

SAP Manufacturing Execution WPMF Developers Guide



Applicable Release: SAP ME 15.1

Version 1.0

December 2015

SAP ME 15.1 WMPF Developers Guide

© 2015 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company.

SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries. All other product and service names mentioned are the trademarks of their respective companies.

Please see <http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark>  for additional trademark information and notices.

Document History

<u>Document Version</u>	<u>Description</u>
1.0	Initial version

Table of Contents

1	Guide Overview	1
2	Introducing WPMF.....	1
2.1	Users.....	2
2.1.1	User Prerequisites	2
2.2	Terms and Acronyms.....	2
3	WPMF Overview	3
3.1	WPMF Architecture.....	3
3.2	WPMF Configuration.....	6
3.2.1	POD Layout	6
3.3	Framework Control Classes	7
3.4	Framework Context.....	7
3.5	Framework Events	8
4	WPMF Plugin Overview.....	11
4.1	Plugin Architecture.....	11
4.2	POD Selection Model	15
4.3	Special POD Selection Model.....	15
4.4	Selection Model Events	16
4.5	Notification Messages.....	17
5	Developing WPMF Plugins	18
5.1	Planning.....	18
5.2	Developing Plugins.....	19
5.2.1	Adding plugin to faces-config.xml	19
5.2.2	Plugin Event Handling.....	20
5.2.3	Composite Plugin Event Handling	20
5.2.4	UI Plugins	21
5.2.4.1	Plugin Message Area.....	23
5.2.4.2	UI Plugin View	23
5.2.5	Non-UI Plugins.....	23
5.2.5.1	Programmer controlled execution	24
5.2.6	Messages	24

SAP ME 15.1 WMPF Developers Guide

5.2.6.1	Adding Informational Messages.....	24
5.2.6.2	Adding Exception Messages.....	25
5.2.7	Dialogs	25
5.2.8	Fixed, Popover and Popup Areas	25
5.2.8.1	Popup Window	26
5.2.8.2	Modeless Popups	26
5.2.9	IFrame Plugins.....	27
5.2.9.1	IFrame Plugin Communication	27
5.2.9.2	UI5 Plugin Example	28
5.2.10	GAPI Support.....	40
5.2.11	Converters.....	40
5.2.12	Data Table Support.....	42
5.2.12.1	Column Bindings	45
5.2.12.2	Table Row Selections	47
5.2.12.3	Table Column Sorting.....	47
5.2.12.4	Table Column Widths.....	48
5.2.12.5	Column Cell Editors and Renderers	48
5.2.12.6	Column Converters and Validators.....	49
5.2.12.7	Table Row Details Column	49
5.2.13	Browse Support.....	50
5.2.13.1	Deprecated Browse	50
5.2.13.2	New Browse	53
5.2.14	Unsaved Data Behavior	60
5.2.14.1	Unsaved Data Example	61
5.3	Adding Plugins to ME	62
5.3.1	Create the Plugin Activity	62
5.3.2	Assign permissions for Activity.....	62
5.3.3	Add Plugin Activity to a POD	63
5.3.4	Button Activity Processing	63
5.4	Current Limitations	64
5.5	Best Practices.....	64
6	LSF Extended Controls	66

SAP ME 15.1 WMPF Developers Guide

6.1	Page.....	66
6.2	Containers.....	66
6.3	Layouts	66
6.4	Command	66
6.5	Select.....	67
6.6	Input.....	67
6.7	Output	67
6.8	Data	67
6.9	Miscellaneous	67
6.10	Internal Control Migration.....	68
7	Customization with WPMF.....	71
7.1	Configuration Filter	71
7.2	Toolbar State Handler	72
7.2.1	Initial POD Loading	73
7.2.2	Plugin Event Processing	73
7.2.3	Group Buttons.....	74
8	Standalone Application Development.....	75
8.1	Applications using JSF / LSF	75
8.2	Applications using WPMF	76
8.2.1	Example faces-config.xml	77

1 Guide Overview

This developer's guide provides information required for developing for the Web Plugin Management Framework (WPMF). The following topics will be covered by this document:

Major Section	Description
Introducing WPMF	A general introduction to the Web Plugin Management Framework (WPMF), including usage and term definitions.
WPMF Overview	An overview of the Web Plugin Management Framework architecture.
WPMF Plugin Overview	An overview of WPMF Plugin architecture
Developing WPMF Plugins	A complete description of all necessary steps to develop Plugins.
LSF External Controls	A description of controls that replace the LSF internal controls.
Customizing WPMF	A description of features in WPMF that allow for customizing a POD.
Standalone Applications	A description on how to develop standalone applications using LSF and/or WPMF.

2 Introducing WPMF

In computing, a *plugin* consists of a program that interacts with an application (a web browser for example) to provide a certain, usually very specific, function. The Web Plugin Management Framework (WPMF) in SAP Manufacturing Execution manages the display and executions of plugins. It utilizes [Java Server Faces \(JSF\)](#) technologies to provide a clean separation between behavior logic and presentation. User Interface (UI) Plugins are implemented as backing and/or managed beans in most cases and provide the properties and methods that are associated with the client-side components. In addition there are Non-UI Plugins that are not associated directly with a user interface but execute business logic on the server. They are able to communicate to other UI Plugins and the framework for providing feedback to the user. Plugins communicate through defined events and shared models. SAP Manufacturing Execution provides the ability to define one or more Production Operator Dashboards (POD's) which define the layout and execution behavior of plugin's.

2.1 Users

SAP consultants or other System Integrators (programmers) develop plugins to operate from a Manufacturing Execution POD.

2.1.1 User Prerequisites

In order to write plugins, the Integrator must have a working knowledge of Java programming, Java Server Pages and Java Server Faces technologies.

2.2 Terms and Acronyms

Understanding the following terms is necessary to understand the Web Plugin Management Framework (WPMF).

JSP

JavaServer Pages (JSP) technology provides an easy way to create dynamic web pages and simplify the task of building web applications that work with a wide variety of web servers, application servers, browsers and development tools.

JSF

JavaServer Faces (JSF) is a Java-based, request driven MVC web framework based on a component driven UI design model.

LSF

Lightspeed Faces (JSF) is a SAP extension of JSF providing an extended component set.

Plugin

A plugin is a program that interacts with an application (i.e.; web browser) to provide a certain, usually very specific, function.

POD

The Production Operator Dashboard provides the main view which displays and manages the interaction of plugins for the user. A POD is configured using the Manufacturing Execution applications POD Maintenance activity.

3 WPMF Overview

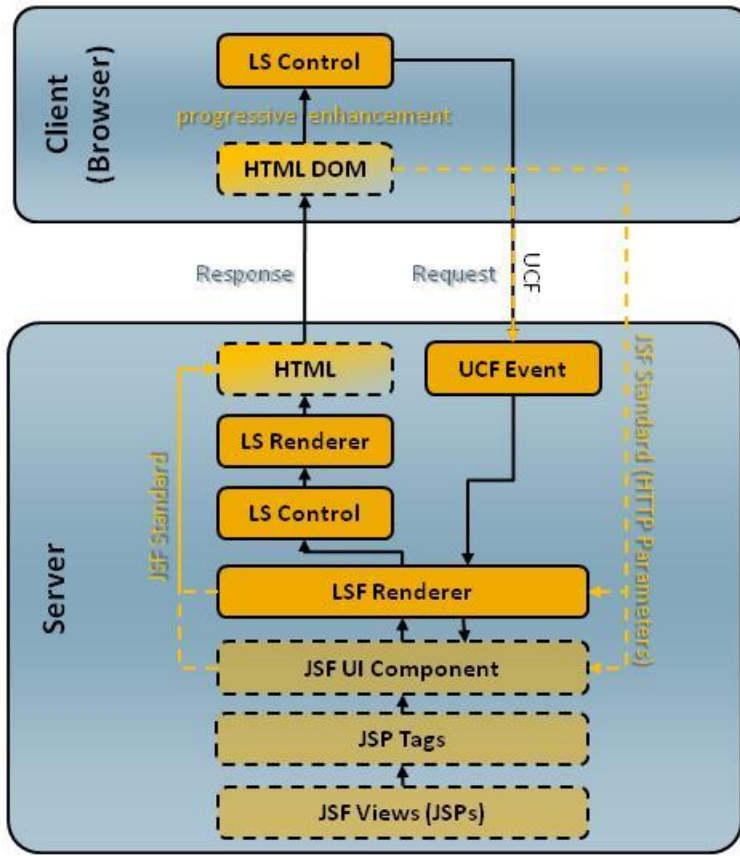
This section provides an overview of the WPMF architecture, configuration and key classes.

3.1 WPMF Architecture

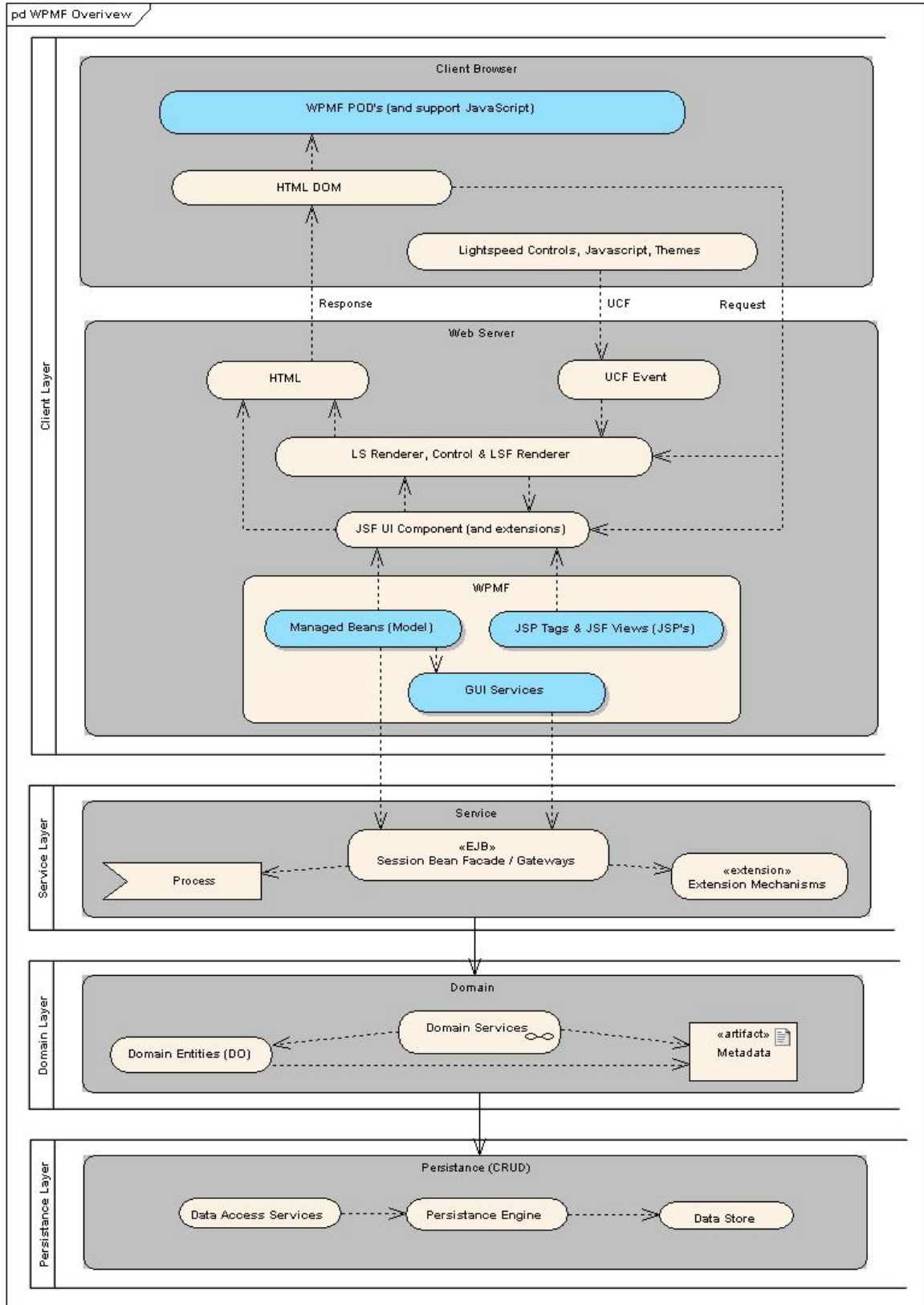
WPMF consists of a set of classes and interfaces that provides the developer all that is needed to implement Plugins that can be executed within a POD. WPMF is designed to operate using JSF and LSF. Manufacturing Execution utilizes a layered architecture approach consisting of 4 layers:

1. Client Layer consists of the integration layer (web service clients) presentation layer.
2. Service Layer consists of the stable API, service implementations and processes that contain application logic and extension mechanisms.
3. Domain Layer consists of domain entities, domain services, metadata and extension mechanisms.
4. Persistence Layer consists of two distinct approaches which use WIP and ODS data storages.

WPMF resides in the Client Layer of this architecture and is tightly integrated with JSF and LSF. The following diagram provides a description of the relationships between JSF and LSF. The “Server” denoted in this diagram is referring to the Web Server and would be considered part of the Client Layer in this architecture.



The following diagram provides an overview of where WPMF fits within the overall Manufacturing Execution architecture.



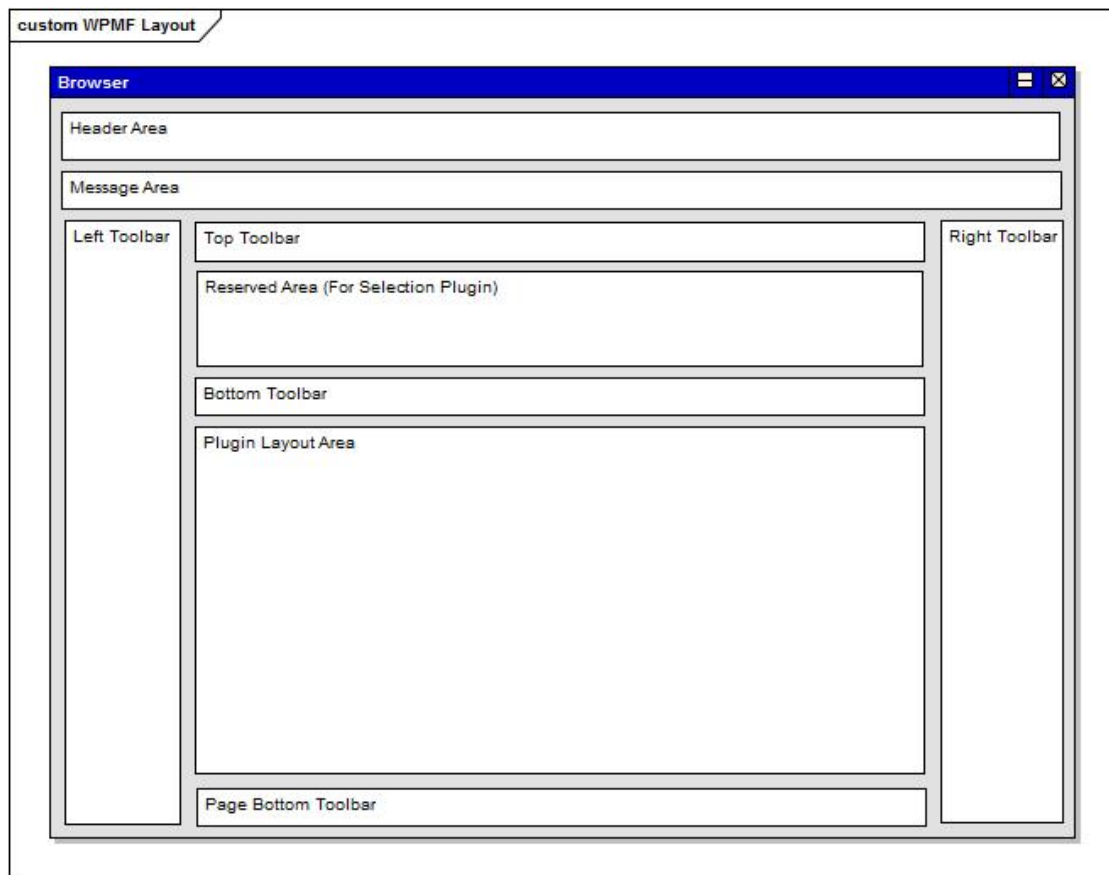
3.2 WPMF Configuration

WPMF provides a framework that allows for the display and execution of multiple PODs concurrently with only one active at a given time. WPMF provides the following high level features:

- A framework Lifecycle and Phase Listener for managing the session data associated with a POD during its lifecycle.
- A predefined layout and layout handler which manages the display of a POD and its associated plugins.
- Event handler for managing the processing of events between plugins.
- Execution handler for managing the execution of activities assigned to toolbar buttons.
- Dialog handler that manages the display of plugins in modal or modeless popup dialogs.

3.2.1 POD Layout

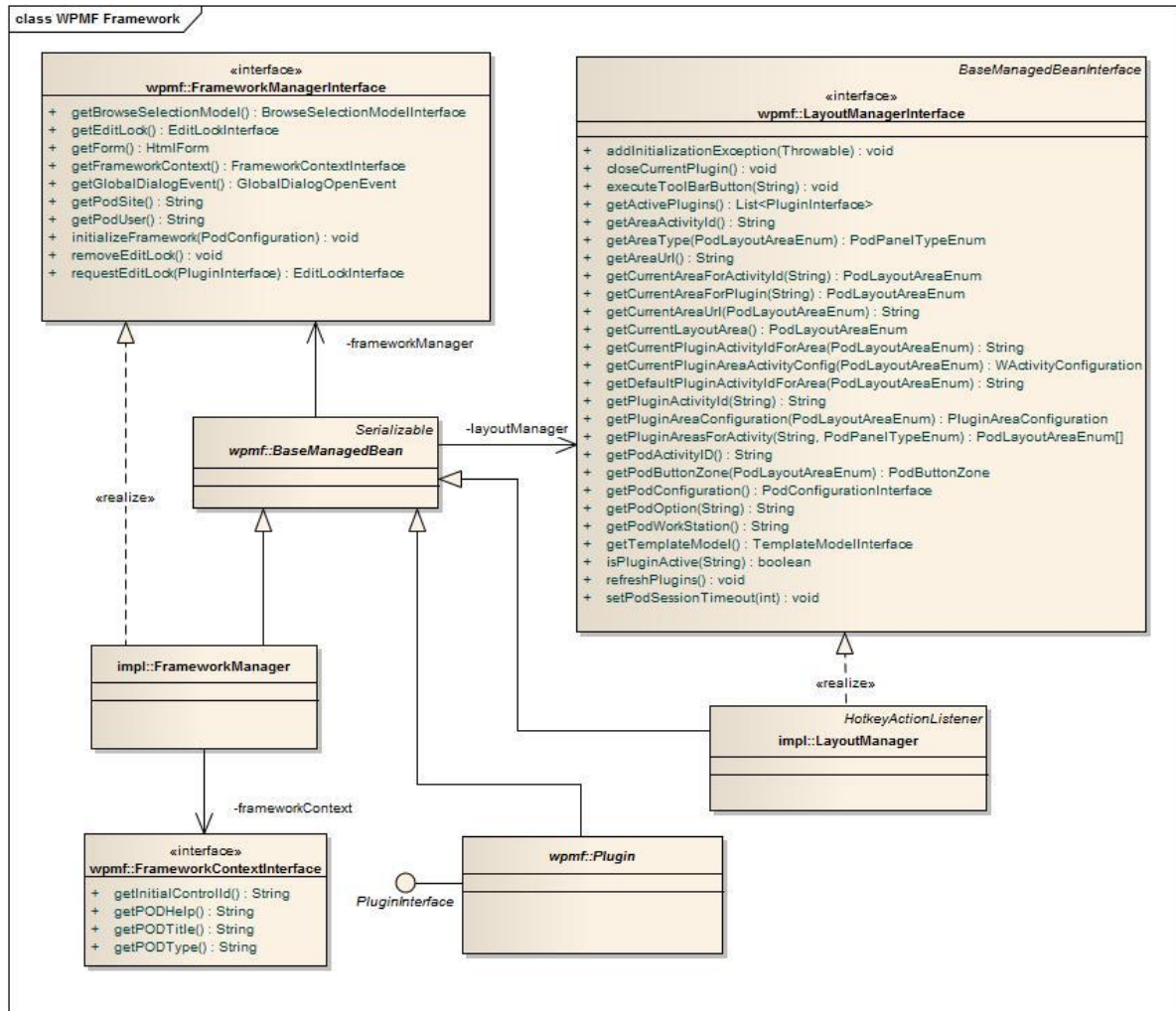
A POD layout consists of five toolbar areas for displaying buttons, a message display area, a reserved area for displaying a single reserved Plugin, optional header layout area and a plugin layout area for displaying plugins. If no JSP page is provided for the specified area, nothing will be displayed in that area. Only one Top or Bottom toolbar can be displayed; not both at same time.



