appMobi {!}

Native Mobile Application Development Ecosystem

Plugin Documentation

Build 3.4.0   09/14/2011

Revision History

Version	**Date**	**Author**	**Notes**
0.1	2010/08/12	RDS	Initial version
0.2	2011/05/21	Shahab	Updated plugin API
0.3	2011/06/06	Eric	Updated plugin programming requirements. Updated plugin packaging.
0.4	2011/06/12	Shahab	Minor corrections
0.5.1	2011/07/06	Tony	Added XDK Plugin Notes section. Minor corrections and updates.
0.5.2	2011/09/13	RDS, AHS	Minor updates
1.0	2011/09/14	RDS	Added final sections

Table of Contents

1.0 PURPOSE

The purpose of this document is to describe to application programmers the structure of the appMobi plugin system. It will also walk through the sample project to show you how to create your own plugin for the appMobi platform.

2.0 PROGRAMMING PREREQUISITES

In order to develop a plugin, you will be writing code that will run natively on the mobile platform for which you are targeting. In general, it is also preferred (but not required) that you develop your plugin for all of the appMobi supported mobile platforms. We want to extend the cross-platform development goal to all of the appMobi plugins as well.

If you are going to develop your plugin for Android, you will need Eclipse and the Android SDKs. An Android mobile device is also recommended. You can develop Android code on your Windows or MAC computer. You can find more information about developing for Android at http://developer.android.com.

In order to integrate a plugin, the Android plugin should be developed with:

- Android 2.1 or Android 2.2 SDK
- Java 1.6

If you are going to develop your plugin for iOS (iPhone, iPad, iPod Touch), you will need XCode and the iOS SDKs. An iOS mobile device is also recommended. You can develop iOS code on your MAC computer. You can find more information about developing for iOS at http://developer.apple.com.

In order to integrate a plugin, the iOS plugin should be developed with:

- iOS 4.3 SDK (SDKROOT)
- Support for both Armv6 and Armv7 (ARCHS)
- Support for the appropriate platforms: 1 – iPhone/iPod Touch, 2 – iPad, or 1, 2 – iPhone/iPod Touch/iPad. (TARGETED_DEVICE_FAMILY)
- Support the appropriate minimum operating system (3.1.3 or 4.0)
- The plugin should only utilize the following frameworks:
    - Addressbook
    - AddressbookUI
    - AudioToolbox
    - AVFoundation
    - CFNetwork
    - CoreGraphics
    - CoreLocation
    - CoreMedia
    - CoreVideo
    - EventKit
    - EventKitUI
    - Foundation

- MediaPlayer
- MessageUI
- MobileCoreServices
- QuartzCore
- Security
- SystemConfiguration
- UIKit

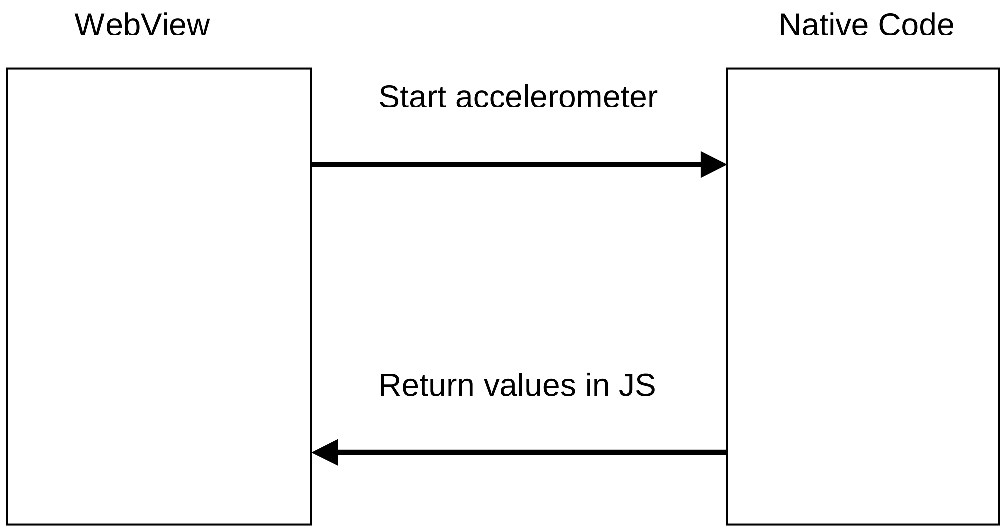
NOTE: If you have requirements that need another framework, please contact us at plugins@appmobi.com and we can work with you.

