Mobile Application Framework Custom Native Plugin

Version 5.0

MobileSmith, Inc.

June 2, 2017

Table of Contents

1. Revision History 4
2. Overview 5
3. Common Native Plugin Information 5
    - 3.1. NPI Configuration 5
    - 3.2. Class Name Configuration 5
    - 3.3. Multiple NPIs 5
4. Android Native Plugin Information6
    - 4.1. Native Plugin Requirement 6
    - 4.2. System Frameworks 6
    - 4.3. Supported Tools 7
    - 4.4. Design and Implementation Constraints 7
    - 4.5. Code Requirements 7
    - 4.6. Android Fragments 7
    - 4.7. Android Plugin Deployment Information 8
    - 4.8. EventBus Library 8
    - 4.9. Android 6.0 Permissions 8
    - 4.10. Android Caveats 9
        - 4.10.1. Multiple AAR Files 9
        - 4.10.2. ActionBar Title 9
        - 4.10.3. Kotlin Language Support 9
        - 4.10.4. Nested Views 9
5. iOS Native Plugin Information 10
    - 5.1. iOS Dynamic Framework 10
        - 5.1.1. Dynamic Framework Deployment 10
    - 5.2. iOS Static Library Native Plugin 11

5.2.1. iOS NPI Zip File *11*

5.2.2. NPI View Controller *11*

5.2.3. NPI Bundle *12*

5.3. Fly-out Menu Integration *12*

5.4. UIAppearance *13*

5.5. Code Requirements *13*

5.6. System Frameworks *14*

5.7. iOS Caveats *16*

5.7.1. iOS File extended resources *16*

1. Revision History

Name	Date	Reason for Changes	Version
Initial	04/16/14	Initial Document	1.0
Update	01/02/15	Updated for Android Studio	2.0
Update	04/05/15	Updated for Gradle Build Script	3.0
Update	06/24/15	Update for Support for Activities	3.1
Update	10/08/15	Updated for the MobileSmith 4.0 Release and merged the iOS Documentation into this document	4.0
Android 6.0 Update	03/29/16	Updated for Android 6.0 documentation.	4.1
iOS Dynamic Framework Update	06/02/17	Updated for iOS Swift Dynamic Framework support	5.0

2. Overview

2.1. Purpose

This document describes a way to inject a custom Native Plugin (NPI) into the Android and iOS Mobile Application Framework (MAF) Platforms. The NPI is a standalone/self contained part of an application and will allow you a way to incorporate your own functionality into your Apps through the MAF Platforms. NPI's should be written in the native code for the platform:

OS Platform	Language
Android	**Java**
iOS	• **Swift** • **Objective C**

NPI's give the developer the power to create custom sections of the Application, utilizing tools such as the Layout Editor/Interface Builder in order to define UI layouts and incorporating different backend API's that are not currently supported by the platform.

3. Common Native Plugin Information

3.1. NPI Configuration

In the app design UI, the designer can enter in arbitrary key/value pairs to associate with the NPI. This will be converted into a JSON file with a single object and key/value pairs for each field, and the JSON data will be provided to the NPI at runtime. All of the values are passed to the NPI's as string JSON data, data conversion will need to be done by the NPI (string to boolean, string to integer, etc.).

3.2. Class Name Configuration

The only required key in the configuration JSON is the classname, which is the main entry point into the Native Plugin (Fragment for Android and UIViewController/UINavigationViewController for iOS):

Platform	JSON Key	Classname Example
Android	**FragmentClassName**	**com.example.ui.NpiFragment**
iOS - Objective C	**ViewControllerClassName**	**ViewControllerClassName**
iOS - Swift	**ViewControllerClassName**	**Module.ViewControllerClassName**

3.3. Multiple NPIs

If a single app has multiple NPIs, care should be taken since both NPIs will be linked into the final app. So any shared code should only be defined in one of the NPIs to prevent linker errors.

4. Android Native Plugin Information

4.1. Native Plugin Requirement

For the NPI, the following information is needed:

- Android Library Project packaged into an Android Archive File (AAR).

