UFW iOS Integration Document

Author	Team Huma, DL_Huma@philips.com
Approved by	

Contents

1. INTRODUCTION

This document provides an overview of integration procedure for UAPPFramework library in iOS mobile applications.

2. INITIALIZATION

Only the POD file setup is required for UAPPFramework.

2.1 POD Set up

In the destination application, we need to add the “UAPPFramework” in the POD file of the application. Please refer to the line of code that we need to add in the destination POD file for UAPPFramework integration.

pod 'UAPPFramework' , '2.0.0-rc.101’

3. iOS

- In case of IOS, each micro app is expected to return view controller which needs to be launched by the app.
- Micro app framework defines standard launch API as part of common protocol which needs to be implemented by each micro app as described in below sections.
- Also as part of launch API, we need to have completion handler with error block returning NSError type.
- A micro app should throw an exception if app does not have navigation controller and mention about this expectation from app in integration guidelines.

- (**instancetype** _**Nonnull)**initWithDependencies:(UAPPDependencies * _Nonnull) dependencies andSettings : (UAPPSettings * _Nullable) settings;

- (**UIViewController *** _**Nullable**) instantiateViewController:(UAPPLaunchInput * _Nonnull) launchInput withErrorHandler:(void (^ _Nullable) (NSError * _Nullable error))completionHandler;

- A micro app has to subclass UAPPDependencies and give list of the dependencies needed for their intialisation.
- Now,by default AppInfra has been added as Dependency for all microApps
- A micro app has to subclass UAPPLaunchInput and give list of Inputs which has to be given to initialize the micro app.
- A micro app has to subclass UAPPSettings and give list of the one-time settings needed for their intialisation.

4. Flow Manager

4.1 TFS Link:

http://tfsemea1.ta.philips.com:8080/tfs/TPC_Region24/CDP2/_git/ufw-ios-uappframework

4.2 Flow Manager:

Flow-Manager is a class which extends BaseFlowManager, responsibility includes to pass context and JSON path to BaseFlowManager and mapping respective State with AppState Constants, mapping respective Conditions with Condition Constants and return respective State based on current State and eventId passed.

Find the below Flow Manager File for reference

BaseFlowManager.swift

For detailed explanation please find below the link:

https://atlas.natlab.research.philips.com/confluence/display/BA/Flow+Manager

Some of the pre-requisites for using Flowmanager are as follows:

· FlowManager requires a hard-coded state that needs to be defined.

· This is required so that there is some state that is loaded while the AppFlow.json file is parsed.

· This first state needs to implement the **FlowManagerListener**.

· This listener get the call back once the **AppFlow.json** is parsed.

· After this the first state to be loaded can be fetched by passing the first event and the current state to getNextState method in the flowmanager instance.

4.2.1 API to override in BaseFlowManager are:

1. public func initialize(withAppFlowJsonPath jsonPath : String, successBlock : @escaping () -> (), failureBlock : @escaping (FlowManagerErrors) -> ()) {

This method gets a callback once the AppFlow.json file is parsed. If Appflow.json is parsed successfully successBlock closure will get callback in case of failure failureBlock closure will get callback. Parameter to this method is Json Path and Success and Failure Closure.

1.1 When this method is called from Main Thread,Flow Manager spawns a background Thread and does Json Parsing and give success/failure callback in Main Thread.

1.2 When this method is called from Background Thread,Flow Manager does the parsing in that Thread and gives success/failure callback on that Thread.

2 public func initialize(withAppFlowModel flowModel : AppFlowModel?) throws

It initializes AppFlow which takes appflow model as input, and throws exception when appflow model passed is nil

3. public func getNextState(_ forEventId : String?) throws -> BaseState? {

Returns next state for the specified EventID.

4. public func getNextState(_ currentState: BaseState?,forEventId : String?) throws -> BaseState? {

Returns next state for the specified state.

If `currentState` is `nil`, it returns first state in the flow.

- parameter currentState: The current state.

- returns: The next state to load.

5. public func getCondition(_ conditionId : String) -> BaseCondition? {

This method is called to get the condition, which is mapped with the corresponding conditionId

- parameter conditionId: condition Id defines, at which condition the app is in

- returns: state of type BaseCondition

6. public func getState(_ stateId : String) -> BaseState? {

This method is called to get the state, which is mapped with the corresponding stateId

- parameter stateId: State Id, defines the which state is the app in

- returns: state of type BaseState

7. **public func** getCurrentState() -> BaseState? {

To get the current state of the app

- returns: the current State which the App is in

8. public func getBackState(_ forState : BaseState) throws -> BaseState? {

This class also contains methods to get the Previous State.

9. open func populateStateMap(_ stateMap : inout StateMapType) {

Method to be implemented by the child classes for mapping the app states with corresponding state classes, inherited from baseState class

- parameter stateMap: Variable that hold the base state

10.**open func populateConditionMap(_ conditionMap : inout ConditionMapType)** {

Method to be implemented by the child classes for mapping the app conditions with corresponding condition classes, nherited from baseCondition class

- parameter conditionMap: Variable that hold the base condition

4.4 State

Any flow element can be represented as state and a state class is supposed to extend BaseState Class and implement the abstract methods for initialization, navigation and handling callbacks.

