Native UI elements in Cordova Windows 8 Universal applications

Relevant to:

- Hybrid
- Download MobileFirst project (https://github.com/MobileFirst-Platform-Developer-Center/NativeUIInHybrid)

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 iOS 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
- Sample application

Taking control of the startup flow

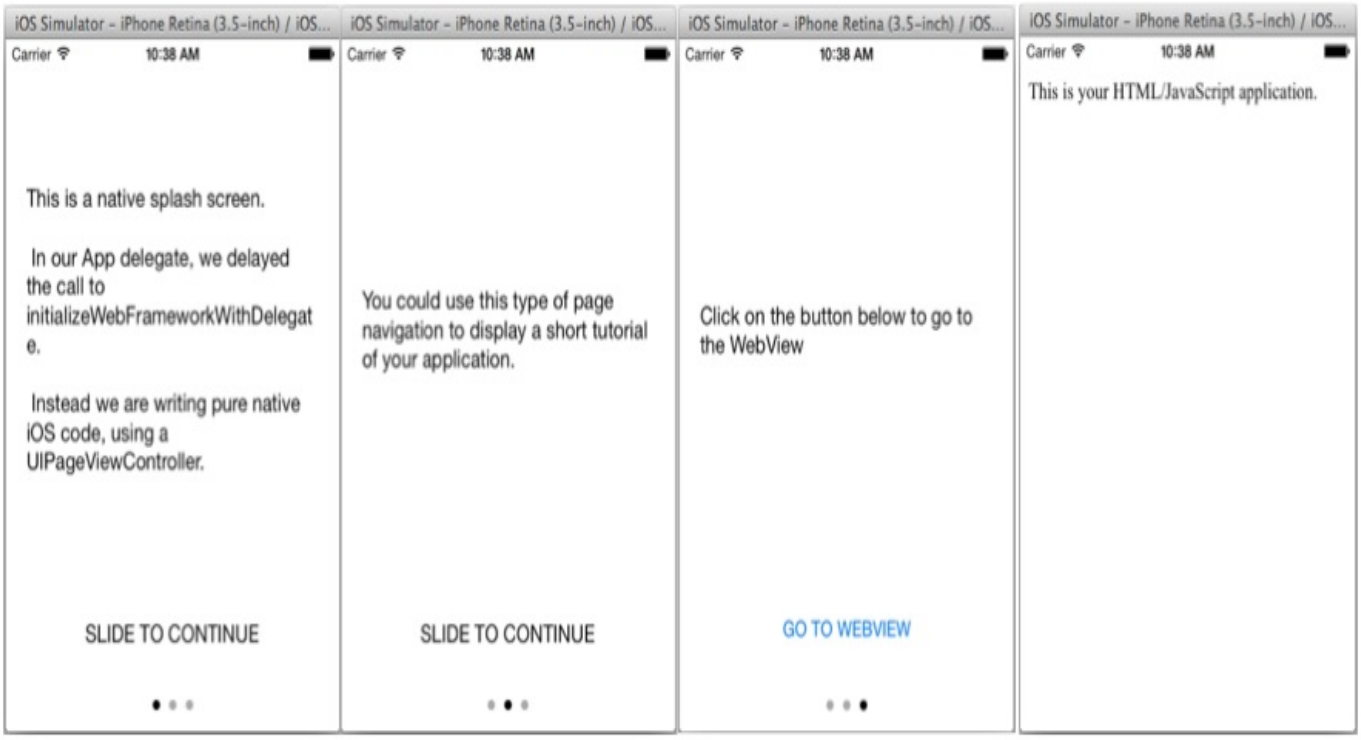
When you create a hybrid application, MobileFirst Platform generates an App Delegate (`*YourAppName*.m`) which handles the various stages of the application startup flow, as follows:

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. 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.
3. At this point, by default, the application is still displaying the splash screen and no web view is displayed yet.
   You can modify the implementation to handle more of the status codes that are returned by the framework.
   By default, a successful load calls the `wlInitDidCompleteSuccessfully` method.
4. 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.

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

Native SplashScreen sample



(http://developer.ibm.com/mobilefirstplatform/wp-content/uploads/sites/32/2014/10/07_08_splashscreen_sample.png)

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.

