|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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](#_Toc476701020)

[2. uApp-framework overview 2](#_Toc476701021)

[3. uApp-framework dependencies 2](#_Toc476701022)

[3.1 Artifactory 2](#_Toc476701023)

[3.2 Library dependencies 3](#_Toc476701024)

[3.3 Root gradle changes 3](#_Toc476701025)

[4. INITIALIZATION 3](#_Toc476701026)

[5. Android Manifest Changes 3](#_Toc476701027)

[5.1 Other User Permissions 4](#_Toc476701028)

[6. Base Classes 4](#_Toc476701029)

[6.1 Classes for Launching Mechanism 4](#_Toc476701030)

[7. Interfaces 4](#_Toc476701031)

[8. Flow Manager 5](#_Toc476701032)

[8.1 TFS Link: 5](#_Toc476701033)

[8.2 What is FlowManager 5](#_Toc476701034)

[8.3 State 5](#_Toc476701035)

[8.4 Condition 6](#_Toc476701036)

[8.5 How is Dynamic flow supported in FlowManager 7](#_Toc476701038)

[8.6 Flow Manager pre-requisites 8](#_Toc476701039)

[8.7 Flow Manager Initialization 8](#_Toc476701040)

[8.8 Forward Navigation using Flow Manager 9](#_Toc476701041)

[8.9 Backward Navigation using Flow Manager 9](#_Toc476701042)

[8.10 uApp Architecture Overview 9](#_Toc476701043)

[8.11 Listeners 10](#_Toc476701044)

[8.12 Notes 10](#_Toc476701046)

# INTRODUCTION

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

# uApp-framework overview

uApp-framework is a component which provides framework to all common components to integrate each common component in uniform way and provides support of flow manager to support dynamic flow of micro-app.

# uApp-framework dependencies

Integration can be done in following ways.

## Artifactory

All dependent libraries should be downloaded from artifactory.

Artifactory path:

<http://maartens-mini.ddns.htc.nl.philips.com:8081/artifactory/simple/libs-release-local-android/com/philips/cdp/uAppFramework/>

## Library dependencies

Kindly make gradle changes as shown below

dependencies {

**compile 'com.philips.cdp:uAppFwLib:1.4.4-release@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

While using uAppframework to use flow manager, flow manager need to be initialized , kindly refer section **8.7**.

# 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)

Micro-app need to write logic to show enter and exit animation on their end based on parameters sent by proposition if required.

* 1. Class: ActivityLauncher

Needs to be instantiated for launching micro app as activity.

**public** ActivityLauncher(ActivityLauncher.ActivityOrientation screenOrientation, ThemeConfiguration dlsThemeConfiguration,int dlsUiKitTheme, Bundle bundle)

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

To define ColorRange,NavigationColor,ContentColor and AccentRange, pass

It through ThemeConfiguration object, find below code for reference.

**new** ThemeConfiguration(**this**, ColorRange.***GROUP\_BLUE***, NavigationColor.***BRIGHT***, ContentColor.***VERY\_DARK***, AccentRange.***GROUP\_BLUE***);

Pass required theme like R.style.defined\_theme to parameter dlsUiKitTheme

To Pass dependent parameters we can inject through bundle extras and pass it as parameter

* 1. Class: FragmentLauncher

Needs to be instantiated for launching micro app as Fragment.

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

# Interfaces

1.ActionBarListener

voidupdateActionBar(String titleText,**boolean** enableBackKey);

Proposition need to implement this interface when launched as Fragment to update micro-app title on Action Bar, micro-app should call above API by sending titleText and enableBackKey on interface sent by proposition and proposition should update title accordingly, micro-apps should send value of enableBackKey as false if foreground screen of micro-app is first screen and true if not the first screen.

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 should expose class with component name appended to Interface for example UserRegInterface.java 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);

Micro-app need to do initialization if required under this API.

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

Micro-app should write logic to navigate based on uiLauncher sent by proposition

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

Proposition should set micro-app specific launch input parameters for example : product registration requires CTN as launch Input to start its micro-app

# 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, proposition 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

### 8.4.1 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** 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 state classes with their respective ID’s and add it to the maps which are sent as parameters in the methods **populateStateMap** and **populateConditionMap**.

Find the below code for reference

**public class** FlowManager **extends** BaseFlowManager {  
  
 **public** FlowManager(Context applicationContext, String path) {  
 **super**(applicationContext,path);  
 }  
  
 **public** FlowManager(){}  
  
 @Override  
 **public void** populateStateMap(**final** Map<String, BaseState> uiStateMap) {  
 uiStateMap.put(AppStates.***WELCOME***, **new** WelcomeState());  
 uiStateMap.put(AppStates.***SPLASH***, **new** SplashState());  
 }  
  
 @Override  
 **public void** populateConditionMap(**final** Map<String, BaseCondition> baseConditionMap) {  
 baseConditionMap.put(AppConditions.***CONDITION\_APP\_LAUNCH***, **new** ConditionAppLaunch());  
 baseConditionMap.put(AppConditions.***IS\_DONE\_PRESSED***, **new** ConditionIsDonePressed());  
  
 }  
}

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 used as single instance and is the responsibility of any app or component and we prefer to set it in Application class with below code

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

1. Flow manager also provides flexibility to initialize by passing raw resource\_id of file, find below code for reference

FlowManager flowManager = new FlowManager();  
flowManager.initialize(getApplicationContext(), R.raw.appflow, this);

Note: Accessing of only raw resource files are supported

1. Flow manager also provides flexibility to initialize by passing mapped Json root object AppFlow, find below code for reference

FlowManager flowManager = new FlowManager();

AppFlowModel appflow = new Gson().fromJson(response, AppFlowModel.class)

flowManager.initialize(getApplicationContext(), appflow, this);

1. Since the parsing of Appflow.json happens on a different thread, based on how any app/component initialize flow manager, UI can be held with a progress dialog/bar if required.
2. 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();

**Note: Logic to navigate to specific screen should be written under the API navigate () of respective state.**

## 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.

## uApp Architecture Overview

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

[https://bitbucket.atlas.philips.comconfluence/display/BA/BaseApp+A rchitecture+Overview](https://bitbucket.atlas.philips.comconfluence/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, user can write logic to continue the flow under this API.

## 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