The POD's defined for Manufacturing Execution already are pre-configured to utilize this layout. It further provides 7 different plugin layout areas to choose from when defining a POD in the POD Maintenance activity.

3.3 Framework Control Classes

The Framework Manager is responsible for the initialization of the WMPF framework and providing access to framework level attributes. The Layout Manager is responsible for managing the display of Plugins in the layout area. The following diagram describes the Framework and Layout Managers.



3.4 Framework Context

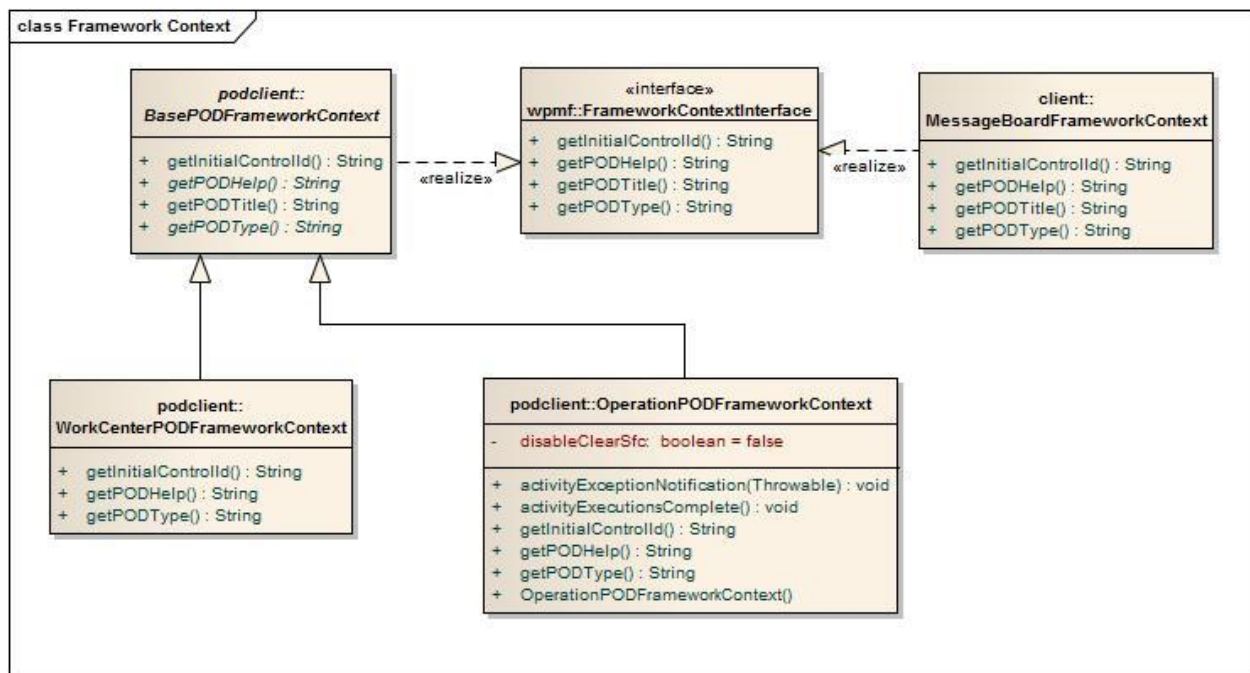
A Multi-POD Plugin is a plugin that can be configured to run in two different PODs. For example, the same Message Board Plugin could run under a Work Center POD or under a standalone Message Board POD. A Multi-POD Plugin needs to obtain a selection model based on the context it is running under. The Accept Buyoff Plugin is one example that is Multi-POD. This can get

executed as an activity assigned to a button on the Work Center POD (acting on Operation List) or as a button on the Message Board Details Plugin.

The Framework Context currently provides the developer the POD type (i.e.; Message Board, Work Center POD, etc) and the title for the POD. The POD type provided is a String representation of the Pod Type as defined in the `com.sap.me.wpmf.PodTypeEnum` enumeration. You can access the context using the following method in the Plugin class:

```
FrameworkContextInterface context = getFrameworkManager().getFrameworkContext();
```

Each POD is responsible for implementing a managed bean that implements the `FrameworkContextInterface`. The managed bean class must be defined in the `faces-config.xml` with a unique name. The name of the bean is defined in the POD Configuration class that is passed to the framework during the initialization of the POD. The following diagram describes the contexts currently defined in the Manufacturing Execution application.



3.5 Framework Events

The Frameworks Plugin Event Manager is responsible for managing events for the framework. This includes plugin to plugin events as well as events for posting dialogs and handling button events. In the case of execution (Non-UI) plugins, the Plugin Event Manager works in conjunction with the Plugin Execution Manager which handles the actual execution of the plugins.

The event manager processes events one of two ways depending on whether client-side updates are required between events. By default events are processed sequentially on the server during the same request thread until all pending events are processed (Sequential event processing). Once all

events are processed the server sends the response back to the server so the client can process any pending control updates. In some cases there is a need for each event to be processed separately as a new request (Event Queue processing). In this case after an event is processed, control is returned to the client to process control updates and then a new request is made back to the server to handle the next event on the queue. While this method is slower in performance than the first type, it is necessary when processing multiple executable (Non-UI) plugins where UI updates are required after execution of each plugin in the sequence.

All event listener interfaces to be managed by the framework must extend from the “com.sap.me.wpmf.PluginEventListener” interface. This interface is only a marker interface in that it does not contain any methods in and of itself. When a plugin is instantiated it is up to the plugin to register the listeners with the framework using the Plugin Event Managers addPluginListeners() method. The base Plugin class contains the following method used to fire an event to other plugins that implement the event listener:

```
/**
 * Fires the specified event by placing it in queue to be processed.
 *
 * @param event EventObject to fire
 * @param listenerType Class of listener
 * @param listenerMethodName String method name in listener class
 */
public void fireEvent(EventObject event, Class listenerType,
                     String listenerMethodName);
```

The following code examples describe a plugin (ExamplePlugin1) firing an event (ExamplePluginEvent) to all ExamplePluginEventListener's; which in this case is ExamplePlugin2.

The Event:

```
public class ExamplePluginEvent extends EventObject {
    public ExamplePluginEvent(Object source) {
        super(source);
    }
    .....
}
```

The Listener:

```
public interface ExamplePluginEventListener extends com.sap.me.wpmf.PluginEventListener {
    public void processExampleEvent(ExamplePluginEvent e);
}
```

The Plugin firing the event:

```
public class ExamplePlugin1 extends Plugin {
    public ExamplePlugin1() {
        super();
    }
    private void fireExampleEvent() {
        ExamplePluginEvent eobj = new ExamplePluginEvent(this);
        fireEvent(eobj, ExamplePluginEventListener.class, "processExampleEvent");
    }
}
```

```
}  
}
```

The Plugin listening for event:

```
public class ExamplePlugin2 extends Plugin implements ExamplePluginEventListener {  
    public ExamplePlugin1() {  
        super();  
        getPluginEventManager().addPluginListeners(this.getClass());  
    }  
    public void processExampleEvent(ExamplePluginEvent e) {  
        // handle the event fired from ExamplePlugin1..  
    }  
}
```

4 WPMF Plugin Overview

This section provides an overview of the features of WPMF Plugins.

4.1 Plugin Architecture

WPMF provides base classes that provide core methods and attributes for a plugin to operate within the framework. The `com.sap.me.wpmf.BaseManagedBean` class provides useful methods for accessing framework level information that all managed beans may find useful.

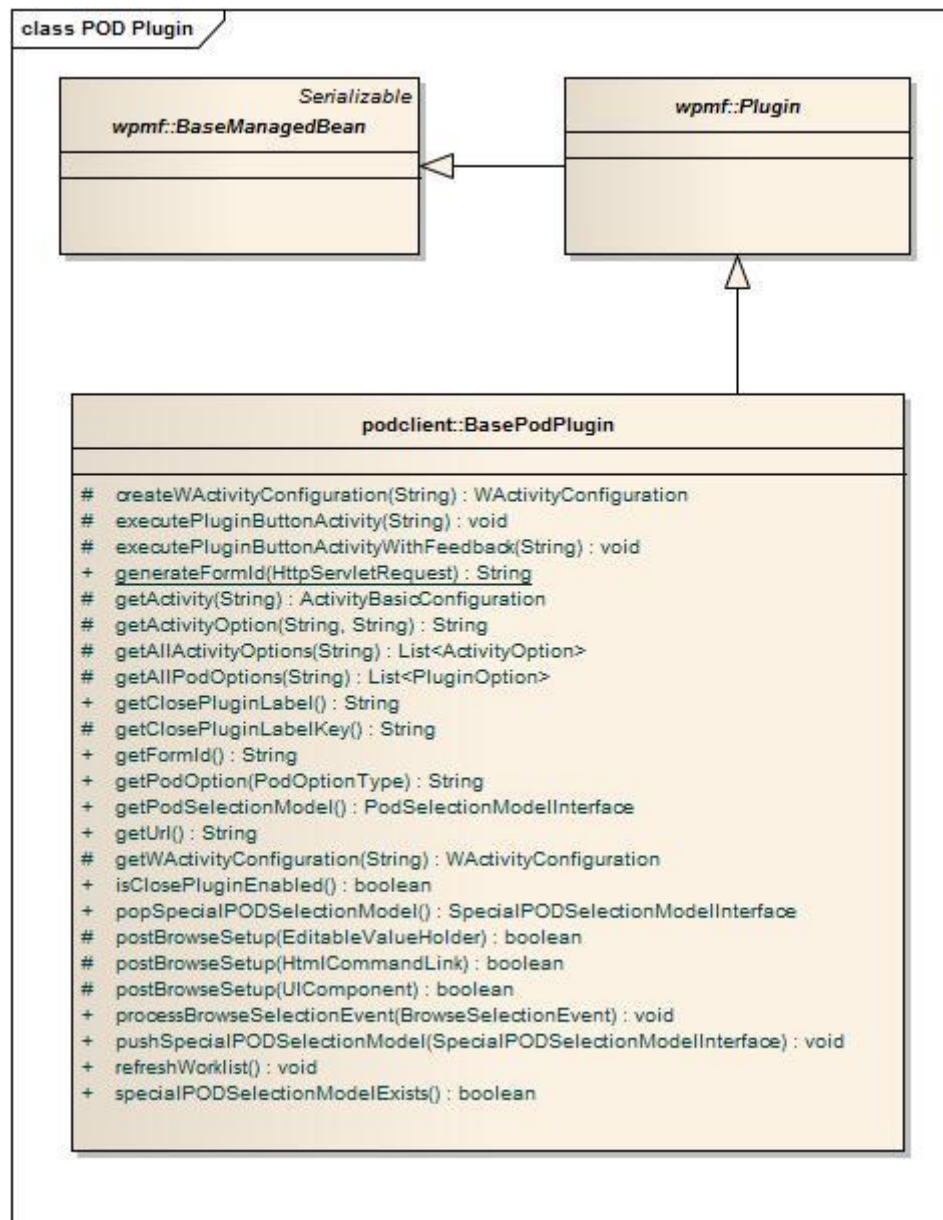


All managed beans that will operate as UI or Non-UI type Plugins by the framework must extend from the com.sap.wpmf.Plugin class as shown in the following diagram. It extends from the BaseManagedBean class so that plugins will have access to all framework level information.



Plugins that operate in a POD defined in the Manufacturing Execution application and wish to have access to production related information should extend from the `com.sap.me.production.podclient.BasePodPlugin` class. This class provides the following additional abilities.

- Access to the Pod Selection Model which is a session data model that maintains the current selection(s) of various key entities such as current SFC(s), Operations(s), etc. It also provides a mechanism for defining special or custom models that adhere to a specific interface.
- Default browse selection event handling methods for those subclasses that use a browse component (i.e.; input field with F4lookup).
- A method for handling standard Unsaved Data capability based on edit lock detection provided by WPMF.



4.2 POD Selection Model

All plugins running in a production POD should be able to interact with a data model that gives information about the various states within the POD during a given session. This model is provided as a session bound object that implements the “com.sap.me.production.podclient.PodSelectionModelInterface” and can be accessed using the following BasePodPlugin class method.

```
/**
 * Returns a reference to the POD selection model.
 * @return PodSelectionModelInterface
 */
public PodSelectionModelInterface getPodSelectionModel();
```

The object represented by the interface provides the appropriate methods for getting and setting of various key states such as the currently selected Sfc(s), Operation(s), RouterStep(s), Resource, ShopOrder(s), etc. Please see the javadoc for a complete list of the methods and their description. The selection model should be updated before firing events to other event listeners.

4.3 Special POD Selection Model

An alternative special POD selection model is available from the BasePodPlugin class. The SpecialPODSelectionModelInterface is a marker Interface so it can hold any selection model that implements this marker. A pop and push method is supplied for storing/accessing this special model from a stack.

```
/**
 * Adds the special POD Selection POD Model.
 *
 * @param SpecialPODSelectionModelInterface
 * @throws TooManySpecialPODSelectionModelsException
 */
public void pushSpecialPODSelectionModel(SpecialPODSelectionModelInterface model)
    throws TooManySpecialPODSelectionModelsException;

/**
 * Returns the SpecialPODSelectionModelInterface. Method removes the
 * SpecialPODSelectionModelInterface once it is procured by a Plugin.
 *
 * @return SpecialPODSelectionModelInterface
 */
public SpecialPODSelectionModelInterface popSpecialPODSelectionModel();
```

There can only be one special POD selection model stored in the BasePodPlugin at a given instant for the session. Once it is procured by the interested Multi-POD Plugin, it will be removed from the BasePodPlugin (set to null) by the framework. In this way, the special POD selection model will be null and any Multi-POD plugins will operate using the standard POD selection model if they are triggered via a standard POD Context.

Multi-POD Plugins should be coded such that it will first look for model data from this special Plugin. If the special selection model is null, then it will work with the Standard POD selection model. It is the responsibility of the invoking Plugin to supply the required special POD selection model for the Multi-POD Plugin.

4.4 Selection Model Events

The selection model is usually updated or refreshed by a plugin before an event is fired or as the result of an event fired through a listener. Any Plugin or event handler should be able to update the selection model when appropriate. The following is a list of core Pod events that are currently associated with an update of the model.

Event	Processing Behavior	Listener interface to be implemented for receiving event notification
OperationChangeEvent	Updates selection model with input or selected value converted to OperationKeyData DTO object.	OperationChangeListenerInterface
RouterStepChangeEvent	Updates selection model with input or selected value converted to RouterStepSelection DTO object.	RouterStepChangeListenerInterface
ResourceChangeEvent	Updates selection model with input or selected value converted to ResourceKeyData DTO object.	ResourceChangeListenerInterface
SfcChangeEvent	Updates selection model with input or selected values converted to SfcSelection objects.	SfcChangeListenerInterface
	Sets inputId field.	
SfcQtyChangeEvent	Sets sfcRef, shopOrderRef, or processLotRef field based on type of worklist or tasklist object. Updates selection model with input or selected value converted to BigDecimal.	SfcQtyChangeListenerInterface
PodSelectionEvent	The system (Work Center Pod Selection Plugin) updates PodSelection model with the current Selection Criterion and fires this event.	PodSelectionListenerInterface
WorklistSelectionChangeEvent	The system (Work Center Worklist Plugin) updates Pod Selection model with selected values converted to SfcSelection DTO objects and fires this event.	WorklistSelectionChangeListener
WorklistChangeEvent	The system (Work Center Worklist Plugin) clears the SfcSelection DTO objects from the PodSelection model and fires this event.	WorklistChangedListener
RefreshWorklistEvent	Upon receiving this event, the system (Work Center Worklist Plugin) refreshes itself using the existing Selection Criterion. This will update the list but will maintain the previous applicable selections, if any. The Pod Selection model also refreshes itself with the selections.	RefreshWorklistListenerInterface

4.5 Notification Messages

Notification Messages provide support for real-time event messages from the server to the client browser. Utilizing [Server-Sent Events](#) (SSE) technology, messages can now be pushed from the server to the client without traditional “polling” from the client. WPMF provides an API to allow defining a Plugin as a listener for messages sent from the server. A Public API (PAPI) provides methods for sending messages from the server.

Messages are identified by a unique “*topic*” name and contain a textual message and optional attributes (key/value pairs). Server-side code uses the following method in the NotificationMessageService PAPI to send a message to “*Consumers*” that are defined as listeners for messages with a specific “*topic*”:

```
/**
 * Adds a Notification Message based on the information provided in the
 * {@link com.sap.me.messaging.AddNotificationMessageRequest}.
 *
 * @param request {@link com.sap.me.messaging.AddNotificationMessageRequest}
 * @throws RequiredValueMissingException if required values are missing
 * @throws BusinessException thrown if any other business exception occurs.
 */
public void addNotificationMessage(AddNotificationMessageRequest request)
    throws RequiredValueMissingException, BusinessException;
```

The following Example code shows how to add a message to be sent to “*Consumers*” of messages with the topic of “*EXAMPLE_MESSAGE_TOPIC*”:

```
NotificationMessageServiceInterface service = (NotificationMessageServiceInterface)
    ServiceLocator.getService("com.sap.me.messaging", "NotificationMessageService");

AddNotificationMessageRequest request = new AddNotificationMessageRequest("EXAMPLE_MESSAGE_TOPIC");
request.setComments("This is an example message");
service.addNotificationMessage(request);
```

A Plugin can be defined as a “Consumer” of Notification Messages using methods defined in the WPMF NotificationMessageManagerHelper class:

```
/**
 * Helper method to add a new Consumer of a Notification Message topic event
 * @param topic Topic of Notification Message to listen for
 * @param listener NotificationMessageListenerInterface to add
 */
public static void addConsumer(String topic, NotificationMessageListenerInterface listener);
```

The Plugin must implement the interface “NotificationMessageListenerInterface” and add the following method which will be called when a messages with the defined “*topic*” is received. The NotificationMessageEvent contains the message information passed from the server and it is up to the plugin to perform any control updates to the UI if necessary.

```
/**
 * Method for processing a Messaging notification event
 * @param event the MessageNotificationEvent to process
 */
public void processMessage(NotificationMessageEvent event);
```

The following example shows how to add (and remove) a Plugin as a consumer of messages with a “topic” of “EXAMPLE_MESSAGE_TOPIC”:

```
public void processPluginLoaded() {
    try {
        NotificationMessageManagerHelper.addConsumer("EXAMPLE_MESSAGE_TOPIC", this);
    } catch (Exception ex) {
        ExceptionHandler.handle(ex);
    }
}

public void closePlugin() {
    try {
        NotificationMessageManagerHelper.removeConsumer("EXAMPLE_MESSAGE_TOPIC", this);
    } catch (Exception ex) {
        ExceptionHandler.handle(ex);
    }
    closeCurrentPlugin(false, false);
}
```

5 Developing WPMF Plugins

Developing a plugin to operate in a production POD requires the following steps:

- Planning
- Developing the plugin
- Adding the plugin to the system

5.1 Planning

To effectively develop the plugin your requirements need to be clearly defined. The planning phase identifies the questions which need to be answered before development begins. Some of the questions that should be asked as part of this phase are

1. Will the plugin be displayed in a fixed or popover area or as a popup dialog?
2. If the answer to 1 above is yes; will it be used as both a default and non-default type plugin?
3. If the answer to 1 above is no; will it be executed as an activity assigned to a button?
4. If the answer to 3 above is yes; will the plugin display dialogs?
5. Is the plugin used as the managed bean backing a dialog (not popup)?
6. Will the plugin be used for both standard and touch devices?
7. Will the plugin be displayed on different sized display devices?
8. What plugin events will the plugin need to listen for?
9. What new events will the plugin fire to other plugins?
10. What service layer API's will be required?

5.2 Developing Plugins

Once you know the answers to the above questions, you will know the type of plugin you must develop and what is required to configure the plugin to run in a POD.

A UI Plugin is defined as a “managed bean” in a modules “faces-config.xml” file. This allows for referencing the plugin by name in the view (jsp page) using EL expressions. For “managed” beans the lifecycle is managed by JSF and in most cases the beans should be defined to the session scope. A Non-UI type plugins lifecycle is managed by the WPMF and are not considered “managed beans”; therefore it does not necessarily need to be defined in the faces-config.xml file.

A UI Plugin can have multiple sized views (Medium, Large and Extra-Large) defined for each device type (Touch or Standard). Medium is defined for tablet sized devices, Large for devices laptop sized screens and Extra-Large for large display monitors. You define the views in the managed bean definition in the faces-config.xml file.

