|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Document History** | | | | |
| **Version** | **Date** | **Author** | **Section** | **Changes** |
| 1.0 | 04/08/2016 | Deepthi Shivakumar | All | Release 1 features |
| 2.0 | 26/02/2017 | Yogesh HS | All | Release 17.1 features |

|  |  |
| --- | --- |
| Author | Deepthi Shivakumar, Yogesh HS, Rakesh Krishnamurthy |
| Approved by |  |

Contents

[1. INTRODUCTION 2](#_Toc475556013)

[2. INTEGRATION 2](#_Toc475556014)

[2.1 Maven repository (Artifactory based) Integration 2](#_Toc475556015)

[2.2 Library Integration 3](#_Toc475556016)

[2.3 Library versioning 3](#_Toc475556017)

[2.4 .Root gradle changes 3](#_Toc475556018)

[2.5 Proxy dependencies 4](#_Toc475556019)

[3. INITIALIZATION 4](#_Toc475556020)

[4. Android Manifest Changes 4](#_Toc475556021)

[4.1 Other User Permissions 4](#_Toc475556022)

[5. Base Classes 4](#_Toc475556023)

[5.1 BaseClass for Configuration : 4](#_Toc475556024)

[5.2 Classes for Launching Mechanism 4](#_Toc475556025)

[6. Interfaces 5](#_Toc475556026)

[7. Flow Manager 6](#_Toc475556027)

[7.1 TFS Link: 6](#_Toc475556028)

[7.2 What is FlowManager 6](#_Toc475556029)

[7.3 State 6](#_Toc475556030)

[7.4 Condition 7](#_Toc475556031)

[BaseCondition 7](#_Toc475556032)

[7.5 How is Dynamic flow supported in FlowManager 7](#_Toc475556033)

[7.6 Flow Manager pre-requisites 8](#_Toc475556034)

[7.7 Changes in 17.1 BaseFlowManager release: 10](#_Toc475556035)

[7.7.1 Deprecated API’s 10](#_Toc475556036)

[7.7.2 New API’s 10](#_Toc475556037)

[7.8 Base-App Architecture Overview 10](#_Toc475556038)

[7.9 Listeners 10](#_Toc475556039)

[FlowManagerListener 10](#_Toc475556040)

[7.10 Notes 10](#_Toc475556041)

# INTRODUCTION

This document provides an overview of integration procedure for uAppframework library in android mobile applications.

# INTEGRATION

There are two ways to integrate “Mobile App Infrastructure” library with any Android app.

* + 1. **Maven repository based**: At compile time, machine has to be connected with Philips network. Do not follow section 2.2
    2. **Library Integration**: If unable to connect with Philips network then include libraries to your root application. Do not follow section 2.4, 2.5

## Maven repository (Artifactory based) Integration

The easiest and preferred way to use these components is using maven.

All dependent libraries should be downloaded from artifactory.

**Artifactory Path:**

If you are inside Philips network then you can directly refer “**2.5 Gradle dependencies**” section. It will automatically download all nested dependencies from artifactory.

## Library Integration

Need to copy all aar files in libs folder; below are the libraries needed, Please make gradle changes

dependencies {  
compile fileTree(**dir**: **'libs'**, **include**: [**'\*.jar'**])compile **'com.android.support:appcompat-v7:24.2.1'**compile(**group**: **'com.philips.cdp'**, **name**: **'AppInfra'**, **version**: **'1.5.5'**, **ext**: **'aar'**)

}

## . Root gradle changes

**buildscript {**

**repositories {**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/jcenter' }**

**}**

**dependencies {**

**classpath 'com.android.tools.build:gradle:2.2.0'**

**// NOTE: Do not place your application dependencies here; they belong**

**// in the individual module build.gradle files**

**}**

**}**

**allprojects {**

**repositories {**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/libs-release-local-android' }**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/jcenter' }**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/ext-release-local'}**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/libs-release-local-android' }**

**maven { url 'http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/libs-stage-local-android'}**

**}**

**}**

# INITIALIZATION

There is not applicable for uAppFramework. It contains only base classes and interfaces .

# Android Manifest Changes

No special Permission required:

## Other User Permissions

No special Permission required:

# Base Classes

## Classes for Launching Mechanism

An app can be launched as micro app either as activity or fragment. Each micro app need to facilitate both kind of launching mechanism. Micro app (component) framework enables this by providing set of base classes. App or any component which launches a micro app is expected to create an object of either Fragment launcher or Activity launcher and inject to micro app launch API

1. Abstract Class: UiLauncher

Contains methods for setting animations for the activity

**public void** setAnimation(**int** enterAnimResId, **int** exitAnimation)

* 1. Class: ActivityLauncher

Needs to be instantiated for launching micro app as activity.

