Android - Adding native UI elements

fork and edit tutorial (https://github.ibm.com/MFPSamples/DevCenter/tree/master/tutorials/en/foundation/7.0/adding-native-functionality/android-adding-native-ui-elements-hybrid-applications.html) | report issue (https://github.ibm.com/MFPSamples/DevCenter/issues/new)

Overview

You can write hybrid applications by using solely web technologies. However, IBM MobileFirst Platform Foundation also allows you to mix and match native code with web code as necessary.

For example, use native UI controls, use native elements, provide an animated native introduction screen, etc. To do so, you must take control of part of the application startup flow.

Prerequisite: This tutorial assumes working knowledge of native Android development.

This tutorial covers the following topics:

- Taking control of the startup flow
- Native SplashScreen sample
- Sending commands from JavaScript code to native code
- Sending commands from native code to JavaScript code
- The SendAction sample
- Shared session
- Configuration of the SendAction sample
- Sample application

Taking control of the startup flow

When you create a new hybrid application, MobileFirst Framework generates a main `CordovaActivity` class (`appname.java`) that handles various stages of the application startup flow.

1. The `showSplashScreen` method is called to display a simple splash screen while resources are being loaded. *This is the location that can be modified with any native introduction screen.*
2. To initialize the MobileFirst framework and prepare web resources, the `initializeWebFramework` method is called.
3. As soon as the web framework finished initializing and all resources are ready, the `onInitWebFrameworkComplete` method is called. The value of `WLInitWebFrameworkResult` can be checked for and the application can be started.

Native SplashScreen sample

The NativeUIInHybrid project includes a hybrid application called NativeSplashScreen.

The application contains an Activity, `InitiativeActivity`. This Activity is used to show a simple TextView and a Button as our customized splash screen:

In `onCreate`:

```
setContentView(R.layout.activity_initiative);
startAppBtn = (Button) findViewById(R.id.StartApp);
```

The `onClickListener` method:

```
startAppBtn.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        setResult(Activity.RESULT_OK);
        finish();
    }
});
```

The `TextView` and `Button` objects are also added to the Activity's layout file:

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/InitiativeActivityText" /
>
<Button
    android:id="@+id/StartApp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_centerHorizontal="true"
    android:layout_marginTop="150dp"
    android:text="@string/InitiativeButtonText" />
```

It is also required to add an `Intent` object to the MainActivity before the call to the `initializeWebFramework` method. The intent loads the newly created Activity instead of opening the default MobileFirst splash screen.

```
WL.createInstance(this);
Intent intent = new Intent(this, InitiativeActivity.class);
startActivity(intent);
WL.getInstance().initializeWebFramework(getApplicationContext(), this);
```

Sending commands from JavaScript code to native code

In MobileFirst applications, commands are sent with parameters from the web view (via JavaScript) to an Android native class (written in Java).
You can use this feature to trigger native code to be run in the background, to update the native UI, to use native-only features, etc.

Step 1

In JavaScript, the following API is used: `WL.App.sendActionToNative("doSomething", {customData: 12345});`

The `doSomething` parameter is an arbitrary action name to be used in the native side (see the next step), and the second parameter is a JSON object that contains any data.

Step 2

The native class to receive the action must implement the `WLActionReceiver` protocol:

```
public class ActionReceiver implements WLActionReceiver { }
```

The `WLActionReceiver` protocol requires an `onActionReceived` method in which the action name can be checked for and perform any native code that the action needs:

```
public void onActionReceived(String action, JSONObject data) {
    if (action.equals("doSomething")) {
        // Write your code here...
    }
}
```

Step 3

For the action receiver to receive actions from the MobileFirst web view, it must be registered. The registration can be done during the startup flow of the application to catch any actions early enough:

```
WL.getInstance().addActionReceiver(new ActionReceiver(this));
```

Sending commands from native code to JavaScript code

In MobileFirst applications, commands can be sent with parameters from native Android code to web view JavaScript code.
You can use this feature to receive responses from a native method, notify the web view when background code finished running, have a native UI control the content of the web view, etc.

Step 1

In Java, the following API is used:

```
JSONObject data = new JSONObject();
data.put("someProperty", 12345);
WL.getInstance().sendActionToJS("doSomething", data);
```

`doSomething` is an arbitrary action name to be used on the JavaScript side and the second parameter is a `JSONObject` object that contains any data.

Step 2

A JavaScript function verifies the action name and implements any JavaScript code.

```
function actionReceiver(received){
    if (received.action == "doSomething" && received.data.someProperty == "12345")
{
        //perform required actions, e.g., update web user interface
    }
}
```

Step 3

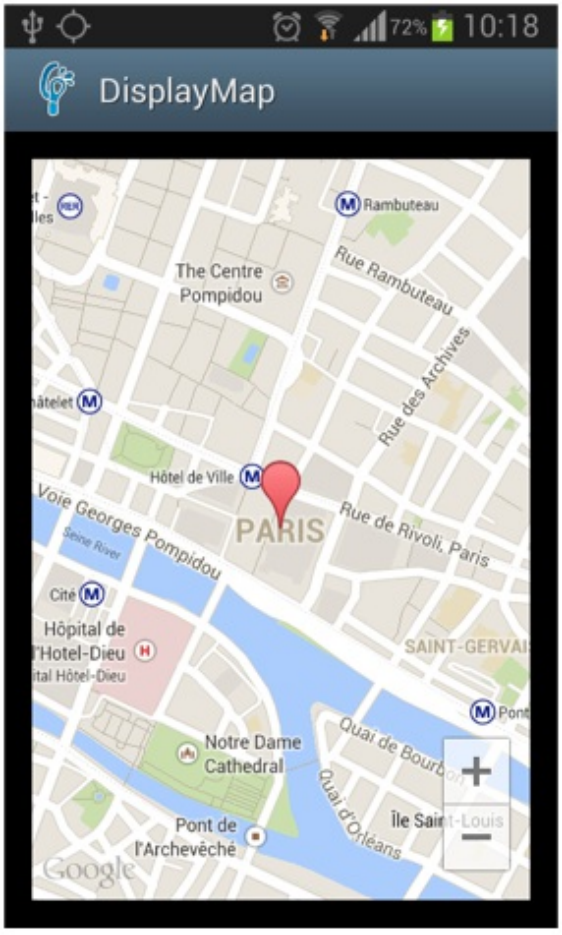
For the action receiver to receive actions, it must first be registered. This should be done early enough in the JavaScript code so that the function can handle those actions as early as possible.

