TOYOTA

Mentor®
A Siemens Business

Window Manager Specification Document

Based on 20170802 HMI-Framework Architecture (V0.2.4)

Aug, 2017

Overview

This document is presenting simplified sequence diagrams for illustrating the interaction between any application and the WindowManager. Assumed that the applications are written in Qt, but can also be HTML5 via a QWebView, or JavaFX. If Java or any other GUI toolkit is used instead of Qt, we do assume that the binding for the Wayland/Weston and a similar QPA interface for configuring the graphics hardware should be in place and have support for the WindowManager API. This binding is not a part of the WindowManager.

First, the application in the system initialization phase, will call requestSurface() with the label as parameter to obtain the areaID. The areaID received from the WindowManager refers to the QT_IVI_SURFACE_ID environment variable related to the IVI Shell that every application has to export in its own running environment.

We do assume that ApplicationFramework is in charge on notifying a specific application when events like start application or show application happen. Such events are generated by a hard key press or button/shortcut selection from HomeScreen. LastUserSession manager might be another usecase, but at the moment this functionality is missing from the AGL system.

When the ApplicationFramework requests to an application to become active/visible, the WindowManager will receive from that application activateSurface() call with the label used for requesting the surface. The PolicyManager and LayoutManager are involved in order to decide if at that time the application can be displayed. In the most favorable case, the WindowManager will notify the Weston/Compositor about its decision and the change the screen layout, emitting notifications to clients about the change. Once notified, the application is responsible that the new surface to become visible and trigger repaint based on this configuration. The Toolkit triggers automatically the swapBuffers(). In the same time the WindowManager notifies the application about the fact that the surface will become visible. In case the policies or the layout requirements are not met the invisible() and inactive() will be emitted to the application.

The decision to display or not an application surface on screen is based on the set of rules from the Policy DB or Layout DB. For simplifying the diagram the WindowResourcesManager, WindowPolicyManager and WindowLayoutManager are not represented. Only the internal calls are included.

In case that a second application is launched and the layout allows both applications to be displayed in the same time, same scenario applies, only that both applications get notified that the surface to draw was changed and a layout update is needed.

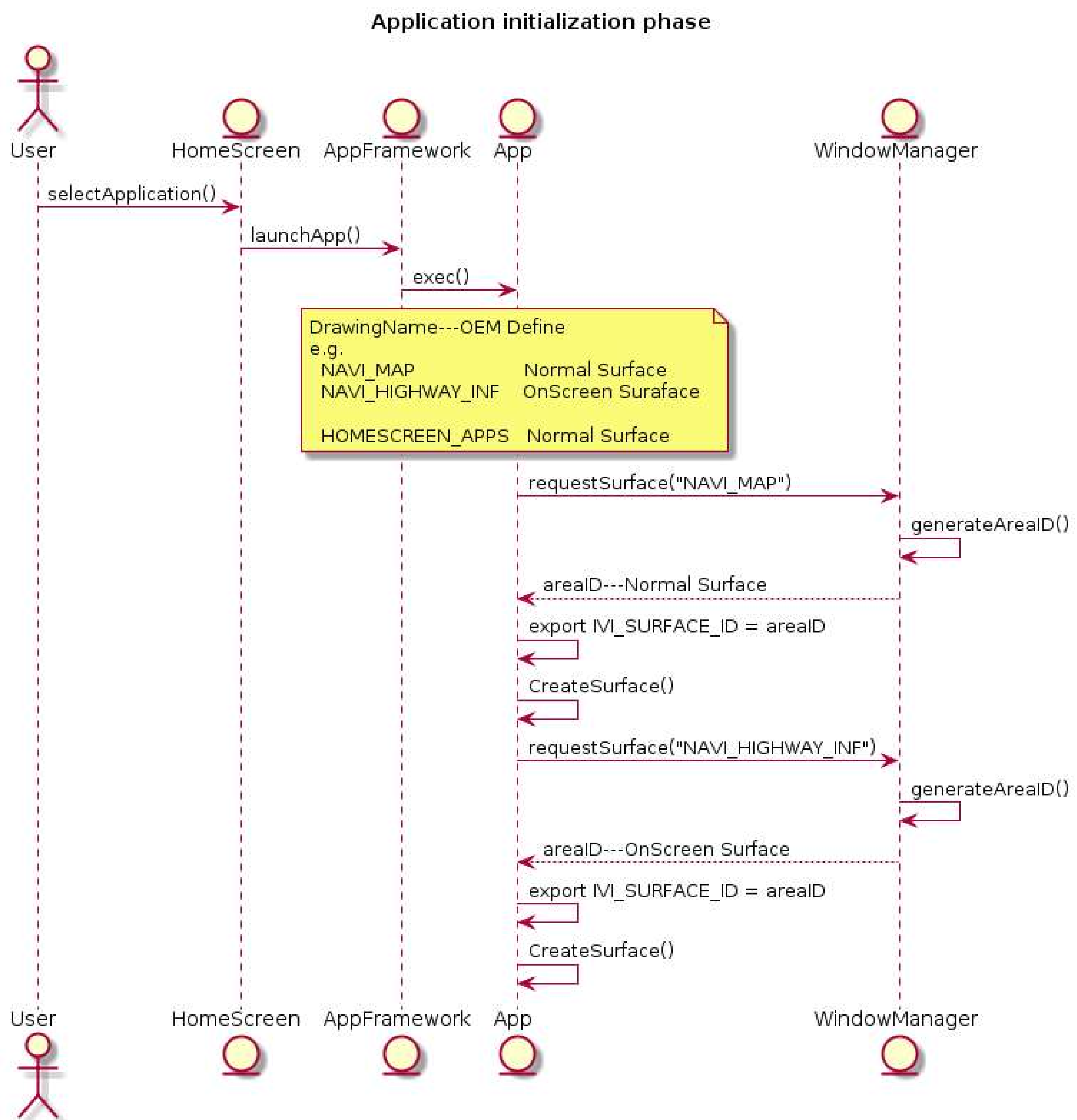
Application Initialization Phase

The ApplicationFramework is an AGL component that is responsible for notifying the applications about the changes in system lifecycle (start, pause, resume, stop, etc.).