**public** ActivityLauncher(ActivityLauncher.ActivityOrientation screenOrientation,Bundle bunde)

ActivityOrientation is a defined set of int defs which can be used to specify ScreenOrientation.

* 1. Class: FragmentLauncher

Needs to be instantiated for launching micro app as Fragment.

**public** FragmentLauncher(FragmentActivity fragmentActivity,  
 **int** parentContainerResId,  
 ActionBarListener actionbarUpdateListener)

# Interfaces

1.ActionBarListener

For Updating title on Action Bar this interface needs to be implemented providing definition of below method:

**void** updateActionBar(String titleText,**boolean** enableBackKey);

2. BackEventListener

Each micro app is expected to implement this interface. Micro app needs to return true if they need to consume and handle back key event else return false if they do not want to any actions there by app calls super.onBackPressed(). The below method needs to be defined.

**boolean** handleBackEvent();

3. uAppInterface

uAppframework provides below interface which has standard APIs to initialise, launch and set configuration.

Each micro app will expose one single class for vertical to launch UI where in it needs to implement this interface.

Below methods needs to be implemented for this interface

**public void** init(Context context, AppInfra appInfra);

**public void** launch(UiLauncher uiLauncher, uAppListener uAppListener);

**public void** setLaunchInput(LaunchInput launchInput);

# Flow Manager

## TFS Link:

http://tfsemea1.ta.philips.com:8080/tfs/TPC\_Region24/CDP2/CDP2%20Portfolio/\_git/ufw-android-uappframework

## What is FlowManager

FlowManager is a component used to support navigation among micro apps and to configure app flow dynamically.

Before implementing flowmanager, we need to understand State and Condition classes.

## State

Each micro-app is a state which extends BaseState Class and implement the abstract methods defined by uAppFramework. Please refer next section to understand the different methods to implement.

#### 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 methods.

Constructor

public AnyState() extends BaseState{

super(“state\_id”);

}

The “state\_id” is **case sensitive** and “state\_id” passed in the constructor **should** be same as the state id passed to populateStateMap() API.

Public methods are:

1. **public abstract void** init(Context context);

This API is used to initialize any state. For example, initialize any Micro app or common component that needs to be used as a state.

2. **public abstract void** updateDataModel();

This API is used to pass data to a state. This will be used for example to pass the CTN to consumer care Micro app.

3. **public abstract void** navigate(UiLauncher uiLauncher);

This API is used to navigate from the current state to another state.

4. **public** String getStateID()

These API is used to get state ID from the state Objects.

## Condition

Condition is the class that defines the condition for transitioning between states. To implement condition, we need to extend BaseCondition class defined by uApp-framework. Implement the abstract method isSatisfied method and return true or false based on condition.

Constructor

public AnyCondition() extends BaseCondition {

super(“condition\_id”);

}

The “condition\_id” is **case sensitive** and “condition\_id” passed in the constructor **should** be same as the condition\_id passed to populateConditionMap() API.

Refer the next section for more details about the API’s defined in BaseCondition

### 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** String getConditionID()

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

2. **public abstract boolean** isSatisfied(Context context)

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

## How is Dynamic flow supported in FlowManager

Dynamic configuration of flows in flowmanager is supported through json approach. Sample json below shows the structure to be followed for different states and conditions:

{  
  **"appflow"**: {  
    **"states"**: [  
      {  
        **"state"**: **"splash"**,  
        **"events"**: [  
          {  
            **"eventId"**: **"onSplashTimeOut"**,  
            **"nextStates"**: [

              {  
                **"condition"**: [  
                  **"isLoggedIn"**],  
                **"nextState"**: **"home"**},  
              {  
                **"condition"**: [],  
                **"nextState"**: **"welcome"**}  
            ]  
          }  
        ]  
      }

]

}

}

1. In the above json, splash is a state. There is an event associated with the state onSplashTimeOut. This event has a condition isLoggedIn, flow manager always picks the first state that has a true condition, if none of conditions are satisfied flow manager will pick default state.
2. With this json structure in place, we need to create a state class called splash state and a condition class called isLoggedIn condition and then associate splash state with a state ID and isLoggedIn condition with a condition ID.

## Flow Manager pre-requisites

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

* Create a class which extends the BaseFlowManager defined by uAppFramework.
* Implement the abstract methods populateStateMap and populateConditionMap in the extended class.
* Associate the condition classes and the state classes with their respective ID’s and add it to the maps which are sent as parameters in the methods **populateStateMap** and **populateConditionMap**.
* Since the parsing happens on a different thread, UI needs to be held with a progress dialog/bar.

