SAMPLE CHAPTER

Apache Cordova IN ACTION

Raymond K. Camden

MANNING

Apache Cordova in Action

by Raymond K. Camden

Sample Chapter 8

brief contents

8 Creating custom plugins

This chapter covers

- Why you'd write your own plugins
- The basic architecture of plugins
- How to build a sample Android plugin

In the past few chapters you've made use of a variety of plugins. Plugins to use the camera. Plugins to provide notifications. Plugins for globalization. You've also seen that an entire directory, (http://plugins.cordova.io) containing over 700 different plugins, exists.

8.1 Why write your own plugins?

But with all those choices, there's still a strong chance you may encounter a need that isn't covered by an existing plugin. Or perhaps a plugin exists, but it performs poorly and hasn't been kept up to date. Perhaps the iPhone 9 is released with an incredible new feature—a cowbell. This feature is available to folks building native applications, but there's no hook yet to use it with Cordova. A plugin would let you have access to this cool new feature. No matter the reason, one day you may need to create your own plugin.

8.2 Plugin architecture

Let's review the basic architecture of how plugins work, shown in figure 8.1.

In figure 8.1, you can see the basic layout of how a plugin built for Android devices would work. Your code makes a call to a JavaScript library specifically built for that plugin. The JavaScript plugin sends a request to Java code specifically written for the Android platform. That Java code then speaks to the device. Finally, everything gets passed up the chain of calls back to your own code. So that's a plugin for Android, but what about plugins that support multiple device types?

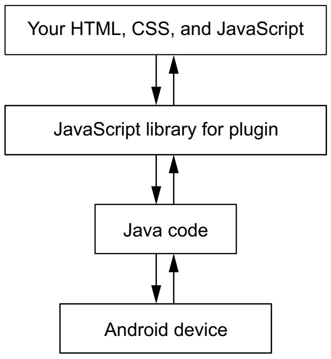


Figure 8.1 An example of an Android plugin

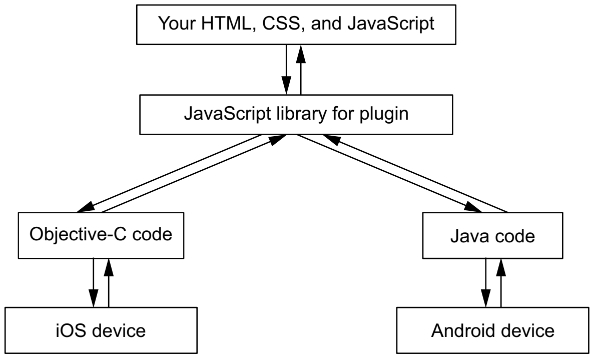


Figure 8.2 A plugin that supports multiple devices

In figure 8.2 the plugin has been updated to support both Android and iOS. In this case, the same JavaScript library for the plugin is used no matter what platform is supported, but on iOS the calls now go to a set of code written in Objective-C or Swift. What's powerful about this setup is that because the same JavaScript library is used, the code within your application remains the same. If the plugin provided an API that looked like `window.cowbell.ring()` and supported both iOS and Android, the developer wouldn't necessarily need to care how it was implemented in Android versus iOS.

Let's look at building a custom plugin and how your Cordova application can make use of it. As before, you're going to use Android versus iOS because anyone on any platform can build it. For folks who want to build an iOS plugin, the process will be similar, outside of the native code of course.

8.3 Building an Android plugin

Before we begin, a word of caution. Android plugins are built using Java. You do *not* need to know how to write Java to continue with this chapter. We're going to keep the code as simple as possible. But unlike HTML, if you make one small mistake in your