Three separate scenarios can be considered for this initialization phase:

1. First application start – this happens when the system is started for the very first time.

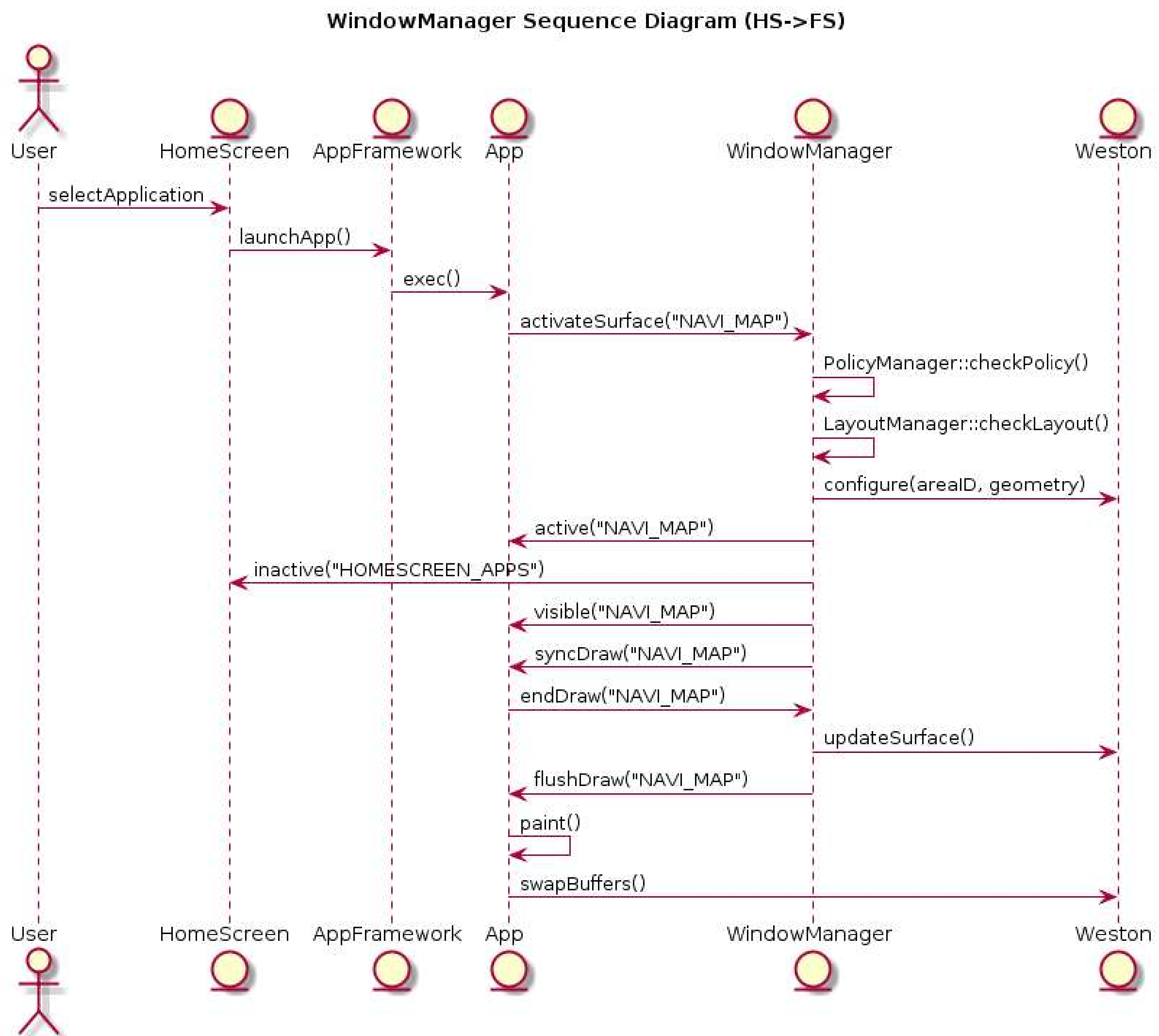
2. Application selected by hard key press or HomeScreen/ApplicationLauncher application shortcut press.



All scenarios reduce to this diagram because in each case the ApplicationFramework component will be involved. The ApplicationFramework is the module in charge with notifying the applications when to start, pause, resume or stop events occur. The ApplicationFramework gets this information from HomeScreen or other AGL components.