When a JSF/LSF control is bound to a managed bean via the “binding” property using an EL expression, it is a best practice to create a “backing bean” defined to the request scope for that purpose. Backing beans provide the server-side representation of the components located on the client-side page. Backing beans should only contain references to the controls bound to the client. They should be defined to request scope since the bound control(s) are only valid for the lifecycle of the request and once the request has been processed; the backing bean properties will not be valid until the next request. The base “com.sap.me.wpmf.PluginBackingBean” class provides for the binding of the plugins “container” and “pluginMessageLayout” components and can be used as is or extended to add additional component bindings.

5.2.1 Adding plugin to faces-config.xml

The following example describes how to add a Plugin which utilizes a PluginBackingBean for the server-side components to a modules faces-config.xml file. If a backing bean is not required, eliminate the “backingBeanName” property from the plugin definition and do not define a backing bean. It also shows 3 different size views defined for both Touch and Standard device types. View definitions are optional. If a view is not defined for the POD definition, the default view will be used. You should always specify at least the default view.

```
<managed-bean>
  <managed-bean-name>examplePlugin</managed-bean-name>
  <managed-bean-class>com.sap.me.example.ExamplePlugin</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>examplePlugin</value>
  </managed-property>
  <managed-property>
    <property-name>backingBeanName</property-name>
    <value>exampleBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>touchMediumUrl</property-name>
    <value>/com/sap/example/view/exampleTMPlugin.jsp</value>
  </managed-property>
</managed-bean>
```

```
<property-name>touchLargeUrl</property-name>
<value>/com/sap/example/view/exampleTLPlugin.jsp</value>
</managed-property>
<managed-property>
  <property-name>touchExtraLargeUrl</property-name>
  <value>/com/sap/example/view/exampleTELPlugin.jsp</value>
</managed-property>
<managed-property>
  <property-name>mediumUrl</property-name>
  <value>/com/sap/example/view/exampleMPlugin.jsp</value>
</managed-property>
<managed-property>
  <property-name>largeUrl</property-name>
  <value>/com/sap/example/view/exampleLPlugin.jsp</value>
</managed-property>
<managed-property>
  <property-name>extraLargeUrl</property-name>
  <value>/com/sap/example/view/exampleELPlugin.jsp</value>
</managed-property>
<managed-property>
  <property-name>defaultUrl</property-name>
  <value>/com/sap/example/view/examplePlugin.jsp</value>
</managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>exampleBackingBean</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.PluginBackingBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>exampleBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>assignedBeanName</property-name>
    <value>examplePlugin</value>
  </managed-property>
</managed-bean>
```

5.2.2 Plugin Event Handling

Plugins communicate to with each other using events. See [Framework Events](#) for more information on how to register event listeners for your plugin.

5.2.3 Composite Plugin Event Handling

For performance reasons it may make sense for a plugin to bundle related events and dispatch them as a single composite event that specific registered plugins can consume. The registered plugin class implements the “com.sap.me.wpmf.CompositeEventListenerInterface” to receive a single composite event containing a list of discrete events.

Optimization is achieved since the framework event manager will only dispatch a single event to the plugin(s) instead of a list of discrete events.

Identifying related events is purely based on use-case / functional needs. If related events have to be fired sequentially based on a user action, it must be fired as a Composite Event. In this case the

originating plugin creates a List of CompositeEventObject's for each discrete event and then fires the list using Plugin classes "fireEvents(List<CompositeEventObject >)" method.

As an example there could be a case where the originating plugin encounters a condition where 2 events need to be fired (e.g.; Example1Event and Example2Event); hence being a candidate for performance optimization. The Plugin instead fires a single composite event containing a list of related events. The receiving plugin will only process one event notification eliminating redundant processing of multiple events.

```
public void processCompositeEvent(List<EventObject> eventObjList) {
    if (eventObjList!= null && eventObjList.size() > 0) {
        Iterator<EventObject> eventObjListIt = eventObjList.iterator();
        while (eventObjListIt!= null && eventObjListIt.hasNext()) {
            EventObject evtObj = eventObjListIt.next();
            if (evtObj instanceof Example1Event || evtObj instanceof Example2Event ) {
                .....
                break;
            }
        }
    }
}
```

5.2.4 UI Plugins

UI Plugins at minimum consist of a plugin class and one or more client-side views (i.e.; jsp page) containing JSF / LSF controls with properties that are bound to the managed bean. An example view is described below:

```
<%@page pageEncoding="UTF-8"%>
<%@ page language="java"%>
<%@ taglib prefix="h" uri="http://java.sap.com/jsf/html"%>
<%@ taglib prefix="f" uri="http://java.sap.com/jsf/core"%>
<%@ taglib prefix="sap" uri="http://java.sap.com/jsf/html/extended"%>
<%@ taglib prefix="icon" uri="http://www.sap.com/ui/icons"%>

<f:subview id="exampleView">

    <sap:panel id="examplePanel" width="100%" height="100%" isCollapsible="false"
        contentAreaDesign="transparent" isCollapsed="false"
        binding="#{exampleBackingBean.container}">
        <f:facet name="header">
            <h:outputText value="#{ examplePlugin.title}" />
        </f:facet>
        <f:facet name="toolbar">
            <sap:toolBar>
                <f:facet name="rightAlignedItems">
                    <h:panelGroup>
                        <h:commandButton id="exampleHelp" image="#{icon:url('Icon','Help')}}"
                            alt="#{gapiI18nTransformer['toolTip.help.TEXT']}"
                            title="#{gapiI18nTransformer['toolTip.help.TEXT']}"
                            action="#{examplePlugin.showPluginHelp}">
```

```
<sap:ajaxUpdate />
</h:commandButton>
</h:panelGroup>
</f:facet>
</sap:toolBar>
</f:facet>

<sap:ajaxUpdate render="#{sap:toClientId('examplePanel')}" />

<sap:panelGrid width="100%" height="100%" cellHalign="start" cellValign="top">
  <sap:panelRow cellHalign="start" cellValign="top" >
    <sap:panelGroup backgroundDesign="transparent" width="100%" height="100%"
      halign="start" valign="top">

      <sap:panelGrid width="100%" height="100%" cellHalign="start" cellValign="top">
        <sap:panelRow cellHalign="start" cellValign="top" >
          <sap:panelGroup backgroundDesign="transparent" width="100%"
            halign="start" valign="top">
            <sap:panelGrid binding="#{examplePlugin.pluginMessageLayout}" width="100%"/>
          </sap:panelGroup>
        </sap:panelRow>

        <sap:panelRow cellHalign="start" cellValign="top">
          <sap:panelGroup backgroundDesign="transparent" width="100%" height="100%"
            halign="middle" valign="top">

            Plugin controls go here....

          </sap:panelGroup>
        </sap:panelRow>
      </sap:panelGrid>
    </sap:panelGroup>
  </sap:panelRow>
  <sap:panelRow cellHalign="center" cellValign="top">
    <sap:panelGroup backgroundDesign="transparent" width="100%"
      halign="middle" valign="top">
      <sap:commandButton value="#{examplePlugin.closePluginLabel}"
        rendered="#{examplePlugin.closePluginEnabled}"
        action="#{examplePlugin.closePlugin}">

        <sap:ajaxUpdate />
      </sap:commandButton>
    </sap:panelGroup>
  </sap:panelRow>
</sap:panelGrid>
</sap:panel>
</f:subview>
```

The `<f:subview>` tag is required for the view and must contain a unique ID.

The LSF extended panel control provides a standard panel with the ability to add a title and toolbar icons such as the help button icon. The panel “id” property should contain a unique control id which is used to support delta update requests to the server and is referenced in the `sap:toClientId(‘exampleContainer’)` helper function. The “binding” property should be bound to

the `PluginBackingBean`'s "container". The container is used to support plugin control updates from the server.

The `<sap:ajaxUpdate render="{sap:toClientId('examplePanel')}" />` tag defines the container that will be updated when "delta" submits are made to the server if a submit is made without a specific `ajaxUpdate` control assigned. The container and all children will be updated with changes made on the server. In the example above the container that is updated will be the panel and all children within it. The `render` attribute must be set to the full path of the container which in this case is provided by the special `sap:toClientId('exampleContainer')` function. The `ajaxUpdate` control can be assigned as a child to any extended command control. If used without specifying the `render` attribute, the delta update will be the control it is assigned to (i.e.; the `commandButton` controls in the example above).

The toolbar facet is used to add the help button to the panel. The action event method "showPluginHelp" is a method already defined to the base `Plugin` class. The panel control is only able to contain a single child content control and in most cases will be the extended `<sap:panelGrid>` layout control.

5.2.4.1 Plugin Message Area

If the plugin you are creating will utilize its own area for display of messages, you must include the appropriate panel grid component assigned to the "pluginMessageLayout" property. The base `PluginBackingBean` class already contains the implementation of this property. The following example shows this binding.

```
<sap:panelGrid binding="{examplePlugin.pluginMessageLayout}" width="100%"/>
```

5.2.4.2 UI Plugin View

The `Plugin` class provides the framework the URL to the view by calling the `Plugin` classes "getUrl()" method. The default implementation will obtain the appropriate URL from what is defined in the `faces-config.xml` file (See [Developing Plugins](#)). If the activity rule "PLUGIN_URL" is used to assign the URL in Activity Maintenance, this URL will be used regardless of what is defined in the `faces-config.xml` file. Dialog plugins are not defined in Activity Maintenance so must override the "getUrl()" method and return the URL to the view as the following example shows.

```
public String getUrl() {  
    return "/com/example/client/example.jsp";  
}
```

5.2.5 Non-UI Plugins

Non-UI Managed Bean classes must extend from the `Plugin` super class. The "execute()" method is the only method that **must** be implemented for use in the framework. The default behaviour is that once the `execute()` method is finished, the execution process is complete. In some cases the

execute method is used to just start the execution of the plugin (perhaps causing a dialog to display). For this reason the framework supports programmer controlled execution.

5.2.5.1 Programmer controlled execution

The ability for Non-UI plugins to control when they are finished executing is also supported. The following two methods in the Plugin class support “blocking” between plugin executions:

```
/**
 * called to end processing of plugin and start next plugin in chain
 */
protected void complete();

/**
 * Returns whether or not this plugin will use blocking during
 * the execution of the plugin. This is only valid for executable
 * plugins. By default blocking is not used. When enabled the
 * plugin will remain active until the complete() method is called.
 * Developers should override this method to return true to use blocking.
 */
public boolean isBlockingEnabled();
```

By default blocking is NOT enabled so that execution of the plugin will end when the execute() method is done. When enabled, the next activity in the chain is not executed until complete() is called. The complete method will then signal to the framework to start execution of the next plugin in the chain (if any).

5.2.6 Messages

Messages are generally classified as informational (Information or Warning), confirmational (Success) and error type messages (Exceptions or other User specified errors). There are two areas where messages can be displayed; the main POD area located at the top of the POD and the Plugin area which should be located at the top of each UI Plugin.

5.2.6.1 Adding Informational Messages

Use the “com.sap.me.wpmf.util.MessageHandler” utility class for adding or clearing informational or confirmational messages to the POD or plugin message area. The following example describes adding a message using a key in the LocaleSpecificText.properties resource to a plugin message area.

```
Map<String, Object> messageMap = new HashMap<String, Object>();
messageMap.put("%EXAMPLE_ID%", this.getActivityId());
MessageHandler.handle("example.success. default", messageMap, MessageType.SUCCESS, this);
```

To display same message in the POD main area use the following:

```
MessageHandler.handle("example.success. default", messageMap, MessageType.SUCCESS);
```

5.2.6.2 Adding Exception Messages

Use the “com.sap.me.wpmf.ExceptionHandler” utility class for adding or clearing exception messages to the POD or plugin message area. The following example shows how to add an exception message to a plugins message area.

```
ExceptionHandler.handle(ex, this);
```

To display same exception to the POD message area use the following:

```
ExceptionHandler.handle(ex);
```

5.2.7 Dialogs

The framework supports the display of up to 5 modal dialogs displayed at the same time where only one is active. The dialog is a UI Plugin that is not necessarily defined as an activity in the POD configuration. The following steps are required to open a dialog (or nested dialog):

1. Create the dialogs plugin class and view (jsp).
2. Create and load the “com.sap.me.wpmf.GlobalDialogOpenEvent” that will be used to notify the framework to open the dialog. You may also extend the base GlobalDialogOpenEvent to add custom properties that will be available to the dialog plugin after the dialog is opened.
3. Call the Plugin class openDialog(event) method to open the dialog.

The following Plugin class methods are provided to close a dialog.

1. Call “closeDialog()” to close the currently active dialog using the base “com.sap.me.wpmf.GlobalDialogCloseEvent”.
2. Call “closeDialog(event)” to close the dialog described by the input “com.sap.me.wpmf.GlobalDialogCloseEvent” object.
3. Call “closeDialogAndExecute(executionPlugin)” to close the currently active dialog and then invoke the execute() method on the input PluginInterface class.

Note: Do not confuse a [Popup Window](#) with a Dialog. A Popup is a modal or modeless dialog that is specially designed to display a UI Plugin as defined to the Popup layout area. It functions exactly the same as a Popover plugin would and is managed entirely by the framework.

The GlobalDialogOpenEvent class provides methods for rendering and modifying buttons on a framework toolbar for legacy purposes only. Please refrain from using the framework toolbar for new dialog plugins. It is preferred that the plugin will add and manage any buttons that are required.

5.2.8 Fixed, Popover and Popup Areas

The framework recognizes 3 types of layout areas that are used in the POD to describe where UI Plugins will be displayed. Plugin writers need to create their plugins to be aware of the type of area they are defined to so that the required “Close” button is defined to allow the plugin to be removed from the layout area when processing is complete. Plugins defined as a default Fixed

area plugin do not have a “Close” button, however if a plugin is configured as an optional (replaceable) fixed area plugin the ability to close the plugin is required. Plugins defined to the Popover and Popup areas must provide the ability to close the plugin. The Plugin class provides the following methods to assist in determining the type of Layout area and methods to close a plugin:

Plugin Method	Purpose
public PodPanelTypeEnum getAreaType();	Returns FIXED, POPOVER or POPUP Area Type
public boolean isDefaultPlugin();	Returns true if this the default plugin assigned to the area
public boolean isPopupPlugin();	Returns true if this is assigned as a Popup
public void closeCurrentPlugin();	Will close the plugin.
public void closeCurrentPlugin(boolean clear);	Will close the plugin and optionally clear messages in the plugin

5.2.8.1 Popup Window

The Plugin class also provides methods to allow the plugin writer the ability to define the attributes the Popup Window should have as described in the following methods:

Plugin Method	Purpose
Public String getPopupPosition()	Returns the top / left position to display popup where values are separated by a comma.
public String getPopupTitle();	Returns the title for the Popup Window
public int getPopupWidth();	Returns window width in “em” measurement units
public int getPopupHeight();	Returns window height in “em” measurement units
public boolean isPopupResizable()	Returns whether window should be resizable or not.
public boolean isPopupModal()	Returns whether popup is modal (default) or modeless.
public boolean isNativeModelessPopup()	Returns whether or not to use the native browser dialog or default panel popup.

5.2.8.2 Modeless Popups

A popup window is modal by default, however it can be defined as modeless by overriding the isPopupModal() method and returning false. All dialogs and popups use the <sap:panelPopup> control for displaying the popup window instead of a browser dialog. While this helps

performance it does limit the area the popup can be displayed to the actual form area. In some instances it is desirable to display a browser dialog for the modeless popup (i.e.; VTR Plugin). The `isNativeModelessPopup()` method was provided to support this. If used, the browser dialog will be displayed and can be moved anywhere on the desktop as well as minimized if needed.

JSF only allows the active window to send and receive Ajax updates to/from the server. In order for a modeless popup to cause the main pod plugins to be updated it must call the Plugin classes `activateMainWindow()` method. This will cause the main pod window to receive focus and a `LayoutRefreshEvent` to be fired to all plugins that have implemented the `LayoutRefreshEventListener` interface.

In order for a main pod plugin to cause a modeless popup to be updated it must call the Plugin classes `activateModelessPopup()` method. This will cause the modeless popup window to receive focus and the popup plugins `processModelessPopupFocused()` method to be called.

5.2.9 IFrame Plugins

IFrame plugins allow non-JSF UIs to function within the WPMF plugin architecture. JavaScript APIs are provided for communication between the page included in the iframe and the WPMF plugin on the server. There are basically four components that make up an IFrame plugin:

- WPMF plugin class that extends “`com.sap.me.production.podclient.BaseIFramePlugin`” and implements the “`com.sap.me.production.podclient.IFrameOnMessageListenerInterface`” to listen to events (messages) from the page defined to the iframe.
- View (JSP page) for the plugin that has an `<ls:iframe>` control to load non-JSF view into.
- Non-JSF view (JSP page) that is loaded into the iframe.
- JavaScript APIs for messaging between the Non-JSF view and the WPMF plugin.

5.2.9.1 IFrame Plugin Communication

The communication between the page defined to the iframe and the WPMF plugin is performed using structured messages that include a message text and map of key/value pairs. JavaScript functions are used to perform the communication to and from the plugin on the server.

The IFrame plugin receives messages from the Non-JSF view by implementing the interface “`IFrameOnMessageListenerInterface`” and adding the following method:

```
public void processIFrameOnMessage(IFrameOnMessageEvent event) {
    String msg = event.getMessage();
    Map<String,String> messageData = event.getMessageData();
    ....
}
```

The IFrame plugin sends messages back to the Non-JSF view using the following method that is provided by the `BaseIFramePlugin` class:

```
/**
 * Fires message event with input message and data to client
 */
```

```
* @param clientId String client id of iframe
* @param message String message to send
* @param messageData Map<String,String> data to send with message
*/
protected void fireIFramePostMessageEvent(String clientId, String message, Map<String,String> messageData) {
    IFramePostMessageEvent event = new IFramePostMessageEvent(this, clientId, message, messageData);
    fireEvent(event, IFramePostMessageListenerInterface.class, "postIFrameMessage");
}
```

The “clientId” is the full client ID of the <iframe> component and must be provided so that the message is sent to the correct page.

The Non-JSF page receives messages from the IFrame plugin by implementing the following JavaScript function:

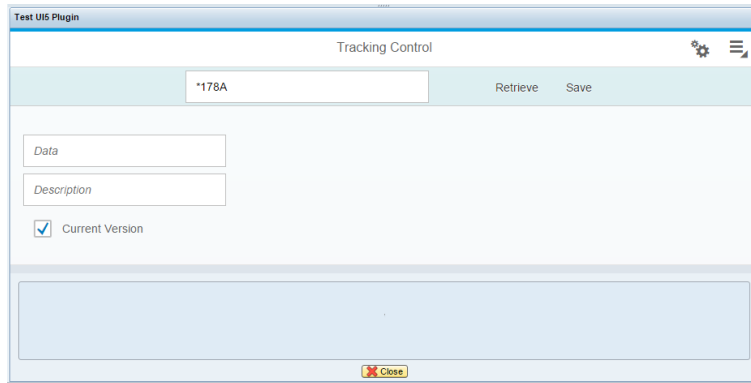
```
/**
 * receives messages from POD
 *
 * @param sMessage <string> message
 * @param oMessageData <object> map of name / value pairs
 */
function receiveMessageFromPod(sMessage, oMessageData) {
    for (var sKey in oMessageData){
        var sValue = decodeURIComponent(oMessageData[sKey]);
        ....
    }
};
```

The Non-JSF page sends messages to the IFrame plugin by calling the core JavaScript function “sendMessageToPod()” as the following example shows:

```
var oMessageData = {
    "key1" : "value1",
    "key2" : "value2",
    ....
    "lastkey" : "last value"
};
window.sendMessageToPod("example message", oMessageData);
```

5.2.9.2 UI5 Plugin Example

This example shows how to create a plugin using UI5 controls. UI5 requires that you run the browser in the Standards mode. The first step is to create the IFrame plugin and view. The example UI will look like the following screenshot. The example code shows how to send data to and from the plugin on the server. This can be used to initialize the UI5 view as well as retrieve and save data. The example plugin view (JSP) includes an <iframe> control that will reference the UI5 page. In the screenshot below the text area above the Close button is a JSF control and above that is the UI5 view displayed in the <iframe> control.