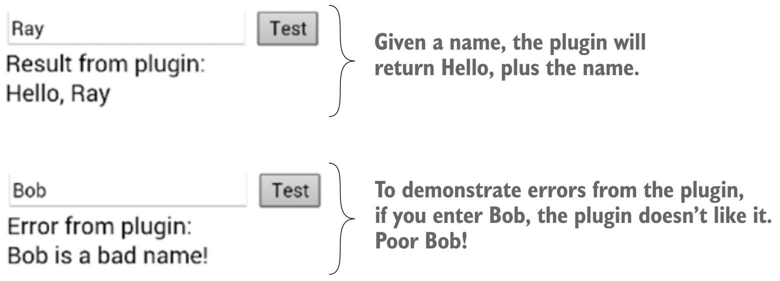

Ray
Test
Result from plugin:
Hello, Ray
Given a name, the plugin will return Hello, plus the name.
Bob
Test
Error from plugin:
Bob is a bad name!
To demonstrate errors from the plugin, if you enter Bob, the plugin doesn't like it. Poor Bob!

Figure 8.3 Two examples of the plugin in action

Java code, your application will fail to build. Later in the chapter you'll see an example of how the process fails when something is wrong in the Java code. The particular plugin you're building here will not make use of any particular device features, but will return a simple "Hello" message. While you wouldn't need a plugin like this in a production application, this chapter will illustrate the process of working with custom plugins. In figure 8.3, you can see an example of the plugin in action.

As noted before, this is a fairly trivial plugin. You'd normally do this in JavaScript alone. The plugin will take any input representing a name and return "Hello", plus the name sent to it. But if the name "Bob" is sent, the plugin will consider this an error and react accordingly. This functionality will demonstrate how plugins can take input, return output, and let the application know that something went wrong. Let's start building.

8.3.1 Setting up the plugin

In all other examples we've worked on in this book, you've created one folder per application. A plugin may not be tied to one application though. You could be building a plugin that you plan on releasing to the world, so, for this example you'll work with two different folders. The first folder will be for the plugin itself. The second folder will be for the application. If you copy the code from the zip file associated with this book, you'll find a plugin folder and a plugintest folder. The plugin folder (figure 8.4) is where you'll build your plugin and the plugintest folder is the actual Cordova application.

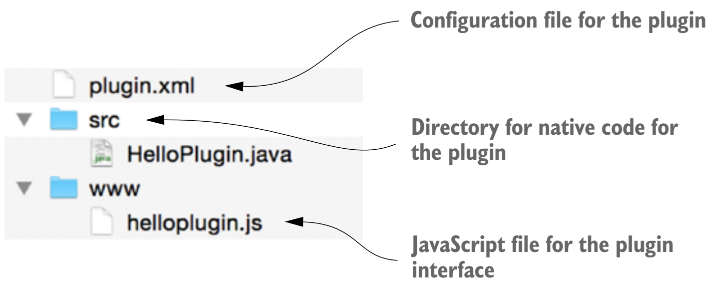

Configuration file for the plugin
plugin.xml
src
HelloPlugin.java
www
helloplugin.js
Directory for native code for the plugin
JavaScript file for the plugin interface

Figure 8.4 Sample plugin folder structure

Plugins consist of three main parts:

- *The native code, HelloPlugin.java*—The native code is the Java, Objective-C, and so forth files that represent the native code for the device. This could be one file or many. For this example it will be one Java file.
- *The JavaScript code, helloplugin.js*—This is where you create the interface that Cordova developers will use to interact with your plugin. How simple or complex this interface is is completely dependent on what your plugin does. As you can imagine for this sample application, the interface will be rather simple.
- *A special file called plugin.xml*—This XML file helps Cordova understand how to package your plugin. It will tell Cordova where the native code exists, where the JavaScript code is, and provide identifying information about the plugin, like the name.

Figure 8.4 demonstrates how your sample plugin looks. If you didn't copy the code from the zip, you should create a folder for these files and call it plugin. (That folder name is a bit vague and you should feel free to name it something more precise if you want, like firstplugin. Just be sure to *remember* what you used.)

8.3.2 Writing the plugin code

Let's begin by looking at the Java code for the plugin shown in the following listing. Again, don't worry if you don't know how to use Java. The code here is rather simple and I'd be willing to bet you can get the gist of what's being done.

