Cordova - Guide - Custom Plugins

- Dr Nick Hayward

A brief overview of custom plugin development for Cordova.

Contents

intro

Developing custom plugins for Cordova, and by association your apps, is a useful skill to learn and develop. However, it should be noted that it is not always necessary to develop a custom plugin to produce a successful project or application. It will be dependent upon the requirements and constraints of the project itself.

The use and development of Cordova plugins is not a recent addition. They've been available within the project for many years, in one form or another. With the advent of Cordova 3, and the introduction of Plugman and the Cordova CLI, plugins have started to change. They have now become more prevalent in their usage and scope, and their overall implementation has become more standardised.

structure and design

As we start developing our custom plugins, it makes sense to understand the structure and design of a plugin. In effect, what makes a collection of files a plugin for use within our applications.

In essence, we can think of a plugin as a set of files that as a group extend or enhance the capabilities of a Cordova application. We've already seen a number of examples of working with plugins, each one installed using the CLI, and its functionality exposed by a JavaScript interface. Therefore, whilst a plugin could interact with the host application without developer input, the majority of plugin designs provide access to the underlying API to provide additional functionality for an application.

A plugin is, therefore, a collection of contiguous files packaged together to provide additional functionality and options for a given application.

Each plugin includes a `plugin.xml` file, which describes the plugin, and informs the CLI of installation directories for the host application. In effect, where to copy and install the plugin's components. This file also includes the option to specify files per installation platform, again via the CLI. There are many options available within the `plugin.xml` file to help customise installation and setup of a Cordova plugin.

A plugin also needs at least one JavaScript source file. This file is used within the plugin to help define any methods, objects, and properties that are required by the plugin. This source file is, therefore, used to help expose the plugins API to the developer and the host application.

Within our plugin structure, we can easily contain all of the required JS code in one file, or divide into multiple files. Such structure will really depend on the complexity and underlying dependencies within the plugin itself.

For example, we might choose to additionally bundle other useful jQuery plugins, handlebars.js. maps functionality, and so on.

So, beyond the requirement for a plugin.xml and plugin JS source file, the rest of the plugin's structure is simply dependent upon the nature and design of the plugin itself. With the exception of JS-only plugins, we will usually also include one or more native source code files for each of the supported mobile platforms. Each plugin may also include additional native libraries, or required content such as stylesheets, images, media, data files, and so on.

architecture - Android

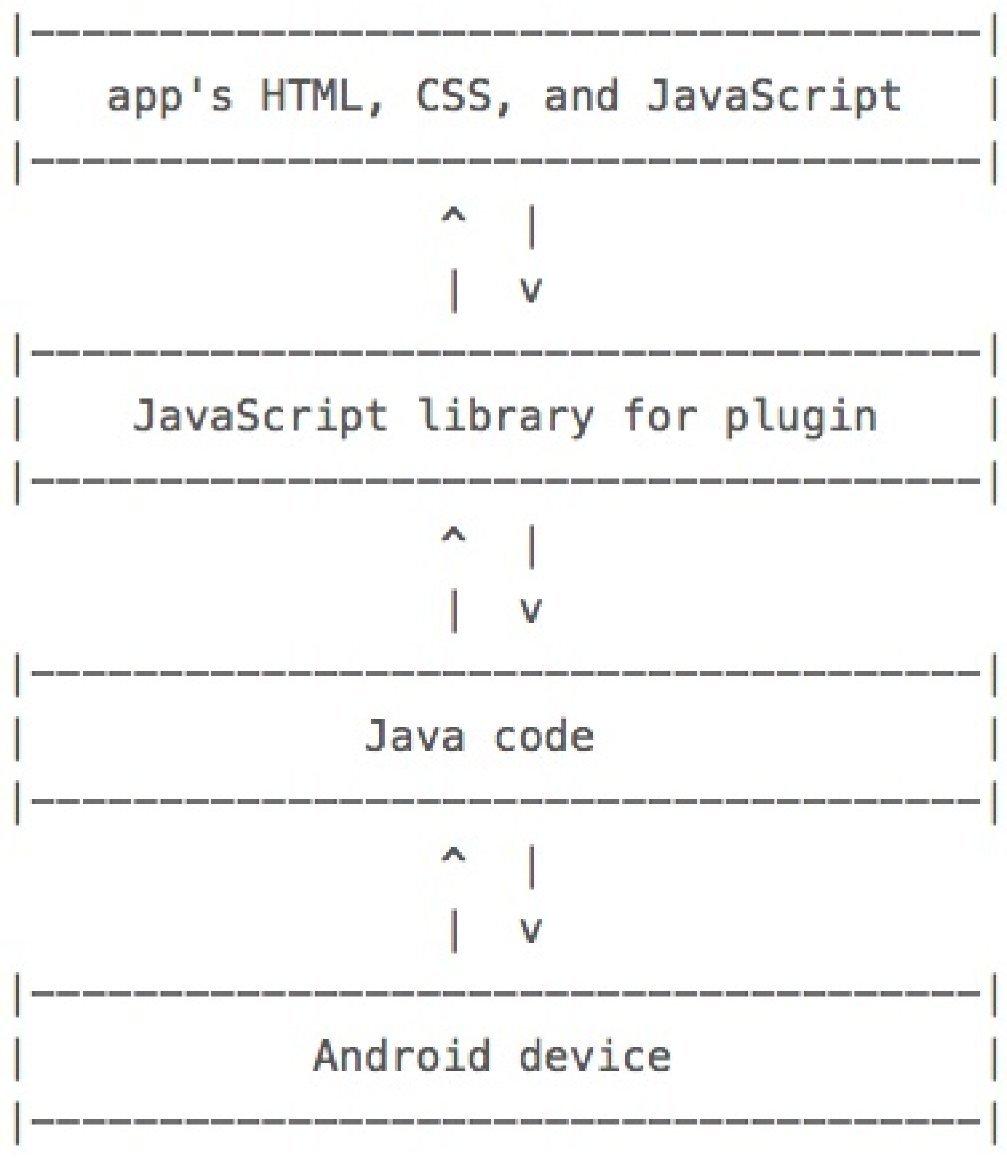
For our plugins, we can choose to support one or multiple platforms for an application.