3.0 PLUGIN ARCHITECTURE

All appMobi apps run in a web-enabled view and communicate with native code through a bridge of some kind. The exact mechanism of this bridge is mobile platform specific. The picture below shows one such high level example of the system.

This example shows a command coming from the WebView requesting that the accelerometer be started. The native code starts up the accelerometer and then uses JavaScript injection to return the accelerometer values to the code running in the WebView.

This is the mechanism we will take advantage of when developing our plugin. The plugin will be able to accept a command and then start some work, maybe in a background thread. In a scenario like this, the thread could then inject a JavaScript event when it is completed. What your plugin does is up to you but the goal of appMobi is to make things easier for the end developer. We want to expose the power of the native APIs to the developer who writes his code in HTML, CSS and JavaScript.



4.0 SAMPLE PROJECT

4.1 Key Classes

There are some key classes that are used to create appMobi plugins. We will discuss each of them and their role in the system. Some of them may be named slightly differently based on the mobile platform but they provide the same functionality (they will be noted accordingly). In the sample project we have created sample classes that derive from these key classes to enable the plugin functionality.

NOTE There are many more classes in the appMobi library besides those included in the sample plugin project. They are specific to the entire appMobi development platform and are not needed to develop plugins.

NOTE In this document we have tried to mark native code objects and functions in bold and JavaScript objects and functions in quotations. Also, if while developing your plugin you feel that you need access to information that we don't expose through our objects, email us at plugins@appmobi.com with a subject about appMobi plugins and we'll consider extending the interfaces as needed.

- appMobi plugin key classes
  - **AppMobiCommand**
    This is the base class for command objects in the native code. For example, we derive all our section specific objects in appMobi from this class (e.g. **AppMobiDevice**, **AppMobiNotification**, **AppMobiPlayer**). Using this class gives automatic reference to the WebView and Application objects that your plugin might need. In the sample, we derive a **MyPluginCommand** from **AppMobiCommand** as a place for you to add a reference or data to anything that you want all your command objects to have access.
  - **AppMobiModule**
    This is the base class for your plugin in the native code. This is the interface through which your plugin can do initialization and other setup. This is the class that sets up the bridge that your plugin will use to communicate with the WebView. In the sample, we derive a **MyPluginModule** from **AppMobiModule** as an example of the kind of setup you may want to do.
  - **AppMobiDelegate**(iOS) or **AppMobiActivity**(Android)
    This is the class for the Application object in the plugin sample project. You will not need to derive anything from this class.
  - **AppMobiWebView**
    This class provides access to WebView that will be included in your app. This class also provides access to information such as the directory that the application is installed in. This might be used to persist data to disk so it can be available across sessions. You will not need to derive anything from this class.
  - **AppMobiDevice**
    This class provides access to some device specific information such as Platform, OS Version, UUID, that you can use in your plugin project. You will not need to derive anything from these classes.

4.2 Index.html

All appMobi apps have an index.html that is the initial page loaded into the WebView. This page is responsible for including the script tag for “appmobi.js” that initializes the bridge between the WebView and the native code. We have included a default index.html that you can use to test your plugin. You will obviously have to modify it accordingly to test your functionality, but this sample is already setup and connected to our sample plugin. Let’s take a look at the HTML.

```
<html>
  <head>
    <meta name="viewport" content="width=320; user-scalable=no" />
    <title>MyPlugin</title>

    <script type="text/javascript" charset="utf-8" src="_appMobi/appmobi.js"></script>
    <script type="text/javascript" charset="utf-8" src="_appMobi/myplugin.js"></script>

    <script type="text/javascript" charset="utf-8">
      function onLoad()
      {
            alert('onload');
            document.addEventListener("appMobi.device.ready",onappMobi,false);
            document.addEventListener("myplugin.ready",onReady,false);
            document.addEventListener("myplugin.workstart",onStart,false);
            document.addEventListener("myplugin.workstop",onStop,false);
      }

      function onappMobi()
      {
            alert('appMobi ready');
      }

      function onStart()
      {
            alert('work started');
      }

      function onStop()
      {
            alert('work stopped');
      }

      function onReady()
      {
            alert('myplugin ready');
      }
    </script>

  </head>

  <body style="background-color:#ffffff" onload="javascript:onLoad();">

      <h3>MyPlugin Test Page</h3>

      <button onclick="javascript:MyPlugin.worker.startWork('hi',
33);">startWork</button>

      <br/><br/>
      <button onclick="javascript:MyPlugin.worker.stopWork();">stopWork</button>

  </body>

</html>
```