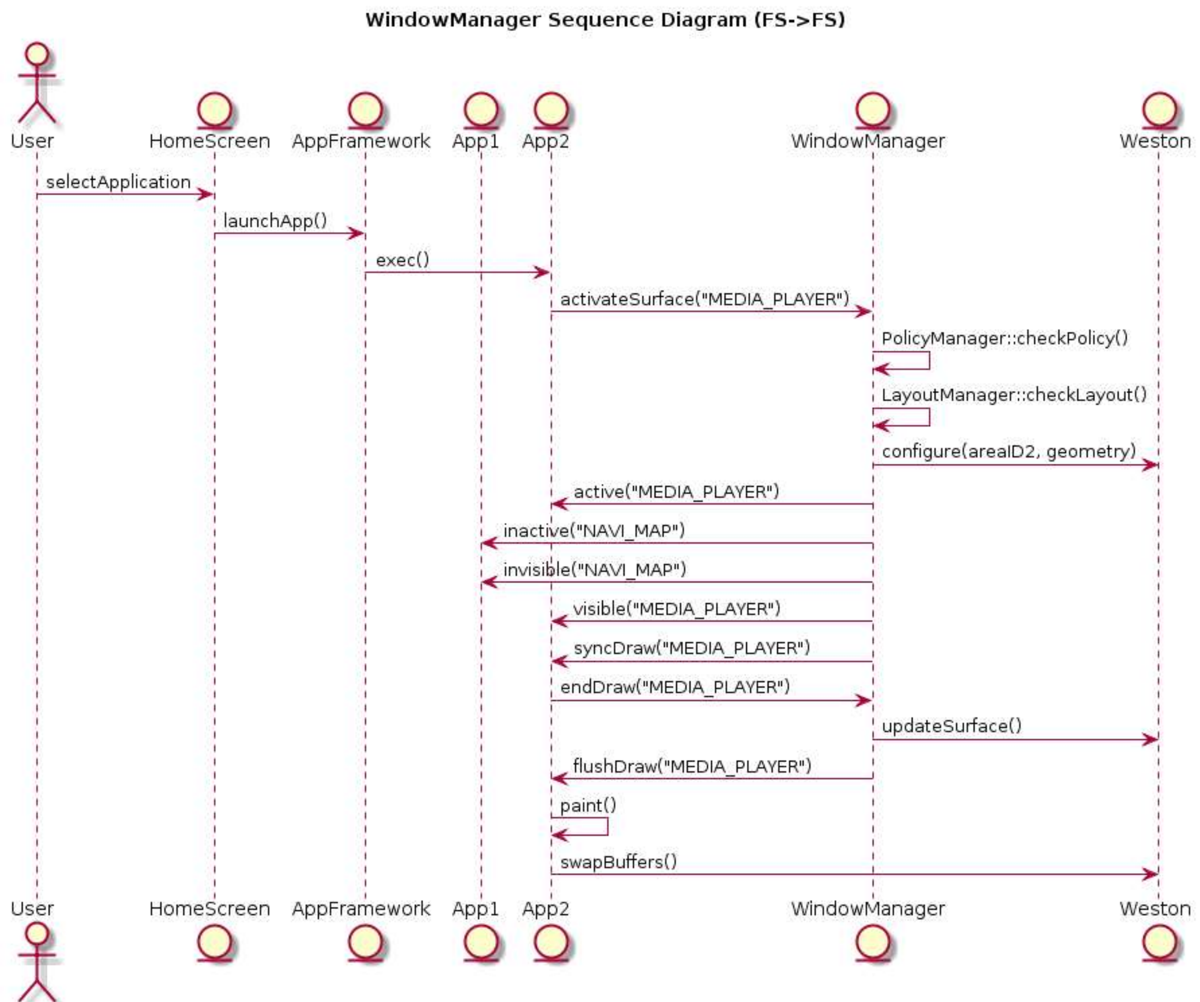
Application Active/Visible Phase

The application activation sequence is simple, the HomeScreen notifies ApplicationFramework what application was selected by user. If the application is not already started, then will initialize it. The Application will call first activateSurface() with the label used previously for configuring the surface. Here internal WindowManager components are responsible to decide if the policies are met and the layout is valid. If this statement is true, the WindowManager will send the new setup to the Weston and then notify the application that will get visible. Next step is to emit the syncDraw() event. From now on the application draws the content to the surface. When finished, the Application will call endDraw(). WindowManager will notify Weston about this event and immediately will issue flushDraw() to the application in order to trigger the swapBuffers() call.

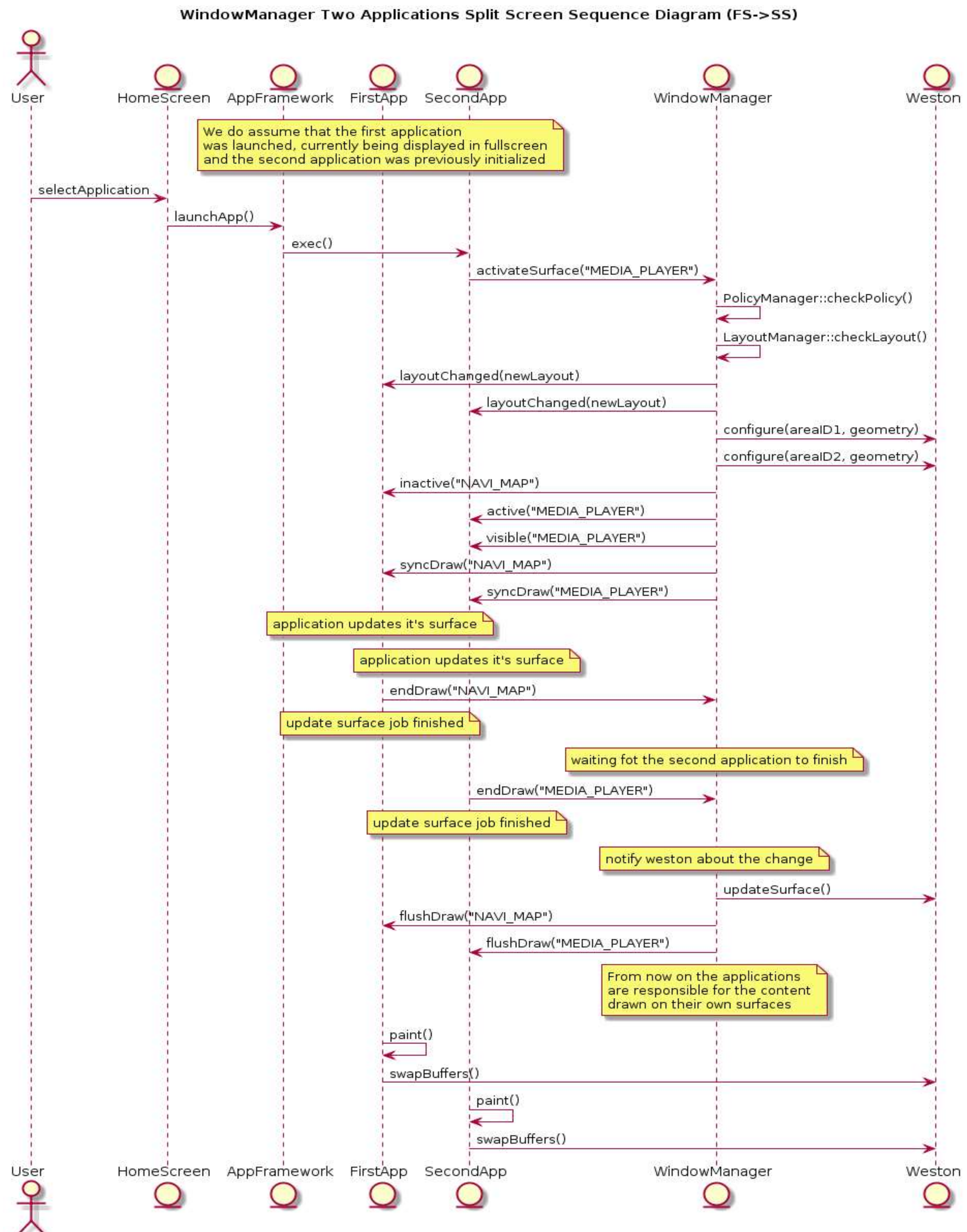


If the Application breaks policy rules or the layout doesn't allow drawing, the WindowManager will call invisible() and inactive() to the application.

