FPM Cookbook

Version 13

08. November 2007
Table of Contents:

1 Introduction ...........................................................................................................................................5

1.1 Support ...............................................................................................................................................5

1.2 Prerequisites .......................................................................................................................................5
  1.2.1 Development Skills ........................................................................................................................5
  1.2.2 System Requirements ......................................................................................................................5
  1.2.3 Important expressions and terms .....................................................................................................5

2 Creating Simple FPM Applications ...............................................................................................6

2.1 Providing UI Building Blocks .........................................................................................................6

2.2 Creating very simple applications .................................................................................................7
  2.2.1 Creating a simple OIF application ......................................................................................................10
  2.2.2 Creating a simple GAF application ....................................................................................................12
  2.2.3 Creating a simple header (for OIF and GAF) .....................................................................................12

2.3 What an FPM based application can configure statically during design-time .............................13
  2.3.1 Configuration of the toolbar ............................................................................................................13
  2.3.1.1 General information .......................................................................................................................13
  2.3.1.2 OIF Toolbar ...................................................................................................................................16
  2.3.1.3 GAF Toolbar ..................................................................................................................................17
  2.3.2 Configuration of the header ............................................................................................................18
  2.3.3 Configuration of the primary help ....................................................................................................19

2.4 FPM Events ..........................................................................................................................................21
  2.4.1 Raising standard events ....................................................................................................................21
  2.4.2 Triggering application specific events ...............................................................................................21
  2.4.3 Reacting on framework events ..........................................................................................................22
  2.4.3.1 WDDOINIT ....................................................................................................................................22
  2.4.3.2 WDDOBEFOREACTION ................................................................................................................23
  2.4.3.3 Action handler (ONACTION…) .......................................................................................................23
  2.4.3.4 IF_FPM_UI_BUILDING_BLOCK~FLUSH .......................................................................................23
  2.4.3.5 IF_FPM_UI_BUILDING_BLOCK~NEEDS_CONFIRMATION .........................................................23
  2.4.3.6 IF_FPM_UI_BUILDING_BLOCK~PROCESS_EVENT ......................................................................23
  2.4.3.7 IF_FPM_UI_BUILDING_BLOCK~AFTER_FAILED_EVENT ...........................................................24
  2.4.3.8 IF_FPM_UI_BUILDING_BLOCK~PROCESS_BEFORE_OUTPUT ......................................................24
  2.4.4 Triggering a data-loss popup .............................................................................................................24

2.5 Initial Screen ........................................................................................................................................24

2.6 Confirmation Screen ...........................................................................................................................25
  2.6.1 Confirmation Screen in OIF .............................................................................................................25
  2.6.2 Confirmation Screen in GAF .............................................................................................................26

2.7 Using FPM Message Manager ..........................................................................................................27
# 3 Using Advanced FPM Features

## 3.1 Variants

## 3.2 Subroadmaps

## 3.3 Using Application Parameters

## 3.4 Dynamic Adjustments

### 3.4.1 Adjusting the toolbar - using the CNR API

#### 3.4.1.1 General information

#### 3.4.1.2 OIF specific parameters

#### 3.4.1.3 GAF specific parameters

### 3.4.2 Adjusting the header - using the IDR API

## 3.5 Sharing Data across different UIBBs

### 3.5.1 Using a Shared Data component

### 3.5.2 Other options

## 3.6 Using an application specific Configuration Controller

### 3.6.1 Event Cancellation

### 3.6.2 Selecting a Variant

### 3.6.3 Adjustment of Events

### 3.6.4 Read the configuration at runtime

### 3.6.5 Change the configuration at runtime

## 3.7 Handling of Transactions

## 3.8 Navigation API

### 3.8.1 How to access the Navigation API

To get access to the Navigation API you can use the interface IF_FPM. It provides the method

### 3.8.2 Interface IF_FPM_NAVIGATION (Runtime class CL_FPM_NAVIGATION)

### 3.8.3 Integration: Navigation in the Eventloop

## 3.9 Composite UIBBs

### 3.9.1 Master-detail or tabbed UIBB

#### 3.9.1.1 General

#### 3.9.1.2 Creating an application that contains a master-detail composite UIBB

#### 3.9.1.3 Changing the master-detail composite UIBB dynamically during runtime

## 4 FPM for Experts

## 4.1 Handling of application specific Popups

## 4.2 Skipping the initial screen

## 4.3 Work-Protect Mode

## 4.4 Application specific Commit

## 4.5 Embedding an FPM

## 4.6 The Resource Manager Interface
1 Introduction

FPM (Floorplan Manager) for Web Dynpro ABAP is a framework to configure UIs (user interfaces) based on the OIF (Object Instance Floorplan) or the GAF (Guided Activity Floorplan).

The following cookbook shall provide you with easy to use instructions how to use the FPM.

Chapter 2 describes the basic functionality of FPM. This allows you to create simple FPM applications.

Chapter 3 describes more advanced FPM functionality, especially dynamic behavior of the header area which is called IDR (Identification Region), dynamic behavior of the toolbar, the data sharing between different views and event handling.

Chapter 4 collects all expert functionality of FPM which is helpful in special cases, but not needed for most of the applications.

1.1 Support

The CSS component for FPM is BC-WD-CMP-FPM (Floorplanmanager Web Dynpro ABAP).

1.2 Prerequisites

1.2.1 Development Skills

Knowledge of ABAP OO and Web Dynpro for ABAP

1.2.2 System Requirements

This document describes FPM as it is available with SAP_ABA 7.0 SP13 and SAP_ABA 7.10 SP3.

1.2.3 Important expressions and terms

- **UIBB:**

  This stands for User Interface Building Block. From an FPM point of view those are the visual parts on the overall screen that come from the application. Technically spoken it is a WD ABAP window (in other words a interface-view). Furthermore it is necessary that the WD ABAP component that window belongs to implements the WD ABAP interface Interface IF_FPM_UI_BUILDING_BLOCK. The content of a UIBB is chosen freely be the applications. The technical key for a UIBB consists of the name of the WD ABAP component, the name of the WD ABAP window and optionally a configuration ID of that WD ABAP component (i. e. the name, type and id of such an configuration).
2 Creating Simple FPM Applications

2.1 Providing UI Building Blocks

The main task of FPM is to connect subscreens provided by the applications. As FPM is a framework for WD ABAP based applications these subscreens are WD ABAP Interface Views. In FPM these subscreens are called UIBB or User Interface Building Block.

In order to communicate with the FPM Framework, it is mandatory that the WD component(s) providing the UIBBs implement(s) at least the WD Interface IF_FPM_UI_BUILDING_BLOCK (see screenshot).
This interface ensures that the application can take part in the FPM event loop; it consists of the following methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLUSH</td>
<td>This method can be used to transport data from your UIBB to a central data model. It’s the first method which is called for each visible UIBB.</td>
</tr>
<tr>
<td>NEEDS_CONFIRMATION</td>
<td>With this Method a UIBB can trigger a data-loss popup before the event processing is continued</td>
</tr>
<tr>
<td>PROCESS_EVENT</td>
<td>Within this method the UIBBs should do all event processing</td>
</tr>
<tr>
<td>AFTER_FAILED_EVENT</td>
<td>This method is called after an event is failed.</td>
</tr>
<tr>
<td>PROCESS_BEFORE_OUTPUT</td>
<td>This method is called for all UIBBs which will be visible after the event processing is finished</td>
</tr>
</tbody>
</table>

### 2.2 Creating very simple applications

At the first create a Web Dynpro application with the following set of parameters:

<table>
<thead>
<tr>
<th>Component</th>
<th>OIF</th>
<th>GAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface View</td>
<td>FPM_OIF_COMPONENT</td>
<td>FPM_GAF_COMPONENT</td>
</tr>
<tr>
<td>Plug Name</td>
<td>FPM_WINDOW</td>
<td>DEFAULT</td>
</tr>
</tbody>
</table>
If not already done, you need to create your UIBBs (User Interface Building Blocks). The WD component that you create will need to implement at least the `IF_FPM_UI_BUILDING_BLOCK` WD interface. Only when this is implemented will FPM know that the Interface views from this component are participating in the final view composition. For more information on UIBBs see chapter 2.3.

Then you have to create a configuration for your application. To do this call the WD ABAP configuration editor for your new application (the easiest way is to call it out of SE80, see screenshot)

The configuration editor is a Web Dynpro application itself and will start up in a browser:
To create an application configuration enter a configuration name and press the Create link (Configuration names are global! You can’t use the same configuration name for different applications).

An application configuration is not much more than a composition of component configurations. An FPM application is composed of a lot of different Web Dynpro components; however most of them are created dynamically at runtime. There are only 2 components which are always there: a floorplan specific component (FPM_GAF_COMPONENT for GAF and FPM_OIF_COMPONENT for OIF) and a component for the Header Area (FPM_IDR_COMPONENT). So the main task in configuring an FPM application is the configuration of these two components.

After you have created the application configuration the screen looks more or less like the following screenshot:
In the hierarchical table in the middle of the right side the basic component structure of any FPM application is displayed - the main component is FPM_OIF_COMPONENT (respectively FPM_GAF_COMPONENT for GAF). This component uses a 2\textsuperscript{nd} component, which is FPM_IDR_COMPONENT.

In order to create component configurations you have to enter the name in the Configuration column of this table, mark the line and press 'Go to Component Configuration'.

For very basic applications there is no need to configure the header. Therefore simply create a configuration for the main component. This configuration is now specific for each floorplan.

2.2.1 Creating a simple OIF application

After you have created the configuration for component FPM_OIF_COMPONENT your screen should look more or less like this:
The editing is done in the tab strip on the lower right side.

On the left hand side of this tab you see the structure of the configuration context. If you select a node or an item in this tree on the right hand side the related attributes can be edited.

For a simple application only the node Variant is relevant. You have to create at least one variant (name and id are of no relevance). Then, for each view that you want to have in your application you have to create a mainview (the name you provide here will be displayed as the name of the tab).

Note: The Id has to follow the guidelines for creating ids (like no spaces, no special characters expect underscore etc.)

The OIF offers the possibility for complex applications to use subviews, which will be rendered as links within each tab. Therefore you have to specify at least one subview for each mainview. As long as you only specify one subview there will be no subview rendering and therefore name and id of the subview are of no relevance. If you specify more than one subview for a mainview then the subview name will be displayed as link text.

Finally you have to add the UI Building Blocks (UIBB) to each subview. You have to specify at least one UIBB for each subview, but you can add as many UIBBs as you like to each subview. These UIBBs will be rendered below each other (i.e. in the same order as it is configured here in the editor).

When specifying a UIBB the field’s component and view are mandatory, where component is the name of the Web Dynpro component implementing IF_FPM_UI_BUILDING_BLOCK and view is the name of the interface view containing the view to be displayed.

The UIBB here is nothing but the interface view that has been defined in your application. The interface view of your application will serve as the most atomic unit of application UI that can be configured. By assigning the interface view as a UIBB you are in effect now composing how your application content area will look when the application runs within FPM.

The component name should be the name of your application component. The view should be the name of your interface view. For a simple application the rest of the fields can be left with their default values.

At the end the configuration for a simple OIF with 2 mainviews without visible subviews should look like this:
2.2.2 Creating a simple GAF application

Unlike an OIF the GAF is composed of several roadmap steps and the UIBBs will be assigned to those steps. One roadmap step may contain one or more UIBBs.

A simple configuration for the component FPM_GAF_COMPONENT could look like this:

In the example shown above two main steps are declared. You have to populate the input fields for the following nodes:

- **VARIANT**: You have to create at least one entry for node VARIANT. Simply enter an ID (e.g. VARIANT1). For simple applications this is of no relevance.
- **MAINSTEP**: Enter qualifiers for id and name. The entity name will be shown at runtime beneath the corresponding main step.
- **MAINSTEP_UIBB**: Enter the values for Component and View as depicted above via the red arrows. If you make a very simple GAF application, all views would belong to one WD component. Therefore the component in all entries for MAINSTEP_UIBB would be the same. In the field View put the name of the WD interface-view (not the name of the WD view itself).