As you can see this is a pretty bare bones web page. Let's walk through it and look at some key items. Let's look at the first two script tags. The first includes "appMobi.js" and the second includes "myplugin.js". "appMobi.js" is required in any appMobi app that uses a plugin even if that appMobi doesn't use any appMobi features. "appMobi.js" is required in any appMobi app that uses a plugin even if that app doesn't use any other appMobi features. We will talk about "myplugin.js" in a moment.

Next you'll see that the page adds some JavaScript event listeners. The first is the appMobi event telling us appMobi has been initialized ('appMobi.device.ready'). The others are sample events that the sample plugin fires. This gives us a way to test the communication and also serves as an example of how to fire an event for the completion of a background task that might be performed by your plugin. All the event listeners merely display an alert to give us visual feedback. Lastly you'll notice two buttons. These are buttons that call from the WebView to the native code in your plugin. This should give you an example of how to write some code that exercises your plugin.

4.3 myplugin.js

So let's circle back around to "myplugin.js". This file defines the JavaScript objects that are associated with your plugin. This is also the file that causes the bridge to be created between your plugin's JavaScript objects and your plugin's native code objects. One thing to note in this JavaScript is that we account for difference in the mobile device platform with branching code. Later, when you go to actually publish your plugin, you can either do the same thing or you can have a different version of the JavaScript for each mobile device platform. Now, let's look at some code from the sample "myplugin.js".

```
// Handle Loading of appMobi Library
document.addEventListener("appMobi.device.ready",MyPluginRegister,false);

function MyPluginRegister()
{
        if(AppMobi.isxdk) {
                AppMobi.device.registerLibrary("myplugin",
                "com.mycompany.myplugin.MyPluginModule");
        } else {
                if( AppMobi.device.platform == "Android" )
                        AppMobi.device.registerLibrary(
                        "com.mycompany.myplugin.MyPluginModule");
                if( AppMobi.device.platform == "iOS" )
                        AppMobi.device.registerLibrary("MyPluginModule");
        }
}

function MyPluginLoaded()
{
      while (MyPlugin._constructors.length > 0) {
            var constructor = MyPlugin._constructors.shift();
            try {
                  constructor();
            } catch(e) {
                  alert("Failed to run constructor: " + e.message);
            }
      }

      // all constructors run, now fire the ready event
      MyPlugin.available = true;
      var e = document.createEvent('Events');
      e.initEvent('myplugin.ready',true,true);
      document.dispatchEvent(e);
}

// Global name-prefixed object to store initialization info
if (typeof(MyPluginInfo) != 'object')
    MyPluginInfo = {};

/**
 * This provides a global namespace for accessing information
 */
MyPlugin = {
    queue: {
        ready: true,
        commands: [],
        timer: null
    },
    _constructors: []
};

/**
 * Add an initialization function to a queue that ensures our JavaScript
 * object constructors will be run only once our module has been loaded
 */
MyPlugin.addConstructor = function(func) {
    var state = document.readyState;
    if ( ( state == 'loaded' || state == 'complete' ) &&
                         MyPluginInfo.ready != "undefined" )
      {
            func();
      }
    else
      {
        MyPlugin._constructors.push(func);
      }
};

// Begin Javascript definition of MyPlugin.Worker which bridges to MyPluginWorker
MyPlugin.Worker = function()
{
}

MyPlugin.Worker.prototype.startWork = function(message, interval)
{
        if(AppMobi.isxdk) {
                AppMobi.device.pluginExec("myplugin",
                "com.mycompany.myplugin.MyPluginWorker", "startWork", message, interval);
        } else {
                if( AppMobi.device.platform == "Android" )
```

```
                        MyPluginWorker.startWork(message, interval);
                if( AppMobi.device.platform == "iOS" )
                        AppMobi.exec("MyPluginWorker.startWork", message, interval);
        }
}

MyPlugin.Worker.prototype.stopWork = function()
{
        if(AppMobi.isxdk) {
                AppMobi.device.pluginExec("myplugin",
        "com.mycompany.myplugin.MyPluginWorker", "stopWork");
        } else {
                if( AppMobi.device.platform == "Android" )
                        MyPluginWorker.stopWork();
                if( AppMobi.device.platform == "iOS" )
                        AppMobi.exec("MyPluginWorker.stopWork");
        }
}

MyPlugin.addConstructor(function() {
    if (typeof MyPlugin.worker == "undefined") MyPlugin.worker = new MyPlugin.Worker();
});

// Begin Javascript definition of MyPlugin.Setup which bridges to MyPluginSetup
MyPlugin.Setup = function()
{
    this.ready = null;
    try
      {
            this.ready = MyPluginInfo.ready;
    }
      catch(e)
      {
    }
}

MyPlugin.addConstructor(function() {
    if (typeof MyPlugin.setup == "undefined") MyPlugin.setup = new MyPlugin.Setup();
});
```

