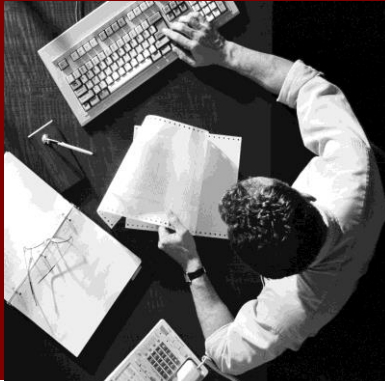




Developer Guide

SAP® Manufacturing **Execution 15.2** **Customization**

Using SAP® NWDI/NWDS

Target Audience

- Software Developers
- Technology Consultants

Document Version 2.0 – November 2018



SAP SE

Dietmar-Hopp-Allee 16
69190 Walldorf
Germany
T +49/18 05/34 34 34
F +49/18 05/34 34 20
www.sap.com

© Copyright 2018 SAP SE. 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. The information contained herein may be changed without prior notice.

These materials are subject to change without notice. These materials are provided by SAP SE and its affiliated companies ("SAP Group") for informational purposes only, without representation or warranty of any kind, and SAP Group shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP Group products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

Disclaimer

Some components of this product are based on Java™. Any code change in these components may cause unpredictable and severe malfunctions and is therefore expressly prohibited, as is any decompilation of these components.

Any Java™ Source Code delivered with this product is only to be used by SAP's Support Services and may not be modified or altered in any way.

Documentation in the SAP Service Marketplace

You can find this documentation at the following Internet address:
service.sap.com/instguides

Typographic Conventions

Type Style	Represents
<i>Example Text</i>	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths and options. Cross-references to other documentation
Example text	Emphasized words or phrases in body text, titles of graphics and tables
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example, SELECT and INCLUDE.
Example text	Screen output. This includes file and directory names and their paths, messages, names of variables and parameters, source code as well as names of installation, upgrade and database tools.
Example text	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
<Example text>	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.
EXAMPLE TEXT	Keys on the keyboard, for example, function keys (such as F2) or the ENTER key.

History of Changes

Version	Important Changes
1.00 (December 2015)	First version
1.1 (January 2016)	Updated the <i>Configuring the Default Formatting Strategy</i> section.
1.2 (June 2016)	Updated the <i>Customization Checklist</i> section.
2.0 (November 2018)	Changed the <i>Database Schema</i> section.

Contents

History of Changes	4
Overview	8
Prerequisites	8
Development System Installation	8
Developer Skillset	8
SAP NetWeaver Development Infrastructure and Developer Studio ..	8
SAP ME Functional Training	9
Development Environment Setup.....	9
Creating Required Components	9
Preparing Your Workbench.....	11
SAP ME Extension Development	13
Service	13
Concepts.....	13
Tasks.....	23
References.....	25
Service Extension	26
Concepts.....	26
Tasks.....	31
Standard Service Extension Activities	37
System Integration	37
Activity Hook	41
Concepts.....	41
Tasks.....	43
References.....	45
Document Printing	47
Concepts.....	47
Tasks.....	48
Example	49
Production Operator Dashboard (POD) Plug-Ins.....	53
Concepts.....	53
Tasks.....	53
Browseable Input Fields.....	59
Concepts.....	59
Creating a Browse.....	67
Session Timeouts	68
References.....	68
Legacy Browse IDs and Types.....	68
Initial Data Load	72
Concepts.....	72
Tasks	75

Creating Configuration Data with Initial Data Load.....	75
Database Schema	76
Concepts.....	76
Customizing the Database	76
Data Migration	77
Tasks	78
Creating Database Schema	78
Reference	79
Script DC Directory Structure	79
Additional Rules Regarding Custom SQL Scripts.....	81
OData	84
Security and Site Management.....	84
Available Services.....	84
Java Docs	84
References.....	85
General Guidelines	85
Basic Concepts and Steps	85
New Services without Access to CORE EDMs.....	86
New Services with Access to CORE EDMs	86
Creating and Deploying Extensions.....	93
Software Component Configuration	93
DC Configuration	93
Web Project Configuration.....	94
General Development Aspects.....	98
Building MII Extensions.....	98
Disclaimer	98
Localization	98
Concepts.....	98
Tasks	100
Reference.....	101
Error Handling	102
Concepts.....	102
Tasks	103
Security	105
References.....	105
Logging	106
Customization Checklist	106
Specific Examples.....	108
Automatic POD Refresh	108
Procedure	108
Sample Code	109

Assemble Component Plug-In Service Extension	115
Background	115
Operational Specification	115
Service Requirements and Naming	115
SDK Process.....	116
Sample Code	117
Tasks	121
Migrating Global Configuration.....	121
Migrating Supporting Classes and Resources	123
Migrating Business Process Extensions	123
Migrating POD Plug-Ins	126
Migrating EJBs	127
Migrating Web Services.....	128
Migrating Web Service Clients	128
Migrating the Web Extension Application.....	129
Migrating the Custom Servlets	129
Migrating SAP ME Login Module	129
Reference.....	131
DC Configuration by Extension Type.....	131
API Reference.....	133

Overview

This guide is intended for software developers who customize SAP Manufacturing Execution. This section describes prerequisites and introduces you to the development environment preparation.

Prerequisites

Development System Installation

You use the Developer Workplace installation to install NetWeaver Developer Studio and your local NetWeaver AS Java instance. The SAP NetWeaver 7.5 Master Guide describes the Java Development use case for NetWeaver, including system landscape overview and installation media.

Note: If you plan to develop printing extensions, then you must install the Adobe Document Services usage type and perform the [Basic ADS Configuration in a Java Environment](#).

For more information see [SAP NetWeaver 7.5 Installation and Upgrade Information](#) on the SAP Help Portal.

Developer Skillset

You should be proficient with the following core technologies and software:

- SAP NetWeaver Development Infrastructure (NWDI)
- SAP NetWeaver Developer Studio for NetWeaver 7.5 (NWDS)
- SAP NetWeaver Java Application Server 7.5 (basic administration and tools)
- Java EE 1.6 (EJB, Servlet/JSP, JSF, Web Services (JAX-WS))
- SQL
- SAP ME 15.2 Functional Training

SAP NetWeaver Development Infrastructure and Developer Studio

You develop SAP ME extensions using the SAP NetWeaver Development Infrastructure (NWDI) and NetWeaver Developer Studio (NWDS). NWDI and NWDS are the standard development tools for customization of Java based SAP products.

You are not required to have experience with NWDI and NWDS related technologies such as Web Dynpro, OpenSQL, or Java Dictionary since these technologies are not used for SAP ME extensibility. For example, NWDI and NWDS provide tools for creating Java Web Dynpro applications, but since SAP ME does not use Web Dynpro we cannot give any recommendation concerning its use in building custom applications for SAP ME. The scope of this document is limited to describing how you can build extensions using mechanisms specific to SAP ME.

You should be familiar with basic NWDI and NWDS concepts such as the Software Component Model, Development Components (DC), Development Configurations, and various services such as the Design Time Repository, Component Build System, and Change Management Services (CM Services).

Documentation is available on the SAP Community Network and SAP Help Portal you can refer to for information about how to install, configure, administer, and use NWDI and NWDS. Because of the extensive nature of these topics, this guide does not attempt to repeat that information here. If you are new to NWDI and NWDS, we strongly encourage you to review the product documentation before attempting to develop SAP ME extensions. The following references are recommended as a good place to start:

- [SAP NetWeaver Development Infrastructure](#)
- [SAP NetWeaver Developer Studio Basics](#)

SAP ME Functional Training

Understanding functional aspects of SAP ME is essential in deciding how to extend the system. You should at minimum have a good understanding of the following functional areas:

- Production activities such as start, complete, assembly, and non-conformance
- Activity hooks
- Document printing
- Custom Data
- Production Operator Dashboard

[SAP Help Portal](#) provides online help covering many of the functional aspects of SAP ME.