5.2.9.2.1 UI5 Example Plugin Class

The UI5 plugin class, shown below, is much like other WMPF plugin classes except it provides additional functions for working with client views that have been defined in an iframe.

```
package com.sap.me.production.client.test;

import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;

import com.sap.me.production.SfcKeyData;
import com.sap.me.production.podclient.BasePodPlugin;
import com.sap.me.production.podclient.IFrameOnMessageEvent;
import com.sap.me.production.podclient.IFrameOnMessageListenerInterface;
import com.sap.me.production.podclient.IFramePostMessageEvent;
import com.sap.me.production.podclient.IFramePostMessageListenerInterface;
import com.sap.me.production.podclient.PodSelectionModelInterface;
import com.sap.me.production.podclient.SfcSelection;
import com.sap.me.wpmf.util.FacesUtility;
import com.sap.me.wpmf.util.MessageHandler;
import com.sap.me.frame.Utills;

public class UI5ExamplePlugin extends BasePodPlugin implements IFrameOnMessageListenerInterface{
    private static final long serialVersionUID = 1L;

    private String message = null;

    public UI5ExamplePlugin () {
        super();
        getPluginEventManager().addPluginListeners(this.getClass());
    }

    @Override
    public String getBeanName() {
        if (!Utills.isBlank(super.getBeanName())) {
            return super.getBeanName();
        }
    }
}
```

```
}
return "ui5ExamplePlugin";
}

@Override
public String getBackingBeanName() {
    if (!Utils.isBlank(super.getBackingBeanName())) {
        return super.getBackingBeanName();
    }
    return "ui5ExampleBackingBean";
}

@Override
public String getUrl() {
    String requestContextPath = FacesUtility.getExternalContext().getRequestContextPath();
    return requestContextPath + "/com/sap/me/production/client/test/ui5example.jsp";
}

@Override
public boolean isPluginUpdatedAfterEvent(Object event) {
    return false;
}

public void closePlugin() {
    MessageHandler.clear(this);
    closeCurrentPlugin();
}

public String getInitialFocusId() {
    UIComponent panel = getContainer();
    if (panel == null) {
        return null;
    }
    UIComponent comp = findComponent(panel, "ui5CloseButton");
    if (comp == null) {
        return null;
    }
    return comp.getClientId(FacesContext.getCurrentInstance());
}

public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}

@Override
public void processIFrameOnMessage(IFrameOnMessageEvent event) {
    String msg = event.getMessage();
    Map<String,String> messageData = event.getMessageData();

    if ("retrieve".equalsIgnoreCase(msg)) {
        retrieveData(msg, messageData);
    } else if ("save".equalsIgnoreCase(msg)) {
        saveData(msg, messageData);
    }
}
```

```
} else if ("afterRendering".equalsIgnoreCase(msg)) {
    PodSelectionModelInterface model = getPodSelectionModel();
    List<SfcSelection> sfcs = model.getSfcs();
    if (sfcs != null && sfcs.size() == 1) {
        SfcKeyData sfckey = sfcs.get(0).getSfc();

        Map<String,String> msgData = new HashMap<String,String>();

        msgData.put("searchData", sfckey.getSfc());

        fireMainFramePostMessageEvent("updates", msgData);
    }
}

private void retrieveData(String msg, Map<String,String> messageData) {
    Map<String,String> msgData = messageData;
    if (messageData == null) {
        msgData = new HashMap<String,String>();
    }
    String searchValue = null;
    if (messageData != null && !messageData.isEmpty()) {
        searchValue = messageData.get("searchData");
    }

    if (!Utils.isBlank(searchValue)) {
        msgData.put("inputData", searchValue);
        msgData.put("inputDescription", "Fantastic!");
        msgData.put("checkBoxVersion", "false");
    }
    fireMainFramePostMessageEvent("updates", msgData);
}

private void saveData(String msg, Map<String,String> messageData) {

    StringBuffer buf = new StringBuffer();
    if (!Utils.isBlank(getMessage())) {
        buf.append(getMessage());
        buf.append("\n");
    }
    buf.append("message: ");
    buf.append(msg);
    buf.append("\n");

    if (messageData != null && messageData.size() > 0) {
        Set<String> s = messageData.keySet();
        Iterator<String> i = s.iterator();
        while (i.hasNext()) {
            String key = i.next();
            String value = messageData.get(key);
            buf.append(key);
            buf.append(": ");
            buf.append(value);
            buf.append("\n");
        }
    }
}
```

```
setMessage(buf.toString());

// update the text area control
UIComponent panel = getContainer();
if (panel != null) {
    UIComponent comp = findComponent(panel, "messageTextArea");
    if (comp != null) {
        FacesUtility.addControlUpdate(comp);
    }
}

private String getIFrameClientId() {
    UIComponent panel = getContainer();
    if (panel == null) {
        return null;
    }
    UIComponent comp = findComponent(panel, "ui5ExampleFrame");
    if (comp == null) {
        return null;
    }
    return comp.getClientId(FacesContext.getCurrentInstance());
}

private void fireIFramePostMessageEvent(String message, Map<String,String> messageData) {
    String clientId = getIFrameClientId();
    IFramePostMessageEvent event = new IFramePostMessageEvent(this, clientId, message, messageData);
    fireEvent(event, IFramePostMessageListenerInterface.class, "postIFrameMessage");
}
}
```

5.2.9.2.2 UI5 Example Plugin JSP

The UI5 plugin view (“ui5exampleplugin.jsp”) is a standard LSF JSP that uses the internal `<iframe>` control to hold the UI5 page as shown below:

```
<%@page pageEncoding="UTF-8"%>
<%@ page language="java"%>
<%@ taglib prefix="h" uri="http://java.sap.com/jsf/html"%>
<%@ taglib prefix="f" uri="http://java.sap.com/jsf/core"%>
<%@ taglib prefix="sap" uri="http://java.sap.com/jsf/html/extended"%>
<%@ taglib prefix="ls" uri="http://java.sap.com/jsf/html/internal"%>
<%@ taglib prefix="icon" uri="http://www.sap.com/sapme/ui/icons"%>

<f:subview id="ui5ExampleView">

    <sap:panel id="ui5ExamplePanel" width="100%" height="100%" isCollapsible="false"
        contentAreaDesign="transparent" isCollapsed="false" binding="#{ui5ExampleBackingBean.container}">
        <f:facet name="header">
            <h:outputText value="Test UI5 Plugin" />
        </f:facet>

        <f:attribute name="sap-delta-id" value="#{sap:toClientId('ui5ExamplePanel')}" />

        <sap:panelGrid width="100%" height="100%" cellHalign="start" cellValign="top">

            <sap:panelRow cellHalign="start" cellValign="top" >
```

```
<sap:panelGroup backgroundDesign="transparent" width="100%" valign="top">
  <sap:panelGrid binding="#{ui5ExampleBackingBean.pluginMessageLayout}" width="100%" />
</sap:panelGroup>
</sap:panelRow>

<sap:panelRow cellHalign="start" cellValign="top">
  <sap:panelGroup id="reportHtml" width="100%">
    <sap:panel id="htmlContainer" isCollapsible="false" contentAreaDesign="transparent"
borderDesign="none" width="100%">
      <sap:panelGrid id="htmlGrid" columns="1" width="100%" cellHalign="start" cellValign="top">
        <sap:panelGroup backgroundDesign="transparent" width="100%" height="300px" valign="start"
valign="top">
          <iframe id="ui5ExampleFrame" url="#{ui5ExamplePlugin.url}" width="100%" height="100%"
scrolling="AUTO" />
        </sap:panelGroup>
      </sap:panelGrid>
    </sap:panel>
  </sap:panelGroup>

  </sap:panelRow>

  <sap:panelRow cellHalign="start" cellValign="top">
    <sap:panelGroup id="messageTextArea" backgroundDesign="transparent" width="100%" height="100%"
      halign="start" valign="top" padding="10px 10px 0px 10px">
      <sap:inputTextarea id="messageText" readOnly="true" width="100%" height="100%"
value="#{ui5ExamplePlugin.message}" />
    </sap:panelGroup>
  </sap:panelRow>

  <sap:panelRow cellHalign="center" cellValign="top">
    <sap:panelGroup backgroundDesign="transparent" width="100%" padding="4px 0" halign="middle"
valign="top">
      <sap:commandButton id="ui5CloseButton" value="#{ui5ExamplePlugin.closePluginLabel}"
title="#{gapi18nTransformer['toolTip.close.TEXT']}"
image="#{icon:deviceDependantUrl('cancel')}"
rendered="#{ui5ExamplePlugin.closePluginEnabled}"
action="#{ui5ExamplePlugin.closePlugin}" />
    </sap:panelGroup>
  </sap:panelRow>
</sap:panelGrid>
</sap:panel>

</f:subview>
```

5.2.9.2.3 UI5 Example Page

The UI5 page (“ui5example.jsp”) is a JSP that renders and displays the UI5 view. The following example shows what to include to perform communication with the UI5 example plugin and how to get and use globalization APIs. It also shows how to implement domain relaxing into the page to prevent cross-domain attacks.

```
<%@ page import="java.util.*,
com.sap.me.coral.core.util.WebUtils,
com.sap.me.wpmf.util.FacesUtility,
com.google.gson.Gson" %>
```

```
<%
String pluginTitle = FacesUtility.getLocaleSpecificText("trackingControl.default.LABEL");

Map<String,String> i18nMap = new HashMap<String,String>();
i18nMap.put("search.default.BUTTON", FacesUtility.getLocaleSpecificText("search.default.BUTTON"));
i18nMap.put("save.default.BUTTON", FacesUtility.getLocaleSpecificText("save.default.BUTTON"));
i18nMap.put("retrieve.default.BUTTON", FacesUtility.getLocaleSpecificText("retrieve.default.BUTTON"));
i18nMap.put("DATA.default.LABEL", FacesUtility.getLocaleSpecificText("DATA.default.LABEL"));
i18nMap.put("DESCRIPTION.default.LABEL", FacesUtility.getLocaleSpecificText("DESCRIPTION.default.LABEL"));
i18nMap.put("CURRENT_REVISION.default.LABEL",
FacesUtility.getLocaleSpecificText("CURRENT_REVISION.default.LABEL"));
i18nMap.put("trackingControl.default.LABEL", pluginTitle);
Gson gson = new Gson();
String i18nMapJson = gson.toJson(i18nMap);
%>

<!DOCTYPE HTML>
<html>
<head>
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta http-equiv='Content-Type' content='text/html; charset=UTF-8'/>

<title><%=pluginTitle%></title>

<script
src="<%=WebUtils.buildContextPath(request,response,"/com/sap/me/coral/core/view/component/javascript/Doma
nRelax.js")%>" language="JavaScript">
</script>

<script
src="<%=WebUtils.buildContextPath(request,response,"/com/sap/me/production/podclient/iframePluginCommunica
tion.js")%>" language="JavaScript">
</script>

<script src="/sapui5/resources/sap-ui-core.js"
id="sap-ui-bootstrap"
data-sap-ui-libs="sap.m"
data-sap-ui-theme="sap_bluecrystal">
</script>

<link rel="stylesheet" type="text/css" href="ui5example.css">

<script>

// required for communication to POD JavaScript functions
DomainRelax.relax();

var sI18nMapJson = '<%=i18nMapJson%>';
window.ol18nTextMap = {};
if (sI18nMapJson && sI18nMapJson.length > 0) {
    window.ol18nTextMap = parseStringToJSON(sI18nMapJson);
}

sap.ui.localResources("ui5exampleviews");
var app = new sap.m.App({initialPage:"exampleview1"});
```

```
var page = sap.ui.view({id:"exampleview1", viewName:"ui5exampleviews.ui5example",
type:sap.ui.core.mvc.ViewType.XML});
app.addPage(page);
app.placeAt("content");

// save reference for later use
window.ui5Application = app;

/* called to receive messages from POD */
function receiveMessageFromPod(sMessage, oMessageData) {

    var sMsg = sMessage;
    var oMdata = oMessageData;

    jQuery.sap.delayedCall(100, this, function() {

        if (!window.ui5Application) {
            return;
        }

        var oView = window.ui5Application.getPage("exampleview1");
        if (!oView) {
            return;
        }

        var oController = oView.getController();
        if (!oController) {
            return;
        }

        oController.processMessageData(sMsg, oMdata);

    });
};

function processUnload() {
    if (window.ui5Application) {
        var oView = window.ui5Application.getPage("exampleview1");
        if (oView) {
            var oController = oView.getController();
            if (oController) {
                oController.onExit();
            }
        }
    }
    window.closeIFramePlugin();
};

function getLocaleSpecificText(sKey) {
    if (!window.ol18nTextMap || !sKey || sKey === "") {
        return sKey;
    }
    var sValue = window.ol18nTextMap[sKey];
    if (!sValue || sValue === "") {
        return sKey;
    }
    return sValue;
}
```

```
    };  
  
</script>  
  
</head>  
<body class="sapUiBody" onload="window.processUnload();" style="margin: 0; width: 100%; height: 300px;"  
role="application">  
  <div id="content"></div>  
</body>  
</html>
```

Below is the example style sheet (“ui5example.css”) that is referenced above.

```
html, body {  
  height: 300px !important;  
  width: 100%;  
  overflow: auto;  
}  
.inputUpperCase {  
}  
.inputUpperCase > .sapMInputInner {  
  text-transform: uppercase;  
}  
.inputUpperCase > .sapMInputBaseInner {  
  text-transform: uppercase;  
}
```

5.2.9.2.4 UI5 Example View

The UI5 example view in this case is defined as an XML view (“ui5exampleviews/ui5example.view.xml”). The view is defined in a child folder from the main UI5 page (“ui5exampleviews”). You can use other methods to define the view (JavaScript, HTML, or JSON) but UI5 recommends that you use XML views because you must separate controller logic from the view definition.

```
<mvc:View xmlns:core="sap.ui.core"  
  xmlns:mvc="sap.ui.core.mvc"  
  xmlns:l="sap.ui.layout"  
  xmlns:f="sap.ui.layout.form"  
  xmlns="sap.m"  
  controllerName="ui5exampleviews.ui5example"  
  xmlns:html="http://www.w3.org/1999/xhtml">  
  
  <Page id="pageComponent" title="UI5 Plugin" showNavButton="false">  
  
    <headerContent>  
      <Button icon="sap-icon://settings" />  
      <Button icon="sap-icon://drop-down-list" />  
    </headerContent>  
  
    <content>  
      <l:FixFlex vertical="true" fixFirst="true" fixContentSize="25%">  
        <l:fixContent>  
          <Panel id="searchBar">
```

```
<headerToolbar>
  <Toolbar>
    <ToolbarSpacer />
    <Input id="searchData" width="300px" placeholder="Search Data" />
    <core:HTML content='&lt;pre&gt;    &lt;/pre&gt;' />
    <Button id="retrieveButton" text="Retrieve" press="onRetrievePress"/>
    <Button id="saveButton" text="Save" press="onSavePress"/>
    <ToolbarSpacer />
  </Toolbar>
</headerToolbar>
</Panel>
</l:fixContent>
<l:flexContent>
  <Panel id="mainPanel" applyContentPadding="true">
    <FlexBox
      alignItems="Start"
      justifyContent="Start">
      <items>
        <l:VerticalLayout>
          <Input id="inputData" width="250px" placeholder="Data" value="" />
          <Input id="inputDescription" width="250px" placeholder="Description" value="" />
          <CheckBox id="checkBoxVersion" text="Current Version" selected="true" />
        </l:VerticalLayout>
      </items>
    </FlexBox>
  </Panel>
</l:flexContent>
</l:FixFlex>

</content>
</Page>
</mvc:View>
```

5.2.9.2.5 UI5 Example Controller

The UI5 example controller (“ui5exampleviews/ui5example.controller.js”) that is defined for the view is shown below.

```
sap.ui.controller("ui5exampleviews.ui5example", {

  onInit: function () {
  },

  onBeforeRendering: function() {

    var oView = this.getView();
    if (!oView) {
      return;
    }

    var oComponent = oView.byId("searchData");
    if (oComponent) {
      oComponent.addClass("inputUpperCase");
    }
  },

  onAfterRendering: function() {
```

```
var oView = this.getView();
if (!oView) {
    return;
}
var oComponent = oView.byId("pageComponent");
if (oComponent) {
    oComponent.setTitle(getLocaleSpecificText("trackingControl.default.LABEL"));
}
oComponent = oView.byId("searchData");
if (oComponent) {
    oComponent.setPlaceholder(getLocaleSpecificText("search.default.BUTTON"));
}
oComponent = oView.byId("retrieveButton");
if (oComponent) {
    oComponent.setText(getLocaleSpecificText("retrieve.default.BUTTON"));
}
oComponent = oView.byId("saveButton");
if (oComponent) {
    oComponent.setText(getLocaleSpecificText("save.default.BUTTON"));
}
oComponent = oView.byId("inputData");
if (oComponent) {
    oComponent.setPlaceholder(getLocaleSpecificText("DATA.default.LABEL"));
}
oComponent = oView.byId("inputDescription");
if (oComponent) {
    oComponent.setPlaceholder(getLocaleSpecificText("DESCRIPTION.default.LABEL"));
}
oComponent = oView.byId("checkBoxVersion");
if (oComponent) {
    oComponent.setText(getLocaleSpecificText("CURRENT_REVISION.default.LABEL"));
}

// send message stating view has been rendered
if (window.sendMessageToPod) {
    window.sendMessageToPod("afterRendering", null);
}

},

onExit: function() {
    // add any code to free resources / listeners here
},

processMessageData : function (sMessage, oMessageData) {

    var oView = this.getView();
    if (!oView) {
        return;
    }

    for (var sKey in oMessageData){
        var sValue = oMessageData[sKey];

        var oControl = oView.byId(sKey);
        if (oControl) {
            if (oControl.setValue) {
                oControl.setValue(sValue);
            }
        }
    }
}
```

```
    } else if (oControl.setSelected) {
        if (sValue.toLowerCase() === "true") {
            oControl.setSelected(true);
        } else {
            oControl.setSelected(false);
        }
    }

    } else if (oControl.setText) {
        oControl.setText(sValue);
    }

    } else {
        jQuery.sap.log.error("ui5example: oConrol type not implemented for id = " + sKey);
    }
}
},
onRetrievePress : function (oEvent) {

    var oView = this.getView();
    if (!oView) {
        jQuery.sap.log.error("onRetrievePress: view for controller not defined");
        return;
    }

    var oSearchField = oView.byId("searchData");
    if (!oSearchField) {
        jQuery.sap.log.error("onRetrievePress: 'searchData' control not found");
        return;
    }
    var sFilter = oSearchField.getValue();
    if (sFilter) {
        sFilter = sFilter.toUpperCase();
    }

    // send retrieve message
    if (window.sendMessageToPod) {
        var oMessageData = {
            "searchData" : sFilter
        };
        window.sendMessageToPod("retrieve", oMessageData)
    }
},

onSavePress : function (oEvent) {

    var oView = this.getView();
    if (!oView) {
        jQuery.sap.log.error("onRetrievePress: view for controller not defined");
        return;
    }

    var oMessageData = {};

    var oControl = oView.byId("inputData");
    if (oControl) {
        oMessageData["inputData"] = oControl.getValue();
    }
}
```

```

    }

    oControl = oView.byId("inputDescription");
    if (oControl) {
        oMessageData["inputDescription"] = oControl.getValue();
    }

    oControl = oView.byId("checkBoxVersion");
    if (oControl) {
        if (oControl.getSelected()) {
            oMessageData["checkBoxVersion"] = "true";
        } else {
            oMessageData["checkBoxVersion"] = "false";
        }
    }
}

// send save message
if (window.sendMessageToPod) {
    window.sendMessageToPod("save", oMessageData)
}

});

```

5.2.10 GAPI Support

The objective is to use the Manufacturing Execution GAPI for i18nization. This is achieved via an interceptor to the standard ResourceBundle. The GapiI18NTransformer delegates to use the GAPI i18n mechanism.

For any control on the jsp page the expression "#{gapiI18nTransformer['resource key']}" can be used to derive i18nized text. The key has to be present in LocaleSpecificText properties file.

5.2.11 Converters

WMPF adds two new converters to the list of available JSF converters. The first converter is a Date / Time converter that supports the Manufacturing Execution Globalization API and the DateTimeInterface type; the second is a String transformation converter that is based on GAPI Text services.

Date Time Converter		
Tag Name	<gapiDateTimeConveter>	
Tag Library URI	http://java.sap.com/jsf/html/sapme	
Class Name	com.sap.me.wpmf.convert.GapiDateTimeConverter	
Converter Id	wpmf.GapiDateTimeConverter	
Attributes	Name	Valid Values

	formatType*	date, time
	dateStyle	full, long, medium, short
	timeStyle	full, long, medium, short
	Pattern*	Anything supported by DateFomat (e.g., MM/DD/YY)

* The pattern and formatType attributes are mutually exclusive with the formatType taking precedence.