A slightly different approach is presented in the next diagram that describes the case when an application running in FullScreen configuration is replaced by another application. In this case the policy specifies that App1 will be replaced by App2 and the FullScreen configuration will be kept.



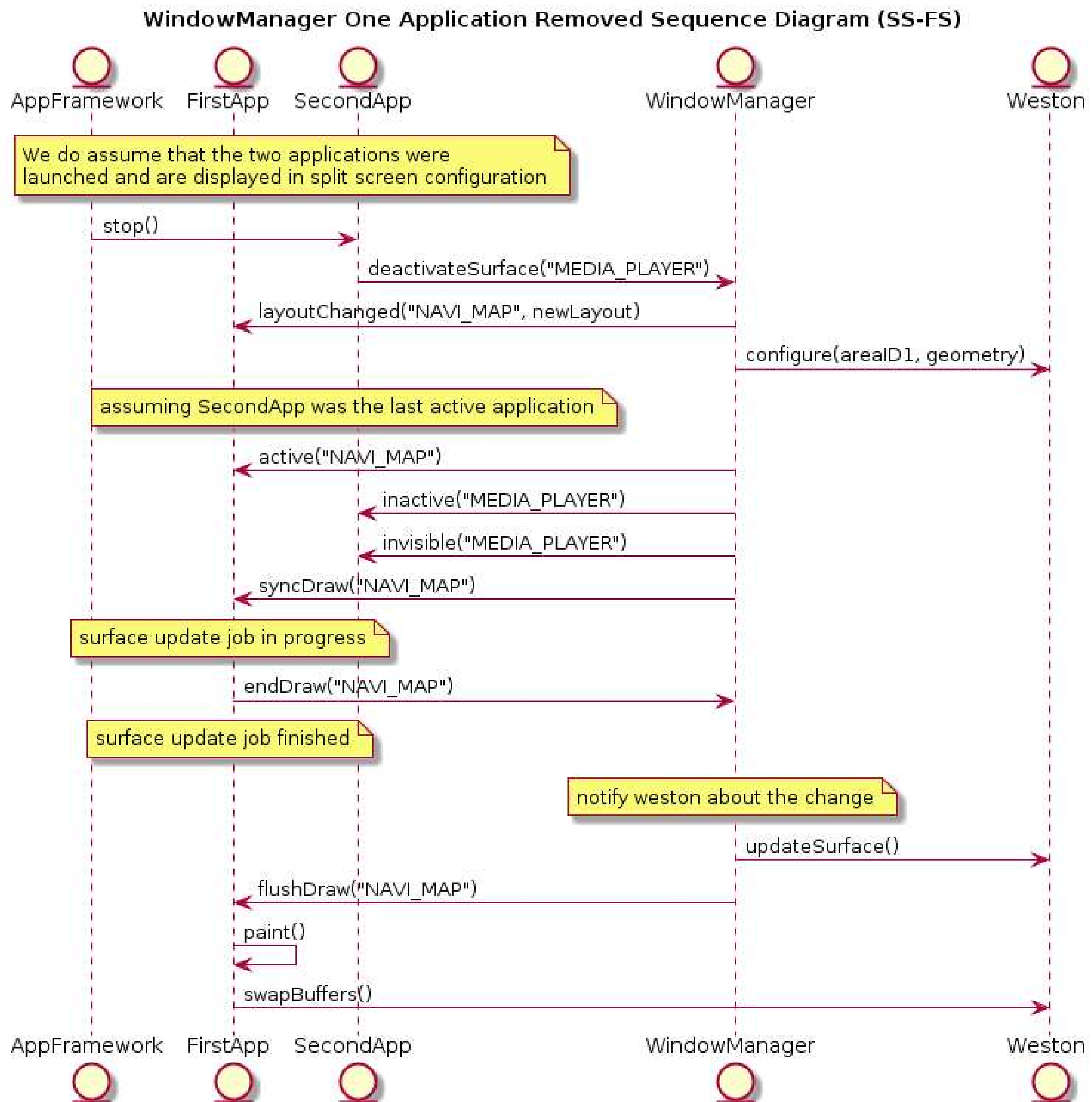
Below is described the scenario for a second application that is selected by the user and the policies allow to display both applications at the same time in SplitScreen configuration. In this case the policy specifies that App1 and App2 can be displayed in the same time in the SplitScreen configuration.