Therefore, if we consider a plugin for Android, we can follow a useful, set pattern for its development.

The application's code makes a call to the specific JS library, API, for that particular plugin. This plugin's JS then sends a request down the chain to the specific Java code written for supported versions of Android. It is this Java code that actually communicates with the native device. Upon success, any return is then similarly passed up the plugin chain to the app's code for Cordova.

So, we have a bi-directional flow from the Cordova app to the native device, and back again.

Image - Cordova Plugin Architecture - Android



```
|-------------------------------------|
|   app's HTML, CSS, and JavaScript   |
|-------------------------------------|
                 ^  |
                 |  v
|-------------------------------------|
|    JavaScript library for plugin    |
|-------------------------------------|
                 ^  |
                 |  v
|-------------------------------------|
|              Java code              |
|-------------------------------------|
                 ^  |
                 |  v
|-------------------------------------|
|            Android device           |
|-------------------------------------|
```

architecture - cross-platform

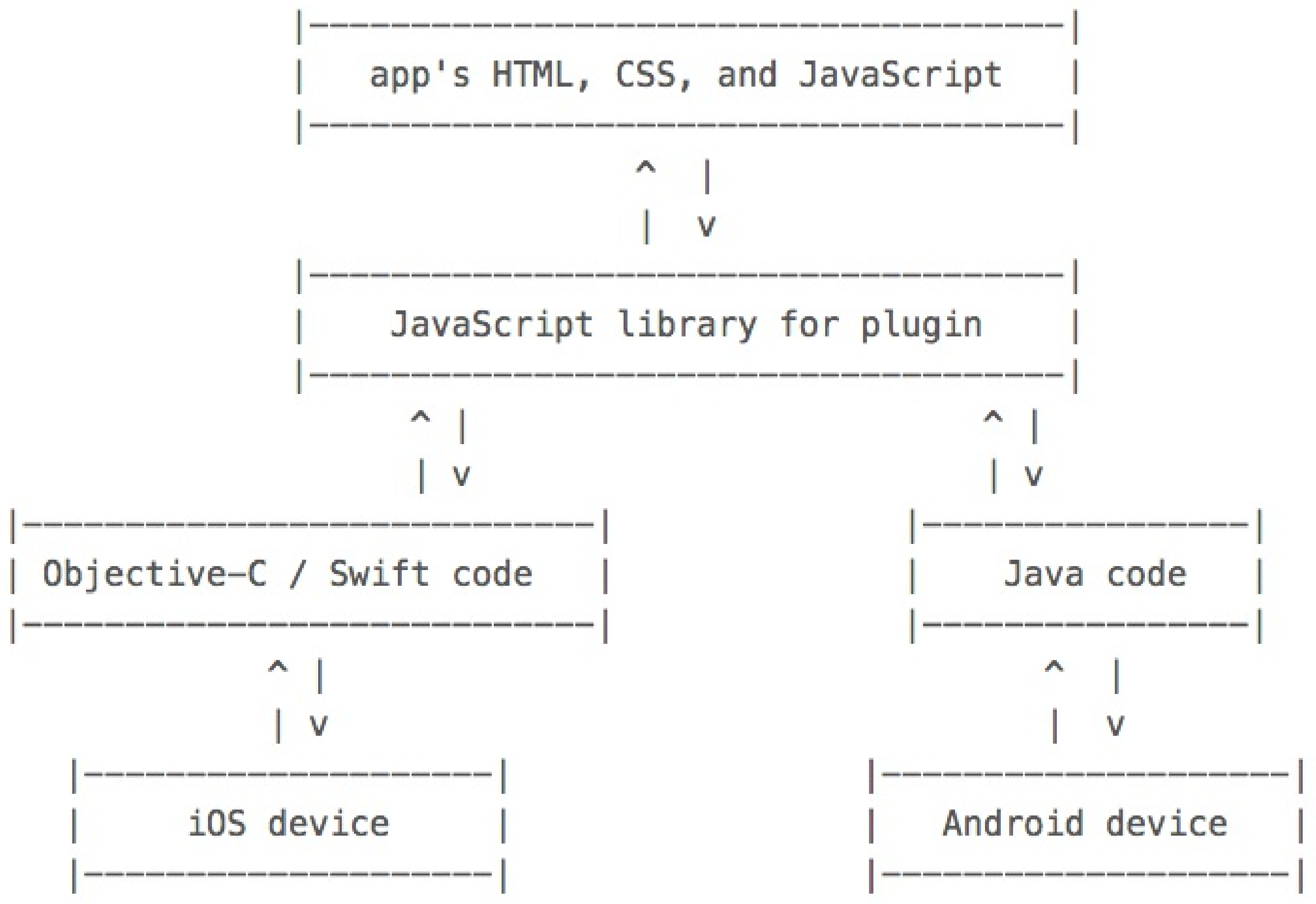
We can, of course, update our architecture to support multiple platforms within our plugin design.

