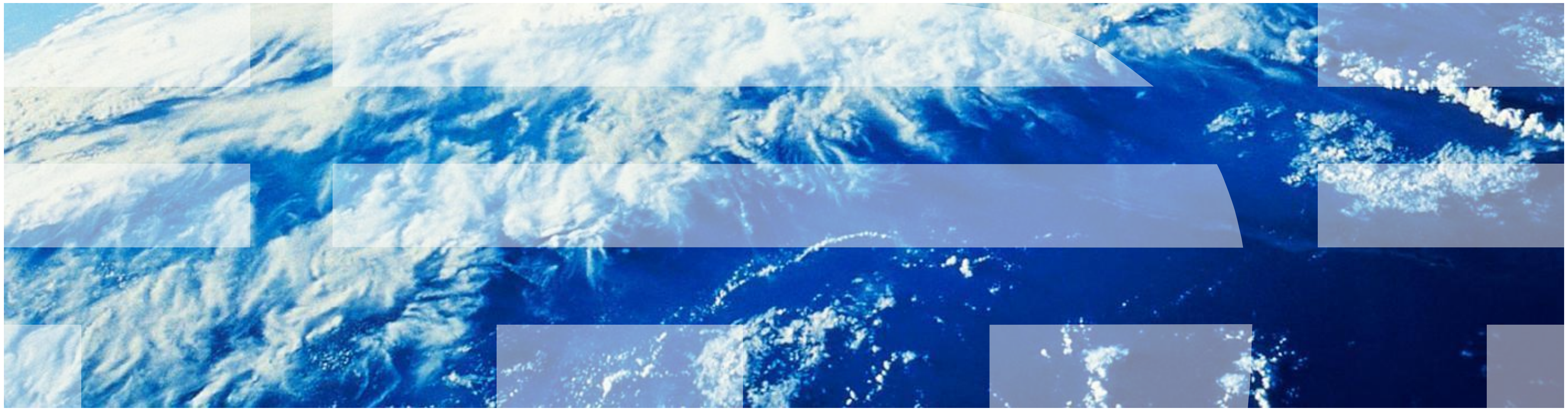
IBM Worklight Foundation V6.2.0
Getting Started

Adding Native iOS UI Elements In Hybrid Applications



2 Sep 2014

Trademarks

About IBM®

- See http://www.ibm.com/ibm/us/en/

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Overview

- While you could write your hybrid application by using solely web technologies, with IBM Worklight® Foundation® you can mix and match native code with web code as needed.
- For example, you could choose to use some native UI controls, provide an animated native introduction screen, use native elements that are provided by iOS or Android, etc.
- To do so, you need to take control of part of the startup flow of your hybrid application.
- This tutorial assumes that you have a working knowledge of native iOS development.

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Startup Flow (1 of 2)

- When you create a new hybrid application, Worklight generates an App Delegate for you, handling various stages of the application startup flow. Open this file (YourAppName.m) and study the default flow.
- Worklight starts a blank `UIViewController` instance (called `Compatibility50ViewController` in this sample), and sets it as the root view of the application.
- The `showSplashScreen` method is called to display a simple splash screen while resources are being loaded. Here, you can replace this line with any native introduction screen you wish.
- 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.

