iOS - Adding native UI elements

Overview

While writing hybrid application can be done by using solely web technologies, IBM MobileFirst Platform Foundation also allows to mix & match native code with web code as needed.

For example, use native UI controls, use native elements, provide an animated native introduction screen, etc. In order to do so, taking control of part of the application startup flow is needed. This tutorial assumes working knowledge of native iOS development.

Taking control of the startup flow

When creating a new hybrid application, MobileFirst Studio generates an App Delegate (YourAppName.m) that handles various stages of the application startup flow.

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.*

The `initializeWebFrameworkWithDelegate` method loads the resources that the web view needs to work correctly.
As soon as the web framework initialization finishes, the `wlInitWebFrameworkDidCompleteWithResult` method is called.

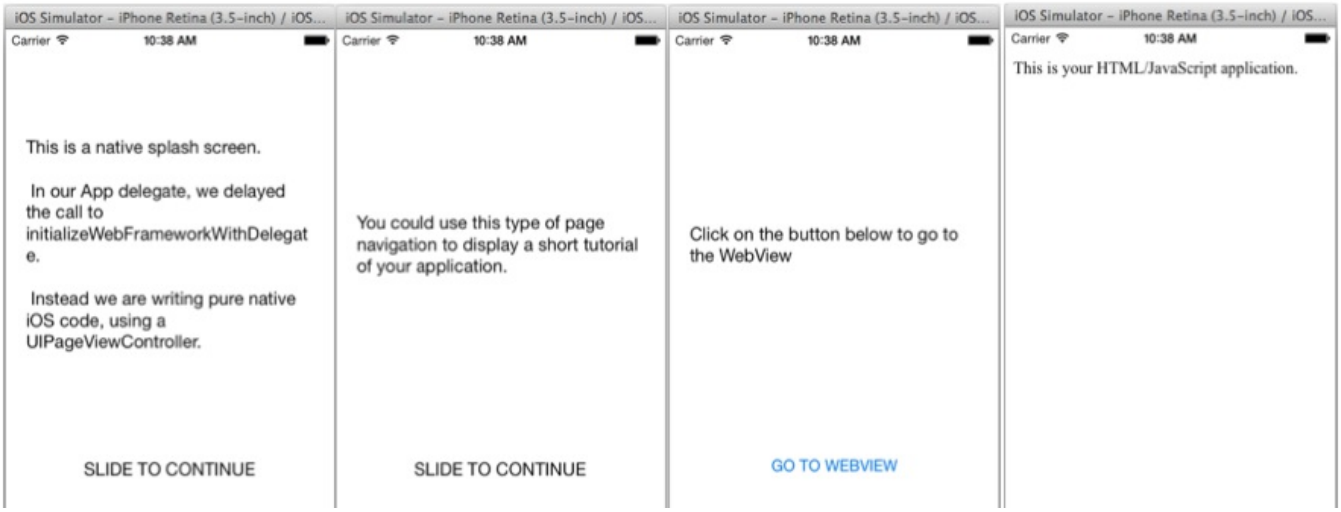
At this point, by default, the application is still displaying the splash screen and no web view is being displayed yet.
The implementation can be modified to handle more of the status codes that are returned by the framework.

By default, a successful load calls the `wlInitDidCompleteSuccessfully` method.

A Cordova web view controller (`CDVViewController` class) is initialized and the start page of the application is set.
The Cordova controller is then added as a child to the root view and displayed on the screen.

If it is decided to implement a custom introduction screen as explained before, consider delaying the display the Cordova view until the user is done with the custom native screen.

Native SplashScreen sample



The NativeUIInHybrid project includes a hybrid application called NativeSplashScreen.
This example uses a Page View Controller to show a sliding introduction to the application. The user interface was created using a Storyboard in XCode.
Two classes, `PageViewController` and `PageContentViewController`, were created to handle the slides.

