Cordova - Guide - App Introduction

- Dr Nick Hayward

A brief overview and introduction to Apache Cordova application development.

Contents

- Development paths
  - cross-platform CLI workflow
  - platform-centric workflow
  - careful choice
- Creating a new project
- Cordova App - basic outline & structure
  - anatomy of a template
  - `index.html`
  - `index.js`
- Architecture outline
  - webview
  - native functionality
- API Example Call
- Cross-platform power
- A few initial useful CLI commands

Development paths

Since the advent of version 3.0 of Apache Cordova, it's been possible to develop apps using two distinct workflows and paths. These workflows include,

- Cross-platform CLI
- Platform-centric

cross-platform CLI workflow

The cross-platform CLI allows us to develop apps for a broad, wide-ranging collection of mobile OSs and native devices. This workflow is focused upon leveraging the power of the `cordova` utility, which has become known as the Cordova CLI. This CLI is considered a high-level tool, which has been designed to help abstract many of the development considerations normally associated with multiple platforms. In effect, it has been designed to abstract much of the functionality of lower-level shell scripts, normally required for the other workflow option, *platform-centric*.

e.g.

```
cordova platform add android --save
```

platform-centric workflow

We use *platform-centric* to focus our development on one native platform, for example just Android or just iOS. This approach is slightly more complicated, and involved, but it also allows greater flexibility and customisation for focused, platform-specific app development. For example, if you need to augment your Cordova app with native components developed in the SDK, then this workflow is preferable. This workflow has been tailored for each platform, and as such relies on a set of lower level shell scripts. It also features a custom plugin utility designed to help us apply plugins.

careful choice

Therefore, we will begin our development and journey with Cordova using the *cross-platform* workflow, and the command-line interface. We will then have the option to migrate and use the platform-centric workflow as the nature of our apps becomes more complicated and lower-level.

However, a few notes of caution. Once you switch from the CLI-based workflow to the platform-centric you can't go back easily. This is because the CLI approach keeps a common set of cross-platform source code. On each build, it uses this code to ensure broad support and compliance, and therefore overrides any platform-specific modifications and code. To maintain such modifications, it's best to develop using the platform-centric shell tools.

Creating a new project

So, the basic outline for creating a shell or template for our project is as follows. This project creation is for our newly minted Android environment.

The simplest options are as follows,

```
cordova create basic com.example.basic Basic
cd basic
cordova platform add android --save
cordova build
```

We can then launch this new Cordova project application, for example in the emulator

```
cordova emulate android
```

and you should now see the Cordova start screen for the newly minted project application.

Cordova App - basic outline & structure

So, if we take a closer look at the previous commands we can start to see the initial, basic structure of our Cordova apps.

```
cordova create basic com.example.basic Basic
```

The first parameter of this new project command represents the path of our project. In this instance, we are simply creating a new directory in the current working directory called **basic**. The second and third parameters are initially optional, but it tends to help to define at least the third parameter, which is the visible name of the project. In our case, we've now called our new project **422Basic**.

We can subsequently edit either of the last two parameters in the projects `config.xml` file, which is at the root level of our newly created project.

anatomy of a template

So, our new project includes the following default structure. These are the parts initially of interest to our app development,

```
config.xml
```

```
hooks
  - README.md
platforms
  - android
  - platforms.json
plugins
  - android.json
  - cordova-plugin-whitelist
  - fetch.json
res
  - icon
  - screen
www
  - css
  - img
  - index.html
  - js
```

Initially, our main focus will be within the `www` directory.

As mentioned, the `www` directory will be the initial primary focus. It includes three child directories,

- css
- img
- js

with the all important `index.html` file at the root level as well. In effect, this is the key directory where we store our application's HTML, JavaScript, and CSS code. These directories currently include three primary files, which include

- `index.css`
- `index.js`
- `logo.png`

The `logo.png` image serves as the initial, default Cordova logo. After initial testing, we no longer need this specific logo, and can then customise it for our specific app.