Listing 8.1 Source code for the plugin (c8/plugin/src/HelloPlugin.java)

```
package org.camden.plugin;

import org.apache.cordova.CallbackContext;
import org.apache.cordova.CordovaPlugin;
import org.json.JSONObject;
import org.json.JSONArray;
import org.json.JSONException;

public class HelloPlugin extends CordovaPlugin {

    public static final String ACTION_SAY_HELLO = "sayHello";

    @Override
    public boolean execute(String action, JSONArray args,
                           CallbackContext callbackContext)
        throws JSONException {

        if (ACTION_SAY_HELLO.equals(action)) {
            JSONObject arg_object = args.getJSONObject(0);
            String name = arg_object.getString("name");
            //If Bob, we have an error
            if (name.equals("Bob")) {
                callbackContext.error("Bob is a bad name!");
                return false;
```

1. A required class for Android-based Cordova plugins
2. Extends CordovaPlugin
3. Main method when plugin is called from a Cordova application
4. Checks to see which "action" or use of plugin is being executed
5. Gets value passed to plugin
6. Special handling for "Bob"

```
            }
            String result = "Hello, "+name;
            callbackContext.success(result);
            return true;
        }

        callbackContext.error("Invalid action");
        return false;

    }

}
```

7 Returns result

8 Handles invalid uses of plugin

The Java code begins by importing the classes it needs to work properly, but most importantly it grabs Cordova-specific classes required for plugin development **1**. This allows the `HelloPlugin` class to extend `CordovaPlugin` **2**. The `execute()` method **3** is run whenever a Cordova application speaks to the plugin and this is where all your logic will be run. An action string is passed to this method telling the plugin what exactly is being requested. Because a plugin may have a few different features (imagine you added a "goodbye" action to your plugin), the code within the `execute()` method has to see which particular action is being used this time. If the action is `sayHello` **4** then the Java code grabs the input from a JSON data structure passed in **5**. The sample plugin is simple and only has one input value, but if your plugin needed additional inputs, you'd grab them from the JSON as well. There's a special bit of code **6** to handle an input of "Bob," but if the name weren't Bob, you pass back the result **7** using a `callbackContext` object. This comes from the imports done earlier **1** and is part of the Cordova plugin API you get out of the box. If an invalid action was requested, an error **8** is returned to the caller.

For the most part, any Android-based plugin you build will look very similar to this, with the main differences being within the `execute()` method. Even an incredibly complex plugin will still follow this same pattern. Now that you've seen the Java code, let's look at the JavaScript interface shown in the next listing.

Listing 8.2 Source code for the JavaScript interface (c8/plugin/www/helloplugin.js)

```
var helloplugin = {

    sayHello:function(name, successCallback, errorCallback) {

        cordova.exec(
            successCallback,
            errorCallback,
            'HelloPlugin',
            'sayHello',
            [{
```

1 Top-level object for plugin interface

2 sayHello API that Cordova applications will use

3 Handles "speaking" to native plugin

4 When native plugin executes successfully, specifies what to run

5 When native plugin fails to execute successfully specifies what to run

6 Specifies name of plugin that will be used

7 Specifies action to pass to native code

```
                "name": name
            }]
        );
    }

}

module.exports = helloplugin;
```

8 Input passed to plugin

Returns object so that Cordova can make it available to app

Because the plugin is so simple, the corresponding JavaScript code is also rather simple. You begin by creating a top-level object 1 that will store the various APIs. In this case, the plugin's API only supports a single action, `sayHello` 2. The actual communication between JavaScript and the native code is done via `cordova.exec` 3. If you're wondering where this is defined in the listing, it's not. When this code is run in context of a Cordova application, the `cordova` object, and the `exec()` method, will be available. The `exec()` method is passed two arguments that represent the code to run when things work well 4 and what to do if something goes wrong 5. In this example, you expect the user to pass functions in to handle those results. You then tell Cordova what plugin to call 6 and what action 7 to pass to the plugin. As a final step, you pass input 8 to the plugin. What's passed to the plugin will depend on what the plugin is doing. Remember your plugin takes a name and returns a message.