In the `MyAppDelegate.didFinishLaunchingWithOptions` method, the window is initialized and the PageViewController instance is set as the root of the window.

```
@implementation MyAppDelegate
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launch
Options
{
    //...
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    PageViewController* pageViewController = [[UIStoryboard storyboardWithName:@"Storyboard" bun
dle: nil] instantiateViewControllerWithIdentifier:@"PageViewController"];
    [self.window setRootViewController:pageViewController];
    [self.window makeKeyAndVisible];
    //..
}
```

The `initializeWebFrameworkWithDelegate` method is called from within the `didFinishLaunchingWithOptions` method.
This method initializes the MobileFirst framework in the background and calls the `wlInitWebFrameworkDidCompleteWithResult` method once the framework is initialized.

```
@implementation MyAppDelegate
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launch
Options
{
    //...
    [self.window setRootViewController:pageViewController];
    [self.window makeKeyAndVisible];
    [[WL sharedInstance] initializeWebFrameworkWithDelegate:self];
    return result;
}
```

Inside the `wlInitWebFrameworkDidCompleteWithResult`, method, different scenarios handled depending on the statusCode value of the `WLWebFrameworkInitResult` object.
In this sample, only the common case of the `WLWebFrameworkInitResultSuccess` value is modified.

```
-(void)wlInitWebFrameworkDidCompleteWithResult:(WLWebFrameworkInitResult *)result
{
    if ([result statusCode] == WLWebFrameworkInitResultSuccess) {
        [self wlInitDidCompleteSuccessfully];
    } else {
        [self wlInitDidFailWithResult:result];
    }
}
```

In `wlInitDidCompleteSuccessfully`, a Cordova controller is being prepared but is not displayed yet. Optionally, the frame can be set to itself so that the web view initializes in the background if initialization of the JavaScript code is required to start in the background.

```
-(void)wlInitDidCompleteSuccessfully
{
    // Create a Cordova View Controller
    self.cordovaViewController = [[CDVViewController alloc] init];
    self.cordovaViewController.startPage = [[WL sharedInstance] mainHtmlFilePath]
;
   //This will trigger initialization in the background, optional
    self.cordovaViewController.view.frame = self.cordovaViewController.view.frame;
}
```

In this sample, the `PageViewController` instance ends with a button that triggers a custom method called `onSplashScreenDone` in the AppDelegate.
The `onSplashScreen` custom method resumes where the flow was interrupted and displays the previously initialized Cordova view.

```
-(void)onSplashScreenDone {
    UIViewController* rootViewController = [[UIViewController alloc] init];
    [self.window setRootViewController:rootViewController];
    [self.window makeKeyAndVisible];
    self.cordovaViewController.view.frame = rootViewController.view.bounds
;
    [rootViewController addChildViewController:self.cordovaViewController];
    [rootViewController.view addSubview:self.cordovaViewController.view];
}
```

Send Action From JavaScript to Native

In MobileFirst applications, commands are sent with parameters from the web view (via JavaScript) to a native class (written in Objective-C).
This feature can be used 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});
```

`doSomething` 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:

```
@interface MyReceiver: NSObject <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:

```
-(void) onActionReceived:(NSString *)action withData:(NSDictionary *) data {
    if ([action isEqualToString:@"doSomething"]){
      // perform required actions, e.g., update native user interface
    }
}
```

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 sharedInstance] addActionReceiver:[[myReceiver alloc] init]];
```

Send Action From Native to JavaScript

In MobileFirst applications, commands can be sent with parameters from native Objective-C code to web view JavaScript code.
This feature can be used 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 Objective-C, the following API is used:

```
NSDictionary *data = @{@"someProperty": @"12345"};
[[WL sharedInstance] sendActionToJS:@"doSomething" withData:data];
```

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

Step 2
A JavaScript function, which verifies the action name and implements any JavaScript code.

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

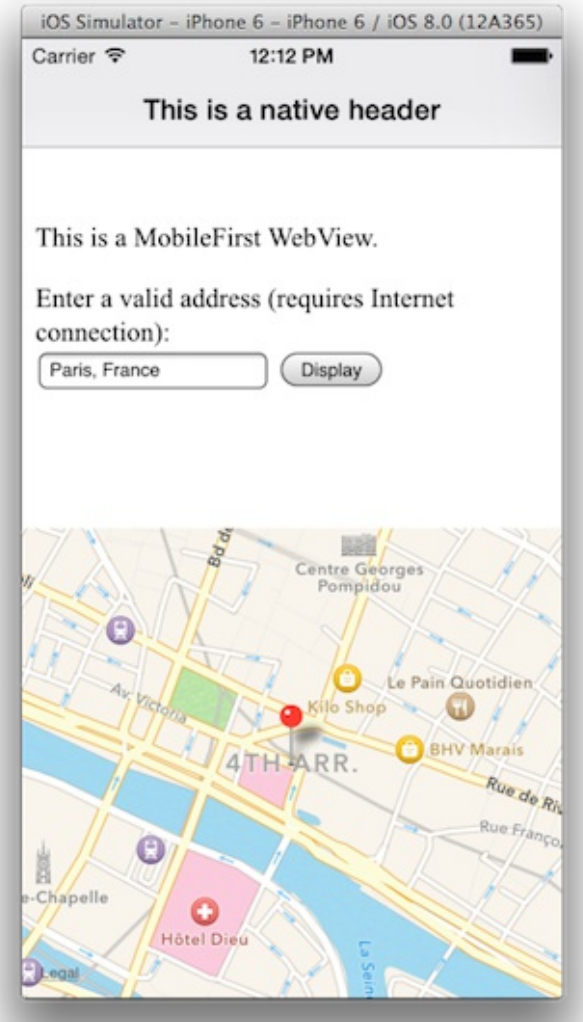
Step 3
For the action receiver to receive actions, it must first be registered. This should be done early enough in the JavaScript code so that the function will handle those actions as early as possible.

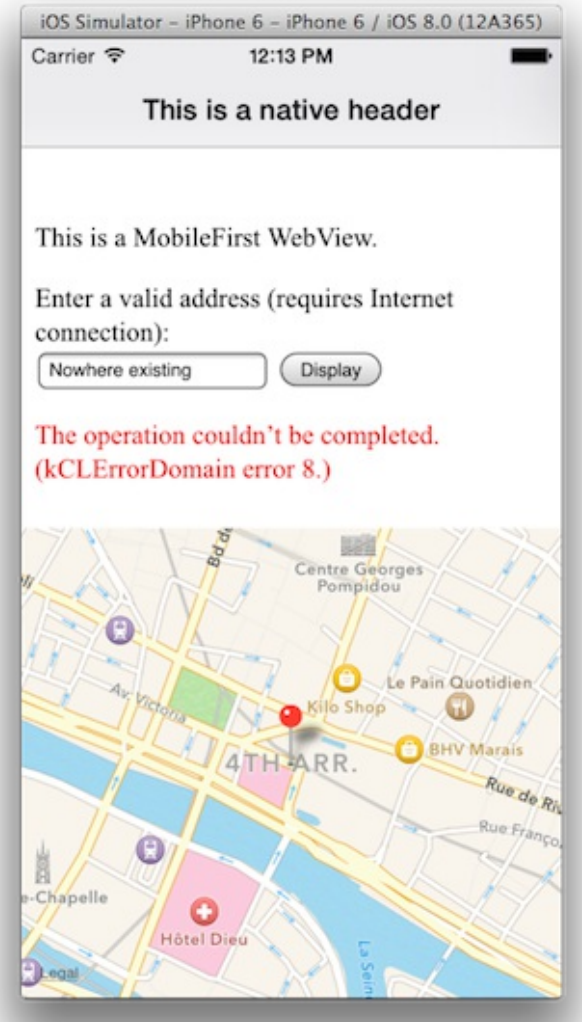
```
WL.App.addActionReceiver ("MyActionReceiverId", actionReceiver);
```

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.
Note: This sample requires the MapKit framework.

This sample divides the screen in two parts.

- The top half is a Cordova web view with a form to enter a street address
- The bottom half is a native map view that shows the entered location if it is valid

If the address is invalid, the native map forwards the error to the web view, which displays it.

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 WebView.</p>
<p>Enter a valid address (requires Internet connection):<br/><br />
    <input type="text" name="address" id="address"/><br />
<input type="button" value="Display" id="displayBtn"/><br />
 </p><br />
 <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);
 }
});
```

Storyboard
The interface was designed with a Storyboard file. It features a generic view controller with the `ViewController` custom class (described later).

The view controller contains a `MKMapView` object and a Container View.
The Container View contains a view controller, which is set to use the `HybridScreenViewController` class (described later).

HybridScreenViewController
`HybridScreenViewController` extends `CDVViewController`, the Cordova web view provided by MobileFirst (`@interface HybridScreenViewController : CDVViewController`).