2.2.3 Creating a simple header (for OIF and GAF)

Save your configuration, go back to the application configuration and now create a simple configuration for the FPM_IDR_COMPONENT, just as you did for FPM_OIF_COMPONENT (respectively FPM_GAF_COMPONENT). That could look like this:
Enter values for your application title and tooltip as marked above via the red arrows. You don’t need to populate the node IDR_EXTENDED as those values will not be displayed within a GAF application. Save the configuration and go back to the application configuration.

The first very simple application is finished now. Press the button “test” and see how it works.

### 2.3 What an FPM based application can configure statically during design-time

#### 2.3.1 Configuration of the toolbar

##### 2.3.1.1 General information

Within the node toolbar the application has the possibility to configure the use of standard buttons/button-choices, self-configured buttons/button-choices and navigation menu bars.

Directly linked to the toolbar is the attribute “duplicate” which can be set, so that the toolbar is duplicated on the very bottom of the screen (see screenshot).

One of the features of FPM is that the toolbar is automatically build-up according to the current UI-Guidelines. This means that the available standard buttons (like edit, print, check, …) just have to be made visible via the configuration and they are placed automatically within the toolbar in the correct sequence (see next screenshot depicted below). For a few standard buttons it is possible to set the label and a tooltip because these texts are application-specific (e.g. ‘next object’, there the ‘object’ should be replaced by the name of the object’s type).
Below the toolbar node there is a node for each standard button or button choice. Each of those standard buttons/button-choices is connected to a certain FPM-Event (the edit button has e.g. the FPM-Event cl_fpm_event=>gc_event_edit). The connection to those FPM-Events is hard-coded and can’t be changed.

Some of those events are handled by the FPM (Navigation events, Save), but for most of them the FPM only takes care about informing all affected UIBBs. So any event processing has to be done by the applications. This means e.g. that although FPM offers a print button, there is no print support in FPM. This button is only offered to ensure that it’s rendered according to the UI Guidelines. Any print functionality has to be implemented by the applications.

There are certain standard buttons which can be made available as button-choices as well (print, print preview, send and new). Please note, if both are marked as visible (the print button as well as the print button-choice), then only the button is rendered within the toolbar!

Below the button-choice node you can define several menu entries with certain properties. For those entries there are no FPM Event IDs predefined by FPM, you have to define your own event name in field FPMEventid. This event will be raised by the FPM when this menu entry has been pressed:
Finally every floorplan offers the OTHER_FUNCTIONS node. In here application-specific buttons/button-choices/separators can be defined. The sequence of those elements can be defined via the attribute "otherFunctionIndex". Also an own FPMEventId has to be set. The attribute "elementId" is only relevant, if you want to change the properties of a UI-Element dynamically during runtime.

To configure the navigation menu bars, you have to use the nodes YOU_CAN_ALSO and RELATED_LINKS. Prerequisite for filling those menu bars is the configuration of report launchpads. This has then to be entered within the attributes “Role” and “Instance” from the report launchpad customizing.
2.3.1.2 OIF Toolbar

In every variant of the OIF, there exists exactly one toolbar. This toolbar contains more standard buttons than the toolbar in GAF. Additionally, we have – beside the OTHER_FUNCTIONS node – two more possibilities to create application-specific buttons. These are the nodes ACTIVATION_FUNCTIONS and ALTERNATE_FUNCTIONS.

**Hint:** To get all properties defined use the arrows on the very bottom of the table.

Events which are raised from the OIF toolbar:

<table>
<thead>
<tr>
<th>Button</th>
<th>Event which is raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation Function section</td>
<td>self-defined via configuration</td>
</tr>
<tr>
<td>Alternate Activation section</td>
<td>self-defined via configuration</td>
</tr>
<tr>
<td>Check</td>
<td>cl_fpm_event=&gt;gc_event_check</td>
</tr>
<tr>
<td>Delete Object</td>
<td>cl_fpm_event=&gt;gc_event_delete_current_object</td>
</tr>
<tr>
<td>Edit</td>
<td>cl_fpm_event=&gt;gc_event_edit</td>
</tr>
<tr>
<td>Load Draft</td>
<td>cl_fpm_event=&gt;gc_event_load_draft</td>
</tr>
<tr>
<td>New</td>
<td>cl_fpm_event=&gt;gc_event_new</td>
</tr>
<tr>
<td>Next Object</td>
<td>cl_fpm_event=&gt;gc_event_next_object</td>
</tr>
<tr>
<td>Other function section</td>
<td>self-defined via configuration</td>
</tr>
<tr>
<td>Previous Object</td>
<td>cl_fpm_event=&gt;gc_event_previous_object</td>
</tr>
<tr>
<td>Print</td>
<td>cl_fpm_event=&gt;gc_event_print</td>
</tr>
<tr>
<td>Print Preview</td>
<td>cl_fpm_event=&gt;gc_event_print_preview</td>
</tr>
<tr>
<td>Read Only</td>
<td>cl_fpm_event=&gt;gc_event_read_only</td>
</tr>
<tr>
<td>Redo</td>
<td>cl_fpm_event=&gt;gc_event_redo</td>
</tr>
<tr>
<td>Refresh</td>
<td>cl_fpm_event=&gt;gc_event_refresh</td>
</tr>
<tr>
<td>Save As</td>
<td>cl_fpm_event=&gt;gc_event_save_as</td>
</tr>
<tr>
<td>Save Draft</td>
<td>cl_fpm_event=&gt;gc_event_save_draft</td>
</tr>
<tr>
<td>Send</td>
<td>cl_fpm_event=&gt;gc_event_send</td>
</tr>
<tr>
<td>Start Over</td>
<td>cl_fpm_event=&gt;gc_event_start_over</td>
</tr>
<tr>
<td>Undo</td>
<td>cl_fpm_event=&gt;gc_event_undo</td>
</tr>
</tbody>
</table>
### 2.3.1.3 GAF Toolbar

Within GAF, every main step below a variant has its own toolbar and also every substep below a substep variant has its own toolbar:

<table>
<thead>
<tr>
<th>Node Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL_SCREEN</td>
</tr>
<tr>
<td>VARIANT</td>
</tr>
<tr>
<td>CONF_STEP_VIEW</td>
</tr>
<tr>
<td>MAINSTEP</td>
</tr>
<tr>
<td>MAINSTEP (Step!)</td>
</tr>
<tr>
<td>VARISTEP_LIBB</td>
</tr>
<tr>
<td>SUBSTEP_VARIANT</td>
</tr>
<tr>
<td>SUBSTEP_VARIANT[1]</td>
</tr>
<tr>
<td>SUBSTEP</td>
</tr>
<tr>
<td>SUBSTEP_LIBB</td>
</tr>
<tr>
<td>TOOLBAR</td>
</tr>
<tr>
<td>TOOLBAR_LIBB</td>
</tr>
<tr>
<td>EXIT_TO_MAINSTEP</td>
</tr>
<tr>
<td>OTHER_FUNCTIONS_LIBB</td>
</tr>
<tr>
<td>RELATED_LINKS_LIBB</td>
</tr>
<tr>
<td>YOU_CAN_ALSO_LIBB</td>
</tr>
<tr>
<td>MAINSTEP (Step0)</td>
</tr>
</tbody>
</table>

So the toolbar configuration can be exactly defined for every step.

There are some special elements:

- The **FINISH** button can only be defined within a main step toolbar. If this button is pressed, the roadmap will be executed sequentially, this means the FPM tries to automatically navigate to the next step as many times as possible. If an error occurs in a certain screen then the process will be stopped at this screen. Else the FPM will lead the user to the last step of the roadmap (before the confirmation screen).

- Within the node **EXIT_TO_MAINSTEP**, the attributes “label” and “tooltip” can be defined (only within substep toolbars). This button will lead the user from a substep back to its main step.

- One **FINAL_ACTION** can be configured per variant. This final action defines a special label and FPMEventId for the Next-Step button on the last step before the confirmation screen.
Events which are raised from the GAF toolbar:

<table>
<thead>
<tr>
<th>Button</th>
<th>Event which is raised</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other function section</td>
<td>self-defined via configuration</td>
</tr>
<tr>
<td>Exit to main step</td>
<td>cl_fpm_event=&gt;gc_event_exit_to_main_step</td>
</tr>
<tr>
<td>Next step</td>
<td>if no final action is defined:</td>
</tr>
<tr>
<td></td>
<td>cl_fpm_event=&gt;gc_event_next_step</td>
</tr>
<tr>
<td></td>
<td>with defined final action: the self-configured event in the final action node and the next step event are raised</td>
</tr>
<tr>
<td>Previous step</td>
<td>cl_fpm_event=&gt;gc_event_previous_step</td>
</tr>
<tr>
<td>Save Draft</td>
<td>cl_fpm_event=&gt;gc_event_save_draft</td>
</tr>
</tbody>
</table>

### 2.3.2 Configuration of the header

The header (also known as IDR: Identification Region) consists of the following three areas: Header area, ticket area, items area. Only the header and the ticket area can be configured statically. If you would like to add/change properties in the items area you need to do it dynamically. In that case please refer to chapter 3.4.2. Besides, the ticket area is only used for OIF based applications.

The following entities may be configured statically:

- **Header area**: The application title and the corresponding tooltip:

- **Ticket area**: The top ticket, the bottom ticket and their corresponding tooltips:
2.3.3 Configuration of the primary help

The primary help of a floorplan application is always displayed above the Content Area when the Quick Help of the screen is switched on. By right-clicking your mouse a context menu opens, take the menu entry 'Display Quick Help' and the help mode is activated:

![Toolbar configuration image]

To configure this help text inside your application configuration, navigate to the node EXPLANATION (in OIF under node SUBVIEW, in GAF under node MAINSTEP and SUBSTEP). Here you have the possibility to create an explanation per each subview and step/substep.
To configure an explanation you have two alternatives. You can enter a text by your own (type String) or you can enter a document object which has to be created on backend side (type DOKU_OBJ). To create a document object you have to use transaction SE61. As document class choose ‘General Text’. Now you are able to create those objects and use them inside your application configuration. Afterwards it will be displayed inside the primary help.

**Document Maintenance: Initial Screen**

**Settings**

- **Document Class**: General text
- **Language**: English

**Document**

- **Name**: ACTION

**Document Source**

- **Local**
- **Get via RFC Destination:**
2.4 FPM Events

As with standard Web Dynpro ABAP, a user interaction is reflected by a Web Dynpro action. However, in case this interaction is not only local to single component but (potentially) affects other parts or components in the application it needs to be transferred to an FPM event (see chapter 2.4.1 + 2.4.2).

This FPM event passes a FPM phase model (also called “event loop”) which is integrated into the WD phase model. Within the FPM event loop all involved components can participate in the processing of the event (see chapter 2.4.3).

If the FPM event results in another screen assembly (e.g. navigation to another screen in the guided activity floorplan or the selection of another view/subview in the object instance floorplan this is handled by the FPM itself (e.g. there is no need for firing plugs by the application or similar).

2.4.1 Raising standard events

Normally, most of the standard user interactions in a floorplan based application are triggered from the generic parts of the floorplan (e.g. previous/next in GAF, or view switches in OIF). For these interactions the FPM automatically initiates the FPM event loop. Furthermore, since these “standard events” are “known” by the FPM they can in many cases be handled in a generic way. Additionally, the application can react on those events as described in chapter 2.4.3.

However, there might be scenarios where a standard event needs to be triggered from within an application specific UI building block as well: e.g. automatically leaving (i.e. by-passing) the initial screen in case all necessary startup parameters have already been provided as URL parameters.

Generally, each FPM event is represented at runtime by an instance of the class CL_FPM_EVENT. This class encapsulates all information which is needed to execute this event, like ID and additional parameters.

Triggering the FPM event loop for a concrete event normally requires two steps:

1. Create an instance of CL_FPM_EVENT with the appropriate attributes.
   For all the standard events IDs, there are constants available at IF_FPM_CONSTANTS.
2. Raise the event by calling the method IF_FPM~RAISE_EVENT and passing on the instance of CL_FPM_EVENT.

