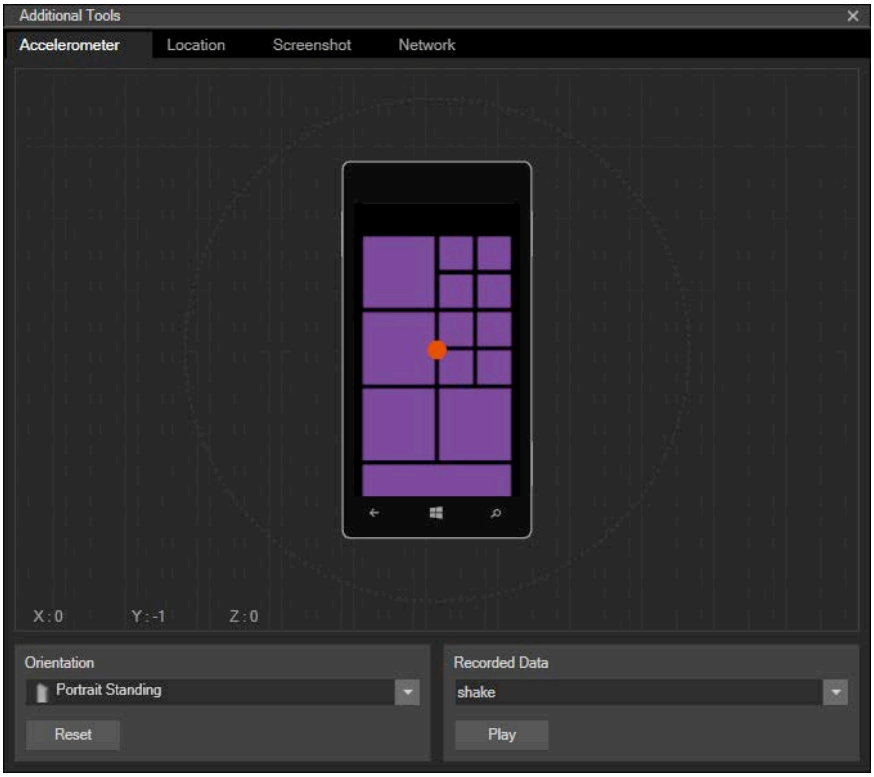



**Figure 10.9** Emulator Additional Tools: Accelerometer

Figure 10.10 shows the Location pane; from here you can manipulate the emulator's location. You can search for a specific location or address and push it into the emulated device's GPS coordinates. You can also play back recorded trips.

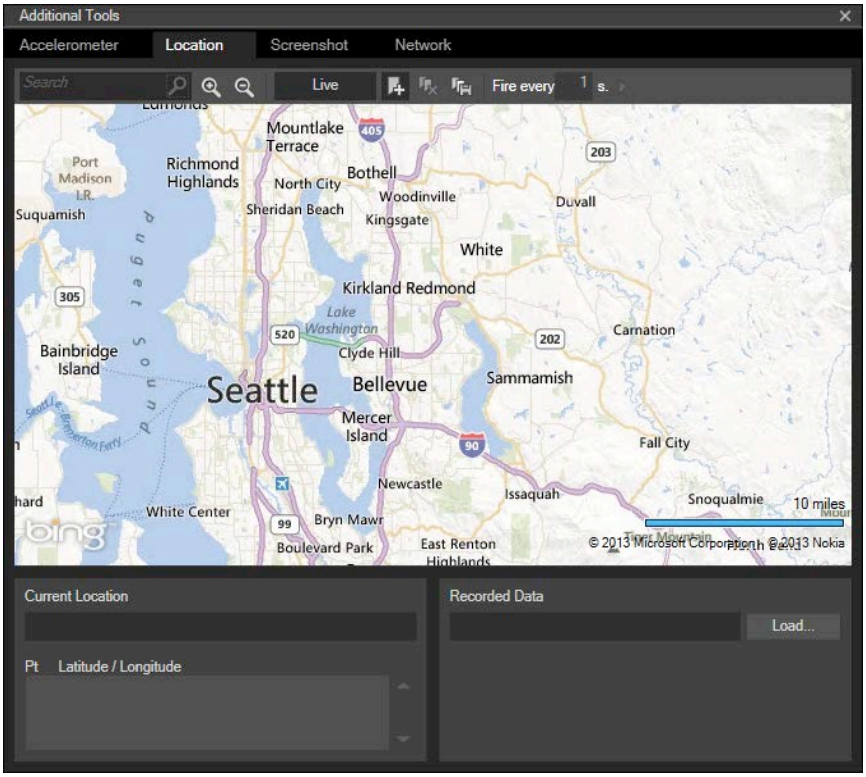



**Figure 10.10** Emulator Additional Tools: Location

When writing documentation for your application, you can easily grab device screenshots using options in the Screenshot pane.

# Wrap-Up

In this chapter, I showed you how to use the free developer tools from Microsoft to test your Cordova applications for Windows Phone 8. Due to the implementation of Cordova framework on Windows Phone 8, you can't leverage the debugging tools available through Visual Studio, but the tools still provide a great way to perform initial testing of a Cordova application.

# 11

# Using PhoneGap Build

Before the Cordova CLI became available, building Cordova applications for multiple device platforms was a challenge. You had to install the native SDK for each mobile device platform you were supporting and copy your Cordova application's web application content from project to project. You couldn't build or test multiple apps simultaneously. The CLI fixes much of that, but there are still limitations. The PhoneGap Build service provides the means to build PhoneGap applications in the cloud, without the need to install a bunch of software on a developer workstation. All you have to do is write your web applications using your web content editor of choice, then upload the files to the cloud and let PhoneGap Build do the rest.

In this chapter, I show you how to set up and use the PhoneGap Build service to package your Cordova applications and share those applications with others. There's a lot to PhoneGap Build, so I won't be able to cover it all here. This chapter focuses on the Build service and how to use it, not on the device platform–specific intricacies.

## What Is PhoneGap Build?

PhoneGap Build (https://build.phonegap.com) is a cloud-based build service for PhoneGap applications. It is a commercial offering from Adobe; there's a free version of the service plus for-fee options that offer more capabilities.

With Build, a developer loads a web application into an Application definition on the Build server, and the service automatically builds a native application for each of the supported mobile device platforms. Figure 11.1 illustrates how the build service works.

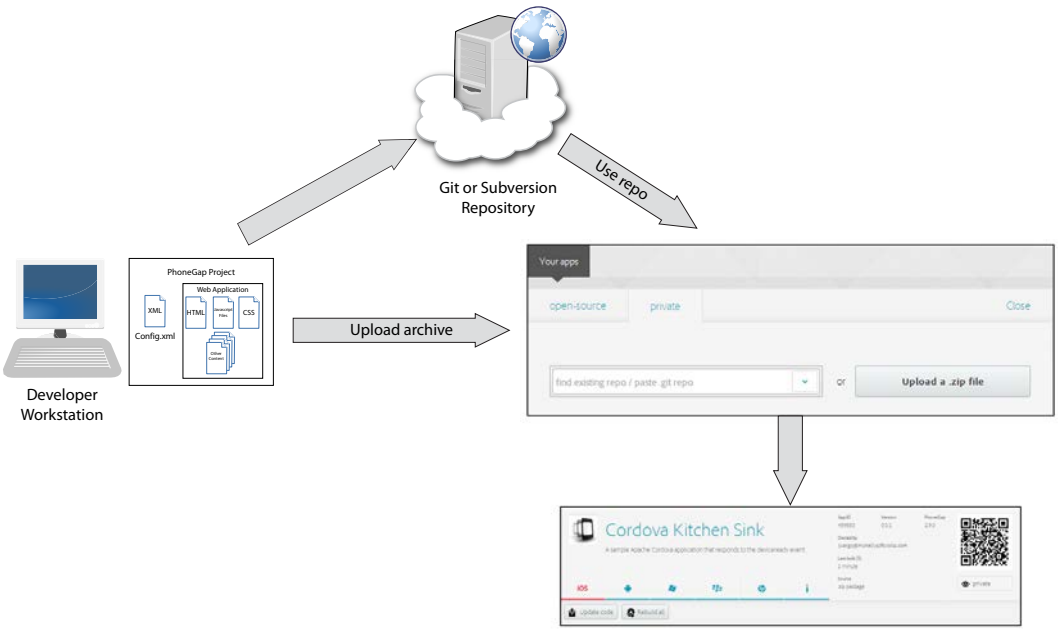



**Figure 11.1** PhoneGap Build Overview

PhoneGap Build currently supports packaging applications for the following mobile device platforms:

- Android
- BlackBerry
- iOS
- Symbian
- webOS
- Windows Phone

Currently, the service is a little behind on BlackBerry version support, but hopefully, BlackBerry 10 support will be available by the time you read this book. If not, don't worry—you can still work with BlackBerry 10 using the BlackBerry NDK and the Cordova CLI.

There are numerous features of the service, many of which are described later in the chapter. For developers evaluating using the service, there are a few key ones, which are discussed in the following sections.

### My Take on PhoneGap Build

PhoneGap Build was in beta when I wrote *PhoneGap Essentials*, but I used it heavily throughout the preparation of that book's manuscript. It simply made it too easy for me to spin up a quick web application and make it available on multiple devices or emulators. I still had to install every SDK so I could write about them, but every application in the book was built and tested using PhoneGap Build. I found the service to be very slow and pretty buggy (don't forget, it was in beta), but regardless of those limitations, it still provided more benefits than pain.

Now that we have the Cordova CLI, and you can manage and test application projects for multiple mobile device platforms simultaneously, I'm not sure I see the value for PhoneGap Build. It's cool and a big time saver, but for free I can install the supported mobile device platform SDKs locally and use the CLI to orchestrate them seamlessly. That plus the fact the Build service doesn't support all plugins and is generally behind on releases make it a less interesting offering for me.

## Quick Prototyping

PhoneGap Build allows a developer to quickly build a web application and deploy it into native applications for multiple mobile device platforms without installing any mobile device SDKs. This gives designers a quick way to flush out a concept without having to spend the time downloading and configuring mobile SDKs. While this prototyping can be easily done using a mobile web browser, if it is a requirement that the application be distributed in the Cordova (or PhoneGap) container, PhoneGap Build helps jumpstart the process.

Once a concept has been validated, the web application source code can be passed onto a developer to add data integration and other complex aspects of the application later. The developer can continue to use the Build service or can switch over to the CLI and local copies of the native SDKs for continued development.

## Collaboration

While version control systems allow developers to collaborate on their development projects on-premise or in the cloud, Build allows you to configure a project so others can work on the same PhoneGap Build project with you. You have the ability to define testers who get read-only access and developers who get read/write access.

This feature allows you to provide a private, hidden area of Build where you can collaborate with others and give them access to build results so they can easily test your applications as you develop them.

## Content Refresh through Hydration

One of the reasons many developers use Build is so that they can provide testers with a quick and simple mechanism for updating an application they are testing. Mobile application testers are used to having to update native applications on their test device as they move from app version to app version, but PhoneGap Build makes this process simpler through a feature called Hydration (https://build.phonegap.com/docs/advanced-hydration).

When a developer enables Hydration on a PhoneGap Build application, the Build service creates a version of the application that receives its web content from the build server instead of having its web content packaged within the application. This feature significantly improves application build time and allows web content updates to be repeatedly deployed to the PhoneGap application over the air, whenever the application launches and a new version is available.

When a developer uploads a new version of the web application content to the Build service, the content is packaged for delivery (instead of the whole native application being packaged). The next time the user opens the application, the application will notify the user that a new version of the application's content is available. Users can download the update or continue to work with the version of the application they already have.

The Hydration feature is really designed to support the testing process for mobile applications; you shouldn't try to use this feature with your production applications. For enterprise applications that need this feature, you may want to take a look at the SAP Mobile Platform (SMP); its Kapsel component provides an over-the-air web content update for production applications.

Hydration is supported in PhoneGap 2.0 and higher, and the capability is available for Android and iOS applications.

# Using PhoneGap Build

In this section, I show you how to use the PhoneGap Build service. I start by showing you how to create a PhoneGap Build account. Then I walk you through a quick example of creating a PhoneGap application using Build and show you how to leverage more of the configuration options for the service.

It all starts by opening your browser of choice and pointing it to the Build service at http://build.phonegap.com, as shown in Figure 11.2.

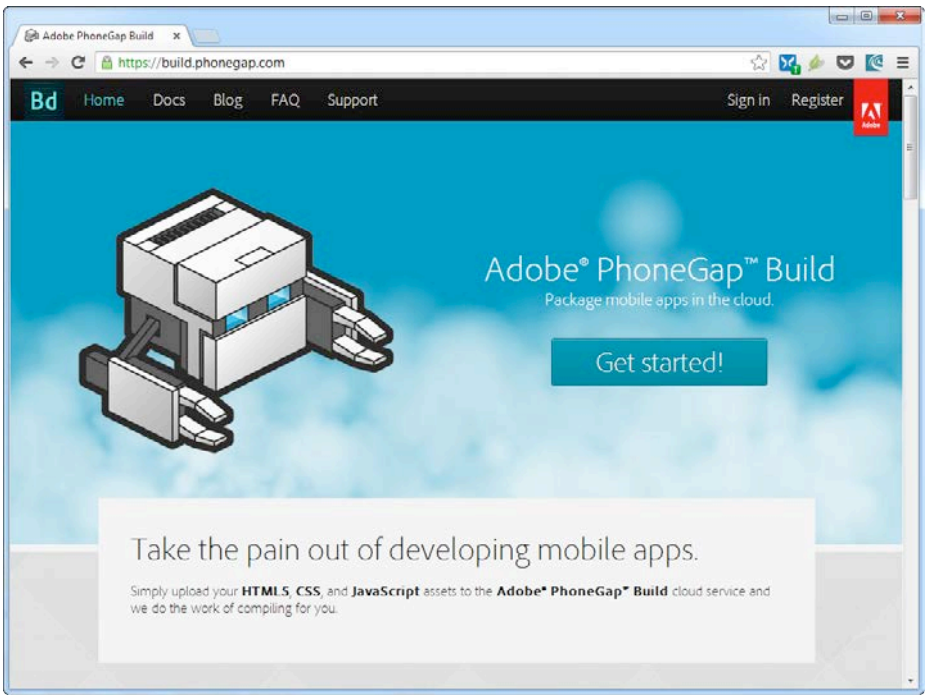



**Figure 11.2** PhoneGap Build Website

If you have an existing Adobe Creative Cloud account, you can sign in using the credentials you already have. If not, you can create a new account or sign in using your GitHub account credentials.

## A Quick Example

Before I show you all of the intricacies of PhoneGap Build, I thought I'd throw together a quick example. At the barest minimum, all PhoneGap Build needs to have in order to create a mobile application for each supported mobile device platform is just a single HTML file. To prove this, I created a quick HTML5 application, shown in Listing 11.1.

**Listing 11.1 Sample Single File Web Application**

```
<!DOCTYPE HTML>
<html>
  <head>
    <title>Sample Application</title>
  </head>
  <body>
    <h1>Sample Application</h1>
    <p>This is a sample web application.</p>
  </body>
</html>
```

Next, I signed into the PhoneGap Build website. If you don't have any applications defined on the system, then PhoneGap Build will open to a page where you can create your first application. Otherwise, click the + new app button on the Build home page to create a new application.

In either case, the Build service will open the page, shown in Figure 11.3; here you can either provide a URI for a Git repository or upload a file. In this case, the upload button label is mislabeled (and I've suggested to the Adobe folks that they fix this) because you can upload more than a zip file—you can also upload the .html file I just showed you. Regardless of which approach you want to use, you have to make sure Build has access to the application file(s) you want packaged into a Cordova application.

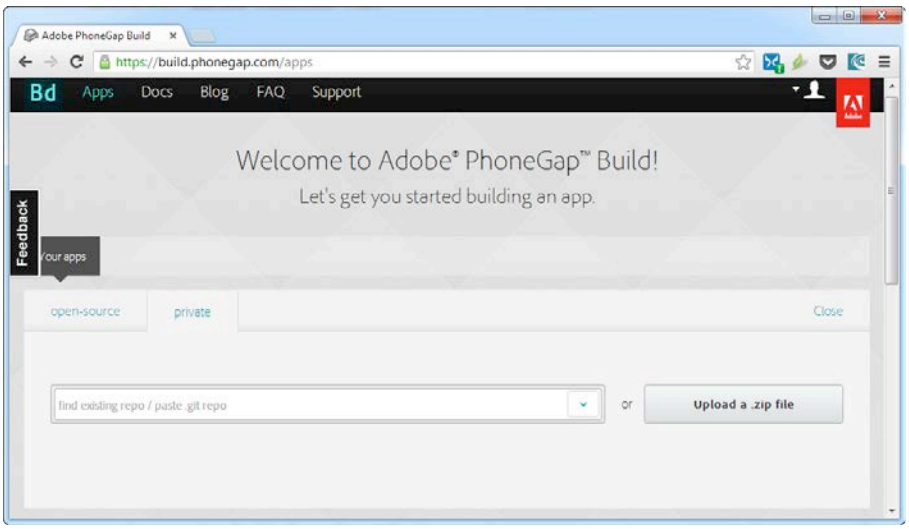



**Figure 11.3** PhoneGap Build: Creating a New Application

For this example, I simply clicked Upload a .zip file and pointed to the .html file I created. As soon as I finished that process, the Build service opened the new page shown in Figure 11.4. Since I didn't provide a configuration file (more on that later), Build set a default application name of PG Build App, as shown in the figure. You should change the application name and provide a brief description for your application so you and others will be able to easily recognize it later.

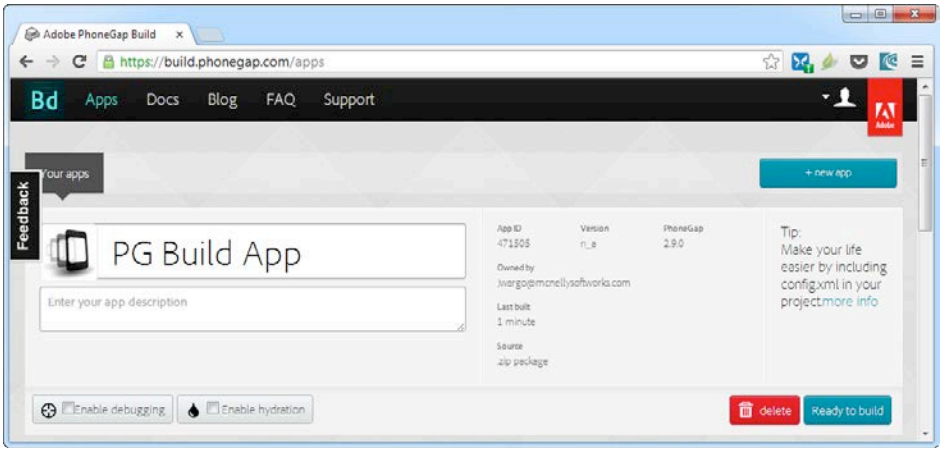



**Figure 11.4** PhoneGap Build: New Application

If you check the Enable debugging checkbox, PhoneGap Build enables weinre debugging in the packaged application; weinre was discussed in Chapter 6, "The Mechanics of Cordova Development." If you check the Enable hydration checkbox, Build enables the Hydration feature described earlier in the chapter. Figure 11.5 shows the application definition with a more appropriate name and description.

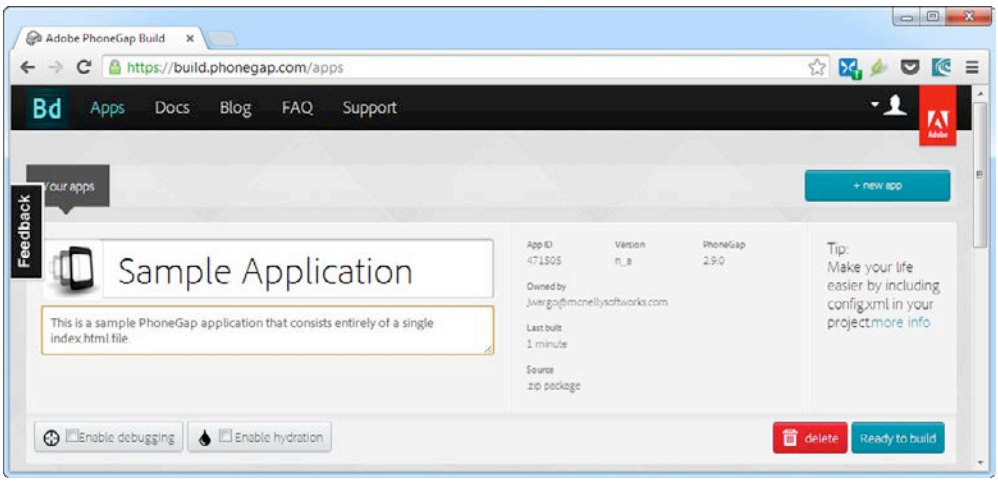



**Figure 11.5** PhoneGap Build: New Sample Application

When you have the application settings set the way you want them, click the Ready to build button to start the packaging process. Build will begin the process and display the page shown in Figure 11.6. What the service does at this point is spawn off a bunch of tasks to build native Cordova applications for each supported mobile platform using the web application content you have provided.

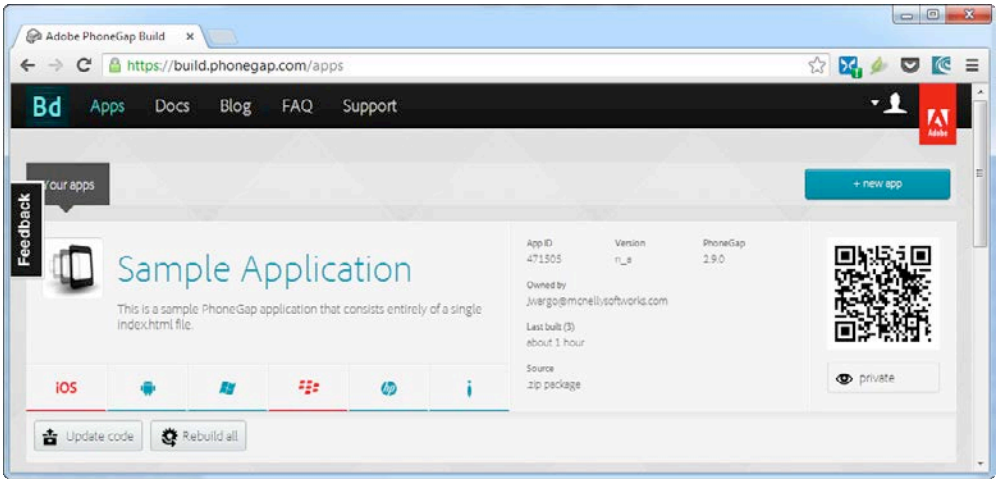



**Figure 11.6** PhoneGap Build: Sample Application Being Built

The page will show the platform icon for each supported mobile device platform; you should be able to easily recognize them. The HP logo is for webOS, and the stylized "i" is for Symbian. Platforms that are being built will be grayed out as the build process progresses. The applications with blue icons have completed, and red icons indicate that the build has failed. I'll explain why the build failed in a minute.

The build process fails sometimes because of glitches with the Build service. Simply click the Rebuild all button to redo the build, and it usually fixes itself. In this example, though, the build is failing for a specific reason: the application signing keys needed to complete the build process have not been defined for this application.

If you click on the application title in Figure 11.6, or on one of the mobile platform icons toward the bottom of Figure 11.6, the Build service will open a page similar to the one shown in Figure 11.7. From this page, you can see more details about the particular application. In this example, you can see where applications have been built successfully and where the build has failed.

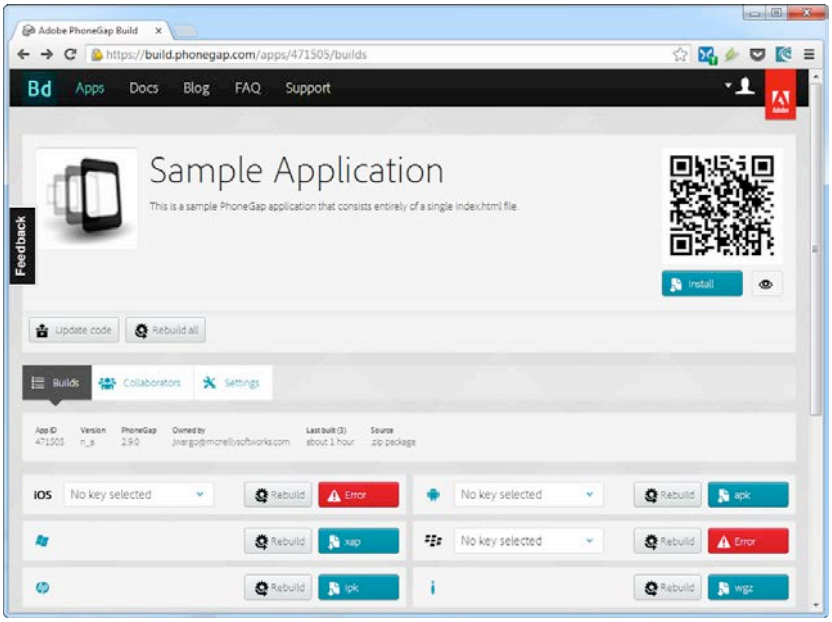



**Figure 11.7** PhoneGap Build: Application Details

If you click one of the red Error buttons, the page will expand an area beneath the button and display the error that caused the build to fail; an example of the iOS failure is shown in Figure 11.8. In Figure 11.7, both the iOS and BlackBerry builds have failed because of signing key problems.

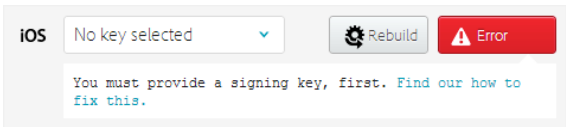



**Figure 11.8** PhoneGap Build: iOS Build Error Details

Most mobile device platforms have some sort of signing process that must be completed before an application can be deployed to a device or into an app store. Apple is much more restrictive than other platforms, so you can't accomplish anything here without providing keys. Most other platforms either don't require a key, or they allow you to build with or without one.

For BlackBerry, Build used to use their own keys to sign a BlackBerry application and allow you to add yours later; in this example, the build is failing because of a password problem with the default keys. Either there is something wrong with the keys, or you can't build BlackBerry applications anymore without providing your own signing keys.

All you have to do to resolve these build issues is provide the necessary signing keys and rebuild the application. To do this, click the appropriate drop-down menus next to the operating systems that have failed, shown in Figure 11.7, and add your keys to the project. Some keys are global—BlackBerry keys, for example, can be used to sign any application—and some keys are more restrictive, as is the case with iOS application signing.

When you create the keys, you're able to provide a title for the key set. This allows developers who work for multiple customers to define signing key settings for different projects or customers and select the appropriate key(s) depending on the scenario.