Development Environment Setup

Creating Required Components

The NWDI administrator creates a Software Components (SC) to package your SAP ME extensions. Within each SC, you must create a *Java EE Enterprise Application Development Component (DC)* and a *Java EE Web Module DC*.

This procedure describes how to create the DCs required to support SAP ME extension development.

Prerequisites

The NWDI administrator has created the software component definition in the System Landscape Directory (SLD) and a development configuration using CM Services.

The following table describes the software component dependencies that must be configured for the SAP ME extension software component:

Software Component	Name	Release
J2EE ENGINE FACADE	ENGFACADE	7.5
ENGINEAPI	ENGINEAPI	7.5

Software Component	Name	Release
JAVA FRAMEWORK OFFLINE	FRAMEWORK	7.5
SAP MFG EXECUTION CORE	SAPMECORE	15.1
DI BUILD TOOL	SAP_BUILDDT	7.5
ESI - WSRM	WSRM	7.5

For information about creating a software component, see [Creating a Software Component in the SLD](#). For information about creating a development configuration, see [Using CM Services](#).

Procedure

1. Import the development configuration from SLD containing the new SC (see [Working with Development Configurations](#)).
2. Open the *Development Infrastructure* perspective and synchronize the SAPMECORE[sap.com]/me/papi development component by right clicking and selecting *Sync / Create Project* → *Synch Archives*.
3. Create a *Java EE Web Module* type DC in your SC (see [Creating Development Components](#)). Use the following JSF settings:
 - a. *Use JSF: (selected)*
 - b. *JSF Version: 1.2*
 - c. *Use SAP Component Libraries for JSF: (selected)*
4. Remove the following JSF servlet and servlet mapping elements from the generated web.xml:

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-
class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Note: These entries are automatically added by the SAP ME build plug-in when the DC is built.

5. Add a *Build Time* dependency from your *Java EE Web Module* DC to the SAP ME build plug-in DC SAPMECORE[sap.com]/me/buildplugin-resources public parts:
 - def

- RESOURCES

Caution: This dependency enables a SAP ME specific build plug-in that is required for initializing services and building WPMF user interfaces. Failure to define this dependency will prevent initialization of SAP ME extensions.

If your Web DC uses mobile UI, use the `me/mobile-buildplugin-resources` dependency.

6. Create a `service-config.xml` in the root of your *Java EE Web Module* DC (see [Creating a Service Configuration Descriptor](#)).

7. Add import and service elements to enable service loading to your `service-config.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-config SYSTEM "http://www.visiprise.com/dtd/service-
config.dtd">
<service-config moduleId="com.vendor.web">
  <import moduleId="com.sap.me.activity" service="ExtensionLoaderParent"
/>
  <service name="ExtensionLoader" parent="ExtensionLoaderParent" />
</service-config>
```

Important: You can have multiple Java EE Web Module DC in your SC. This needs to be done only for one Web Module DC. However, there are no issues having multiple build plugin dependencies in different Web DCs, but it is not recommended, as it will increase SCA file size, and means will increase deployment time for custom code.

8. Create a *Java EE Enterprise Application* type DC in your SC, selecting your web DC as a referenced project (this will automatically add the web ASSEMBLY public part to your EAR).
9. Synchronize the `SAPMECORE[sap.com]/me/ear` DC.
10. Add *Deploy Time* and *Run Time* dependencies from your EAR to

`SAPMECORE[sap.com]/me/ear`.

Important: It is recommended to have only one EAR DC within custom Software Component.

Preparing Your Workbench

Configuring Archive File Association

In the course of development it is sometimes convenient to open JAR files from within the NWDS workspace. By default NWDS has no file association configured for JAR files. Use the following procedure to associate the WinRAR archive utility with the JAR file type.

Prerequisites

You have installed WinRAR on your system.

Procedure

1. Open *Window* → *Preferences* → *File Associations*.
2. Select *Add...*
3. Specify *File Type*: `*.jar` and click *OK*.
4. Go to *Associated Editors* and select *Add...*

5. Select *External Programs* → *WinRAR* archive and click *OK*.

Adding XML Catalog Entries

Several SAP ME extension types require you to make entries into the SAP ME service configuration descriptor file, `service-config.xml`. You can add an XML catalog entry in NWDS to help you create, validate, and autocomplete `service-config.xml` entries.

This procedure describes how to add an XML catalog entry in NWDS for `service-config.xml`.

Prerequisites

You have configured the [archive file association](#).

Procedure

1. Open the *Java EE* perspective.
2. Create a new resource project by selecting *File* → *New* → *Project...* → *General* → *Project* and name it **SAP ME Resources**.
3. Open the *Development Infrastructure* perspective.
4. Open the SAP ME service framework module archive at

`SAPMECORE[sap.com]/gen/default/public/api/lib/java/service-core.api-x.x.jar`
5. Select `service-config.dtd` and copy the file using the WinRAR function *File* → *Copy files to clipboard*.
6. Open the *Java EE* perspective.
7. Paste the service descriptor DTD into the *SAP ME Resources* project.
8. Select *Window* → *Preferences* → *XML* → *XML Catalog* → *Add*.
9. Select *Workspace...* → *SAP ME Resources* → *service-config.dtd* and click *OK*.
10. Enter **System ID** into the *Key Type* field.
11. Enter `http://www.visiprise.com/dtd/service-config.dtd` into the *Key* field.
12. Save your entries.

SAP ME Extension Development

Service

A business service is a unit of software with a well defined interface that provides manufacturing functionality. You can create, consume, expose, and extend business services to achieve new business requirements.

SAP ME exposes manufacturing functionality through business services. Examples of SAP ME business services include `SfcStartService` for starting SFCs and `AssemblyService` for assembling components.

Business services are documented in the SAP ME javadoc. See *API Reference* for technical documentation.

Concepts

SAP ME Service Framework

The service framework manages the registration, instantiation, and execution of business services and extensions.

Service Descriptor

You register a business service with the service framework by creating a service definition entry in the service descriptor `service-config.xml`. A service descriptor describes a set of services for a single service module and provides the framework with the information it needs to make the module's services available in the runtime system.

The service descriptor grammar is described by a DTD provided with the SAP ME core SC. The DTD is installed during the development environment configuration as described in [Adding XML Catalog Entries](#).

The following descriptor declares a single business service for the "com.acme.production.services" module:

```
<service-config moduleId="com.acme.production">

    <!-- Import the parent service configuration for our business
    services. All business services inherit the
    BusinessServiceParent service configuration.-->
    <import moduleId="com.sap.me.common"
    service="BusinessServiceParent"/>

    <!-- Import the ItemConfigurationService so that it can be
    injected into the new service. Similar to an EJB reference,
    the injected service is instantiated automatically and its
    interface is set as a field in the consuming service.-->
    <import moduleId="com.sap.me.productdefinition"
    service="ItemConfigurationService"/>

</service-config>
```

Service

```
<!-- New business service definition. You must declare the
service name, interface, class, and parent. The service name
is used by consumers to obtain a reference to the service's
interface. -->
<service name="MyService"
    interface="com.acme.production.MyServiceInterface"
    class="com.acme.production.impl.MyService"
    parent="BusinessServiceParent">
    <!-- Service injection. -->
    <property name="itemService"
        service-ref="ItemConfigurationService"/>
    <!-- Simple property injection of a String value -->
    <property name="myProperty" value="set me"/>
</service>
<service-config/>
```

Module ID

A module must be given a unique name using the `service-config` element's `moduleId` attribute. A name should be chosen such that it does not conflict with SAP ME core modules or other custom service module names. You can insure uniqueness by combining the name and vendor values of the containing DC according to the following format:

```
<vendor_id>.<dc_name>
```

The vendor ID value should be the reverse of the DC vendor and the DC name should use a period instead of forward slashes as a delimiter between the DC prefix and name. For example, if our example service descriptor is created in a DC by the software vendor `acme.com` and named `production/services`, then the module ID would be as follows:

```
<service-config moduleId="com.acme.production.services">
```

Service Import

Services from one module may refer to services from another module if they are imported with the `<import/>` element. Importing a service allows you to use it in the service definition of another service declared within the module. There are two cases in which you import a service:

1. You want to use a service as a parent for another service.
2. You want to declare the service as a property that will be injected into another service.

A parent service is a configuration construct that allows you to inherit the service definition of another service. This eliminates the need to define configuration elements that are frequently required by new services. For example, `BusinessServiceParent` must be used as the parent for every service that supports extension. Under the hood it provides the configuration necessary to make extensibility possible. If it were not available, you would need to repeat the same extensibility configuration in every business service you create.

Another reason to import a service is if you want to consume the service from another service in the module using a service property.

Service Properties

Service properties allow you to declare custom configuration values supported by your service. When the service framework instantiates the service it injects each property value into the service. The service Java interface must declare a corresponding Java setter method that accepts the property value.

A property can be either a simple string or a reference to another service. For example, the following property would cause the service framework to invoke the method `setMyProperty(String val)` with the value "set me":

```
<property name="myProperty" value="set me"/>
```

You can inject a reference to another service by declaring a property and using the `service-ref` attribute to specify the service name:

```
<property name="itemService" service-ref="ItemConfigurationService"/>
```

The corresponding Java interface method would be declared as follows:

```
public void setItemService(ItemConfigurationService itemService);
```

For more information, see [Property Injection](#).

Extension Service Configuration

You import the `Prototype` service as the parent for activity hooks and service extensions:

```
<!-- Import required to create a new service extension.-->
<import moduleId="com.sap.me.common" service="Prototype"/>

<!-- New service extension definition. You must declare the service
class name. -->
<service name="MyExtension"
    class="com.acme.production.serviceext.MyExtension"
    parent="Prototype"/>
```

A service extension configuration must declare `Prototype` as a parent.

You can also add custom properties to your extension configuration just as you can with business services (see [Service Properties](#)).

Service Lookup

You look up a service using one of the `getService()` methods defined in `com.sap.me.extensions.Services`. To look up a service you must pass the service module ID and name.

When you look up a business service, the framework instantiates a new instance of the service class and returns the service interface.

You look up core SAP ME business services using the following naming convention:

- Module ID is the Java package name of the business service interface
- Service name is the business service interface name without the “Interface” suffix

For example:

```
com.sap.me.certification.CertificationConfigurationServiceInterface  
com.sap.me.inventory.InventoryServiceInterface
```

The code to obtain the interfaces to each of these services would look as follows:

```
CertificationConfigurationServiceInterface certService =  
Services.getService("com.sap.me.certification",  
"CertificationConfigurationService");  
  
InventoryServiceInterface invService =  
Services.getService("com.sap.me.inventory", "InventoryService");
```

Property Injection

If you declare properties in your service configuration, you must create setter methods in your service that accept the property value. If the property is of type `String`, a string value will be set. If the property is a service reference, the dependent service’s interface will be set. The service framework will automatically look up dependent services.

The following example illustrates how to define class fields corresponding setter methods to accept `myProperty` and `itemService` properties:

```
public class MyService implements MyServiceInterface {  
    private ItemConfigurationServiceInterface itemService = null;  
    private String myProperty = null;  
  
    public void setItemService(ItemConfigurationServiceInterface  
itemService) {  
        this.itemService = itemService;  
    }  
  
    public void setMyProperty(String myProperty) {
```

```
        this.myProperty = myProperty;
    }
    // implementation of the rest of the class below
    ...
}
```

SAP ME Business Service API

Business service APIs can be categorized as configuration services and production services. Configuration services are responsible for maintenance of configuration data, such as routers, operations, resources, non-conformance codes, materials, and bills of material. Production services are responsible for execution of manufacturing processes, such as SFC start, SFC complete, non-conformance, and assembly.

This section describes key aspects of configuration and production service APIs.

Data Transfer Object

Business services use data transfer objects (DTO) as input and output parameters. A data transfer object is a Java bean whose purpose is to act as a container for a set of business object fields. Each field has a corresponding `set` and `get` methods for setting and retrieving the value of the field.

Configuration services use a class hierarchy of DTOs designed to give you only the data you need. The base class includes only key fields and all fields are provided by the full configuration DTO.

The following table describes the DTO class hierarchy, moving from the most basic DTO type to the final DTO containing all fields:

Level	Description	Example Classes
Key Fields	Key field DTOs expose a business object's primary keys and reference.	<code>ItemKeyData</code> , <code>OperationKeyData</code>
Basic Configuration	Basic configuration DTOs include key fields along with a basic set of root level fields (the exact fields included are based on those anticipated to be needed most often). Basic configuration does not contain child DTO collections.	<code>ItemBasicConfiguration</code> , <code>OperationBasicConfiguration</code>
Configuration	Configuration DTOs include basic configuration fields and add all remaining fields,	<code>ItemConfiguration</code> , <code>OperationConfiguration</code>

Level	Description	Example Classes
	including custom data and child DTO collections (wherever a parent/child relationship exists).	
Full Configuration	Full configuration DTOs include configuration fields and system details fields. Currently there is only one system details field called <code>modifiedDateTime</code>. This field is a timestamp managed by the ME persistence framework that records the last time that the business object was updated. Note: You should never change this field.	<code>ItemFullConfiguration</code>, <code>OperationFullConfiguration</code>

Object Reference

An object reference is a unique identifier for a business object generated by SAP ME when the object is created and persisted. SAP ME manages object references and you never create them directly. However, they are found in business service DTOs, so you should be familiar with how to recognize and use them.

Object references are `String` attributes in DTOs. By convention, an object reference attribute is named:

- `ref` - when it refers to an object itself
- `<buisness_object>Ref` - when one object refers to another object.

For example, `ref` in `com.sap.me.activity.ActivityKeyData` refers to an activity itself while `activityRef` in `com.sap.me.plant.WorkstationButtonActivity` refers to the activity associated with the workstation button.

You use finder methods to obtain object references. For example, in `com.sap.activity.ActivityConfigurationService` you can call `findActivityConfigurationByName(FindActivityByNameRequest request)` to obtain the object reference you need for `readActivity(ObjectReference request)`.

Configuration Services

Configuration services provide methods for maintenance of SAP ME configuration data including `find`, `create`, `read`, `update` and `delete`.

For example, these are some of the methods declared by

`ItemConfigurationServiceInterface`:

- `ItemFullConfiguration createItem(ItemConfiguration request)`
- `ItemFullConfiguration readItem(ObjectReference itemRef)`
- `ItemFullConfiguration updateItem(ItemFullConfiguration request)`
- `int deleteItem(ObjectReference itemRef)`
- `ItemBasicConfiguration findItemConfigurationByRef(ObjectReference itemRef)`

Configuration interfaces use the following naming convention:

`<business_object>ConfigurationServiceInterface`

where `<business_object>` is the name of the business object the service represents.

For example, the router configuration service is named

`RouterConfigurationServiceInterface` and the operation configuration service is named `OperationConfigurationServiceInterface`.

Read Methods

Read methods accept an object reference and return the full configuration for the object. Unlike finders, readers throw an exception if there is no business object matching the specified reference.

Readers can be easily identified since they begin with the word “read” followed by the business object name.

Create Methods

Create methods accept a configuration DTO and return the full configuration. Field values in the returned DTO may differ from the input DTO depending on whether the service sets a default value for fields not set in the input. For example, the `ItemConfigurationService` will set the item status to the product definition group default status if the status field is not specified in the input DTO.

Create methods can be easily identified since they begin with the word “create” followed by the business object name.

Delete Methods

Delete methods accept an object reference and return nothing. There is no indication given if the passed reference points to an object that does not exist (although having an invalid reference is highly unlikely). In other words, deleting an object that does not exist is considered semantically equivalent to deleting an object that does exist. The result is the same in both cases.

Delete methods can be easily identified since they begin with the word “delete” followed by the business object name.

Finders

Finders are methods that return one or more DTOs given a specific set of search criteria. Search criteria is itself a DTO where the value contained in each field defines a search criteria constraint (field values are combined together as a logical AND). Finder methods can be easily identified since they begin with the word “find” followed by the business object name.

Production Services

Production services are services that provide shop floor production functionality, such as assembling components into a parent assembly or advancing an SFC to the next step on a router. Production services will likely form the main staple of any extensions that you write, and so you should become as familiar as possible with the functionality that they expose.

The best way to learn production services is to browse the ME API javadoc. A good place to start is by looking through the interfaces defined in the `com.sap.me.production` package. This contains all of the central production service interfaces that you will use to work with SFCs.

Working with a Business Service API

To further explain, let's look at the `findItemConfiguration` method in the `ItemConfigurationServiceInterface`. The input is an `ItemSearchRequest` DTO which contains the following field definitions for item and the item revision:

```
protected String item;  
protected String revision;
```

Note that the item field is meant to hold the item ID and not a reference to the item (if the item field was in fact a reference of type `String` then it would be named "itemRef"). This highlights the overriding purpose of finders, which is to return anything that matches the values specified by the input DTO fields without any assumptions about the validity of the search criteria. In this case this means that no item need actually exist that matches the values of the item and revision fields.

You will find DTOs used for both method input and output method values. In fact, this is exactly what we find with the item configuration service `findItemConfiguration` method. The `ItemSearchRequest` parameter contains the item ID and item revision, while the `ItemSearchResult` return type contains a collection of `ItemBasicConfiguration` DTOs that match the item ID and revision specified by the request. In the case of the `ItemSearchRequest`, the revision value is optional, so if there was more than one item defined in the system, each having a different revision, then the finder would return an `ItemBasicConfiguration` object for each item revision.

But let's step back a moment. How did we know that the revision value was optional? For that answer let's go to the `ItemSearchRequest` javadoc and see what it tells us. Below is a snippet of the `ItemSearchRequest` class description:

Java class for `ItemSearchRequest` complex type.

The following schema fragment specifies the expected content contained within this class.

```
<complexType name="ItemSearchRequest">  
  <complexContent>  
    <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">  
      <sequence>  
        <element name="item"  
type="{http://www.sap.com/me/common}Item"/>  
        <element name="revision"  
type="{http://www.sap.com/me/common}Revision" minOccurs="0"/>  
      </sequence>  
    </restriction>  
  </complexContent>  
</complexType>
```

If you are familiar with reading javadoc you would probably expect to see a sentence or two describing the overall purpose of the class. What you would probably not expect to see is an XML schema definition of the class. This is because the code for all DTOs is generated from XML schema using JAXB. The schema definition is provided as part of the javadoc so that you can understand certain semantics about the DTO. For example, in the `ItemSearchRequest` schema we see that there are two elements, one for the item ID and the other for the item revision:

```
<element name="item" type="{http://www.sap.com/me/common}Item"/>
<element name="revision"
type="{http://www.sap.com/me/common}Revision" minOccurs="0"/>
```

What is important to note is the element's occurrence constraints, or multiplicity. This is where the semantics come in. Multiplicity is declared by the element's `minOccurs` and `maxOccurs` attributes and signifies whether the corresponding DTO field value is a required value (must be non-null) or whether it can be a collection of values. By default both `minOccurs` and `maxOccurs` have a value of 1, and so the "item" property shown above is a single value and must be set to a non-null value. The "revision" property explicitly declares `minOccurs="0"`, so this means that the revision is not required and may be left null.

Note: You can quickly see what fields are required for a given DTO by looking its constructor. Each DTO will implement a no-arg constructor and a constructor declares a parameter for each required field.

Although not shown in this example, if an element set `maxOccurs="unbounded"`, then this would signify that the field represented a collection of zero or more values. You typically see unbounded multiplicity where the field represents a parent/child relationship between two DTO types. For example, the `findItemConfiguration` method returns an `ItemSearchResult` DTO whose schema defines a single element with unbounded multiplicity:

```
<element name="itemList"
type="{http://www.sap.com/me/productdefinition}ItemBasicConfiguration"
maxOccurs="unbounded"/>
```

This means that `ItemSearchResult` is a parent DTO to the `ItemBasicConfiguration` DTO and exposes a list of `ItemBasicConfiguration` objects through the `itemList` property.

Note: You can also determine which DTO fields represent a collection by looking at the *Field Summary* section of the DTO javadoc and looking for fields with a Java type of `List`.

Now let's look at the `findItemConfigurationByRef` method. Here we see that the input is an `ObjectReference` DTO. On occasion you may see finders such as this one that act as a lightweight reader (readers are described below) for a business object. By "light weight" we mean that only a subset of the object's fields are returned rather than the full set returned by a `read` method. These finders serve as a convenience when you have an object reference but want to avoid the overhead of performing a full read.

If you poke around the ME API javadoc and look at other configuration service interfaces you will quickly realize that finders come in many flavors. This variation provides you with many ways to read business objects, but it also forces you to get to know them individually, service

by service. The best way to understand what a particular finder does is to read the method description and descriptions of its input and output.

Fortunately there are some general rules surrounding finder behavior that you can expect, regardless of its input and output.

- Finders that return a single result will always return null if no business object is found
- Finders that return a collection of DTOs will always return an empty collection if no matches are found

The first rule implies that a finder will never throw an exception if it cannot find a match. If you look at the javadoc for the `ItemConfigurationServiceInterface`, you will see that all finders do throw an exception of type `BusinessException`. This does not contradict the statement that finders never throw an exception. Remember, service API methods are extendible, meaning that there could be a service extension configured to execute before or after the finder is called. The service extension interface allows the service extension to throw a `BusinessException` if it encounters an error condition, and so the finder must also throw the exception to ensure that it is propagated back to the method caller.

The second rule implies that finders that return collections are null safe. This makes coding easier since you can safely iterate through the result without always having to check the result for null.

The topic of finders has been fairly extensive because there were many supporting concepts that needed to be described along the way. Fortunately many of these concepts are used throughout the API, so once you understand finders you are well on your way to understanding other APIs.

Service Security

SAP ME provides security layer to ensure that user is allowed to perform service invocation. Security around service invocation comes into play during lookup of the service and depending on the extension nature the appropriate method from `com.sap.me.extensions.Services` should be used. For more detailed information on specific methods, please refer to javadoc for `com.sap.me.extensions.Services`.

Supplying sufficient information to perform authentication and authorization in many cases is handled by SAP ME framework. However, there are scenarios when you must provide the values to the service lookup method. These situations occur when extension becomes an entry point for user and is not invoked from SAP ME, e.g. remote EJB, web service. In these cases, user will be authenticated by SAP NetWeaver AS Java. Authorization is not controlled by SAP ME so you must supply enough information to satisfy the following security checks at service lookup:

- User is authorized against security role `SAP_ME_USER` or `SAP_ME_INTEGRATOR`
- User is authorized against the site
- User is authorized against the activity

Authorization and SAP ME Security Roles

SAP ME defines two security roles that are important to take into account when developing extensions:

Security Role	Usage
SAP_ME_USER	Identifies users that are allowed to use user interface of SAP ME application
SAP_ME_INTEGRATOR	Identifies users that are allowed to use integration scenarios with SAP ME, eg. web services

When extension does not utilize SAP ME authorization, you must specify what security role should be enforced at service lookup, i.e. if the purpose of the extension is integration scenario, then `SAP_ME_INTEGRATOR` role should be used and `SAP_ME_USER` in other cases. At service lookup authorization of user against supplied security role will be performed.

Authorization and SAP ME Site

When extension is not invoked through SAP ME, the site will not be available. You must provide a site used for authorization. For example, if you have created a web service, you could include site field as part of the request and pass it to the service lookup method. Service lookup will ensure that user is authorized for the given site.

Authorization and SAP ME Activities

Similar to site concept, users must have permission to execute an activity. Extensions that are not invoked through SAP ME will not have activity set, so you must determine what activity should be used for authorization. Service lookup will ensure that user is authorized for the given activity.

Tasks

Creating a Service DC

You use the procedure below to create and configure your service DC.

Procedure

1. Create and configure a new DC for your services. This includes the creation of public parts and dependencies to SAP ME DCs (see [DC Configuration by Extension Type](#)).
2. Add the `ASSEMBLY` public part as a `Build Time` dependency to your *Java EE Enterprise Application* DC.
3. Create the service descriptor file `service-config.xml` (see [Creating a Service Configuration Descriptor](#)).

Creating a Service Configuration Descriptor

The service configuration descriptor `service-config.xml` is essential for implementing SAP ME business extensions. It contains information about business extensions that are exposed by the DC where the descriptor is located. For more information about service descriptor concepts, see [Service Descriptor](#).

Prerequisites

You have configured the service descriptor XML catalog (see [Adding XML Catalog Entries](#)).

Procedure

1. Open the Java EE perspective.
2. Right-click the java source folder of your DC and choose *New*→*XML*.
Note: If creating a descriptor for a Java EE web DC, the source folder is `/Java Resources/source`.
3. Enter `service-config.xml` in the *File name* field and choose *Next*.
4. Choose *Create XML file from a DTD file* and choose *Next*.
5. Choose *Select XML Catalog entry*.
6. Select the `http://www.visiprise.com/dtd/service-config.dtd` catalog and choose *Next*.
7. Accept all defaults on the *Select Root Element* screen and choose *Finish*.
8. Specify the name for the `moduleId` attribute according to module naming convention (see [Service Descriptor](#)).
Caution: Your `moduleId` must be unique across all SCs deployed with SAP ME.
9. Save your entries.
10. (**Java DC only!**): Add your `service-config.xml` to the `ASSEMBLY` public part of the DC by right clicking the file and selecting *Development Component*→*Publish as File*.
Caution: Do not add the service descriptor as a public part to a Java EE web DC. It will be automatically added to the WAR public part.

Creating a Service

You can extend SAP ME functionality with new services which implement custom business logic and collaborate with SAP ME core services via PAPI.

Prerequisites

- You have created a DC for your services (see [Creating a Service DC](#)).
- You have created a service configuration descriptor (see [Creating a Service Configuration Descriptor](#)).
- You have designed the service interface methods, including input/output DTOs and exceptions

Procedure

1. Open the JEE perspective and select the service DC.
2. Create DTO classes for all methods.
3. Create exception classes for all methods (see [Creating an Exception](#)).
4. Create the Java interface for the service and define all methods.
5. Create the Java class that implements the service interface.
6. Open `service-config.xml` and import the `BusinessServiceParent` service (see [Service Descriptor](#)).
7. Add the service configuration definition in `service-config.xml` and extend the `BusinessServiceParent` parent.

8. Add service interface, DTO, and exception classes to the `COMPILATION` public part.
9. Include all service classes in the `ASSEMBLY` public part.

References

SAP ME core services are documented in the SAP ME API javadoc. For more information, see [API Reference](#).

Service Extension

To extend a SAP ME business service, you create a service extension and attach it to the extension point you want to customize. Service extensions can be used to add the following:

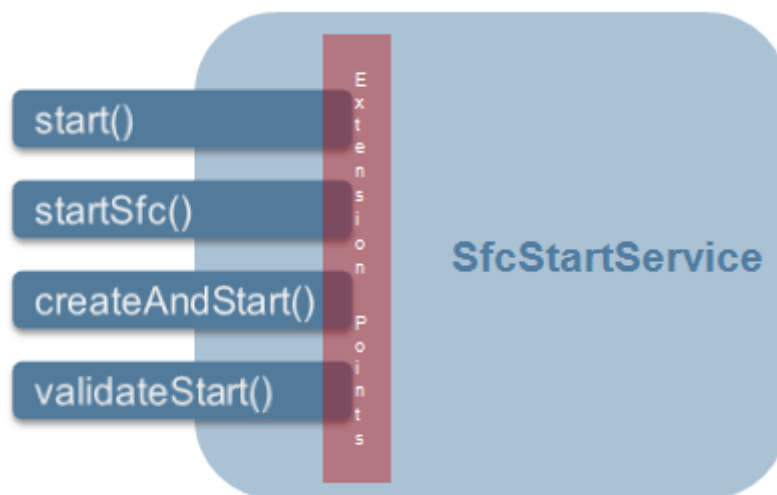
- New steps to a business process, such as adding new validations and integrating with external systems
- Specialized error handling

For a functional description of service extension, including execution mode, options, and filters, see [Service Extension Maintenance](#).

Concepts

Extension Points

From an extensibility point of view, each method defined by a business service serves as a point of extension and is therefore referred to as an **extension point**. For example, the `SfcStartService` declares the following methods, each of which is extensible using service extensions:



While all extension points are *technically* extensible, only a subset are *functionally* extensible. Technical extensibility means that if there are extensions attached to an extension point, then those extensions will be processed when the service method is executed. Functional extensibility means that the service method is technically extensible **and** is executed in response to an action triggered by the user while working with a SAP ME user interface. For example, the `start()` and `createAndStart()` methods are functionally extensible since they are executed from the Production Operator Dashboard start plug-in, while the `startSfc()` and `validateStart()` methods are only technically extensible since they are not triggered from any SAP ME user interface.

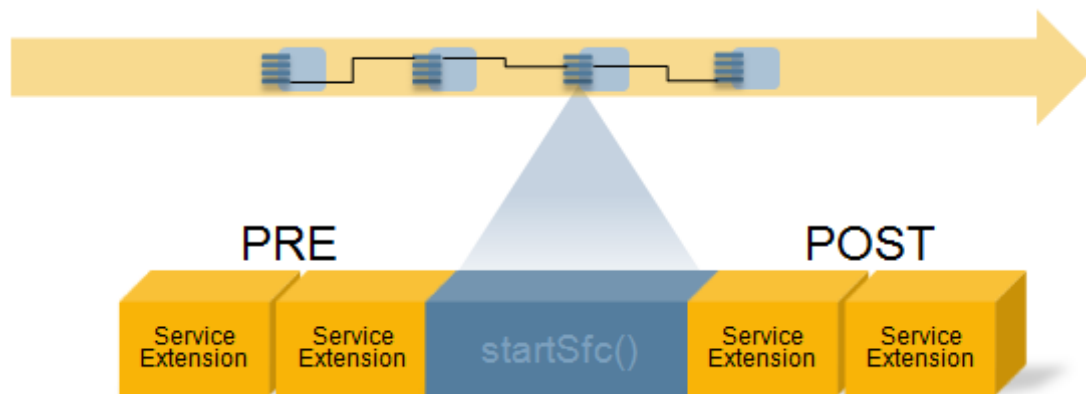
Knowing whether an extension point is functionally extensible is very important if you expect your extension to execute as part of a business function triggered from a SAP ME user

interface. To determine which extension points are functionally extensible, see [Identifying Functionally Extensible Extension Points](#).

Execution Points

Every extension point has three execution points where you can attach a service extension:

- PRE
- POST
- EXCEPTION

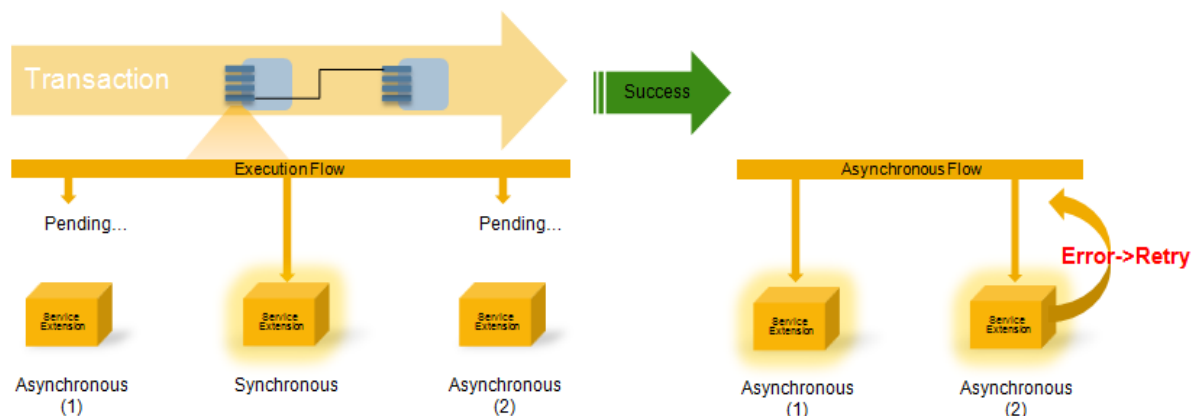


Service extensions attached to PRE are executed before the service method and those attached to POST are executed after the service method.

Service extensions attached to EXCEPTION can provide special handling of error conditions. The attached service extensions receive exceptions originating from the service method or other service extensions.

Execution Type

A service extension can be executed either synchronously or asynchronously. When executed synchronously, the service extension is executed immediately at the execution point. When executed asynchronously, the execution is not immediate, but rather triggered only if the overall transaction completes successfully.



If an exception is thrown from a synchronous service extension then the transaction is aborted. If an exception is thrown from an asynchronous service extension then you can throw a `com.sap.me.activity.PerformRetryException` with the number of retry attempts, a retry interval, and the retry type (retry properties). The runtime system catches this exception and uses the retry properties to reinitiate the extension execution process.

You may restrict the execution type to one mode or another by adding the `com.sap.me.activity.ExecutionType` annotation to the class (extensions are unrestricted by default).

Valid values to the `com.sap.me.activity.ExecutionType` are listed below:

<code>com.sap.me.activity.ActivityRuntimeType.SYNC</code>	The service extension must run as a synchronous extension.
<code>com.sap.me.activity.ActivityRuntimeType.ASYNC</code>	The service extension must run as an asynchronous extension.
<code>com.sap.me.activity.ActivityRuntimeType.ANY</code>	(Default) The service extension may run as either a synchronous or asynchronous.

If you set the execution type to `SYNC` or `ASYNC` then the user will not be able to change the mode in *Service Extension Maintenance*.

The following example illustrates a service extension that is set to the synchronous execution type:

```
import com.sap.me.activity.ActivityRuntimeType;
import com.sap.me.activity.ExecutionType;

@ExecutionType(ActivityRuntimeType.SYNC)
public class SfcCompleteExtension extends
ServiceExtension<CompleteSfcRequest> {

    public void execute(CompleteSfcRequest req) throws Exception {
        ...
    }
}
```

Options

Options are parameters the user can set using *Service Extension Maintenance* (see [Service Extension Maintenance](#)).

For example, the SAP ME base application provides the `HTTP_TRANSPORTER` activity (service extension) that supports an option called `HTTP_DESTINATION_NAME`. This option specifies the NetWeaver HTTP destination used by the extension to communicate with an external system. By making the HTTP destination configurable, the user can reuse the transporter in scenarios requiring different destination systems.

You expose an option by creating a setter method that accepts the option value and annotating the method with the `com.sap.me.activity.ActivityOptionField` annotation. The code snippet below illustrates the implementation of an option that allows the user to configure a URL value used by the extension to make an HTTP request:

```
private String url=null;

@ActivityOptionField
public void setUrl(String url) {
    this.url = url;
}
```

There are several design aspects that you should consider when creating extension options, including:

- Type
- Default value
- Validation

The following sections discuss details concerning extension option design.

Option Type

When you create an extension option you will need to choose the appropriate Java type based on the option's purpose. For example, an option representing a discrete numeric value should have the Java type `Integer` whereas an option representing a partial numeric value should have the Java type `Double` or `BigDecimal`.

When an extension option is displayed in *Service Extension Maintenance* its type is inspected and an input field matching the type is displayed. The following table lists the Java types that you may assign to your option and the input field that is displayed in *Service Extension Maintenance*:

Java Type	UI Input Field
String	Text
Boolean	Check box
Integer, Double, BigDecimal	Text (type enforced by UI component)
DateTimeInterface	Date Chooser
Enumeration	Select List

Caution: Use only Java object types instead of simple types such as `int` and `boolean`.

Option Default Value

You can implement a default value for an option by assigning an initial value and implementing a corresponding getter method. The default value is populated in the option input field when the extension configuration is displayed in *Service Extension Maintenance*.

This example illustrates setting a default value of 7 for a option that holds a quantity:

```
//Set the default quantity to 7
private Integer quantity = new Integer(7);

@ActivityOptionField
public void setQuantity(Integer quantity) {
    this.quantity = quantity;
}