Now that you've seen the native code and the JavaScript API, the final part is the XML configuration that puts it all together, as shown next.

Listing 8.3 Plugin definition file (c8/plugin/plugin.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<plugin xmlns="http://www.phonegap.com/ns/plugins/1.0"
        id="org.camden.plugin.HelloPlugin"
        version="1.0.0">
    <name>HelloPlugin</name>
    <description>Stupid Simple Cordova Plugin</description>
    <author>Raymond Camden</author>
    <keywords>hello,sample</keywords>
    <license>MIT</license>

    <js-module src="www/helloplugin.js" name="HelloPlugin">
        <clobbers target="window.helloplugin" />
    </js-module>

    <!-- android -->
    <platform name="android">
        <config-file target="res/xml/config.xml" parent="/*">
            <feature name="HelloPlugin">
                <param name="android-package"
                ➥ value="org.camden.plugin.HelloPlugin"/>
            </feature>
        </config-file>
```

1 Top-level XML declaration for the plugin

Human-readable data about the plugin 2

3 Controls how the JavaScript API is available to the plugin

4 Specifies how the plugin works with a particular platform

Platform-specific directive for plugin 5

6 Package and class of Java file

```
        <source-file src="src/HelloPlugin.java"
        ➥ target-dir="src/org/camden/plugin" />        ◁ ⑦ Specifies where source code
    </platform>                                           for this platform exists
</plugin>
```

Like any good XML file, the top-level element (`<plugin>`) ❶ wraps the entire content of the file and is required. Within this tag are two required elements, `id` and `version`. The `id` value should be a unique, reverse-domain-style identifier for your plugin. In some ways, this is much like the ID value you use for Cordova applications. The `version` value is also required, and while you can specify whatever version makes sense to you, it *must* follow a particular pattern; for example, *`number.number.number`*, where the first number represents a major version, the next a minor version, and the last a patch.

You've probably seen applications use versioning similar to this. The next version of Windows (at the time I write this chapter) is 10. That would be the major version. It could be written as 10.0.0. This represents the initial release of Windows 10. Let's say a week later a few very small bugs are patched. Microsoft could then release 10.0.1. Because this is an arbitrary decision, they could also name it 10.0.2. If, a few months later, there have been many small fixes and perhaps a new feature added, Microsoft could decide that it's time for a minor version update and use 10.1.0. To an end user, 10.1 should represent a version that's updated from 10.0.2. It should convey an important update. A user may decide not to update to 10.0.2, but 10.1.0 is probably important. (For folks curious to learn more about this type of versioning, see www.semver.org.)

Again, this is arbitrary but your version must follow the same format. Because this is the first release of the plugin, you'll use 1.0.0. If you used 10.0.0, no one would come arrest you, but you should try to be sensible and follow the norms here.

Next come four tags (`<name>`, `<description>`, `<keywords>`, and `<license>`) ❷ that provide human-readable metadata about the plugin. These help describe the plugins to developers who may be using them. The license you select is up to you. For more information about licenses and which may be best for you, see http://opensource.org/licenses. To be clear, a plugin does *not* have to be open source. You don't have to share your code.

The `<js-module>` ❸ block is where you tell Cordova how to make your JavaScript API available. The plugin.xml supports copying files to your project automatically via `<asset>` tags (not used in this plugin), but `<js-module>` is more powerful. This directive will copy your JavaScript code to the Cordova project, and when the application is actually running on a device, it can make the API available automatically to your JavaScript code. The code here uses the top-level `window` variable, but `navigator` could be used as well (as other plugins do).

The first attribute, `src`, points to the JavaScript you created. The inner tag, `<clobbers>`, is where the magic occurs. What happens here is that the module code you defined in your JavaScript will be made available to Cordova developers as window.helloplugin. As an example of this, consider the Camera plugin. When you add it to a project, you suddenly have access to navigator.camera. In a desktop environment, this doesn't exist, but Cordova makes it available via a plugin. In this example, you're telling Cordova you want it to be made available as part of the core browser window object. As you can guess, <clobbers> implies "blow away." If in the future a window.helloplugin object became part of the core browser API, then your code would blow it away, which would be bad, but at that point your Cordova application wouldn't need a plugin! There are other options available, including merging, instead of clobbering, but for your plugin you can use the clobbers to good effect. The name value is only important if you're using Cordova's require functionality, which isn't covered in this chapter.

Next there's a section 4 that defines how the plugin should be integrated with the native platform. Your plugin supports only Android, but if it supported iOS, Windows Phone, and other platforms, you'd have repeated <platform> blocks for each. The particular items within a platform will depend on the platform itself.

For your plugin, you use the <config-file> 5 tag to specify that you want text inserted into an Android configuration file. The text inside this block will be copied into the config file at the root level (due to the parent attribute). In this case, you're simply specifying to Android what your Java code's package and class value are 6. The last part 7 is a pointer to the source code itself for the plugin. The target-dir attribute is specifying where the code should be deployed on the native platform. In this case, you're specifying a subdirectory based on the package used in the Java code. This helps ensure your code doesn't mess with anyone else's code.

There are additional attributes you can specify for the plugin.xml but these are the minimum required to get your project working correctly. A full listing of the XML tags and their meanings can be found at http://cordova.apache.org/docs/en/4.0.0/plugin_ref_spec.md.html.

8.3.3 Working with your plugin

At this point, you've defined your native code (listing 8.1) and your JavaScript API (listing 8.2) and described to Cordova how to integrate the plugin with a project (listing 8.3). Now it's time to use the plugin!

Begin by creating a new project and adding the Android platform. (Remember that your plugin supports only Android.) The first file, index.html in the following listing, will handle the UI for the application.

Listing 8.4 Application HTML file (c8/plugintest/index.html)