For those cases where the event does not need any additional parameters beside the event-id itself, the FPM offers an additional convenience method RAISE_EVENT_BY_ID which makes the first step obsolete. By this, raising an FPM event typically happens in the following way:

   data lo_fpm type ref to if_fpm.
   lo_fpm = cl_fpm_factory=>get_instance( ).
   lo_fpm->raise_event_by_id( IF_FPM_CONSTANTS=>GC_EVENT-LEAVE_INITIAL_SCREEN ).

An example of an event with parameters is given in the next chapter.

NOTE: Since at the point the event is raised, it is unknown whether the event can be executed successfully or not, there must be no coding after the call to RAISE_EVENT[._BY_ID]!

2.4.2 Triggering application specific events

For raising an application specific event the same rules apply as described in the previous chapter. The only difference is that the FPM – since it does not know the semantics of the event – does not perform any specific actions for this event. However, the processing of the event is identical. That means all involved components participate in the event loop in the same way as with ‘standard events’ (see chapter 2.4.3).

An example of triggering an application specific event including event parameters:

   data: lo_fpm type ref to if_fpm,
   lo_event type ref to cl_fpm_event.

   create object lo_event
   exporting
iv_event_id = 'DELETE_AIRPORT'.

lo_event->mo_event_data->if_fpm_parameter_set_value(  
iv_key = 'AIRPORT_ID'  
iv_value = lv_airport_id  
).

lo_fpm = cl_fpm_factory->get_instance( ).  
lo_fpm->raise_event( io_event = lo_event ).

NOTE: Since at the point the event is raised it is unknown whether the event can be executed successfully or not, there must be no coding after the call to RAISE_EVENT[ função_ID]!

2.4.3 Reacting on framework events

The FPM event loop is integrated into the Web Dynpro phase model:

The following chapters describe the methods an UI building block inherits by implementing the interface IF_FPM_UI_BUILDING_BLOCK and some relevant Web Dynpro methods. For a description of the interfaces IF_FPM_SHARED_DATA, IF_FPM_WORK_PROTECTION, IF_FPM_RESOURCE_MANAGER, and IF_FPM_TRANSACTION please refer to the corresponding chapters.

2.4.3.1 WDDOINIT
This standard Web Dynpro method is called exactly once in the lifetime of a Web Dynpro component by the
Web Dynpro runtime. This method should be used to generally initialize your component: e.g. initialize
attributes, create helper classes, etc.

In case a UI building block often needs to call methods on the IF_FPM interface the CL_FPM_FACTORY
class can be queried once in WDDOINIT and the retrieved IF_FPM instance can be kept as component
attribute for easier access:

```java
method WDDOINIT.
  wd_this->mo_fpm = cl_fpm_factory=>get_instance().
endmethod.
```

2.4.3.2 WDDOBEFOREACTION

This standard Web Dynpro method is called by the Web Dynpro runtime on all visible UI building blocks
when the end-user triggered a round trip.

According to the Web Dynpro programming guideline, generic validations should be handled in this method;
e.g. check that all mandatory fields are filled.

2.4.3.3 Action handler (ONACTION…)

If the end-user triggered a request from within your UI building block (e.g. application specific button) the
registered Web Dynpro action handler is called.

Basically, there are now two options how to proceed:

If the user interaction does not have any (side) effects on other UI building blocks and there is no need for
FPM features like data-loss popups, the action can be handled locally in your UI building block by means of
standard Web Dynpro programming; e.g. selection of another radio-button leads to different
enabled/disabled settings of other controls on the same view.

However, for all actions which might have (side) effects to other UI building blocks an FPM event must be
raised (see chapters 2.4.1 and 2.4.2).

2.4.3.4 IF_FPM_UI_BUILDING_BLOCK~FLUSH

This is the first method called after the FPM event loop has started.

In this method the UI building block needs to transport all modified data from the views to other components
the UI building block wants to communicate with later on.

Normally this data transport is done completely automatically via Web Dynpro context mapping. Therefore, a
specific implementation of this method is only needed if these automatic mechanisms are not used.

2.4.3.5 IF_FPM_UI_BUILDING_BLOCK~NEEDS_CONFIRMATION

In this method the UI building block can request that the further event processing is blocked and the end-
user is asked for a confirmation. This confirmation is rendered in a popup. Depending on the action the end-
user does on the popup, the event loop continues or is being cancelled. For more details please see chapter
2.4.4.

2.4.3.6 IF_FPM_UI_BUILDING_BLOCK~PROCESS_EVENT

In this method the UI building block needs to:

- check for local consistency
- do the actual event processing

The “local check” is desired to inform the end-user about potential input errors as soon as possible:
according to the UX guideline checks should be performed on an ongoing basis, as long as they are not too
much performance intensive. E.g. when switching from one view to another view in an Object-Instance
floorplan, the UI building block which is left should check for local consistency.

However, this does not free the application to do a complete check (including performance critical checks)
before saving. This must be handled in the method IF_FPM_TRANSACTION_CHECK BEFORE_SAVE.

Besides the consistency check this method contains the actual processing of the event. For this the current
event can be identified through the attributes mv_event_id and MO_EVENT_DATA on the passed on event
instance io_event. Depending on whether the event could be processed successfully or not the exporting
parameter EV\_RETURN must be filled with either IF\_FPM\_CONSTANTS\~GC\_EVENT\_RESULT\~OK or IF\_FPM\_CONSTANTS\~GC\_EVENT\_RESULT\~FAILED.

By this, a typical implementation of PROCESS\_EVENT looks like:

```abap
IF io_event->mv_event_is_validating = abap_true.
  Do local checks and report messages if needed
ENDIF
CASE io_event->mv_event_id.
  WHEN XYZ
    Handle event and fill EV\_RETURN accordingly with a value from IF\_FPM\_CONSTANTS\~GC\_EVENT\_RESULT
ENDCASE.
```

If the event processing requires further user interaction (e.g. ask for further data in a popup) the event processing can be deferred by returning EV\_RETURN = IF\_FPM\_CONSTANTS\~GC\_EVENT\_RESULT\~DEFER. For details see chapter 4.1.

2.4.3.7 IF\_FPM\_UI\_BUILDING\_BLOCK\~AFTER\_FAILED\_EVENT
This method is called by the FPM if an event could not be processed successfully. In this case the UI building block needs to take care that its UI is reverted to its state before the user interaction.

Example:
The changed lead selection in a table triggers the display of the details of the new line in another UI building block. Now the event could fail if the UI building block for the details contains unsaved data for the previously selected table line. Since after the (failed) event the detail form still contains the details of the original table line, the lead selection must be reverted to the original table line as well.

If the PROCESS\_EVENT method of the current UI building block has been processed successfully but the event processing failed due to a problem in another UI building block, the actual event processing needs to be reverted as well. This situation is indicated by the parameter IV\_REVERT.

2.4.3.8 IF\_FPM\_UI\_BUILDING\_BLOCK\~PROCESS\_BEFORE\_OUTPUT
As last method on the UI building block interface PROCESS\_BEFORE\_OUTPUT is called. Here typically the data which needs to be displayed is read from the model.

2.4.4 Triggering a data-loss popup
As described in the previous chapter each UI building block can request a data-loss popup during the FPM event loop. For this simply return the pre-defined instance of the class CL\_FPM\_CONFIRMATION\_REQUEST:

```abap
METHOD needs_confirmation .
  IF ... .
    eo_confirmation_request = cl_fpm_confirmation_request=>go_data_loss.
  ENDIF.
ENDMETHOD.
```

The display of other confirmation popups can be achieved by creating an own instance of the class CL\_FPM\_CONFIRMATION\_REQUEST which e.g. contains an application specific text. For more sophisticated popups (e.g. containing own screens see chapter 4.1)

2.5 Initial Screen
The optional Initial Screen is composed of UI Building Blocks as every other screen within an OIF Subview or a GAF Roadmap main step or substep.

The configuration is done within the configuration of the Floorplan components FPM\_OIF\_COMPONENT respectively FPM\_GAF\_COMPONENT at the node INITIAL\_SCREEN.
You have to provide the UI Building Blocks as elements of node INITIAL_SCREEN_UIBB. If you don’t provide at least one UIBB here no initial screen is processed and the application starts immediately with the View Switch (OIF) or the 1st roadmap step (GAF).

Additionally you can configure the toolbar of the Initial Screen the same way as described in 2.3.1. However there is a Start-Button in the toolbar which is mandatory, and therefore can’t be configured. When this button is pressed the event IF_FPM_CONSTANTS=>GC_EVENT-LEAVE_INITIAL_SCREEN is raised and the initial screen is left.

In case you want to leave the initial screen out of your application you can do this by raising the LEAVE_INITIAL_SCREEN event within your application specific coding:

```plaintext
data: lo_fpm type ref to if_fpm.
lo_fpm = cl_fpm_factory=>get_instance( ).
lo_fpm->raise_event_by_id(
  IF_FPM_CONSTANTS=>GC_EVENT-LEAVE_INITIAL_SCREEN ).
```

## 2.6 Confirmation Screen

### 2.6.1 Confirmation Screen in OIF

In OIF the confirmation screen is only displayed when the object currently processed in the application has been deleted. After an object has been deleted the confirmation screen is displayed in place of the normal view switch. Therefore if your application doesn’t have such a delete-functionality you don’t need a confirmation screen.

A confirmation screen is configured the same way as any subview or initial screen by simply specifying the UIBBs which shall be displayed. This is done in the configuration of the floorplan component in the node CONF_STEP_VIEW (which is a child of node VARIANT). There is no toolbar configurable for the confirmation screen.
The confirmation screen in OIF can be accessed via standard button DELETE_OBJECT (see 2.3.1), which raises the Event IF_FPM_CONSTANTS=>GC_EVENT-DELETE_CURRENT_OBJECT or out of your application coding:

\[
\text{data: lo_fpm type ref to if_fpm.} \\
\text{lo_fpm = cl_fpm_factory=>get_instance( ).} \\
\text{lo_fpm->raise_event_by_id(} \\
\text{\quad IF_FPM_CONSTANTS=>GC_EVENT-DELETE_CURRENT_OBJECT \).}
\]

### 2.6.2 Confirmation Screen in GAF

In most cases GAF applications will have a last step to confirm the action that has been done, the so called confirmation step. The purpose of this step is to inform the user that the action he executed has been finished successfully. The configuration of the FPM_GAF_COMPONENT explicitly supports such a use case. If your application needs a confirmation step you should use the configuration scheme as depicted below:
2.7 Using FPM Message Manager

In order to guarantee a consistent and guideline compliant message handling FPM provides an own Message Manager which should be used instead of the generic WebDynpro Message Manager.

**How to use the message manager.**

1. Declare an attribute at the component globally (say MR_MESSAGE_MANAGER) referring to an object of type IF_FPM_MESSAGE_MANAGER

2. In the WDDOINIT method of your component controller get a handle to the FPM Message Manager which is a read only attribute in the IF_FPM interface.

   **Method** WDDOINIT.
   
   "Get the handle to the IF_FPM interface.

   ```
   wd_this->MR_FPM = CL_FPM_FACTORY=>GET_INSTANCE( ).
   wd_this->MR_MESSAGE_MANAGER
   = wd_this->MR_FPM->MO_MESSAGE_MANAGER.
   ```

   endmethod.

3. Use the message manager reference to report messages, raise exceptions or to clear messages. Some examples are provided below.

   **Example:** T100 based message. This example is taken for the demo applications and can be found in the WD component FPM_HELLOSFLIGHT_OIF_DEMO which lies in the APB_FPM_DEMO package.