The other primary file, at this stage of development, is `config.xml`. This file stores configuration settings for the application, including settings such as the app's name, a description, author email and name, and the src url for the primary content of the application. This configuration file also specifies permitted domains for our application, which is set to global by default. For development purposes, this is normally OK, but it can quickly become a security concern for production apps. We'll return to this setting as we prepare our apps for distribution and publication.

We are also not limited by the default configuration settings. We can add an element for `preference`, which allows us to specify, for example, whether the app will appear fullscreen by default across all deployed platforms, or whether to hide the accessory bar above iOS and BlackBerry keyboards. There's quite a bit we an customise within such settings.

There are a further three important directories that allow us to manipulate and configure our Cordova application. They include,

- platforms
- plugins
- hooks

The **platforms** directory includes, unsurprisingly, our application's currently supported native platforms. For our current app, this includes Android, which we added as part of the initial project creation.

plugins includes all of the application's used plugins. This is the add-on functionality that we add to a Cordova app to enable access to the device's native functions. After we add a given plugin, we find a matching directory at the root level of this directory. For example, if we add the `media` plugin, we will have a matching media directory in this plugins directory. This directory simply includes the required project code for that given plugin.

hooks contains a set of scripts used to customise commands in Apache Cordova. Effectively, this allows us to customise scripts for code that will execute either before or after a given Apache Cordova command is set to run. These are written for fine tuning, and custom control of Cordova commands, primarily on the developer's local system. For example, we might create a hook that updates project text, such as `development`, to `production` when we set Cordova to build a finished project, and so on...

Let's now move back to the `www` directory, and start working with the three primary files that will initially help us develop our Cordova application. These include

- `index.html`
- `index.js`
- `index.css`

These files will, effectively, become second nature to us as we develop more Cordova applications.

`index.html`

- `index.html` - default template for new project

```
<body>
    <div class="app">
        <h1>Apache Cordova</h1>
        <div id="deviceready" class="blink">
            <p class="event listening">Connecting to Device</p>
            <p class="event received">Device is Ready</p>
        </div>
    </div>
    <script type="text/javascript" src="cordova.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
</body>
```

The default `index.html` page for a Cordova project application is very straightforward. However, there are a few initial points that we need to consider.

- `<div class="app">` is the primary parent section, which acts as the app's container. It contains a child `div`, with the unique ID `deviceready`, and this `div` has two key paragraphs that are triggered relative to state changes in the app. So the app can simply update its state relative to the event being actioned and listened.

The events are monitored and controlled using the app's initial JavaScript. In the `initialize()` method we are calling the `bindEvents()` method, which adds an event listener to this `deviceready` div. When the device is ready, the `onDeviceReady()` method is called, and a subsequent call is then made to the app's object.

So, in real terms, this simply means that when the device is ready the `event listening` paragraph will be hidden, and the `event received paragraph` is now shown. In its default state, this will output to the user that the `Device is Ready`, and that Cordova has now fully loaded.

`index.js`

- `js/index.js`

```
var app = {
    // Application Constructor
    initialize: function() {
        this.bindEvents();
    },
    // Bind Event Listeners
    //
    // Bind any events that are required on startup. Common events are:
```

```
    // 'load', 'deviceready', 'offline', and 'online'.
    bindEvents: function() {
        document.addEventListener('deviceready', this.onDeviceReady, false);
    },
    // deviceready Event Handler
    //
    // The scope of 'this' is the event. In order to call the 'receivedEvent'
    // function, we must explicitly call 'app.receivedEvent(...);'
    onDeviceReady: function() {
        app.receivedEvent('deviceready');
    },
    // Update DOM on a Received Event
    receivedEvent: function(id) {
        var parentElement = document.getElementById(id);
        var listeningElement = parentElement.querySelector('.listening');
        var receivedElement = parentElement.querySelector('.received');

        listeningElement.setAttribute('style', 'display:none;');
        receivedElement.setAttribute('style', 'display:block;');

        console.log('Received Event: ' + id);
    }
};
```

Image - Cordova Splash Screen

Architecture outline

So, as we've now seen, the core architecture for applications developed using Cordova includes,

- HTML5
- CSS
- JS

We can also supplement this core with additional helper files, including JSON (JavaScript Object Notation) resource files.