```
<!DOCTYPE html>
<html>
  <head>
  <meta charset="utf-8">
  <title>Plugin Demo</title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width">
  <script type="text/javascript" src="jquery.min.js"></script>
  <script type="text/javascript" src="app.js"></script>
  </head>

  <body>
    <input type="text" id="nameField" placeholder="Your Name">
    <button id="testButton">Test</button>

    <div id="results"></div>

    <script src="cordova.js"></script>
  </body>
</html>
```

Name field used for user input

Triggers call to custom plugin

Displays results

As the application is fairly trivial, there isn't much here. You've got a field for users to input their name, a button to click when they're ready, and a div to display the results from the plugin. Now look at the JavaScript in the following listing.

Listing 8.5 Application JavaScript file (c8/plugintest/app.js)

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

function init() {

  $("#testButton").on("touchend", function(e) {
    var name = $("#nameField").val();
    if(name === "") return;

    window.helloplugin.sayHello(name,
        function(result) {
            $("#results").html("Result from plugin:<br/>"+result);
            $("#results").html("Result from plugin:<br/>"+result);
        },
        function(err) {
            $("#results").html("Error from plugin:<br/>"+err);
        }
    );

  });

}
```

❶ deviceready is still required.

❷ This API was defined in JavaScript file for plugin.

❸ Success handler

❹ Failure handler

As a reminder, you still need to wait for deviceready ❶ before doing anything with the device. Just because your plugin is custom doesn't mean this requirement has changed. Because the plugin copied its JavaScript code to window.helloplugin, you can run window.helloplugin.sayHello to execute it from the application ❷. You simply pass in the name to the API and then define a success ❸ and failure ❹ handler. In both cases you'll update a div with what the plugin returned.