We maintain the same exposed app content, again using HTML, CSS, and JavaScript. We also maintain the same JavaScript library, API for our plugin.

However, we can now add some platform specific code and logic for iOS devices. We add the Objective-C/Swift code at the same level as the Java code for the Android-only plugin, and then connect to the user's native iOS device.

The inherent benefit of this type of plugin architecture is the JavaScript library for the plugin. As we support further platforms, the JavaScript library for the plugin should not need to change per platform.

Image - Cordova Plugin Architecture - Cross-platform


app's HTML, CSS, and JavaScript
JavaScript library for plugin
Objective-C / Swift code
Java code
iOS device
Android device

Plugman utility

For many plugin tasks in Cordova, working with the CLI is more than sufficient. We can manage the installation and removal of plugins, and ensure that our applications are working fine.

As a quick segue, if you are developing Cordova applications using the *platform-centric* workflow, instead of the *cross-platform* option, it might be worth using the *Plugman* tool. We can also use this tool to help with our plugin development,

- it helps create a simple, initial template for building our plugins
- we can add or remove a platform from a custom plugin
- add users to the Cordova plugin registry
- publish our custom plugin to the Cordova plugin registry
- likewise, unpublish our custom plugin from the Cordova plugin registry
- search for plugins in the Cordova plugin registry

We can now use this tool to help us develop our custom Cordova plugin.

So, we now need to install *Plugman* for use with Cordova. We can use NPM to install this tool,

```
npm install -g plugman
```

For OS X, depending upon your installation, you may need to install this tool using the `sudo` command.

After installing Plugman, we can use it to start creating our custom Cordova plugin. It helps us create the shell for our plugin, which will include a basic structure, default files, name, and associated initial metadata.

If we now `cd` to the directory for our new custom plugin, we can create the initial template

```
plugman create --name cordova-plugin-test --plugin_id org.csteach.plugin.Test --
plugin_version 0.0.1
```

With this command, we are setting the following parameters for our plugin

- `--name` = the name of our new plugin
- `--plugin_id` = sets an ID for the plugin
- `--plugin_version` = sets the version number for the plugin

We can also choose whether to add optional commands for additional metadata, such as author or description, and the path to the plugin.

We now have a new directory for our test plugin, which contains the `plugin.xml` file, a standard `www` directory, and a `src` directory.

Using `plugman`, we can also add any supported platforms to our custom plugin,

```
// add android
plugman platform add --platform_name android
// add ios
plugman platform add --platform_name ios
```

Again, this command needs to run from the working directory for the custom plugin. So, we now have our template in place

```
|- plugin.xml
|- src
   |- android
      |- Test.java
|- www
   |- test.js
```

We now have three important files that will help us develop our custom plugin. The `plugin.xml` file for our plugin's general definition, settings etc, `Test.java`, which contains the initial Android code for the plugin, and the plugin's JavaScript API in the `test.js` file.

We can now update our plugin's definition and general settings in the `plugin.xml` file. This configuration file helps us define the general structure of our plugin.

After creating a template for our plugin using Plugman, the Plugin.xml file includes general metadata, details for the JS file for the JS API, and then any platform specific details, including native source code for the plugin.

Within the `<plugin>` element, we can identify our plugin's metadata, including

- `<name>`, `<description>`, `<licence>`, and `<keywords`

Then, we need to clearly define and structure our JS module. This simply corresponds to a JS file for our plugin, but it's intrinsically important as it helps expose the plugin's underlying JS API.

One of the more important, and unusually named, elements is the `<clobbers>` element. It is a sub-element of `<js-module>`, and its main job is to insert the JS object for our JS API into the application's window. In effect, expose the API for development.