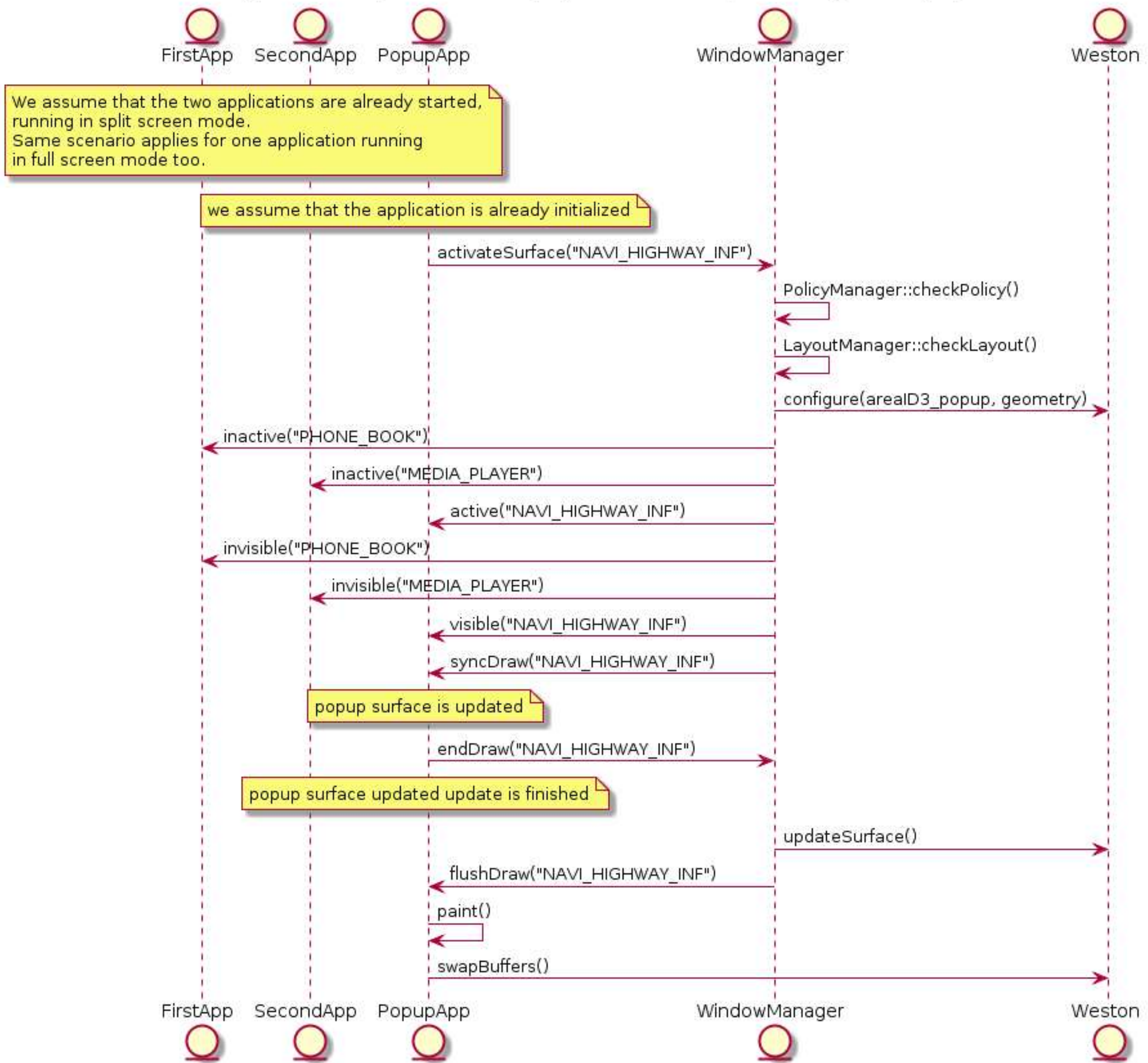
One important aspect is that between the moment when the first application is notified about the fact that the surface used for drawing changed and the moment when both applications updated surfaces

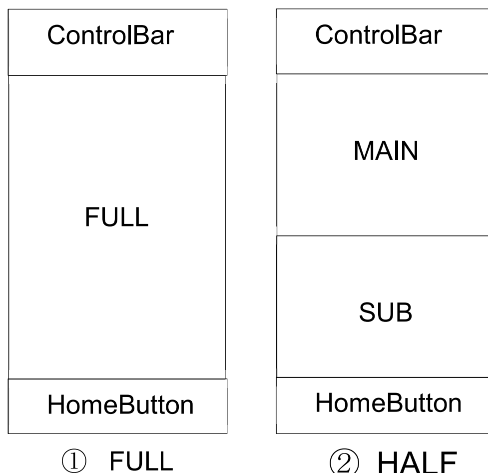
accordingly, the screen will "freeze". It depends on both applications launched how much the screen will remain frozen. The first application that was the only active application until that moment, now becomes inactive, being replaced by the second application.

The next scenario represents the case when one of the previously started applications is no longer wanted to be displayed. This means that when the application receives the notification will immediately call the deactivateSurface() for the requested surface. The sequence diagram is presented below:



The next scenario to be handled is the event of showing a popup. Let's assume that current screen configuration is split-screen and we have two applications displayed. A third application that is already initialized and has requested a surface for a popup to be displayed. In this case will request to the WindowManager to display that popup on screen. The sequence diagram for this scenario is listed below:


Two Applications SplitScreen to Popup FullScreen Sequence Diagram (Popup)
FirstApp
SecondApp
PopupApp
WindowManager
Weston
We assume that the two applications are already started,
running in split screen mode.
Same scenario applies for one application running
in full screen mode too.
we assume that the application is already initialized
activateSurface("NAVI_HIGHWAY_INF")
PolicyManager::checkPolicy()
LayoutManager::checkLayout()
configure(areaID3_popup, geometry)
inactive("PHONE_BOOK")
inactive("MEDIA_PLAYER")
active("NAVI_HIGHWAY_INF")
invisible("PHONE_BOOK")
invisible("MEDIA_PLAYER")
visible("NAVI_HIGHWAY_INF")
syncDraw("NAVI_HIGHWAY_INF")
popup surface is updated
endDraw("NAVI_HIGHWAY_INF")
popup surface updated update is finished
updateSurface()
flushDraw("NAVI_HIGHWAY_INF")
paint()
swapBuffers()



The LayoutManager is based on the information already provided by AGL_HMI-FW_arch document. There are only two layouts supported, Full Screen and Split Screen as described in the picture from left.

The applications can be separated in three different categories:

1. HomeScreen/ApplicationLauncher – will always be displayed in Full Screen configuration.

2. Navigation – that might be displayed in Full Screen or Split Screen configuration. In case of Split Screen, the Main area will be used.

3. Base or Generic applications – that might be displayed in Full Screen or Split Screen. In case of Split Screen, the Main and Sub areas can be used.

Based on this information, a layout json file is used to define the way the layout is configured. An example of this kind of file might be the following:

```
[
    {
      "areas": [
        {
            "height": 1920,
            "name": "ControleBar",
            "width": 1080,
            "x": 0,
            "y": 0,
            "zorder": 0
        },
        {
              "height": 1920,
              "name": "HomeButton",
              "width": 100,
              "x": 0,
              "y": 0,
              "zorder": 0
           }
        ],
```

```
        "name": "HomeScreenBasic"
    },
    {
        "areas": [
            {
                "height": 1280,
                "name": "Main",
                "width": 1080,
                "x": 0,
                "y": 300,
                "zorder": 0
            },
            {
                "height": 1280,
                "name": "Sub",
                "width": 1080,
                "x": 0,
                "y": 600,
                "zorder": 0
            }
        ],
        "name": "ApssHalfBasic"
    }
]
```

Policy Manager

The PolicyManager is the module responsible with reading the policy rules from the database, mapping them into memory, then deciding based on this set of rules if an application surface can be displayed or not. The simplest policy table that can be considered as an example for a stopped vehicle is presented in the table below.