Now let’s walk through the code. The first thing that happens is that we add a listener for the appMobi ready event. This tells us that the page is loaded and ready for us to setup our plugin from the JavaScript side. In the listener function (“MyPluginRegister”) we make a call using appMobi to register our JavaScript library. You’ll notice that the name here may be slightly different depending on the platform.

In Android, we need the full package name of the class not just the class. When we call “registerLibrary”, it causes code on the native side to be called. It will create a new **MyPluginModule** object and call its initialize function (more on this later). Next you’ll notice a function called “MyPluginLoaded”. This function is called from **MyPluginModule** after it has completed its initialization. This lets your JavaScript know that your plugin is ready on the native side and that you are ready to finish connecting the interfaces from JavaScript to the native objects.

Before we continue let’s talk about the structure of the objects in “myplugin.js”. We have adopted a very specific structure for how we name everything and where the objects are defined. The reason for this is to make sure that the plugins that different people make won’t interfere. Although it is possible to not use this structure and have everything work, we very strongly recommend that you use the model we have already defined. It will make the development of your plugin easier and help reduce conflicts with other plugins. We will discuss this more in a later section.

Back to the code. Next you see that we define a top level “MyPlugin” object. This is parent of all our JavaScript objects. It has a queue object for sending messages and a constructors array that is used at initialization time. In the code, you’ll see that once **MyPluginModule** calls “MyPluginLoaded” that the function loops over all the constructors and makes all the JavaScript objects for our plugin. When it is done, it fires the ready event for our plugin (“myplugin.ready”).

The next part of the code actually defines the JavaScript objects for our plugin. We have two objects defined: MyPlugin.Worker and MyPlugin.Setup. You’ll notice that these both exist under our top level MyPlugin object. The first thing defined for each object is its constructor. Then we also define the two functions (“startWork”, “stopWork”) for the “MyPluginWorker” object. You’ll notice that the way we call through the “bridge” to the native code is different depending on the mobile device platform.

Lastly, you see that there is an initialization function that is added using “addConstructor”. When “myplugin.js” is initially loaded, this code is run and the function to create each of our JavaScript objects is added to the constructors array. Later, when the “MyPluginLoaded” function is called from the native code, we loop over this array and create all of our JavaScript objects. At this point the plugin is completely initialized and ready to be used.

4.4 MyPluginModule

Now let’s go back and look at what happens between when we call “registerLibrary” in myplugin.js and when the “MyPluginLoaded” function gets called from **MyPluginModule**. This is derived from **AppMobiModule** and must implement two functions: **setup** and **initialize**. These functions are called by the appMobi library and are the place that you will add native code to initialize any data, objects, etc. needed by your plugin.

First let’s look at **setup**. This function is called by the native code before index.html or any JavaScript files are loaded. This includes “myplugin.js”. The purpose of this function to register your command objects to the appMobi structure that bridges calls from the JavaScript to the native code. More about command objects later. Below is the Android version of the code.

```
static private MyPluginSetup mysetup;
static private MyPluginWorker myworker;

public void setup(AppMobiActivity activity, AppMobiWebView webview)
{
      super.setup(activity, webview);

      mysetup = new MyPluginSetup(activity, webview);
      myworker = new MyPluginWorker(activity, webview);

      // You can get the application's shared activity with the following code
      // AppMobiActivity parent = AppMobiActivity.sharedActivity;

      webview.registerCommand(mysetup, "MyPluginSetup");
      webview.registerCommand(myworker, "MyPluginWorker");
}
```