   ```
   CALL METHOD WD_THIS->MR_MESSAGE_MANAGER->REPORT_T100_MESSAGE
   EXPORTING
   IV_MSGID = 'APB_FPM_DEMO'
   IV_MSGNO = 009
   IO_COMPONENT = wd_this
   IV_SEVERITY = if_fpm_message_manager=>GC_SEVERITY_ERROR
   IV_LIFETIME = if_fpm_message_manager=>GC_LIFE_VISIBILITY_AUTOMATIC
   IV_PARAMETER_1 = lv_carrid_string
   IO_ELEMENT = lo_el_sflight_selection
   IV_ATTRIBUTE_NAME = 'CARRID'.
   ```

The corresponding T100 message is as follows
The parameter &1 will get replaced with the actual flight name in the message. This is how it will get rendered in the message area.
3 Using Advanced FPM Features

3.1 Variants

In some cases the final setup of the OIF view switch or the GAF roadmap can only be decided at runtime. This is the case for example if you can select objects of different types on the initial screen and depending of the selected object’s type you want to provide a different view set respective roadmap to your users.

For this FPM offers the possibility to configure several variants. Each variant is a complete set of configuration data for the OIF view switch respective the GAF roadmap. Based on the input of the initial screen or some other startup information (like application parameters) you can then select one of these variants.

The variant selection has to be done programmatically – to do this you have to use an application specific configuration controller (AppCC) (see 0 for detailed explanation).

To do this you have to implement the interface IF_FPM_OIF_CONF_EXIT for OIF respective IF_FPM_GAF_CONF_EXIT for GAF in one of the application components or in a new component. This interface has only one method OVERRIDE_EVENT_OIF (respective OVERRIDE_EVENT_GAF) which passes a handler object of type IF_OIF (respective IF_GAF). This handler object provides the API for manipulating the floorplan configuration at runtime. One method of this object is SET_VARIANT, which you can use for selecting the variant.

For OIF:

```java
method OVERRIDE_EVENT_OIF .
...
    case io_oif->mo_event->MV_EVENT_ID.
        when if_fpm_constants=>gc_event-leave_initial_screen.
            io_oif->set_variant( <ID of variant> ).
...
```

For GAF:

```java
method OVERRIDE_EVENT_GAF .
...
    case io_gaf->mo_event->MV_EVENT_ID.
        when if_fpm_constants=>gc_event-leave_initial_screen.
            io_gaf->set_variant( <ID of variant> ).
...
```

In this code example the variant selection is done when the navigation from the initial screen to the main floorplan area is done. This is the latest point in time for selecting the variant. But selecting the variant before is also possible.

The last thing to do is to announce the AppCC to FPM. This has to be done in the floorplan component configuration in the node APP_SPECIFIC_CC. There you have to provide at least the name of the component you’re using as AppCC.
3.2 Subroadmaps

In some cases it might be necessary for GAF based applications to have subroadmaps. Subroadmaps may exist between two mainsteps if necessary. When the application is started they are not visible at all, but all mainsteps that are a possible starting point for subroadmaps have a special look. At runtime it depends on the application context and the user input and choice whether a subroadmap has to be traversed or not. So the application has to decide this. Therefore subroadmaps are statically declared but activated at runtime via the FPM API by the application.

The image beneath shows the configuration of a simple subroadmap (or substep variant) that contains one step with one UI Building Block inside.
In principle the configuration for subroadmaps is the same as for main steps. An arbitrary amount of subroadmap steps may be defined with an arbitrary amount of UI Building Blocks inside one subroadmap step.

You have to populate the input fields for the following nodes:

- **SUBSTEP**: Enter qualifiers for id and name. The entity name will be shown at runtime beneath the corresponding sub step.
- **SUBSTEP_UIBB**: Enter the values for Component and View as depicted above via the red arrows. In the field Component enter the name of the WD component. In the field Window put the name of the WD window of the interface-view (not the name of the WD view itself).

After the subroadmap has been configured statically, you may invoke it at runtime via the FPM API. Basically this is done by raising a special FPM event. Before raising it, the event parameters have to be populated with the corresponding subroadmap id you want to use. Here is the coding how this is achieved:

```plaintext
DATA: lo_fpm TYPE REF TO if_fpm,
     lr_event TYPE REF TO cl_fpm_event.

* get reference to FPM API
lo_fpm = cl_fpm_factory=>get_instance( ).
* create event
lr_event = cl_fpm_event=>create_by_id( 
   cl_fpm_event=>gc_event_change_step ).
* fill event parameters
lr_event->mo_event_data->set_value( 
   iv_key = cl_fpm_event=>gc_event_param_mainstep_id
   iv_value = <ID of Mainstep> ).
lr_event->mo_event_data->set_value( 
   iv_key = cl_fpm_event=>gc_event_param_substep_id
   iv_value = <ID of Substep> ).
lr_event->mo_event_data->set_value( 
   iv_key = cl_fpm_event=>gc_event_param_subvariant_id
   iv_value = <ID of Substep variant> ).
* now raise event
wd_this->fpm->raise_event( io_event = lr_event ).
```

### 3.3 Using Application Parameters

Application parameters are defined on Web Dynpro Application level:
You can add arbitrary parameters as “application-specific attributes” to your Web Dynpro application. During runtime, these parameters are exposed via IF_FPM->MO_APP_PARAMETER. MO_APP_PARAMETER stores an instance of IF_FPM_PARAMETER. With this interface you are able to retrieve the parameters.

Please note that there is no concept of mandatory or optional parameters available. For security reasons, you should never trust parameters passed by a different application. Please always do a proper validation before using application parameters.

3.4 Dynamic Adjustments

3.4.1 Adjusting the toolbar - using the CNR API

3.4.1.1 General information

During runtime the content and visibility of the OIF and GAF toolbar may be changed via the CNR API. You have to be aware that there exists different APIs for the OIF and for the GAF.

These APIs are accessed via the CL_FPM_SERVICE_MANAGER:

* for OIF:

\[
\begin{align*}
\text{DATA: } & \text{lo_cnr_oif TYPE REF TO if_fpm_cnr_oif,} \\
& \text{lo_fpm TYPE REF TO if_fpm.} \\
lo_fpm & = \text{cl_fpm_factory=>get_instance( ).} \\
lo_cnr_oif & = \text{lo_fpm->get_service( cl_fpm_service_manager=>gc_key_cnr_oif ).}
\end{align*}
\]

* for GAF:

\[
\begin{align*}
\text{DATA: } & \text{lo_cnr_gaf TYPE REF TO if_fpm_cnr_gaf,} \\
& \text{lo_fpm TYPE REF TO if_fpm.} \\
lo_fpm & = \text{cl_fpm_factory=>get_instance( ).} \\
lo_cnr_gaf & = \text{lo_fpm->get_service( cl_fpm_service_manager=>gc_key_cnr_gaf ).}
\end{align*}
\]

Basically the two APIs offer both the same methods. These are:

- **DEFINE_BUTTON**
  
  With this method either standard buttons or application-specific buttons can be created and changed. With the parameter IV_FUNCTION the kind of button is defined (see
IF_FPM_CONSTANTS=>gc_button). The ELEMENT_ID is necessary if application-specific buttons have to be changed later on.

- **DEFINE_BUTTON_CHOICE**
  With this method either standard button-choices or application-specific button-choices can be created and changed. With the parameter IV_FUNCTION the kind of button-choice is defined (see IF_FPM_CONSTANTS=>gc_button_choice). The ELEMENT_ID is necessary if application-specific buttons have to be changed later on.

- **CREATE_SEPARATOR**
  When a separator has to be created during runtime for the OTHER_FUNCTIONS area (application-specific), this method can be used.

- **SET_DUPLICATE_TOOLBAR**
  With this method the duplication of the toolbar can be activated or deactivated.

- **DEFINE_YOU_CAN_ALSO**
  This method can be used to change the menu bar for YOU_CAN_ALSO (see also chapter 3.8 ‘Navigation API’).

- **DEFINERELATED_LINKS**
  This method can be used to change the menu bar for RELATED_LINKS (see also chapter 3.8 ‘Navigation API’).

As the logical usage of these toolbars differs between the floorplans, in the following two subchapters the different parameters will be explained.

### 3.4.1.2 OIF specific parameters

As for every OIF variant a toolbar exists, only the variant_id has to be passed with every OIF CNR API method.

<table>
<thead>
<tr>
<th>Method parameters</th>
<th>DEFINE_YOU_CAN_ALSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
<td>Type</td>
</tr>
<tr>
<td>VARIANT_ID</td>
<td>Importin</td>
</tr>
<tr>
<td>IV_ROLE</td>
<td>Importin</td>
</tr>
<tr>
<td>IV_INSTANCE</td>
<td>Importin</td>
</tr>
<tr>
<td>IV_LABEL</td>
<td>Importin</td>
</tr>
<tr>
<td>IV_VISIBILITY</td>
<td>Importin</td>
</tr>
</tbody>
</table>
An example of method calls to change the CNR of the OIF at runtime would be:

```plaintext
DATA: lo_cnr_oif TYPE REF TO if_fpm_cnr_oif,
     lo_fpm TYPE REF TO if_fpm.

lo_fpm = cl_fpm_factory=>get_instance( ).
lo_cnr_oif = lo_fpm->get_service( cl_fpm_service_manager=>gc_key_cnr_oif ).

lo_cnr_oif->define_button(
    EXPORTING
    iv_variant_id = 'variant_1',
    iv_function  = < e.g. ACTIVATION FUNCTIONS (appl-specific buttons),
                    ALTERNATE FUNCTIONS (appl-specific buttons), CHECK,
                    DELETE OBJECT, EDIT, LOAD DRAFT, NEW NEXT OBJECT,
                    OTHER FUNCTIONS (appl-specific buttons), RELOAD OBJECT,
                    PRI NT, PRI NT PREVIEW, READ ONLY, REDO, REFRESH,
                    SAVE AS, SAVE DRAFT, SEND, START OVER, UNDO,
                    see also IF FPM CONSTANTS=>gc_button >
    iv_element_id = <optional; only if you want to change the button properties afterwards>
    iv_sequence_id = <optional; only if you use OTHER FUNCTIONS; determines the place where to insert this button>
    iv_design     = <optional; Button-Design>
    iv_enabled    = <optional; Button-Enabling>
    iv_explanation = <optional; Button-Explanation>
    iv_on_action  = <optional; determines the Event-Id for a button; not possible with standard buttons>
    iv_text       = <optional; Button-Label>
    iv_tool tip   = <optional; Button-Tooltip>
    iv_visibility = <optional; Button-Visibility> ).

lo_cnr_oif->define_button_choice(
    EXPORTING
    iv_variant_id = 'variant_1',
    iv_function  = < e.g. NEW OTHER FUNCTIONS (appl-specific button-choices), PRI NT, PRI NT PREVIEW SEND
                    see also IF FPM CONSTANTS=>gc_button_choice >
    iv_element_id = <optional; only if you want to change the button-choice properties afterwards>
    iv_sequence_id = <optional; only if you use OTHER FUNCTIONS; determines the place where to insert this button-choice>
    iv_enabled    = <optional; Button-Choice-Enabling>
    iv_text       = <optional; Button-Choice-Label>
    iv_tool tip   = <optional; Button-Choice-Tooltip>
    iv_visibility = <optional; Button-Visibility>
    it_menu_action_items = <menu elements of a Button-Choice > ).
```

### 3.4.1.3 GAF specific parameters

For the GAF CNR API methods more parameters have to be passed.