The intention is to give the OEM the possibility to change the rules for each car model. The rules presented are generic, might change on the final version.

Current Screen Configuration	Next Selected App	Resulting Screen Configuration
Full (HomeScreen)	HomeScreen	Full (Homescreen)
Full (HomeScreen)	Navigation	Full (Navigation)
Full (HomeScreen)	Base	Full (Base)
Full (Navigation)	HomeScreen	Full (HomeScreen)
Full (Navigation)	Base	Split (Main: Navigation, Sub: Base)
Full (Base)	HomeScreen	Full (HomeScreen)
Full (Base)	Navigation	Full (Navigation)
Full (Base)	Base	Split (Main: Base1, Sub: Base2)
Split (Main: Navigation, Sub: Base)	Navigation	Full (Navigation)
Split (Main: Navigation, Sub: Base)	Base2	Split (Main: Navigation, Sub: Base2)
Split (Main: Base, Sub: Base2)	Base3	Split (Main: Base3, Sub: Base2)

Resource Manager

Resource Manager is the component that keeps track of every surface created by any application. The current approach is that only one window/surface/area will be available for each application.

This component will have a database that links each application with the areaID. As already described in the sequence diagram. The WindowManager works with areaID, but at some point in order to be able to implement the layout rules, needs to know what areaID corresponds to what application or surface created with a specific label.

We assume this table might look like the following:

Application	Area ID	Surface Label	Surface Type
HomeScreen	1000	HOMESCREEN_APPS	**Normal Surface**
Navigation (Main)	2000	NAVI_MAP	**Normal Surface**
Navigation (Main)	3000	NAVI_HIGHWAY_INF	**OnScreen Surface**
Base	2001	MEDIA_PLAYER	**Normal Surface**
Base2	2002	PHONE_BOOK	**NormalSurface**
Base2	3001	PHONE_CALL	**OnScreen Surface**
Base3	3002	ENGINE_ALERT	**OnScreen Surface**

This table intention is to clarify how the applications are linked with the actual surfaceID's used by the WindowManager to identify the surfaces that needs to control. For configuring the WindowManager, the layers.json file provided within the package needs to be updated.

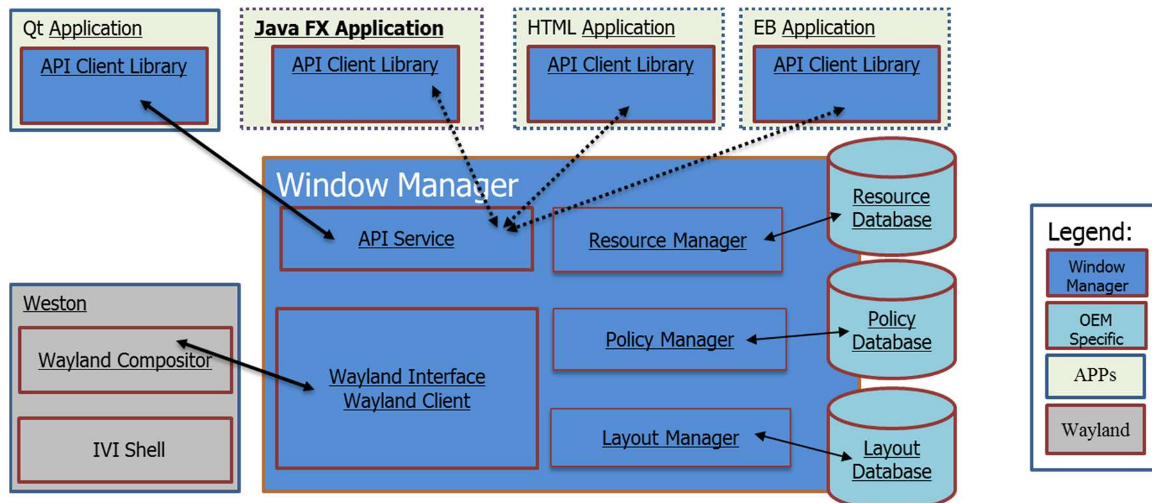
The most important part is that the areaID will only be used by the WindowManager internal components for the communication with the Wayland/Weston.

Window Manager

The WindowManager consists of multiple modules as presented in the diagram bellow. The most important three modules are presented in the previous chapters of this document. The rest will be presented briefly in this section.

The WindowManager provides an interface for communication with other applications or with external AGL components like the HomeScreen. The API client library will provide to the applications the easiest way to use the WindowManager API.

An interface between the Weston/Wayland Compositor and the WindowManager is also included in this diagram, because the WindowManager is intended to be an extension of Weston that meets client requirements. The WindowManager will be a separate process that implements a protocol for communicating with the WaylandCompositor.