Text Converter		
Tag Name	<gapiDateTextConveter>	
Tag Library URI	http://java.sap.com/jsf/html/sapme	
Class Name	com.sap.me.wpmf.convert.GapiTextConverter	
Converter Id	wpmf.GapiTextConverter	
Attributes	Name	Value
	textStyle	upper, lower, title, normalize

The standard JSF convention is used for using these converters and can be added to the appropriate components by either nesting the converter tag within the parent component tag or programmatically adding it to the component. These converters can be used by either specifying the tag names and attributes values above or by using the <converter> tag in conjunction with converter id.

```
<%@ taglib prefix="me" uri="http://java.sap.com/jsf/html/sapme" %>

<h:outputText value="#{testConverterPlugin.dateVar}">
  <me:gapiDateTimeConverter formatType="date" dateStyle="full"/>
</h:outputText>
```

Or

```
<h:outputText value="#{testConverterPlugin.dateVar}">
  <f:attribute name="formatType" value="date" />
  <f:attribute name="dateStyle" value="full" />
  <f:converter converterId="wpmf.GapiDateTimeConverter" />
</h:outputText>
```

The preferred way is to use the specific converter tag and define the attribute values on the converter itself rather than using the converter id and the attaching the attributes to the component. However, if the attribute values on the converter cannot be preserved across requests, then using the converter id will eliminate the issue.

5.2.12 Data Table Support

WPMF supports the Table and Hierarchical Tree Table components provided by the LSF library belonging to the extended namespace. Both components support all Manufacturing Execution core requirements but have the following limitations:

- Fixed row height unique for all rows. The row height is not adjustable.
- Wrapping within table cells is not supported.
- No col span or row span of table cells possible
- No table popins possible
- No Paginator available, only scrolling.

The table components provide support for the following features:

- Single and multiple row selection including range selection.
- Server side vs. client side selection processing.
- Enabling/disabling row selection at the table or row level.
- Horizontal and vertical scrolling.
- Fixed height vs. content fill mode (adjustable height based on its container).
- Smart table paging to avoid loading all rows to the client side.
- Column sorting.
- Column resizing.
- Column reordering.
- Row double-click action.
- Inherently designed to prevent XSS and XSRF attacks.
- Support SAP accessibility standard.

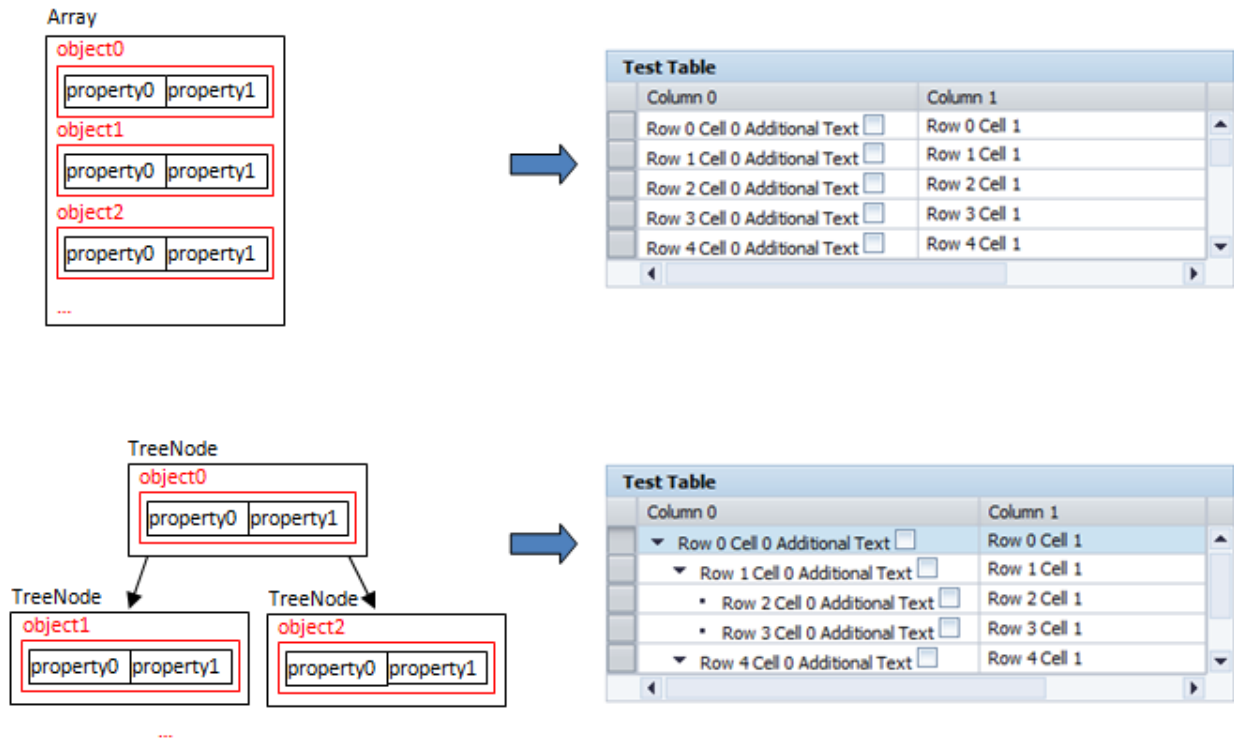
In order to simplify and provide a unified way for configuring tables, WPMF provides a JSF backing bean that can be bound to the table component.

Table(s) are defined within plug-ins and each plug-in can have one or many tables as well as other components. The table itself is defined using the SAP implementation of JSF tag library within a JSP page representing a plug-in as follows:

```
<%@ taglib prefix="sap" uri="http://java.sap.com/jsf/html/extended"%>
<sap:dataTable binding="#{exampleConfigurator.table}"
    value="#{examplePlugin.worklist}" var="row" first="0" rows="5" height="100%"
    id="example_table">
</sap:dataTable>
```

In order to use a hierarchical tree table, simply use the SAP component tag for it and use a supported Data Model, as defined by the ‘value’ attribute, that must be of hierarchical nature consisting of nodes.

```
<sap:treeDataTable binding="#{exampleConfigurator.table}"
    value="#{examplePlugin.myTreeModel}" var="row" first="0" rows="5"
    height="100%" id="example_table">
</sap:treedataTable>
```



The ‘var’ attribute is the name of the variable holding a row of data and is used by JSF during the rendering process. The value of this attribute *must* always be set to “row” at this point while we are working to remove this restriction in the future. The ‘value’ attribute is the actual DataModel behind the table whose individual row bound to the ‘var’ attribute at runtime.

WPMF requires that the object representing a row within the DataModel must provide a meaningful implementation of its equals() and hashCode() methods.

Please refer to SAP tag library documentation at “./sap-extended/dataTable.html” and “./sap-extended/treeDataTable.html”

The ‘binding’ attribute is a backing bean java object that defines the table/hierarchical table configuration. This object must be of class **com.sap.wpmf.TableConfigurator** or a subclass of. This backing bean creates a new instance of the table component and sets it up with default values and options that can be modified using the appropriate [faces-config.xml](#) or through its API programmatically. There will be one of these beans for each table defined within a plugin with bean names being unique and representing that instance only.

The POD also provides a subclass of the wpmf configurator that integrates and uses List Maintenance data using the category and name of the list. This configurator adds the following three properties that correspond to the List Maintenance key attributes above:

Configurator property	List Maintenance Attribute
-----------------------	----------------------------

listType	category
listName	name
listKind (derived)	type

The listType and listName properties can be defined within the [faces-config.xml](#) or within the plugin inside the injector method. The listKind property is a read only and derived property that is obtained based on the listType and listName.

This configurator bean must be available to the plugin so that the plugin can dynamically define or modify table behavior. Therefore, the configurator needs to be injected into your plugin using standard JSF injection methodology for managed beans.

The following excerpts show how a typical table can be defined within wpmf.

JSP Page:

```
<sap:panel id=" tableScroller " isCollapsible="false" contentAreaDesign="transparent"
borderDesign="none" width="100%" height="100%" >

    <sap:dataTable binding="#{operationListConfigurator.table}"
        width="100%" height="100%" rows="20"
        value="#{operationListPlugin.model.operations}" var="row" first="0"
        id="operationlist_table" >
    </sap:dataTable>

</sap:panel>
```

Faces-config.xml:

```
<managed-bean>
    <managed-bean-name>operationListPlugin</managed-bean-name>
    <managed-bean-class>
        com.sap.me.production.podclient.impl.OperationListPlugin
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>configBean</property-name>
        <value>#{operationListConfigurator}</value>
    </managed-property>
</managed-bean>

<managed-bean>
    <managed-bean-name>operationListConfigurator</managed-bean-name>
    <managed-bean-class>
        com.sap.me.production.podclient.PodClientTableConfigurator
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>beanName</property-name>
        <value>operationListConfigurator</value>
    </managed-property>
</managed-bean>
```

The Plugin Class:

```
public class OperationListPlugin extends BasePodPlugin implements
    InternalTableSelectionEventListener, RefreshOperationListListenerInterface,
    WorklistChangedListener, WorklistSelectionChangeListener,
    CompositeEventListenerInterface {

    private TableConfigurator configBean = null;

    /**
     * @return the configBean
     */
    public TableConfigurator getConfigBean() {
        return configBean;
    }

    /**
     * @param configBean the configBean to set
     */
    public void setConfigBean(TableConfigurator configBean) {
        this.configBean = configBean;
        configBean.setMultiSelectType(true);
        if (configBean.getListName() == null) {
            LayoutManagerInterface lm = getLayoutManager();
            configBean.setListType(PodOptionType.POD_OPERATION.value());
            configBean.setListName(lm.getPodOption(PodOptionType.POD_OPERATION));
        }
        for (String colName : configBean.getListColumnNames()) {
            if (colName == null || colName.trim().length() == 0) continue;
            if (colName.indexOf("DATE") >= 0) {
                GapiDateTimeConverter conv = new GapiDateTimeConverter();
                conv.setFormatType("both");
                conv.setDateStyle("short");
                conv.setTimeStyle("short");
                configBean.setColumnConverter(conv, colName);
            }
        }
        olpsi.setIconGroupRendering(configBean);
        configBean.addInternalTableSelectionEventListener(this);
    }
}
```

Although it is possible and allowed to write custom configurators by extending the core TableConfigurator class, it is highly recommended not to do so because using the injector method above along with definition in [faces-config.xml](#) should be adequate for almost all circumstances. Please avoid extending the TableConfigurator unless you have requirements that absolutely mandate it.

The TableConfigurator API's JavaDoc fully describes all the features that are exposed for configuring a table. However, there are a few that may need a more detailed description which are described next.

5.2.12.1 Column Bindings

<property-name>columnBindings</property-name>

The explicit binding between the column names and the user defined data object properties defining a table row. The names must be unique within a given table and can be thought of as key names for columns. If the POD production version of the configurator is used, these names will be the same as the ones coming back from the selected list within the list maintenance module.

Any column names that do not have a binding defined will be skipped and will not show as a column. These binding are defined as a map whose key/values can be defined programmatically or using a `<map-entry>` tag in the `faces-config.xml`. The `<value>` definition in the `<map-entry>` defines the object property name as well as an optional locale specific text key that is used to find the column title as shown below:

```
<map-entry>
  <key>SOME_COLUMN</key>
  <value>someProperty;SOME_COLUMN.awl.LABEL</value>
</map-entry>
```

If you do not define an optional title key or if the key entered is not found the `TableConfigurator` will then search for the title using the following keys until a match is found. If no match is found for the following keys, the column name will be returned as the title:

```
someColumn.default.LABEL
SOME_COLUMN.default.LABEL
some_column.default.LABEL
```

You can optionally define any valid bundle to be used in the following format:

```
<map-entry>
  <key>SOME_COLUMN</key>
  <value>someProperty;@{localSpecificText['A valid bundle key']} </value>
</map-entry>
```

Since this specific notation is using the system bundle, it is equivalent to using the label keys above.

This gives you the option of using other valid user defined bundles for keys that should not be in the system bundle. An example can be local specific labels for custom fields.

Custom fields attached to SAP ME will know objects such as SFC, SHOP ORDER, etc. can also be displayed as additional columns. The definition of these additional columns is automatically added if the table is configured to use List Maintenance definitions, otherwise it is up to the plug-in writer to specify them manually. The values for the custom fields should come from a value binding similar to the ones defined above. It is recommended to use a `Map<String, String>` to hold the name-values within the DTO that is bound to the table due to the inherent ability of JSF to read Maps as part of a binding. This is just a recommendation and the plug-in writer is free to use other appropriate data structures. The example below assumes a property defined [customData](#)

whose data type is a map that holds the values for custom data fields and one of them in this example is called `SFC_CUSTOM.Field1`:

```
<map-entry>
  <key>SFC_CUSTOM.Field1</key>
  <value>
    customData['SFC_CUSTOM.Field1'];@{SomeBundle['A valid bundle key']}
  </value>
</map-entry>
```

5.2.12.2 Table Row Selections

Table row selections for the whole table can be enabled or disabled by setting the property `allowSelections` to true or false. If this property is set to 'true' the `multiSelectType` property can be set to true or false to enable a single vs. a multi row selection model. Row selection mechanism can also be enabled or disabled based on the identity of a given object within the table dynamically.

Table selection processing can happen either in 'server' or 'client' mode which is controlled by the `selectionBehavior` property. The server mode causes an immediate roundtrip after the selection while the client mode requires a user initiated explicit roundtrip through a UI interface such as a button. Most plugins are designed with a 'server' selection behavior. Browsers are usually designed with a 'client' selection behavior where clicking the 'OK' button initiates the roundtrip.

Double click action on a given table row is also supported by setting the `doubleClick` property to true or false. If this property is set to true, then an action listener needs to be defined by setting the `selectionActionExpression` property value to a valid method expression syntax. Double click actions are only supported in the 'client' processing mode.

If selections are allowed, your plug-in must implement the **InternalTableSelectionEventListener** interface in order to receive table selections. Additionally, your plugin needs to add itself as a listener to the **TableConfigurator** instance it is working with somewhere within the injection process.

5.2.12.3 Table Column Sorting

Table and individual table columns can be explicitly defined as sort-able. If sorting is allowed at the table level, then you can turn sorting on or off on individual columns. If a column is defined as sortable, then clicking on that column header generates a **SortActionEvent** that can be handled by the plugin if the plugin implements the **TableColumnSortListener** interface and has added itself as the listener to the **TableConfigurator**.

It is up to the plugin implementation to handle the sort event and implement the actual sort of the data as appropriate. WPMF does provide a `TableSortComparator` that can optionally be used to perform the sorting as a convenience.

5.2.12.4 Table Column Widths

Table column widths can be explicitly defined either within the faces-config.xml using the “[columnwidths](#)” property or through the API calls that sets the column width for individual columns using their column name/key. The column widths can be defined in percentages (e.g. 10%), pixels (e.g. 150) or em (e.g. 10em).

Specifying the width in pixels or em ensures that the column(s) are rendered using that width. The remainder of columns will be evenly distributed in the available remaining table width. If all or a subset of columns are specified by pixels or em and the sum of widths exceeds the available width, then a horizontal scroll bar is shown. If all are not defined as pixels or em and the sums of the width for the defined ones or subset of exceeds the available width, then some columns may disappear and only can be seen if the table width is expanded, no scroll bar is displayed.

```
<managed-property>
  <property-name>columnWidths</property-name>
  <property-class>java.util.HashMap</property-class>
  <map-entries>
    <key-class>java.lang.String</key-class>
    <value-class>java.lang.String</value-class>
    <map-entry>
      <key>SFC</key>
      <value>15em</value>
    </map-entry>
    ...
  </map-entries>
</managed-property>
```

5.2.12.5 Column Cell Editors and Renderers

Table columns can define other components to be used as their cell editor and cell renderer. A good example of cell editor is a plain input field or one that allows lookups. An example of a cell renderer is icon group that allows the insertion of multiple actionable icons in a cell. Please see **[com.sap.wpmf.IconGroup](#)** for more information.

A column must be explicitly defined as editable before a cell editor component can be assigned to it. A cell editor or renderer component can only be defined using API calls and it should belong or be based on SAP components that are defined in the extended namespace. The table does not support components from internal namespace and using them is not recommended.

```
ArrayList<String> editCols = new ArrayList<String>();
```

```
// make these columns editable
editCols.add("WORK_CENTER");
editCols.add("ACTIVE");
editCols.add("DUE_DATE");
configBean.setEditableColumns(editCols);
```

```
configBean.setCellEditor("DUE_DATE", new com.sap.ui.faces.component.sap.HtmlInputDate());
configBean.setCellEditor("ACTIVE", new HtmlCommandBooleanCheckbox());
configBean.setCellEditor("SHOP_ORDER", new UICommandInputText());

//use the IconGroup as the cell renderer for the multi icon column.
IconGroup iconGroup = new IconGroup();
...
// configure icon group and set it as the cell renderer for column.
...
configBean.setCellRenderer("ICON_HOLDER", iconGroup);
```

5.2.12.6 Column Converters and Validators

You can define any valid JSF converter for a given column to automatically convert string values to the appropriate back end java property type through the provided API call. One or compliant JSF Validators may be assigned to a given column programmatically as well.

```
// define converters for the DUE_DATE and Active columns
DateTimeConverter dateTimeConv = new DateTimeConverter();
dateTimeConv.setType("date");
dateTimeConv.setDateStyle("short");
dateTimeConv.setTimeStyle("short");
configBean.setColumnConverter(dateTimeConv, "DUE_DATE");
configBean.setColumnConverter(new BooleanConverter(), "ACTIVE");
```

5.2.12.7 Table Row Details Column

A “Details” column containing the “show details icon” becomes visible when the property “rowDetailActivityId” is set to a valid Activity ID that needs to be invoked upon clicking on the icon. The system does not make any assumptions about the validity of the activity or what the specific implementation of the activity is. The system simply generates a “DisplayTableRowDetailEvent” that is sent to all the registered plug-in(s) that implement the “DisplayTableRowDetailListener” interface.

The event contains the current object associated with the row on the table and the activity id that was defined. It is the responsibility of the plugin writer to handle this event in order to execute the activity.

If the property “rowDetailActivityId” is set to a not null and not empty value, the Details column will get rendered as the last column regardless of the validity of this activity or whether it is defined in the system or not. This means that the system will generate the **DisplayTableRowDetailEvent** and the plug-in that implements the **DisplayTableRowDetailListener** interface and is listening can add any custom code in the event handler.

An alternative approach is to define button(s) as the Cell Renderer for the desired column(s) and provide an action handler for it. This approach allows you to have multiple details columns where desired.

5.2.13 Browse Support

ME Browse capability is handled using the LSF Command Input Field component. This section describes how browse is supported by the framework. There are two methodologies for supporting browse. The first describes how browse was implemented in ME 6.0 and still is used in many current plugins (Deprecated Browse). The second section describes the way all new browses going forward should be implemented. The SAP standard shortcut key to invoke Browse is F4.

5.2.13.1 Deprecated Browse

5.2.13.1.1 Browse Approach

Browse is activated from an action listener assigned to the LSF input field. The ME Browse module API provides the utility classes and top level BrowseServiceInterface that is used to get the browse results. The framework invokes the Browse via the BrowseServiceInterface. The BrowseService's parent is Prototype as this is strictly internal. The Service browse method expects a BrowseDTO that will contain the required parameters for a successful browse (search field, browseid etc). The entire form entries are being supplied to the backend as is the case with existing Web framework Browse.

The base Plugin class's browseActionListener creates the browseOpenEvent with the required browse parameters and fires the Browse open Dialog. The Browse Plugin eventually invokes the BrowseService for getting the Browse Results.

BrowseService delegates to BrowseUtility which performs required DTO to ApplicationVO conversion and calls the corresponding Browse.

Note: For new Browses, the above round-trip is not required as we can work with the BrowseDTO directly. This feature has to be coded when we encounter a new Browse within the WPMF.

The current Browse framework implementation is currently being reviewed so this very possibly will change in the near future. The next sections will describe how to implement Browse as it is used today.

5.2.13.1.2 Adding Browse to View

Use the <sap:commandInputText> or <h:commandLink> tag to assign browse to.

To leverage framework level Browse mapping to automatically populate the parent/invoking fields with the Browse results; use the following convention for the setter methods on the Plugin's that host the Browse:

set + "First Character of the Browse Field's Id" + "rest of the Browse Field's Id characters in lowercase"

This standard mapping should cover most of the Browse variants. Plugin writers can override the "processTableSelectionEvent" method to dictate how they want to handle the Browse mapping for any custom needs.

The following Browse specific attributes are required. Note that these attributes are similar to the Browse attributes that are to be defined in the current Web Framework.

Browse Attribute	Default Value	Description
browseable	false	Enables browse mode
browseId		Specifies BrowseID to be invoked (i.e.; OPERATION_CUSTOM)
browseCustom		Specified if the browse is "standard" or "custom"
browseSelectionModel	single	Specify if the Browse selection model is "single" or "multi"
browseTitle		Resource bundle key, String or expression to bean method that will return title.
browseDialogWidth	44em	Width to make browse dialog
browseDialogHeight	30em	Height to make browse dialog
browsePlugin		Optional name of managed bean to handle the browse. The managed bean should extend AbstractBrowsePlugin.
legacyBrowse	true	Specifies if the Browse expects ApplicationVO(legacy browse) vs. new service API DTO.