In this code you'll notice that we maintain a pointer to each of our command objects. **setup** is only called one time and that is as the native application is starting. In the code, you'll see that we make each of our command objects and save a reference to them for later. Then we can get a pointer to the shared **AppMobiDelegate** or **AppMobiAcitivty**. These are the parent objects that manage the appMobi library on each of the platforms.. Calling **registerCommand** connects the bridge that allows our plugin's JavaScript to call directly to the native code. You'll notice that it takes a command object and a string. The string is the same name we use in our native calls in the JavaScript. For example, if you look at the code for "AppMobi.Worker.prototype.startWork", you can see that it calls a function startWork on "MyPluginWorker". The name you use in **registerCommand** and the name you use in your plugin.js must match or the bridge won't work.

The other function is **initialize**. This is called when a "registerLibrary" is called from the JavaScript. The **MyPluginModule** object is created and then we call **initialize**. This will happen for every .html page loaded in the users application. This is a chance for your plugin to inject some JavaScript about the initial state of your plugin or about persisted data saved by your plugin. Remember, any JavaScript state will be normally lost on a .html page transition. This is also the location where you will call your "MyPluginLoaded" function in the JavaScript. Below is the iOS version of the code.

```
- (void)initialize:( AppMobiWebView*) webView
{
NSDictionary *props = [mysetup initialize];
 NSMutableString *result = [[NSMutableString alloc] initWithFormat:@"MyPluginInfo = %@;",
      [props JSONFragment]];

 [webView stringByEvaluatingJavaScriptFromString:[result
      stringByAppendingString:@"MyPluginLoaded();"]];
}
```

In this code you'll notice that we use our previously created **MyPluginSetup**(mysetup) object. We can do any resetting or setup that our command objects may need on a per page basis. We can then inject that information to the WebView using JavaScript. Here we build out our "MyPluginInfo" with any initialization data we want to pass along. Note that at this point in time our JavaScript objects have not yet been created so information written here can be accessed in the constructors when they are called. Next the call to "MyPluginLoaded" is also injected. If we refer back to "myplugin.js", this is the call where we run all the constructors and fire the ready event for our plugin. Then the plugin is ready and available for use.

One thing to note here is that multiple **MyPluginModule** objects will be created. If you want to have data or other state information available across the page transitions, then you should save that data in the command objects or make a new utility class that you make in **setup** and save a reference to it so that you can use it in any later calls to **initialize**.

4.5 AppMobiDelegate(iOS) or AppMobiActivity(Android)

This is one of the key objects of the appMobi library. There is only one function exposed for your use from your module object or command objects: **sharedDelegate**(iOS)/**sharedActivity**(Android). **sharedDelegate**(iOS)/**sharedActivity**(Android) provides a static way to get access to this parent object. You will have to call one of these to get this object if you need to access the application's delegate or Activity.

4.6 AppMobiWebView

This is the object that contains the WebView. Three functions are exposed for your use from your module object or command objects: **appDirectory**, **baseDirectory**, **registerCommand**, **appDirectory** gives you access to the path of the native application's documents directory. This can be used as a root location to persist data needed by your plugin. **basetDirectory** gives you access to the path where index.html currently exists. This can be used to save a file inside that directory on disk so that it may then be accessed in a relative fashion by JavaScript in the WebView.

For example, let's say your plugin downloads images from a remote database. You can save that image to this directory in a file called picture.jpg. In the WebView, the user could then add html like this "<img src='picture.jpg'>" and it would show in the WebView. This is because the base index.html page for all appMobi apps is also in this root directory, so relative pathing gains the user access to files you write in the same location.

4.7 MyPluginCommand

All command objects that are used in a call to **registerCommand** must be derived from **AppMobiCommand**. However, we suggest that you first derive your own **MyPluginCommand** class from **AppMobiCommand**. Since this also derives from **AppMobiCommand**, these objects are still valid objects to pass to **registerCommand**. The main reason to do this is that it is highly likely that there will be some shared data or objects that all of your command objects will need access to. By using this class, you will only have to write code that gets those references once because all of your command objects will automatically inherit these references. This is just a good object oriented approach for this part of your plugin. It isn't absolutely required but we strongly suggest it.