Always compulsory are the variant_id and the mainstep_id. This would be enough for defining UI-Elements on a mainstep. If UI-Elements on a substep should be changed dynamically, additionally the subvariant_id and the substep_id have to be passed.
An example of method calls to change the CNR of the GAF at runtime would be:

```
DATA: lo_cnr_gaf TYPE REF TO if_fpm_cnr_gaf,
     lo_fpm TYPE REF TO if_fpm

lo_fpm = cl_fpm_factory->get_instance( ).
lo_cnr_gaf = lo_fpm->get_service( cl_fpm_service_manager->gc_key_cnr_gaf ).

lo_cnr_gaf->define_button( EXPORTING
   iv_variant_id = 'variant_abc',
   iv_mainstep_id = 'mainstep_1',
   iv_subvariant_id = < optional; 'subvariant_xyz'>
   iv_substep_id = < optional; 'substep_99'>
   iv_function = < e.g. EXIT_TO, FINISH, OTHER_FUNCTIONS (appl-specific buttons), SAVE_DRAFT> see also IF_FPM_CONSTANTS->gc_button >
   iv_element_id = < optional; only if you want to change the properties of application specific buttons afterwards>
   iv_sequence_id = < optional; only if you use OTHER_FUNCTIONS; determines the place where to insert this button >
   iv_design = < optional; Button-Design >
   iv_enabled = < optional; Button-Enabling >
   iv_explanation = < optional; Button-Explanation >
   iv_on_action = < optional; determines the Event-Id for a button; not possible with standard buttons >
   iv_text = < optional; Button-Label >
   iv_tooltip = < optional; Button-Tooltip >
   iv_visibility = < optional; Button-Visibility >).

lo_cnr_gaf->define_button_choice( EXPORTING
   iv_variant_id = 'variant_1',
   iv_mainstep_id = 'mainstep_1',
   iv_subvariant_id = < optional; 'subvariant_xyz'>
   iv_substep_id = < optional; 'substep_99'>
   iv_function = < e.g. OTHER_FUNCTIONS (appl-specific button-choices)>
   iv_element_id = < optional; only if you want to change the button-choice properties afterwards>
   iv_sequence_id = < optional; only if you use OTHER_FUNCTIONS; determines the place where to insert this button-choice >
   iv_enabled = < optional; Button-Choice-Enabling >
   iv_text = < optional; Button-Choice-Label >
   iv_tooltip = < optional; Button-Choice-Tooltip >
   iv_visibility = < optional; Button-Visibility >
   it_menu_action_items = < menu elements of a Button-Choice >).
```
3.4.2 Adjusting the header - using the IDR API

During runtime the content and visibility of the header (IDR) may be changed via the IDR API. To do so, first of all one needs to get access to the IDR API as depicted below:

```
DATA: lo_idr TYPE REF TO if_fpm_idr,
      lo_fpm TYPE REF TO if_fpm.

lo_fpm = cl_fpm_factory=>get_instance( ).
lo_idr ?= lo_fpm->get_service( cl_fpm_service_manager=>gc_key_idr ).
```

The description of the IDR API is divided into the IDR’s sub-areas:

- **Header area:**
  
  `GET_APPLICATION_TITLE`
  Retrieves the title text and its corresponding tooltip.

  `SET_APPLICATION_TITLE`
  Puts a new title text and its corresponding tooltip into the header.

  `SET_HEADER_VISIBILITY`
  This method can be used to change the headers visibility.

- **Ticket area:**

  `GET_TICKET`
  Retrieves the texts of the ticket top and bottom and its corresponding tooltips.

  `SET_TICKET`
  Changes the texts of the ticket top, ticket bottom and its corresponding tooltips.

  `SET_TICKET_VISIBILITY`
  Alters the visibility of the ticket area.

- **Items area:**

  `ADD_ITEM_GROUP_BY_VAL`
  Use this method to add a new item group to the item area. One item consists of a label, its tooltip, a value and the value's tooltip. A group of items consists of an arbitrary amount of such items. By using this method the items are added as simple static text strings to the FPM. Therefore if a value of an item needs to be changed on a later point of time, you will need to call method `change_item_group_by_val`. There exists also the method `add_item_group_by_ref` which can be used to pass WD context nodes to the IDR. In that case the value changes automatically, when the value of the corresponding attribute in the context node changes.

  `CHANGE_ITEM_GROUP_BY_VAL`
  This method changes the label and values that were passed to the IDR via the method `add_item_group_by_val`.

  `ADD_ITEM_GROUP_BY_REF`
  Similar than `add_item_group_by_val`, this method is used to add label/value items to the IDR. In this case the value is not passed as a static text but as reference to a WD context node attribute. The advantage here is first, that the value can be of type other than string. Secondly updating the value happens automatically whenever the attribute of the context node changes, the IDR changes the visible
value. Moreover it is possible to tell the IDR to show also the unit of the value. This is just done by a flag; the actual unit is taken from the DDIC information of the value's type. Therefore this feature will work only, if the type of the attribute in the context node that is passed to the IDR has a defined DDIC unit.

CONTAINS_ITEM_GROUP
Checks whether a certain item group exists within the IDR (it does not matter whether the group has been added via add_item_group_by_val or add_item_group_by_ref).

REMOVE_ITEM_GROUP
Removes a certain item group (it does not matter whether the group has been added via add_item_group_by_val or add_item_group_by_ref).

INITIALIZE_ITEMS
This method is used to clear all items.

SET_ITEMS_VISIBILITY
This method is used to change the visibility of the item area (i.e. the visibility status of all items, not single items).

3.5 Sharing Data across different UIBBs

When the UI building blocks of an application are implemented in several components there is often the need for sharing data across these components. Technically this can be achieved with several approaches as described in the following chapters.

For this purpose, the FPM offers so called “shared data” components. This is an optional FPM feature which should suit most of the application demands. However, if needed it can be replaces by other technical alternatives as described in chapter 3.5.2.

3.5.1 Using a Shared Data component

A shared data component is a ‘normal’ Web Dynpro component which implements the IF_FPM_SHARED_DATA interface. This interface contains no methods or attributes but serves as a marker interface only.

Each component (e.g. UI building block, FPM_OIF|GAF_CONF_EXIT-component) which wants to use such a shared data component only needs to declare a usage to the shared data component. For this, the technical type of the usage does not need to refer to IF_FPM_SHARED_DATA (by this it would not have any accessible methods/attributes) but links to the actual component itself.

The lifecycle handling is now handled completely by the FPM: Whenever a component is instantiated by the FPM (e.g. a UIBB which is configured for a given screen), the FPM analyzes all usages of that component. In case it detects a usage pointing to a component which implements the IF_FPM_SHARED_DATA interface a singleton of this shared data component is automatically attached to the usage.

This leads to the following steps an application has to follow to share data using a shared data:

- Create a component which implements the IF_FPM_SHARED_DATA interface.
- This component now typically contains methods to retrieve data from the business logic and exposes the extracted UI data via it’s Web Dynpro context or interface methods.
- Each component accessing these shared data defines a usage to the shared data component. This usage is automatically instantiated by the FPM.
- The consuming component can now communicate with the shared data component via Web Dynpro context mapping, attribute access or method calls.
3.5.2 Other options

Besides the optional FPM offering of the “shared data” concept, sharing data between several Web Dynpro components can be achieved by other approaches as well. Reasons for not using a Shared Data component could be:

- There is already an application specific API available which can serve as a ‘data container’ and can be accessed by several components.
- The data need not only be shared between Web Dynpro components but between other entities like ABAP OO classes, function groups, etc. as well.
- The amount of data to be shared is so large that putting it into a Web Dynpro context would lead to performance and memory consumption issues.

In these cases the application can consider using techniques like

- An ABAP OO class which is accessible as a singleton, so that all consumers share the same instance
- A function group with appropriate function modules.

3.6 Using an application specific Configuration Controller

Using an application specific Configuration Controller (AppCC) allows to

- Make global checks (Checks affecting more than one UIBB)
- Make global adjustments for FPM events
- Read the structure of your application at runtime
- Change the structure of your application dynamically

This is the place where all the things affecting more than one single UIBB can be performed. The use is optional, so you need to provide an AppCC only if you need one of the features AppCC offers.

In order to provide an AppCC you have to implement the interface IF_FPM_OIF_CONF_EXIT (respective IF_FPM_GAF_CONF_EXIT for GAF) in a Web Dynpro Component. This Web Dynpro-Component is either one of the components already used within your application or a completely new one.

You have to announce the AppCC component to FPM by entering its name (and optional a configuration key) in the floorplan components configuration in the node APP_SPECIFIC_CC.

If your AppCC has declared a static usage to a component implementing IF_FPM_SHARED_DATA this shared data component is instantiated and attached automatically by the FPM framework, assuring that all components within your application accessing this shared data component will see the same instance of it.

The AppCC Interface contains only one method OVERRIDE_EVENT_OIF (respective OVERRIDE_EVENT_GAF) which passes an object of type IF_FPM_OIF (respective IF_FPM_GAF) which serves as API for the applications.

The method OVERRIDE_EVENT_*-Method is called at the beginning of the event processing, right after FLUSH has been called on all visible UIBBs.
3.6.1 Event Cancellation

AppCC is the only place where you can perform global checks which are relevant for more than one UIBB. For checking the event it is available as an attribute of IF_FPM_OIF (respectively IF_FPM_GAF). If it’s necessary to cancel the event out of AppCC simply call the method CANCEL_EVENT.

3.6.2 Selecting a Variant

With method IO_FPM_OIF->SET_VARIANT you can select the variant which shall be used in case that there is more than one variant configured.

3.6.3 Adjustment of Events

IF_FPM_OIF (respectively IF_FPM_GAF) provides the currently processed FPM event as changeable attribute. Therefore it’s possible to change the event (add/remove/change event parameters even replace the event). As AppCC is called right at the beginning of the event loop, changing the event has the same result as if instead of the original event the changed event has been raised.

3.6.4 Read the configuration at runtime

The AppCC-API offers several methods for applications who want to read the configuration data at runtime:

- GET_CURRENT_STATE returns the current navigation state within the application (which step, view, … is currently displayed, which variant is used, …).
- GET_VARIANTS returns a list of all available variants.
- GET_UIBB_KEYS returns a list of all UIBBs assigned to a specified step, substep or subview.
- GET_UIBB_KEYS_FOR_CONF_STEP and GET_UIBB_KEYS_FOR_INIT_SCREEN return lists of UIBBs assigned to the initial screen and the confirmation screen.

Only for OIF:
- GET_MAINVIEWS returns a list of all views for a specified variant.
- GET_SUBVIEWS returns a list of all subviews for a given mainview.

Only for GAF:
- GET_MAINSTEPS returns a list of all main steps within a given variant.
- GET_SUBSTEP_VARIANTS returns a list of all subroadmaps for a given main step.
- GET_SUBSTEPS returns a list of all substeps for a subroadmap.

3.6.5 Change the configuration at runtime

Changing the configuration at runtime is also supported by AppCC-API:

- ADD_UIBB offers the possibility to dynamically add another UIBB to a subview, step or substep.
- REMOVE_UIBB offers the possibility to remove a UIBB from a subview, step or substep.

Only for OIF:
- ADD_MAINVIEW allows to dynamically adding a new view at runtime.
- REMOVE_MAINVIEW allows to dynamically deleting a view at runtime.
- ADD_SUBVIEW allows to dynamically adding a new subview.
- REMOVE_SUBVIEW allows removing a subview.
- RENAME_MAINVIEW rename of a mainview.
- RENAME_SUBVIEW rename of a subview.
• **SET_SELECTED_SUBVIEW** allows changing the target subview within a mainview. This method is only for the special case, while making a mainview switch that another subview than the first is displayed.

Only for GAF:
• **RENAME_MAINSTEP** rename a mainstep dynamically at runtime.
• **RENAME_SUBSTEP** rename a substep dynamically at runtime.

### 3.7 Handling of Transactions

Transactions can be handled in FPM by implementing the WD interface IF_FPM_TRANSACTION. This is an optional interface, meaning that not all applications will need to implement this interface unless they choose to handle transactions in a systematic manner as defined by FPM. An application can choose to handle the transactions all by themselves without having the need to implement the interface but by using this interface the following can be guaranteed:
1. There is a logical sequence of which the interface methods are called.
2. The transaction steps can be split up into the sequence in which they are supposed to be processed.
3. There is a check – save – validate sequence that provides high transaction integrity.
4. The check – save – fail – recover sequence provides the required robustness to the transaction.

The following are the interface methods of the WD interface

```
CHECK_BEFORE_SAVE
SAVE
AFTER_COMMIT
AFTER_REJECTED_SAVE
```

The methods are called in the sequence depicted beneath:
Please consult the FPM design document for the detailed sequence diagram of the method calls.

*How to use the transaction interface*
1. Select a component that will contain the business logic to be executed on a save event. This could be any component known to the FPM – this means any UIBB or any Shared Data component used by the UIBBs of your application.