When you have the appropriate keys added to the project, simply click the individual Rebuild buttons or the Rebuild all button to start the process of rebuilding the application.

## Deploying PhoneGap Build Applications

When the build process completes, it will provide you with access to the packaged applications for each of the supported mobile device platforms. You can deploy these applications to mobile devices in different ways, depending on what is supported by the device manufacturer and even the mobile carrier. In this section of the chapter, I show you in general how to deploy the applications; you will have to refer to the specific documentation for each mobile device platform to determine which options are appropriate for the platform.

The simplest way to deploy an application built using PhoneGap Build is to grab a compatible device and use a code-scanning application on the device to scan the Quick Response (QR) code, shown in the upper right corner of Figure 11.7. The code-scanning application will convert the scanned code into a URL and open the web page on the mobile device. All the user has to do is follow the instructions on the web page to initiate the application download and installation.

Each mobile device platform has restrictions on what can be loaded directly onto a device, so you may have to change settings on the device to allow this. Android, for example, has an issue with downloading applications from unknown sources, but a quick configuration change can enable it. The setting is typically in the Security area of the Settings application and should refer to Unknown Sources. You can see an example of the setting in Figure 11.9.

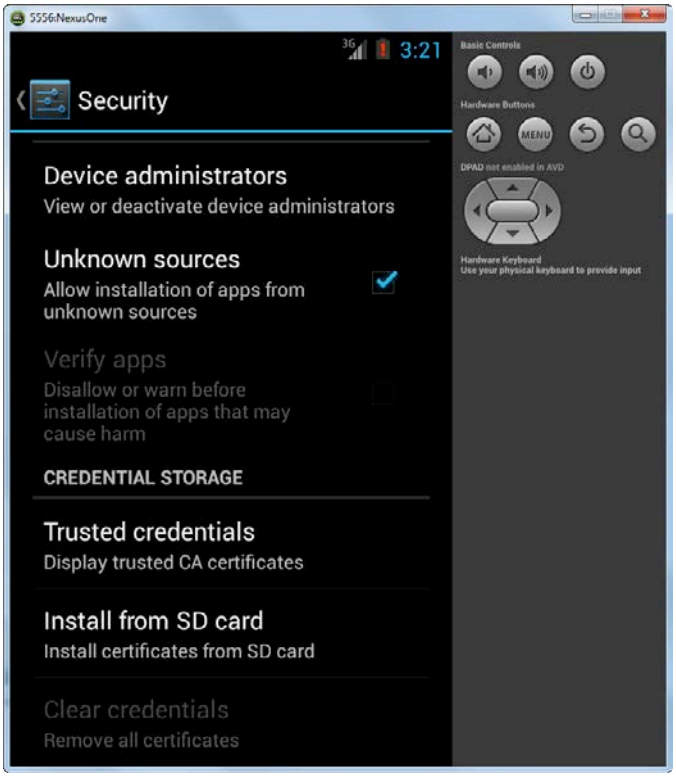



**Figure 11.9** Enabling Unknown Sources on Android

If you need to make your application available to other users, you can use the Collaborators settings for the application to allow other users to access your packaged applications. You can define collaborators for your application by clicking the Collaborators button, shown in the middle of Figure 11.7, and following the steps that appear, defining an email address for each authorized collaborator. Those users can log in to the PhoneGap Build website using the specified email address and download the applications they're authorized to access.

In some cases, you will want to have access to the native application executable file. In this case, you can download the packaged application directly to your development system. If you take another look at Figure 11.7, you'll see that there are blue buttons to the right of each mobile device platform shown at the bottom of the figure. If you click the button for a particular mobile device platform, the browser will download the appropriate files for the platform, downloading a .xap file for a Windows application, an .apk file for Android, and so on.

You can also click the Install button immediately below the QR code shown in Figure 11.7 to be taken to another page where you can download all of the application files directly, as shown in Figure 11.10.

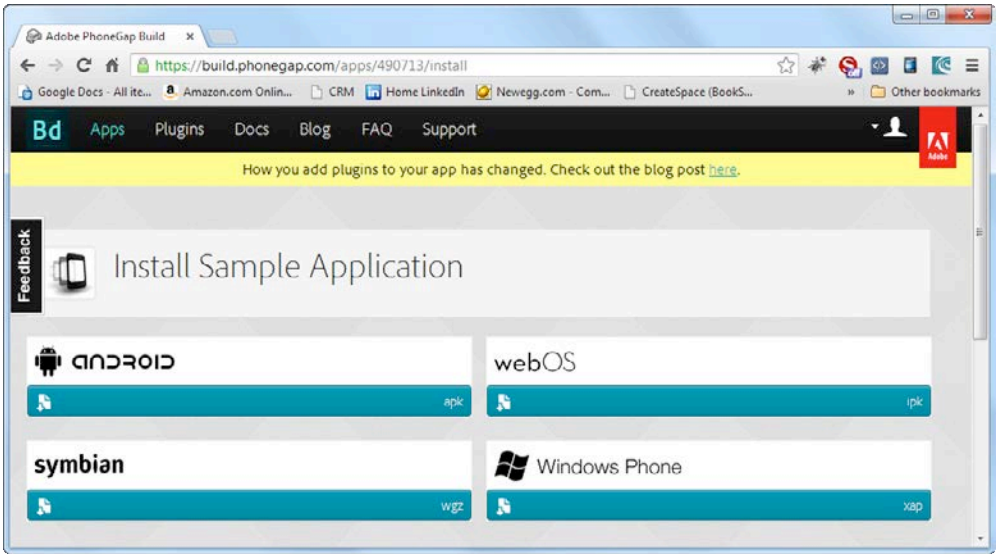



**Figure 11.10** PhoneGap Build: Install Page

Once you have the application files downloaded, you can distribute the applications to others and deploy the applications directly to supported devices.

To test these applications on one or more device simulators, you can open the mobile browser on the simulator, then navigate and log in to the PhoneGap Build service using your credentials. If the simulator supports it, you can click the Install link and download the application's files directly to the device from the site.

When you install the application, you may be prompted to enable features of the application, as shown in Figure 11.11. Even though I've built only a simple application that doesn't even use any of the PhoneGap APIs, since I didn't tell the Build service what to enable or disable, it enables all features by default. I'll show you in the next section how to configure a PhoneGap Build project so you can leverage a lot more features of the service.

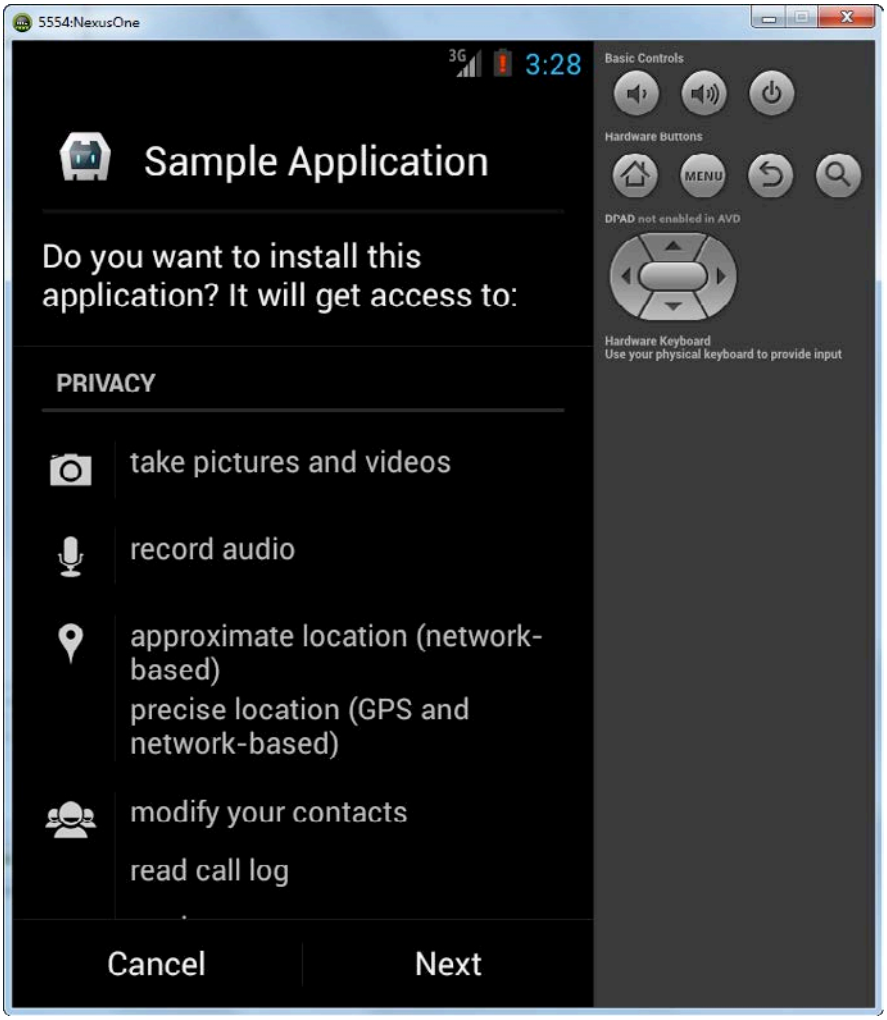



**Figure 11.11** Android Application Security Prompt

At this point, you have all of the information you need to get started with PhoneGap Build. In the next section, I show you how to configure an application to make better use of the platform.

## Configuring a PhoneGap Build Application

As shown in the previous section, PhoneGap Build will use default settings such as application icon, splash screen, security settings, and more unless you tell it differently. To configure application-specific settings for your PhoneGap application, PhoneGap Build uses the config.xml file defined as part of the W3C Widget Packaging and XML Configuration specification.

In the previous example, I uploaded just a single HTML file to the service, and it built the application for me. If I create a properly configured config.xml and include it in a .zip file with all of the web application's content, the service will use the config.xml file to set many of the properties and security settings for the application. The .zip file can contain just the index.html I used in the previous example, or it can be a complete web application with .html files, JavaScript code, CSS files, and more. It doesn't matter what the complexity of the app is; as long as it is packaged into a standard .zip file with the config.xml, the Build service should be able to package it into native Cordova applications for you.

The CLI will generate a config.xml file for you automatically when it creates a Cordova project. Listing 11.2 shows the config.xml file created by the CLI for the HelloWorld4 application highlighted in Chapter 5, "Anatomy of a Cordova Application."

**Listing 11.2 Default Cordova Project Config.xml File**

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="com.cordovaprogramming.HelloWorld4" version="0.0.1"
  xmlns="http://www.w3.org/ns/widgets"
  xmlns:cdv="http://cordova.apache.org/ns/1.0">
  <name>Hello World 4</name>
  <description>A sample Apache Cordova application that
    responds to the deviceready event.</description>
  <author email="dev@callback.apache.org" href="http://cordova.io">
    Apache Cordova Team
  </author>
  <access origin="*" />
  <preference name="fullscreen" value="true" />
  <preference name="webviewbounce" value="true" />
</widget>
```

The example config.xml file lists a limited amount of information needed to describe the application and a little bit about the application's preferences. If you updated the application name and author information in that file, packaged the config.xml into an archive, and uploaded it to PhoneGap Build, you would see that the new application project would be created using those settings.

If you remember from Chapter 8, "BlackBerry 10 Development with Cordova," I showed how the BlackBerry platform also uses the Widget specification for its applications, and the sample config.xml in Listing 8.1 shows that there's a whole bunch of additional information provided in the file to help configure security permissions and more for the application. Since the Build service has to accommodate a wider range of mobile platforms and their specific project settings, the PhoneGap Build config.xml file can be pretty complicated. It has to contain settings that apply across multiple device platforms, such as settings for application icons, splash screen graphics, and more. It also has to accommodate specific security settings that need to be enabled depending on which PhoneGap APIs are used.

Initially, PhoneGap Build supported only a small catalog of plugins. With the release of PhoneGap 3.0, PhoneGap Build added support for a much larger catalog of plugins, along with the ability for developers to publish their plugins to the service so others can use them. This caused a change in the way plugins were identified in the config.xml. Instead of describing all of the config.xml options here, the topic is well covered in the PhoneGap Build documentation at https://build.phonegap.com/docs /config-xml. On that page, you'll find a detailed description of each of the options for the file plus a

pointer to a sample config.xml file you can use for reference or as a starting point for your own applications.

## Wrap-Up

In this chapter, I showed you how you can use the PhoneGap Build service to package and deploy your web applications into the PhoneGap container. You should look at leveraging this service if you don't want to worry about managing multiple SDK installations or if you need an easy way to build and share your applications with a distributed audience of testers.

# 12

# Working with the Cordova APIs

So far I've shown you a lot about Cordova: how to set up a Cordova development environment and use the tools provided by the Cordova team and the mobile platform vendors. Now it's time, in this chapter and the two that follow, to talk more about how to build Cordova applications. In this chapter, I introduce you to the Cordova APIs and show you how to use them in your applications.

I'm not going to go super deep into each of the APIs and how they work; the Cordova documentation is a lot better than it used to be, but I will show you how to use them in your applications. In Chapter 13, "Creating Cordova Plugins," I show you how to write your own Cordova plugins and wrap it all together in a complete Cordova application in Chapter 14, "Building a Cordova Application."

## The Cordova Core APIs

Hybrid applications become much more powerful when they can do things that simply cannot be done within the mobile browser directly. As mentioned in the first chapter, one of the purposes behind Cordova was to provide mobile web applications running within the Cordova container access to native APIs. Cordova provides access to only a subset of the APIs available on a mobile device; the APIs are

- Accelerometer
- Camera
- Capture
- Compass
- Connection
- Contacts
- Device
- Events
- File
- Geolocation
- Globalization

- InAppBrowser
- Media
- Notification
- Splashscreen
- Storage

In some cases, an API mimics a capability that is already provided in modern mobile browsers. In most cases, applications simply use the API exposed through the native browser, WebView, embedded in the Cordova container. For devices for which their browser does not expose the API, the Cordova team will implement the API so it is available across all supported mobile device platforms. You can see examples of this in the Cordova Geolocation, File and Storage APIs; they're pretty much standard on most mobile devices, so there's little the Cordova team has to do with them. As an example, take a look at the documentation for the Geolocation API:

> This API is based on the W3C Geolocation API Specification. Some devices (Android, BlackBerry, bada, Windows Phone 7 and webOS, to be specific) already provide an implementation of this spec. For those devices, the built-in support is used instead of replacing it with Cordova's implementation. For devices that don't have geolocation support, the Cordova implementation adheres to the W3C specification.

Because the File and Storage APIs are standards that are already available in the mobile browser, I don't cover them in this chapter. The Geolocation API falls into the same category, but the "Hardware APIs" section of the chapter covers other APIs that work just like it, so Geolocation gets some coverage there as well. The Contacts API is also based on a standard, but I'll give it some coverage here as it's unique in how it works compared to other Cordova APIs.

As mentioned in previous chapters, instead of APIs being readily available to any Cordova application, beginning with Cordova 3.0, all of the core APIs have been removed from the Cordova container and implemented as plugins. So, before your Cordova application can use any of the Cordova APIs, you must add the associated plugin to your project using a command similar to the following:

```
cordova plugin add path_to_plugin
```

So, for example, to use the Camera API in your Cordova application, you must add the camera capabilities using the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-camera.git
```

This will pull the camera plugin files from the public Git repository; if you want to use the camera plugin from a local repository, you can pass in instead the local file path pointing to where the plugin files are located, as shown below:

```
cordova plugin add d:\cordova\cordova-3.0.0\cordova-plugin-camera
```

You can find more information about how to use the CLI to manage plugins in Chapter 4, "Using the Cordova Command-Line Interface."

# Working with the API Cordova Documentation

As I mentioned earlier, the Cordova API documentation is pretty good. Just as there is a team working full time on the Cordova container, APIs, and plugins, there is also a dedicated team working very hard on the docs. As soon as I finish this manuscript, I plan on joining them as well. Because of

all of this effort, the docs are pretty good, and you will need to spend a fair amount of time with them as you build your Cordova applications; I'll explain why in a little bit.

To access the Cordova API documentation, point your browser of choice to www.cordova.io and look for the documentation link at the bottom of the page. When I select that link, I am redirected to http://cordova.apache.org/docs/en/3.0.0, which points to the appropriate language (English) version plus points me to the latest and greatest released version of the framework.

From the documentation page, use the drop-down shown in the upper right corner of Figure 12.1 to switch Cordova versions. The drop-down provides access to the documentation for the current Cordova version, previous versions, and even one future release of the framework.

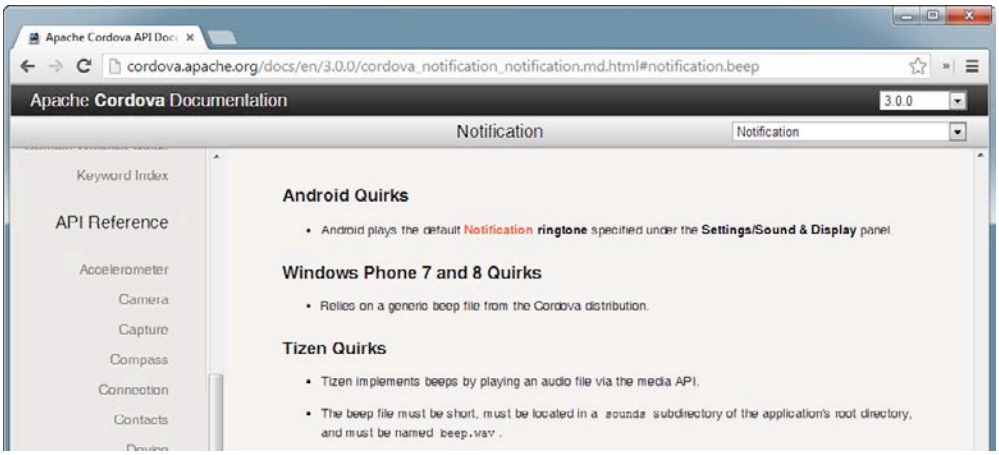



**Figure 12.1** Cordova Documentation: API Quirks

When working with the APIs, you will want to pay special attention to the quirks section of the documentation for each exposed API; an example is shown on the Notification API page in Figure 12.1. Many of the APIs display different behavior on specific hardware platforms. As hard as the Cordova development team works to ensure consistency across all platforms, on some platforms an API needed to implement some feature simply isn't available or doesn't work as needed for the Cordova API. If you're trying something with a Cordova API and it is not working as you expect it to, go check out the quirks for the particular API you're working with.

# Setting Application Permissions

Because most of the Cordova APIs leverage native APIs and many of them have security restrictions placed on them, in order to protect the user and/or device, developers using some Cordova APIs must set special application configuration settings for their applications before deploying them to devices. These settings configure the native application container so that it informs the mobile OS during installation that it uses specific restricted APIs. This causes the mobile OS for some platforms to prompt the user of the application during installation to allow or disallow the restricted API or APIs for the application.

As an example of how this will appear for users, take a look at Figure 12.2, which shows the security prompt displayed for users when installing the default HelloCordova application (described in Chapter 5, "Anatomy of a Cordova Application") on an Android device.

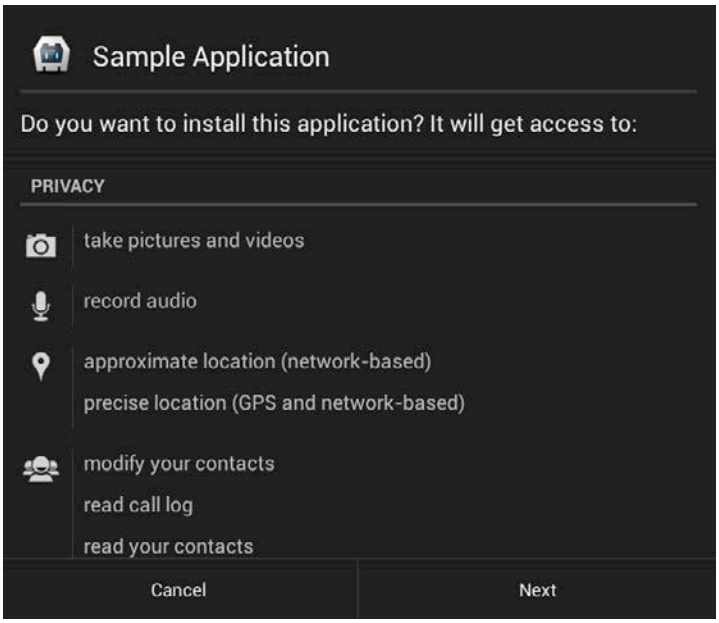



**Figure 12.2** Android Application Permissions Settings Dialog

### Warning

On many mobile device platforms, if a particular API or feature isn't enabled for a Cordova application as shown in this section, or if the user disables the feature through the dialog shown in Figure 12.1, the portion of the application that uses the feature will simply fail silently when the application runs.

You would think that the application would throw some sort of error when your application tries to use a disabled or restricted API, but that's not the case—the application simply skips the execution of the API and you're left scratching your head as to why the feature isn't working. If you've implemented a feature and it's simply not working and not telling you why, be sure to check your application permissions. In Chapter 14, I'll show you a trick you can use to catch those errors.

To find out what configuration settings must be set for each of the Cordova APIs your application uses, you must refer to the specific APIs documentation. Figure 12.3 shows a portion of the documentation page for the Camera API. The page contains a section called "Accessing the Feature," which describes how to add the API plugin to the project and what configuration settings must be set for the API to work in your applications.

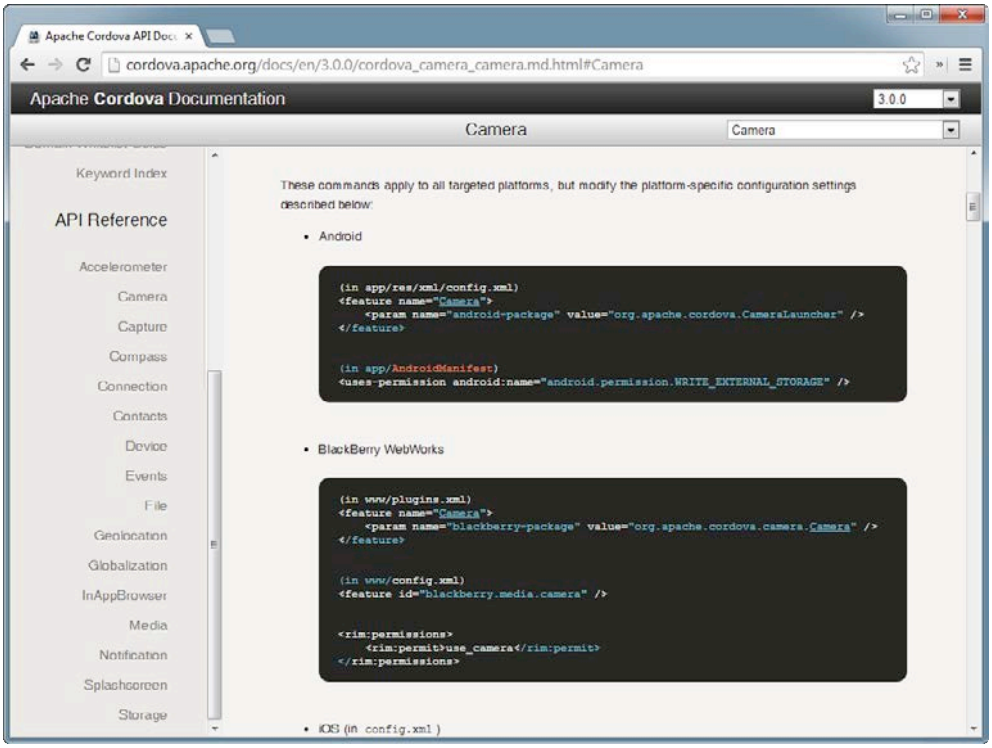



**Figure 12.3** Cordova API Documentation: Configuration Settings

As you can see from the figure, for each supported mobile platform, it describes the configuration file that must be updated and shows the content that must be included in the file to enable this feature. It's usually a config.xml file, but the location for the file will vary across platforms.

The good news is that the Cordova CLI manages setting application permissions for you. If you add the camera API to a Cordova application using the CLI commands highlighted earlier in the chapter, the CLI will make the appropriate configuration changes for you across all of the mobile OS targets you have added to the application project. So, when you add the camera API to an application project, the appropriate settings will make it into the Android config.xml file, as shown in Listing 12.1.

### Listing 12.1 Android Project config.xml File

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="io.cordova.helloCordova" version="2.0.0" xmlns="http://www.w3.org/ns/widgets">
  <name>Hello Cordova</name>
  <description>A sample Apache Cordova application that
    responds to the deviceready event.</description>
  <author email="dev@cordova.apache.org"
    href="http://cordova.io">Apache Cordova Team</author>
  <content src="index.html" />
  <feature name="App">
    <param name="android-package"
      value="org.apache.cordova.App" />
  </feature>
```

```
  <feature name="Camera">
    <param name="android-package"
      value="org.apache.cordova.camera.CameraLauncher" />
  </feature>
  <access origin="*" />
  <preference name="useBrowserHistory" value="true" />
  <preference name="exit-on-suspend" value="false" />
  <preference name="fullscreen" value="true" />
  <preference name="webviewbounce" value="true" />
</widget>
```

Okay, now that I've covered all of the background information, let's start talking about the APIs

# Cordova Objects

Some of the Cordova APIs are not necessarily APIs; instead they're useful objects that are exposed to a Cordova application through the plugin. In this section, I describe the `connection` and `device` objects and show you how to use them in your Cordova applications.

## Connection Type

There are times when a mobile application needs to know what type of network connection it has available to it. With many mobile carriers capping data usage, the application should be careful when doing large updates to make sure it is doing its update, for example, on a lower cost (free) Wi-Fi connection rather than a cellular connection. As a best practice, mobile developers should categorize an application's data usage patterns and write their code in order to minimize the impact on the device and the device user's data plan costs.

To accommodate this, the Cordova framework exposes a `connection` object, which can be used to determine the network connection type currently in effect. To leverage the information exposed by the `connection` object, you must first add the network information plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-network-
information.git
```

With that in place, a Cordova application can determine the connection type by executing the following JavaScript code:

```
var ct = navigator.connection.type;
```

Next, you can compare the value of `ct` against the possible connection types exposed through the following properties:

- `Connection.CELL`
- `Connection.CELL_2G`
- `Connection.CELL_3G`
- `Connection.CELL_4G`

- `Connection.ETHERNET`
- `Connection.NONE`
- `Connection.UNKNOWN`
- `Connection.WIFI`

So, if you want to make sure the application has a network connection before you do something networky, your application can do something like this:

```
ct = navigator.connection.type;
if (ct != Connection.NONE) {
  console.log("You have a connection");
  //Do whatever you want to do with the connection

} else {
  //Warn the user that they can't do that without a
  //network connection
  alert("No connection available!");
}
```

> **Warning**
>
> Your application has to query the `connection` object every time it wants to know what the current network status is. The `ct` variable shown in the previous example won't maintain a valid status value as the network status changes. When I cover events later in the chapter, you will see how you can monitor network connectivity.