1. In the `MyAppDelegate.didFinishLaunchingWithOptions` method, the window is initialized and the PageViewController instance is set as the root of the window.
   [code lang="obj-c" highlight="7-8"]@implementation MyAppDelegate
   - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
   {
   //...
   self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

   PageViewController* pageViewController = [[UIStoryboard storyboardWithName:@"Storyboard" bundle: nil] instantiateViewControllerWithIdentifier:@"PageViewController"];
   [self.window setRootViewController:pageViewController];
   [self.window makeKeyAndVisible];

//..
}[/code]

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

   [code lang="obj-c" highlight="7"]@implementation MyAppDelegate
   - (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
   {
   //...
   [self.window setRootViewController:pageViewController];
   [self.window makeKeyAndVisible];
   [[WL sharedInstance] initializeWebFrameworkWithDelegate:self];
   return result;
   }[/code]

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

   [code lang="obj-c"]-(void)wlInitWebFrameworkDidCompleteWithResult:(WLWebFrameworkInitResult *)result
   {
   if ([result statusCode] == WLWebFrameworkInitResultSuccess) {
   [self wlInitDidCompleteSuccessfully];
   } else {
   [self wlInitDidFailWithResult:result];
   }
   }[/code]

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

   [code lang="obj-c"]-(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;
   }[/code]

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

   [code lang="obj-c"]-(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];
   }[/code]

Sending commands from JavaScript code to native code

In MobileFirst applications, commands are sent with parameters from the web view (via JavaScript) to a native class (written in Objective-C).
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, use the following API:
`WL.App.sendActionToNative("doSomething", {customData: 12345});`

- `doSomething` is an arbitrary action name to be used on the native side (see the next step).
- 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:
[code gutter="false"]@interface MyReceiver: NSObject <WLActionReceiver>[/code]

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:

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

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:
[code lang="obj-c"][[WL sharedInstance] addActionReceiver:[[myReceiver alloc] init]][/code]

Sending commands from native code to JavaScript code

In MobileFirst applications, commands can be sent with parameters from native Objective-C code to the JavaScript code of web views.
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 Objective-C, the following API is used:
[code lang="obj-c"]NSDictionary *data = @{@"someProperty": @"12345"};
[[WL sharedInstance] sendActionToJS:@"doSomething" withData:data];[/code]

- `doSomething` is an arbitrary action name to be used on the JavaScript side.
- 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.