2. In the component view in SE80 you should add the interface IF_FPM_TRANSACTION as an implemented interface.

   (In the below example we have used the main controller to implement the interface)

   ![Implemented Interfaces](image1)

   In edit mode add the IF_FPM_TRANSACTION interface.

   ![Implemented Interfaces](image2)

   Click the re-implement button to get the interface methods to reflect in your component controller.

   ![Implemented Interfaces](image3)

   You will see the following methods in the component controller

   ![Component Controller](image4)

3. On a save event you will always need the data to be saved. This can be realized in many ways:
   a. Use a shared data component (see section on shared data for further details).
b. Use direct context binding.
c. Use an assistance class.

The decision to use any or a combination of the above methods is done by the application developer.

The following is the use case for each of the methods:

CHECK BEFORE SAVE
This method has a returning parameter which indicates if the validation before a save to the database will be successful. Use this method as the trial for save and then return a true if the trial save was successful and false otherwise.

SAVE
This method is where you can do the actual save and any possible commit. This is called in case the check_before_save in case the former returns false (please note that the semantic of the return parameter of the check_before_save is negative it reads 'rejected' hence a false value for rejected means that the check_before_save is successful) i.e. if rejected = false (meaning the trial was successful). In case of errors while saving you should return rejected = true so that the AFTER_REJECTED_SAVE can be called. If the save was indeed successful then the method AFTER_COMMIT is called. Please refer to the flow chart of a better understanding.

AFTER_COMMIT
You can perform clean up activities like releasing database locks, releasing other resources, trigger an event for processing after successful commit.

AFTER_REJECTED_SAVE
Typically here you can perform your roll back activities. Also locks and resources can be released.

Note:
Once the interface is implemented the FPM takes care of identifying the corresponding component that has the method to be called in sequence and takes care of calling the methods on this component in the same sequence as defined above.

FAQ on transaction
1. On what event will these methods be called?
   → These methods will be invoked on the standard FPM SAVE event.

2. Can I make the FPM to call these methods to my own custom event?
   → No. These methods are called as part of the standard FPM event loop and hence will not react on custom events.

3. Can I have multiple components implementing this interface in the same application?
   → Yes, you can. The FPM will call all the methods on all the implementing components. But our general recommendation is to use only one central component for transaction handling.
3.8 Navigation API

To navigate to a specific target out of your application you can use the ‘You can also’ and ‘Related links’ functionality.

The usage and configuration of ‘You can also’ and ‘Related Links’ is identical. So only the ‘You can also’ functionality is described. You can find this functionality in the toolbar at the right hand side.

At first you must create a launchpad. To do that call transaction LPD_CUST. In the first screen press the button ‘New Launchpad’. At the popup enter a name for the role and instance and a description. In the following screen you can enter some navigation targets. Press the button ‘New Application’, enter a text and choose an ‘Application Type’. Depends on this application type you must enter further information like URL or transaction code.
For further information concerning the Launchpad use the following link:
\Dwdf051\erp_all\26_ERP_Architecture\PUBLIC\guidelines\Generic\RLP\WD_abap_launchpad.ppt

To activate the ‘You can also’ functionality you must go to the component configuration of your application, navigate to the toolbar node and choose ‘YOU_CAN_ALSO’. On the right hand side you must set the visibility to ‘visible’ and enter the role and instance of your launchpad. After saving of the changes the ‘You can also’ link will be displayed in your application.
3.8.1 How to access the Navigation API

To get access to the Navigation API you can use the interface IF_FPM. It provides the method

GET_NAVIGATION()

It returns an instance of the Navigation API IF_FPM_NAVIGATION

3.8.2 Interface IF_FPM_NAVIGATION (Runtime class CL_FPM_NAVIGATION)

The Navigation interface provides a list with all customized applications of a given Launchpad.

<table>
<thead>
<tr>
<th>Table: MT_TARGETS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters:</td>
</tr>
<tr>
<td>entry_type</td>
</tr>
<tr>
<td>parent</td>
</tr>
</tbody>
</table>
folder or initial

<table>
<thead>
<tr>
<th>key</th>
<th>Type</th>
<th>string</th>
<th>Guid of an application</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>Type</td>
<td>string</td>
<td>A unique identifier for an application. It is defined in the customizing of the Launchpad</td>
</tr>
<tr>
<td>text</td>
<td>Type</td>
<td>text255</td>
<td>Text of the link</td>
</tr>
<tr>
<td>description</td>
<td>Type</td>
<td>string</td>
<td>Description</td>
</tr>
<tr>
<td>icon_path</td>
<td>Type</td>
<td>string</td>
<td>Path to an icon</td>
</tr>
<tr>
<td>enable</td>
<td>Type</td>
<td>boole_d</td>
<td>Determine if an application is active/enabled or inactive/disabled</td>
</tr>
<tr>
<td>visible</td>
<td>Type</td>
<td>boole_d</td>
<td>Determine the visibility of an application</td>
</tr>
</tbody>
</table>

Domains:

**Domain: FPM_NAVIGATION_TARGET_TYPE**

<table>
<thead>
<tr>
<th>ID:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP</td>
<td>Line contains an Application</td>
</tr>
<tr>
<td>FOL</td>
<td>Line contains a folder</td>
</tr>
<tr>
<td>SEP</td>
<td>Line contains a separator</td>
</tr>
</tbody>
</table>

The Navigation interface provides the following methods:

**Method: NAVIGATE**

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>Direction</th>
<th>Type kind</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV_TARGET_KEY</td>
<td>importing</td>
<td>Type</td>
<td>string</td>
<td>Guid of an Application</td>
</tr>
</tbody>
</table>

This method is used to start the navigation of an application.

**Method: MODIFY**

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>Direction</th>
<th>Type kind</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV_VISIBLE</td>
<td>importing</td>
<td>Type</td>
<td>BOOLE_D</td>
<td>Set an application visible/invisible</td>
</tr>
<tr>
<td>IV_ENABLE</td>
<td>importing</td>
<td>Type</td>
<td>BOOLE_D</td>
<td>Enable/disable an application</td>
</tr>
<tr>
<td>IV_TEXT</td>
<td>importing</td>
<td>Type</td>
<td>string</td>
<td>An alternative text for the application</td>
</tr>
</tbody>
</table>
The method MODIFY is used to change some attributes of an application. You can change the visibility of an application, you can enable/disable an application, and change the description and text.

### Method: SET_FILTER

<table>
<thead>
<tr>
<th>Parameters:</th>
<th>Direction</th>
<th>Type kind</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT_Filter</td>
<td>importing</td>
<td>Type</td>
<td>T_Filter</td>
<td>Guids of application that shouldn't be provided in the list of application</td>
</tr>
</tbody>
</table>

The SET_FILTER method provides the functionality to remove some applications from the list of applications that the Launchpad provides.

### 3.8.3 Integration: Navigation in the Eventloop

If you call the method IF_FPM_NAVIGATION~NAVIGATE a new event object of type cl_fpm_navigation_event will be created. This event object contains all application parameters.

In the interface IF_FPM_UI_BUILDING_BLOCK there is a method process_event in that you can get the navigation event again to change any parameters:

In this method you need coding like this:

At first check if the event a navigation event
check io_event->MV_EVENT_ID = io_event->gc_event_navigate.

and make a cast from the event object to the cl_fpm_navigation_event object
DATA lr_event type ref to cl_fpm_navigation_event.
lr_event ?= io_event.

Get the business parameter
lr_bus_parameter ?= lr_event->mo_event_data.

Get the launcher parameter
lr_launcher_parameter ?= lr_event->mo_launcher_data.

With the method lr_parameter->TO_LPPARAM( ) you will get a internal table with the parameters.
If you want to change any parameter you can use the methods of lr_parameters, like get_value, set_value or delete_value.

If the event processing requires further user interaction (e.g. ask for further data in a popup) the event processing can be deferred by returning EV_RETURN = IF_FPM_CONSTANTS~GC_EVENT_RESULT-DEFER. If the result of the event processing is ok you can return EV_RETURN = IF_FPM_CONSTANTS~GC_EVENT_RESULT-OK otherwise EV_RETURN = IF_FPM_CONSTANTS~GC_EVENT_RESULT-FAILED.

To prevent a lost of data you can implement the method NEEDS_CONFIRMATION that is located in the interface IF_FPM_UI_BUILDING_BLOCK.

In this method you get also the navigation event and you can decide to raise a ‘Data Loss’ Popup. If you want to do that you must return the following value:

\[ \text{eo_confirmation_request} = \text{cl_fpm_confirmation_request} \Rightarrow \text{go_data_loss}. \]

For further information have also a look at the Powerpoint presentation of Holger Bohle:
3.9 Composite UIBBs

A composite UIBB is a UIBB that has one or more other child-UIBBs inside its body arranged in a certain layout. A composite UIBB implements the WD interfaces `IF_FPM_UI_BUILDING_BLOCK` and `IF_FPM_COMPOSITE_COMP`. There is currently one instance of a composite UIBB available: A master/detail or tabbed UIBB.

3.9.1 Master-detail or tabbed UIBB

3.9.1.1 General

The tabbed UIBB is an instance of a composite UIBB. That means it has several other child-UIBBs inside its body arranged in a special layout. In this case the child UIBBs are arranged in a master-detail layout. A typical tabbed UIBB may look like depicted below:

![Test the fpm tabbed component](image)

The tabbed UIBB is implemented by the WD component `FPM_TABBED_UIBB`. The tabstrip comes from the component `FPM_TABBED_UIBB`, whereas the UIBBs depicted in the areas master and detail come from the application. The component `FPM_TABBED_UIBB` has its own component configuration, which looks like depicted below:
The nodes have the following meaning:

- **MASTER_UIBB**: Contains the child-UIBBs that are showed in the master area.
- **VARIANT**: Contains different configurations variants for the detail area.
- **TAB**: Contains the tabs that the tabstrib is made up of.
- **UIBB**: Contains child-UIBBs for the tabs.
- **APP_SPECIFIC_CC**: Optionally filled with an entry for a component that implements the WD interface IF_FPM_TABBED_CONF_EXIT. This WD interface contains the method OVERRIDE_CONFIG_TABBED. This method is called during the FPM eventloop and gives the possibility to alter the configuration, i.e. to add/remove tabs, add/remove uibbs etc. There is no possibility to change the event itself. For more information on this see chapter 3.9.3.
- **PROPERTIES**: This node gives the possibility to change the layout. In the picture showed above the master UIBBs are on top and the detail UIBBs are at the bottom. It is also possible to change the layout and to have the master UIBBs on the left side and to have the detail UIBBs on the right side.

### 3.9.1.2 Creating an application that contains a master-detail composite UIBB

Currently the master-detail composite UIBB can only be realized by using the OIF floorplan. If you want to realize an application where only the master-detail UIBB is visible and not the view switch of the OIF you will have to configure only one main switch with only one subview switch (by doing so the view switch will get invisible). Now to realize an application the contains a master-detail composite UIBB you will have to do the following steps:

- Create an OIF floorplan application. Refer to chapter 2.2.1. (“Creating a simple OIF application”) for this task.
- Create a new configuration for WD component FPM_TABBED_UIBB. Refer to chapter 3.9.1.3 for more details about this configuration.
- Now go back to your configuration of the OIF application. In the field for the subview put an entry for an UIBB and populate it with: Component: FPM_TABBED_UIBB, interface-view: TABBED_WINDOW and then the name of the configuration you created for your master-detail composite UIBB like depicted below:
That is all. Now start your application and test it.

3.9.1.3 Changing the master-detail composite UIBB dynamically during runtime

You may rename tabs, add/remove tabs or child-UIBBs from your master-detail composite UIBB during runtime. To do this you will have to implement the following steps:

- Choose one of your own application WD components and let it implement the WD interface IF_FPM_TABBED_CONF_EXIT. Usually this is one of the WD components that provide a child UIBB.