We can update the `target` attribute for this element to reflect this required exposure, thereby adding a window value,

```
<clobbers target="window.test" />
```

This object now corresponds to the object defined in the `www/test.js` file, and is exported into the app's window object as `window.jsplugin`. So, our developers will now be able to access the underlying plugin API using this `window.jsplugin` object.

Test plugin 1 - JS plugin

Whilst the vast majority of Cordova plugins include native code, for platforms such as Android, iOS, Windows Phone...it is not a formal requirement for such plugins. We can, of course, simply develop our custom plugin using just JavaScript. We might wish to create a custom plugin to package a JavaScript library, or a combination of libraries, to create a structured plugin for our application.

So, we'll start by creating a simple JavaScript only plugin to help demonstrate plugin development, and general preparation and usage.

We'll need to quickly update our `plugin.xml` file to correctly detail our new plugin,

```
<description>output a daily random travel note</description>
```

Let us now start to modify our plugin's main JS file, `www/test.js`. We'll be using this JS file to help describe the plugin's primary JS interface that a developer can call within their Cordova application. This helps them leverage the options for the installed plugin.

So, by default, when Plugman creates a template for our custom plugin it includes the following JS code for `test.js` file

```
var exec = require('cordova/exec');

exports.coolMethod = function(arg0, success, error) {
    exec(success, error, "test", "coolMethod", [arg0]);
};
```

Part of the default JS code is created based upon the assumption that we are creating a native plugin, for example for Android or iOS. It basically loads the `exec` library, and then defines an export for a JS method called `coolMethod`. Therefore, inside this export structure, the code is trying to execute a native method called `coolMethod()`. As we develop a native code based plugin for Cordova, we would need to provide this method for each target platform.

However, as we are currently working with a JS-only plugin, we can instead simply export a function for our own plugin. In effect, our plugin's primary functionality.

We can now update this JS file for our custom plugin as follows,

```
module.exports.dailyNote = function() {
return "a daily travel note to inspire a holiday...";
}
```

So, to be able to use this plugin, a Cordova application simply calls `test.dailyNote()`, and the note string will be returned.

At the moment, we are simply exposing one test method through the available custom plugin. We could easily build this out, and expose more by simply adding extra exports to the `jsplugin.js` file. We can, of course, also add further JS files to the project, which can export functions for plugin functionality.

At the moment, we are simply exposing one test method through the available custom plugin. We could easily build this out, and expose more by simply adding extra exports to the `test.js` file. We can, of course, also add further JS files to the project, which can export functions for plugin functionality.

We also need to update our plugin to work in an asynchronous manner, a more Cordova like request pattern for a plugin. So, when the API is called, at least one callback function needs to be passed, then the function can be executed, and then passed the resulting value. We can, therefore, update our JS plugin as follows,

```
module.exports = {

  // get daily note
  dailyNote: function() {
      return "a daily travel note to inspire a holiday...";
  },

  // get daily note via the callback function
  dailyNoteCall: function (noteCall) {
    noteCall("a daily travel note to inspire a holiday...");
  }
};
```

We are now exposing a couple of options for requests to the plugin. A Cordova application can now call `dailyNote()`, and get the return result immediately. or it can call `dailyNoteCall()` and get the result passed to the callback function.

We can, therefore, update our JS plugin as follows,

```
module.exports = {

  // get daily note
  dailyNote: function() {
      return "a daily travel note to inspire a holiday...";
  },

  // get daily note via the callback function
  dailyNoteCall: function (noteCall) {
    noteCall("a daily travel note to inspire a holiday...");
  }
};
```

We are now exposing a couple of options for requests to the plugin. A Cordova application can now call `dailyNote()`, and get the return result immediately. or it can call `dailyNoteCall()` and get the result passed to the callback function.

We now need to test this plugin, and make sure that it actually works as planned.

So, the first thing we need to do is create a simple test application. We'll follow the usual pattern for creating our app using the CLI, add our default template files, and then we can start to add and test the plugin files.

```
cordova create customplugintest1 com.example.customplugintest1 customplugintest1
```

We can also add our required platforms,

```
cordova platform add android
```

We can then add our new custom plugin,

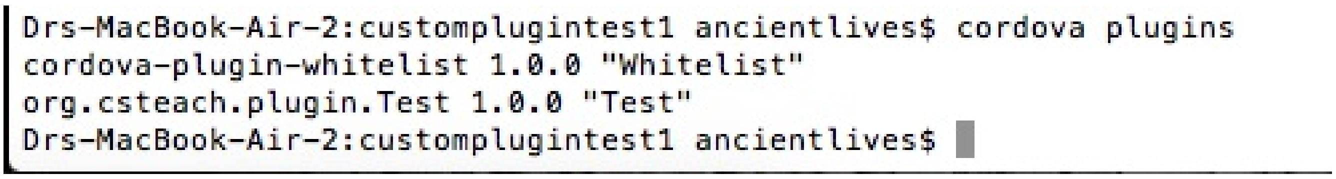
```
cordova plugin add ../custom-plugins/cordova-plugin-test
```

At the moment, we are simply installing this plugin from a relative local directory. When you publish your plugin to the Cordova plugin registry you can, of course, follow the familiar pattern for installing your plugin. For example, the same pattern used to install the device, media, camera etc plugins.