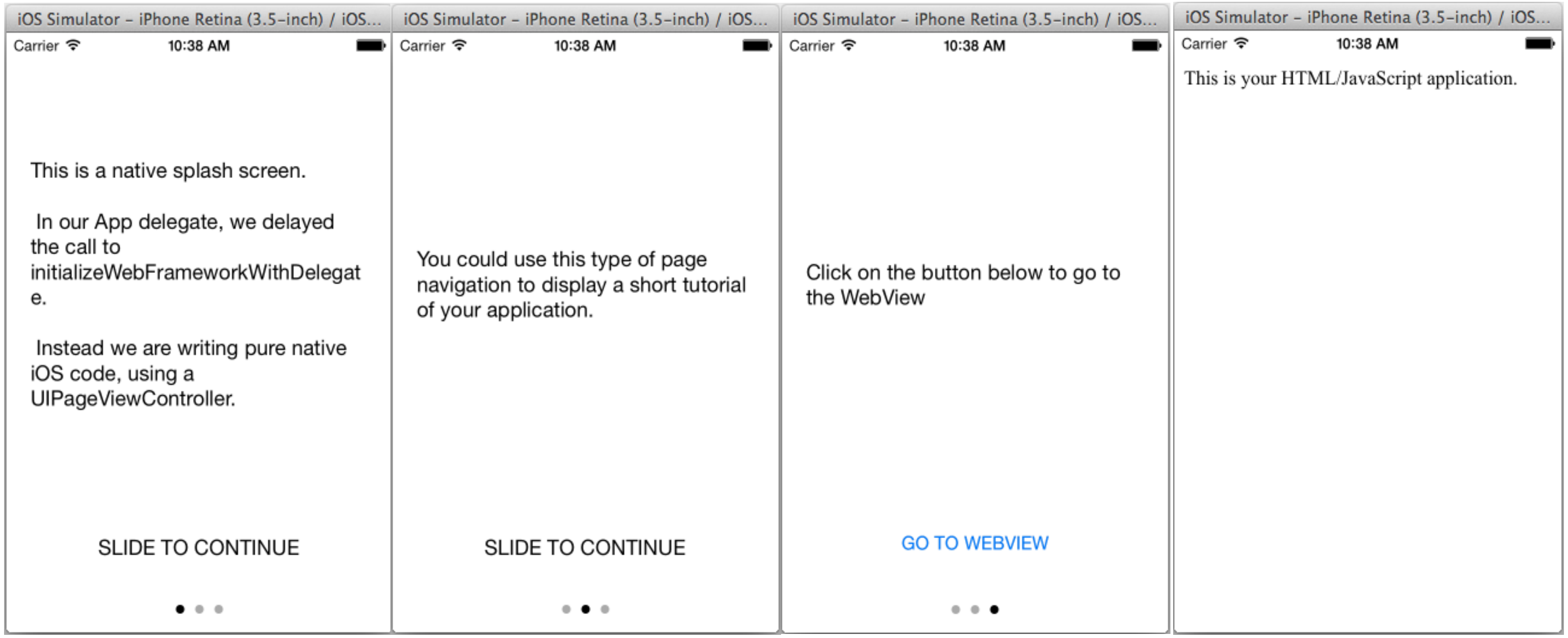
Startup Flow (2 of 2)

- At this point, by default, the application is still displaying the splash screen and no web view is being displayed yet.
- You can modify this implementation 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 your application is set.
- The Cordova controller is then added as a child to the root view and displayed on the screen.
- If you decide to write your own custom introduction screen as explained before, you might want to delay displaying the Cordova view until the user is done with your native screen.

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Native SplashScreen Sample (1 of 7)


iOS Simulator – iPhone Retina (3.5-inch) / iOS...

Carrier 10:38 AM

This is a native splash screen.

In our App delegate, we delayed the call to initializeWebFrameworkWithDelegate.

Instead we are writing pure native iOS code, using a UIPageViewController.

SLIDE TO CONTINUE

iOS Simulator – iPhone Retina (3.5-inch) / iOS...

Carrier 10:38 AM

You could use this type of page navigation to display a short tutorial of your application.

SLIDE TO CONTINUE

iOS Simulator – iPhone Retina (3.5-inch) / iOS...

Carrier 10:38 AM

Click on the button below to go to the WebView

GO TO WEBVIEW

iOS Simulator – iPhone Retina (3.5-inch) / iOS...

Carrier 10:38 AM

This is your HTML/JavaScript application.

Native SplashScreen Sample (2 of 7)

- Download the NativeUIInHybrid project, which 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.
- Search online for tutorials on `UIPageViewController`.

Native SplashScreen Sample (3 of 7)

- 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 *)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];
    //..
}
```

Native SplashScreen Sample (4 of 7)

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

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

Native SplashScreen Sample (5 of 7)

- Inside the `wlInitWebFrameworkDidCompleteWithResult`, method, you can handle different scenarios, 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];
    }
}
```

Native SplashScreen Sample (6 of 7)

- In `wlInitDidCompleteSuccessfully`, a Cordova controller is being prepared but is not displayed yet.
- Optionally, set the frame to itself so that the web view initializes in the background if you want the initialization of your JavaScript code 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;
}
```

Native SplashScreen Sample (7 of 7)

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

```
-(void)onSplashScreenDone {
    UIViewController* rootViewController = [[Compatibility50ViewController
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];
}
```

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Send Action From JavaScript to Native (1 of 2)

- In Worklight applications, you send commands with parameters from the web view (via JavaScript) to a native class (written in Objective-C).
- Use this feature to trigger native code to be run in the background, to update the native UI, to use native-only features, etc.
- From JavaScript, write

  ```
  WL.App.sendActionToNative("doSomething", { customData: 12345} );
  ```

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

Send Action From JavaScript to Native (2 of 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 you check the action name 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
      }
}
```

- For your action receiver to receive actions from the Worklight web view, you must register it. You can register it during the startup flow of the application to catch any actions early enough.

```
[[WL sharedInstance] addActionReceiver:myReceiver];
```

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Send Action From Native to JavaScript (1 of 2)

- In Worklight applications, you can send commands with parameters from native Objective-C code to web view JavaScript code.
- 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.
- From Objective-C, use `sendActionToJS`

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

where “doSomething” is an arbitrary action name to be used on the JavaScript side (see the next slide) and the second parameter is an `NSDictionary` object that contains any data.