4.2. System Frameworks

The Android MAF Platform builds towards the following versions of the Android Platform:

Android Build Information	Version
Compile SDK Version	25
Build Tools Version	25.0.2
Target API Version	23
Minimum SDK Version	16
JDK Version	1.5

Android Platform App builds include the following libraries:

Library Information	Version	Gradle Compile Information
Android Support Compat Library	7:25.1	com.android.support:appcompat-v7:25.1.+
MultiDex	1.0.0	com.android.support:multidex:1.0.0
RX Java	0.20.7	• com.netflix.rxjava:rxjava-core:0.20.7 • com.netflix.rxjava:rxjava-android:0.20.7
RX Permissions	0.6.1	com.tbruyelle.rxpermissions:rxpermissions:0.6.1
Google Play Services	10.2.0	• com.google.android.gms:play-services-location:10.2.0 • com.google.android.gms:play-services-maps:10.2.0 • com.google.android.gms:play-services-analytics:10.2.0
DiskLRUCache Library	2.0.2	com.jakewharton:disklrucache:2.0.2
HttpMine	4.2	org.apache.httpcomponents:httpmime:4.2
Flurry Analytics	3.4.0	Provided by JAR File
Google Cloud Messaging		
Picasso Library	2.5.2	com.squareup.picasso:picasso:2.5.2

Library Information	Version	Gradle Compile Information
Spring Android	1.0.1	• org.springframework.android:spring-android-core: 1.0.1.RELEASE • org.springframework.android:spring-android-rest-template:1.0.1.RELEASE
Scribe (OAuth Library)	1.3.7	org.scribe:scribe:1.3.7
GreenRobot EventBus	2.4.0	de.greenrobot:eventbus:2.4.0

The NPI must provide any 3rd party libraries in the deployed AAR file.

4.3. Supported Tools

We support the following tools for NPI development:

Tool	Information
Android Studio 2.3	Integrated Development Environment
Roboelectric Unit Test	Unit Testing Framework
Espresso 2.0 UI Test Tool	Automated UI Testing Tool

4.4. Design and Implementation Constraints

Android MAF Platform will provide a Fragment container for 3rd party NPI User Interface elements. These Fragment containers will allow the Platform the ability to embed the NPI's inside the current UI without changing the underlying code.

4.5. Code Requirements

The Android NPI should abide by the following:

- Anything going against the Play Store guidelines is prohibited in third party code.
- Exceptions are not handled outside of the custom Fragment: if the custom Fragment does not handle the exception the app will crash.

4.6. Android Fragments

Android Fragments (from the Android Support Library v4) should be used as a UI Container for NPI's, it represents a portion of the UI in an Android Activity. This will allow the Android MAF Framework to control most of the screen that has the NPI. The Fragment approach will allow the Custom Native Plugin to have its own lifecycle, receive its own input events, and the ability to swap out its own Fragments without the knowledge of the MAF Platform.

The Android MAF Platform will pass in an Android Bundle to the Fragment that includes the following:

- JSON Configuration String (Bundle key = “json_config") - this will contain the JSON provided in the JSON Configuration File.
- UI Fragment Container Id (Bundle key = “container_view_id”) - this should be used to replace the current fragment in the Platform:

```
final FragmentTransaction fragmentTransaction =  activity.getSupportFragmentManager().beginTransaction();
fragmentTransaction.replace(container_view_id, fragment);
if (addToBackStack) {
        fragmentTransaction.addToBackStack(null);
}
fragmentTransaction.commit();
```

4.7. Android Plugin Deployment Information

The Custom Native Plugin should be package in an AAR file with the structure of an Android library as described in the Android Build System Documentation (http://tools.android.com/tech-docs/new-build-system/aar-format)

4.8. EventBus Library

The Android MAF platform utilizes a publish/subscribe event bus for communications between the platform components. This event bus can be utilized by the NPI for NPI to NPI communication and in the future NPI to MAF/MAF to NPI communication. For more information about the EventBus, see:

https://github.com/greenrobot/EventBus

4.9. Android 6.0 Permissions

With the update to Android 6.0, developers are now required to ask at runtime for dangerous permissions. The MobileSmith Android Platform utilizes the Rx Permissions library to handle the runtime permissions.

http://developer.android.com/training/permissions/requesting.html
https://github.com/tbruyelle/RxPermissions

For Rx Permissions, we do not utilize the lambda functionality, instead we use regular callbacks. The NPI Sample Code includes Rx Permissions functionality.