If we now check the installed plugins for our app, we should see our custom plugin installed and ready for use.

```
cordova plugins
```

Image - Cordova Custom Plugin



```
Drs-MacBook-Air-2:customplugintest1 ancientlives$ cordova plugins
cordova-plugin-whitelist 1.0.0 "Whitelist"
org.csteach.plugin.Test 1.0.0 "Test"
Drs-MacBook-Air-2:customplugintest1 ancientlives$
```

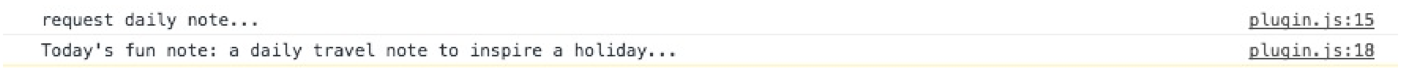
Within our test application, we now need to setup our home page, add some jQuery to handle events, and then call the exposed functions from our plugin.

So, we'll start by adding some buttons to the home page,

```
<button id="dayNote">Daily Note</button>
<button id="dayNoteSync">Daily Note Async</button>
```

We can then update our app's `plugin.js` file to include the logic for responding to button events. We can then call the plugin's exposed functions relative to the requested button.

Image - Cordova Custom Plugin



```
request daily note...                                                plugin.js:15
Today's fun note: a daily travel note to inspire a holiday...       plugin.js:18
```

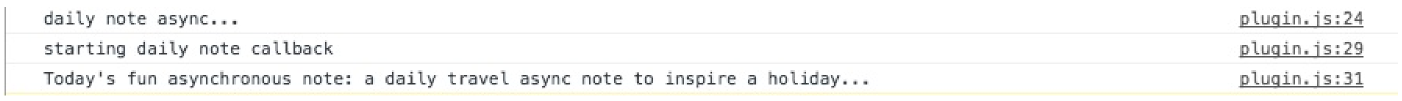
We can also request our asynchronous version of the daily note function from the plugin's exposed API. Again, we can add an event handler to our `plugin.js` file, which will respond to the request for this type of daily note.

```
//handle button press for daily note - async
$("#dayNoteSync").on("tap", function(e) {
  e.preventDefault();
  console.log("daily note async...");
  var noteSync = test.dailyNoteCall(noteCallback);
});
```

- we can then add the callback function

```
function noteCallback(res) {
  console.log("starting daily note callback");
  var noteOutput = "Today's fun asynchronous note: "+ res;
  console.log(noteOutput);
}
```

Image - Cordova Custom Plugin



```
daily note async...                                                            plugin.js:24
starting daily note callback                                                   plugin.js:29
Today's fun asynchronous note: a daily travel async note to inspire a holiday...  plugin.js:31
```

Test plugin 2 - Android plugin

We've now setup and tested our initial JS only plugin application. Whilst this can be a particularly useful way to develop a custom plugin, it's often necessary to create one using the native SDK for a chosen platform. For example, a custom Android plugin.

So, let us now create a second test application, and then start building our test custom Android plugin. We'll start by creating our new test application,

```
cordova create customplugintest2 com.example.customplugintest2 customplugintest2
```

and then, of course, adding our standard template for a basic test application. We'll then start to design and develop our second custom plugin.

So, with our test application setup and ready for use, we can start to consider developing our custom Android plugin. Android plugins are written in Java for the native SDK, and it is your choice, as a developer, the extent to which you'd like to leverage the functionality of the native SDK.

We'll build a test plugin to help us understand the process for working with the native SDK. We'll test processing user input, returning some output to the user, and some initial, basic error handling from a native Java based plugin.

Let us now consider how to setup and configure our application to help us develop a native Android plugin.

As mentioned earlier, there are, effectively, three parts to a plugin that need concern us as developers.

```
|- plugin.xml
|- src
   |- android
      |- Test2.java
|- www
   |- test2.js
```

In particular, we are initially concerned with developing our native plugin code with the `Test2.java` file. So, let's build another plugin for our second test application,

```
plugman create --name cordova-plugin-test2 --plugin_id org.csteach.plugin.Test2 --
plugin_version 0.0.1
```

and we can then add our required platforms for development

```
// add android
plugman platform add --platform_name android
```

This time, we're going to focus upon the Android platform for the plugin.

Let's start to build our native Android plugin.

We'll begin by modifying the `Test2.java` file.