- Now somewhere in your coding you want to e. g. rename a tab. There you have to raise your own FPM event, e. g. "CHANGE_TAB_NAME", like this:

```plaintext
DATA: lo_fpm TYPE REF TO if_fpm,
      lo_event TYPE REF TO cl_fpm_event.
lo_fpm = cl_fpm=>get_instance( ).
lo_event = cl_fpm_event=>create_by_id( 'CHANGE_TAB_NAME' ).
lo_event->mo_event_data->set_value( iv_key   = 'ID'
                                     iv_value = lv_tab_id ).
lo_event->mo_event_data->set_value( iv_key   = 'NAME'
                                     iv_value = lv_tab_name ).
lo_fpm->raise_event( io_event = lo_event ).
```
Now in the component controller implement the method OVERRIDE_CONFIG_TABBED. This could look like the following, if you want to rename a tab:

```abap
CASE io_tabbed->mo_event->mv_event_id.
  WHEN "CHANGE_TAB_NAME".
    DATA lv_name TYPE string.
    DATA lv_id TYPE string.
    io_tabbed->mo_event->mo_event_data->get_value( EXPORTING iv_key   = 'ID'
      IMPORTING ev_value = lv_id
    ).
    io_tabbed->mo_event->mo_event_data->get_value( EXPORTING iv_key   = 'NAME'
      IMPORTING ev_value = lv_name
    ).
    io_tabbed->rename_tab( iv_tab_id   = lv_id
      iv_new_name = lv_name ).
```

That is all. For other available functionality other than renaming a tab, please have a look at the interface IF_FPM_TABBED.

4 FPM for Experts

4.1 Handling of application specific Popups

In order to process a given event (in method IF_FPM_UI_BUILDING_BLOCK~PROCESS_EVENT; see chapter 2.4.3.6.) it can be necessary to gather additional information from the end-user by means of a popup. This can be as simple as a standard popup with some text and a set of buttons or it can contain more complex screens with input fields, checkboxes, etc. as well.

With normal WD programming this can be very cumbersome to achieve since WD popups cannot be processed in a synchronous ways (i.e. trigger the popup, wait for it to be closed and continue processing). That means UI building block would need to return the result of the event processing (OF or FAILED) before the popup can be processed.

In order to achieve this synchronous popup handling, the FPM offers the possibility to defer the processing of the event loop and resume it after the popup is being processed:

- **Defer the current event processing**
  This is done in method process_event:
  ```abap
  ev_result = if_fpm_constants=>gc_event_result-defer.
  ```

- **Register a popup**
  This is purely WD ABAP and not a feature of the FPM. Therefore you should look up the WD ABAP documentation regarding WD ABAP popups in general. Nevertheless here comes a short description on how this could be done. First of all the registration of the popup needs to be triggered also in the method process_event because this will be the last method until to program control goes back to the FPM. However, for an application specific popup you need your own WD ABAP View and the registration of the popup is only possible from within this View. Therefore in method process_event you need to call a method of the View that is supposed to be used for the application specific popup. Since view methods cannot be accessed from within methods of the component controller the WD ABAP eventing mechanism needs to be used: raise an event in the method process_event and register an event handler on the corresponding view. Here is a detailed list:

  a) Create a new WD ABAP View, e. g. "POPUP_CARRIER".
  b) In the Component Controller create a new WD ABAP Event, e. g. "REGISTER_POPUP".
  c) In the method process_event raise the WD ABAP Event "REGISTER_POPUP_EVNET".
  d) In the View POPUP_CARRIER create a new method e. g. "REGISTER_POPUP" of type event handler for the event "REGISTER_POPUP_EVNET".
  e) In the method REGISTER_POPUP use the ABAP Window API in order to create a popup, register action handler methods to the buttons of the popup and register the popup for opening. Here is a
coding fragment how this could look like (this example coding uses a standard popup with the two buttons “yes” and “no”):

```abap
DATA: lo_api              TYPE REF TO if_wd_component,
     lo_window_manager   TYPE REF TO if_wd_window_manager,
     lo_view_api         TYPE REF TO if_wd_view_controller,
     lo_popup            TYPE REF TO if_wd_window,
     lo_api            = wd_comp_controller->wd_get_api( ).
     lo_window_manager   = lo_api->get_window_manager( ).
     lo_view_api       = wd_this->wd_get_api( ).
     lo_popup = lo_window_manager->create_popup_to_confirm(  
                      text            = 'some popup text... ',
                      button_kind     = if_wd_window=>co_buttons_yesno  
                      message_type    = if_wd_window=>co_msg_type_question
                      window_title    = 'some popup title...'
                      window_position = if_wd_window=>co_center ).
     CALL METHOD lo_popup->subscribe_to_button_event
    EXPORTING
         button      = if_wd_window=>co_button_yes
         action_name = 'ONRESUME_EVT_OK'
         action_view = lo_view_api.
     CALL METHOD lo_popup->subscribe_to_button_event
    EXPORTING
         button      = if_wd_window=>co_button_no
         action_name = 'ONRESUME_EVT_FAILED'
         action_view = lo_view_api.
     lo_popup->open( ).
```

g) Create WD ABAP actions and handler methods for the actions that come from the popup, in this case from the yes- and the no-button. In the example above the names are ONRESUME_EVT_OK and ONRESUME_EVT_FAILED.

- **Resume the event**
  After the necessary user input has been gathered, the once frozen FPM event may now be continued, either having the result OK or FAILED. To do so you simply have to call the FPM method resume_event_processing within the action handler methods for the buttons of the popup. This could look like this:

```abap
DATA lo_fpm TYPE REF TO if_fpm.
lo_fpm = cl_fpm_factory=>get_instance( ).
lo_fpm->resume_event_processing( if_fpm_constants=>gc_event_result-ok ).
```

After the event is resumed the remaining UI building block will be processed (in case there are more than one).

The following diagram summaries the described behavior:
4.2 Skipping the initial screen

A typical OIF application starts with an initial screen where you select the object you intend to work with. Now if the object is already determined because you’re calling the application parameterized you don’t want to see the now obsolete initial screen (Of course there are also similar use cases for GAF). So the question is, how to skip an initial screen at runtime.

To do this you have to launch the FPM event LEAVE_INITIAL_SCREEN by yourself. You can do this in the OVERRIDE_EVENT_*-Method of your AppCC or in the PROCESS_BEFORE_OUTPUT Method of one of your initial screen UIBBs in case you don’t use an AppCC:

```plaintext
data: lo_fpm type ref to if_fpm,
    lv_object_id type string.

* Check event id
if lv_event_id = if_fpm_constants=>gc_event_start.
* Determine if Parameter OBJECT_ID is provided
  lo_fpm = cl_fpm_factory=>get_instance( ).
  lo_fpm->mo_app_parameter->get_value( )
```
exporting iv_key = 'OBJECT_ID'
importing ev_value = lv_object_id).

* In case OBJECT_ID is set navigate directly to the main floorplan * area
  if not lv_object_id is initial.
    lo_fpm->raise_event_by_id(
      if_fpm_constants=>gc_event-leave_initial_screen ).
  endif.
endif.

4.3 Work-Protect Mode

The FPM allows the application to make use of the “work-protect mode” offered by the Portal and the NWBC (i.e. showing a data-loss popup when the application is closed but contains unsaved data).

In order to achieve this behaviour the application needs to ‘tell’ the FPM whether it contains unsaved (i.e. ‘dirty’) data or not. For this the FPM provides the WD-Interface IF_FPM_WORK_PROTECTION with the one and only method IS_DIRTY:

```
@interface
IF_FPM_WORK_PROTECTION

IS_DIRTY : bool
```

This interface can be implemented by any application (WD-) component which is known to the FPM (e.g. any UI building block or a shared-data component). At runtime the FPM will detect all components implementing
this interface. If any of these components signals unsaved data then the application is marked as ‘dirty’. This application dirty-state is then passed on by the FPM to the shell (i.e. the portal or the NWBC).

Important:

Since the shell-API requires to get this information as soon as the application state changes, the IF_FPM_WORK_PROTECTION-IS_DIRTY method is called by the FPM runtime during each roundtrip. Therefore it needs to perform very fast.

A typical implementation might use this skeleton:

<table>
<thead>
<tr>
<th>Component Controller</th>
<th>COMPONENT CONTROLLER</th>
<th>Inactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties</td>
<td>Context</td>
<td>Attributes</td>
</tr>
</tbody>
</table>

Method: IF_FPM_WORK_PROTECTION

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF_FPM_WORK_PROTECTION</td>
<td>Comp. Interface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Ref</th>
<th>Opt</th>
<th>Associated Type</th>
<th>Short Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EV_DIRTY</td>
<td>Type</td>
<td>Data</td>
<td>Data</td>
<td>Data</td>
<td>Data</td>
<td>Data</td>
</tr>
</tbody>
</table>

```
1. METHOD is_dirty.
2. IF "component contains unsaved data.
3.   ev_dirty = abap_true.
4.   ELSE:
5.     ev_dirty = abap_false.
6.   END_IF.
7. END_METHOD.
```

### 4.4 Application specific Commit

(Not yet finished)

### 4.5 Embedding an FPM

FPM was designed for building standalone applications. It’s however possible with some restrictions to embed an FPM “application” within another application.

To do this, create a usage for component FPM_OIF_COMPONENT for OIF (FPM_GAF_COMPONENT for GAF) within the embedding component.

Then embed the Interface View FPM_WINDOW within one of your views.

Finally you have to create the used component manually (as you have to provide the configuration key of the floorplan component). This is done best as soon as possible. In most cases this is the WDDOINIT method of the embedding component controller:

```abap
method WDDOINIT.

* delete FPM component if already existing
  data: lo_usage type ref to if_wd_component_usage,
    ls_conf_key type wdy_config_key.
  lo_usage = wd_this->wd_cpuse_fpm_usage( ).
  if lo_usage->has_active_component( ) = abap_true.
    lo_usage->delete_component( ).
  endif.
  ls_conf_key->config_id = <ID configuration of FPM component>.

* recreate component using new configuration ID
  try.
    call method lo_usage->create_component
      EXPORTING
        component_name = 'FPM_OIF_COMPONENT'
  catch...
```

configuration_id = ls_conf_key.
catch cx_wd_runtime_api .
endtry.

Some remarks to the code example:
- The configuration you have to pass is the configuration key of component FPM_OIF_COMPONENT. You can't pass the application’s configuration key.
- The code example assumes that the usage is named 'FPM_USAGE'. If you give it another name you have to adjust the line lo_usage = wd_this->wd_cpuse_fpm_usage( ).
- The example is for OIF, for GAF you simply have to replace ‘FPM_OIF_COMPONENT’ by ‘FPM_GAF_COMPONENT’.
- The call of delete_component() is not necessary for simple static embedding, however in case you want to change the embedded FPM application later on you need it.

There are some limitations using this technique:
- FPM allows only one instance running at the same time within one internal mode. Therefore you can’t embed more than one ‘FPM application’ at the same time. It’s possible to switch the embedded ‘FPM application’, replacing one ‘FPM application’ by another. You can assure this if you only use one Usage to an FPM component within your application. This forces you to delete the old FPM component before creating a new one.
- You can’t pass any configuration key for the IDR (header area). Therefore the header is displayed without any configuration settings. This means you have to do all header settings programmatically at runtime.
- You can’t use application parameters for the FPM as the application is now unknown to FPM.