The "actionListener" attribute has to be specified to handle the Browse invocation when either the icon is clicked or F4 key pressed. It is strongly recommended NOT to override the WMPF Plugin class's Browse Action Listener method (browseActionListener) to leverage the standard default behavior.

Example: `actionListener="#{examplePlugin.browseActionListener}"`

Below is an example using the <sap:commandInputText> control for an operation browse in an example plugin.

```
<sap:commandInputText id="Operation" width="16em"
    submitOnChange="false" submitOnFieldHelp="true"
    value="#{examplePlugin.operationText}"
    actionListener="#{examplePlugin.browseActionListener}">

    <f:attribute name="upperCase" value="true" />
    <f:attribute name="legacyBrowse" value="true" />
    <f:attribute name="browseable" value="true" />
    <f:attribute name="browseId" value="OPERATION_CUSTOM" />
    <f:attribute name="browseCustom" value="custom" />
    <f:attribute name="browseSelectionModel" value="single" />
    <f:attribute name="browseDialogWidth" value="20em" />
    <f:attribute name="browseDialogHeight" value="25em" />

    <sap:ajaxUpdate render="#{sap:toClientId('exampleContainer')}" />
</sap:commandInputText>
```

The following example shows the same browse using a <h:commandLink> control:

```
<h:commandLink id="Operation"
    value="#{examplePlugin.operationText}"
    actionListener="#{examplePlugin.browseActionListener}">

    <f:attribute name="upperCase" value="true" />
    <f:attribute name="legacyBrowse" value="true" />
```

```
<f:attribute name="browseable" value="true" />
<f:attribute name="browseId" value="OPERATION_CUSTOM" />
<f:attribute name="browseCustom" value="custom" />
<f:attribute name="browseSelectionModel" value="single" />
<f:attribute name="browseDialogWidth" value="20em" />
<f:attribute name="browseDialogHeight" value="25em" />

<sap:ajaxUpdate render="#{sap:toClientId('exampleContainer')}" />
</h:commandLink>
```

5.2.13.1.3 Browse in Plugin class

Ensure that your Plugin class implements `BrowseSelectionEventListener`.

The base Plugin class “`processBrowseSelectionEvent()`” method will handle automatic mapping of selected Browse results onto the Plugin fields, provided the following conventions are followed for the Browse related field backing bean setter methods :

set + “First character of the browse field’s Id” + “rest of the browse field’s Id characters in lowercase”

For example if the browse input field column name is defined as `OPERATION`, the setter method is `setOperation(String operation)`. In this method, set the backing bean property accordingly so that the UI is rendered and the browse field gets updated with the selected values.

```
public void setOperation(String operation) {
    setOperationText(operation);
}
```

Subclass plugins should override the above behavior for any special processing as well as for Browsers that involve multi-select. It is recommended to use the framework provided mapping for default cases.

5.2.13.1.4 Triggering new Browsers

New Browsers use a DTO based Service API invocation; Legacy browses use `ApplicationVO`.

Use the attribute `legacyBrowse="false"` for the input field tag in the jsp page. This will ensure that a DTO to `ApplicationVO` conversion does not take place. Be sure to override the below method (in the Browse class) in your new Browse sub class to provide the Service API implementation details:

```
public BrowseDTO browse(BrowseDTO browseDTO) throws BasicBOBeanException
```

Note: Since a vast majority of ME Browsers are legacy based, the default is set to `legacyBrowse`, and therefore, a conversion of `BrowseDTO` to `ApplicationVO` takes place in the `BrowseUtility` class.

This behavior will be changed eventually once we convert our legacy browses to use the Service API. Until then, the input field’s `legacyBrowse` attribute should be set to “false” for any new browses.

5.2.13.1.5 Browse Callbacks

If specified, WPMF accepts and performs a callback method execution after a Browse dialog is closed. This can be useful in the following scenarios:

- A Plugin has to perform specific processing after a Browse Selection is made on a browse field.
- A Plugin might need to enable or render other fields depending on the Browse value selected.

To specify a callback if the browse input field is bound to a backing bean do this:

```
browseField.getAttributes().put("browseCallBack", "examplePlugin.processCallBack");
```

If the Browse input field is defined in the jsp, then specify the following attribute in addition to your other browse attributes:

```
<f:attribute name="browseCallBack" value="examplePlugin.processCallBack"/>
```

The convention for the callback method is pluginId.callbackMethodName where pluginId is the name assigned to the plugin in the faces-config.xml file.

Make sure that on the callback method the Plugin is added to the delta refresh using addtoAjaxUtils (plugin), if applicable.

In case you override the Plugin's browseActionListener method, then ensure that you add the callback attribute to the browseParamMap in your overriding method.

```
if (event.getComponent().getAttributes() != null &&
    event.getComponent().getAttributes().get("browseCallBack") != null) {

    browseParamMap.put("browseCallBack", event.getComponent().getAttributes().get("browseCallBack"));
}
```

5.2.13.2 New Browse

Browse Orchestration Framework was developed to provide SAP ME a consistent approach to create and reuse browse configurations. Later it will be used to display browse window with collection of filtered objects.

Initial goal of Browse Orchestration Framework is to address the following use cases:

- Use PAPI Service Method as a finder of objects for browse window in WPMF;
- Use PAPI Service Method as a finder of objects for browse window in old-web applications;
- Reuse legacy browses in WPMF;

5.2.13.2.1 Modules and Packages

Base classes are located in me.common.api , me.common.web and me.common.client modules. This location gives access to them from all other client modules.

Browse Configurations are defined as services. This makes available reuse of browse configurations in other client's modules. Because browse configurations are client-centric, the service configurations must be declared in client's sub-module of corresponding functional module.

5.2.13.2.2 BrowseServiceInterface

Browse Configurations must implement `com.sap.me.common.client.BrowseServiceInterface`:

```
/**
 * Interface of browse service that will return {@link BrowseResponse}
 * with information needed to display browse window and
 * list of {@link BrowseItem}, based on the criteria in {@link BrowseRequest}
 */
public interface BrowseServiceInterface {
    /**
     * Method returns {@link BrowseResponse} with list of {@link BrowseItem},
     * based on the criteria in {@link BrowseRequest}.
     * @param {@link BrowseRequest}, containing the criteria
     * @return {@link BrowseResponse}, containing the information required
     *         to display browse window
     * @throws BusinessException
     */
    public BrowseResponse browse (BrowseRequest request) throws BusinessException;
}
```

This interface provides a method that returns a `BrowseResponse`, containing all information needed. The `com.sap.me.common.api.BrowseResponse` object contains enough information to setup the Browse Window UI header, list of columns along with the column headers and the list of `BrowseItems`. The `com.sap.me.common.api.BrowseItem` object contains enough information to display one record in the browse collection. The Primary and Revision fields are framework-level fields needed to populate the browsable field with dependant revision field. The `com.sap.me.common.api.BrowseRequest` object defines a list of `AttributeValue` objects defining the criteria that is eventually translated to API Service Method request.

5.2.13.2.3 Implementations of BrowseServiceInterface

Two implementations of `BrowseServiceInterface` are provided to meet the required use cases; the `BrowseService` and the `LegacyBrowseService`.

The main purpose of `com.sap.me.common.client.BrowseService` is to provide the base functionality for calling the PAPI Service Method and transform returned objects into list of `BrowseItem`'s. More detailed use of this implementation is described below.

The `com.sap.me.browse.LegacyBrowseService` is created to support calls to legacy browse classes instantiated through the `com.sap.me.browse.BrowseFactory`. Implementation of `LegacyBrowseService` is responsible for transformations between `ApplicationVO` and `BrowseResponse` in order to support usage of legacy browses.

These two implementations address two major use cases; “Use PAPI Service Method as a finder of objects for browse window in WPMF” and “Reuse legacy browses in WPMF”.

5.2.13.2.3.1 Creation of Browse Configuration

Browse Configuration can be declared in two ways – declarative and programmatic.

5.2.13.2.3.1.1 Declarative Browse Configuration

The declarative approach is recommended when there is simple correspondence between an API service method and browse use case. Good examples would be browses for configuration objects like item, operation, etc. Usually such browses do not require a lot of transformations. Declarative approach in such cases will allow easy configuration of browse service.

```
<service name="SimpleOperationBrowseService" parent="BrowseService">
  <property name="finderService" service-ref="OperationConfigurationService" />
  <property name="finderMethod" value="findOperationsByKeyFieldMask" />
  <property name="columns">
    <!-- ordered list of columns to display, these are logical names and are
    user defined except for the highlighted ones which are reserved names -->
    <entry value="PRIMARY"/> <!-- this a mandatory name -->
    <entry value="REVISION"/> <!-- this must be used for revision field if any
    <entry value="DESCRIPTION" />
    <entry value="STATUS" />
    <entry value="OPERATION_TYPE" />
    <entry value="CURRENT_REVISION" />
  </property>
  <!-- You can define value bindings to these columns using a simple property name -->
  <property name="columnPropertyBindings">
    <entry key="PRIMARY" value="operation"/>
    <entry key="REVISION" value="revision"/>
    <entry key="OPERATION_TYPE" value="operationType"/>
  </property>
  <!-- You can define a strategy service and bind it to the column for advanced value
  binding that may need processing. -->
  <property name="columnStrategyBindings">
    <entry key="DESCRIPTION" service-ref="CopyNamedFieldStrategy"/>
    <entry key="STATUS" service-ref="ResolveStatusByRefStrategy"/>
    <entry key="CURRENT_REVISION" service-ref="FormatCurrentRevisionStrategy"/>
  </property>
  <!-- The column to sort the browse result data with -->
  <property name="sortColumn" value="DESCRIPTION"/>
  <!-- This a custom bundle name to be used for localization. If not specified, the ME core bundle is used. !-->
  <property name="bundleService" service-ref="yourBundleserviceName" />
  <!-- the browse title key, if any -->
  <property name="browseTitleKey" value="I18N[operation.default.LABEL]" />
  <!-- The bundle lookup names associated with column headers. If not specified,
  Column name is used. -->
  <property name="columnHeaders">
    <entry key="PRIMARY" value="I18N[operation.default.LABEL]"/>
    <entry key="REVISION" value="I18N[operationRevision.default.LABEL]"/>
    <entry key="DESCRIPTION" value="I18N[description.default.LABEL]"/>
    <entry key="STATUS" value="I18N[status.default.LABEL]"/>
    <entry key="OPERATION_TYPE" value="I18N[type.default.LABEL]"/>
  </property>
</service>
```

```
<entry key="CURRENT_REVISION" value="I18N[currentRevision.default.LABEL]"/>
</property>
</service>
```

The declaration of Browse Configuration can include the following properties with some required and some optional.

finderService – reference to the PAPI Service. **(Required)**

finderMethod – name of the method in PAPI Service to call in order to get collection of objects. **(Required)**

browseField – name of the field that represents collection of objects (optional; required only if response is a single DTO with collection defined as attribute). **(Optional)**

columns – ordered list of columns. Needed to set the order. **(Required)**

The columns names “PRIMARY” and “REVISION” are reserved names. The “PRIMARY” column name is always required. The “REVISION” is only required if you have a revision field.

ColumnPropertyBinding - Simple column property value bindings. These binding correspond to an actual property name within the collection of objects returned by the finder method. These bindings take precedence over the ColumnStrategyBindings below. **(Required if there is not a strategy bindings defined.)**

columnStrategyBindings – Advanced columns value bindings. The definition represents a reference to a service that implements the StrategyInterface. Execute method will be called to do any user defined or advanced processing. **(Required if there is not a simple bindings defined.)**

bundleService- A valid bundle service name to be used for localizations of values. If not defined, then a service using the ME LocaleSpecificText is used. **(Optional)**

sortColumn- The column name that resulting data will be sorted by alphanumerically. **(Optional)**

browseTitleKey– The bundle key to the browse title that is inserted as part of the browse window header. **(Required)**

columnHeaders - The bundle look up names associated with column headers. If not specified, the column definition above is used. **(Optional)**

Declarative approach allows reuse of a lot of strategies to define columns. The following strategy based services have been made available as services and can be used for advanced column bindings.

```
<service name="CopyNamedFieldStrategy" interface="com.visiprise.frame.service.StrategyInterface"
class="com.sap.me.common.client.impl.CopyNamedFieldStrategy" parent="Prototype" />
```

This service copies the field value for Object in the PAPI response collection to BrowseItem. By default the “fromField” and “toField” equal the column name. For the cases when “fromField” is different from column name, the CopyNamedFieldStrategy can be overridden as this:

```
<service name="CopyOperationFieldStrategy" parent="CopyNamedFieldStrategy">
  <property name="fromField" value="operation" />
</service>

<service name="FormatNumberStrategy"
  interface="com.sap.me.common.client.BrowseValueFormattingStrategyInterface"
  class="com.sap.me.common.client.impl.FormatNumberStrategy" parent="Prototype"
/>
```

This service formats the number in accordance with GAPI rules.

```
<service name="FormatBooleanStrategy"
  interface="com.sap.me.common.client.BrowseValueFormattingStrategyInterface"
  class="com.sap.me.common.client.impl.FormatBooleanStrategy" parent="Prototype"
/>
```

This service formats the boolean in accordance with GAPI rules.

```
<service name="FormatDateTimeInterfaceStrategy"
  interface="com.sap.me.common.client.BrowseValueFormattingStrategyInterface"
  class="com.sap.me.common.client.impl.FormatDateTimeInterfaceStrategy"
  parent="Prototype"
/>
```

This service formats the DateTimeInterface in accordance with GAPI rules.

```
<service name="FormatCurrentRevisionStrategy"
  interface="com.visiprise.frame.service.StrategyInterface"
  class="com.sap.me.common.client.impl.FormatCurrentRevisionStrategy"
  parent="Prototype"
/>
```

This service formats the current revision in accordance with SAP ME standards (conversion is made from true/false to Yes/No).

Also, the following very common strategy was developed in status/client module:

```
<service name="ResolveStatusByRefStrategy" interface="com.visiprise.frame.service.StrategyInterface"
  class="com.sap.me.status.client.impl.ResolveStatusByRefStrategy" parent="Prototype" >
  <property name="statusService" service-ref="StatusService" />
</service>
```

This service resolves status description based on status ref. This may be required by most browses.

5.2.13.2.3.1.2 Programmatic Browse Configuration

If the use case of the browse requires more complex logic, for example calling several API services, grouping records, or anything else that will make declarative approach not usable, it is recommended to create new implementation of BrowseServiceInterface.

5.2.13.2.3.2 Support of Legacy Browsers

In order to reuse existing browses anywhere in application, the LegacyBrowseService.java was developed. It implements the BrowseServiceInterface and encapsulates some logic on mapping between BrowseResponse and ApplicationVO. This service relies on naming conventions based on the browse id property and instantiates the appropriate legacy browse class using the id and the package name. Use of legacy browse service is not appropriate for customer or vendor external applications because of class loading issues.

The LegacyBrowseService depends on configuration and, in particular, on three fields; packagePrefix, browseId and searchField:

```
<service name="ResourceBrowseService" parent="LegacyBrowseService">
  <property name="packagePrefix" value="com.sap.me.browse."/>
  <property name="browseId" value="Resource"/>

  <property name="searchField" value="RESRCE"/>
</service>
```

packagePrefix indicates whether the browse is standard or custom. In most cases will be either “com.sap.me.browse.” or “com.sap.me.browse.select.”.

browseId – id of the browse.

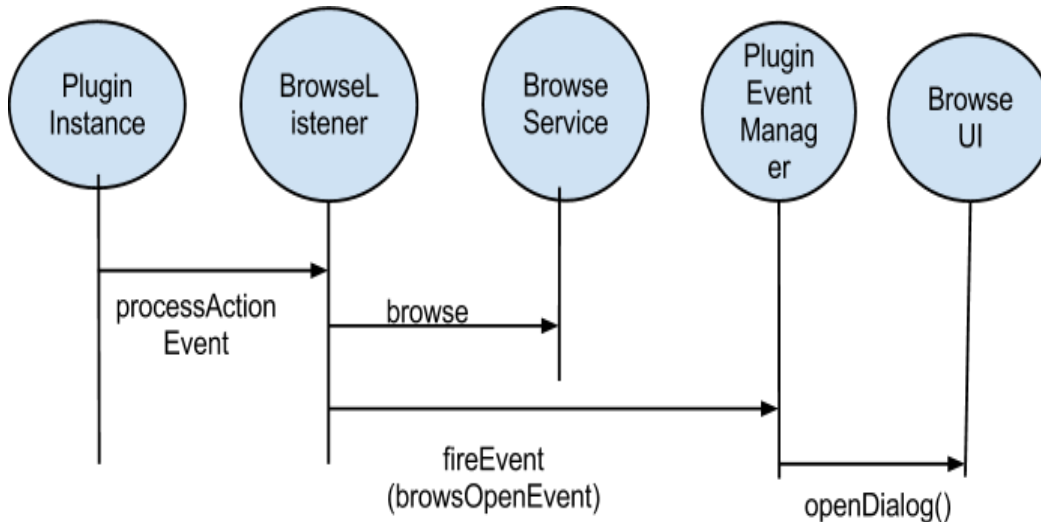
searchField – indicates which field from BrowseRequest criteria map should be used as SEARCH_VALUE.

Defining legacy browses as services will allow addressing the implementation of that service to a newer approach.

5.2.13.2.4 WMPF client & the UI

Any WMPF or POD plugin can be a client to the browse service by defining a UICommandInput field and binding it to a set of predefined attributes and an action listener. The client side does not care if a browse is legacy or not. The client side definition only points to a service and the service definition determines if we have a legacy or a new browse that uses the PAPI and the finders.

The client architecture invoking the Browse UI is simple as shown below:



The UICommandInput component on a given plugin UI is triggered by user action which sends the event to a wpmf core browse listener. The browse listener inspects the initiating component and retrieves a set of browse definitions made by the plugin author. A list of all the applicable attributes are defined below.

Attribute	Description
REQUESTING_PLUGIN_NAME	This is used to define the plugin bean name that is requesting a browse to be displayed. This optional but recommended.
SEARCH_METHOD_NAME	This is used to define a JSF method expression name that returns the browse criteria. This is required for any browse that needs to accept at least one search criteria which is most of the time, otherwise it is optional. The method signature should be: public List<AttributeValue> anyMethodName(EditableValueHolder comp)
SERVICE_MODULE	This is used to define the back end module id the browse service. This is required.
SERVICE_NAME	This is used to define the back end service name. This is required.
BROWSE_PLUGIN_NAME	This is used to define the browse plugin bean name. This is only used for custom browses. This is optional.
IS_MULTI_SELECT	This defines if the browse supports a multiple selection model. This is optional and if omitted, a single selection model is used.
BROWSE_CALLBACK	A user defined call back method name. This is optional.
BROWSE_DIALOG_WIDTH	The browse window width in px or em. This optional, and If omitted, a default value is used.
BROWSE_DIALOG_HEIGHT	The browse window height in px or em. This is optional and If omitted, a default value is used.

The UI component also needs to have an ActionListener defined. WPMF provides a core action listener comp.sap.me.wpmf.browse.BrowseListener that needs to be attached to the component. Below is an example of such a component as defined within a plugin UI (jsp page).

```
<sap:panelGroup valign="middle" height="22">
  <sap:commandInputText id="exampleInputField" width="16em"
    submitOnChange="false" submitOnFieldHelp="true"
    value="{exampleBean.browseValue}"
    title="{gapiI18nTransformer[some.tooltip.TITLE]}">
    <f:attribute name="upperCase" value="true" />
    <f:actionListener type="com.sap.me.wpmf.browse.BrowseListener"/>
    <f:attribute name="REQUESTING_PLUGIN_NAME" value="examplePlugin" />
    <f:attribute name="SEARCH_METHOD_NAME" value="examplePlugin.getCriteria" />
    <f:attribute name="SERVICE_MODULE" value="com.sap.me.productdefinition.client" />
    <f:attribute name="SERVICE_NAME" value="SimpleOperationBrowseService" />
    <f:attribute name="BROWSE_PLUGIN_NAME" value="defaultCoreBrowsePlugin" />
    <f:attribute name="IS_MULTI_SELECT" value="false" />
    <f:attribute name="BROWSE_CALLBACK" value="examplePlugin.processBrowseClosed" />
    <f:attribute name="BROWSE_DIALOG_WIDTH" value="20em" />
    <f:attribute name="BROWSE_DIALOG_HEIGHT" value="25em" />
  </sap:commandInputText>
</sap:panelGroup>
```

5.2.14 Unsaved Data Behavior

It is possible to perform an activity such as Data Collection within a dynamic or popover panel that is not modal in nature. This poses a problem if the user who is collecting data purposely or accidentally changes the selection model (e.g. selects a different SFC or Operation from the Work and/or Operation List). The user will lose data that has not been saved. WPMF and the POD should therefore provide a mechanism to prevent such losses.