The first thing you'll notice with a Cordova Android plugin are a couple of required classes for working with Android-based plugins,

```
import org.apache.cordova.CordovaPlugin;
import org.apache.cordova.CallbackContext;
```

So, our Java code begins by simply importing required classes for a standard plugin. These include, for example, our Cordova-specific classes, which are required for general plugin development.

With our required classes in place, we can now start to build our plugin's class.

We can start by creating our class, which will extend CordovaPlugin.

```
public class Test2 extends CordovaPlugin {
  ...do something useful...
}
```

We can then start to consider the internal logic for the plugin. Each Android based Cordova plugin requires an `execute()` method. This is run whenever our Cordova application requires interaction or communication with a plugin. As such, this is where all of our logic will be run.

```
@Override
public boolean execute(String action, JSONArray args, CallbackContext
callbackContext) throws JSONException {
    if (action.equals("coolMethod")) {
        String message = args.getString(0);
        this.coolMethod(message, callbackContext);
        return true;
    }
    return false;
}
```

So, this is the default method the Plugman provides each time you create an initial plugin.

So, for the execute method, we are simply passing an action string, which effectively tells the plugin exactly what is being requested.

Our plugin uses this requested action to basically check which action is being used at a given time. For example, plugins will often have different features. The code within our `execute()` method needs to be able to check the required action.

So, if we now update our `execute()` method,

```
@Override
public boolean execute(String action, JSONArray args, CallbackContext
callbackContext)
throws JSONException {
    if (ACTION_GET_NOTE.equals(action)) {
        JSONObject arg_object = args.getJSONObject(0);
        String note = arg_object.getString("note");
    }
    String result = "Your daily note: "+note;
    callbackContext.success(result);
    return true;
}
```

With our updated `execute()` method, if the request action is `getNote`, our Java code will basically grab the requested input from a JSON data structure.

Our current test plugin has a single input value, but if we started to build it out, thereby requiring additional inputs, we could simply grab them from the JSON as well.

You'll also notice that we have some basic error handling. We're able to leverage the default `callbackContext` object provided by the standard Cordova plugin API. In effect, we're able to simply return an error to the caller if an invalid action is requested.

One of the good things about developing an Android plugin for Cordova is the pattern they follow. For the majority of examples, any Android based plugin will look very similar to this example. The main differences will normally be seen within the `execute()` method, which, as noted earlier, is the main engine or driving force behind each plugin.

So, we now have our latest version of the Java code for our plugin.

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

public class Test2 extends CordovaPlugin {

  public static final String ACTION_GET_NOTE = "dailyNote";

    @Override
    public boolean execute(String action, JSONArray args, CallbackContext
callbackContext)
    throws JSONException {
        if (ACTION_GET_NOTE.equals(action)) {
            JSONObject arg_object = args.getJSONObject(0);
            String note = arg_object.getString("note");
        String result = "Your daily note: "+note;
        callbackContext.success(result);
        return true;
    }
    callbackContext.error("Invalid action requested");
    return false;
    }
}
```

Now, we need to update the JavaScript for our plugin, which helps us expose the API for the plugin itself.

So, the first thing we need to do is create a primary, or top level, object for our plugin. We can then use this to store the APIs needed to be able to request and use our plugin.

```
var noteplugin = {
... do something useful...
}

module.exports = noteplugin;
```

Our current API will support one action, our `getNote` action.

```
getNote:function(note, successCallback, errorCallback) {
...again, do something useful...
}
```

For our plugin, the communication between JavaScript and the native code in the Android plugin will be performed using the `cordova.exec` method.

This method is not explicitly defined within our application or plugin. Instead, when this code is, in effect, run within the context of our Cordova application, the `cordova` object itself, and the required `exec()` method, become available. They are, therefore, part of the default structure of a Cordova application and plugin.

We can now add our `cordova.exec()` method.

```
cordov.exec (
...add something useful...
);
```

We can now pass our `exec()` method two required arguments, which simply represent necessary code for success and failure.

In our current example, we are basically telling Cordova how to react to a given user action. We can then tell Cordova which plugin is required, and which to call, and the associated action to pass to the plugin.

We also need to pass any input to the plugin. So, our updated `exec()` method is as follows,

```
cordova.exec(
  successCallback,
  errorCallback,
  'Test2',
  'getNote',
  [{
  "note": note
  }]
);
```

Our JavaScript code should now look as follows,

```
var noteplugin = {

  getNote:function(note, successCallback, errorCallback) {

    cordova.exec(
      successCallback,
      errorCallback,
      'Test2',
      'getNote',
      [{
        "note": note
      }]
    );