The implementation of the class is almost empty, except for setting the startPage of the Cordova web view.

```
@implementation HybridScreenViewController
- (id)initWithCoder:(NSCoder*)aDecoder {
    self = [super initWithCoder:aDecoder];
    self.startPage = [[WL sharedInstance] mainHtmlFilePath]
;
    return self;
}
//...
```

ViewController
The `ViewController` class extends `UIViewController`.
This class:

- Contains a reference to the `MKMapView` object as a property
- Adheres to the `MKMapViewDelegate` protocol to receive updates about the map
- Adheres to the `WLActionReceiver` protocol to receive actions from the MobileFirst web view
- Contains a reference to a `CLGeocoder` object to enable geocoding addresses

```
@interface ViewController ()<MKMapViewDelegate, WLActionReceiver>
@property (weak, nonatomic) IBOutlet MKMapView *map;
@property CLGeocoder* geocoder;
@end
```

The title of the controller is set to be displayed as part of a `UINavigationController` object.
The geocoder is initialized.
The map delegate is set.
ViewController is registered as an action receiver for MobileFirst.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    self.title = @"This is a native header";
    self.geocoder = [[CLGeocoder alloc] init];
    [self.map setDelegate:self];
    [[WL sharedInstance] addActionReceiver:self]
;
}
```

The `onActionReceived` method is called when the user submits the form. The action name is checked and the entered address is retrieved.
The geocoder is given the address.

```
-(void) onActionReceived:(NSString *)action withData:(NSDictionary *) data {
    if ([action isEqualToString:@"displayAddress"]
        &amp;&amp; [data objectForKey:@"address"]){
        NSString* address = (NSString*) [data objectForKey:@"address"];
        [self.geocoder geocodeAddressString:address
              completionHandler:^(NSArray* placemarks, NSError* error){
                  //DO STUFF - next slide...
        }];
    }
}
```

If a location is found, the region is centered and a new `MKPlacemark` is added to the map

```
completionHandler:^(NSArray* placemarks, NSError* error){
    if([placemarks count]){
        CLPlacemark *topResult = [placemarks objectAtIndex:0];
        float spanX = 0.00725;
        float spanY = 0.00725;
        MKCoordinateRegion region;
        region.center.latitude = topResult.location.coordinate.latitude;
        region.center.longitude = topResult.location.coordinate.longitude;
        region.span = MKCoordinateSpanMake(spanX, spanY);
        [self.map setRegion:region animated:YES];
        MKPlacemark *placemark = [[MKPlacemark alloc]initWithPlacemark:topResult]
;
        [self.map addAnnotation:placemark];
    }
}
```

If the search fails or no location is found, the `sendActionToJS` method is called to transmit the error to the web view.

```
completionHandler:^(NSArray* placemarks, NSError* error){
    if([placemarks count]){
        //...
    }
    else{
        [[WL sharedInstance] sendActionToJS:@"displayError"
            withData:@{@"errorReason": [error localizedDescription
]}
         ];
    }
}
```

MyAppDelegate

- The `didFinishLaunchingWithOptions` method of the app delegate is generated by MobileFirst Studio and is left unchanged in this example
- The `wlInitDidCompleteSuccessfully` status code is modified to load the `ViewController` object from the Storyboard instead of loading the `CDVViewController` object directly
- The splash screen is hidden in native code because JavaScript has not started yet

```
-(void)wlInitDidCompleteSuccessfully
{
    UINavigationController* rootViewController = self.window.rootViewController;
    ViewController* viewController = [[UIStoryboard storyboardWithName:@"Storyboard" bundle:nil] inst
antiateViewControllerWithIdentifier:@"ViewController"];
    [rootViewController pushViewController:viewController animated:YES];
    [[WL sharedInstance] hideSplashScreen];
}
```

Shared Session

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

HTTP requests are explained in other tutorials (both for hybrid and native applications).
IBM MobileFirst Platform Foundation 6.2 and later keeps your session (cookies and HTTP headers) automatically synchronized between the JavaScript client and the native client.

Sample application

Click to download (http://public.dhe.ibm.com/software/products/en/MobileFirstPlatform/docs/v630/NativeUIInHybridProject.zip) the Studio project.