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

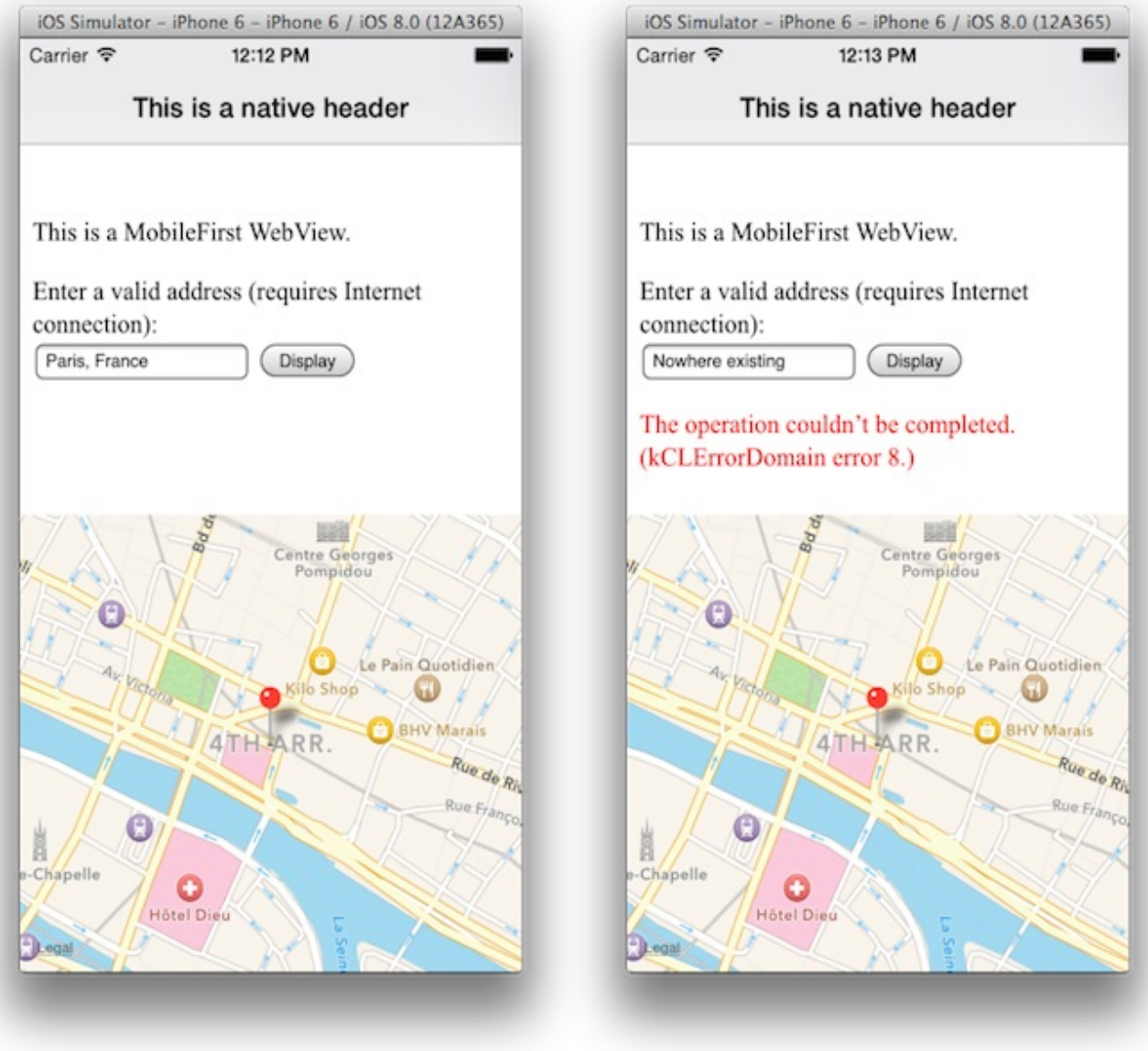
Step 3

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.

[code lang="js"]WL.App.addActionReceiver ("MyActionReceiverId", actionReceiver);[/code]

The first parameter is an arbitrary name. It can be used later to remove an action receiver.
[code lang="js"]WL.App.removeActionReceiver("MyActionReceiverId");[/code]

The SendAction sample



(https://developer.ibm.com/mobileplatform/wp-content/uploads/sites/32/2014/10/07_08_screenshot.png)

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

[code lang="html"] This is a MobileFirst WebView.
Enter a valid address (requires Internet connection):
<input type="text" name="address" id="address"/>
<input type="button" value="Display" id="displayBtn"/>

[/code]