As part of the architecture of Cordova, to enable access to a device's native functionality JS application objects (or methods) allow us to call Cordova APIs for the chosen mobile OS. If you want to access functionality on an Android device, you use the Cordova Android API. Likewise, for iOS we simply call the API for iOS made available by Cordova.

As noted, we can also develop our own custom plugins to augment and improve support and access to native functionality. We'll cover this aspect of Cordova development later in the semester.

Image of Apache Cordova architecture

The following diagram summarises the core architecture for Cordova application development.

![Apache Cordova Architecture](./media/images/cordova-architecture.png)
Source - [Apache Cordova](https://cordova.apache.org/)
So, the core architecture of Cordova creates a single screen in the native application. This single screen simply contains a WebView, which uses all the device's available screen space. Cordova uses this native WebView to enable loading the application's HTML, and any associated required CSS and JavaScript files.

A WebView is a native view, which allows us to develop using HTML based content. In effect, this native mobile OS view allows us to leverage the power and functionality of a mobile web browser within a contained native app.

webview

Using this WebView, as the native application launches Cordova loads the application's default startup page, in essence its `index.html` page, and then passes control of the app to the native WebView. This allows our user to control the application as normal, and interact with our application in a native manner. They open a Cordova developed app, and get a native app experience. For all intent and purpose, the user will not be able to tell the difference.

This user interaction can include all manner of standard native interaction, for example entering data in input fields, selecting items or buttons, and viewing data and results as requested.

Due to this pattern of interaction with the WebView, a user feels they are simply interacting with a native application. To the user, there is no apparent difference between an app developed with Cordova or the native Android or iOS SDK and APIs.

It's useful to know, even at this early stage, that a WebView thankfully has an implementation in all of the major mobile OSs. For example, Android has a class called

`android.webkit.WebView`

Likewise, iOS references the `UIWebView` which is part of the UIKit framework. Windows refers to a WebView class,

`Windows.UI.Xaml.Controls`

n.b. XAML means Extensible Application Markup Language

native functionality

Cordova provides access to many types of native functionality including, for example, sound and audio controls and recording, camera and photo access and capturing, and much more.

Cordova leverages sets of JavaScript APIs, which allow developers to access this native functionality from within their JavaScript code.

Many different APIs are currently available, each with differing levels of default support for native functionality.

Image - Apache Cordova Native Functionality

The following diagram shows an overview of how this works conceptually,

```
|-------------------------------------------------------------------|
| |---------------------------------------------------------------| |
| |                          WebView                              | |
| |                                                               | |
| | |------------|   |--------------|          |----------------| | |
| | | HTML files |   |   JS files   | -------> |                | | |
| | |------------|   |--------------|          |                | | |
| |                                            |   Cordova JS   | | |
| | |------------|   |--------------|          |      APIs      | | |
| | | CSS files  |   | Helper files |          |                | | |
| | |------------|   |--------------|          |----------------| | |
| |                                                     |         | |
| |-----------------------------------------------------|---------| |
|                                                       |           |
|                                                       v           |
| |---------------------------------------------------------------| |
| |                    Native Device APIs                         | |
| |---------------------------------------------------------------| |
|                                                                   |
|-------------------------------------------------------------------|
```

Source - Apache Cordova

This architecture is a particularly elegant approach to solving the issue of cross-platform development. It allows developers to leverage a unified API interface to perform specific native functions, including camera captures, photo rendering, audio, and so on. This call to the native camera, audio &c. is transparent across the various mobile platforms via the available APIs.

Therefore, to call the native functionality per platform, we simply call the required API for the chosen mobile OS, such as Android or iOS. The API will vary per mobile OS, but the call from Cordova is essentially the same. It's the plugins that give Cordova its power.