Or, if you are looking for a specific network type, your application can do something like this:

```
ct = navigator.connection.type;
if (ct == Connection.WIFI) {
  console.log("You have a WI-FI connection");
  //Do whatever you want to do over the connection

} else {
  //Warn the user that they can't do that without a WI-FI
  //network connection
  alert("Function requires WI-FI network connection!");
}
```

An application can also monitor the network status in real time using the `online` and `offline` events described later in the chapter.

## Device

The Cordova framework also exposes a `device` object, which can be used to determine a limited amount of information about the device. The available device properties are shown in the following list:

- `device.name`
- `device.cordova`
- `device.platform`
- `device.uuid`
- `device.version`
- `device.model`

To leverage the information exposed by the `device` object, you must first add the device plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device.git
```

To see an example of the `device` object in action, take a look at the HelloWorld3 application described in Chapter 5.

# Alerting the User

There are often times when a developer will want to notify application users of some activity. A web application can display some information on a page within the application or even open up an HTML popup, but in both cases, the management of the notification (and its removal from view) is the responsibility of the developer. To make this easier, the Cordova API includes some JavaScript functions a developer can use to provide a notification to users. There are two types of notifications, what I call *hardware notifications* and *visual notifications*. Each is described in the following two sections.

## Hardware Notifications

Any modern smartphone provides an API that allows a developer to have an application make the device beep or vibrate; Cordova also exposes methods an application can call to make the device beep or vibrate as well.

To leverage hardware notifications in your Cordova applications, you must first add the vibration plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-vibration.git
```

### Beep

To cause a mobile device to beep, a Cordova application should make a call to the `navigator.notification.beep` method. To have the device vibrate, a Cordova application should make a call to the `navigator.notification.vibrate` method. Each takes a parameter that controls how many times or for how long the notification lasts, as shown in the following examples.

The `beep` method accepts a parameter that controls how many times the device beeps when the method is called:

```
navigator.notification.beep(quantity);
```

To have a device beep three times, simply pass in a 3:

```
navigator.notification.beep(3);
```

### Vibrate

The `vibrate` method accepts a duration parameter instead, which controls how long the vibration lasts:

```
navigator.notification.vibrate(duration);
```

Duration is expressed in milliseconds, so to make the device vibrate for half a second, you pass a value of 500 to the `vibrate` method:

```
navigator.notification.vibrate(500);
```

To make the device vibrate for 1 second, do the following:

```
navigator.notification.vibrate(1000);
```

## Visual Notifications

Cordova exposes a number of methods a web application can call to allow the application to interact with the user. Web developers have always had access to the synchronous JavaScript `alert()`, `confirm()`, and `prompt()` methods, which can be used to interact with the user, but the Cordova versions of these functions are asynchronous and allow for additional control over the content in the dialog that is displayed to users.

To leverage visual notifications in your Cordova applications, you must first add the dialogs plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-dialogs.git
```

### Alert

To display an alert dialog in a web application, a web developer can have an application execute the following JavaScript code:

```
alert("This is a JavaScript alert.");
```

When the code runs in a Cordova application, you will see something similar to Figure 12. 4.


**Figure 12.4** JavaScript `alert` Results

If you instead use the Cordova `alert()` method by executing the following code:

```
navigator.notification.alert("This is a Cordova Alert.",
  myCallback, "Alert Test", "Click Me!")
```

you will see a dialog similar to the one shown in Figure 12.5.


**Figure 12.5** Cordova `alert` Results

Notice that I was able to set the title for the dialog (instead of the Cordova application reporting index.html, as shown in Figure 12.4) as well as the text on the button the user taps to close the alert.

I mentioned that the JavaScript alert was synchronous and the Cordova alert was asynchronous; this means that the JavaScript alert will likely display immediately, but the Cordova alert will display whenever the Cordova container gets around to rendering it. This is an oversimplification of what's going on, but you can read more about it in an article I wrote here: www.johnwargo.com/index.php/mobile-development/phonegap-alerts.html.

Now, in that example, I didn't tell you a lot about the format of the call to `alert`; in the following example, you can see the descriptive names of the parameters:

```
navigator.notification.alert(message, callback, [title], [buttonLabel])
```

Notice the `callback` parameter; it's there to allow you to define the function that is executed when the user taps the button on the alert dialog. The way you use the Cordova alert is to execute the `alert` method, then use the `callback` function to continue program execution after the user taps the button. What happens is that any code you have after the call to `alert` is executed, perhaps finishing out the code in a JavaScript function, and the application will sit idle until the user taps the button; then the code in the `callback` function executes.

In the earlier example, I passed in a `null` value for the `callback` parameter. In this scenario, by not telling `alert` what function to call after the user taps the button, the Cordova container will render the specified alert dialog, then continue executing the JavaScript code that follows the call to `alert` and not wait for the user to tap the button.

The values for `title` and `buttonLabel` are optional; the value for `title` passed to the method will be used as the title for the dialog, and the `buttonLabel` value will be used as the text on the single button on the dialog.

### Confirm

The Cordova `confirm` method is similar to `alert` except that it allows you to specify more than one button label, and the `callback` function is passed a numeric value indicating which button was tapped by the application user. Here's the method signature:

```
navigator.notification.confirm(message, callback, [title],
  [buttonLabels]);
```

The following code snippet shows how to use the Cordova `confirm` method:

```
navigator.notification.confirm('Do you want to continue?',
  doContinue, 'Please confirm', 'Yes, No');

function doContinue(buttonNum){
  navigator.notification.alert('You chose option #' +
    buttonNum + '?', null, 'Really?','Yes');
};
```

The value passed to the callback function is a numeric value indicating which button was tapped. The callback function will receive a 1 if the first button was clicked, a 2 for the second button, and so on. Figure 12.6 shows `confirm` in action on iOS.


**Figure 12.6** Cordova `confirm` Dialog

Figure 12.7 shows the results of the `doContinue` function being executed, indicating that the No button was tapped in Figure 12.6.


**Figure 12.7** Showing `confirm` Results

### Prompt

A Cordova application often needs to collect information from the application user outside of a web form; Cordova provides the `prompt` method to accommodate this requirement. The method works just like the other methods discussed in this section and has the following method signature:

```
navigator.notification.prompt(message, callback, [title],
  [buttonLabels], [defaultText]);
```

The parameters in brackets are optional. To use `prompt` in your Cordova applications, make a call to the `prompt` method and provide the necessary callback function to process the user's input:

```
navigator.notification.prompt('Please enter your nickname',
```

```
  gotData, 'Nickname?', ['Cancel', 'OK'], 'Jimmy');

function gotData(res) {
  navigator.notification.alert('You chose option #' +
    res.buttonIndex + '\nYou entered: ' + res.input1, null,
    'Results', 'OK');
};
```

> **Note**
>
> Notice that `prompt` uses a different format for button labels than `confirm` uses. `prompt` expects an array of strings, as shown in the previous example, and `confirm` expects a single string with the button labels separated by a comma.

Figure 12.8 shows the example code in action on an Android emulator.

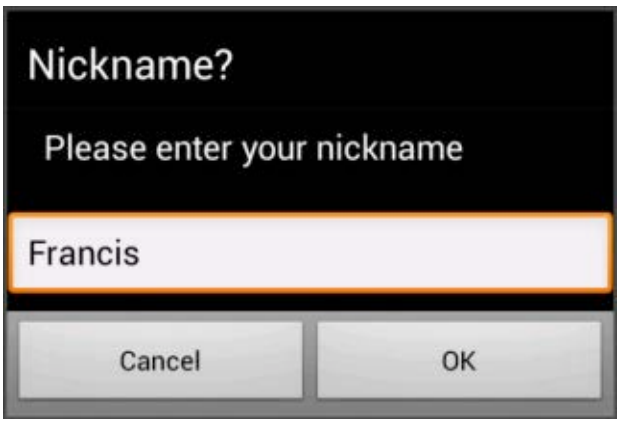



**Figure 12.8** Cordova `prompt` Dialog

Figure 12.9 shows the results displayed by `gotData` function.

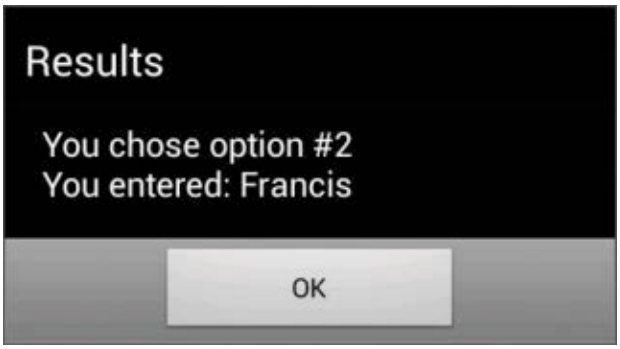



**Figure 12.9** Cordova `prompt` Results

# Cordova Events

The Cordova framework exposes a set of events a developer can use to react to certain things that happen on the device running the Cordova application. In some cases, the exposed events deal with hardware-related activities such as changes in the battery status or a physical button press by the user. In other cases, the exposed events deal with application status changes such as the application being paused or resumed. The purpose of these events is to expose to a web application the same device status events that are available to native applications.

The complete list of supported events is provided in Table 12.1.

**Table 12.1 Cordova Events**

| Cordova Event | Description |
|---|---|
| `backbutton` | Fired when the device's back button is pressed by the user. |
| `batterycritical` | Fired when the device battery reaches critical status. What is considered critical varies across mobile device platforms. |
| `batterylow` | Fired when the device battery reaches low status. What is considered low varies across mobile device platforms. |
| `batterystatus` | Fires when the battery status changes by at least 1% (up or down). |
| `deviceready` | Fires when the Cordova container has finished initialization and is ready to be used. |
| `endcallbuton` | Fires when the user presses the phone's end call button. |
| `menubutton` | Fires when the user presses the device's menu button. |
| `offline` | Fires when a device that has a network connection loses that connection. |
| `online` | Fires when a device goes online, when it switches from not having network connectivity to having network connectivity. |
| `pause` | Fires when the Cordova application is suspended. This typically happens when the device user switches to another application and the native OS pushes the current application to the background. |
| `resume` | Fires when a paused application is brought to the foreground. |
| `searchbutton` | Fires when the device user presses the search button. |
| `startcallbutton` | Fires when the device user presses the start call button. |
| `volumedownbutton` | Fires when the device user presses the volume decrease button. |
| `volumeupbutton` | Fires when the device user presses the volume increase button. |

### Regarding Buttons

Older devices had physical menu buttons and physical buttons to start and end phone calls. On newer devices, those buttons have been removed and replaced with virtual buttons that appear only when needed (like the menu and search buttons) or are specific to an application (like the phone buttons).

Most of the listed events are built into the Cordova container. Only battery status is implemented as a plugin. To enable an application to monitor battery events, you must first add the battery status plugin to your project by issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-battery-status.git
```

The `devicready` event was discussed at length in Chapter 5; to see examples of that event in action, refer to the HelloWorld applications highlighted in that chapter.

To monitor one of these events, you simply have to have an application register a listener for the particular event.

```
document.addEventListener("eventName", eventFunction);
```

As an example, to listen for the loss of network connectivity, you would register an `offline` event listener using the following code:

```
document.addEventListener("offline", isOffline);


function isOffline() {
  //Do whatever you want to do when the device goes offline

}
```

The `isOffline` function is called whenever the Cordova application that has a network connection detects that it has lost the network connection.

> **Warning**
>
> The events might not fire exactly when you expect them to on different mobile platforms. Be sure to test your application on different physical devices to make sure your application works as expected.

# Hardware APIs

Cordova exposes a few APIs that allow developers to build applications that interact with common hardware components found in most modern smartphones. These APIs help make Cordova applications feel more like native applications, as they let a Cordova application interact with the outside world in some way.

There's no specific grouping of these APIs in the Cordova documentation; bundling them together under the banner of hardware APIs is just my way of keeping things organized. The following APIs are discussed in this section:

- Accelerometer
- Camera
- Capture
- Compass
- Geolocation

The Accelerator, Compass, and Geolocation APIs all work in essentially the same manner; your application can measure the current value for the particular metric, or you can set up a watch that allows your application to monitor the particular metric as it changes over time. The Camera and Capture APIs both allow you to capture photographs using the device camera, but they operate differently, plus the Capture API allows you to record video and audio files as well.

The World Wide Web Consortium has worked on defining specifications for some of these capabilities. The Compass API is defined at http://dev.w3.org/2009/dap/system-info/compass.html, the Geolocation API specification is defined at http://dev.w3.org/geo/api/spec-source.html, and the Device Orientation specification is defined at http://dev.w3.org/geo/api/spec-source-orientation.

What you'll find is that some of the Cordova APIs align closely with the W3C specifications and others do not. For example, the Cordova Compass API has a `getCurrentHeading` method, while the W3C specification uses `getCurrentOrientation`. I assume that the Cordova APIs will align with the standards over time; you will need to monitor progress and update your applications accordingly.

In the sections that follow, I show you a little about how each of the APIs operates. Some of the APIs support a lot of options, so deep coverage is beyond the scope of this book; you can find complete examples of how to use each of these APIs in *PhoneGap Essentials* (www.phonegapessentials.com).

## Accelerometer

The Cordova Accelerometer API allows an application to determine the device's orientation in a three-dimensional space (using a three-dimensional Cartesian coordinate system). The Cordova documentation still says, "Captures device motion in the x, y, and z direction"; but that doesn't make sense to me, as it's not motion that the device is reporting, but orientation. If you hold the device steady in a particular orientation, the Accelerometer API will still report its orientation even though it's not moving.

To enable an application to use the Accelerometer API, you must first add the device motion plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device-motion.git
```

The API exposes three methods:

- `accelerometer.getCurrentAcceleration`
- `accelerometer.watchAcceleration`
- `accelerometer.clearWatch`

The `getCurrentAcceleration` method allows an application to query the device's current orientation. The `watchAcceleration` and `clearWatch` methods are used to allow an application to capture device orientation over a period of time, taking repeated measurements from the accelerometer at a specific time interval.

To measure the device's current orientation, you use something like the following in your application:

```
navigator.accelerometer.getCurrentAcceleration(onSuccess,
  onFailure);
```

In this example, the call to `getCurrentAcceleration` includes the names of two functions: `onSuccess` is executed when the orientation has been successfully measured, and `onFailure` is

executed when an error occurs; following are examples of some functions that can be used for this purpose:

```
function onSuccess(res) {
  x = res.x;
  y = res.y;
  z = res.z;
  var d = new Date(res.timestamp);
  timestamp = d.toLocaleString();
}

function onFailure() {
  alert('I have no idea why this failed, but it did.');
}
```

The `onSucess` function is passed an object that represents the different parts of the accelerator measurement. The X, Y, and Z values represent the device's orientation in a three-dimensional coordinate system, and the `timestamp` value indicates the date/time that the measurement was made. If you write those values out to an application's screen, you will see something similar to what is shown in Figure 12.10, captured on an Android emulator.


**Figure 12.10** Accelerator Measurement Results

On an Android device, with the device lying flat on a tabletop, the accelerometer will return approximately the following values: X:0, Y:0, Z:10. As the device is flipped so it's standing on its left edge, the values will adjust to approximately X:10, Y:0, Z:0. If you instead move the device so it's standing on its bottom edge, the values will adjust to approximately X:0, Y:10, Z:0. Standing the device on its top edge will result in approximate accelerometer values of X:0, Y:-10, Z:0. An application uses these values to determine how a user is holding the device and is most useful for games and interactive applications.

Unfortunately, Cordova doesn't tell you anything about any errors that occur when `onFailure` function is called and nothing is passed to it (an error code or error message), which can be used to identify the source of the error. So, it either works or it doesn't, but as the app is talking directly to a physical device API that doesn't have much complexity, if you try to determine the orientation and the call fails, it's most likely because the device doesn't have an accelerometer.

`getCurrentAcceleration` is useful if you want a quick check of a device's orientation before doing something within your application. If you want to monitor a device's orientation over time, in a game, for example, `getCurrentAcceleration` isn't that useful. To use it in a game, you would have to write code that checks the orientation periodically. To help developers in this situation, the Cordova API

allows a developer to periodically read the accelerometer by watching it using the `accelerometer.watchAcceleration` method.

To use `accelerometer.watchAcceleration`, an application sets up a watch using the following code:

```
var options = {frequency : 1000};
watchID = navigator.accelerometer.watchAcceleration(onSuccess,
  onFailure, options);
```

In this particular example, the code uses the same `onSuccess` and `onFailure` functions from the previous example. The `options` object defines the `frequency` of the accelerator measurement in milliseconds. So, to measure the current accelerometer values every 1 second, you would use 1000, as shown in the example. To measure every half second, use a `frequency` of 500.

In the code, the result of the call to `watchAcceleration` is assigned to a variable called `watchID`, which is used later to cancel the watch, as shown here:

```
navigator.accelerometer.clearWatch(watchID);
```

With all of this in place, the application will read the accelerometer every second and pass the values to the `onSuccess` function to process the results.

## Compass

The compass API allows a developer to read the mobile device's heading (using the device compass if available). The API works almost the same as the Accelerometer API; you can either query the heading value once or define a watch to periodically measure the heading value. The only differences between the two are in the `results` object, which is passed to the `onSuccess` function, and options that can be set when creating a watch.

To enable an application to read the heading, you must first add the device orientation plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device-
orientation.git
```

Like the Accelerator API, the Compass API exposes three methods:

- `compass.getCurrentHeading`
- `compass.watchHeading`
- `compass.clearWatch`

The `getCurrentHeading` method allows an application to query the compass's current orientation. The `watchHeading` and `clearWatch` methods are used to allow an application to capture compass headings over a period of time, taking repeated measurements from the compass at a specific time interval.

To measure the compass's orientation, you use something like the following in your application:

```
navigator.compass.getCurrentHeading(onSuccess, onFailure);
```

In this example, the call to `getCurrentHeading` includes the names of two functions: `onSuccess` is executed when the heading has been successfully measured, and `onFailure` is executed when an error occurs; following are examples of some functions that can be used for this purpose:

```
function onSuccess(res) {
  magneticHeading = res.magneticHeading;
  trueHeading =res. res.trueHeading;
  headingAccuracy = res.headingAccuracy
  var d = new Date(res.timestamp);
  timestamp = d.toLocaleString();
}

function onFailure(err) {
  alert("Error: " + err.code);
}
```

The `heading` object (`res` in the example) returned to the `onSuccess` function has the properties described in Table 12.2.

**Table 12.2 Compass Results Values**

| Property | Description |
|---|---|
| `magneticHeading` | The compass heading in degrees from 0 to 359. |
| `trueHeading` | The compass heading relative to the geographic North Pole in degrees from 0 to 259. A negative value indicates that the true heading cannot be determined. |
| `headingAccuracy` | The difference in degrees between the magnetic heading and the true heading values. |
| `timestamp` | The date and time that the measurement was made (number of milliseconds since midnight January 1, 1970) |

When an error occurs, the `onFailure` function is passed an error code, which can be queried to determine the cause of the error. Possible values are `CompassError.COMPASS_INTERNAL_ERR` and `CompassError.COMPASS_NOT_SUPPORTED`.

To use `compass.watchHeading`, an application sets up a watch using the following code:

```
var options = {frequency : 1000};
watchID = navigator.compass.watchHeading(onSuccess, onFailure,
  options);
```

In this particular example, the code uses the same `onSuccess` and `onFailure` functions from the previous example. The `options` object defines the `frequency` of the accelerator measurement in milliseconds. So, to measure the current heading values every 1 second, you would use 1000, as shown in the example. To measure every half second, use a `frequency` of 500. You can also specify a `filter` value, which defines a minimum degree value change, which must occur before the watch is fired. Because compass values fluctuate pretty rapidly, you will want to set a `filter` to reduce the number of times heading measurement is made (and returned to your program) so your program can respond only to more dramatic changes in heading.

In the code, the result of the call to `watchHeading` is assigned to a variable called `watchID`, which is used later to cancel the watch, as shown in here:

```
navigator.compass.clearWatch(watchID);
```

With all of this in place, the application will read the compass every second and pass the values to the `onSuccess` function to process the results. To see a complete application using the Compass API, refer to Chapter 14.

## Geolocation

The Cordova Geolocation API allows an application to determine the physical location of the device running an application. The API is based on the W3C's Geolocation API and works almost the same as the Accelerometer and Compass APIs; you can either query the location once or define a watch to periodically calculate the location. The only differences between them are in the `results` object, which is passed to the `onSuccess` function, and options that can be set when creating a watch.

To enable an application to determine the device's location, you must first add the Geolocation plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

The Geolocation API exposes three methods:

- `compass.getCurrentPosition`
- `compass.watchPosition`
- `compass.clearWatch`

The `getCurrentPosition` method allows an application to determine the device's current location. The `watchPosition` and `clearWatch` methods are used to allow an application to periodically calculate the device's location over a period of time, making repeated calculations at a specific time interval.

When the Geolocation API returns a location object, the object exposes `coordinates` and `timestamp` properties. The `timestamp` property contains the date and time that the measurement was made in number of milliseconds since midnight January 1, 1970. The `coordinates` property is another object that includes the properties described in Table 12.3.

**Table 12.3 Coordinates Properties**

| Property | Description |
|---|---|
| `accuracy` | The accuracy of the latitude and longitude coordinates in meters. |
| `altitude` | The device's height in meters above the ellipsoid (http://tinyurl.com/nje9d2o). |
| `altitudeAccuracy` | The accuracy of the altitude coordinate in meters. |
| `heading` | The device heading (direction of travel) in degrees. |
| `latitude` | The latitude portion of the location in decimal degrees. |
| `longitude` | The longitudinal portion of the location in decimal degrees. |
| `speed` | The device's current speed in meters per second. |

Refer to the Accelerator and Compass sections for details on how this API works.

## Camera

The Cordova framework provides two APIs for working with the device camera. One is the Camera API, which provides developers with direct access to the native camera APIs, and the other is the Media Capture API described in the next section. The difference between the two options is that the Camera API exposes only the ability to take pictures with the camera, while the Media Capture API provides an interface to the camera that includes photos as well as videos plus the ability to record audio files. In this section, I show you how to use the Camera API to capture pictures from a Cordova application; refer to the section "Capturing Media Files" for information on the other options.

To enable an application to take photos using the Camera API, you must first add the Camera plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-camera.git
```

From a programming standpoint, getting a Cordova application to take a picture is pretty simple; all you have to do is have the program call the Camera API using the following code:

```
navigator.camera.getPicture(onCameraSuccess, onCameraError);
```

To show how this works, I created a simple camera application that makes that call to `getPicture` then displays the data returned from the camera. You can see a screenshot of the application in Figure 12.11.

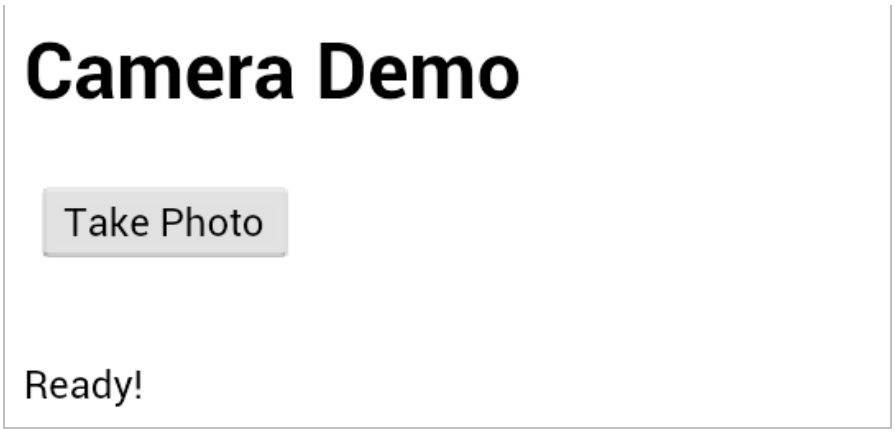



**Figure 12.11** Camera Demo Application

As soon as you click the Take Photo button, the device's native camera application opens, as shown in Figure 12.12. Manipulate the camera and image any way you want, then take the picture; in this example, you would click the camera aperture image on the right. What you will see here will vary depending on the mobile device platform the on which the application is running.

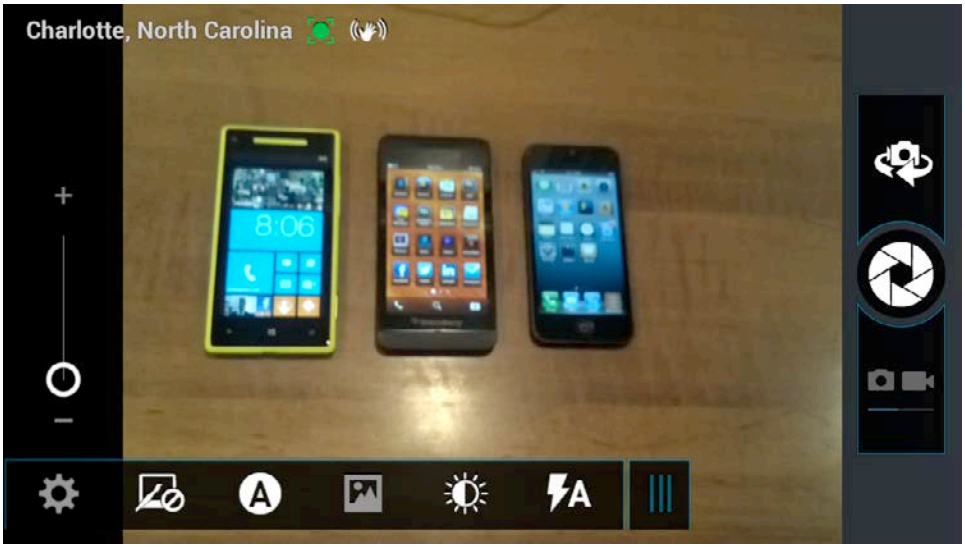



**Figure 12.12** Device Camera Application: Confirming Photo

Notice in Figure 12.12 that there isn't a way to cancel taking the photo. Even if you decide not to capture a photo, you will have to snap one here; then you can cancel it in the next step.

Depending on the mobile device, you may be prompted to approve the photo, as shown in Figure 12.13. In this example, you can tap the checkmark to select the photo and return to the Cordova application; tap the X in the bottom right to cancel, returning the photo information to the Cordova application; or click the camera icon on the lower left to discard this photo and take another one. No, I'm not sure why the photo is rotated in the screen shot.

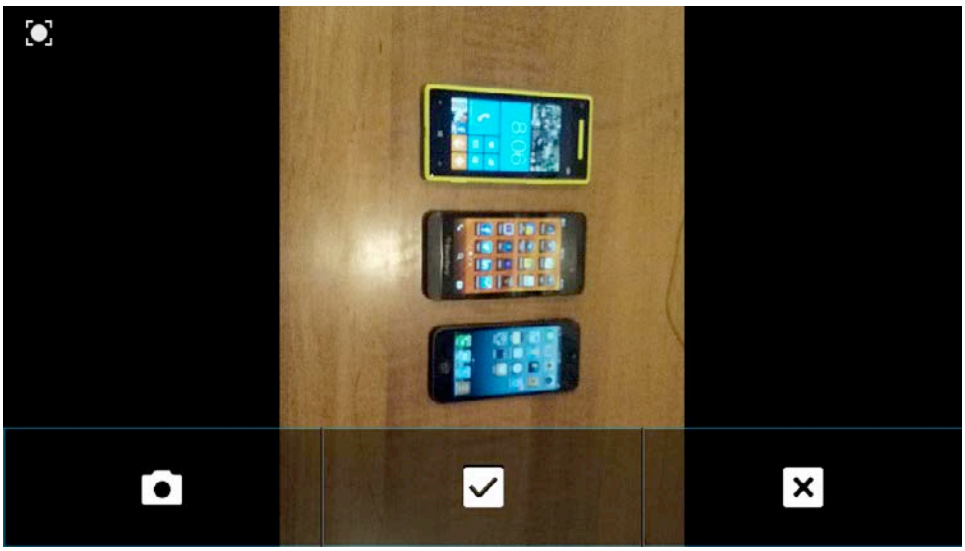


**Figure 12.13** Device Camera Application: Approving a Photo

When you accept a photo by tapping the checkmark icon, the device camera application will close and return information about the selected photo to the Cordova application, as shown in Figure 12.14. In this case, since I didn't tell `getPicture` anything about how to take the picture or what to do with it, the API used its default settings and simply returned a file URI pointing to the image file. At this point, the Cordova application can access the file using the URI it received from the API and render the image on the screen, upload it to a server, or do anything else it wants.

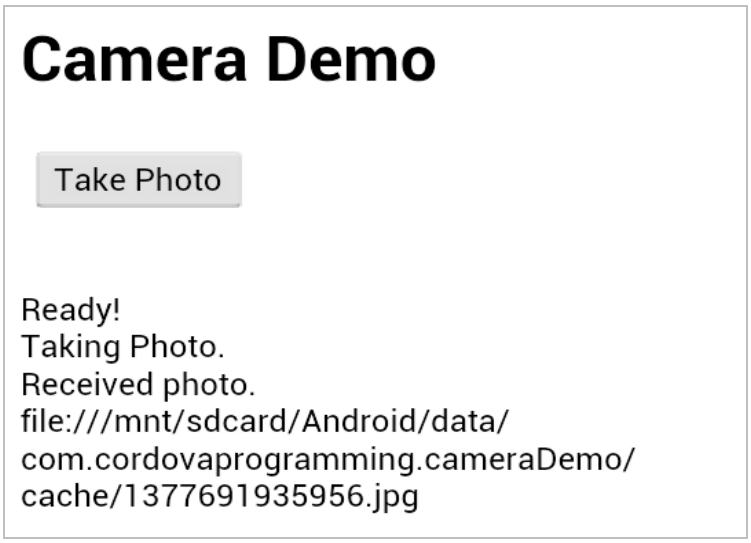



**Figure 12.14** Camera Demo Application: Photo Results

If you choose to cancel the photo you've taken, the Camera API will return an error message, "Camera Cancelled," to the Cordova application.

Now that's not all there is to the Camera API—I was just showing you the API's default behavior. You can also call `getPicture` and pass in an options object, which tells the API what to do and how to do it. Here's an alternative way to call `getPicture`:

```
navigator.camera.getPicture(onCameraSuccess, onCameraError,
  cameraOptions);
```

The `cameraOptions` shown in the example is a JavaScript object, which can look like the following example from the Cordova documentation:

```
var options = {
  quality : 75,
  destinationType : Camera.DestinationType.DATA_URL,
  sourceType : Camera.PictureSourceType.CAMERA,
  allowEdit : true,
  encodingType: Camera.EncodingType.JPEG,
  targetWidth: 100,
  targetHeight: 100,
  popoverOptions: CameraPopoverOptions,
  saveToPhotoAlbum: false
};
```

> **Warning**
> Different platforms ignore some of these properties, so be sure to check the quirks section in the Cordova documentation before using them in your applications.

A developer will use some or all of these properties to control the picture capture process using the Camera API. Each of the possible options is described in Table 12.4.

**Table 12.4 Camera Options**

| Property | Description |
|---|---|
| `allowEdit` | Boolean value: Specifies whether the photo can be edited by the user before being returned to the Cordova application. Not all mobile device platforms support this option, nor will many users expect it depending on your application. |
| `cameraDirection` | Numeric value: Directs the API on which camera (front or back) to use to take the photo. Use `navigator.camera.Direction.FRONT` for front-facing camera (screen side) and, wait for it, `navigator.camera.Direction.BACK` to use the camera on the back of the device. |
| `correctOrientation` | Boolean value: Instructs the API to rotate the image to correct for the orientation of the device when the picture is taken. |
| `destinationType` | Numeric value: Specifies how the API will return the captured image. Options are `Camera.DestinationType.FILE_URI`, which is the default option and illustrated in the earlier example; `Camera.DestinationType.DATA_URL`, which returns the image as a base64-encoded string; and `Camera.DestinationType.NATIVE_URI`, which returns a native URI for the image file.<br>Be careful using `DATA_URL` because JavaScript simply isn't capable of dealing with the image as an encoded string. Your app may crash if a high-resolution image is returned as a string to your program. |
| `encodingType` | Numeric value: Specifies the output format for the photo. Specify `Camera.EncodingType.JPEG` to have the API return a JPEG image or `Camera.EncodingType.PNG` to have the API return a PNG file. |
| `mediaType` | Numeric value: When `SourceType` is set to `PHOTOLIBRARY` or `SAVEDPHOTOALBUM`, this property specifies what type of files can be selected by the application user. Use `Camera.MediaType.PICTURE` to allow selection of photos only, `Camera.MediaType.VIDEO` to allow video files to be selected, and `Camera.MediaType.ALLMEDIA` to allow any supported media file to be selected.<br>When VIDEO is selected, the API will only return a File URI; otherwise, the API will return the image information in the format specified by the `destinationType` property. |

| Property | Description |
|---|---|
| `quality` | Numeric value: Used to control the quality of the captured image as a percentage of image quality from 0 to 100%. A value of 100 refers to full image quality with no compression.<br>One of the reasons you would reduce the quality is to reduce the overall file size for the captured photo to make it easier to manipulate within your application. |
| `saveToPhotoAlbum` | Boolean value: Instructs the API to save images to the device's photo album after capture. |
| `sourceType` | Numeric value: Specifies where the image should come from. Possible values are `Camera.PictureSourceType.CAMERA` (default), `Camera.PictureSourceType.PHOTOLIBRARY`, and `Camera.PictureSourceType.SAVEDPHOTOALBUM`. The behavior of this option will vary depending on the mobile device the application is running on, as some devices do not expose photo libraries or photo albums. |
| `targetHeight` | Numeric value: Used to specify the target height of the captured image (in pixels). This is a weird one; see the note that follows the table for more information. |
| `targetWidth` | Numeric value: Used to specify the target width of the captured image (in pixels). This is a weird one; see the note that follows the table for more information. |

> **Note**
>
> The camera options `targetHeight` and `targetWidth` properties have always perplexed me; when I wrote *PhoneGap Essentials* more than a year and a half ago I wrote the following:
>
> > The targetHeight & targetWidth parameters are supposed to control the height and width of the image obtained using getPicture. In my testing though, the parameters did not affect the resulting picture. The documentation says as well that the parameters must be used in conjunction with each other and that aspect ratio is maintained. This further reinforces that these options cannot work as documented (which my testing has proven) since it doesn't make sense that you have to set both height and width while at the same time maintaining an aspect ratio for the picture. If it truly was maintaining aspect ratio, then I'd expect that only one of the values would be able to be set.
>
> As I wrote this section, I posted a question on the Cordova dev list, and from the responses I got back, it was clear that it wasn't known how this should work. So, I committed to retest all of this and post the results. I'll either update the Cordova documentation so it makes more sense or the Cordova developers will take a look at making this work more as expected based on what I learn. Stay tuned.

Since the camera is a separate, independent application and an application user could take several photos before finally getting the one he or she wants to return to the Cordova application, there could be quite a few photos left lying around. In *PhoneGap Essentials*, I described how a developer could go into the file system and clean up the orphan photos manually. Since then, the Cordova team has added a

`cleanup` method, which can be used to clean up the orphaned photos. To use this method, call the method and pass in the names of success and failure callback functions, as shown in the following example:

```
navigator.camera.cleanup(onCameraCleanupSuccess,
  onCameraCleanupError);
```

Unfortunately, the method is currently supported only on iOS. When you try to perform the cleanup on an Android device, you will receive an "Invalid action" error message.

## Capturing Media Files

The Capture API is like the Camera API in that you can use it to capture photographs, but you can also use it to capture video and audio files. The API was originally based on the W3C Media Capture API (www.w3.org/TR/media-capture-api), but at the time, the PhoneGap team didn't or weren't able to implement some of the APIs features. The W3C stopped work on that specification as the Device API Working group focused instead on the Media Capture and Streams API (http://dev.w3.org/2011/webrtc/editor/getusermedia.html), which isn't like the Capture API at all.

To enable an application to use the Capture API, you must first add the Media Capture plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-media-capture.git
```

The API exposes the following methods:

- `capture.captureAudio`
- `capture.captureImage`
- `capture.captureVideo`
- `MediaFile.getFormatData`

The first three work exactly the same—I'll show you how in a minute. The `getFormatData` is supposed to allow you to retrieve information about a media file, but because of limitations on mobile devices, very little or no information is available through this method.

To use the Capture API is pretty simple: you make a call to one of the three capture methods (audio, image, video) using the following method signature:

```
navigator.device.capture.captureAudio(onSuccess, onFailure,
  options);
```

Like many of the other APIs discussed in this chapter, the `onSuccess` and `onFailure` functions are called after the capture has completed successfully or when it fails.

The `onSuccess` function is passed a `fileList` object, which can be iterated through to access the path pointing to each captured file, as shown in the following example:

```
function onSuccess(fileList) {
  var len, I, path;
  //See how many files are listed in the array
  len = fileList.length;
  //Make sure we had a result; it should always be
```

```
  //at least greater than 0, but you never know!
  if(len > 0) {
    //Media files were captured, so let's process them...
    for( i = 0, len; i < len; i += 1) {
      //Get the path to the file
      path = fileList[i].fullPath;
     //Do something with the file here

    }
  } else {
    //This will probably never execute
    alert("Error: No files returned.");
  }
}
```

Once you have the path to each media file, you can upload the file to a server, play or display it within the app, and more.

When called, the `onFailure` function will be passed an `error` object, which can be queried to determine the error code, as shown in the following example:

```
var onError = function(error) {
  alert('Capture error: ' + error.code);
};
```

The possible error codes are

- `CaptureError.CAPTURE_INTERNAL_ERR`
- `CaptureError.CAPTURE_APPLICATION_BUSY`
- `CaptureError.CAPTURE_INVALID_ARGUMENT`
- `CaptureError.CAPTURE_NO_MEDIA_FILES`
- `CaptureError.CAPTURE_NOT_SUPPORTED`

The optional `options` parameter controls how many media files are captured and, for audio captures, a `duration` property dictating the length of the audio capture.

```
var options = { limit: 3, duration: 10 };
```

Some platforms ignore some options, so be sure to check the quirks section of the Cordova Capture API documentation and test on an appropriate sampling of devices to make sure you understand how this API works.

# Globalization

Many mobile applications target audiences who speak and read different languages. If you create a popular Cordova app, it probably won't be long before you need to make it available in multiple languages. To make the globalization of a mobile application easier for developers, the Cordova team added a Globalization API that allows an application to query the OS for locale settings. Developers can use this API to determine the user's preferred language, then load the content in the appropriate

language, and can also use methods in the API to better understand how to display dates, times, numbers, and currency appropriately for the user's preferred language.

To enable an application to leverage this API, you must first add the Globalization plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-globalization.git
```

The `globalization` object's methods all work in a similar manner; like most of the Cordova APIs, they're asynchronous, so you call the method and pass in success and failure functions, as shown in the following example:

```
navigator.globalization.getPreferredLanguage(onGPLSuccess, onGPLFailure);
```

The success function is passed an object your application can query to access the value or values that are returned from the method. For most generic methods, such as the preceding `getPreferredLanguage` method, the method returns a string value that can be passed as shown in the following example:

```
function onGPLSuccess(lang) {
  alert("Preferred language: " + lang.value);
}
```

In this example, when the call to `getPreferredLanguage` is made, the `onGPLSuccess` function executes and displays the dialog shown in Figure 12.15.


**Figure 12.15** Results of `getPreferredLanguage`

Finishing out the `getPreferredLanguage` example, the failure function is passed an `error` object, which can be queried to determine an error `code` and an error `message`, as shown in the following example:

```
function onGPLFailure(err) {
  alert("Error: " + err.code + " - " + err.message);
}
```

The possible error codes are

- `GlobalizationError.UNKNOWN_ERROR`
- `GlobalizationError.FORMATTING_ERROR`
- `GlobalizationError.PARSING_ERROR`
- `GlobalizationError.PATTERN_ERROR`

Table 12.5 lists all of the Globalization API methods as well as the parameters passed to the method and the results that are returned.

**Table 12.5 Globalization Methods and Parameters**

| Method | Accepts | Returns |
|---|---|---|
| `dateToString` | JavaScript Date value, options | String value representing the date formatted based on options and user's current language settings. |
| `getCurrencyPattern` | Currency Code | Pattern object that describes currency format and components of a currency value based on the user's current language settings. |
| `getDateNames` | Options | An array of month or day names, narrow or wide versions, depending on options and user's current language settings. |
| `getDatePattern` | Options | Pattern object that describes the format of a date value based on the user's current language settings. |
| `getFirstDayOfWeek` | | Numeric value indicating the first day of the week based on the user's current calendar settings. |
| `getLocaleName` | | String representation of the user's current locale as an ISO 3166 country code (http://tinyurl.com/7xhpa9m). |
| `getNumberPattern` | Options | Pattern object that describes the format of a numeric value based on the user's current language settings. |
| `getPreferredLanguage` | | String representation of the user's preferred language as an ISO 639-1 two-letter code (http://tinyurl.com/l55ttxf). |
| `isDayLightSavingsTime` | Date | String value indicating whether or not Daylight Savings Time is in effect. |
| `numberToString` | Number, Options | String value representing the number formatted using options and user preferences. |
| `stringToDate` | String, Options | Parses a date string into individual components based on options and user's preferences. |
| `stringToNumber` | String, Options | Parses a number string into individual components based on options and user's preferences. |

As you can see from the table, some of the methods accept a properties object, which allows a developer to control how a method operates. For example, to use the `dateToString` method to convert a date object into a string, a Cordova application would do something similar to the following:

```
var d = new Date();
navigator.globalization.dateToString(d, onSuccess, onFailure);
```

The `onSuccess` function is called after the date conversation has completed and is passed an object that can be queried to display the result, as shown in the following example.

```
function onSuccess(res) {
  alert("Result: " + res.value);
};
```

When the application runs on the on the Android platform, it displays a result similar to the one shown in Figure 12.16.

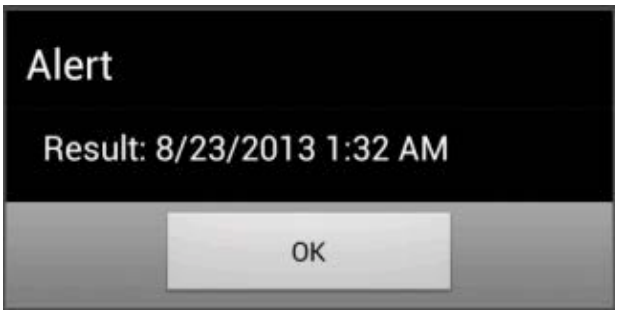



**Figure 12.16** Displaying Results from `dateToString` Using Default Options

The `dateToString` method supports an `options` parameter, which a developer can use to change the format of the output, as shown in the following example.

```
var d = new Date();
var options = {
  formatLength : 'short',
  selector : 'date'
};
navigator.globalization.dateToString(d, onSuccess, onFailure,
  options);
```

In this example, the `options` object is specifying `formatLength` and `selector` properties, which are used to control how long the resulting string is and whether it should include date and/or time values (in this case, I'm asking for only date). When the application runs, you will see a dialog similar to the one shown in Figure 12.17.

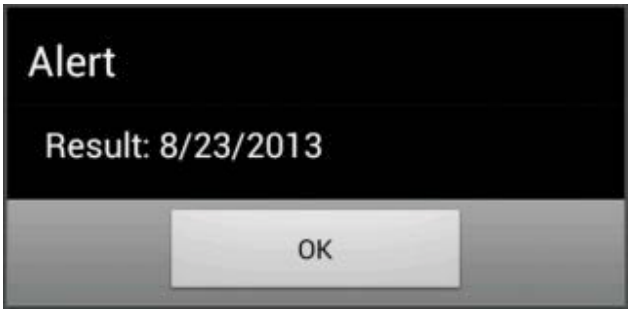



**Figure 12.17** Displaying Results from `dateToString` with Options (Short Date)

You can use an `options` object to specify `long` format, as shown in the following example:

```
var options = {
  formatLength : 'long',
  selector : 'date
}
```

Using long date format changes the application's output to that shown in Figure 12.18.

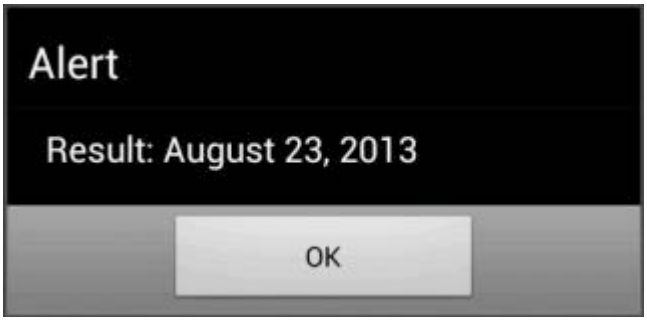



**Figure 12.18** Displaying Results from `dateToString` with Options (Long Date)

### Warning

In my testing, on iOS you get the same results no matter what you pass in for options.

In this example, the success function is passed an object with only a single property: `value`, but several methods will return an object with multiple properties. The `stringToDate` method, for example, returns an object with separate properties for each date component, as shown here:

```
{
  "month":7,
  "second":0,
  "millisecond":0,
```

```
  "day":31,
  "year":2013,
  "hour":10,
  "minute":47
}
```

When you use the Globalization API in your applications, refer to Table 12.5 and the corresponding Cordova documentation to understand exactly what each method accepts and what is returned.

# Working with the Contacts Application

The Cordova Contacts API allows a developer to build an application that interacts with the address book or contacts application on a mobile device. The Cordova Contacts API is based on the W3C Contacts API (www.w3.org/TR/2010/WD-contacts-api-20100121). You would use this API if you wanted to build an application that reads from the contacts list and uses contact data within the application or uses data from within the application to write a new contact to the contacts list.

To enable an application to access a device's contact list, you must first add the Contacts plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-contacts.git
```

The Contacts API is sometimes challenging to use because the contacts capabilities available on each mobile platform differ; so some of the contact fields you might use on an Android device will differ from what you would use on iOS. Additionally, the implementation of the Contacts is a little different than what we've seen with the other APIs.

The Contacts API essentially exposes two methods and a `contacts` object. The methods are used to create new `contacts` objects and search for contacts on a device, and the `contacts` object is the representation of a contact record on the device.

To create a new contact, an application makes a call to the API's `create` method, as shown in the following example:

```
var newContact = navigator.contacts.create();
```

Unlike the other API methods we've discussed so far, this particular call is synchronous, so instead of having to provide success and failure callbacks, this operation happens immediately. However, this method call doesn't actually create a contact in the device's contacts application; instead, all it does is create a new `contact` object, nothing more. In this example, it's creating a new, empty `contact` object called `newContact`; nothing is saved to the contacts application until you call the object's `save` method, which I describe a little later.

You can also populate the `contact` object during the creation process by passing in a `contact` object to the `create` method, as shown in the following example:

```
var newContact = navigator.contacts.create(
  {"displayName": "John M. Wargo"});
```

In this example, I'm populating the `contact` object's `displayName` property when I create the new `contact` object.

The `contact` object consists of the following properties; you can set some or all of these properties for a contact:

- `addresses`: An array containing of all the contact's different addresses
- `birthday`: The contact's birthday
- `categories`: An array containing all of the user-defined categories associated with the contact
- `displayName`: The display name for the contact
- `emails`: An array containing all of the email addresses for the contact
- `id`: A globally unique identifier for the contact
- `ims`: An array containing all of the contact's instant messaging addresses
- `name`: An object representing each of the components of the contact's name
- `nickname`: The nickname for the contact
- `note`: Any notes associated with the contact
- `organizations`: An array containing all of the organizations associated with the contact
- `phoneNumbers`: An array containing all of the phone numbers associated with the contact
- `photos`: An array containing images associated with the contact
- `urls`: An array containing the web pages associated with the contact

Notice that some of the properties are arrays of other properties. A contact can typically have two or more mailing addresses, home and work at a minimum. Additionally, many contacts have more than one email address, so the `contact` object has to be able to accommodate a dynamic number of properties.

> **Note**
>
> I could spend about 10 pages here describing all of the different arrays that are associated with the `contact` object, but that's really beyond the scope of this chapter. Instead, let me show you how to save the contact, then I'll show you how to search for contacts and dump the contact information to the console so you can more easily see how the `contact` object looks in the real world on different devices.
>
> There are some inconsistencies in how different mobile devices store contact information, so seeing this in real-world scenarios is better anyway.

You can populate the `contact` object when you create the object, as shown in the earlier example, or you can create the `contact` object, then populate the object's properties, as shown in the following example:

```
var newContact = navigator.contacts.create();
//Populate the contact object with values
var fullName = "John M. Wargo";
newContact.displayName = fullName
newContact.nickname = "John";
```

```
//Populate the Contact's Name entries
var tmpName = new ContactName();
tmpName.givenName = "John";
tmpName.familyName ="Wargo";
tmpName.formatted = fullName;
//Then add the name object to the contact object
newContact.name = tmpName;
```

In this example, I've created a new `contact` object, then started populating it with values. When it comes to populating the contact's name information, the code populates a `ContactName` object (defined within the Contacts API), then adds it to the `newContact` object. The `ContactName` object includes the following properties:

- `familyName`
- `formatted`
- `givenName`
- `honorificPrefix`
- `honorificSuffix`
- `middleName`

There are many different object types and arrays of objects that can be added to a contact record; I've shown only a small example of what can be done. Be sure to use the Contacts API documentation for details on all of the supported options.

Once you have all of the `contact` object properties set, you must call the `contact` object's `save` method to write the changes to the actual contact record:

```
newContact.save(onSuccess, onError);
```

The `save` method accepts the typical success and failure functions that you've seen with most of the other Cordova APIs. The failure function is passed an `error` object you can use to identify the cause of an error and respond accordingly, as shown in the following example:

```
function onError(err) {
  console.log("Error Saving Contact: " + err.code);
};
```

To manipulate an existing contact, you can use the Contacts API `find` method to locate the record, as shown in the following example:

```
navigator.contacts.find(contactFields, onSuccess, onError, options);
```

In this example, the `contactFields` object represents an array of field names, as shown in the following example:

```
var contactFields = ["displayName", "name", "phoneNumbers", "emails", "addresses"]
```

The `find` method defines which field values are returned in the search results; it does not define which fields are searched. Not what you expected, right? Me too!

The `options` object defines parameters around how the search is performed; a sample `options` object is shown here:

```
var options = {filter: "Wargo", multiple: true};
```

The `filter` property is used to provide the `find` method with the search string to use when searching records. The `multiple` property is a Boolean value that controls whether only a single (`false`) contact is returned or multiple (`true`) contacts are returned.

Let's take a look at a complete example. The following code sample shows how to call `find` and pass in a list of contact fields and search options. In the `onSuccess` function, the code simply writes the contact details to the console.

```
function findContact() {
  var contactFields = ["displayName", "name", "phoneNumbers", "emails",
    "addresses"];
  var contactOptions = {
    filter : "Wargo",
    multiple : true
  };
  navigator.contacts.find(contactFields, onSuccess, onError,
    contactOptions);
}

function onSuccess(contacts) {
  for (var i = 0; i < contacts.length; i++) {
    console.log("Contact[" + i + "]: " + JSON.stringify(contacts[i]));
  }
}
```

Remember that I mentioned that there was a way to see how every aspect of a contact record was structured? Well, the following chuck of JSON is the result (slightly modified to hide my real identity) of the previous example code running against the contacts database on my personal Android device. You can run the application, search for a particular contact, then grab the JSON text returned from the call to `find` and analyze it to see how to populate these fields within your application.

```
{"id":"1370", "rawId":"109", "displayName":"Wargo, John M.", "name":{"middleName":"M.",
"familyName":"Wargo", "formatted":"John M. Wargo", "givenName":"John"}, "nickname":"John",
"phoneNumbers":[{"type":"mobile", "value":"(555) 555-3333", "id":"58", "pref":false},
{"type":"work", "value":"(555) 555-4444", "id":"59", "pref":false}, {"type":"home",
"value":"(555) 555-6666", "id":"12860", "pref":false}], "emails":[{"type":"custom",
"value":john@somedomain.com ,"id":"901", "pref":false}],
"addresses":[{"region":"CA","streetAddress":"99 Cordova lane ", "id":"902", "formatted":"99
Cordova Lane\nSan Francisco, CA 99215\nUnited States of America", "postalCode":"99215",
"locality":"San Francisco", "type":"home", "pref":false, "country":"United States of
America"}, {"region":"CA", "streetAddress":"One Sybase Drive", "id":"12861",
"formatted":"One Sybase Drive\nDublin, CA 94568\nUnited States of America",
"postalCode":"94568", "locality":"Dublin", "type":"work", "pref":false, "country":"United
States of America"}], "ims":null, "organizations":null, "birthday":null, "note":null,
"photos":null, "categories":null, "urls":"www.johnwargo.com"}
```

You can see, for example, how phone numbers are managed within the contacts application in this example:

```
"phoneNumbers":[
  {"type":"mobile", "value":"(555) 555-3333", "id":"58", "pref":false},
  {"type":"work", "value":"(555) 555-4444", "id":"59", "pref":false},
  {"type":"home", "value":"(555) 555-6666", "id":"12860", "pref":false}
]
```

Each phone number has a specific type, value, ID, and preferred status value. ID should be created automatically when adding the phone number to the contact record.

Different mobile device platforms manage contact data differently, so be sure to test the contact format on each mobile device you will be supporting—you might have to deal with each platform differently.