The current design deals with this requirement by intercepting the relevant events during their corresponding event handling based on the existence of a lock in the system.

Requesting and the creation of the Lock object are performed at the WPMF level, but WPMF does not make any assumptions about the use of this object. It is the responsibility of the POD to make use of this lock object to intercept the events at the proper places. A lock can be requested or removed by any plugin in the system by calling the following Framework Manager methods:

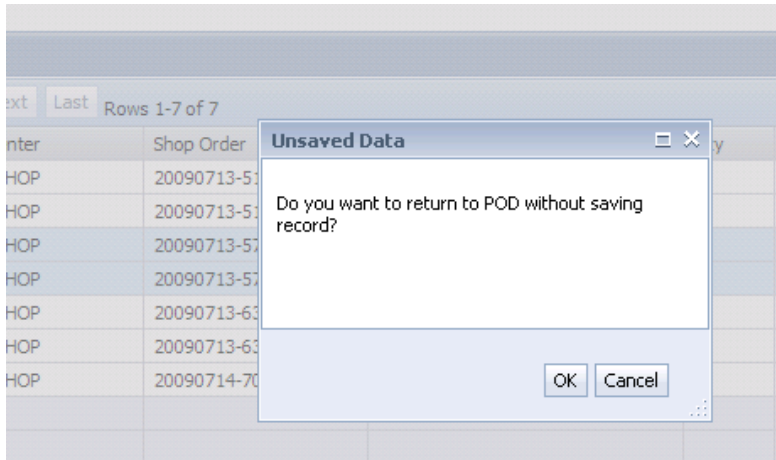
```
/**
 * Request creation of an edit lock object. Once an edit lock is
 * created, a further call to this method does not create new ones.
 *
 * @param plugin The plugin requesting the lock.
 * @return The EditLock object
 */
public EditLockInterface requestEditLock(PluginInterface plugin);

/**
 * Remove the lock object from the session.
 */
public void removeEditLock();
```

Once a lock is created it stays with the session until it is explicitly removed either by the originating plugin or any other entity at the proper time.

5.2.14.1 Unsaved Data Example

In the Production POD implementation, the Pod Selection and Worklist plugins for both the Operation and Work Center Pods intercept the appropriate events based on detecting the lock object. The intercepted event posts a dialog asking the user if they like to continue (ok) with event processing or cancel. Selecting **OK removes the lock** and continues with the normal event processing.



This behaviour can be replicated by any other plugin in the system if required using the method calls provided in the “**com.sap.me.production.podclient.BasePodPlugin**” class as shown below:

```
/**
 * This method checks for the existence of a lock object and posts a standard dialog.
 *
 * @param methodExper The fully qualified callback method name of the event handler.
 * @param event The event which can be null
 * @param returnType The return type of the event handler method which can be null;
 */
protected boolean checkForUnsavedData(String methodExper, Object event, Class<?> returnType);

/**
 * Determine if a lock object exists.
 *
 * @return true or false.
 */
protected boolean isEditLock();
```

In the following example the Work List in the Work Center POD intercepts its TableSelectionEvent causing table selections to be intercepted.

```
public void processTableSelectionEvent(TableSelectionEvent event) {
    if (checkForUnsavedData(
        "#{workcenterWorklistConfigurator.fireTableSelectionEvent}",
        event, null)) {
        return;
    }
}
```

This shows how the event is intercepted when the Clear button is selected on the Work Center Pod Selection Plugin.

```
/**
 * Clear the worklist and the selection criterion in response to the
 * user action.
 */
public void clear() {
    if (checkForUnsavedData(
        "#{workCenterPodSelectionPlugin.clear}", null, null) ) {
        return;
    }
}
```

5.3 Adding Plugins to ME

Once you have developed your plugins and built and deployed the SAP ME Application you must define the plugins as activities in Activity Maintenance and assign your activities to a POD in POD Maintenance. This section describes the steps you must perform to run your plugins in a ME POD.

5.3.1 Create the Plugin Activity

Activity Maintenance is used to define your plugin to the ME application. The following steps describe how to create the activity for the plugin.

1. Enter in a unique Activity name and Description.
2. Check the Enabled property and leave the “Visible in Activity Manager un-checked.
3. Enter the managed bean name in the Class/Program field. In 6.0.x releases you were required to enter the fully qualified plugin class in the Class / Program field. This is still supported but in order to use multiple views you must use the managed bean name as defined in the faces-config.xml file.
4. Select “WPMF Plugin” as the activity Type.
5. If the plugin is defined using the full class name or it is desirable to force the use of only one view (JSP), select the Rules tab and insert a new rule named “PLUGIN_URL”. In the settings field enter the fully qualified path of the plugins JSP starting with a “/” (i.e.; /COM/<VENDORID>/.../EXAMPLEPLUGIN.JSP). The Setting field only allows upper case characters to be entered. These will be converted to all lower case when rendering the actual display. If multiple views are required, do not use the PLUGIN_URL rule.
6. Select the Activity Group tab and assign the activity to an activity group.

5.3.2 Assign permissions for Activity

In order for the activity to be displayed or executed in a POD; you must assign the user group permission to use the activity. The following steps describe how to assign permission for the activity in User Group Maintenance.

1. Select and retrieve a user group to assign permissions to the plugin activity for.

2. Select the Permissions tab and find the plugin activity under the activity group you had assigned the plugin to,
3. Check the Execute check-box and select Save.

5.3.3 Add Plugin Activity to a POD

POD Maintenance is used to define a Production Order Dashboard to ME. You assign your plugin activity to a POD either to a layout area for plugins with a view or to a button if your plugin is an execution-only plugin. If your view plugin is assigned as an alternate plugin to a fixed layout area or to a Popover or Popup layout area, you will need to assign it to a button in order to trigger the plugin to be displayed in the layout area or popup window. See the [POD Maintenance Documentation](#) for more details on creating or modifying existing PODs.

5.3.4 Button Activity Processing

Multiple activities that are assigned to a button will be processed in sequential order as defined for the button. The activities can be a mixture of view and executable type plugins. The following screen shows the button details for a button assigned a mixture of plugin types. Any view type plugins listed must also be assigned to a Layout area. In the example below, the first and last activities listed are assigned to fixed layout areas (i.e.; Panel C & B) as alternate plugins. These are referred to as Popin plugins. The second view plugin is assigned as a Popover and the fourth is assigned as a Popup. All the rest of the plugins are execution-only plugins (i.e.; no view).

* Activity Sequence	* Activity	
10	SAPDEV_VIEW1	Layout Area 'C' (Popin)
20	SAPDEV_VIEW2	Popover
30	SAPDEV_EXE1	
40	SAPDEV_POPUP	Popup
50	SAPDEV_EXE3	
60	SAPDEV_POPIN	Layout Area 'B' (Popin)

The following steps describe what will occur when the button is pressed to run the activities for this example.

1. The first two view plugins will be displayed in the POD where SAPDEV_VIEW1 will display in layout area “C” and SAPDEV_VIEW2 will display in the Popover layout area. Further execution of activities is stopped until the user closes the plugin for SAPDEV_VIEW2.
2. After SAPDEV_VIEW2 plugin is closed the next execution plugin (SAPDEV_EXE1) will run. This plugin will complete when the processing in the execute() method is finished

unless the plugin has implemented controlled execution, in which case it will finish when the complete() method is called (See [Programmer Controlled Execution](#)).

3. After SAPDEV_EXE1 is finished, the next plugin will be displayed in a Popup window and further execution is stopped until the Popup closes.
4. After the plugin in the Popup is finished the next execution plugin will be executed in the same way as the previous one (SAPDEV_EXE1) was.
5. When the execution plugin is finished, the final plugin (SAPDEV_POPIN) will be displayed in layout area B.

In this example the first two view plugins were assigned to different layout areas; thus were displayed simultaneously. If they had been defined to the same layout area, they would have been displayed sequentially with the 2nd one not displaying until the first plugin was closed.

5.4 Current Limitations

Current there are some limitations as to what you can do from a plugin as described below.

1. Cannot modify current core SAP ME Plugin classes; however you can copy and modify the view (JSP) used by a core plugin (as defined by the plugins activity) but you should not alter the class.
2. Plugins can only be triggered by WPMF defined events, button clicks or during POD loading; not through Hook extensions, ejb's, etc.
3. Plugins must run within a defined POD. They cannot run standalone outside of the POD framework.

5.5 Best Practices

The following describe best practices to use when developing plugins.

1. Use only the extended <sap: panelGrid> layout control for laying out your UI. This works in conjunction with the <sap:panelRow> and <sap:panelGroup> controls to create the desired grid layout.
2. Use LSF external or JSF standard controls when possible.
3. Do not hard code the size of components. Let the layout manager place and manage them. If you must hard code values; use “em” units where possible and supported.
4. Limit the amount of nested LSF containers unless the design mandates it.
5. Use the <sap:panel> as the main container along with <sap: panelGrid> to manage the design layout. The panel control can also be used as a scroll container elsewhere in your layout if needed (I.e.; as a parent of the dataTable).
6. Use the provided standard Table Configurators. Do not attempt to extend these unless absolutely required.
7. Use EL expressions rather than hard coded values where possible.
8. Try to keep a one-to-one relationship between the view (JSP) and model (plugin class). Do not overcomplicate this by creating unnecessary classes/objects if not required. Keep it simple.
9. Whenever you use a “binding” to a JSF/LSF control create a “backing bean” defined to the request scope to hold the reference as described in [Developing WPMF Plugins](#).

6 LSF Extended Controls

With prior ME releases (i.e.; ME 6.0.x) it was necessary to use LSF “internal” controls to create your views. These controls are deprecated and being replaced by “external” controls that more closely follow standard JSF control structure.

6.1 Page

The following controls are used when defining standalone LSF pages.

html	The html component acts as the root element of the page, if consists of head and body components and is the root element of the control tree.
head	The head component comprises as a container for html head elements (scripts, style sheets, etc.)
body	The body component contains the html content (text, hyperlinks, images etc.). It is possible to configure the page in case of design, domain relaxing, focus handling, session timeout.
hotkey	The hotkey component allows to register hot keys for the body component defining additionally processing methods for the corresponding hot keys.

6.2 Containers

The following container controls are supported.

panel	This component renders a panel which is collapsible and can have different designs for header, border and con-tent area.
panelTabbed	This component can nest multiple panel components which are displayed in different tabs.
panelSplitter	This component provides two layout areas separated by a horizontal or vertical splitter bar that can be configured to be fixed or moveable.

6.3 Layouts

The following controls should be used for all layouts.

panelGrid	A component to define a rectangular column/row based grid layout. Row break can be achieved either with the column property or by nesting components within a panelRow component (it's not supported to define columns and nest panelRow).
panelRow	This component can be used either within an extended panelGrid component to break the components into a new row or standalone to implement a flow layout.
panelGroup	This component can be used either within an extended panelGrid or panelRow component to group multiple components in one cell or standalone to set layout properties for a single area.

6.4 Command

The following controls are provided to perform submits to the server.

commandBooleanCheckbox	Renders an HTML input element of type “checkbox”.
commandManyCheckbox	Renders an HTML input element of type “checkbox”.
commandManyListbox	Renders an HTML select element.
commandManyMenu	Renders an HTML select element.
commandManyShuttle	Renders a shuttle control for moving list items between two lists.
commandOneListbox	Renders an HTML select element.
commandOneMenu	Renders an HTML select element.
commandOneRadio	Renders an HTML input element of type “radio”.
commandInputText	Renders an HTML input element of type "text".
commandButton	Extends the standard commandButton in areas like setting access keys, title designation, width etc.

commandButtonLarge	Establishes with setting width and the height creation of large command buttons.
--------------------	--

6.5 Select

The following controls are provided to allow selection without submits to the server.

selectManyListbox	Renders an HTML select element.
selectManyMenu	Renders an HTML select element.
selectOneListbox	Renders an HTML select element.
selectOneMenu	Renders an HTML select element.

6.6 Input

The following controls will handle inputting data.

inputDate	Renders an input field with a date picker.
inputSuggest	Renders an input element of type "text".
inputText	Renders an input element of type "text".
inputTextFormatted	This component renders an input component with a simple WYSIWYG editor for formatting the text (e.g. bold or italic font).
inputTextArea	Renders an HTML "textarea" element.

6.7 Output

The following controls will handle outputting data.

outputTextFormatted	This component renders an output component for displaying formatted the text (e.g. bold or italic font).
outputLabel	Renders an HTML "label" element.

6.8 Data

The following controls are provided to handle table and tree structures.

dataTable	Data Table with extended features like selection, sorting etc.
tree	The tree component is designed for hierarchical data presentation based on tree nodes.
dataGrid	Like the data table but row data can be rendered like a grid layout (i.e.; also known as Row Repeater or Smart Table)

6.9 Miscellaneous

The following controls handle other helpful tasks.

ajaxUpdate	The ajaxUpdate component enables delta rendering updates on the specified UI components.
poll	This component sends periodical poll requests to the server. It is used for repeatedly updating a page at defined time intervals.
separator	A horizontal line to use as a separator respectively divider in a layout.
toolbar	The Toolbar component is a grouping component for action items and other content, like input fields

6.10 Internal Control Migration

The following table describes what control should be used to replace “internal” LSF controls. In the following table the following tag libraries are referenced:

```
<%@ taglib prefix="h" uri="http://java.sap.com/jsf/html"%>
<%@ taglib prefix="s" uri="http://java.sap.com/jsf/html/extended"%>
<%@ taglib prefix="ls" uri="http://java.sap.com/jsf/html/internal"%>
```

Internal Component	Replacement Component	Comments
sap-delta-id	<s:ajaxUpdate>	
<ls:button>	<s:commandButton>	
<ls:buttonRow>	<h:panelGrid> <h:panelGroup> <s:panelGrid> <s:panelGroup> <s:toolBar>	
<ls:checkboxGroup>	<h:selectManyCheckbox>	
<ls:container>	<h:panelGrid> <h:panelGroup> <s:panelGrid> <s:panelGroup>	
<ls:contentArea>	<s:panel>	Can be replaced with <s:panel> without any header facet and with property isCollapsible="false"
<ls:flowLayout>	<h:panelGroup>	
<ls:gridLayout>	<h:panelGrid> <s:panelGrid>	
<ls:gridLayoutCell>	<s:panelGroup> inside <s:panelGrid>	<s:panelGroup> is not needed if only one component should be rendered within the cell, this is done automatically. <s:panelGroup> is only needed if you want to group more than one component within one cell.
<ls:gridLayoutRow>	<s:panelRow>	Nest <s:panelRow>s inside <s:panelGrid>. <s:panelRow> is not needed if the "columns" attribute of <s:panelGrid> is used.
<ls:group>	<s:panel>	Collapse can be disabled to get same result as group.
<ls:horizontalDivider>	<s:separator>	
<ls:HTMLTextView>	<h:outputText>	The control was developed to integrate HTML code inside the view description of frameworks which do not support this directly. This component is not needed for JSF where it is easily possible within

Internal Component	Replacement Component	Comments
		the JSP to define any arbitrary HTML.
<ls:hotkey> and <ls:hotkeys>	<s:hotkey>	
<ls:iframe>	HTML <frame> (use no component)	Can be replaced with plain html <iFrame>. If EL expressions are needed you must use an immediate expression with \$ notation.
<ls:imageMap>	HTML <map> and <area> (use no component)	Can be replaced with plain html <map>. If EL expressions are needed you must use an immediate expression with \$ notation.
<ls:inputField>	<h:inputText> <s:commandInputText>	<s:commandInputText> supports "fieldHelpButton" and a "submit on change". For uppercase characters add <f:attribute name="upperCase" value="true" /> to the control as a child.
<ls:label>	<h:outputLabel>	
<ls:largeButton>	<s:commandButtonLarge>	
<ls:link>	<h:commandLink> <h:outputLink>	
<ls:matrixLayout>	<h:panelGrid> <s:panelGrid>	See <ls:gridLayout> for further details.
<ls:matrixLayoutCell>	<s:panelGroup>	See <ls:gridLayoutCell> for further details
<ls:matrixLayoutRow>	<s:panelRow>	See <ls:gridLayoutRow> for further details
<ls:messageBar>	<h:messages> <h:message>	
<ls:page>	<s:html> <s:head> <s:body>	
<ls:pageHeader>	<s:pageHeader>	
<ls:panel>	<s:panel>	
<ls:script>	HTML <script> tag inside of <h:outputText> or <s:head>	
<ls:scrollContainer>	<s:panel>	Can be replaced with <s:panel> without a header facet and with property isCollapsible="false".
<ls:singleColumnLayout>	<s:panelGrid> with columns="1"	

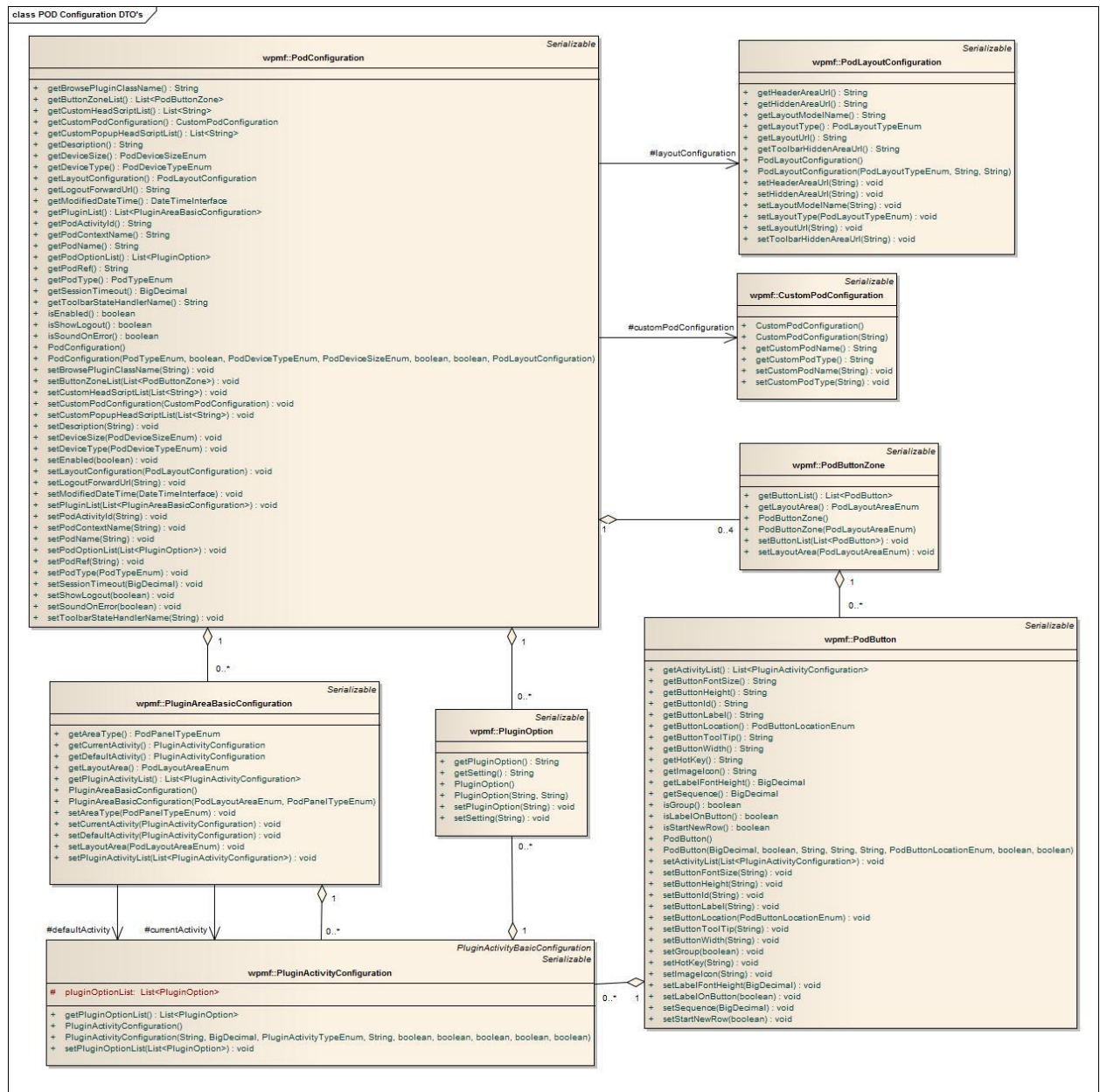
Internal Component	Replacement Component	Comments
<ls:singleColumnLayoutCell>	<s:panelGroup> inside <s:panelGrid>	<s:panelGroup> is not needed if only one component should be rendered within the cell, this is done automatically. <s:panelGroup> is only needed if you want to group more than one component within one cell.
<ls:splitterCell> <ls:splitterCol> <ls:splitterRow>	<s:panelSplitter>	
<ls:textView>	<h:outputText> or HTML content	
<ls:toolbar>	<s:toolBar>	
<ls:viewContainer>	<h:panelGrid> <h:panelGroup> <s:panelGrid> <s:panelGroup>	

7 Customization with WPMF

It is possible to make modifications to the POD Configuration information prior to initialization and rendering of the POD. In addition the ability to modify the state (enable/disable) of the buttons on the toolbar during initialization of the POD and during plugin event processing was added. This section covers the ways the POD can be customized by 3rd party implementations.