8.3.4 Adding the plugin

So you're done, right? Nope! Don't forget you have to actually add a plugin to a project before you can use it. Previously you've seen examples of plugins that were hosted at the main plugin registry (http://plugins.cordova.io). This allowed you to add plugins via their ID, for example:

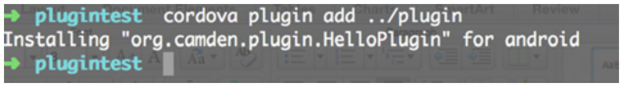
```
cordova plugins add org.apache.cordova.io
```

While you could add your plugin to this registry, most likely you don't want to do that while you're still testing it. Luckily the Cordova CLI also allows you to add plugins from local directories. In case you've forgotten, you can run `cordova help plugin` to see documentation about the plugin command.

If you followed the directory structure from the zip, you should have a folder called plugin for the plugin and one called plugintest for the application. (For the application, you should obviously install the Android platform first.) Both folders should be at the same level (that is, next to each other). Within the plugintest folder, you can then add your plugin with the following command:

```
cordova plugin add ../plugin
```

The result should be no different than adding any other plugin (figure 8.5).



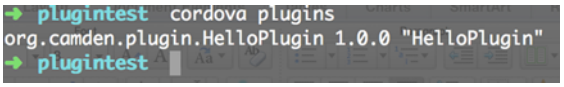
```
➜ plugintest  cordova plugin add ../plugin
Installing "org.camden.plugin.HelloPlugin" for android
➜ plugintest
```

Figure 8.5 Adding the custom plugin from the CLI

At this point you can either emulate or run the project and you should see the application as it was shown in figure 8.3. But what happens if you did something wrong? Perhaps you have a typo in your Java code? You can go back to HelloPlugin.java and on the first line add something that will break, for example:

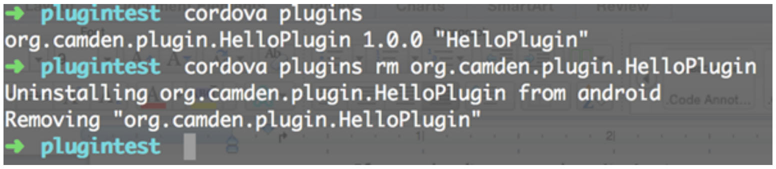
```
<donut tag is not valid java>
package org.camden.plugin;

import org.apache.cordova.CallbackContext;
import org.apache.cordova.CordovaPlugin;
import org.json.JSONObject;
import org.json.JSONArray;
import org.json.JSONException;
```

This brings up an important issue. How do you update the plugin with your application? Changing the source code isn't enough. First you must remove the plugin from the application. While you added the plugin via a directory, you have to remove it by its ID value. If you don't remember it, run `cordova plugins` to list your currently installed plugins (figure 8.6).



```
➜ plugintest  cordova plugins
org.camden.plugin.HelloPlugin 1.0.0 "HelloPlugin"
➜ plugintest
```

Figure 8.6 Currently installed plugins

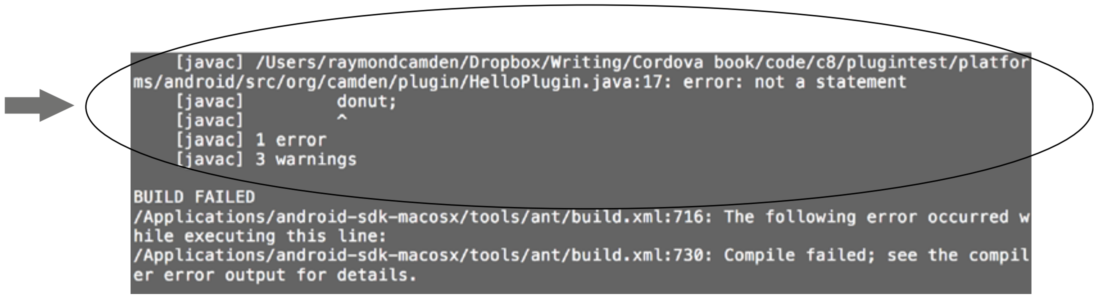


```
➜ plugintest  cordova plugins
org.camden.plugin.HelloPlugin 1.0.0 "HelloPlugin"
➜ plugintest  cordova plugins rm org.camden.plugin.HelloPlugin
Uninstalling org.camden.plugin.HelloPlugin from android
Removing "org.camden.plugin.HelloPlugin"
➜ plugintest
```

Figure 8.7 Removing the custom plugin

Given the output in figure 8.6, you can remove your custom plugin by using an ID value of `org.camden.plugin.HelloPlugin` (figure 8.7).