Once you have a `contact` object returned from the call to `find`, you can change the properties of the object and write the changes back to the contacts application using the `save` method discussed earlier. To remove the contact, first get a handle to the contact object and make a call to `remove`:

```
foundContact.remove(onSuccess, onFailure);
```

There's a lot more to the Contacts API. I've only touched the surface—be sure to leverage the Cordova documentation for more information.

# Playing/Recording Media Files

The Cordova APIs include a Media API an application can use to record and play media files. This is the API you would use to play audio files in the background of a smartphone or tablet video game, for example.

To enable an application to work with media files, you must first add the media plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-media.git
```

The Media API is like most of the other Cordova APIs in that the API's methods are asynchronous, but what triggers the callback functions is a little different. To use this API, an application starts by creating a `Media` object using code similar to the following:

```
var mediaObj = new Media(srcFile, onSuccess, onError, onStatus);
```

What this code does is create a `mediaObj` object that points to the media file specified in the `srcFile` parameter shown in the example. The application doesn't open or connect to the file yet; it merely creates an object that refers to the file, nothing more. Some methods I'll show you in a little while are used to actually play the file.

The `onSuccess` and `onFailure` functions shown in the example are the success and failure callback functions you should be familiar with by now, but they don't fire when you might expect. Since the code I've shown is only creating an object, there are no real callback functions that need to be executed as part of that process. The `onSuccess`, `onFailure`, and `onStatus` callback functions shown in the example are actually called when any of the following methods are called against the media object just created:

- `getCurrentPosition`
- `getDuration`
- `pause`
- `play`
- `release`
- `seekTo`

- setVolume
- startRecord
- stop
- stopRecord

So, to play a media file called soundtrack.mp3, an application would execute the following code:

```
srcFile = 'soundtrack.mp3';
var mediaObj = new Media(srcFile, onSuccess, onError, onStatus);
mediaObj.play();

function onSuccess() {
  console.log("Media: Success");
}

function onError(error) {
  alert('Media Error: ' + error.code + ': ' + error.message);
}

function onStatus(statCode) {
  console.log("Media Status: " + statCode);
}
```

To stop a media file from playing, simply call `mediaObj.stop();`.

As you can see in the example, the `onStatus` function is passed a status code parameter that allows an application to understand what's currently going on with media playback or recording. The possible status codes are

- Media.MEDIA_NONE
- Media.MEDIA_STARTING
- Media.MEDIA_RUNNING
- Media.MEDIA_PAUSED
- Media.MEDIA_STOPPED

With all of the methods and callbacks available with the Media API, you can build a complete media player application, or you can simply load up an audio file and play it without any UI; the API provides the flexibility a developer needs to do either.

> **Warning**
>
> The location where a Cordova application stores the media files packaged with the application varies across the different mobile device platforms. Android files are located in the /android_asset folder, whereas on iOS, files are located within the root of the application's file area.

# InAppBrowser

The InAppBrowser is a more recent edition to the Cordova APIs. It allows a web application to load content into a separate window. Originally created as independent Cordova plugins called ChildBrowser for Android and iOS, it was added to the Cordova project as InAppBrowser, then expanded to support other mobile device platforms.

Say, for example, that you want to show your application users additional web content. You could easily load additional content within your application and manage transitions to and from the content within your application, but sometimes you want a different experience for your users. You can also load the content in the system browser, but on iOS, for example, the user would have to perform manual steps to navigate back to your application after looking at the content in the browser. InAppBrowser loads web content in such a way that your application users can more easily return directly to the main application.

To enable an application to use the InAppBrowser, you must first add the InAppBrowser plugin to your project by opening a terminal window and issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-inappbrowser.git
```

## Loading Content

With the plugin added to your project, you can now open web content in a window using the following code:

```
var ref = window.open('http://www.johnwargo.com', '_blank',
  'location=yes');
```

In this example, the application will open my personal website and return an object that represents the browser window. You can later use the returned object to interact with the browser window.

You could also create the browser window, but not display it, by using the following code:

```
var ref = window.open('http://www.johnwargo.com', '_blank',
  'hidden=yes');
```

Later on, when you're ready to display the browser window, you can open it using:

```
ref.show();
```

Notice that there are no callback functions that we're used to seeing with the Cordova APIs.

Of the parameters passed to the call to `window.open`, the `_blank` tells the application to open the content in its own window; you could also use `_self`, which tells it to open the page within the current window, and `_system`, which tells it to open the content in the system's default web browser.

The problem with a target of `_self` is that the page that is being loaded replaces the current web content for the application. For application users, this means that there's no going back—there isn't an iOS button to use for that, anyway, and on Android, the escape button won't take you back either.

The `'location=yes'` tells the InAppBrowser to display the page location within the browser window. There are several other options that can be used when loading a page, but the options vary depending on the mobile device platform; refer to the Cordova documentation for more information.

If you run the application on an Android device, you will see a screen similar to the one shown in Figure 12.19; I've cropped the screen to show only the top portion of the window. In this case, the

browser window opens with an address bar a user can use to navigate to other sites as well. The user should tap the Done button to return to the original program.


**Figure 12.19** InAppBrowser Opening an External Web Page with an Address Bar on Android

When the same application is executed on an iOS device, the user will see the web page, and at the bottom, iOS will display the page address (without the ability to manipulate it or navigate to other sites) plus the Done button to close the window, as shown in Figure 12.20.


**Figure 12.20** InAppBrowser Opening an External Web Page with an Address Bar on iOS

Figure 12.20 highlights one of the key benefits of using InAppBrowser on an iOS device. When done reading the content, all the user has to do here is tap the Done button to return to the main application. If the page were loaded by spawning the system browser to open the page, the user would have to double-tap the device's Home button to bring up a list of running applications, then tap on the application name to return to the application—not the best user experience.

To turn off display of the page address, open a page with the following code:

```
var ref = window.open('http://www.johnwargo.com', '_blank',
  'location=no');
```

On Android, the page will load like in the previous examples but will not display the address bar, as shown in Figure 12.21.


**Figure 12.21** InAppBrowser Opening an External Web Page without an Address Bar on Android

On iOS, the Done button remains, but the page address is removed, as shown in Figure 12.22.


**Figure 12.22** InAppBrowser Opening an External Web Page without an Address Bar on iOS

You can also use InAppBrowser to load local content, as shown in the following example:

```
var ref = window.open('help.html', '_blank');
```

In this case, I've added an HTML file called help.html to the Cordova project's www folder. The Cordova `prepare` command makes sure the file is copied over to the right location for each supported mobile platform project and is available to be loaded by the application as needed. You can see an example of the page loading on an Android device in Figure 12.23.

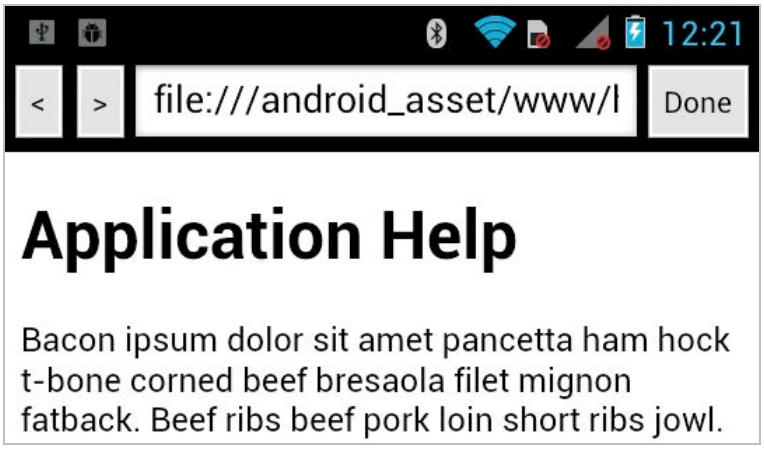



**Figure 12.23** InAppBrowser Opening a Local Web Page

Notice that the location bar is loaded by default, since I didn't specify one in the call to `window.open`, and that the file's location on Android is in the android_asset/www folder.

To close an InAppBrowser window, simply call the `close` method on the window object:

```
ref.close();
```

As fun an interesting as it is to load web content into another window, there are also ways that an application can interact with the window. In the next few sections, I highlight the different ways for interacting with the browser window from within a Cordova application.

## Browser Window Events

There are many scenarios in which an application will want to know what is going on within an InAppBrowser window. To accommodate those requirements, the InAppBrowser API fires different events at different times in the InAppBrowser window lifecycle. The supported events are

- `loadstart`: Fires when the InAppBrowser begins to load a URL
- `loadstop`: Fires when the InAppBrowser completes loading a URL
- `loaderror`: Fires when the InAppBrowser encounters an error while loading a URL
- `exit`: Fires when the InAppBrowser window is closed (either by the user or by the application calling the `close` method)

To flesh out my example from earlier, here's a block of code that opens a local HTML file using InAppBrowser, then defines event listeners for each of the new window's events:

```
var ref = window.open('help.html', '_blank');
ref.addEventListener('loadstart', onEvent);
ref.addEventListener('loadstop', onEvent);
ref.addEventListener('loaderror', onLoadError);
ref.addEventListener('exit', onEvent);
```

Notice that instead of having a separate event callback function for each, I have only implemented two callback functions, one for errors and another for everything else. This is because when anything but an error event fires, the callback function is passed an `event` object that describes the event that was fired, as is illustrated in the following code:

```
function onEvent(event) {
  console.log('Type: ' + event.type);
  console.log('URL: ' + event.url);
  //Do something based on event.type

}
```

Developers can query `event.type` and do whatever is appropriate for the particular event that has fired and dramatically simplify the code being executed.

When an error occurs, the error callback function is passed an object that includes code and message properties, as illustrated in the following code. Developers can then query `event.code` and display an appropriate error message or perform the appropriate recovery steps as needed.

```
function onLoadError(event) {
  console.log('onLoadError: ' + event.code + ' - ' + event.message));
}
```

## Execute Scripts

There are times when simply loading web content in a separate window isn't enough; you might need to modify content or execute some JavaScript within the page. To accommodate this need, the InAppBrowser includes a method that allows an application to execute JavaScript code within the InAppBrowser window.

To make use of this feature in your application, you have your application execute the `executeScript` method, as shown in the following example:

```
ref.executeScript(scriptInfo, onSuccess);
```

The `onSuccess` function passed to the method is the standard success callback function you've seen used throughout this chapter. The `scriptInfo` parameter in the example defines what JavaScript code

is executed and where the code is obtained from: either passed directly to the method or loaded from a file.

To execute a specific piece of JavaScript code, you would pass in a JavaScript object with a property of `code` and a value that consists of a string containing the JavaScript code being executed:

```
{code : "$('#heading').replaceWith('<h2>This is some injected text</h2>');"}
```

In this example, the code is using the jQuery `replaceWith` function to replace some of the content within the loaded web page.

You can't execute your JavaScript code until the page has finished loading, so you will most likely add the call to `executeScript` to some part of your code that you know will execute after the page has completed loading, such as the `loadstop` event listener shown in the following example:

```
var ref = window.open('help.html', '_blank', 'location=no');
ref.addEventListener('loadstop', function() {
  ref.executeScript({
    code : "$('#heading').replaceWith('<h2>This is some injected text</h2>');"
  }, onSuccess);
});
```

To illustrate this example, I added a div called `heading` to the top of the local help.html file used in a previous example. Then, when the example code provided executes, after the page loads, you will see the div's content update, as shown in Figure 12.24.

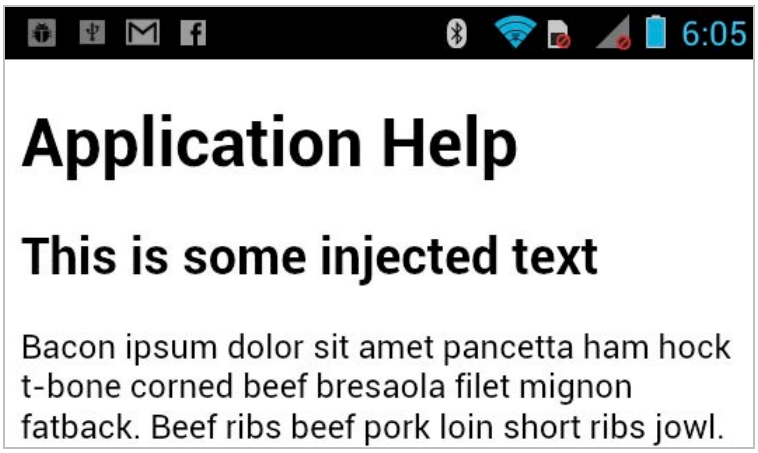



**Figure 12.24** InAppBrowser Showing the Results of `executeScript`

Instead of passing in your JavaScript code directly to the `executeScript` method, you can save your code to a file and pass the file name to `executeScript` via the `scriptInfo` parameter, as shown here:

```
{file : "myscript.js"}
```

The end result is the same; only the source of the JavaScript code changes in the example code shown earlier.

## Insert CSS

Along with the ability to execute script within an InAppBrowser window, you can also use a method exposed by the InAppBrowser to insert CSS into its window. Say, for example, the page you've loaded into the InAppBrowser window came from an external source, and you want to change the styling of the page to match the rest of your application, y ou can easily change the CSS for the page on the fly. To do this, code the application to call the InAppBrowser's `insertCSS` method and pass in either the CSS or a reference to a CSS file you want inserted, as shown in the following example:

```
ref.insertCSS(cssInfo, onSuccess);
```

The `onSuccess` function passed to the method is the standard success callback function you've seen used throughout this chapter. The `cssInfo` parameter in the example defines what CSS is inserted and where the CSS is obtained from: either passed directly to the method or loaded from a file.

To pass in a specific piece of CSS, you would pass in a JavaScript object with a property of `code` and a value that consists of a string containing the CSS being inserted:

```
{code : "body {background-color:black; color:white}"}
```

You can't insert your CSS until the page has finished loading, so you will most likely add the call to `insertCSS` to some part of your code that you know will execute after the page has completed loading, such as the `loadstop` event listener shown in the following example:

```
var ref = window.open('help.html', '_blank', 'location=no');
ref.addEventListener('loadstop', function() {
  ref.insertCSS({
    code : "body {background-color:black; color:white}"
  }, onSuccess);
});
```

Figure 12.25 shows the page loaded with the modified CSS; notice how I switched page colors, making the background black and the text color white.

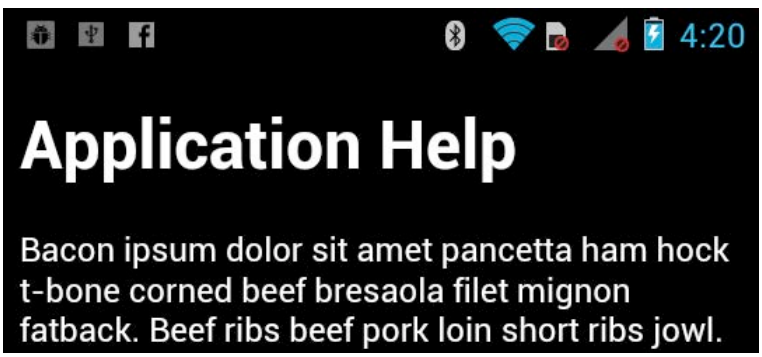



**Figure 12.25** InAppBrowser Showing the Results of `insertCSS`

Instead of passing in your CSS directly to the `insertCSS` method, you can save it to a file and pass the file name to `insertCSS` via the `cssInfo` parameter, as shown here:

```
{file : "mystuff.css"}
```

The end result is the same; only the source of the CSS changes in the example code shown earlier.

# Splash Screen

Cordova provides a Splashscreen API an application can use to display a custom splash screen when a Cordova application launches. To enable an application to use the Splashscreen API, you must first add the Splashscreen plugin to your project by issuing the following CLI command from the Cordova project folder:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-splashscreen.git
```

You need to do a bit of work to create the appropriate splash screen graphic files configured and scaled to support the variety of mobile device platforms as well as the multiple form factors per mobile OS. There's also some manual setup you have to do for each mobile device platform: modifying the Android Cordova container source code. For those reasons, this particular topic is a little more complicated than appropriate for the scope of this chapter.

From a coding standpoint, once you have the appropriate graphics created and added to your Cordova and platform projects, you can display and hide your application's splash screen using the following code:

```
function showSplash() {
  navigator.splashscreen.show();
  setTimeout(hideSplash, 2000);
}

function hideSplash() {
  navigator.splashscreen.hide();
}
```

In this example, the `showSplash` function displays the splash screen, then sets up a timer to have the splash screen hide itself after 2 seconds.

# Wrap-Up

A lot went on in this chapter. I introduced most of the Cordova APIs and showed you how to leverage them in your Cordova applications. I didn't go into a lot of detail for each but tried to show you enough for you to understand how the APIs worked and get you started with the way they're invoked and what the responses look like. You will need to spend some time digging into the Cordova documentation for additional information about all of the options supported by each API.

# 13

# Creating Cordova Plugins

So far we've talked a lot about the tools and plugins that are part of the Cordova framework, but what if you want to do something within a Cordova application that isn't exposed through one of the existing plugins (either core plugins or third-party plugins)? Well, you will have to go it alone and build your own plugins. In this chapter, I show you how to create your own plugins for Apache Cordova.

Cordova plugins are not new—they've been around for a while—but with the release of Cordova 3.0 and the capabilities provided by plugman and the Cordova CLI, plugins have changed a bit. In this chapter, I show you how to create a JavaScript-only plugin as well as a native plugin for both Android and iOS. The process is essentially the same for other mobile platforms, so you need to refer to the documentation for coverage of other platforms, such as BlackBerry 10 and Windows Phone 8.

## Anatomy of a Cordova Plugin

Before I jump into how to create a plugin, I thought I'd spend some time explaining the anatomy of a Cordova plugin: what makes a bunch of collected files a Cordova plugin. The Cordova documentation site has great introduction documents that describe how to create plugins. The "Plugin Development Guide" (http://tinyurl.com/ka68thu) describes how to create the JavaScript interface for your plugins, and there are individual guides for creating the native plugin components for the different mobile platforms linked on the bottom of the page. You can find the Android guide at http://tinyurl.com/kwz42ln and iOS at http://tinyurl.com/kcmnp2g.

A Cordova plugin is a set of files that together extend or enhance a Cordova application's capabilities. In general, a developer adds a plugin to a Cordova project (using the tools discussed in Chapter 4, "Using the Cordova Command-Line Interface") and interacts with the plugin through an exposed JavaScript interface. A plugin could do something without any coding by the developer, but in general, you add the plugin to your project, then use it as you see fit by coding against the exposed API.

I mentioned that a plugin is a collection of files; it consists of a configuration file called plugin.xml, one or more JavaScript files, plus (optionally) some native source code files, libraries, and associated content (HTML, CSS, and other content files) that are used by the plugin.

The plugin.xml describes the plugin and tells the CLI which parts get copied where and which files are unique to each mobile platform. There are even settings in the plugin.xml, which are used by the CLI to set platform-specific config.xml settings. There are a lot of available options within the plugin.xml file; I won't cover all of them here.

A plugin has to have at least one JavaScript source file; this one file is expected to define the methods, objects, and properties that are exposed by the plugin. Your application may wrap everything into one

JavaScript file or may have several; it's all up to you. You can also bundle in additional JavaScript libraries (jQuery Mobile, lawnchair, mustache.js, handlebars.js, and so on) as needed for your plugin.

Beyond those first two requirements, the rest of the plugin files are anything else you need to define your plugin. In general, a plugin includes one or more native source code files for each supported mobile device platforms. On top of that, a plugin might include additional native libraries (in source code form or precompiled) or content (image files, style sheets, HTML files, who knows?).

The good thing about building your own plugins is that there are a whole bunch of examples readily available to you. Figure 13.1 shows the contents of the Cordova 3.0 download package. The .zip files that have "plugin" in their name were created by the Cordova team; you can extract the files and analyze them for hints on how to build your own plugins. One of the easiest ways to get started is to take an existing plugin and modify it to suit your particular needs.

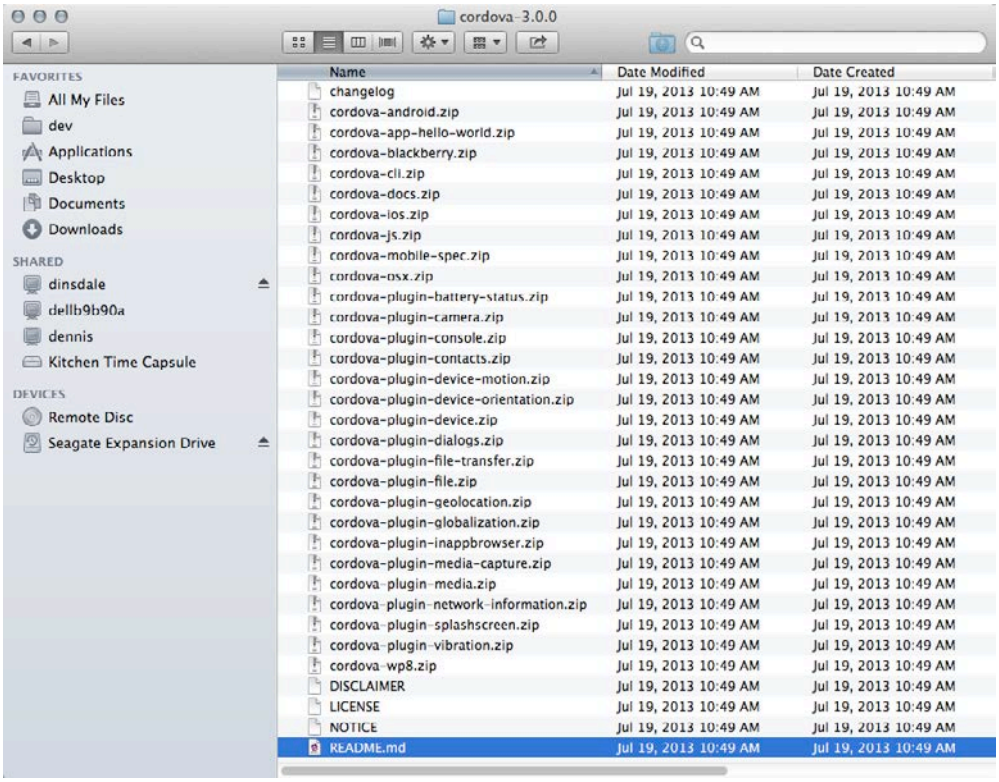



**Figure 13.1** Cordova 3.0 Installation Files

# Creating a Simple Plugin

A Cordova plugin does not have to have native code in it; instead it can be completely composed of JavaScript code. I thought it would make sense to start with a simple plugin, something that shows what a plugin is all about, before digging into plugins with native code for Android and/or iOS. In this section, I show you how to create a plugin composed of only JavaScript code. In the sections that follow, I expand on what we learn here and build a more complicated example plugin.

> **Note**
>
> The plugin I create here isn't going to be very useful; it's merely designed to help teach you about the structure and format of a Cordova plugin. Here I create a plugin that calculates the meaning of life. For those of you who are aware of Douglas Adams's work, this will make perfect sense to you; for the rest of you, in the *Hitchhiker's Guide to the Galaxy*, the "Answer to the Ultimate Question of Life, The Universe, and Everything" (shortened to "Meaning of Life" by me) is 42, so the one method exposed through the plugin will simply return 42 to the calling program. You can learn more about it here: http://en.wikipedia.org/wiki/42_(number).

The plugin I'm creating is called Meaning of Life, which I will abbreviate to MoL. To create this plugin, I created a mol folder in my system's dev folder. Next, I created a file called plugin.xml and placed it in the new folder. The plugin.xml file is a simple XML file that describes the plugin for plugman and the Cordova CLI; you can see the complete listing of the file I used in Listing 13.1; I simply copied it from another plugin and modified it to suit my plugin's needs.

**Listing 13.1: plugin.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://apache.org/cordova/ns/plugins/1.0"
id="com.cordovaprogramming.meaningoflife" version="1.0.0">
  <name>MoL</name>
  <description>Calculates the Meaning of Life (an obscure
    reference to Douglas Adams's Hitchhiker's Guide to the
    Galaxy)
  </description>
  <author>John M. Wargo</author>
  <keywords>mol, Meaning of Life</keywords>
  <license>Apache 2.0 License</license>
  <engines>
    <engine name="cordova" version="3.0.0" />
  </engines>
  <js-module src="mol.js" name="mol">
    <clobbers target="mol" />
  </js-module>
  <info>This plugin exists for no other reason but to
    demonstrate how to build a simple, JavaScript-only
    plugin for Apache Cordova.</info>
</plugin>
```

Some of the information in the plugin.xml file is for documentation purposes, to allow others to understand who created the plugin and why. The other options in the file are used to drive the plugman or the Cordova CLI plugin installation process. You can find a listing of the different plugin.xml elements in Table 13.1.

**Table 13.1 Cordova plugin.xml Elements**

| Component | Description |
|---|---|
| `plugin` | Defines the name space, ID, and version for the plugin. Cordova 3 plugins should utilize namespace defined at http://apache.org/cordova/ns/plugins/1.0. The ID for the plugin is what will be displayed in the plugin list when you issue a `cordova plugins` command in a terminal window from a Cordova project folder. |
| `name` | Defines the name for the plugin. |
| `description` | Defines the description of the plugin. |
| `author` | Defines the name of the author who created the plugin. |
| `keywords` | Defines the keywords associated with the plugin. The Cordova team is working on a public, searchable repository for plugins, and the keywords you add here will eventually help in the discovery process if you later add your plugin to the repository. |
| `license` | Defines the license for the plugin. In my example, I just used the default license that Cordova uses, but you can enter anything you want here, either a license description and/or link to license terms. |
| `engines` | Used to define the versions of Cordova the plugin supports. Add an additional `engine` element for each supported Cordova version. |
| `js-module` | Refers to the file name for a JavaScript file for which a `<script>` tag will be automatically added to a Cordova project's startup page (index.html by default). By listing your plugin's main JavaScript files within `js-module` tags, you eliminate some of the work developers have to do to use your plugin in their Cordova applications.<br>The `clobbers` element specifies that the `module.exports` (mol in the case of this plugin) is automatically added to the `window` object, giving your plugin's methods scope at the window level. |
| `info` | I added this to the file just for fun; it's another place besides the `description` element to tell developers something about your plugin. |

Next, I created a JavaScript file called mol.js and placed it in the mol folder. The complete file listing is provided in Listing 13.2, and it is only 7 lines of code. The code simply creates a `mol` object, then defines within it the `calculateMOL` function used to calculate the Meaning of Life. The last line in the file exports the `mol` object, which is what makes the object and its associated function available to Cordova applications that use the plugin.