7.1 Configuration Filter

The configuration for a POD is made in POD Maintenance. When the POD is to be rendered the configuration information is read and loaded into a set of WPMF classes as shown below:



It is now possible to intercept and modify the contents of this information prior to rendering of the POD using the following steps:

1. Create a new “filter” class which implements “com.sap.me.wpmf.application.ConfiguratorFilterInterface”.
2. Define the filter class as a managed bean to a faces-config.xml using the bean name of “podConfiguratorFilterBean”.

The ConfiguratorFilterInterface defines the following method which is invoked after the configuration is loaded from the database and before it is passed to the WPMF.

```
/**
 * Method called to allow modification of PodConfiguration object
 * prior to initialization of POD.
 *
 * @param podConf PodConfiguration object
 */
public void modifyPodConfiguration(PodConfiguration podConf);
```

Care must be taken when modifying the configuration information so as not to impact rendering of the POD.

For example you could define a [Toolbar State Handler](#) and assign it to a POD using the modifyPodConfiguration() method as follows:

```
public void modifyPodConfiguration(PodConfiguration podConf) {
    String podName = podConf.getPodName();
    if (!Utils.isBlank(podName) && podName.equalsIgnoreCase("examplePod")) {
        podConf.setToolbarStateHandlerName("toolbarStateHandler");
    }
}
```

7.2 Toolbar State Handler

The ability to manage the state of buttons on a toolbar or in the popup displayed for Activity Group buttons has been added. The button state can be set during initial loading of the POD and after plugin events have been processed as described below. The following diagram describes the “com.sap.me.wpmf.ToolBarStateHandlerInterface” and “com.sap.me.wpmf.AbstractToolBarStateHandler” class that has been developed to support this. The abstract class should be extended to implement the state handler. The implementing class must be defined as a managed bean in the faces-config.xml file as follows:

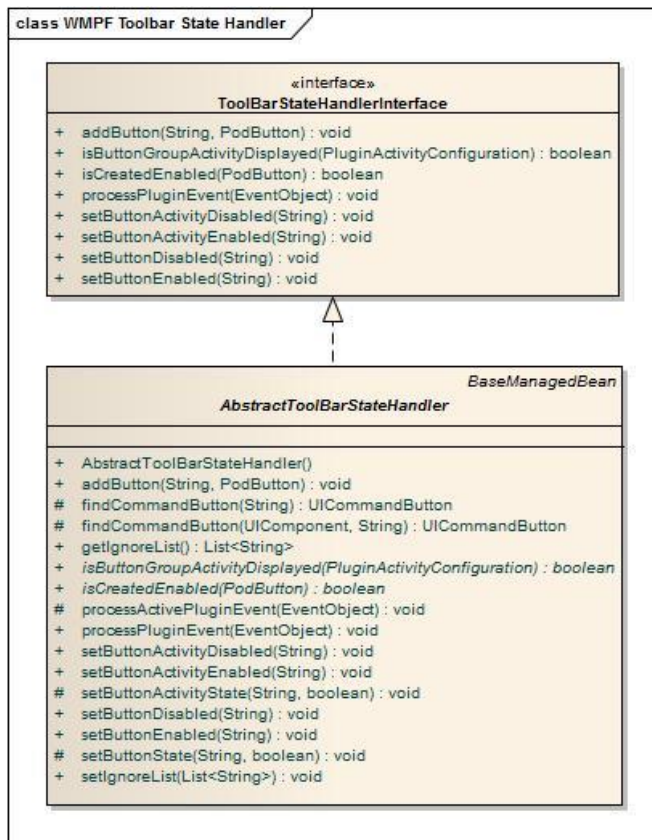
```
<managed-bean>
  <managed-bean-name>toolbarStateHandler</managed-bean-name>
  <managed-bean-class>com.sap.me.example.TestToolBarStateHandler</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>ignoreList</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>com.sap.me.wpmf.MessageEvent</value>
      <value>com.sap.me.wpmf.impl.ActivitySelectDialogOpenEvent</value>
    
```

```

        <value>com.sap.me.wpmf.ToolBarActionEvent</value>
        <value>com.sap.me.wpmf.GlobalDialogCloseEvent</value>
        <value>com.sap.me.wpmf.LayoutRefreshEvent</value>
    </list-entries>
</managed-property>
</managed-bean>

```

The “ignoreList” property allows you define a list of plugin event classes to ignore during event processing. The following class diagram describes the interface and abstract class.



7.2.1 Initial POD Loading

When the POD is loading the isCreatedEnabled(PodButton) method is called to allow setting the initial state (enabled/disabled) for the input button.

```

/**
 * Returns whether the button should be initially created enabled or disabled
 *
 * @param button PodButton button definition
 * @return true to create enabled, false to create disabled.
 */
public boolean isCreatedEnabled(PodButton button);

```

7.2.2 Plugin Event Processing

When events are fired, the processPluginEvent(EventObject) method is called after all plugins listening to the event have been notified. The abstract class implements this method and filters out

events that have been defined in the “ignoreList” managed property. It then invokes the following protected method to allow processing the event.

```
/**
 * Called to process an event that is not on the "ignore" list.
 *
 * @param event EventObject not on ignore list
 */
protected void processActivePluginEvent(EventObject event);
```

The implementing class can then interrogate the EventObject and enable/disable toolbar buttons as required based on the button ID or the Activity ID assigned to the button. When using an activity ID, the ID only needs to match one of the activities in the button's activity list. This does not apply to activity group buttons. A number of setter methods are provided by the abstract class to support enabling and/or disabling buttons on the toolbar.

7.2.3 Group Buttons

The isButtonGroupActivityDisplayed(PluginActivityConfiguration) method is used to allow filtering out activities from the Button Group activity list based on the button activity information in the PluginActivityConfiguration object.

```
/**
 * Returns whether the button group activity should be displayed
 * in the activity popup given for the input PluginActivityConfiguration.
 *
 * @param pac PluginActivityConfiguration describing activity
 * @return true to display in activity list, else false to not display
 */
public boolean isButtonGroupActivityDisplayed(PluginActivityConfiguration pac);
```

8 Standalone Application Development

While WMPF is used to render a POD as defined in POD maintenance, it is also possible to use LSF and/or the WMPF framework to develop standalone applications (activities) that do not run in a POD.

8.1 Applications using JSF / LSF

It is possible to develop UI's using JSF technology utilizing the LSF component set. The following example shows how a standalone JSF page utilizing LSF looks like.

```
<%@page pageEncoding="UTF-8"%>
<%@ page language="java"%>
<%@ taglib prefix="h" uri="http://java.sap.com/jsf/html"%>
<%@ taglib prefix="f" uri="http://java.sap.com/jsf/core"%>
<%@ taglib prefix="sap" uri="http://java.sap.com/jsf/html/extended"%>

<f:view>
  <sap:html>
    <sap:head>
      <script src="{exampleBean.jsFile}" type="text/javascript"></script>
      <script>
        UCF_DomUtil.attachEvent(window, 'beforeunload', windowClosing);

        function loadLightSpeed(oLS) {
          return false;
        };
        function unloadLightSpeed() {
          return false;
        };
        function windowClosing() {
          .....
        };
      </script>
    </sap:head>

    <sap:body id="examplePage" title="Example Page" focusId="exampleForm" browserHistory="disabled"
      onload="return loadLightSpeed(oLS);" onunload="return unloadLightSpeed();">

      <f:attribute name="sap-ui-ls" value="true" />

      <sap:hotKey id="hotkey" key="CTRL_0,CTRL_1,CTRL_2,.....,ESCAPE,ENTER"
        actionListener="#{exampleBean.processHotkeyAction}" />

      <h:form id="exampleForm">
        <f:attribute name="height" value="100%" />

        <sap:ajaxUpdate render="exampleForm"/>

        <sap:panelGrid width="100%" height="100%" cellHalign="start" cellValign="top">
          Controls go here...
        </sap:panelGrid>
      </h:form>
    </sap:body>
  </sap:html>
</f:view>
```

The example above shows how to implement JavaScript as well have access to the LSF JavaScript library. When including JavaScript files using the <script> tag, you must use an immediate expression using the “\$” notation. It does not work if using a hard coded URL to the “src” attribute.

The JSP file is saved using the “.jsp” extension. The URL referencing the JSP page should use the JSF extension “.jsf” so that the JSF Servlet will process the file before it is processed as a JSP page.

8.2 Applications using WPMF

In many cases developing applications using JSF/LSF technology is sufficient; however for more complex applications that use popup windows, browses and advanced layouts, it is possible to leverage the WPMF framework. The following steps describe how to create a WPMF dashboard (POD).

1. Create UI and Non-UI plugins by extending the ApplicationPlugin class.
2. Create views for the UI Plugins.
3. Create optional “Reserved Area” plugin that will display in the Reserved Area of the [POD Layout](#).
4. Create optional “header” JSP page that will display in the Header Area of the [POD Layout](#).
5. Create optional JSP pages that contain “hidden” controls that can be accessed via JavaScript. This is useful for including information that will be sent to server during submits. One JSP is reserved to be included in the main layout page and the second will be included in all toolbars. These JSP’s must NOT define components that will be displayed.
6. Define UI and Non-UI plugins in the faces-config.xml.
7. Define optional buttons for display on toolbars. Buttons are defined in the faces-config.xml file using the ButtonConfigurationBean as the <managed-bean-class>. The button definition includes the list of plugins that should be run when the button is pressed.
8. Define panels that will be assigned to the layout areas. The layout areas defined is dependent on the actual layout used (i.e.; one panel horizontal or vertical, three panel horizontal or vertical, etc). Panels are defined in the faces-config.xml using the PanelConfigurationBean as the <managed-bean-class>. The panel definition includes the name of the default plugin and a list of optional plugin names that are assigned to the panel.
9. Define the POD which describes the layout to be used, the panels to be included and the buttons to be assigned to the different toolbars. The POD is defined in the faces-config.xml using the PodConfigurationBean as the <managed-bean-class>.

The WPMF POD can then be rendered using the following URL:

http://host:port/manufacturing/com/sap/me/wpmf/client/template.jsf?POD_ID=examplePod

Where POD_ID is assigned the name given to your POD plugin managed bean in step 9.

The following class diagram shows the classes that are used to define a WPMF dashboard (POD).



8.2.1 Example faces-config.xml

The following faces-config.xml file defines a POD using the one panel vertical layout with a left hand toolbar assigned including four buttons. One button contains a Non-UI plugin, the second defines a UI plugin to be displayed in the popover area, the third is a UI-Plugin to display in as a popup and the last is a “Group” type button that includes all three plugins.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
```

<!--POD Definition -->

```
<managed-bean>
  <managed-bean-name>examplePOD</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.PodConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>description</property-name>
    <value>Example POD</value>
  </managed-property>
  <managed-property>
    <property-name>layoutUrl</property-name>
    <value>/com/sap/me/production/podclient/OnePanelVerticalLayout.jsp</value>
  </managed-property>
  <managed-property>
    <property-name>layoutType</property-name>
    <value>1_PANEL_VERTICAL</value>
  </managed-property>
  <managed-property>
    <property-name>layoutModelName</property-name>
    <value>podLayoutModel</value>
  </managed-property>
  <managed-property>
    <property-name>sessionTimeout</property-name>
    <property-class>java.lang.Long</property-class>
    <value>30</value>
  </managed-property>
  <managed-property>
    <property-name>panels</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>panelA</value>
      <value>panelB</value>
      <value>panelZ</value>
    </list-entries>
  </managed-property>
  <managed-property>
    <property-name>leftButtons</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>leftButton1</value>
      <value>leftButton2</value>
      <value>leftButton3</value>
      <value>leftButton4</value>
    </list-entries>
  </managed-property>
</managed-bean>
```

<!--Panel Definitions -->

```
<managed-bean>
  <managed-bean-name>panelA</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.PanelConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>area</property-name>
    <value>A</value>
  </managed-property>
  <managed-property>
    <property-name>panelType</property-name>
    <value>FIXED</value>
  </managed-property>
  <managed-property>
    <property-name>defaultActivity</property-name>
    <value>exampleDefaultPlugin</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>panelB</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.PanelConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>area</property-name>
    <value>B</value>
  </managed-property>
  <managed-property>
    <property-name>panelType</property-name>
    <value>FIXED</value>
  </managed-property>
  <managed-property>
    <property-name>activities</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>examplePopoverPlugin</value>
    </list-entries>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>panelZ</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.PanelConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>area</property-name>
    <value>Z</value>
  </managed-property>
  <managed-property>
    <property-name>panelType</property-name>
    <value>POPUP</value>
  </managed-property>
  <managed-property>
    <property-name>activities</property-name>
    <list-entries>
```

```
        <value-class>java.lang.String</value-class>
        <value>examplePopupPlugin</value>
    </list-entries>
</managed-property>
</managed-bean>
```

<!--Button Definitions -->

```
<managed-bean>
    <managed-bean-name>leftButton1</managed-bean-name>
    <managed-bean-class>com.sap.me.wpmf.application.ButtonConfigurationBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>buttonId</property-name>
        <value>POPOVER</value>
    </managed-property>
    <managed-property>
        <property-name>buttonType</property-name>
        <value>SINGLE</value>
    </managed-property>
    <managed-property>
        <property-name>label</property-name>
        <value>Example Popover</value>
    </managed-property>
    <managed-property>
        <property-name>tooltip</property-name>
        <value>Example for showing plugin in popover area</value>
    </managed-property>
    <managed-property>
        <property-name>activities</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>examplePopoverPlugin</value>
        </list-entries>
    </managed-property>
</managed-bean>

<managed-bean>
    <managed-bean-name>leftButton2</managed-bean-name>
    <managed-bean-class>com.sap.me.wpmf.application.ButtonConfigurationBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>buttonId</property-name>
        <value>POPUP</value>
    </managed-property>
    <managed-property>
        <property-name>buttonType</property-name>
        <value>SINGLE</value>
    </managed-property>
    <managed-property>
        <property-name>label</property-name>
        <value>Example Popup</value>
    </managed-property>
    <managed-property>
        <property-name>tooltip</property-name>
```

```
<value>Example for showing plugin in popup area</value>
</managed-property>
<managed-property>
  <property-name>size</property-name>
  <value>100</value>
</managed-property>
<managed-property>
  <property-name>iconPath</property-name>
  <value>/themes/sap_me/icons/document_list.gif</value>
</managed-property>
<managed-property>
  <property-name>activities</property-name>
  <list-entries>
    <value-class>java.lang.String</value-class>
    <value>examplePopupPlugin</value>
  </list-entries>
</managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>leftButton3</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.ButtonConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>buttonId</property-name>
    <value>EXECUTE</value>
  </managed-property>
  <managed-property>
    <property-name>buttonType</property-name>
    <value>SINGLE</value>
  </managed-property>
  <managed-property>
    <property-name>label</property-name>
    <value>Example Execute</value>
  </managed-property>
  <managed-property>
    <property-name>tooltip</property-name>
    <value>Example for running an execution plugin</value>
  </managed-property>
  <managed-property>
    <property-name>activities</property-name>
    <list-entries>
      <value-class>java.lang.String</value-class>
      <value>exampleExecutionPlugin</value>
    </list-entries>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>leftButton4</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.application.ButtonConfigurationBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>buttonId</property-name>
```

```
        <value>GROUPBTN</value>
    </managed-property>
    <managed-property>
        <property-name>buttonType</property-name>
        <value>GROUP</value>
    </managed-property>
    <managed-property>
        <property-name>label</property-name>
        <value>Activities 1...</value>
    </managed-property>
    <managed-property>
        <property-name>hotkey</property-name>
        <value>CTRL_A</value>
    </managed-property>
    <managed-property>
        <property-name>tooltip</property-name>
        <value>Example button group functionality</value>
    </managed-property>
    <managed-property>
        <property-name>activities</property-name>
        <list-entries>
            <value-class>java.lang.String</value-class>
            <value>examplePopoverPlugin</value>
            <value>examplePopupPlugin</value>
            <value>exampleExecutionPlugin</value>
        </list-entries>
    </managed-property>
</managed-bean>
```

<!--Plugin Definitions -->

```
<managed-bean>
    <managed-bean-name>exampleDefaultPlugin</managed-bean-name>
    <managed-bean-class>com.sap.me.example.client.impl.ExampleDefaultPlugin</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>beanName</property-name>
        <value>exampleDefaultPlugin</value>
    </managed-property>
    <managed-property>
        <property-name>backingBeanName</property-name>
        <value>exampleDefaultBackingBean</value>
    </managed-property>
    <managed-property>
        <property-name>pluginId</property-name>
        <value>EXAMPLE_DEFAULT</value>
    </managed-property>
    <managed-property>
        <property-name>description</property-name>
        <value>Default Plugin</value>
    </managed-property>
    <managed-property>
        <property-name>url</property-name>
        <value>/com/sap/me/example/client/exampleDefaultPlugin.jsp</value>
    </managed-property>
</managed-bean>
```

```
</managed-bean>
<managed-bean>
  <managed-bean-name>exampleDefaultBackingBean</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.PluginBackingBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>exampleDefaultBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>assignedBeanName</property-name>
    <value>exampleDefaultPlugin</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>examplePopoverPlugin</managed-bean-name>
  <managed-bean-class>com.sap.me.example.client.impl.ExamplePopoverPlugin</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>examplePopoverPlugin</value>
  </managed-property>
  <managed-property>
    <property-name>backingBeanName</property-name>
    <value>examplePopoverBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>pluginId</property-name>
    <value>EXAMPLE_POPOVER</value>
  </managed-property>
  <managed-property>
    <property-name>description</property-name>
    <value>Example Popover</value>
  </managed-property>
  <managed-property>
    <property-name>url</property-name>
    <value>/com/sap/me/example/client/examplePopoverPlugin.jsp</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>examplePopoverBackingBean</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.PluginBackingBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>examplePopoverBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>assignedBeanName</property-name>
    <value>examplePopoverPlugin</value>
  </managed-property>
</managed-bean>
```

```
<managed-bean>
  <managed-bean-name>examplePopupPlugin</managed-bean-name>
  <managed-bean-class>com.sap.me.example.client.impl.ExamplePopupPlugin</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>examplePopupPlugin</value>
  </managed-property>
  <managed-property>
    <property-name>backingBeanName</property-name>
    <value>examplePopupBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>pluginId</property-name>
    <value>EXAMPLE_POPUP</value>
  </managed-property>
  <managed-property>
    <property-name>description</property-name>
    <value>Example Popup</value>
  </managed-property>
  <managed-property>
    <property-name>url</property-name>
    <value>/com/sap/me/example/client/examplePopupPlugin.jsp</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>examplePopupBackingBean</managed-bean-name>
  <managed-bean-class>com.sap.me.wpmf.PluginBackingBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>examplePopupBackingBean</value>
  </managed-property>
  <managed-property>
    <property-name>assignedBeanName</property-name>
    <value>examplePopupPlugin</value>
  </managed-property>
</managed-bean>

<managed-bean>
  <managed-bean-name>exampleExecutionPlugin</managed-bean-name>
  <managed-bean-class>com.sap.me.example.client.impl.ExampleExecutionPlugin</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>pluginId</property-name>
    <value>EXAMPLE_EXECUTION</value>
  </managed-property>
  <managed-property>
    <property-name>description</property-name>
    <value>Example Execution Plugin</value>
  </managed-property>
</managed-bean>

</faces-config>
```

If the optional reserved area plugin or hidden jsp pages are to be used you can add them using the following properties:

```
<managed-property>
  <property-name>reservedPanelName</property-name>
  <value>reservedAreaPlugin</value>
</managed-property>
<managed-property>
  <property-name>headerAreaUrl</property-name>
  <value>/com/sap/me/example/client/layout/templateHeader.jsp</value>
</managed-property>
<managed-property>
  <property-name>hiddenAreaUrl</property-name>
  <value>/com/sap/me/example/client/layout/templateHiddenArea.jsp</value>
</managed-property>
<managed-property>
  <property-name>toolbarHiddenAreaUrl</property-name>
  <value>/com/sap/me/example/client/layout/toolbarHiddenArea.jsp</value>
</managed-property>
```