4.4.1 BaseState

This class is the base class for all the state objects. Any new state that is created should extend from this class and implement the abstract classes.

4.4.2 Public methods are:

1. **4.** open func getViewController() -> UIViewController? {

Method getViewController is to be implemented by all state classes which gives the next ViewController

- returns: Viewcontroller that is to be navigated

2. open func updateDataModel() {}

Method updateDataModel is implemented to pass data to any Common components like InAppPurchase, UserRegistration, ConsumerCare, ProductRegistration

4.5 Condition

```
As the name says Condition is a class which is responsible to define conditions to move to next state. The API isSatisfied() is called while navigating to each state, based on the conditions defined FlowManager returns State.
```

4.5.1 BaseCondition

This is the base class for all the conditions that needs to be used by the proposition. This is an abstract class. Any condition that is defined in the **AppFlow.json** will have create a corresponding class and extend from this base class. This ensures the FlowManager can access and check for this condition in getNextState internally.

```
1. public var conditionId: String!
```

This variable is used to get the condition ID for this particular Condition Object.

```
2. open func isSatisfied() -> Bool
```

This method is used to write the condition by the extending condition class.

3. public override init()

This method is used to Initialise the Condition and pass ConditionId for the Condition.

4.6 Rules to be followed while using UAPPFramework:

· Events are case insensitive.

· State id is case sensitive.

· Condition id is case sensitive.

· If anyone wants to handle custom back, they should give **eventId** as “**back**” in Flow Json.

· If anyone wants to handle custom back,they should have their custom back button and should call **getBackState** API from the Back button action.

· In case of Custom Back Handling, apps must modify their own navigation stack.

4.7 Backward Navigation Using UAPPFramework:

When Propositions want to Implement Custom Back for their app,they have to follow the below Steps :

· Add **“back”** event in AppFlow.json in the below format:

“states”: [

{

“state”: “home”,
{

“eventId”: “back”,

“nextStates”: [

Unknown macro: { “condition”}

]

}
]

· Add Custom Back Button in their app and call **getBackState** API from the back button action

· FlowManager maintains a stack of States for the app. Whenever App navigates to a State, it is pushed to the Stack.

· When a back event happens,and there is no **“back”** event in the Json for that state,by default the top State gets popped from the Stack.

· If there is a custom **“back”** event in the Json for that state,FlowManager will search for that State in the Stack,and if it finds that State,it will pop all the States upto that State.

· If there is a custom **“back”** event in the Json for that state,FlowManager will search for that State in the Stack,and if it doesnot find the State in the Stack,FlowManager will pop the top state from Stack and replace it with the new State.

4.8 Error Handling in Flow Manager :-

- FlowManagerErrors is a Structure which holds all the errors of FlowManager
- public func message() -> String
  - This API is used to get the error message of Flow Manager
- public func code() -> Int
  - This API is used to get the error code of Flow Manager

4.9 Demo:

For **example** if we should add a new state to the application, then we should create a new json for the state inside the states array [] just like below (for example Product Registration):

```
{
"state": "consumerCare",
"events": [
"eventId": "Register Your Product",
"nextStates": [
{
  "condition": [],
  "nextState": "productRegistration"
                                    }
]
}
]}
```

Then the state will be parsed with the help of AppFlowParser.swift class :

static func parseConfig(withJsonPath jsonPath : String) throws -> AppFlowModel? {var stateFlowModel : AppFlowModel?

Then we need to add states in **AppStates**.

static let InAppPurchaseCatalogueView = "InAppPurchaseCatalogueView"

If there is a condition exist in the json framework, then we must add the same in **AppConditions.swift** class.

In Flow Manager (FlowManager.swift) we need to populate State

open func populateStateMap(_ stateMap : inout StateMapType) {}

And we need to populate Condition in

open func populateConditionMap(_ conditionMap : inout ConditionMapType) {}

Now depending on the events, the screen will be loaded with the action items as **eventID** with the **nextStates** as the next information to load **next State** along with the **condition**.

In the BaseFlowManager.swift class, we have an API called :

public func getNextState(_ currentState: BaseState?,forEventId : String?) throws -> BaseState? {}

which will fetch the **stateID** from the **BaseState.swift** class and also will get the viewcontroller to load the new state with help of the below API which other (new) class needs to overide in order to load its content/view:

open func getViewController() -> UIViewController? {}

Then we have another API in the BaseFlowManager.swift class, which is used to come back to the last state/screen of the application:

public func getBackState(_ forState : BaseState) throws -> BaseState? {}

This is the overview of the flow from one state to another and back to the same state.

5. Notes

1. **1.** Please refer Integration Documents of CoCos for implementation details of various CoCos.