Send Action From Native to JavaScript (2 of 2)

- Write a JavaScript function, with the name of your choice, which will verify the action name and implement any JavaScript code.

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

- Register this JavaScript function to receive actions. Do it early enough in your JavaScript code so that your function handles those actions as early as possible.

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

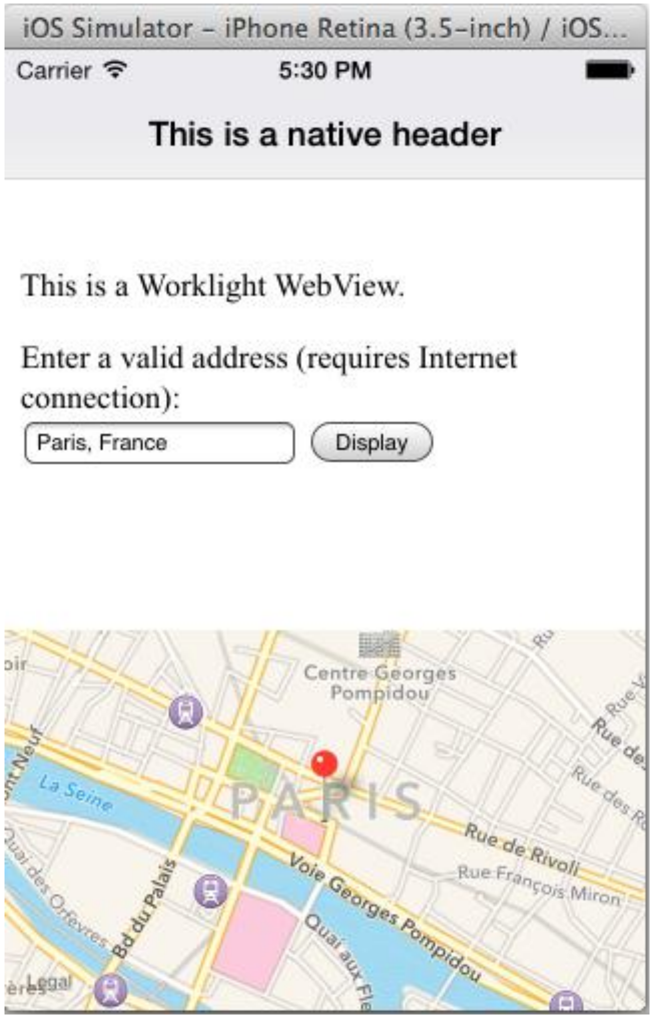
- The first parameter is an arbitrary name. You can use it later to remove an action receiver.

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

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

SendAction Sample - Preview


iOS Simulator – iPhone Retina (3.5-inch) / iOS...