Whether you're fixing bugs or adding features, you'll need to remove and then again add the plugin as you modify the source code. If you've modified the Java code to intentionally break it, go ahead and add it back. You won't get an error because the Cordova CLI simply copied in the plugin and got it ready. It hasn't actually *built* it yet. To make this happen you can either use the CLI `build` command or emulate or run to the device. Attempting to run your project with the bad plugin will return an error in your console, like that shown in figure 8.8.



```
    [javac] /Users/raymondcamden/Dropbox/Writing/Cordova book/code/c8/plugintest/platfor
ms/android/src/org/camden/plugin/HelloPlugin.java:17: error: not a statement
    [javac]         donut;
    [javac]         ^
    [javac] 1 error
    [javac] 3 warnings

BUILD FAILED
/Applications/android-sdk-macosx/tools/ant/build.xml:716: The following error occurred w
hile executing this line:
/Applications/android-sdk-macosx/tools/ant/build.xml:730: Compile failed; see the compil
er error output for details.
```

Figure 8.8 The failed build of your Cordova application

While it's a bit verbose, when you look at the output in figure 8.8, you can see both that it failed and how it failed. While you develop your plugin, your editor can provide helpful tips and help flag obvious errors before you get that far. To restore the plugin to working order, remove the invalid tag from the Java code, remove the plugin, add it again, and then try running it once more.

8.4 Summary

Let's review the major topics covered in this chapter.

- While numerous plugins exist, there may come a time when you need to build your own. Cordova provides a framework to make this possible.

- Custom plugins consist of the native code, a JavaScript API, and a plugin.xml file that describes how the plugin is integrated into an application.
- During development, a plugin can be added by using the CLI to point directly to the folder.
- Don't forget you have to remove and add the plugin as you develop it.

In the next chapter, you'll learn about ways to package your application, including adding support for splash-screens and icons.

MOBILE DEVELOPMENT

Apache Cordova IN ACTION

Raymond K. Camden

Developing a mobile app requires extensive knowledge of native programming techniques for multiple platforms. Apache Cordova lets you use your existing skills in web development (HTML, CSS, and JavaScript) to build powerful mobile apps. Your apps also get the power of integration with native device features like the camera and file system.

Apache Cordova in Action teaches you how to design, create, and launch hybrid mobile apps people will want to use. With the help of straightforward, real-world examples, you'll learn to build apps from the Cordova CLI and to make use of native device features like the camera and accelerometer. You'll learn testing techniques and discover the PhoneGap Build service and how to submit your apps to Google Play and the Apple App Store. Along the way, this helpful guide discusses mobile app design and shows you how to create effective, professional-quality UI and UX.

What's Inside

- Build mobile apps
- UI, UX, and testing techniques
- Deploy to Google Play and the Apple App Store
- Employ libraries like Bootstrap, jQuery Mobile, and Ionic

Readers should be familiar with HTML, CSS, and JavaScript. No experience with mobile app development needed.

Raymond Camden is a developer advocate for IBM. He is passionate about mobile development and has spoken at conferences worldwide.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/apache-cordova-in-action

Free eBook
SEE INSERT

"A great resource for beginners and experienced programmers alike."
—Ivo Štimac, KING ICT

"Covers Cordova from top to bottom."
—Jérôme Bâton, DRiMS

"Very thorough and well-written. Walks you through everything you need to write Apache Cordova-based applications."
—Becky Huett, Big Shovel Labs

"An easy-to-follow guide through a Cordova project."
—Gregory Murray, Volusion

ISBN 13: 978-1-63343-006-8
ISBN 10: 1-63343-006-5
53999
9 781633 430068

MANNING $39.99 / Can $45.99 [INCLUDING eBOOK]