  }
}

module.exports = noteplugin;
```

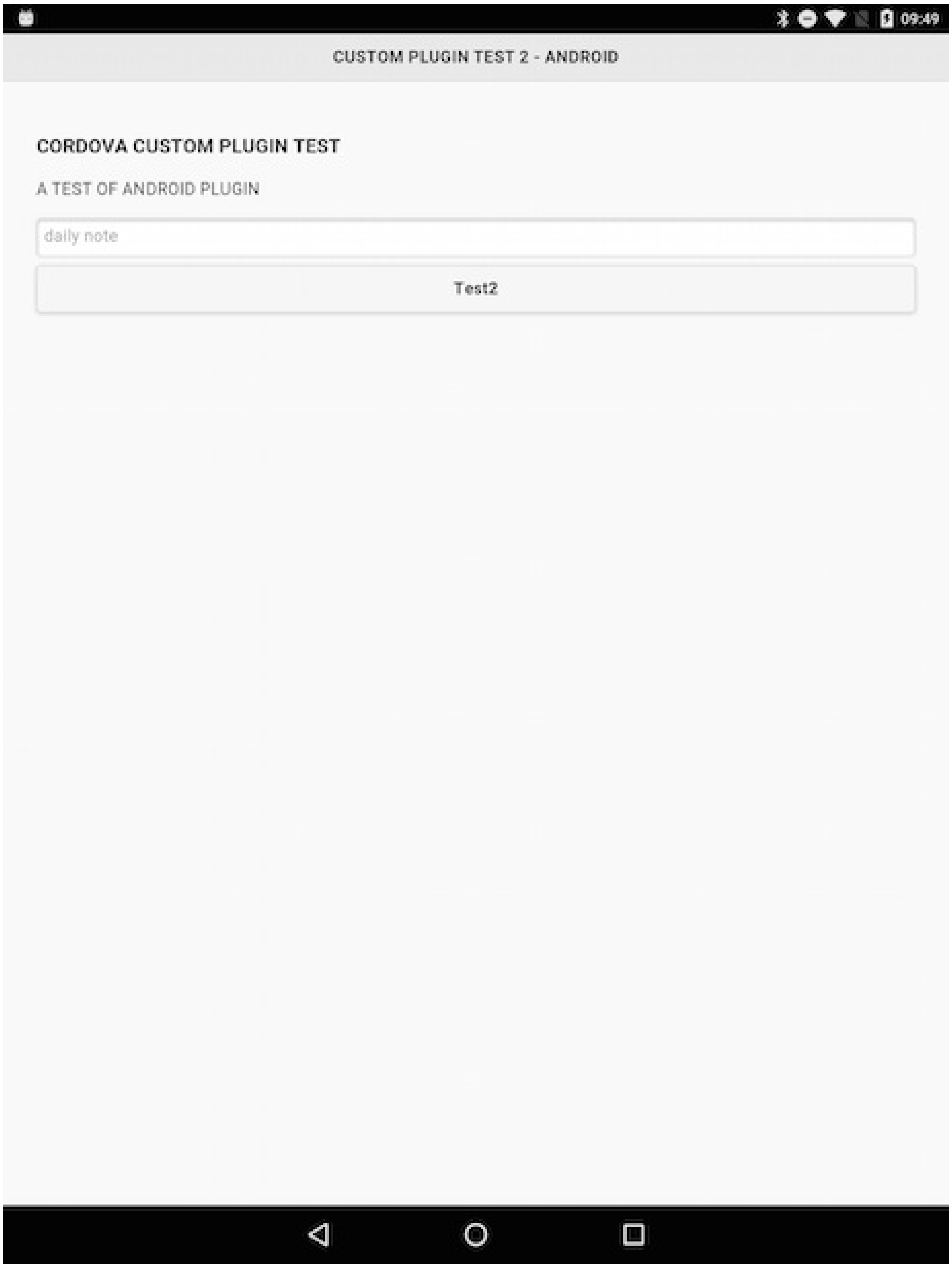
We now need to test our plugin with our application. So, let's update our home page to allow a user to interact with our new custom plugin.

We're going to add an input field for the user requested note, and an accompanying button to submit the request itself.

```
<input type="text" id="noteField" placeHolder="daily note">
<button id="testButton">Test2</button>
```

The exposed plugin API will be able to respond to use the input data from the user to pass to the native Android plugin.

Image - Cordova Custom Plugin 2



To handle this input, and then process for use with our custom plugin, we'll need to update our application's JavaScript.

As with previous Cordova applications, we still need to wait for the `deviceready` event to return successfully, and then we can start to work with our user input and custom plugin.

As we saw with the custom JS only plugin, our native Android plugin's API is similarly exposed using the window object,

```
window.test2
```

So, we can execute it from our application's JS as follows,

```
windows.test2.getNote
```

We can then pass the requested note data to the API, and then define how we're going to work with success and error handlers. In our current test application, we can simply render the returned value to the application's home page.