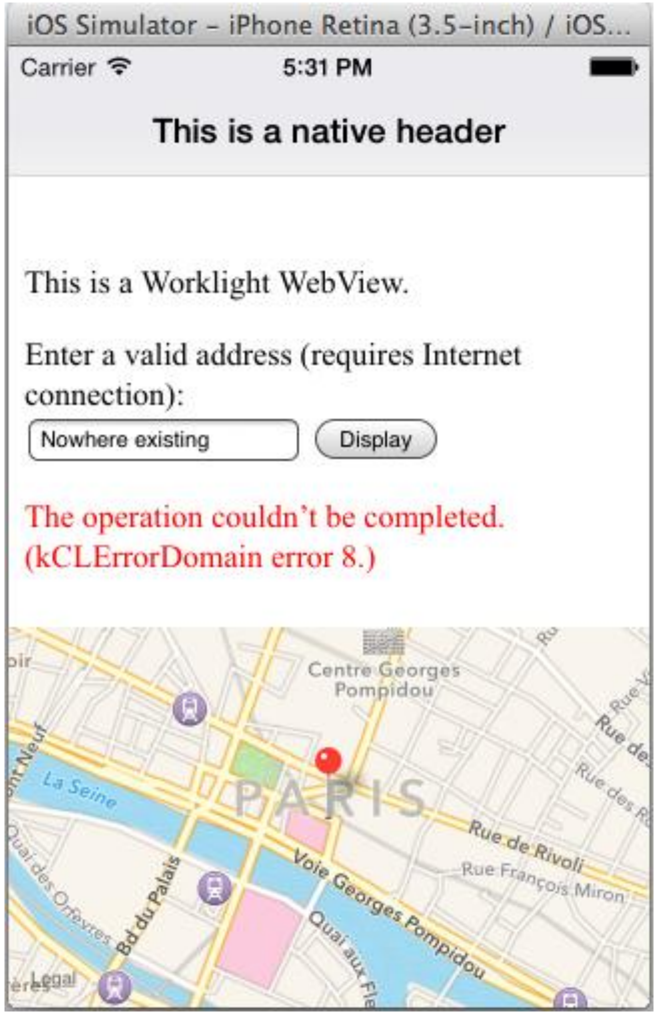
Carrier 5:30 PM

This is a native header

This is a Worklight WebView.

Enter a valid address (requires Internet connection):

Paris, France


iOS Simulator – iPhone Retina (3.5-inch) / iOS...

Carrier 5:31 PM

This is a native header

This is a Worklight WebView.

Enter a valid address (requires Internet connection):

Nowhere existing

The operation couldn't be completed. (kCLErrorDomain error 8.)

SendAction Sample - Overview

- Download the NativeUIInHybrid project, which includes a hybrid application called SendAction.
- 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.
- This sample requires the MapKit framework.

SendAction Sample - HTML

- The HTML page shows the following objects:
  - A simple input field to enter an address.
  - A button to trigger validation.
  - An empty <p> line to show potential error messages.

```
<p>This is a Worklight WebView.</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>
```

SendAction Sample - JavaScript

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

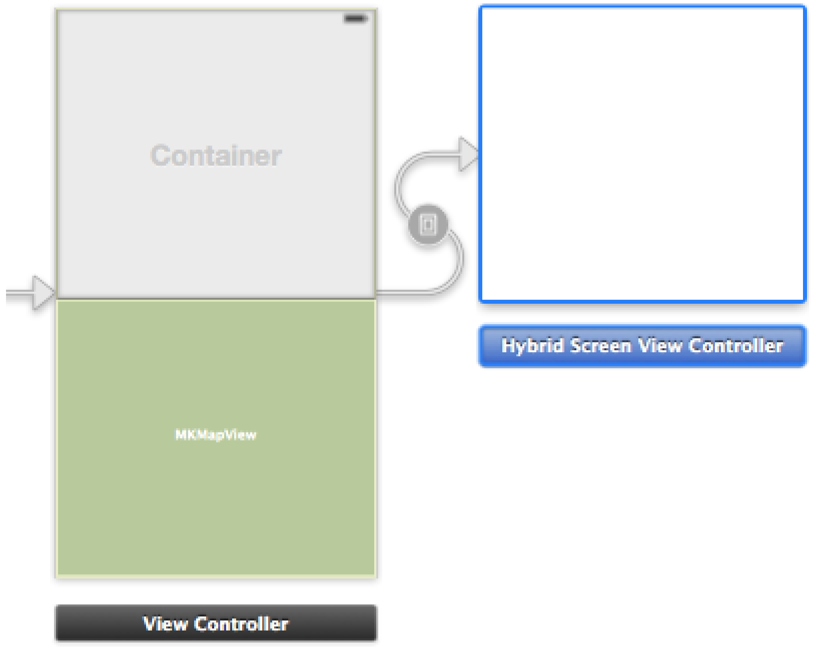
```
$('#displayBtn').on('click', function(){
$('#errorMsg').empty();
WL.App.sendActionToNative("displayAddress",
                         { address: $('#address').val()}
    );
});
```