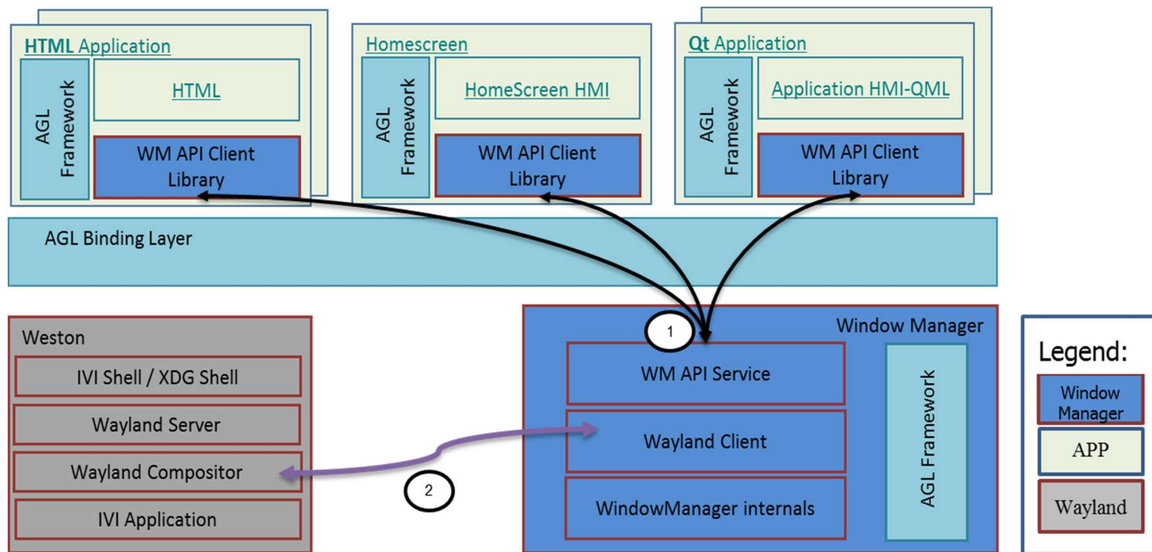
The WindowManager internal structure presented in the picture above remains unchanged. The following diagram presents a layered design that makes easier to understand interaction between applications, AGL components, Weston and WindowManager.

The AGL transport layer is implemented by both WindowManager API Service and the WindowManager API Client library, making easier for developers to create new applications.

Communication between WindowManager and the other system components consists of 2 different aspects:

1. Communication with Applications and HomeScreen that can be considered as a special case of application. Is based on AGL's Transport Layer – AGL Binding mechanism.

2. Communication with Weston. The Wayland client implementation from WindowManager implements a special protocol for talking with Weston's Compositor.



The IVI Application component of Weston it is just a mapping of Wayland surfaces to IDs, so that surfaces can be "addressed" for layout changes globally (in the Wayland Compositor). Without the IVI Application ID's the surfaces are only known to the Wayland Compositor and the Wayland Client that created them, in our case the WindowManager.

WindowManager API

The WindowManager API can be separated into three different sections:

I. WM Functions:

int areaid requestSurface(string label) – Returns the areaid to an application to then create its owned surface with id areaid. The label is a string defined by the OEM, can be NAVI_MAP/NAVI_HIGHWAY_INF for example on the Navigation application.

bool activateSurface(string label) - Activate the application surface corresponding to the label, i.e. make it visible in its assigned area according to layout

bool deactivateSurface(string label) - Deactivate an application surface, based on the label provided by the application as parameter

void endDraw(string label) - Rendering completed for the surface represented by the label provided by the application as parameter

II. WM Events (to clients)

void visible(string label) – Notify that *label* surface has become visible

void invisible(string label) – Notify that *label* surface has become invisible

void active(string label) - Notify that the *label* surface is currently active (has focus)

void inactive(string label) - Notify that the *label* surface has become inactive (has lost focus)

void layoutChanged(string label, layout newlayout) – Notify a client that the *label* surface needs a layout changed

void syncDraw(string label) – Redraw *label* surface after layout change

void flushDraw(string label) – Notify client that it should swap buffers (after SyncDraw)

void popupTimedOut(string label) - Notify a client, that its popup surface timeout is expired (some policy-defined value, e.g. 10 seconds).

III. WM Debugging

void list_drawing_names() – Printing all registered surfaces that have a name attached

void debug_layers() – Printing all the layers currently configured by the WindowManager and their properties

void debug_surfaces() – Printing all the known surfaces and their properties that correspond to the current WindowManager layout configuration (visible surfaces and their geometry)

Annex

For Sequence Diagram design PlantUML was used, for more details visit http://www.planttext.com

The source code for the **Init Application sequence diagram** is:

```
@startuml
title Application initialization phase
actor User
entity HomeScreen
entity AppFramework
entity App
entity WindowManager
User->HomeScreen: selectApplication()
HomeScreen->AppFramework: launchApp()
AppFramework->App: exec()
note over App
  DrawingName---OEM Define
  e.g.
   NAVI_MAP            Normal Surface
   NAVI_HIGHWAY_INF    OnScreen Suraface
      HOMESCREEN_APPS   Normal Surface
end note
App->WindowManager: requestSurface("NAVI_MAP")
WindowManager->WindowManager: generateAreaID()
WindowManager--> App: areaID---Normal Surface
App->App: export IVI_SURFACE_ID = areaID
App->App: CreateSurface()
App->WindowManager: requestSurface("NAVI_HIGHWAY_INF")
WindowManager->WindowManager: generateAreaID()
WindowManager--> App: areaID---OnScreen Surface
```

```
    App->App: export IVI_SURFACE_ID = areaID
    note over App
      Usually the framework is responsible for
      creating the surface
    end note
    App->App: CreateSurface()
@enduml
```

The source code for the **application active/visible sequence diagram** is:

```
@startuml
title WindowManager Sequence Diagram (HS->FS)
actor User
entity HomeScreen
entity AppFramework
entity App
entity WindowManager
entity Weston
User->HomeScreen: selectApplication
HomeScreen->AppFramework: launchApp()
AppFramework->App: exec()
App->WindowManager: activateSurface("NAVI_MAP")
WindowManager->WindowManager: PolicyManager::checkPolicy()
WindowManager->WindowManager: LayoutManager::checkLayout()
WindowManager->Weston: configure(areaID, geometry)
WindowManager->App: active("NAVI_MAP")
WindowManager->HomeScreen: inactive("HOMESCREEN_APPS")
WindowManager->App: visible("NAVI_MAP")
WindowManager->App: syncDraw("NAVI_MAP")
App->WindowManager: endDraw("NAVI_MAP")
WindowManager->Weston: updateSurface()
WindowManager->App: flushDraw("NAVI_MAP")
App->App: paint()
App->Weston: swapBuffers()
@enduml
```