4.10. Android Caveats

The following are caveats with Native Plugins on Android:

4.10.1. Multiple AAR Files

Currently only one AAR file per Native Plugin AppBlock is supported, if multiple AAR files are needed, the user should create one Native Plugin AppBlock for each AAR file.

4.10.2. ActionBar Title

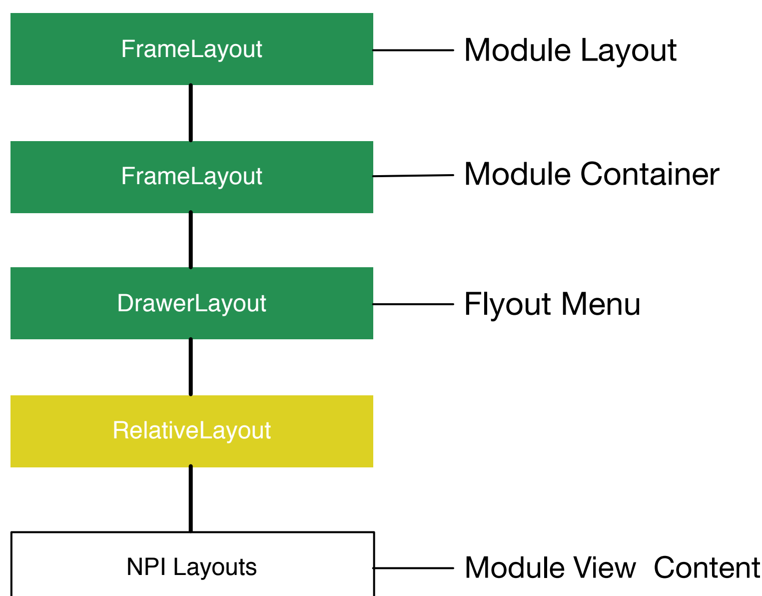
Changing the ActionBar Title is currently not supported in the NPI framework. This functionality may be provided at a later date.

4.10.3. Kotlin Language Support

Kotlin is not currently supported in the platform but there are plans to support the language in the future.

4.10.4. Nested Views

Nested views can cause some latency issues with the NPI's inside the MobileSmith platform, since the NPI resides several layers deep. See the following App View Hierarchy:



5. iOS Native Plugin Information

The iOS MobileSmith platform primarily works by transitioning to various UIViewController subclasses. One might represent a list of data configured in the MobileSmith website, another might have an e-mail feedback form, etc. This gives applications built in the MobileSmith platform a lot of built-in functionality they could utilize, but sometimes an application has needs for some very specific functionality.

NPIs allow app designers to include a custom UIViewController instance in their application that can be selected and navigated to similar to the other internal UIViewController subclasses. Currently the “scope” of an NPI is limited to what can be accomplished by a UIViewController implementation. An NPI cannot modify the Info.plist file of the app, change the application icon, or modify the Xcode linker flags, for example.

There are two primary components that make up an NPI: a ZIP file that contains all third-party code and resources needed by the device, and a configuration JSON file that is setup in the MobileSmith Platform with configuration information that could be utilized by the NPI code.

For the iOS Platform, there are two ways a developer can build NPI’s:

- Dynamic Frameworks (Swift and/or Objective C)
- Static Libraries (Objective C only)

5.1. iOS Dynamic Framework

Swift 3.+ NPI support is provided through the use of dynamic frameworks, more information on iOS frameworks can be found at:

https://developer.apple.com

The same View Controller methodology that static library NPIs utilize applies to dynamic frameworks, there needs to be an initial UIViewController that the platform will initialize. That view controller can implement the following optional method if it needs some options from the JSON Configuration data:

Method	
public func configure(_ json: NSDictionary!)	Optional method for passing in the JSON configuration data.

5.1.1. Dynamic Framework Deployment

The Dynamic Framework should be zipped up for uploading to the platform, if the framework relies on multiple frameworks that are not already included in the platform then those frameworks can be added to the same zip file as the NPI.

5.2. iOS Static Library Native Plugin

5.2.1. iOS NPI Zip File

The zip file has only one required component: a library (.a) file containing the code for the custom UIViewController (or UINavigationController) subclass. In addition it could contain any number of resource bundles or additional library files that are required. If there are multiple items that need to be included in the zip, be sure to select the items in the Finder, right-click and select “Compress # items…” instead of compressing their common parent directory.

