Android - Adding native UI elements

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
- SendAction sample
- Shared session
- 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){<br />
   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");
```

SendAction sample

Overview

Download the NativeUIInHybrid project, which includes a hybrid application called SendAction.

HTML

The HTML page shows the following elements:

- A div that shows the current MobileFirst server URL (currentServerURLDiv)
- A button that triggers the send action to native function
- A div that shows the MobileFirst server connectivity status (ConnectionStatusDiv)

```
<div id="currentServerURLDiv"></div>
   <input type="button" value="Change Server URL" id="changeServerURL" /
>
<div id="ConnectionStatusDiv"></div>
```

JavaScript

First we call the `getServerURL()` function that uses the `WL.App.getServerUrl()` API
To get the current server URL.
Then we update the `currentServerURLDiv` content accordingly.
After that we use `WL.Client.connect()` to connect the MobileFirst server
And use it's `onSuccess/onFailure` callbacks to update the `ConnectionStatusDiv` content.

When the button is clicked, the `sendActionToNative` method is called.
This method has 2 parameters - the first is the requested action - in this case "displayNativeScreen".
The second parameter is a JSON Object to pass data to the native code.

```
$('#changeServerURL').on('click', function(){
   WL.App.sendActionToNative("displayNativeScreen", { requestedNativeScreen: 'ServerURL'})
;
});
```

The code also registers an action receiver to reload the application
After changing the server URL from the native code.

```
WL.App.addActionReceiver ("BackFromNative", function actionReceiver(received){
 if(received.action == "refreshView"){
  WL.Client.reloadApp();
 }
});
```

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 requested action (displayNativeScreen) that was passed in the JavaScript code. We used the JSON object that we passed from the JavaScript code to indicate which native screen do we want to open,
In case we have a few.

```
public void onActionReceived(String action, JSONObject data) {
 // Display Settings Screen
 if(action.equals("displayNativeScreen")){
  try {
   requestedScreen = data.getString("requestedNativeScreen");
  } catch (JSONException e) {
   e.printStackTrace();
  }
  if(requestedScreen.equals("ServerURL")) {
   Intent intent = new Intent(parentActivity, serverurlActivity.class)
;
   parentActivity.startActivityForResult(intent, 1);
  }
 }
}
```

Changing the server URL

The server URL screen (`serverurlActivity`) includes an EditText, which displays the current server URL that you can edit, and a button to save the new server URL. After you press the button, this activity finishes and the `onActivityResult()` method in the Main Activity (SendAction) is called. This method sets the new server URL by using `setServerUrl()` API and sends action to JavaScript to reload the application by using the `sendActionToJS()` API.

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.

Sample application

Click to download (https://github.com/MobileFirst-Platform-Developer-Center/NativeUIInHybrid/tree/release71) the MobileFirst project.