API Example Call

So, let's dive into some code and see an example of how to perform a call to a Cordova JavaScript API. For example, let's see how we can make a call to access the camera functionality of a device.

If we want to get a picture from the camera, we call the following using Cordova

```
navigator.camera.getPicture(onSuccess, onFail, { quality: 75,
  destinationType: Camera.DestinationType.DATA_URL
});

function onSuccess(imageData) {
  var image = document.getElementById('Image');
  image.src = "data:image/jpeg;base64," + imageData;
}

function onFail(message) {
  alert('Error: ' + message);
}
```

So, as you should see with this simple code snippet, we are making an initial call to the method `getPicture()` of the `camera` object. This call is performed with 3 parameters,

- `onSuccess` - a callback that allows us to tell the app what to do if the call and returned data is successful
- `onFail` - another callback that tells the app how to handle an error or false return - for example, an error is thrown, and this callback will handle output of a suitable error message
- `quality`

```
quality: 75, destinationType: Camera.DestinationType.DATA_URL
```

This third parameter is slightly different as it contains a JS object with configuration parameters. These two parameters are for `quality` and `destinationType`. Quality can be from `0 to 100`, and the destinationType refers to the required format for the returned data value. This can be set to one of 3 possible values

- `DATA_URL` - the format of the returned image will be a Base64 encoded string
- `FILE_URL` - this returns the image file URL
- `NATIVE_URI` - this refers to the images native URI

Therefore, for this example, if the return is a success we will get a Base64 encoded string of the image we just captured using the camera on the native device.

So, for this example, we are leveraging the power of the Apache Cordova camera plugin code. So, for the Android camera plugin, this gives us the power of the underlying Android class, wrapped in a layer that we can leverage from our JavaScript code. The plugin is written natively for Android, but we access it using JS with Cordova. The camera plugin for other OSs, e.g. iOS, follows the same pattern.

In effect, we issue a call from JS using Cordova to the native code in the plugin. The plugin processes this request, and then returns the appropriate value, either for a success or a failure. In our example, if the request to the camera is successful, the Android plugin will return a string to the JS Cordova client, as requested.

Similarly, we can leverage the same pattern for accessing a camera's functionality with iOS, for example, or another platform assuming there is an appropriate plugin available for that mobile OS. If not, then we can write our own custom plugin.

Cross-platform power

As you can see from this brief example, we can implement capturing a photo from the device's native camera on multiple mobile platforms. With this Cordova plugin architecture, it is not even necessary to understand how the photo capture is implemented or handled natively, as long as we can call the API from the JS interface. The Cordova plugin handles the native calls and processing for each device.

Naturally, this enables us to concentrate on developing our cross-platform apps without separate development for each required mobile OS.

A few initial useful CLI commands

command	example	description
cordova	cordova	general command - outputs overview with 5 categories of information and help
-v	cordova -v	check current installed version of cordova
requirements	cordova requirements	check requirements for each installed platform
create		

	cordova create basic com.example.basic 422Basic	creates new project with additional arguments for directory name, domain-style identifier, and the app's display title
platform add	cordova platform add android --save	specify target platforms, eg: Android, iOS... (n.b. SDK support required on local machine)
platform ls	cordova platform ls	checks current platforms for cordova development on local machine and lists those available
platform remove (platform rm)	cordova platform rm android	remove an existing platform
build	cordova build	iteratively builds the project for the available platforms
build ios	cordova build ios	limit scope of build to a specific platform (useful for testing a single platform...)
prepare	cordova prepare ios	prepare a project, and then open and build &c. with native IDE (eg: XCode, Android Studio...)
compile	cordova compile ios	compile ios specific version of app
emulate	cordova emulate android	rebuilds an app and then launches it in a specific platform's emulator
run	cordova run android	run an app on a native device connected to the local machine
run --list	cordova run --list	check available emulators, e.g. Android AVDs