4.6 The Resource Manager Interface

{Not yet finished}

4.7 Detailed FPM Message Manager Documentation

The FPM Message Manager is an integral part of FPM and is available to all applications that use the standard floorplans like OIF, GAF etc. The FPM message management consists of the following parts
1. The Message Area
2. The Message Manager Interface

1. Message Area
   The floorplans like GAF and OIF have the message included as a standard UI element and hence it is available to all the applications based on the floorplans provided by FPM.
2. Message Manager Interface
   The Message Manager Interface is available to all applications to report messages into FPM. This interface offers methods to perform the following tasks:
   - Report messages
   - Raise Exceptions
   - Clear messages

API Method documentation:
Public methods
REPORT_MESSAGE
REPORT_T100_MESSAGE
REPORT_BAPIRET2_MESSAGE
RAISE_EXCEPTION
RAISE_T100_EXCEPTION
CLEAR_MESSAGES

Description of the parameters:

**The Report_XX methods**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Relevant methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO_COMPONENT</td>
<td>All</td>
<td>Pass an object reference to the message manager. This object will be used as the key to store the message. Preferably the WD component which is raising the message should be passed here. But only in case of exceptions where the object raising the message does not have a handle to the wd component (like for instance an ABAP OO class) you can pass any other object reference. This is important for those messages whose lifetime is maintained manually by the applications (see lifetime for more details). When you create a message whose lifetime is marked as manual the application creating such a message has then to take care or to delete the message once it is no longer needed. In that case you will need to pass the component whose messages have to be cleared. This helps to prevent messages from a different component being cleared by a component that has not raised it, which can be the case when you are re-using components from different areas.</td>
</tr>
<tr>
<td>IV_SEVERITY</td>
<td>All</td>
<td>The severity of the message to be reported. There are three values 1. Error (E) 2. Warning (W) 3. Success (I) (same as information)  The effect of the severity of messages on the navigation within the floorplan can vary with the floorplan, for instance the navigation can be different if there is an error message in GAF or OIF.  The three values that can passed are 1. GC_SEVERITY_ERROR for Error 2. GC_SEVERITY_WARNING for Warning 3. GC_SEVERITY_SUCCESS for success This is an optional parameter and the default is Error.</td>
</tr>
<tr>
<td>IV_LIFETIME</td>
<td>All</td>
<td>This is a very important parameter and should be given special attention. This parameter determines when, where and how long a message is displayed. This parameter is a combination of the following two dimensions 1. Lifetime 2. Visibility  Lifetime – This determines how long the message will exist in the message area, i.e. the creation and deletion of the message. The possible lifetimes are ‘automatic’ meaning that FPM will handle the destruction of the message as defined by the UI guidelines for the floorplan and ‘manual’ meaning that the application developer has to handle the deletion or clearing of the message from the message area.  Visibility – This will determine when the message will be shown on the message area. This can take the following values: ‘Automatic’ meaning FPM will take care of the visibility based on the guidelines, ‘view’ meaning that the message will be visible as long as the view to which the message is bound is available, ‘controller’ meaning that the message will be visible as long as the controller that has raised the message is available (see the parameter controller for details), ‘application’ meaning the message will be permanent (until deleted manually by the application developer) and will be visible until the application is alive. The last visibility is ‘pop-up’ which means that the message will be visible only in a pop-up. Not all combination of lifetime and visibility is possible and for obvious reasons some combination like Lifetime = Manual + visibility = pop-up.</td>
</tr>
</tbody>
</table>
are not available. The possible combinations are as follows, described with the constant to be used, the lifetime described first and visibility next:

1. GC_LIFE_VISIBILITY_AUTOMATIC, Automatic + Automatic (Fully handled by FPM)
2. GC_LIFE_VISIBILITY_AUT_POPUP - Automatic + Pop-up (Creation and destruction handled by FPM, visible as long as a pop-up is visible)
3. GC_LIFE_VISIBILITY_MANU_VIEW - Manual + View (Should be deleted by the application visible until the view that created it is visible)
4. GC_LIFE_VISIBILITY_MANU_CONT - Manual + controller (Should be deleted by the application visible until the controller that created it is visible)
5. GC_LIFE_VISIBILITY_MANU_APPL - Manual + Application (Should be deleted by the application visible as long as the application is alive)

### IV_PARAMETERS

| All | Bunch of parameters of the type WDR_NAME_VALUE_LIST that can be stored along with the message. This will be passed to the WD message manager as is and will have to visual change in the message behavior. Refer to the WD message manager documentation for further details |

### R_MESSAGE_USER_DATA

| All | Some additional data that can be stored along with the message. This does not visually influence the message. This can be used by the application developers to provide some error resolution mechanism. See the WD help for further details |

### IV_MESSAGE_INDEX

| All | Numerical value on the order in which this message has to be displayed. If no value is passed (as this is an optional parameter) then the messages will be displayed in the order in which the WD runtime chooses to display them. The messages are sorted according to the following attributes:

1. Error severity
2. Message index (parameter MSG_INDEX)
3. Context element (if existing)
4. Context attribute (if existing) |

### IO_ELEMENT

| All | Reference to a context element to which the message should be bound. The effect is the message will then be clickable and that the focus will shift to a UI element bound to this context element |

### IV_ATTRIBUTE_NAME

| All | The element attribute to which the message has to be mapped. This should be used in conjunction with the IO_ELEMENT |

### IV_IS_VALIDATION_INDEPENDENT

| All | With this parameter, you can define if a message referring to a context attribute or a context element has an influence on the execution of a standard action. If the parameter has the value ABAP_FALSE (default value), the standard action is no longer executed after this message is created. However, if the parameter has the value ABAP_TRUE, the standard action is executed |

### IO_CONTROLLER

| All | Pass the reference to the controller whose lifetime will dictate the lifetime of the messages which have the lifetime set to context |

### IV_MESSAGE_TEXT

| REPORT_MESSAGE | Any free text that should be reported in the message area. When used with the Element and attribute parameters becomes a clickable free text message |

### IS_BAPIRET2

| REPORT_BAPIRET2_MESSAGE | Pass the BAPIRET2 structure directly into the message. The severity of the message is automatically mined from the BAPIRET2 structure. The T100 message that is embedded into the BAPIRET2 structure will be used to display the message text. The lifetime, visibility, context mapping can be set additionally along with the BAPIRET2 structure. Note: If the BAPIRET2 structure contains a severity ‘A’ the message is converted into an exception |

### IV_MSGID

| REPORT_T100_MESSAGE | This parameter should be used while reporting a T100 based message. This parameter should be supplied with the message class |

### IV_MSGNO

| REPORT_T100_MESSAGE | Pass the message number of the message class specified by the
### The Raise XX_Exception methods

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Relevant methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV_TEXT</td>
<td>RAISE_EXCEPTION</td>
<td>Optional text that can be passed while raising a simple exception. This text will be logged and can later be used for analysis.</td>
</tr>
<tr>
<td>IV_MSGID</td>
<td>RAISE_T100_EXCEPTION</td>
<td>Message class ID for the T100 message. Use this parameter to Raise an exception whose text is based on the T100 message mechanism.</td>
</tr>
<tr>
<td>IV_MSGNO</td>
<td>RAISE_T100_EXCEPTION</td>
<td>Message number of the T100 message class.</td>
</tr>
<tr>
<td>IV_PARAMETER_1,</td>
<td>RAISE_T100_EXCEPTION</td>
<td>Optional message parameters</td>
</tr>
<tr>
<td>IV_PARAMETER_2,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IV_PARAMETER_3,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IV_PARAMETER_4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IO_EXCEPTION</td>
<td>RAISE_CX_ROOT_EXCEPTION</td>
<td>Raise an exception from using an exception class inherited from CX_ROOT. This parameter is a mandatory parameter.</td>
</tr>
<tr>
<td>IV_ADDITIONAL_TEXT</td>
<td>RAISE_CX_ROOT_EXCEPTION</td>
<td>Any additional text to be added while reporting an exception based on CX_ROOT</td>
</tr>
<tr>
<td>IS_BAPIRET2</td>
<td>RAISE_BAPIRET2_EXCEPTION</td>
<td>The BAPIRET2 structure for raising an exception.</td>
</tr>
</tbody>
</table>

### The Clear method

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Relevant methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IO_COMPONENT</td>
<td>CLEAR_MESSAGES</td>
<td>The component in which messages were previously reported. Only those messages that were reported from this component will be cleared. If this contains object references other than components then those object references will be used. This is a mandatory parameter.</td>
</tr>
<tr>
<td>IV_EXCLUDE_ERROR</td>
<td>CLEAR_MESSAGES</td>
<td>Pass true if error messages belonging to the component are not to be deleted. This is an optional parameter and the default is false. Meaning that all the error messages belonging to this component will be deleted unless this parameter contains a true value. Looking at the name the parameter indicates that the default value (false) has to be overridden only if error messages are to be saved from being cleared and this parameter contains negative semantic with respect to the method name.</td>
</tr>
<tr>
<td>IV_EXCLUDE_WARNING</td>
<td>CLEAR_MESSAGES</td>
<td>Default is false, override with true if warnings raised for the component are to be saved.</td>
</tr>
<tr>
<td>IV_EXCLUDE_SUCCESS</td>
<td>CLEAR_MESSAGES</td>
<td>Default is false, override with true if success messages raised for the component are to be saved.</td>
</tr>
</tbody>
</table>

The following should be noted for the Report XX methods:

Message severity - if not specified - is defaulted to Error, Lifetime and visibility - if not specified - is defaulted to Automatic and Automatic. By default the message is not mapped to a context element. The component that is raising the message is mandatory to report a message.

If there are minor inconsistencies while reporting the message FPM will automatically take some alternative actions else an exception is raised.

The following are the alternative actions that FMP will take.
1. If the message is reported to be bound to a context element and if the element or the attribute is missing then FPM will report the message but without the binding.

The following are the cases when FPM will raise an exception
1. If the message lifetime is marked to be bound to a controller and the controller is null or not reachable
2. If the component for the message is missing
3. If the message lifetime is marked to Manual + view and if the element or attribute is missing

For specific methods the following should be noted.

REPORT_MESSAGE – The message text is mandatory
REPORT_T100_MESSAGE – The message class and number are mandatory
REPORT_BAPIRET2_MESSAGE – The BAPIRET2 structure is mandatory. If the severity of the message as specified in the BAPIRET2 structure is A then the message is converted into an exception.

The following should be noted for Raise_XX_exception methods

1. All the exceptions are logged in the system with details of the method that was used to raise the exception, the text of the exception any additional text if used.
2. As of SP13 there is no recovery mechanism from the exceptions

The following should be noted about the clear_messages method.
1. The method clears messages from the message area and this method acts upon all those methods that have the lifetime defined as Manual.
2. This is the only method to selectively clear the messages with lifetime equaling manual from the message area
3. This method takes care that accidentally those messages from a different component are not cleared
4. The defaults for the parameters contain a negative semantic with respect to the method name and if the method is called with defaults then all the messages are deleted.

Additional functionality offered by FPM Message Manager:

1. FPM Message Manager has a connection to the ABAP log. To activate this feature, go to transaction SAAB and activate the check point group FPM_RUNTIME_MESSAGES for your user or for all users in the server.
2. You can turn on a log history of the messages for a particular application. When the message log is turned on all the previously reported messages can be seen at a glance (see the FPM requirement for details). This can be done by defining an application parameter FPM_SHOW_MESSAGE_LOG and setting it to true, i.e. FPM_SHOW_MESSAGE_LOG = X.

FAQ on the message manager

1. Can I use the WD message along with the FPM message manager
   → Yes. You can. But you will need to create and maintain your own reference of the WD message manager. Messages that are reported directly into the message manager will not be maintained by FPM after they are reported and the application has to handle the message independent of the FPM lifetime and visibility functionality. Exceptions logged directly into the WD message manager will not be logged under the FPM_RUNTIME_MESSAGES checkpoint group.

2. I want to use the FPM floorplan but I do not want to use the FPM message area, can I do so?
   → Yes, you can use the WD message area only, but functionality from the FPM message manager like automatic lifetime handling, consolidated popup display etc will not be available.
3. Should I create a message area to use the FPM message manager
   ➔ No. If you are using a standard floorplan like OIF or GAF the message area comes inbuilt with the floorplan

4. Can I change the position of the message area?
   ➔ No. You can not change the location of the message area. If you create an additional message area then the messages might be repeated in both the message area views.

5. I reported a message mapped to a context, but I see only the text and the message is not navigable. What is happening?
   ➔ The element and the attribute might not contain proper references. In such a case, FPM will still display the message but it will not be navigable.

6. When I raise an exception I get a dump on the screen and in the stack I see the IF_FPM_MESSAGE_MANAGER as the point where the dump is caused.
   ➔ As of SP13 of NW 7.00 and SP03 of NW7.10 there is very basic support for exception handling for FPM applications. Features like recovery mechanisms from exceptions, special exception screens etc will not be available. All Raise_XX_exception methods in FPM will log any exception raised from it and the force a dump. That way the applications will terminate.