**Listing 13.2: mol.js**

```
var mol = {
  calculateMOL : function() {
    return 42;
  }
};

module.exports = mol;
```

That's it—that's all there is to creating a simple Cordova plugin. To prove that it works, I created the simple MoL Demo application shown in Listing 13.3. First, I opened a terminal window, then navigated to my system's dev folder, then issued the following CLI commands:

```
cordova create simplePlugin com.cordovaprogramming.simplePlugin SimplePlugin
cd simplePlugin
cordova platform add android ios
cordova plugin add c:\dev\plugins\mol
```

With those steps completed, all I had to do was open my HTML editor of choice and enter the code in Listing 13.3. In the application, I created a simple button, and when the button is tapped, my `doMOL` function is executed, which makes the call to `mol.calculateMOL()` and displays the result in an `alert` dialog.

### Listing 13.3: MoL Demo Application (simpleplugin.html)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Meaning of Life Demo</title>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>
    <script type="text/javascript" charset="utf-8">

      function onBodyLoad() {
        document.addEventListener("deviceready", onDeviceReady,
          false);
      };

      function onDeviceReady() {
        //alert("onDeviceReady");
      };

      function doMOL() {
        var res = mol.calculateMOL();
        alert('Meaning of Life = ' + res);
      }
    </script>
  </head>
  <body onload="onBodyLoad()">
    <h1>MoL Demo</h1>
    <p>
      This is a Cordova application that uses my custom
      Meaning of Life plugin.
    </p>
    <button onclick="doMOL();">
      Calculate Meaning of Life
```

```
    </button>
  </body>
</html>
```

That's all there is to it. When you run the application and tap the button, you should see the results shown in Figure 13.2.

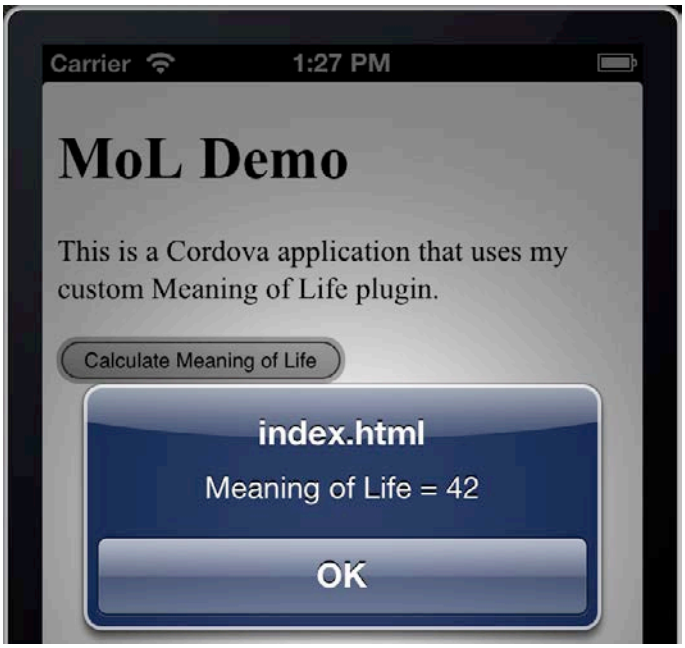



**Figure 13.2** Calculating the Meaning of Life

# Creating a Native Plugin

Now that I've shown you how to create a simple, JavaScript-only plugin, it's time to go beyond that simple example and create a plugin for Cordova that contains native code. In the next two sections, I show you how to create a simple native plugin for Android and how to add the same functionality for iOS.

This example plugin isn't fancy; it simply exposes some native telephony APIs to a Cordova container. For this plugin, I just started poking around in the Android SDK documentation and found something simple and interesting to expose, then coded the plugin. The reason I took this approach was to keep the native API code I was exposing through this plugin as simple as possible. The purpose of this chapter is to show you how to make plugins, not how to do native development, so that's why the plugin is not too complicated. What you'll see in the next couple of sections is how to structure your plugin; adding additional APIs or more sophisticated APIs to your plugins will be easy once you know how plugins work.

The plugin is called Carrier, and it exposes information provided by a smartphone's carrier: the carrier name plus the country code where the device is located or provisioned.

When I created the plugin, I created a folder for the plugin, as shown in Figure 13.3. Within that folder are the JavaScript interface definition plus the plugin.xml. I created a subfolder called src and added android and ios folders there to store that native code for each of the platforms. This isn't a required structure for your plugins but makes it easy to keep things organized.

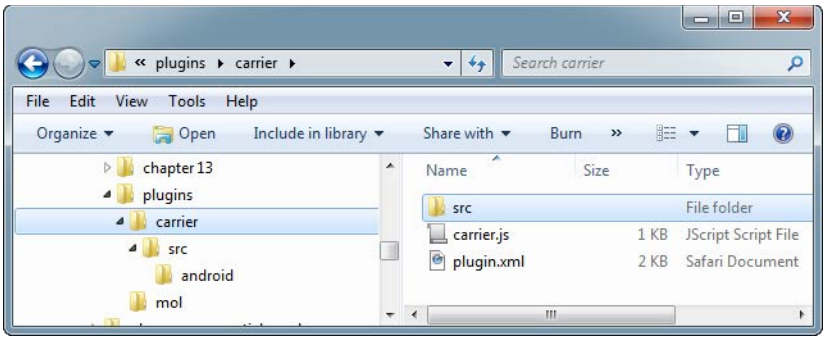



**Figure 13.3** Plugin Folder Structure

Before I start on the native code, I have to first define the JavaScript interface that will be exposed to Cordova applications. The way this works is through the same type of JavaScript file I created for the MoL plugin in the previous section. In the JavaScript file I created for this plugin, I created a simple JavaScript object called `carrier` and defined within it one or more methods that can be called from a Cordova application. For this particular example, the Carrier plugin will expose the `getCarrierName` and `getCountryCode` methods.

Now, unlike the MoL plugin, these methods will not do any calculations and return any values directly; instead, they will make calls to a `cordova.exec` method, which passes control to native code I've created for each target platform. This is the famous JavaScript-to-native bridge that allows Cordova applications to execute native APIs. When the native code is finished doing what it is supposed to be doing, it calls callback functions and passes its results back to the JavaScript object.

The method signature for cordova.exe looks like the following:

```
cordova.exec(successCallback, errorCallback, 'PluginObject',
  'pluginMethod', [arguments]);
```

The `successCallback` and `errorCallback` parameters are the names of the functions that will be executed on success and failure of the particular plugin method call. The `'PluginObject'` parameter is a string that identifies the native object that contains the method being called, and the `'pluginMethod'` parameter is a string that identifies the method that is executed. Lastly, the `arguments` parameter is an optional array of arguments that will be passed to the `pluginMethod`.

In this example, the `getCarrierName` and `getCountryCode` methods don't require that any parameters be passed to them, so the arguments parameter will be empty and represented by empty brackets: `[]`.

Listing 13.4 shows the contents of carrier.js, the JavaScript interface file for the plugin. It begins with the declaration of a `cordova` object, which refers to the loading of the cordova JavaScript library that exposes the `exec` function. With that in place, the `carrier` object is created, and the two methods are defined. Each calls `cordova.exec` and passes the necessary function names and object and method names needed for the `cordova` object to locate the correct native object and methods to do the work I need done.

At the very end of the file is a `module.exports` assignment, which allows the `carrier` object to be exposed to a Cordova application.

**Listing 13.4: carrier.js**

```
var cordova = require('cordova');

var carrier = {
  getCarrierName : function(successCallback, errorCallback) {
    cordova.exec(successCallback, errorCallback,
   'CarrierPlugin', 'getCarrierName', []);
  },

  getCountryCode : function(successCallback, errorCallback) {
    cordova.exec(successCallback, errorCallback,
    'CarrierPlugin', 'getCountryCode', []);
  }
};

module.exports = carrier;
```

With that in place, I am ready to begin coding the native parts of the plugin.

> **Note**
>
> You don't have to define the JavaScript interface for your plugin first. If you look at the different Cordova plugin tutorials available on the Internet, you will find that people do it either way: define the JavaScript interface first or write the native code first. It really doesn't matter. I researched the required native functions first, so I already had what I needed to craft the JavaScript interface.
>
> If you are working with a more complicated plugin, you may find it easier to work through all of the native functions and what they will require before crafting your plugin's JavaScript interface.

## Creating the Android Plugin

With the JavaScript interface defined, it's time to start working on the native part of the plugin. In this case, I did things alphabetically, so I created the Android plugin first, then ported it to iOS later.

You can create Android plugins by creating a new Cordova project, then wiring the Java classes and JavaScript interface file into the application. Then, when the plugin is working in the application, you can pull it out into a separate folder for distribution as a plugin. For this work, I simply created a folder for the plugin and did all my work there, then created a new Cordova project and used the Cordova CLI to add my new plugin there. This allowed me to work on my plugin files separately and test the plugin.xml file at the same time. You may find that this approach doesn't work for you for your plugin development efforts. It probably just depends on how complicated the plugin code is; in this case, the code was simple enough that I didn't have to deal with a lot of compilation errors, so working on the plugin as a standalone project was easier for me.

On Android, the information the plugin will be returning to the Cordova application will come from the Telephony API. To use this API, an application must import the `Context` and `TelephonyManager` classes:

```
import android.content.Context;
import android.telephony.TelephonyManager;
```

Then, within the application, define an instance of an object that exposes the methods we need to call to get the carrier name and country code:

```
tm = (TelephonyManager) getSystemService(
  Context.TELEPHONY_SERVICE);
```

To determine the carrier name, the plugin will make a call to `tm.getSimOperatorName()`, and to get the country code, it will make a call to `tm.getSimCountryIso()`.

Listing 13.5 lists the Java code used for the Android plugin; it defines a simple class called `CarrierPlugin` that exposes the `exec` method, which is what is executed by the call to the JavaScript `cordova.exec` highlighted in Listing 13.4.

The class defines two constants, `ACTION_GET_CARRIER_NAME` and `ACTION_GET_COUNTRY_CODE`, which are used in determining which method was called by the Cordova application. You could hard code the comparison deeper within the Java code, but doing it this way makes it easier to change the names later.

Next, the class defines the `tm` object used to give the plugin access to the Telephony APIs.

The `initialize` method first calls `super.initialize`, which allows the `cordova` object to be initialized properly. Without this call, the Java code has no awareness of the Cordova container. Next, the code gets a handle to the current application context and uses that to wire the `tm` object into the services exposed by the Telephony API.

Next, the Java code overrides the `exec` method and implements the code that deals directly with the calls from the Cordova application. In this implementation, I've implemented a single operation that determines which action has been called (by comparing the action name passed by the call to `cordova.exec` to the constants I defined earlier) and acts accordingly.

If the `exec` method determines that the `getCarrierName` action was requested, then it makes the call to the Android `getSimOperatorName` method and passes the results back to the Cordova application by calling `callbackContext.success()`. If the `getCountryCode` action was requested, then it makes a call to the Android `getSimCountryIso()` and passes the results back to the Cordova application by calling `callbackContext.success()`.

If any part of this process fails, the code executes `callbackContext.error` and passes back an appropriate error message or error object indicating what went wrong.

### Listing 13.5: CarrierPlugin.java

```
package com.cordovaprogramming.carrier;

//Cordova imports
import org.apache.cordova.CordovaInterface;
import org.apache.cordova.CallbackContext;
import org.apache.cordova.CordovaPlugin;
import org.apache.cordova.CordovaWebView;
```

```
//Android imports
import android.content.Context;
import android.telephony.TelephonyManager;

//JSON Imports
import org.json.JSONArray;
import org.json.JSONException;

public class CarrierPlugin extends CordovaPlugin {

  //Define some constants for the supported actions
  public static final String
    ACTION_GET_CARRIER_NAME = "getCarrierName";
  public static final String
    ACTION_GET_COUNTRY_CODE = "getCountryCode";

  public TelephonyManager tm;

  public void initialize(CordovaInterface cordova,
    CordovaWebView webView) {
    super.initialize(cordova, webView);

    //The plugin doesn't have direct access to the
    //application context, so you have to get it first
    Context context =
      this.cordova.getActivity().getApplicationContext();
    //Next we initialize the tm object
    tm = (TelephonyManager)
      context.getSystemService(Context.TELEPHONY_SERVICE);
  }

  @Override
  public boolean execute(String action, JSONArray args,
    CallbackContext callbackContext) throws JSONException {
    try {
      //First check on the getCarrierName
      if (ACTION_GET_CARRIER_NAME.equals(action)) {
        callbackContext.success(tm.getSimOperatorName());
        return true;
      } else {
        //Next see if it is a getCountryCode action
        if (ACTION_GET_COUNTRY_CODE.equals(action)) {
          callbackContext.success(tm.getSimCountryIso());
          return true;
        }
      }
      //We don't have a match, so it must be an invalid action
      callbackContext.error("Invalid Action");
      return false;
    } catch (Exception e) {
      //If we get here, then something horrible has happened
```

```
      System.err.println("Exception: " + e.getMessage());
      callbackContext.error(e.getMessage());
      return false;
    }
  }
}
```

That's all there is to the code; I tried to make it as simple as possible in order to allow you to focus on the parts that are specific to Cordova plugins.

Before I can use the CLI to add this new plugin to a Cordova project, I have to create the plugin's plugin.xml file, shown in Listing 13.6. The file is similar to the one I used for the MoL plugin, but as there are native components of this one, there are some extra settings to explain. First take a look at the file; I'll describe the contents after the listing.

### Listing 13.6: plugin.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://apache.org/cordova/ns/plugins/1.0" id="com.cordovaprogramming.carrier"
version="1.0.0">
  <name>Carrier</name>
  <author>John M. Wargo</author>
  <description>Exposes mobile carrier related values to a Cordova application.</description>
  <keywords>carrier</keywords>
  <license>Apache 2.0 License</license>
  <engines>
    <engine name="cordova" version="3.0.0" />
  </engines>
  <js-module src="carrier.js" name="carrier">
    <clobbers target="carrier" />
  </js-module>
  <platform name="android">
    <!-- android-specific elements -->
    <source-file src="src/android/CarrierPlugin.java" target-
dir="src/com/cordovaprogramming/carrier" />
    <config-file target="res/xml/config.xml" parent="/*">
      <feature name="CarrierPlugin" >
        <param name="android-package" value="com.cordovaprogramming.carrier.CarrierPlugin"/>
      </feature>
    </config-file>
    <config-file target="AndroidManifest.xml" parent="/*">
      <uses-permission android:name="android.permission.READ_PHONE_STATE" />
    </config-file>
  </platform>
  <platform name="ios">
    <!-- ios-specific elements -->
  </platform>
</plugin>
```

The file's `js-module` element defines the name of the JavaScript file, which will be automatically loaded on application startup. It defines the JavaScript interface exposed to the Cordova application. The `clobbers` element specifies the JavaScript object assigned to the loaded JavaScript object. In this example, the Carrier plugin is defined to be exposed to the Cordova application through a `carrier` object.

```
<js-module src="carrier.js" name="carrier">
  <clobbers target="carrier" />
</js-module>
```

The Cordova application accesses the `getCarrierName` method through the `carrier` object:

```
carrier.getCarrierName(onSuccess, onFailure);
```

The only part of the file that is different from the earlier example (Listing 13.1) is the `platform` section:

```
<platform name="android">
  <!-- android-specific elements -->
  <source-file src="src/android/CarrierPlugin.java"
    target-dir="src/com/cordovaprogramming/carrier" />
  <config-file target="res/xml/config.xml" parent="/*">
    <feature name="CarrierPlugin" >
      <param name="android-package"
        value="com.cordovaprogramming.carrier.CarrierPlugin"/>
    </feature>
  </config-file>
  <config-file target="AndroidManifest.xml" parent="/*">
    <uses-permission
      android:name="android.permission.READ_PHONE_STATE" />
  </config-file>
</platform>
```

It defines settings that are particular to specific mobile device platform and contains settings related to the native code I've shown you in this section. There can be one or more platform elements in a plugin.xml file; the following one defines elements that apply to android plugin components:

```
<platform name="android"></platform>
```

Within that element is a `source-file` element that points to one or more Android native source code files, which need to be installed by the CLI when the plugin is installed. The following example instructs plugman or the CLI to copy the file called CarrierPlugin.java located in the plugin source folder's src/android folder to the Cordova project's Android platform folder in the src/com/cordovaprogramming/carrier folder:

```
<source-file src="src/android/CarrierPlugin.java"
  target-dir="src/com/cordovaprogramming/carrier" />
```

You don't have to copy the file to that deep of a folder structure, but it's the norm for Cordova plugins to be installed into such a folder structure. Take a look at some of the Cordova core plugins, and you will see what I mean.

The `config-file` element defines the changes that need to be made during plugin installation. In the following example, it specifies that a feature named `CarrierPlugin` should be added to the Android project's config.xml file and should point to the Java Class `com.cordovaprogramming.carrier.CarrierPlugin`:

```
<config-file target="res/xml/config.xml" parent="/*">
  <feature name="CarrierPlugin" >
    <param name="android-package"
      value="com.cordovaprogramming.carrier.CarrierPlugin"/>
  </feature>
</config-file>
```

The last element in `platform` defines another configuration file setting. On Android, access to the Telephony API requires specific permissions. Any application that uses the API must add an entry to the application's AndroidManifest.xml file, which lists the specific permissions required by the application. In this case, I have to add the `android.permission.READ_PHONE_STATE` permission to the manifest:

```
<config-file target="AndroidManifest.xml" parent="/*">
  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
</config-file>
```

When you look at the application's settings screen for an application that uses this plugin, you will see the entry for "read phone status and identity," shown in Figure 13.4; it shows that the permission has been set correctly.

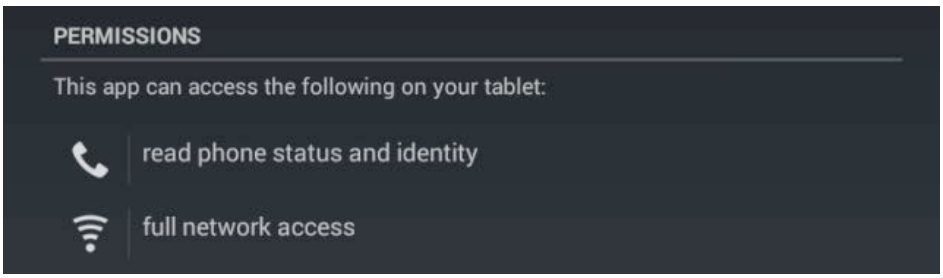



**Figure 13.4** Android Application Permissions

If you do not set these permissions correctly, the part of the application that uses an API that requires the permission will fail without warning. If you forget to do this and wonder later why the Telephony API doesn't seem to be working, be sure to check your permissions.

To test the application, I created the simple application highlighted in Listing 13.6. To create the application, I opened a terminal window, then navigated to my system's dev folder and issued the following CLI commands:

```
cordova create nativePlugin com.cordovaprogramming.nativePlugin NativePlugin
cd nativePlugin
cordova platform add android
cordova plugin add c:\dev\plugins\carrier
```

With those steps completed, all I had to do was open my HTML editor of choice and enter the code in Listing 13.7. The application displays two buttons, one that calls `getCarrierName` and another that calls `getCountryCode`. The same `onSuccess` function is executed to display the results for both methods.

**Listing 13.7: index.html**

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

```
    <title>Carrier Demo</title>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" content="user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1,
      width=device-width;" />
    <script type="text/javascript" charset="utf-8"
      src="cordova.js"></script>
    <script type="text/javascript" charset="utf-8">
      function onBodyLoad() {
        document.addEventListener("deviceready", onDeviceReady,
          false);
      };

      function onDeviceReady() {
        //Nothing to do here really
        //but I always include this just in case
      };

      function doSomething1() {
        carrier.getCarrierName(onSuccess, onFailure);
      }

      function doSomething2() {
        carrier.getCountryCode(onSuccess, onFailure);
      }

      function onSuccess(result) {
        var resStr = "Result: " + result;
        console.log(resStr);
        alert(resStr)
      }

      function onFailure(err) {
        console.log("onFailure: " + JSON.stringify(err));
        alert("Failure: " + err);
      }
    </script>
  </head>
  <body onload="onBodyLoad()">
    <h1>Carrier Demo</h1>
    <p>
      This is a Cordova application that uses my fancy new Carrier plugin.
    </p>
    <button onclick="doSomething1();">
      Get Carrier Name
    </button>
    <button onclick="doSomething2();">
      Get Country Code
    </button>
  </body>
</html>
```

When the application runs, it will display a screen similar to the one shown in Figure 13.5 (which I have cropped here for the sake of page real estate).


**Figure 13.5** Carrier Demo Application

When you tap the Get Carrier Name button, the application calls the appropriate Telephony API and returns the carrier name, as shown in Figure 13.6. I took that particular screenshot on an Android emulator, so it returns Android rather than a particular carrier's name. I have an Android tablet, but it's provisioned for data only, so the API doesn't return anything.

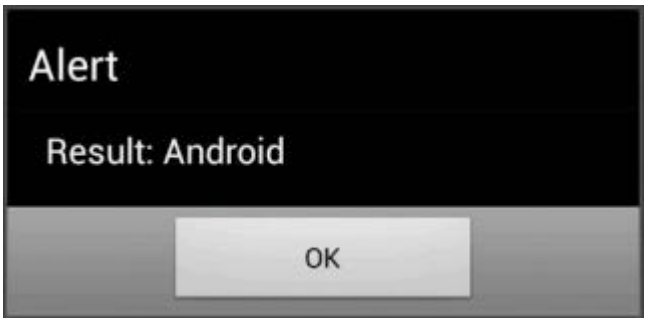



**Figure 13.6** Results of `getCarrierName` on an Android Emulator

When you tap the Get Country Code button, the application calls the appropriate Telephony API and returns the country code, as shown in Figure 13.7.

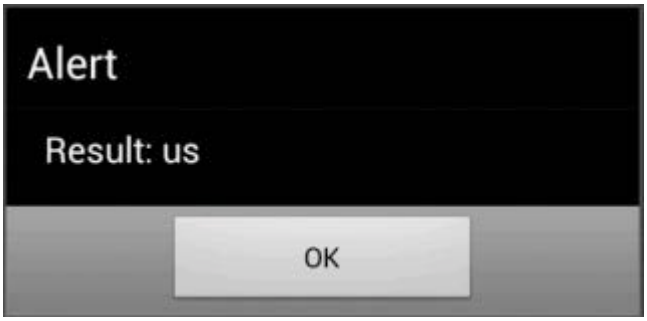



**Figure 13.7** Results of `getCountryCode` on an Android Emulator

That's it—I now have a native plugin for Android, and it's all ready to go.

## Creating the iOS Plugin

In this section of the chapter, I show you how to implement the iOS version of the native plugin discussed in the previous section.

For the iOS version of the plugin, things are a little simpler. On Android, when you call a plugin's methods, `cordova.exec` locates the native class and executes the methods on the class. For iOS, the plugin's methods are executed through URL commands exposed by the plugin. To get started, I navigated to my plugin's src/ios folder and created two files: CarrierPlugin.h and CarrierPlugin.m. The .h file is the iOS plugin's Header file and defines the interface for the iOS plugin code. The .m file is the implementation of the interface and contains the code that does the work for the plugin.

Listing 13.8 shows the content of the CarrierPlugin.h file. Here, I define the interface for `CarrierPlugin` as an implementation of `CDVPlugin`. What this does is let the iOS device know I'm exposing the code as a Cordova plugin. Next, the file defines the two methods that are available: `getCarrierName` and `getCountryCode`. Sound familiar? They should—those are the methods that will be called by the plugin's JavaScript interface shown in Listing 13.1.

### Listing 13.8: CarrierPlugin.h

```
#import <Cordova/CDVPlugin.h>

@interface CarrierPlugin : CDVPlugin {

}

- (void)getCarrierName:(CDVInvokedUrlCommand*)command;
- (void)getCountryCode:(CDVInvokedUrlCommand*)command;

@end
```

Listing 13.9 shows the contents of the CarrierPlugin.m file; this is the implementation of the interface described in Listing 13.8. Here the code imports some of the interfaces used by the plugin, then defines the functions that are called when the Cordova application invokes `getCarrierName` and `getCountryCode`.

As you can see from the code, both methods leverage `CTTelephonyNetworkInfo`; for this plugin, neither of the methods accepts any parameters. For both methods, the plugin first defines a `netinfo` object, which is assigned to an instance of the `CTTelephonyNetworkInfo` class, as shown here:

```
CTTelephonyNetworkInfo *netinfo = [[CTTelephonyNetworkInfo alloc] init];
```

Next, a `carrier` object is created; it provides the plugin with access to the cellular provider properties:

```
CTCarrier *carrier = [netinfo subscriberCellularProvider];
```

With that in place, the carrier name is exposed to the plugin through the `carrier` object's `carrierName` property. At this point, the plugin simply needs to return the value to the calling program. First, the method sets the result of the operation and assigns the return value, as shown here:

```
CDVPluginResult* result = [CDVPluginResult resultWithStatus:CDVCommandStatus_OK
messageAsString:[carrier carrierName]];
```

Then, the method calls the appropriate callback function to complete the process:

```
[self.commandDelegate sendPluginResult:result callbackId:[command callbackId]];

}
```

That's it—that's all there really is to the code. For the `getCountryCode` method, it uses the same process, only returning the value of the `carrier` object's `isoCountryCode` property. Listing 13.9 shows the complete code listing for the file.

### Listing 13.9: CarrierPlugin.m

```
#import "CarrierPlugin.h"
#import <CoreTelephony/CTTelephonyNetworkInfo.h>
#import <CoreTelephony/CTCarrier.h>

@implementation CarrierPlugin

- (void)getCarrierName:(CDVInvokedUrlCommand*)command
{
  CTTelephonyNetworkInfo *netinfo = [[CTTelephonyNetworkInfo alloc] init];
  CTCarrier *carrier = [netinfo subscriberCellularProvider];
  CDVPluginResult* result = [CDVPluginResult resultWithStatus:CDVCommandStatus_OK
messageAsString:[carrier carrierName]];
  [self.commandDelegate sendPluginResult:result callbackId:[command callbackId]];
}

- (void)getCountryCode:(CDVInvokedUrlCommand*)command
{
  CTTelephonyNetworkInfo *netinfo = [[CTTelephonyNetworkInfo alloc] init];
  CTCarrier *carrier = [netinfo subscriberCellularProvider];
  CDVPluginResult* result = [CDVPluginResult resultWithStatus:CDVCommandStatus_OK
messageAsString:[carrier isoCountryCode]];
  [self.commandDelegate sendPluginResult:result callbackId:[command callbackId]];
}

@end
```

With the plugin's code in place, the last thing that has to happen is that the plugin.xml file must be updated with the settings for the iOS portion of the plugin. To do this, I added a new `platform` section to the file; the elements and attributes for this section are shown here:

```
<platform name="ios">
  <!-- ios-specific elements -->
  <header-file src="src/ios/CarrierPlugin.h" />
  <source-file src="src/ios/CarrierPlugin.m" />
  <config-file target="config.xml" parent="/*">
    <feature name="CarrierPlugin" >
      <param name="ios-package" value="CarrierPlugin"/>
    </feature>
  </config-file>
</platform>
```

For iOS plugins, the `platform` element adds a special element called `header-file`, which defines the names of any header files (.h files) used by the plugin. In the plugin.xml file segment shown, I have added a `header-file` element for the plugin's only header file (CarrierPlugin.h) and a `source-file` element for the implementation file (CarrierPlugin.m). They both get copied to the same location within a Cordova project folder structure.