JavaScript

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

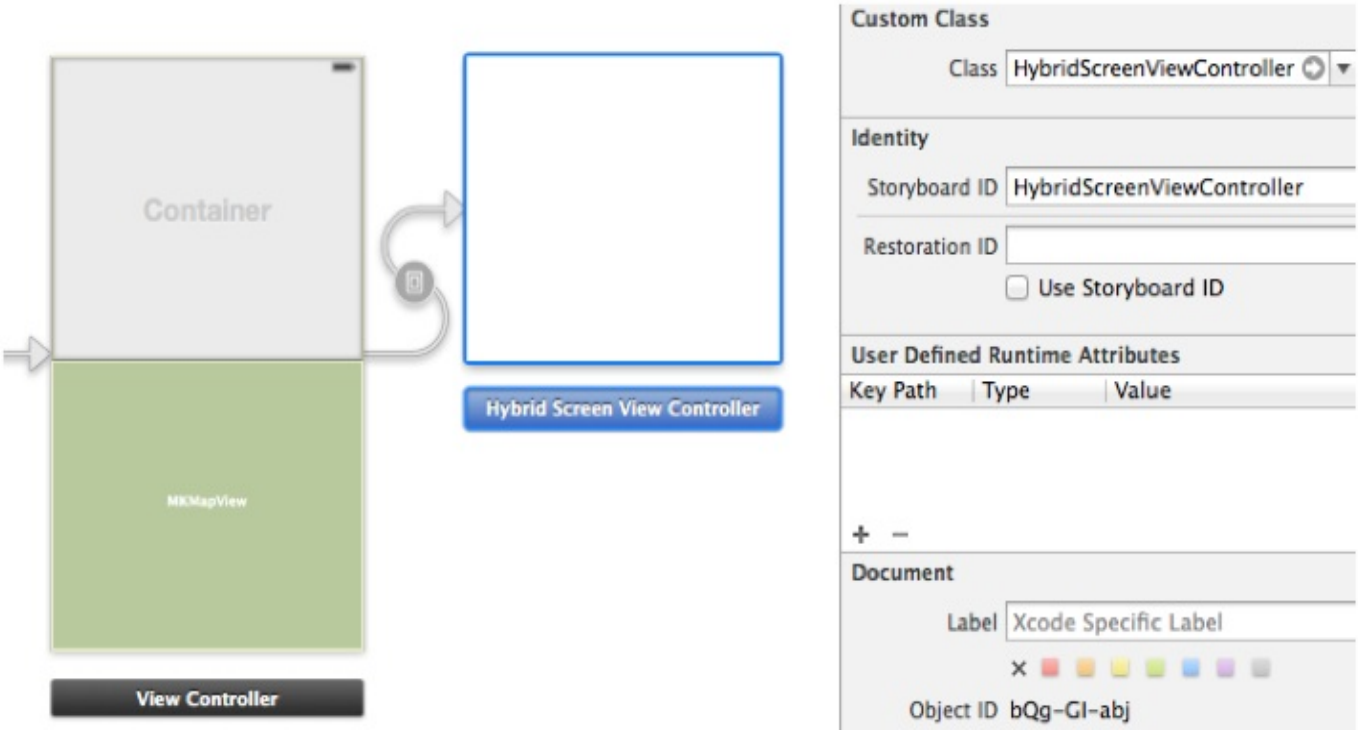
[code lang="js"]$('#displayBtn').on('click', function(){
$('#errorMsg').empty();
WL.App.sendActionToNative("displayAddress",
{ address: $('#address').val()}
);
});[/code]

The code also registers an action receiver to display potential error messages from the native code.
[code lang="js"]WL.App.addActionReceiver ("MyActionReceiverId", function actionReceiver(received){
if(received.action == 'displayError'){
$('#errorMsg').html(received.data.errorReason);
}
});[/code]

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).



(https://developer.ibm.com/mobileplatform/wp-content/uploads/sites/32/2014/10/07_08_storyboard.png)

HybridScreenViewController

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

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

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

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.

[code lang="obj-c"]@interface ViewController ()<MKMapViewDelegate, WLActionReceiver>
@property (weak, nonatomic) IBOutlet MKMapView *map;
@property CLGeocoder* geocoder;
@end[/code]

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.
The `ViewController` object is registered as an action receiver for MobileFirst.

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

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.

[code lang="obj-c"]-(void) onActionReceived:(NSString *)action withData:(NSDictionary *) data {
if ([action isEqualToString:@"displayAddress"]
&& [data objectForKey:@"address"]){
NSString* address = (NSString*) [data objectForKey:@"address"];

[self.geocoder geocodeAddressString:address
completionHandler:^(NSArray* placemarks, NSError* error){
//DO STUFF - next slide...

}];
}
}[/code]

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

[code lang="obj-c" highlight="10,13"]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];
}
}[/code]

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

[code lang="obj-c" highlight="6"]completionHandler:^(NSArray* placemarks, NSError* error){
if([placemarks count]){
//...
}
else{
[[WL sharedInstance] sendActionToJS:@"displayError"
withData:@{@"errorReason": [error localizedDescription]}
];
}
}[/code]

MyAppDelegate

- The `didFinishLaunchingWithOptions` method of the app delegate is generated by MobileFirst Platform 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.

[code lang="obj-c" highlight="4,6"]-(void)wlInitDidCompleteSuccessfully
{
UINavigationController* rootViewController = self.window.rootViewController;
ViewController* viewController = [[UIStoryboard storyboardWithName:@"Storyboard" bundle:nil]
instantiateViewControllerWithIdentifier:@"ViewController"];
[rootViewController pushViewController:viewController animated:YES];
[[WL sharedInstance] hideSplashScreen];
}[/code]

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) the MobileFirst project.