You will also notice some differences in how parameter passing works on iOS and Android. In Android, you can just define the number of parameters and their types directly in the function definition. In iOS, you will have to define your function to receive a dictionary object. This dictionary object will have a list of all the parameters (as strings) that were sent in the call. You will have to get each one out and convert them to the type of object you actually need.

On iOS and Android, you can inject JavaScript containing information about the result of the native function they called. You can also inject JavaScript events as we do in this sample. On Android, you can optionally also return some information directly to the JavaScript but mainly just base types like a string, boolean, or int. In order to do this you just define your native function to have a return type of String instead of void.

4.8 Recap

This has been a lot to take in so let's just walk back through all the initialization and interfaces. This doesn't go through every detail but is a high level walk through of what happens with your plugin.

- At application startup only
  1. The appMobi library creates a **MyPluginModule** and calls **setup**
  2. **MyPluginModule** creates all its **MyPluginCommand** based command objects
  3. **MyPluginModule** calls **registerCommand** for each of its command objects

- At every page load (including index.html)
  1. The appMobi library fires "appMobi.device.ready" calling our listener, "MyPluginRegister"
  2. "MyPluginRegister" calls "AppMobi.device.registerLibrary"
  3. The appMobi library creates a **MyPluginModule** and calls **initialize**
  4. **MyPluginModule** does its initialization and calls "MyPluginLoaded"
  5. "MyPluginLoaded" creates all the plugin's JavaScript objects and fires "myplugin.ready"

4.9 Namespacing

In order to avoid collisions between your plugin and other plugins, you will have to namespace yours. This may not be namespacing in the truest sense, but naming all your objects, calls and files in a specific and consistent way accomplishes the same goal. Initially you should just install the sample plugin project and compile and run it and see it work and what it does. At some point after that, we suggest taking a small amount of time and namespace your plugin. Anything that begins with MyPlugin* should be changed to something specific to your plugin or company. This includes changing things like "myplugin.js" to a different name.

Obviously once you change the name of something, you will have to change all those things that reference that. Also, don't forget to change the loading code that runs at startup. There are constants in strings.xml(Android) and info.plist(iOS) that will need modified as well. Once you have namespaced everything you should be able to run the same sample plugin but have it be structured and named for you. Then you are ready to extend and modify it to add the functionality you need.

5.0 XDK PLUGIN NOTES

Providing support to the appMobi XDK for your plugin is similar to developing a plugin for one of our supported mobile platforms; if your starting point is java code or a java library, it may even make sense to start developing your plugin using the appMobi XDK plugin project and later porting it to Android and finally iOS.

5.1 Architecture

The appMobi XDK consists of a web application designed to run in Chrome backed by a Java-based web server that runs on the user's local machine. The web application running in Chrome is equivalent to the mobile web-enabled view, while the web server emulates the native context of the mobile platform. XDK plugins are packaged as jar files and placed in the $WEBROOT/_plugins directory of the appMobi XDK, where they can be loaded and exercised via javascript. When a plugin is loaded or executed, the module instance and instances of any plugin classes are cached in memory. ***Plugins can be unloaded via the Plugins menu on the appMobi XDK icon in the system tray***.

5.2 Sample Project

The appMobi XDK plugin sample project is an Eclipse project consisting of 2 java packages (com.appMobi.XDKPlugin and com.mycompany.myplugin), a small web application (index.html and "myplugin.js"), and some eclipse project files.

The first java package (com.appMobi.XDKPlugin) provides the classes required for integration. **AppMobiCommand** and **AppMobiModule** are as described previously. **AppMobiPluginBridge** provides an interface between the plugin and the emulated native context; similarly to the **AppMobiDevice**, **AppMobiWebView** and **AppMobiDelegate/AppMobiActivity** classes that are provided for supported mobile platforms. **AppMobiCommand**, **AppMobiModule** and **AppMobiPluginBridge** should not be modified. **AppMobiPluginTest** is a simple test harness for debugging and testing the sample plugin within Eclipse; this provides a means of testing the plugin code in isolation and does not use the web application. This can be modified or used as a reference for testing your plugin.

The second java package (com.mycompany.myplugin) is a sample plugin implementation, as described previously for supported mobile platforms. You can use this as the basis for your own plugin development or as a reference template.