There is one config.xml file update that has to be made: we have to add the `CarrierPlugin` feature to the file and point to the `CarrierPlugin` interface shown in Listing 13.8.

Note that there are no permission settings to be set for this plugin; iOS doesn't require specific application settings to use telephony features as Android does. At this point, I have everything I need defined for the plugin. Listing 13.10 shows the complete listing for the plugin.xml file.

### Listing 13.10: Completed plugin.xml File Contents

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://apache.org/cordova/ns/plugins/1.0" id="com.cordovaprogramming.carrier"
version="1.0.0">
  <name>Carrier</name>
  <author>John M. Wargo</author>
  <description>Exposes mobile carrier related values to a
    Cordova application.</description>
  <keywords>carrier</keywords>
  <license>Apache 2.0 License</license>
  <engines>
    <engine name="cordova" version="3.0.0" />
  </engines>
  <js-module src="carrier.js" name="carrier">
    <clobbers target="carrier" />
  </js-module>
  <platform name="android">
    <!-- android-specific elements -->
    <source-file src="src/android/CarrierPlugin.java"
      target-dir="src/com/cordovaprogramming/carrier" />
    <config-file target="res/xml/config.xml" parent="/*">
      <feature name="CarrierPlugin" >
        <param name="android-package"
          value="com.cordovaprogramming.carrier.CarrierPlugin"/>
```

```
      </feature>
    </config-file>
    <config-file target="AndroidManifest.xml" parent="/*">
      <uses-permission
        android:name="android.permission.READ_PHONE_STATE" />
    </config-file>
  </platform>
  <platform name="ios">
    <!-- ios-specific elements -->
    <header-file src="src/ios/CarrierPlugin.h" />
    <source-file src="src/ios/CarrierPlugin.m" />
    <config-file target="config.xml" parent="/*">
      <feature name="CarrierPlugin" >
        <param name="ios-package" value="CarrierPlugin"/>
      </feature>
    </config-file>
  </platform>
</plugin>
```

To test the plugin, I created a new project and added my plugin to it. When I opened Xcode and tried to run the application, I received an error indicating that there was an undefined symbol in my project, as shown in Figure 13.8.

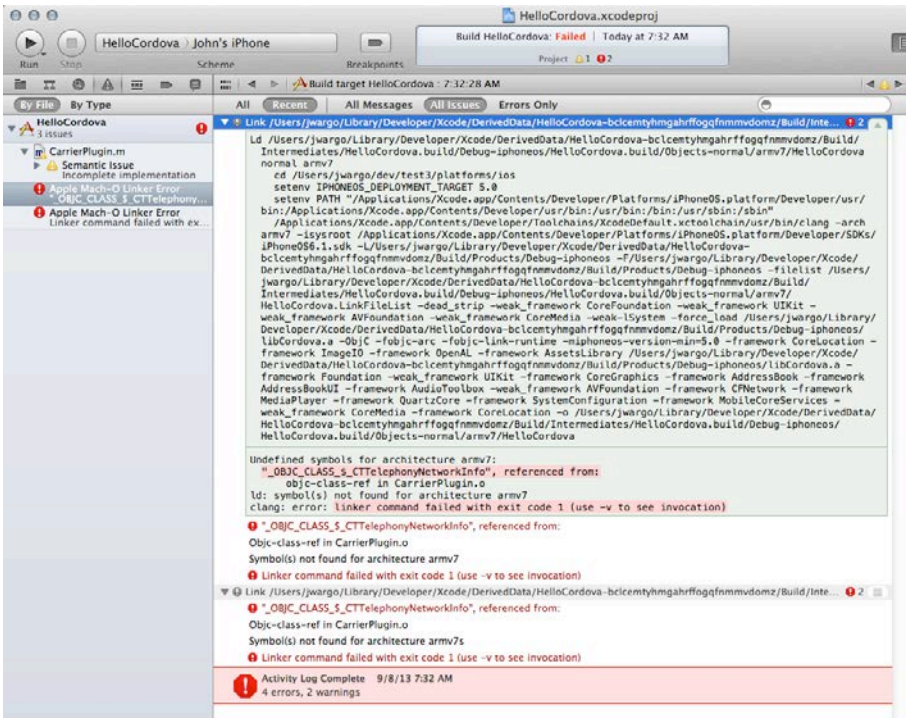



**Figure 13.8** Xcode Build Error

This happened because the project didn't have the iOS CoreTelephony framework added to the project. To fix this error, I clicked on the project name in the Xcode project navigator, opened the Build Phases tab, and added the framework to the Libraries section, as shown already completed in Figure 13.9.

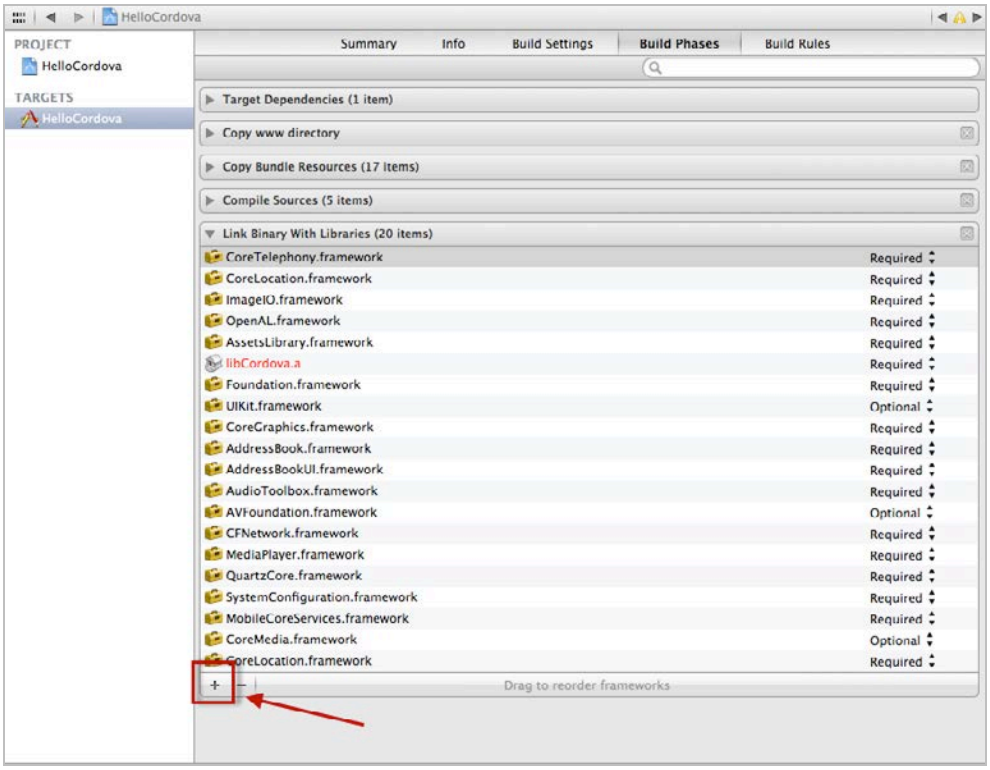



**Figure 13.9** Xcode Project Build Phases Settings

I then clicked the plus sign highlighted in the bottom of Figure 13.9, and in the dialog that appeared, I started to type *telephony*, as shown in Figure 13.10. When the CoreTelephony.framework entry appeared, as shown in the figure, I clicked the Add button to add the framework to my project's configuration.

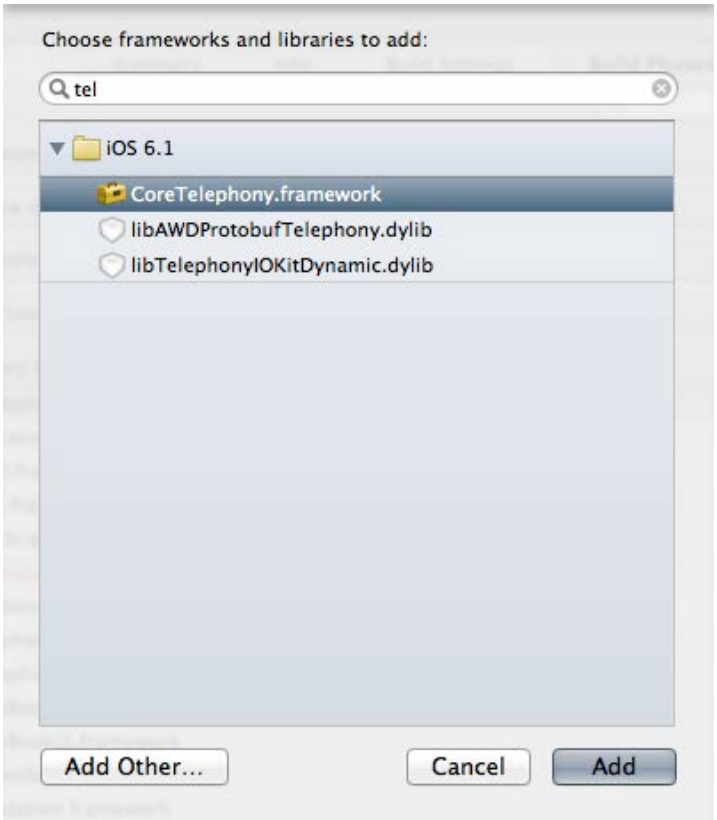



**Figure 13.10** Xcode Add Framework Dialog

With that in place, I copied over my test project's index.html and ran the application on my iOS device, as shown in Figure 13.11.

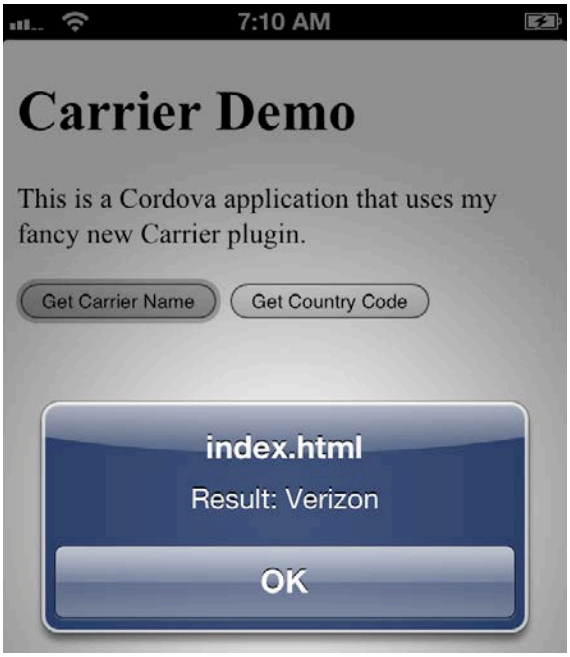



**Figure 13.11** Results of `getCarrierName` on an iOS Device

I couldn't test this on an iOS simulator because the simulator is not provisioned with a mobile carrier and therefore cannot determine any carrier-specific settings, so it returns `null`. When I run the application on the simulator and tap the Get Carrier Name button, I receive the results shown in Figure 13.12.


**Figure 13.12** Results of `getCarrierName` on an iOS Simulator

Now, I could code the plugin's methods so they call the `onFailure` callback function and pass back an error message indicating that the carrier information was not available. Perhaps that is an enhancement you can make to the plugin.

## Deploying Plugins

When it comes to deploying a custom plugin, you have several options. You can zip up the plugin folder, the folder containing your plugin's plugin.xml file plus all of the source files for the plugin, and distribute the .zip file to your users. All the user would have to do is extract the files into a folder and point to that folder when doing a `cordova plugin add` from a terminal window.

You can also distribute your plugins through GitHub (www.github.com). You can create your own GitHub account and post your plugin files there. Then, once the files are up there, you can start socializing your plugin so other developers can learn about it. The PhoneGap team has created a repository located at https://github.com/phonegap/phonegap-plugins where you can publish information about your plugins. The team is also working on a plugin discovery system that would allow developers to more easily locate plugins to use in their applications.

## Wrap-Up

In this chapter, I've shown you how to create Cordova plugins. With the information provided in this chapter, the Cordova documentation, and the core Cordova plugins as examples to study, you have the information you need to create your own custom plugins.

I only skimmed the surface of this topic; there is a lot more you can do with plugins. The plugin.xml supports many more options than what I covered here. For example, if your plugin depends on another plugin, you can configure the plugin.xml to have plugman or the Cordova CLI install the required plugins for you. You can also link in precompiled or custom libraries into your plugin. So, if your plugin leverages some proprietary code, you can still include it, but protect it from prying eyes.

# 14

# Building a Cordova Application

At this point in the book, I've shown you a lot about Cordova programming, but I've not done a complete walkthrough of the process using a real application as an example. So, in this chapter, I take a complete application and show you the steps needed to create and test the application from start to finish.

## About the Application

In *PhoneGap Essentials*, I created complete sample applications for each PhoneGap API; it was a great way to highlight all of the options for each API. For this chapter, I took one of the applications from that book and updated it for Cordova 3.0. I decided to use one of the applications I created to demonstrate how to implement a Compass watch.

The application displays a simple screen showing an image representing a compass dial. As the user rotates the device along a horizontal axis, the compass image rotates, and the heading is displayed on the page below the compass image. The original application's screen is shown in Figure 14.1.

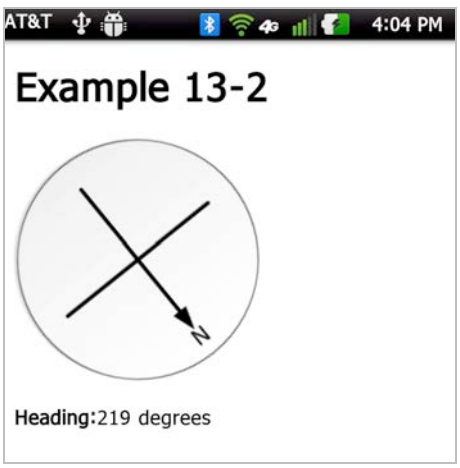



**Figure 14.1** Sample Compass Application

For this chapter, I thought I'd enhance the application's UI using jQuery Mobile plus make use of the Cordova merges folder capability to use a different compass graphic for Android and iOS.

# Creating the Application

To create the application, I opened a terminal window and navigated to my system's dev folder. Next, I issued the following commands to create the Cordova project:

```
cordova create compass
cd compass
cordova platform add android ios
```

At this point, what I have is a new Cordova project with support for the Android and iOS mobile device platforms. I know that the application is going to need to leverage at least one Cordova core plugin, perhaps more, so to make it easy to remember the correct syntax for adding the core plugins, I opened a browser window and navigated to the Cordova command-line interface (CLI) guide at http://tinyurl.com/ob9hanq. If you go to that site and scroll down toward the middle of the page, you will see a listing containing the full command text for adding each of the Cordova plugins to a project, as shown in Figure 14.2.

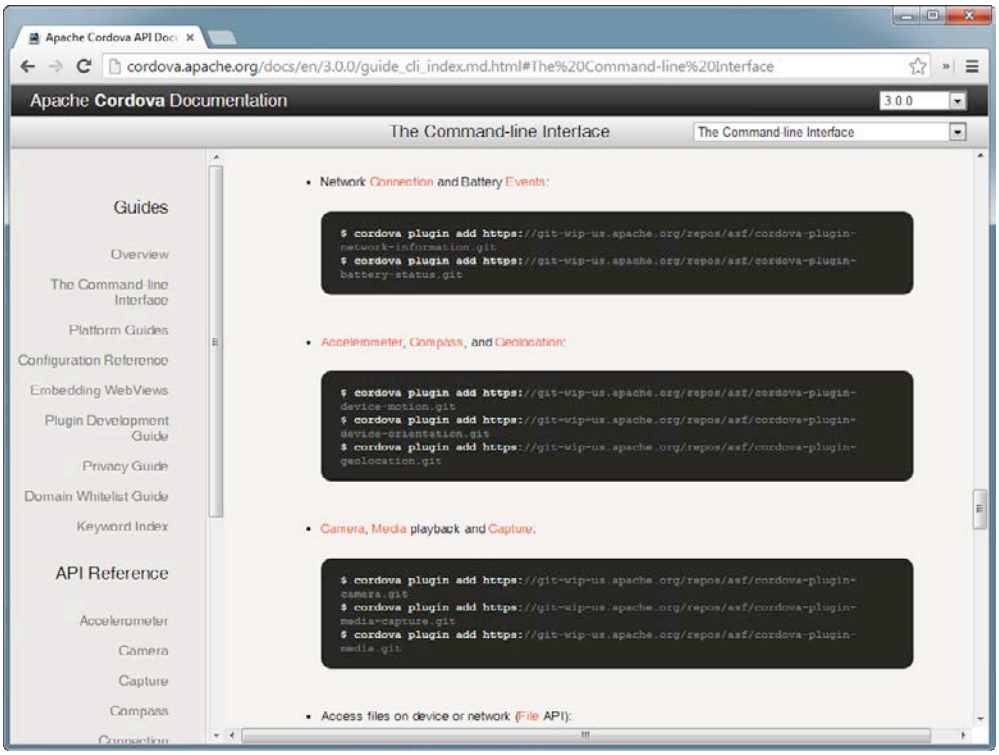



**Figure 14.2** Cordova Documentation: The Command-Line Interface Guide

Knowing that I have to debug the application and therefore must write to the console, I copied the appropriate command from the guide, shown in Figure 14.2, then added the console plugin to the project using the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-console.git
```

Since the application will be using the compass, I also added the compass plugin using the following command:

```
cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device-
orientation.git
```

Remember that Cordova doesn't really show you anything as it works unless you turn on the `-d` flag. So, the complete terminal window output looks like the following:

```
Last login: Mon Sep 9 08:27:24 on ttys000
jmw-mini:~ jwargo$ cordova create compass
jmw-mini:~ jwargo$ cd compass
jmw-mini:compass jwargo$ cordova platform add android ios
jmw-mini:compass jwargo$ cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-
plugin-device-orientation.git
jmw-mini:compass jwargo$ cordova plugin add https://git-wip-us.apache.org/repos/asf/cordova-
plugin-console.git
jmw-mini:compass jwargo$
```

Next, I copied the index.html from the sample project I'm using and placed it in the project's www folder. For your applications, you'll simply open your text or HTML editor of choice and start coding.

The original version of this application had a very simple look to it, which was represented by the following HTML code:

```
<body onload="onBodyLoad()">
  <h1>Example 13-2</h1>
  <img src="compass.png" id="compass" />
  <br />
  <p id="headingInfo"></p>
</body>
```

I wanted to spruce up this version of the application a little bit, so I decided to use jQuery Mobile to create a more mobile phone like look for the app. The first thing I did was download jQuery (www.jquery.com) and jQuery Mobile (www.jquerymobile.com) from their respective websites. I copied the files over to the Cordova project's www folder, then added the following lines to the `head` portion of the application's index.html:

```
<link rel="stylesheet" href="jquery.mobile-1.3.2.min.css" />
<script type="text/javascript" charset="utf-8"
  src="jquery-2.0.3.min.js"></script>
<script type="text/javascript" charset="utf-8"
  src="jquery.mobile-1.3.2.min.js"></script>
```

The application's ability to rotate the image is provided by a free plugin to jQuery called jQuery Rotate; you can find information about the plugin at http://code.google.com/p/jqueryrotate. To add the plugin, I downloaded the plugin's JavaScript file to the Cordova project's www folder, then added the following line immediately after the other `script` tags in the application's index.html file:

```
<script type="text/javascript" charset="utf-8"
  src="jQueryRotateCompressed.js"></script>
```

With the jQuery files in place, I updated the `body` section of the index.html to the following:

```
<body onload="onBodyLoad()">
  <div data-role="page">
    <div data-role="header">
      <h1>Compass</h1>
    </div>
    <div data-role="content">
      <div style="text-align : center;">
        <img src="images/compass.png" id="compass" />
        <br />
        <p id="headingInfo">
          <b>Heading:</b> 0 Degrees
        </p>
      </div>
    </div>
    <div data-role="footer" data-position="fixed">
      <h3>cordovaprogramming.com</h3>
    </div>
  </div>
</body>
```

jQuery Mobile uses `data-role` attributes within elements to style those particular elements with the appropriate look. So, in the preceding example, I created a `header` div on the page and assigned it a `data-role` of `header` and a `footer` div on the page and assigned it a `data-role` of `footer`. The `data-position="fixed"` forces the footer to remain on the bottom of the page.

Next, I moved the content for the page into a div assigned to a `data-role` of `content`. In order to have the compass graphic and heading text centered on the page, I added a new div and styled it with a `style="text-align : center;"`, as shown in the code.

> **Note**
>
> You may be asking why I didn't simply add the `style` the attribute to the div with the `data-role` of `content`. I probably could have done so, but I was trying to stay focused on the educational aspect of this sample and didn't want to clutter things up too much.

With that code in place, the look of the application is changed dramatically, as shown in Figure 14.3.

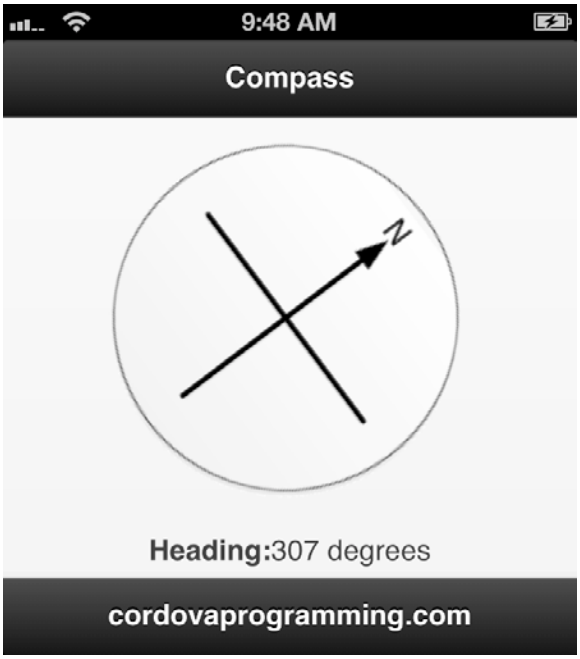



**Figure 14.3** The Compass Application with a jQuery Mobile Look

Now that I have the application's user interface updated, it's time to start looking at the application's JavaScript code. In the application's index.html is a `script` tag that defines a few functions that are used by the application.

The first thing you will see within the `script` tag is the assignment of a function to the `window.onerror` event. Since I know that my code isn't going to work right away (code never works the first time, does it?), and since a Cordova application fails silently most of the time, I added this assignment and the associated function to allow me to see information about every error that occurs when the application runs. The function, shown next, essentially receives an error, the name of the application file URL that encountered the error, and the line number that caused the error; then it generates the appropriate error message and writes it to the log and displays it in a dialog.

```
//Fires whenever there is an error
window.onerror = function(msg, url, line) {
  var resStr;
  var index = url.lastIndexOf('/');
  if (index > -1) {
    url = url.substring(index + 1);
  }
  resStr = 'ERROR in ' + url + ' on line ' + line + ': ' + msg;
  console.error(appName + resStr);
  alert(resStr);
  return false;
};
```

Next comes the `onDeviceReady` function I've shown throughout the book. It is set up as an event listener for the Cordova `deviceready` event and fires after the Cordova container has finished

initialization. In this function, I know that the Cordova container is ready, so I can do almost anything I want to do. Here's the function:

```
function onDeviceReady() {
  console.log('onDeviceReady fired.');
  hi = document.getElementById('headingInfo');
  //Setup the watch
  //Read the compass every second (1000 milliseconds)
  var watchOptions = {
    frequency : 1000
  };
  console.log(appName + 'Creating watch: ' +
    JSON.stringify(watchOptions));
  watchID = navigator.compass.watchHeading(onSuccess, onError,
    watchOptions);
}
```

The first thing the function does is define a `hi` variable, which points to the page element that is assigned an ID of `headingInfo`. The variable is used later to replace the content within that element to show the device's current heading.

Next, the function defines a `watchOptions` variable, which is used to help set up a heading watch that will cause the application to update the heading periodically. The `watchOptions` object can have either a `frequency` property or a `filter` property assigned to it. Right now, the application is set up with the following:

```
var watchOptions = {
  frequency : 1000
};
```

When used with a heading watch, it will cause the Compass API to report the device's heading every 1000 milliseconds (every 1 second). The `filter` property is used to define the amount of heading change (in degrees) before the Compass API will report a heading. In the following example, the `filter` property instructs the Compass API to deliver a heading every time the heading changes more than 1 degree:

```
var watchOptions = {
  filter : 1
};
```

You can specify both properties as shown here, but as soon as you specify a `filter`, the `frequency` property is ignored by the Compass API:

```
var watchOptions = {
  frequency : 1000,
  filter : 1
};
```

With `watchOptions` defined, the function makes a call to `watchHeading` to set up the heading watch:

```
watchID = navigator.compass.watchHeading(onSuccess, onError,
  watchOptions);
```

As with all of the other Cordova APIs you've learned so far, the `onSuccess` function is executed whenever the Compass API sends a heading value, and the `onError` function is executed whenever an error is encountered by the Compass API.

Whenever the `onSuccess` function is executed, the Compass API passes it a `heading` object, which contains properties indicating the device's current heading, as shown here:

```
{
  "magneticHeading":0,
  "trueHeading":0,
  "headingAccuracy":0,
  "timestamp":1378738354661
}
```

The `onSuccess` function uses the `magneticHeading` property of the `heading` object to determine the current heading, then uses that value to rotate the compass graphic by that number of degrees, as shown in the following function:

```
function onSuccess(heading) {
  console.log(appName + 'Received Heading');
  console.log(appName + JSON.stringify(heading));
  var hv = Math.round(heading.magneticHeading);
  console.log(appName + 'Rotating to ' + hv + ' degrees');
  $("#compass").rotate(-hv);
  hi.innerHTML = '<b>Heading:</b> ' + hv + ' Degrees';
}
```

Notice that the compass graphic is being rotated in the opposite direction of the device's current heading. That is because, since the device is rotated, the compass graphic has to rotate in the opposite direction so that the North point on the compass graphic always points to Magnetic North.

Lastly, the application's `onError` function is executed whenever the watch encounters an error. The function uses an `error` object passed to the function to identify the cause of the error and display an appropriate error message for the user. Notice that the `onError` function also cancels the watch, as it makes little sense to continue watching the heading when the application is not able to measure the heading.

```
function onError(err) {
  console.error(appName + 'Heading Error');
  console.error(appName + 'Error: ' +  JSON.stringify(err));
  //Remove the watch since we're having a problem
  navigator.compass.clearWatch(watchID);
  //Clear the heading value from the page
  hi.innerHTML = '<b>Heading: </b>None';
  //Then tell the user what happened.
  if (err.code == CompassError.COMPASS_NOT_SUPPORTED) {
    alert('Compass not supported.');
  } else if (compassError.code ==
    CompassError.COMPASS_INTERNAL_ERR) {
    alert('Compass Internal Error');
  } else {
    alert('Unknown heading error!');
```

```
  }
}
```

Notice that I'm writing to the console much more than I have for most of the other applications I've highlighted in the book. I'm doing this in order to show you a more real-life example of how I build Cordova applications. I use the console a lot to write out all of the application's objects whenever I encounter them; this allows me to better understand what I'm getting from the application so I can more easily troubleshoot problems when they occur.