For detailed explanation please find below the link:

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

## Flow Manager Initialization

1. Flow manager is preferred to be Singleton and we prefer to set it in Application class with below code

FlowManager flowManager = new FlowManager();  
flowManager.initialize(getApplicationContext(), “json\_path”, this);

1. To access it either from Activity or fragment we prefer to get instance through Application class, find below code for reference

* Create below API in Application class

public FlowManager getFlowManager() {  
 return flowManager;

}

* To invoke call getApplicationContext().getFlowManager()

## Forward Navigation using Flow Manager

* For first time, after successful parsing of JSON, to traverse to next State use below code and navigate

BaseState baseState = getApplicationContext().getFlowManager().getNextState(new SplashState(), “onSplashTimeOut”);

baseState.navigate();

## To traverse to next further state in forward direction use below code

BaseState baseState = getApplicationContext().getFlowManager().getNextState(“welcome\_skip”);

baseState.navigate();

## Backward Navigation using Flow Manager

To Navigate Back using flow manager we should use below code

BaseState baseState = getApplicationContext().getFlowManager().getBackState();

baseState.navigate();

while navigating back, “event\_id” should be mandatory as “back” in Appflow.json to navigate to custom State, if not configured any “event\_id” in Appflow.json, by default calling this API returns previous State.

## API description of BaseFlowManager:

**1.public void** initialize (@NonNull final Context context, @NonNull final String jsonPath, @NonNull final FlowManagerListener flowManagerListener)

This API is used to initialize flowmanager like initiating to parse Appflow.json on **separate thread** and call next state from Proposition defined FlowManager.

**2.public** BaseState getBackState(BaseState currentState)

This API is used to get the backstate for back handling.

**3.public abstract void** populateStateMap(final Map<String, BaseState> uiStateMap);

**public abstract void** populateConditionMap(final Map<String, BaseCondition> baseConditionMap);

This API’s are used to populate the various state and condition maps. These methods are abstract and needs to be implemented by the Extending Proposition FlowManager.

**4.public** BaseState getState(String stateId)

**public** BaseState getCurrentState()

This API is used to get the state object corresponding to the stateID passed. There is a method to return the current state of the FlowManager.

**5.public** BaseCondition getCondition(String conditionId)

This API is used to get the condition object for the condition ID that is passes to it.

**6.public** BaseState getNextState(BaseState currentState, String eventId)

This API is used to get the next state. For the first time when getting the next state, currentState is required. “eventId” is the event that triggers the action to fetch next state. This eventId should match with the eventId present in AppFlow.json.

**7.public** BaseState getNextState(String eventId)

This API is used to get the next state, “eventId” is the event that triggers the action to fetch next state. This eventId is case insensitive and should match with the eventId present in AppFlow.json.

8.**public** BaseState getBackState(BaseState currentState)

This API is used to get State while navigating back, “event\_id” should be mandatory as “back” in Appflow.json to navigate to custom State, if not configured any “event\_id” in Appflow.json, by default calling this API returns previous State.

## Changes in 17.1 BaseFlowManager release:

### 7.11.1 Deprecated API’s

@Deprecated  
**public** BaseFlowManager(Context context, String jsonPath) **throws** JsonAlreadyParsedException

The above constructor is deprecated. Please use the default constructor in future and call the initialize method with the FlowManagerListener as the extra parameter. Explanation for FlowManagerListener is explained in section 7.9.

@Deprecated

**public** BaseState getFirstState()

The above API is deprecated, instead made mandate to pass one hard coded state as there should be initial State required to initiate parsing JSON

### 7.11.2 New API’s

**public** BaseFlowManager() {  
}

**public void** initialize(@NonNull **final** Context context, @NonNull **final** String jsonPath, @NonNull **final** FlowManagerListener flowManagerListener) **throws** JsonFileNotFoundException, JsonStructureException, JsonAlreadyParsedException

Introduced Default constructor and API initialize to support parsing on separate thread.

## Base-App Architecture Overview

Please find below the overview for Base App architecture in below link:

[https://atlas.natlab.research.philips.com/confluence/display/BA/BaseApp+A rchitecture+Overview](https://atlas.natlab.research.philips.com/confluence/display/BA/BaseApp+A%20rchitecture+Overview)

## Listeners

### FlowManagerListener

This listener contains the onParseSuccess() method. This method gets a callback from FlowManager once the AppFlow.json file is parsed successfully.

## Notes

2. Please refer interface Spec Doc or Java documents for more details on APIs for each individual component.

3. Please refer demo app for implementation details of various CoCo