```
WL.App.addActionReceiver ("MyActionReceiverId", actionReceiver);
```

The first parameter is an arbitrary name. It can be used later to remove an action receiver.

```
WL.App.removeActionReceiver("MyActionReceiverId");
```

The SendAction sample



Overview

Download the NativeUIInHybrid project, which includes a hybrid application called SendAction.
This sample uses the MapView and the Google Maps services.

Important: This sample will **NOT** work without some extra steps to install the MapView.

For more information about the MapView and Google Maps services, see the Google Map Android page (https://developers.google.com/maps/documentation/android/).
This sample also includes an additional `FragementActivity` (DisplayMap) to display the requested map. To support `FragementActivity` add the android-support-v4.jar from your Android-SDK extras folder.

HTML

The HTML page shows the following elements:

- A simple input field to enter an address
- A button to trigger validation
- An empty line to show potential error messages

```
<p>This is a MobileFirst web view.</p>
<p>Enter a valid address (requires Internet connection):<br/>
<input type="text" name="address" id="address"/>
<input type="button" value="Display" id="displayBtn"/>
</p>
<p id="errorMsg" style="color:red;"></p>
```

JavaScript

When the button is clicked, the `sendActionToNative` method is called to send the address to the native code.

```
$('#displayBtn').on('click', function() {
   $('#errorMsg').empty();
   WL.App.sendActionToNative("displayAddress", { address: $('#address').val()})
;
});
```

The code also registers an action receiver to display potential error messages from the native code.

```
WL.App.addActionReceiver ("MyActionReceiverId", function actionReceiver(received) {
   if(received.action == 'displayError'){
     $('#errorMsg').html(received.data.errorReason);
   }
});
```

Action Receiver

A new class is required, which implements `WLActionReceiver`.
Before the new class, the `addActionReceiver` method is called to register `ActionReceiver` in the main Activity after the initialization process is complete.

```
public void onInitWebFrameworkComplete(WLInitWebFrameworkResult result) {
   if (result.getStatusCode() == WLInitWebFrameworkResult.SUCCESS) {
      super.loadUrl(WL.getInstance().getMainHtmlFilePath());
   } else {
      handleWebFrameworkInitFailure(result);
   }
   WL.getInstance().addActionReceiver(new ActionReceiver(this));
}
```

The `ActionReceiver` class implements the `onActionReceived` method. This method is used to receive the address that was passed in the JavaScript code.

```
public void onActionReceived(String action, JSONObject data) {
   if (action.equals("displayAddress")) {
      try {
         mAddress = data.getString("address");
      } catch (JSONException e) {
         e.printStackTrace();
      }

      Intent intent = new Intent(parentActivity,DisplayMap.class);
      intent.putExtra("RECEIVED_ADDRESS", mAddress);
      parentActivity.startActivity(intent);
   }
}
```

Displaying the map

After the address parameter is received, a Fragment Activity displays it.
This Activity uses the GoogleMap services and the Google Geocoder to display the requested address on a map.

The address parameter needs to be sent to this Activity in order to display the address in the map.

Shared session

When you use both JavaScript and native code in the same application, you might need to make HTTP requests to MobileFirst Server (connection, procedure invocation, etc.)

HTTP requests are explained in other tutorials about authentication, application authenticity, and HTTP adapters (both for hybrid and native applications).
IBM Worklight Foundation 6.2, and IBM MobileFirst Platform Foundation 6.3 and later, keep your session (cookies and HTTP headers) automatically synchronized between the JavaScript client and the native client.

Configuration of the SendAction sample

To configure the SendAction sample:

1. Deploy the SendAction application sample to MobileFirst Server.
2. Add the Google Play Services.
    1. From **Android SDK Manager > Extras**, add the **Google Play Services** option.
    2. Import Google Play Services as a library to the Eclipse workspace:
        1. Select **File > Import**, select **Android > Existing Android Code into workspace**, and browse to the project at `*android_sdk_location*\extras\google\google_play_services\libproject\google-play-services_lib`
        2. After successfully importing `google-play-services_lib` into the workspace, mark it as an Android library project. To do this, right-click **imported-project > properties > Android** and select the **IsLibrary** checkbox.
    3. Right-click the **generated Android project > properties > Android >** and click **Add**.
        1. In the Project Selection dialog, select the **google-play-services_lib** project and click **OK**.
        2. Click **Apply** and **OK** in the Properties window.
3. Put your GoogleMap API Key in the AndroidManifest file where it says "*Put-Your-GoogleMap-API-key-here*".

4. Copy the `android-support-v4.jar` file from the *`android_SDK_Install_Dir`*`/extras/android/support/v4` directory to the `libs` directory of the generated Android project.

Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v700/NativeUIInHybridProject.zip) the Studio project.