The web application is a reference implementation that can be used to exercise the sample plugin in the XDK. For the most part, index.html and "myplugin.js" are as described previously, but there are 2 differences in "myplugin.js" that are worth mentioning. First, "AppMobi.device.registerLibrary" for the xdk takes two parameters: a plugin name and the fully qualified name of the plugin module. The plugin name must be unique; you should pick a descriptive name and if there is a conflict we will work with you to resolve it. The second difference is that you call "AppMobi.device.pluginExec" to access your plugin functionality from javascript. "AppMobi.device.pluginExec" has 3 required parameters: (1) the plugin name, (2) the fully qualified name of the class to access and (3) the name of the method to execute. 0 to n additional parameters can be passed – these will be passed to the native function and typecast appropriately.

The web application can be modified or used as a reference template for use with your plugin.

5.3 Testing with the XDK

If you do not have the appMobi XDK installed, visit http://xdk.appmobi.com and follow the instructions. Make a note of the location you choose on the "Select projects folder for appMobi XDK" dialog: this is the XDK web root ($XDKWEBROOT).

Make a jar file containing only YOUR plugin classes from com.mycompany.myplugin or equivalently renamed package. The com.appMobi.XDKPlugin package classes should NOT be included in your plugin jar. The following steps describe how to do this using Eclipse Export:

1. Right-click on appMobiXDKPlugin project
2. select Export
3. select Java->Jar file
4. deselect the checkbox next to appMobiXDKPlugin (or equivalent if you have renamed)
5. expand the tree control for appMobiXDKPlugin/src (or equivalent if you have renamed)
6. check the checkbox next to appMobiXDKPlugin/src/com.mycompany.myplugin (or renamed equivalent)
7. Only "Export generated class files and resources" should be checked (this should be the default)
8. Note the location of the generated jar file.
9. None of the options are required.
10. Click finish to export the jar file.

Place your plugin jar file in $XDKWEBROOT/_plugins. If you are unsure of the location of $XDKWEBROOT, there are several ways to find it. First, you can look in your java USER.HOME directory (e.g., “C:\Documents and Settings\Username” on Windows XP or “/Users/Username” on OS X) for a file called appMobiToolkit.props; the value of “WEBROOT” is your $XDKWEBROOT. Alternatively, in the XDK, click the “Open project folder” button (it looks like a folder at the top of the screen above the words “Development Tools”), then in the file browser that opens navigate up two levels.

Create a new app in the XDK using the Blank application template. Click the “Open project folder” button and replace the files in the project folder with the web application from the sample project (or your modified version of that web application). Index.html should be placed in the project folder; “myplugins.js” should be placed in the _appMobi subdirectory of the project folder (unless you have modified the script tag for myplugin.js in index.html).

Refresh the project in the XDK and you should be able to exercise the plugin.

6.0 INCLUDING PLUGINS IN AN APPLICATION

6.1 Packaging

In order to publish a plugin for use with the appMobi system, you will have to put together a package containing the appropriate files. iOS native code should be compiled into a static library (.a). Android native code should be packed in a Java archive file (jar).

The plugin files that are required for publication are platform specific. All files must be packaged in a zip compressed file.

For an iOS plugin, please include the following in the zip file:

- All java script files (*.js) for the interface to the plugin.

- A static iOS native library (*.a) that implements the plugin.
- Any other iOS native libraries (*.a) also needed by the plugin.
- All header files (*.h) that define the interface to the plugin.

For an Android plugin, please include the following in the zip file:

- All java script files (*.js) for the interface to the plugin.
- A Java archive file (*.jar) that implements the plugin.
- Any other Java archive files (*.jar) also needed by the plugin.
- All Android native libraries (*.so) that are needed for the plugin.

6.2 Building the plugin library for iOS

- Open the Target node in the Groups and Files window.
- Open and rename the node for MyPlugin to your plugin name.
- Drag all your plugin source files to the Compile Sources folder.
- Make sure you are building for Device
- Set Active Target to your plugin (value set in the second step)
- Set Active Configuration to Distribution
- Build the plugin project
- Your .a lib file will be in the build directory
- Collect your other plugin files as described in 6.1

6.3 Building the plugin library for Android

- Open the src folder and right click on your top level package and select Export.
- Pick Java > JAR file then click Next.
- Only "Export generated class files and resources" should be checked (this should be the default).
- Set the export destination and name the file (e.g. "yourpluginname.jar")
- Click finish and your .jar will be created.
- Collect your other plugin files as described in 6.1