In my application, I defined an `appName` object:

```
var appName = "Compass - ";
```

Whenever the application writes to the console, I append the value for `appName` to the beginning of every entry:

```
console.error(appName + 'some message');
```

When debugging applications for Android, you can use the monitor application to view the log entries in real time. As shown in Figure 14.4, you can then filter on the value passed in `appName` to see only console messages for the application.

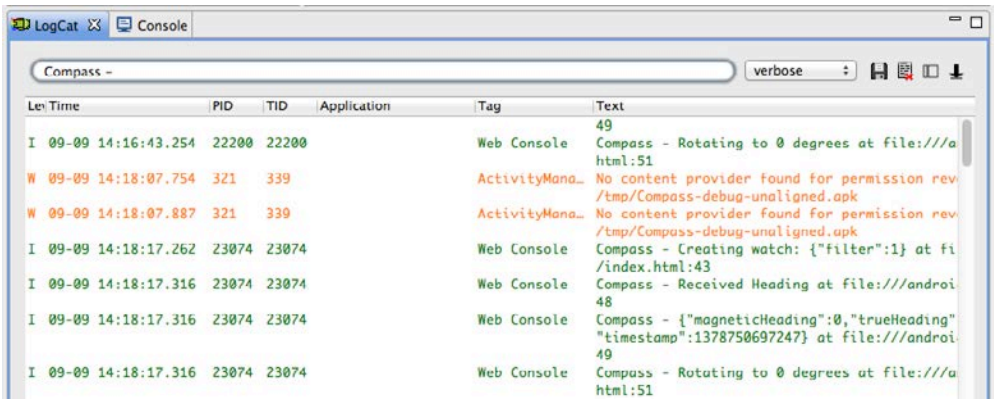



**Figure 14.4** Android Developer Tools LogCat Window

As soon as the application is tested and found to be acceptable, I remove or comment out many of the places where the application writes to the console.

Listing 14.1 shows the complete contents of the application's index.html.

### Listing 14.1: index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-type" content="text/html;
      charset=utf-8">
    <meta name="viewport" id="viewport"
      content="width=device-width, height=device-height,
        initial-scale=1.0, maximum-scale=1.0, user-scalable=no;" />
    <title>Compass</title>
```

```
<link rel="stylesheet" href="jquery.mobile-1.3.2.min.css" />
<script type="text/javascript" charset="utf-8"
  src="jquery-2.0.3.min.js"></script>
<script type="text/javascript" charset="utf-8"
  src="jquery.mobile-1.3.2.min.js"></script>
<script type="text/javascript" charset="utf-8"
  src="jQueryRotateCompressed.js"></script>
<script type="text/javascript" charset="utf-8"
  src="cordova.js"></script>
<script type="text/javascript" charset="utf-8">
  //Some variables used by the application
  var hi, watchID;
  var appName = "Compass - ";

  //Fires whenever there is an error - helps with
  //toubleshooting.
  window.onerror = function(msg, url, line) {
    var resStr;
    var idx = url.lastIndexOf('/');
    if (idx > -1) {
      url = url.substring(idx + 1);
    }
    resStr = 'ERROR in ' + url + ' on line ' + line + ': ' +
      msg;
    console.error(resStr);
    alert(resStr);
    return false;
  };

  function onBodyLoad() {
    document.addEventListener('deviceready', onDeviceReady,
      false);
  }

  function onDeviceReady() {
    console.log('onDeviceReady fired.');
    hi = document.getElementById('headingInfo');
    //Setup the watch
    //Read the compass every second (1000 milliseconds)
    var watchOptions = {
      frequency : 1000,
      filter : 1
    };
    console.log(appName + 'Creating watch: ' +
      JSON.stringify(watchOptions));
    watchID = navigator.compass.watchHeading(onSuccess,
      onError, watchOptions);
  }

  function onSuccess(heading) {
    console.log(appName + 'Received Heading');
    console.log(appName + JSON.stringify(heading));
```

```
        var hv = Math.round(heading.magneticHeading);
        console.log(appName + 'Rotating to ' + hv + ' degrees');
        $("#compass").rotate(-hv);
        hi.innerHTML = '<b>Heading:</b> ' + hv + ' Degrees';
      }

      function onError(err) {
        console.error(appName + 'Heading Error');
        console.error(appName + 'Error: ' +
          JSON.stringify(err));
        //Remove the watch since we're having a problem
        navigator.compass.clearWatch(watchID);
        //Clear the heading value from the page
        hi.innerHTML = '<b>Heading: </b>None';
        //Then tell the user what happened.
        if (err.code == CompassError.COMPASS_NOT_SUPPORTED) {
          alert('Compass not supported.');
        } else if (compassError.code ==
          CompassError.COMPASS_INTERNAL_ERR) {
          alert('Compass Internal Error');
        } else {
          alert('Unknown heading error!');
        }
      }
    </script>
  </head>
  <body onload="onBodyLoad()">
    <div data-role="page">
      <div data-role="header">
        <h1>Compass</h1>
      </div>
      <div data-role="content">
        <div style="text-align : center;">
          <img src="images/compass.png" id="compass" />
          <br />
          <p id="headingInfo">
            <b>Heading:</b> 0 Degrees
          </p>
        </div>
      </div>
      <div data-role="footer" data-position="fixed">
        <h3>cordovaprogramming.com</h3>
      </div>
    </div>
  </body>
</html>
```

# Using Merges

One of the things I haven't done yet is show you how to use the merges capability of the Cordova CLI. In Chapter 6, "The Mechanics of Cordova Development," I explained how it works, but here I'll show you an example.

The compass graphic in my original application was pretty lame; I created it in Microsoft Visio and simply wanted a circle with an arrow pointing north. I got what I wanted, but professional grade it wasn't. For this upgraded version of the application, I wanted something with more panache and found a graphic that, while not technically accurate for my compass, implied a compass theme, so I decided to use it. To leverage the merges aspect of the CLI, I decided to make two versions of the image, one for Android and the other for iOS, using graphics from the PhoneGap website plunked into the center of the compass face. You can see the Android version in Figure 14.5 and the iOS version in Figure 14.6.

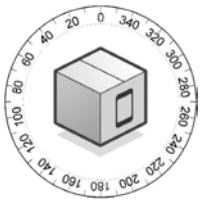



**Figure 14.5** Updated Compass Image for Android


**Figure 14.6** Updated Compass Image for iOS

Once I had the different graphics, it was time to use them to create different versions of the application using the CLI. Notice in the application's code that the compass image is being pulled from images/compass.png. To allow me to use the CLI and merges, I had to create an images folder in the Cordova project's merges/android folder and merges/ios folder. With those folders in place, all I had to do was copy the right image files to the right folders, making sure they both have the same file name. So, assuming my project is in a compass folder, I copied the Android image into compass/merges/android/images/compass.png and the iOS version into compass/merges/ios/images/compass.png.

During the build process, for the Android application, the Cordova CLI copies the file located at compass/merges/android/images/compass.png to compass/platforms/android/assets/www/images/compass.png. For the iOS application, the CLI copies the file located at compass/merges/ios/images/compass.png to compass/platforms/ios/www/images/compass.png.

The result is that I can still maintain one set of code and switch in the right resource files only as needed, depending on the platform. You can see the results of this process in the next section. The use case I used here isn't perfect, but hopefully you can see that, if you've themed your application differently for different platforms, this capability provides you with the means to easily manage the different resources for each platform.

## Testing the Application

To test the application, I can't use the device simulators because compass emulation isn't a standard feature of most simulators. When you try to run the application on an iOS simulator, for example, you will see an error message similar to the one shown in Figure 14.7.


**Figure 14.7** Compass Application Running on an iOS Simulator

Instead, I have to test on real devices. To test on an Android device, I connected my Motorola Atrix 2 to the computer using a USB cable, then issued the following command in a terminal window pointing to the Cordova project folder:

```
cordova run android
```

The Cordova CLI will do its stuff, and 30 seconds to a minute later, the application will launch on the device. Figure 14.8 shows the application running; I've cropped the image to cut away the blank portion of the application screen.

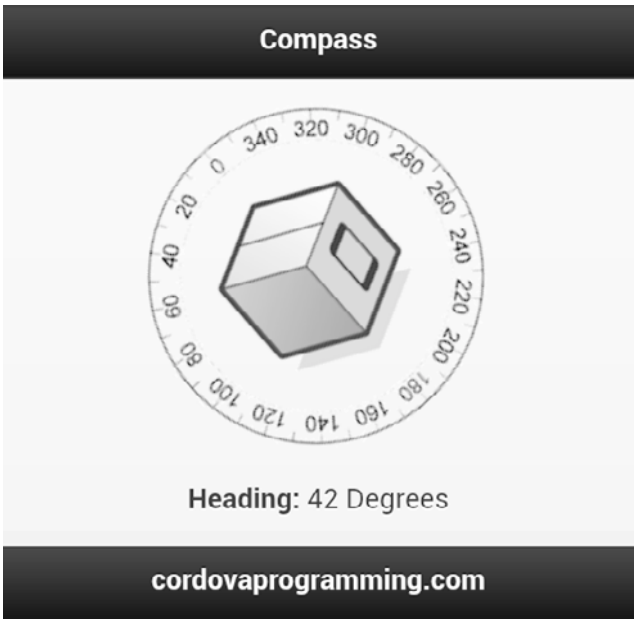



**Figure 14.8** Compass Application Running on an Android Emulator

You should be able to run the application on an iOS device using the CLI, but for some reason, even though it is supposed to work, it's not currently supported, as shown in Figure 14.9. For iOS, `emulate` currently works; `run` does not.

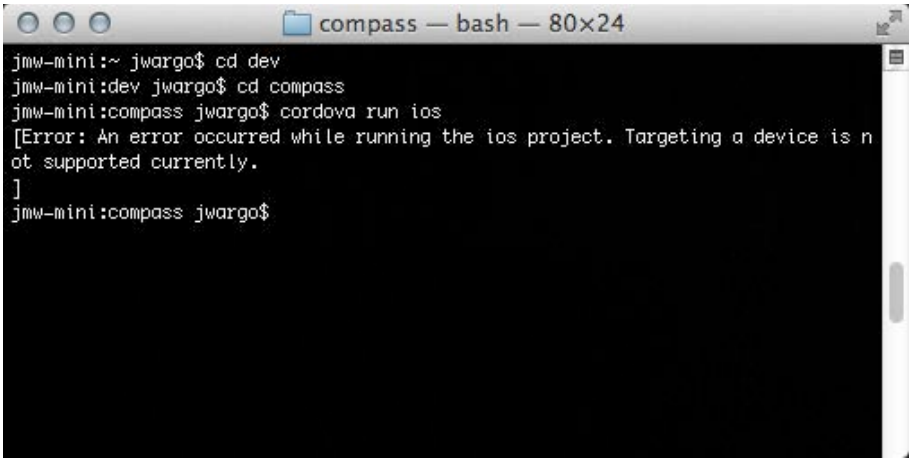



**Figure 14.9** Cordova CLI Error Running Compass on an iOS Device

To run the application on an iOS device, you must first connect the device to the computer system, then open Xcode and open the Cordova project. With the project loaded, you can run the application as you would run any other application. When the application loads, it will display a screen similar to the one shown in Figure 14.10; I've cropped the image to cut away the blank portion of the application screen.

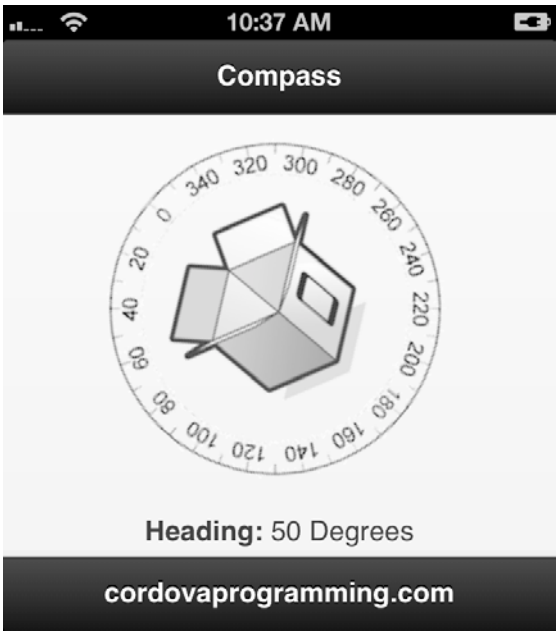



**Figure 14.10** Compass Application Running on an iOS Device

Notice that the compass graphics are different between the two devices, although I didn't code anything to make this happen; this is only one of the cool ways the Cordova CLI enhances cross-platform development.

# Wrap-Up

This chapter wraps up the Cordova programming story by demonstrating how to do cross-platform Cordova development from start to finish. In this chapter, you saw how to create a single application that uses different resources depending on the mobile platform running the application. I also showed you a complete Cordova web application and described each component. You've also seen more details about how to use the Compass API in your applications.

# 15

# Extending Cordova to the Enterprise

As demonstrated throughout this book, Apache Cordova is a great tool for allowing developers to build mobile applications using web technologies. The Cordova team provides the native container that runs the web applications, a suite of JavaScript APIs developers can use to allow the application to interface with native APIs, and tooling that allows developers to manage Cordova applications and Cordova application content.

Cordova doesn't offer any fancy UI controls, and it implements mostly only core APIs, which will someday become web browser standards. The Cordova team focuses on their specific goals and leaves everything else to third-party developers.

If you think about mobile developers, there are different types of developers and developer organizations. There are independent developers, college students, and hobbyists who create apps and games and sell them to consumers. There are professional developers, small, medium, and large organizations focusing on producing productivity applications and games for consumers. There are also many companies focused on building mobile applications for businesses, delivering prebuilt applications that address the needs of many companies, such as customer relationship management (CRM), field service, expense management, time tracking, and more. There are even companies that produce mobile application development platforms, which allow enterprises (a fancy name for business customers) to more easily build mobile applications for their employees, their business partners and even their customers.

In this short chapter, I cover the world of enterprise mobile development, describe the capabilities of mobile application development platforms, and introduce you to the product I am involved with in my day job as a product manager at SAP (www.sap.com). This chapter is in here because most platforms offer some sort of hybrid application approach and several of them use Apache Cordova.

## Mobile Application Development Platforms

Developers building mobile applications for consumers have different needs and requirements than developers building applications for the enterprise. To help these developers, many companies started producing mobile platforms, a suite of tools designed to simplify one or more aspects of mobile development.

For a long time, one of the world's leading analysts categorized mobile development platforms on the basis of the target audience for the applications built using the platform. Platforms for building consumer applications were called mobile consumer application platforms (MCAPs), and platforms for

building mobile applications for the enterprise were called mobile enterprise application platforms (MEAPs—pronounced *meeps*). As the different platforms added features that addressed the needs of both platform types, the industry consolidated the two platform types into a single category called mobile application development platforms (MADPs).

A platform typically includes a server component plus some development tools and a management console. The server component of the platform acts as middleware and manages requests coming in from mobile devices and acts on them, either authenticating the user against a company directory or brokering requests for back-end data (data retrieved from internal or external application servers or databases). Figure 15.1 shows the architecture at a high level.

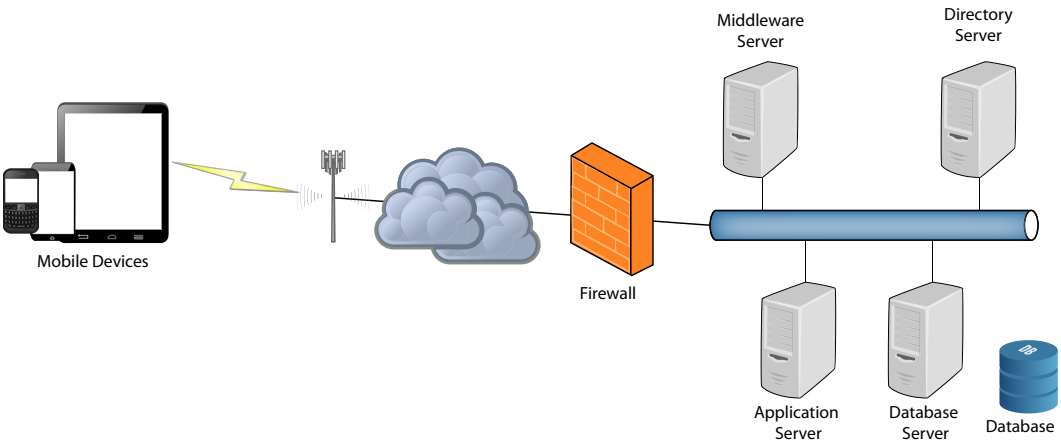



**Figure 15.1** MADP Architecture

A platform's management console is used to administer the platform, manage users and applications, and define the connections to back-end data sources.

A platform's development tools are typically designed to simplify a mobile application's connection to one or more back-end data sources, exposing data from one or more data sources to an application. The tools also help mitigate some of the problems associated with cross-platform mobile development, either enabling a developer to write one application that runs on multiple device platforms or providing a single API that can be consumed across multiple device platforms.

# SAP Mobile Platform

The SAP Mobile Platform (SMP) is an industry leading MADP as recognized by Gartner (www.gartner.com) several years in a row. If your company has a subscription to Garter reports, you can read the latest report at www.gartner.com/id=2570424. The platform offers capabilities for both consumer and enterprise developers and is the consolidation of three leading platforms acquired by SAP: the Sybase Unwired Platform, the Sybase Mobiliser platform, and the Syclo Agentry Platform.

SMP consists of multiple components:

- The SMP server, which provides a suite of services mobile applications can consume such as authentication, provisioning, security, data access, and more

- Multiple mobile application types specifically designed to help address cross-platform development issues

Figure 15.2 illustrates the platform's component architecture.

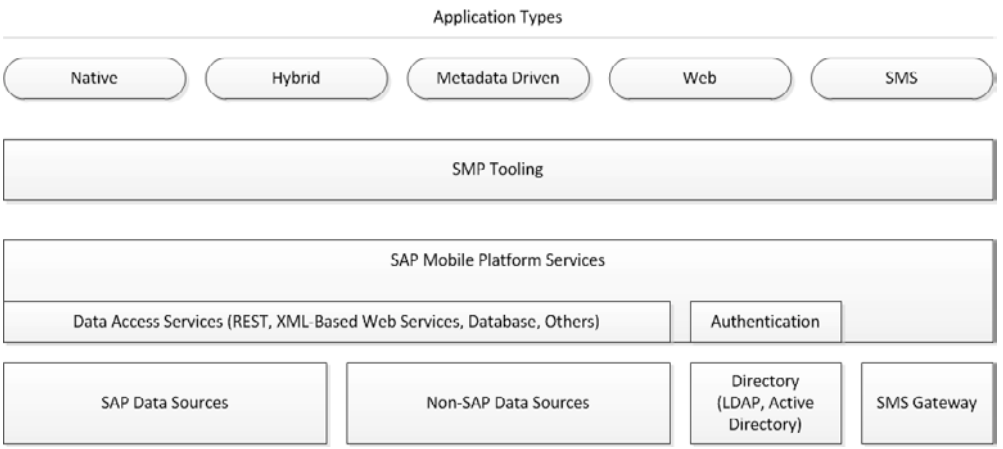



**Figure 15.2** SMP Component Architecture

The server's Data Access Services provide connectivity to both SAP and non-SAP data sources, connecting to those back-end data sources in different ways, as driven by the data source. The data is exposed to mobile devices in OData format (www.odata.org), a standard data format created by Microsoft, published as a standard, and used by many software products. This approach dramatically simplifies how a mobile application accesses data because all data sources are exposed as OData, and therefore the developer only has to learn one way of accessing it.

The SMP server provides many more services than what are shown in the figure. There are services for managing the authentication of a user against a corporate directory (using Microsoft Active Directory or LDAP, for example) as well as services for registering applications, provisioning content and configuration settings, and more.

SMP supports multiple client types, as shown in the figure. Native applications access SMP services using a common API, which is available across multiple device platforms. The server also exposes many of its services as RESTful services, so if you are developing an application that isn't directly supported, there is still an interface your application can use to interact with the server.

For hybrid applications, SMP provides a suite of plugins called Kapsel, which is described in the following section.

For metadata-driven applications, SMP offers the Agentry application player. Developers build applications using the Agentry tools, and the application consists of a meta-application, the description of an application's screens, forms, flows, and more, abstracted into what is essentially XML. When the metadata application is deployed into the Agentry Player application, the application's description is rendered in real-time on the device. This approach provides the ability to build a single version of the application that can be run on multiple mobile device platforms.

# Kapsel

As mentioned earlier, SAP has an industry-leading product in the MADP category called the SAP Mobile Platform. As shown in the previous section, SMP supports hybrid application development through Kapsel, SMP's set of SAP plugins for Cordova. *Kapsel* (or a variant of it) means "container" in several languages. SAP created Kapsel in order to make it easier for a developer to build Cordova applications for SMP. I am currently the product manager for Kapsel.

Kapsel applications are created using the Cordova CLI. You simply create a Cordova application using the procedures highlighted in this book, then add one or more of the Kapsel plugins. Kapsel supports Android and iOS today, but support for other mobile device platforms is planned for future releases.

In the following sections, I highlight a few of the features of Kapsel and how they help make a Cordova application ready for the enterprise. You can read more about Kapsel in my SCN blog at http://scn.sap.com/blogs/johnwargo.

## Registration, Authentication, and Single Sign-on

Enterprise application development is sometimes more difficult than consumer application development because the developer has to deal with security restrictions and back-end systems that often contain proprietary data that must be kept from prying eyes. One of the most complicated problems mobile developers face is application security. The process of ensuring that unauthorized users cannot run the application and that each user of the application is recognized by the server are core requirements for enterprise applications. Basic authentication in the web browser is fine for some consumer applications, but more is needed for enterprise applications.

To help developers deal with this problem, SAP has implemented the Kapsel Logon plugin for Cordova, which allows an application, using essentially a few lines of code, to determine the correct server settings for the environment, connect to the SMP server and register the client application with the server, deal with subsequent locking and unlocking of the application, and even share credentials between multiple applications signed with the same certificate.

Using the Logon plugin, what could be hundreds or potentially thousands of lines of code is abstracted down into just a few simple method calls, which dramatically simplifies the process. The plugin consists of both JavaScript and native application code that manage the whole process.

SMP includes a Client Hub application that, when signed with a company signing key, is deployed to mobile devices and allows Kapsel applications and other applications built with the SMP native libraries to share credentials and stores password information in a secure data vault on the device. This approach makes users happy because they don't have to enter user names and passwords in all of their applications: they do it once, and the information is shared across applications.

## Application Updates

Hybrid applications are great in that you can build them using web technologies, but while it's easy to upgrade a web application by simply posting the updates to a web server, that doesn't work very well for Cordova applications. In the traditional Cordova model, when there are updates to the Cordova application, developers must post the update to the relevant app stores and get their users to complete the update.

With Kapsel, the initial version of an application is packaged with all of its web application content like a normal Cordova application. Whenever updates are available for the web application, those updates can be packaged and deployed to the SMP server for distribution to users.

In this scenario, the developer or an administrator packages the complete, updated web application into an archive (.zip file) and uploads it to the server. When the server receives the file, it unzips everything, then compares this version of the application against the previous version and makes a package containing only the changes from the previous version of the application.

Kapsel includes an AppUpdate plugin, which manages the application update process for the application automatically. A developer adds the plugin to a Kapsel application, then, when the application starts and whenever the application resumes from being suspended, the AppUpdate plugin wakes up and checks with the server to see if there have been any updates to the application. If there is an update available, the server sends the update package, which contains only the changes to the application, to the Kapsel application behind the scenes. Once the Kapsel application has downloaded the entire package, the user is prompted to apply the changes, and the application is restarted once the update is complete.

As you can see, this is a very important feature that can benefit organizations because it can help deal with some of the challenges associated with application lifecycle management. Having the ability to automatically update Cordova applications without requiring an app store update or disturbing the user helps ensure that your users are always running the latest version of your application.

## Offline Access and Data Protection

With consumer applications, critical data is typically maintained on the server and only accessed while online. For enterprise applications, users often need to be able to work with the application data while the device is outside of network coverage or when the user is on a plane and radio is turned off. Because of this requirement, most MADPs provide some sort of offline access for application data. Additionally, while the data is on-device, many organizations often have a requirement that the data be encrypted while in storage.

In Kapsel, SAP has implemented an EncryptedStorage plugin, which adds encryption to the existing key-value pair local storage option provided by the mobile browser and Cordova. Using Kapsel, developers can now easily save their application data locally in an encrypted format and protect it from discovery if a device is lost or stolen.

## Push Notifications

So often, you download an app for your Android or iOS device and during installation you're prompted to allow the application to send push notifications to the device. As cool and interesting as push is for consumer applications, the ability to push information to a mobile device application was available on BlackBerry devices long before Android or the iPhone ever existed. The ability to notify business users that something important is available within a mobile application is a very useful feature.

With Kapsel, SAP includes a Push plugin, which allows an application to register for push notifications. Once the device is registered, the SMP server can send push notifications to the application whenever needed based on the appropriate triggers on the back-end. A corporate sales application can send a notification to a mobile sales application whenever a new PO arrives, or the app can be alerted when an order has been shipped. These types of notifications could be sent to an email account, but it's sometimes more useful to be able to view these things in the context of the application instead.

### Remote Problem Analysis

Enterprise application users have higher expectations regarding the type of support they expect to receive for their corporate issued assets such as mobile phones, tablets, and associated mobile applications. When a mobile application has issues, there could be any number of causes that exist at any point between the mobile device and the back-end server that houses the data. System administrators usually have access to information about everything that is going on within their corporate firewall but don't have the means to understand much about what is going on with the mobile device.

Developers can build a logging mechanism that enables a debug mode within the application, causing the application to collect information about what is happening within it and upload that data to the SMP server for analysis by administrators or developers. Getting the user to enable the debug mode and upload the data when ready can be a challenge.

To help developers with this scenario, Kapsel provides a Logger plugin, which a developer can use to collect information while the application runs and upload the data to the SMP server automatically. Applications make calls to a simple set of methods and passes in content that needs to be written to the log. Administrators can even enable logging from the server without involving the application user in the process.

## Wrap-Up

As you can see, the plugin architecture of Cordova 3 allows organizations to easily extend the capabilities of the Cordova container to include features needed by organizations building mobile applications for their employees, business partners, and even customers.