5.2.2. NPI View Controller

The main entry point for an NPI is the main view controller. Whenever the NPI is selected in the MobileSmith application, this is the view controller that will be presented. There are two primary types of NPIs that are supported, depending on their superclass:

- UIViewController-based NPI. This will typically be added to one of the UINavigationControllers in the MobileSmith application. The only exception would be if the NPI is included in the tab bar of the application, in which case it will be displayed without any UINavigationController. This is the recommended type of NPI. A UIViewController-based NPI should not override the standard “Back” button in its enclosing UINavigationController

- UINavigationController-based NPI: If the NPI view controller is a UINavigationController subclass, then it will typically be presented modally. The only exception would be if the NPI is included in the tab bar of the application. A UINavigationController-based NPI should not be used in the fly-out menu, since the fly-out hide/show button is part of a MobileSmith navigation controller. A UINavigationController-based NPI is required to implement a “Close” button if needed to dismiss the NPI. A UINavigationController-based NPI should only be used if the NPI needs full control over the navigation controller, or if the NPI is going to be selected from the tab bar and a navigation controller is needed.

The view controller class should implement one of the following initialization methods:

Method	Information
+ (id)npiWithJSON:(NSDictionary *)json;	This class method would be appropriate if the NPI is loaded from a UIStoryboard or if some additional logic needed to be done before the NPI knew which UIViewController subclass it wanted to return. It is given the configuration JSON data as an NSDictionary.
- (instancetype)initWithJSON:(NSDictionary *)json;	This instance method is the most commonly used, and assumes that the NPI is either initialized from a XIB file or entirely in code.

The view controller class could also optionally implement the following method:

Method	Information
+(void)setupWithJSON:(NSDictionary *)json	Called on app launch, before the NPI view controller is initialized. This could be useful for downloading additional data that the NPI needs before being displayed.

5.2.3. NPI Bundle

This is optional, and multiple bundles could be provided. All resources (including xib and storyboard files) should be included in this bundle. When creating a new bundle target in Xcode, you need to create a Mac OSX bundle target and then manually change the platform from OSX to iOS. If you are having trouble loading PNG files from the bundle, try removing the COMBINE_HIDPI_IMAGES setting for the bundle target.

Although the bundle target will initially have source code associated with it, the bundles provided for use as a MobileSmith NPI should not have ANY code. You can verify this by examining the bundle and making sure that there is not an executable file in the bundle with the same name as the bundle (e.g., Foo.bundle/Foo)

5.3. Fly-out Menu Integration

If the app designer wants to link an NPI in a fly-out menu, then there are a couple extra considerations:

The NPI should be a UIViewController-based NPI. It will automatically be added in a UINavigationController that has the fly-out toggle button (i.e., a button with a “hamburger” icon)

There are some cases where the NPI coder does not want to be embedded into a MobileSmith UINavigationController. If this is the case, then there is some additional work on the part of the NPI to interact with the application fly-out menu:

- The NPI could use the “btn_flyout” image in the main application bundle, which represents the fly-out “hamburger” icon

- The action to toggle the fly-out menu is @selector(toggleFlyoutMenu:). The NPI could either provide a nil target on a control that uses that selector, or use –[UIResponder targetWithAction:withSender:] to determine what object it should send the action to.

- The JSON configuration provided to the NPI should include a “iOSNavPreference” key with a “never” value to indicate to the MobileSmith code that it should not be placed inside of a navigation controller. Currently this preference is only applicable when the NPI is accessed from the fly-out menu, and behavior is undefined if this setting is used in an NPI that is not accessed from the fly-out menu.

- Whenever the user navigates to the NPI to the fly-out menu, the MobileSmith code will attempt to instantiate a new instance of the NPI. If this is not desired, the NPI could instantiate itself as a singleton.