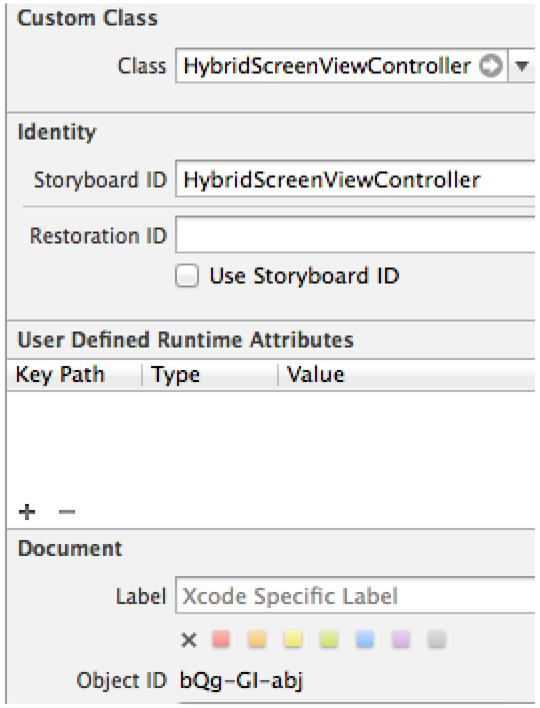
- The code also registers an action receiver to display potential error messages from the native code.

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

SendAction Sample - 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).





Custom Class

Class: HybridScreenViewController

Identity

Storyboard ID: HybridScreenViewController

Restoration ID:

Use Storyboard ID

User Defined Runtime Attributes

Key Path	Type	Value

Document

Label: Xcode Specific Label

Object ID: bQg-GI-abj

SendAction Sample - HybridScreenViewController

- `HybridScreenViewController` extends `CDVViewController`, the Cordova web view provided by Worklight.`@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;
}
//...
```

SendAction Sample – ViewController (1 of 5)

- The `ViewController` class extends `UIViewController`.
- This class contains a reference to the `MKMapView` object as a property.
- This class adheres to the `MKMapViewDelegate` protocol to receive updates about the map.
- This class adheres to the `WLActionReceiver` protocol to receive actions from the Worklight web view.
- This class 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
```

SendAction Sample - ViewController (2 of 5)

- 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 Worklight.

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

SendAction Sample - ViewController (3 of 5)

- 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"]
        && [data objectForKey:@"address"]){
        NSString* address = (NSString*) [data objectForKey:@"address"];
        [self.geocoder geocodeAddressString:address
                    completionHandler:^(NSArray* placemarks, NSError* error){
                              //DO STUFF - next slide...

                    }];
    }
}
```

SendAction Sample - ViewController (4 of 5)

- 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];
    }
}
```

SendAction Sample - ViewController (5 of 5)

- 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]}
         ];
    }
}
```

SendAction Sample - MyAppDelegate

- The default `Compatibility50ViewController` class was modified to extend the `UINavigationController` class instead of the `UIViewController` class. This modification adds a native header.
  `@interface Compatibility50ViewController : UINavigationController`
- The `didFinishLaunchingWithOptions` method of the app delegate is generated by Worklight 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] instantiateViewControllerWithIdentifier:@"ViewController"];
    [rootViewController pushViewController:viewController animated:YES];
    [[WL sharedInstance] hideSplashScreen];
}
```

Agenda

- Overview
- Startup Flow
- Native SplashScreen Sample
- Send Action From JavaScript to Native
- Send Action From Native to JavaScript
- SendAction Sample
- Shared Session

Shared Session

- When you use both JavaScript and native code in the same application, you might need to make HTTP requests to the Worklight server (connect, procedure invocation, etc.)
- HTTP requests are explained in other tutorials (both for hybrid and native).
- Worklight 6.2 and above keeps your session (cookies and HTTP headers) automatically synchronized between the JavaScript client and the native client.

Notices

Support and comments

- For the entire IBM Worklight documentation set, training material and online forums where you can post questions, see the IBM website at:
  - http://www.ibm.com/mobile-docs
- **Support**
  - Software Subscription and Support (also referred to as Software Maintenance) is included with licenses purchased through Passport Advantage and Passport Advantage Express. For additional information about the International Passport Advantage Agreement and the IBM International Passport Advantage Express Agreement, visit the Passport Advantage website at:
    - http://www.ibm.com/software/passportadvantage
  - If you have a Software Subscription and Support in effect, IBM provides you assistance for your routine, short duration installation and usage (how-to) questions, and code-related questions. For additional details, consult your IBM Software Support Handbook at:
    - http://www.ibm.com/support/handbook
- **Comments**
  - We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this document. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.
  - For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.
  - When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state.
  - Thank you for your support.
  - Submit your comments in the IBM Worklight Developer Edition support community at:
    - https://www.ibm.com/developerworks/mobile/worklight/connect.html
  - If you would like a response from IBM, please provide the following information:
    - Name
    - Address
    - Company or Organization
    - Phone No.
    - Email address

Thank You