The source code for the **application replaced by another application sequence diagram:**

```
@startuml
title WindowManager Sequence Diagram (FS->FS)
actor User
entity HomeScreen
entity AppFramework
entity App1
entity App2
entity WindowManager
entity Weston
User->HomeScreen: selectApplication
HomeScreen->AppFramework: launchApp()
AppFramework->App2: exec()
App2->WindowManager: activateSurface("MEDIA_PLAYER")
WindowManager->WindowManager: PolicyManager::checkPolicy()
WindowManager->WindowManager: LayoutManager::checkLayout()
WindowManager->Weston: configure(areaID2, geometry)
WindowManager->App2: active("MEDIA_PLAYER")
WindowManager->App1: inactive("NAVI_MAP")
WindowManager->App1: invisible("NAVI_MAP")
WindowManager->App2: visible("MEDIA_PLAYER")
WindowManager->App2: syncDraw("MEDIA_PLAYER")
App2->WindowManager: endDraw("MEDIA_PLAYER")
WindowManager->Weston: updateSurface()
WindowManager->App2: flushDraw("MEDIA_PLAYER")
App2->App2: paint()
App2->Weston: swapBuffers()
@enduml
```

The source code for **two applications getting visible/active sequence diagram**:

```
@startuml
title WindowManager Two Applications Split Screen Sequence Diagram (FS->SS)
actor User
entity HomeScreen
entity AppFramework
entity FirstApp
entity SecondApp
entity WindowManager
entity Weston
note over FirstApp
    We do assume that the first application
    was launched, currently being displayed in fullscreen
    and the second application was previously initialized
end note
User->HomeScreen: selectApplication
HomeScreen->AppFramework: launchApp()
AppFramework->SecondApp: exec()
SecondApp->WindowManager: activateSurface("MEDIA_PLAYER")
WindowManager->WindowManager: PolicyManager::checkPolicy()
WindowManager->WindowManager: LayoutManager::checkLayout()
    WindowManager->FirstApp: layoutChanged(newLayout)
    WindowManager->SecondApp: layoutChanged(newLayout)
    WindowManager->Weston: configure(areaID1, geometry)
    WindowManager->Weston: configure(areaID2, geometry)
    WindowManager->FirstApp: inactive("NAVI_MAP")
    WindowManager->SecondApp: active("MEDIA_PLAYER")
    WindowManager->SecondApp: visible("MEDIA_PLAYER")
    WindowManager->FirstApp: syncDraw("NAVI_MAP")
    WindowManager->SecondApp: syncDraw("MEDIA_PLAYER")
```

```
    note over FirstApp: application updates it's surface
    note over SecondApp: application updates it's surface
    FirstApp->WindowManager: endDraw("NAVI_MAP")
    note over FirstApp: update surface job finished
    note over WindowManager: waiting fot the second application to finish
    SecondApp->WindowManager: endDraw("MEDIA_PLAYER")
    note over SecondApp: update surface job finished
    note over WindowManager: notify weston about the change
    WindowManager->Weston: updateSurface()
    WindowManager->FirstApp: flushDraw("NAVI_MAP")
    WindowManager->SecondApp: flushDraw("MEDIA_PLAYER")
    note over WindowManager
      From now on the applications
      are responsible for the content
      drawn on their own surfaces
    end note
    FirstApp->FirstApp: paint()
    FirstApp->Weston: swapBuffers()
    SecondApp->SecondApp: paint()
    SecondApp->Weston: swapBuffers()
@enduml
```

The source code for **application getting invisible/inactive sequence diagram**:

```
@startuml
title WindowManager One Application Removed Sequence Diagram (SS-FS)
entity AppFramework
entity FirstApp
entity SecondApp
entity WindowManager
entity Weston
note over FirstApp, SecondApp
  We do assume that the two applications were
  launched and are displayed in split screen configuration
end note
AppFramework->SecondApp: stop()
SecondApp->WindowManager: deactivateSurface("MEDIA_PLAYER")
WindowManager->FirstApp: layoutChanged("NAVI_MAP", newLayout)
WindowManager->Weston: configure(areaID1, geometry)
note over SecondApp: assuming SecondApp was the last active application
WindowManager->FirstApp: active("NAVI_MAP")
WindowManager->SecondApp: inactive("MEDIA_PLAYER")
WindowManager->SecondApp: invisible("MEDIA_PLAYER")
WindowManager->FirstApp: syncDraw("NAVI_MAP")
note over FirstApp: surface update job in progress
FirstApp->WindowManager: endDraw("NAVI_MAP")
note over FirstApp: surface update job finished
note over WindowManager: notify weston about the change
WindowManager->Weston: updateSurface()
WindowManager->FirstApp: flushDraw("NAVI_MAP")
FirstApp->FirstApp: paint()
FirstApp->Weston: swapBuffers()
@enduml
```

The source code for **application requesting for a popup sequence diagram**:

```
@startuml
title Two Applications SplitScreen to Popup FullScreen Sequence Diagram (Popup)
entity FirstApp
entity SecondApp
entity PopupApp
entity WindowManager
entity Weston
note over FirstApp, SecondApp
We assume that the two applications are already started,
running in split screen mode.
Same scenario applies for one application running
in full screen mode too.
end note
note over PopupApp: we assume that the application is already initialized
PopupApp->WindowManager: activateSurface("NAVI_HIGHWAY_INF")
WindowManager->WindowManager: PolicyManager::checkPolicy()
WindowManager->WindowManager: LayoutManager::checkLayout()
WindowManager->Weston: configure(areaID3_popup, geometry)
WindowManager->FirstApp: inactive("PHONE_BOOK")
WindowManager->SecondApp: inactive("MEDIA_PLAYER")
WindowManager->PopupApp: active("NAVI_HIGHWAY_INF")
WindowManager->FirstApp: invisible("PHONE_BOOK")
WindowManager->SecondApp: invisible("MEDIA_PLAYER")
WindowManager->PopupApp: visible("NAVI_HIGHWAY_INF")
WindowManager->PopupApp: syncDraw("NAVI_HIGHWAY_INF")
note over PopupApp: popup surface is updated
PopupApp->WindowManager: endDraw("NAVI_HIGHWAY_INF")
note over PopupApp: popup surface updated update is finished
WindowManager->Weston: updateSurface()
```

WindowManager->PopupApp: flushDraw("NAVI_HIGHWAY_INF")

PopupApp->PopupApp: paint()

PopupApp->Weston: swapBuffers()

@enduml