5.4. UIAppearance

MobileSmith utilizes Apple's UIAppearance protocol for much of its look-and-feel. This automatically sets default colors for some UI elements (e.g., navigation bar background color and text). If no colors are set for these properties in the NPI, then they will automatically get the colors specified for the application on the MobileSmith website. These properties include:

- UINavigationBar text and background colors
- UISearchBar text and background color

5.5. Code Requirements

The iOS NPI should abide by the following:

- NPI library files should be compiled for arm64, armv7, and armv7s architectures. A universal library file (device and simulator) could also be created, but might need to be compiled manually. After generating a device.a and simulator.a library file, a universal version could be created with "lipo –create device.a simulator.a –output universal.a" command in the Terminal.
- Anything going against the AppStore guidelines is prohibited in third party code.
- Method swizzling (changing the implementation of an outside class at runtime) is prohibited in third party code.
- Exceptions are not handled outside of the custom view controller: if it does not handle an exception the app will not handle it and will result in a crash.
- Currently on iPad only landscape mode and only modal launching of the custom view controller are supported.
- Currently on iPhone only portrait mode is supported.
- iPad NPIs cannot currently use UISplitViewControllers
- NPIs should target iOS 9 and work on iOS 9 and iOS 10
- If building a static library, many common filenames are already in use by the core MobileSmith code. NPI filenames and classes should have a common prefix to distinguish itself, e.g. XYZMainViewController.m vs. MainViewController.m
- If an NPI modally presents a view controller, then it is responsible for dismissing the view controller
- If the bulk of the NPI is contained in a modally presented view, then the NPI view controller should still contain meaningful content since it is what will be visible once the modal view controller is dismissed.

- No files or folders contained in the NPI zip should have resource forks attached to them. The xattr command-line tool could be used to verify. (See “File extended resources” section in the iOS Caveats)

5.6. System Frameworks

Below is a listing of system frameworks that we currently link against. If a native plugin requires a system framework that is not listed here, then we will need to include that framework in our project before we can support that native plugin:

Framework
AVFoundation.framework
AdSupport.framework
AssetsLibrary.framework
CFNetwork.framework
CoreData.framework
CoreGraphics.framework
CoreLocation.framework
CoreMedia.framework
CoreVideo.framework
EventKit.framework
EventKitUI.framework
Foundation.framework
ImageIO.framework
MapKit.framework
MediaPlayer.framework
MessageUI.framework
MobileCoreServices.framework
PBXFrameworksBuildPhase
QuartzCore.framework
SystemConfiguration.framework
UIKit.framework
ibiconv.dylib

Framework
libxml2.dylib
libz.dylib

In addition, we also use the following static third-party libraries. If an NPI needs to use one of these, MobileSmith should be contacted to make sure we are using the same version of that library. If they attempt to include it as part of their uploaded resources zip, it could result in a build failure:

Library
libFlurryAnalytics.a
libGoogleAnalytics.a
libGoogleConversionTracking.a
libzbar.a

5.7. iOS Caveats

5.7.1. iOS File extended resources

One potential issue in an NPI that could cause an app store submission to fail is the existence of extended file resources. These resources cannot always be represented in a zip file, and will sometimes be extracted into a __MACOS directory with individual files for the extended resources. This causes an issue with code signing, since those are seen as separate files that were added after code signing, causing app verifications to fail. To prevent this issue, the contents of an NPI zip file should not contain any extended resources.

To view extended resources, the xattr tool can be used. Before zipping up the NPI, if you navigate in the Terminal to the directory with the static library and/or bundles, execute the following command to recursively list all extended resources for all folders and files in that directory:

find . –exec xattr –l “{}” \;

One of the most common attributes that are accidentally added to files is the com.apple.quarantine attribute. This is typically added to files that are downloaded from an e-mail client or browser and indicate to the Mac operating system that the zip file may contain executable code that has not been approved by the downloader to execute. If you ever download an application and get a pop-up from Apple asking if you are sure you want to launch an application that was downloaded from the internet, that is the work of the com.apple.quarantine attribute. To recursively remove that attribute from all files/folders in the directory, execute the following command in the Terminal:

find . –exec xattr –d com.apple.quarantine “{}” \;

Note that this could show a lot of warning messages when it attempts to remove that attribute from files that do not actually have the attribute, but that is to be expected. After running that command for each unique extended attribute that was found (if any other than com.apple.quarantine are found), run the

find . –exec xattr –l “{}” \;

Command again to ensure that all extended attributes are removed, and then zip up the NPI for use in the MobileSmith application.