```
window.test2.getNote(note,
  function(result) {
    console.log("result = "+result);
    $("#note-output").html(result);
  },
  function(error) {
    console.log("error = "+error);
    $("#note-output").html("Note error: "+error);
  }
);
```

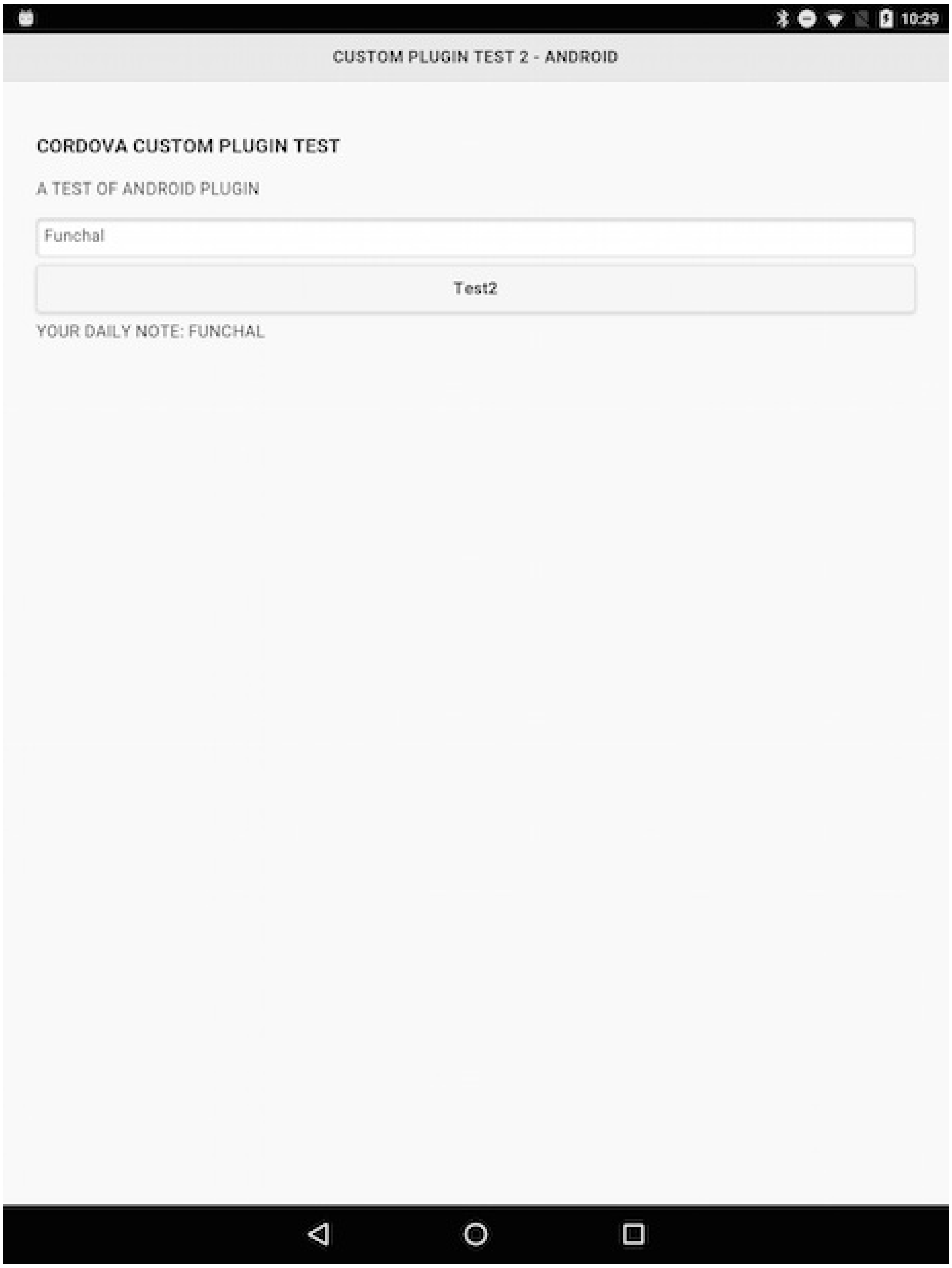
So, our application's JS is now as follows for the `onDeviceReady()` function

```
function onDeviceReady() {

//handle button press for daily note - direct
$("#testButton").on("tap", function(e) {
  e.preventDefault();
  console.log("request daily note...");
  var note = $("#noteField").val();
  console.log("requested note = "+note);
  if (note === "") {
    return;
  }
  window.test2.getNote(note,
    function(result) {
      console.log("result = "+result);
      $("#note-output").html(result);
    },
    function(error) {
      console.log("error = "+error);
      $("#note-output").html("Note error: "+error);
    }
  );
});

}
```

Image - Cordova Custom Plugin 2



Summary of custom plugin development

So, to summarise development of custom plugins for Cordova.

- an initial template for a custom plugin can be created using the *Plugman* tool
- create JS only custom plugins
- create native SDK plugins
  - eg: Android, iOS, Windows Phone...
- custom plugin consists of
  - plugin.xml

    - JavaSript API
    - native code

- create the plugin separate from the application
    - then add to an application for testing
    - remove to make changes, then add again...