//This getter is correlated with the quantity field because it complies with
//the standard JavaBean property naming convention.
public Integer getQuantity() {
    return this.quantity;
}
```

Note: You must provide a default value if an option is marked as required.

Option Validation

You implement special validation methods to insure that the user enters valid option values when the extension configuration is saved. For example, if you define an option that allows the user to specify an SFC quantity, then you could create a validation method to accept only non-negative quantities from the user:

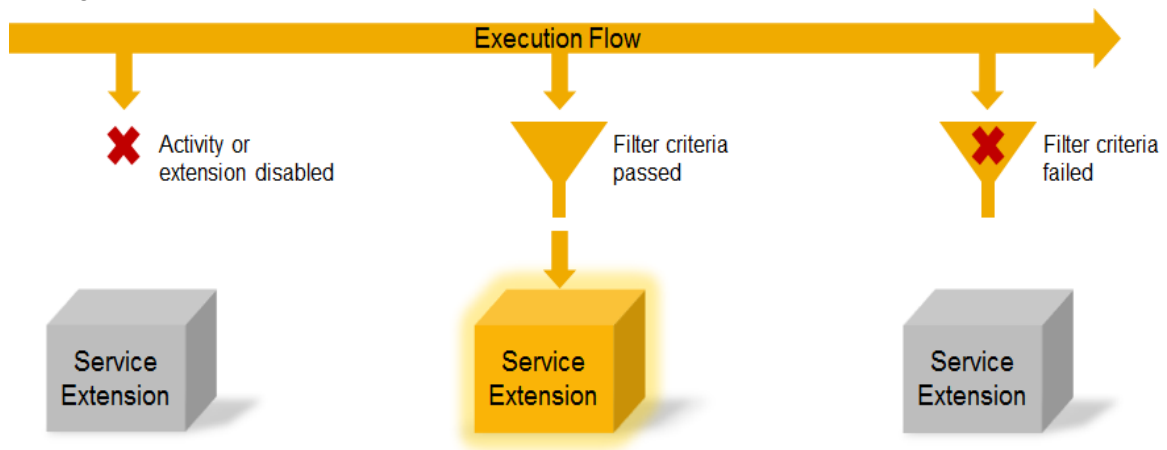
```
public void validateQuantity(Integer quantity) throws
ActivityOptionValidationException {
    if(quantity < 0) {
        throw new
InvalidNumericActivityOptionValidationException(22133, "QUANTITY");
    }
}
```

Validation can be applied to each option individually (as illustrated by the `validateQuantity` method) or as a group if the validity of one option depends on the value of another. The following validation method

Filters

While service extensions can be enabled/disabled at the activity level and the extension point level, it is sometimes necessary to enforce fine grained control over extension execution. Filters allow the user to define filter criteria to insure that the extension is executed under specific conditions. Unlike extension options, filters are a run-time configuration feature and require no development.

The figure below illustrates an execution flow that includes three service extensions:



The first extension is not executed because it is not enabled at the activity level or the extension point level. In this case any filter criteria that is configured will not be evaluated. The second extension is enabled and defines filter criteria that evaluates successfully, and so the extension is executed. The third extension is also enabled, but filter criteria evaluation fails and so the extension is not executed.

Filter criteria may include any combination of service method input values, user, and activity. However, not all method input values are available as filter criteria. Numbers, strings, booleans are always available as filter criteria but object references are restricted to the following types (see [Object Reference](#)):

- SFC
- Operation
- Resource
- NC code
- BOM component
- Workstation

Tasks

Enabling Service Execution Tracing

As described in [Extension Points](#) concepts, functionally extensible extension points are service methods that are executed as the result of user interaction with an SAP ME user interface. SAP ME provides a logging location that will trace the execution of business services so that you can determine which extension points to extend. Once enabled you can

execute your functional scenario and review the trace log to see which extension points are executed.

Procedure

1. Open the SAP Management Console perspective in NWDS.
2. Open the logging configuration for your AS Java instance by expanding `Local/<SID>/<INSTANCE>/AS Java/Log Config`
3. Navigate to the following logging location:
`com/sap/me/frame/service/BusinessServiceExecutionInterceptor`
4. Enable *Severity* **ALL**.
5. Open the *Log File View* and show the default trace file.
6. Open the user interface and execute the user action. For example, if you wanted to trace the extension points executed when an SFC is started, you would open the POD and start an SFC.
7. Pause logging so that you can inspect the trace.

Each log entry represents a trace of a business service invocation made from the user interface and begins with the service reference of the service executed followed by the execution time. There is typically several log messages that correspond to a single user action (it is common for a user action to trigger several UI events that each result in a business service invocation).

When you open a log message you will see the execution chain of all services invoked by the originating service. The log output also includes a trace of the input and output DTOs for each invocation.

Creating a Service Extension

You extend `com.sap.me.activity.ServiceExtension` to create your service extension. As a convenience, this class implements several required interfaces related to the service extension context. The context includes the extension configuration, service method details, and access to general purpose properties that can be passed between extensions at run time.

Prerequisites

- You have created a DC for *Service Extension* extension type (see [Creating a Service DC](#)).
- You have designed your service extension and considered the following aspects:
 - The API methods the service extension will extend and the API method connectivity status
 - The execution modes the service extension will support
 - Options that will be available for configuration in *Service Extension Maintenance*
 - Default filter criteria to control under what conditions the extension will execute
 - What services the extension will use

Procedure

1. Open the JEE perspective and select the service extension DC.

2. Create a service configuration for your extension (see [Extension Service Configuration](#)).
3. Create a new Java class that extends `com.sap.me.activity.ServiceExtension` and accepts the appropriate generic type.
4. Add the service extension class to the DC ASSEMBLY public part by right clicking the Java file and selecting `Development Component → Publish as File`
5. Set the execution mode using the `com.sap.me.activity.ExecutionType` annotation (see [Execution Modes](#)).
6. Add service extension options that you want to support (see [Options](#)).
7. Implement the `TransactionContextSetter` interface if you want access to the transaction context.
8. If you defined service configuration properties for injection, add property setter methods (see [Service Properties](#)).
9. Create an initial data load definition for the extension activity and extension configuration (see [Using Initial Data Load for Service Extension Configuration](#)).

Example

```
@ExecutionType(ActivityRuntimeType.SYNC)
public class SampleServiceExtension extends ServiceExtension implements
TransactionContextSetter {
    private static final long serialVersionUID = 1L;

    //Extension option property
    private void String stringProperty = "This is a String";

    // Interface providing access to the transaction context
    private TransactionContextInterface transactionContext;

    public SampleServiceExtension() {
        super();
    }

    //Implementation of ActivityInterface
    public void execute(Object input) throws Exception {
        //TODO This is what your extension does
    }

    //Setter to accept the TransactionContextSetter interface
    public void setTransactionContext(TransactionContextInterface
                                    transactionContext) {
        this.transactionContext = transactionContext;
    }

    //Setter method for the STRING_PROPERTY extension option.
    @ActivityOptionField(required = true)
    public void setStringProperty(String stringProperty) {
        this.stringProperty = stringProperty;
    }

    //Method to return a default value for the STRING_PROPERTY when a new
    //service extension configuration is created.
    public String getStringProperty() {
        return this.StringProperty;
    }

    //Method to validate the STRING_PROPERTY value when the service
    //extension configuration is saved.
    public void validateStringProperty(String val) throws
        ActivityOptionValidationException {
        //TODO put the validation logic here.
    }
    ...
    ...
}
```

Using Initial Data Load for Service Extension Configuration

The use of Initial Data Load to configure your service extension will simplify and speed up the configuration of the extension for the end user. Normally after deployment, the end user has to create an activity and then has to attach it to an extension point. Service extension configuration via Initial Data Load will automatically load this configuration when the user loads initial data for the site.

Procedure

1. Create an IDAT entry to create an activity for your service extension via `ActivityConfigurationService#createActivity` (see [Creating Configuration Data with Initial Data Load](#))
2. Specify the input parameters for the activity definition:
 - `activity`
ID of the new activity must be uppercase
 - `description`
Description of the new activity
 - `enabled`
Set it to `true`. This will cause your activity to be enabled as soon as it is created.
 - `executionType`
Set this to `SERVICE`
 - `classOrProgram`
Fully qualified service name for service extension in the form of `<moduleId>$<serviceName>`.
For example
`com.acme.production.serviceplugin$LogServicePlugin`.
 - `visible`
Set to `false` because this is not a GUI activity.
 - (Optional) `activityOptionList`
Used to define activity rules for this new activity. Add as many `activityOptionList` items as you have rules. There are two sub-items for `activityOptionList`:
 - `execUnitOption`
Activity rule name must be upper case
 - `setting`
Activity rule value
3. Create IDAT entry for your extension configuration via `ExtensionConfigurationService#createExtension` as described in [Creating Configuration Data with Initial Data Load](#).
4. Specify the input parameters for service extension definition:
 - `extensionName`
Name of the extension must be upper case
 - `extensionPoint`
The fully qualified extension point, including module ID, service name, and service method. The extension point is specified using the following format:

```
<moduleId>${<service_name>#<method_name>
```

For more information see [Extension Points](#).

- **activityRef**
Reference to the activity you created above.
For example:


```
<activity:activityRef>
  <reference type="activity">
    <activity>VENDOR_LOG</activity>
  </reference>
</activity:activityRef>
```
- **sequence**
Execution order for this extension
- **enabled**
true or false, used to enable or disable this specific usage of your service extension
- **asynchronous**
true or false, indicates that the extension must be executed asynchronously
- **extensionInvocationType**
PRE, POST or EXCEPTION. For more information see [Extension Points](#).
- **activityOptionList**
Used to define service extension options for your extension. Add as many **activityOptionList** items as you have options for the extension. There are two sub-items for **activityOptionList**:
 - **execUnitOption**
Extension option name must be uppercase
 - **setting**
Extension option value
- **extensionFilterList**
Filters for the service extension. Add as many **extensionFilterList** items as needed. There are three sub-items for **extensionFilterList**:
 - **filterName**
 - **enabled**
true or false, used to enable or disable this specific filter
 - **extensionFilterFieldList**
List of filter fields. Add as many **extensionFilterFieldList** items as needed. There are four sub-items for **extensionFilterList**:
 - **filterFieldName**
 - **filterValue**
 - **filterContextRef**
You specify **filterContextRef** if the filter field is one of the support filter object references:
 - **ref**
Specify reference to an object
 - **filterContextType**
Type of the referenced object

Standard Service Extension Activities

System Integration

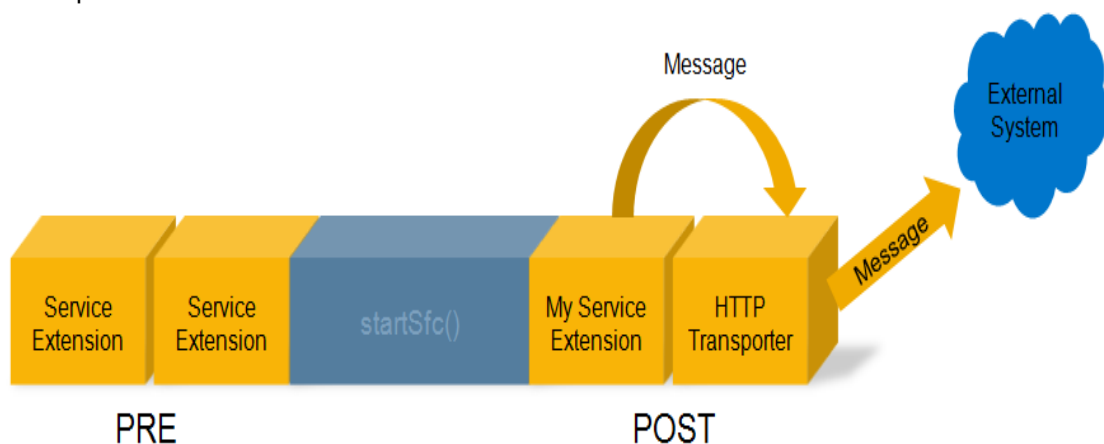
Concepts

SAP ME provides the following standard service extension activities you can use to send messages to external systems when specific actions occur on the shopfloor:

- HTTP_TRANSPORTER
- SAPMEINT_DB_TRANS

See [System Integration](#) in SAP ME application help.

Each system integration activity provides a transport mechanism you can use in conjunction with your extensions for outbound communication. For example, you can create an extension to generate a message when an SFC starts and use HTTP_TRANSPORTER to send it over the HTTP protocol to an HTTP destination.



Formatting Strategy

WEBSVC_TRANSPORTER activity always transports messages without modification.

HTTP_TRANSPORTER and SAPMEINT_DB_TRANS activities format messages before they are transported depending on the Java type of the message. When a `String` type message is passed in, it is transported without modification. For any other Java type, the `Object` is marshaled to XML before it is transported. You can override the default formatting behavior in two ways:

- Create a new formatting strategy implementation.
- Modify the existing formatting strategy with custom parameter settings.

The existing formatting strategy uses JAXB to marshal objects to XML. See [Java Architecture for XML Binding \(JAXB\)](#). You can use the following properties with the existing formatting strategy to customize default JAXB bindings:

Property Name	Description
namespace	Property value is added to JAXB namespace.
contextPath	Property value is added to JAXB contextPath.
rootElementName	Default XML root element is replaced with property value.
useDefaultPrefixes	Default JAXB prefixes (for example, ns1, ns2, common) are replaced with lowercase name of the marshaled class. Default: true

Tasks

Transferring a Message through Invocation Context

You use Invocation Context to transfer a message from your extension to a standard activity.

Prerequisites

You have created a DC for *Service Extension* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a new service extension that places a message in Invocation Context under a specific key.

Example:

```
public class MyServiceExtension extends ServiceExtension {
    public static final String MY_MESSAGE_KEY =
        "com.acme.activity.MyServiceExtension.MY_MESSAGE_KEY";

    public void execute(Object input) throws Exception {
        String myMessage = "This is my message.";
        invocationContext.setProperty(MY_MESSAGE_KEY, myMessage);
    }
}
```

2. Create a new extension configuration for your service extension.
3. Create a new extension configuration on the same execution point as your service extension that uses a system integration activity:
 - a. Set extension `Execution Sequence` to a number higher than your service extension.

- b. Set `MESSAGE_SOURCE` to `SERVICE_EXTENSION` and `MESSAGE_KEY` to the key your service extension that places the message in Invocation Context.

Creating a New Formatting Strategy

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a new class that implements `com.visiprise.frame.service.StrategyInterface` and returns a formatted string from the `execute` method.

Example:

```
import com.visiprise.frame.service.StrategyInterface;
public class MyObjectToXmlStringFormattingStrategy implements
    StrategyInterface<Object> {

    public Object execute(Object input) throws Exception {

        // Implement your own Object formatting logic here
        // and return a formatted string.

    }
}
```

2. Create a `service-config.xml` entry for your new formatting strategy.

```
<service name="MyMessageFormattingStrategy"
    interface="com.visiprise.frame.service.StrategyInterface"
    class="com.acme.common.MyObjectToXmlStringFormattingStrategy"
    parent="Prototype">
</service>
```

3. Create a `service-config.xml` entry for a new service that uses the system integration activity implementation class and your new formatting strategy.

The implementation class for `HTTP_TRANSPORTER` is `com.sap.me.integration.impl.HttpTransporterActivity` and the implementation class for `SAPMEINT_DB_TRANS` is `com.sap.me.collaboration.sap.MIIMessageDBTransportActivity`.

```
<service name="MyMIIMessageDBTransport" parent="Prototype".
    interface="com.visiprise.frame.service.ext.ActivityInterface"
    class="com.sap.me.collaboration.sap.MIIMessageDBTransportActivity">
    <property name="formattingStrategy"
        service-ref="MyMessageFormattingStrategy"/>
</service>
```

4. Create a new activity for your new service.
5. Create a new extension configuration for your new activity

Configuring the Default Formatting Strategy

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a `service-config.xml` entry for a formatting strategy with default `com.sap.me.common.impl.ObjectToXmlStringFormattingStrategy` implementation class with custom parameter values.

```
<service name="MyMIIMessageFormattingStrategy"
  interface="com.visiprise.frame.service.StrategyInterface"
  class="com.sap.me.common.impl.ObjectToXmlStringFormattingStrategy"
  parent="Prototype">
  <property name="namespace" value="http://www.sap.com/me/erp"/>
  <property name="useDefaultPrefixes" value="false"/>
  <property name="methodType" value="xml"/>
</service>
```

2. Create a `service-config.xml` entry for a new service that uses the standard activity implementation class and the new formatting strategy service.

The implementation class for `HTTP_TRANSPORTER` is `com.sap.me.integration.impl.HttpTransporterActivity` and the implementation class for `SAPMEINT_DB_TRANS` is `com.sap.me.collaboration.sap.MIIMessageDBTransportActivity`.

```
<service name="MyMIIMessageDBTransport" parent="Prototype"
  interface="com.visiprise.frame.service.ext.ActivityInterface"
  class="com.sap.me.collaboration.sap.MIIMessageDBTransportActivity">
  <property name="formattingStrategy"
    service-ref="MyMIIMessageFormattingStrategy"/>
</service>
```

3. Create a new activity for your new service.
4. Create a new extension configuration for your service activity.

Activity Hook

An activity hook is a type of business extension that allows you to customize the behavior of a limited set of SAP ME activities.

Note: Service extensions should be the first option to consider when you need to implement activity hook. Service extensions support all the activity hooks features and provide advanced features that simplify development and configuration.

Each activity that supports activity hooks defines a set of locations called “hook points” where activity hooks can be attached, executed, and unattached when no longer needed. For example, the “Start” activity includes the hook points `PRE_START` and `POST_START`, meaning activity hooks can be attached before start processing begins and after the start is completed.

For more information about hook point activities, see [Hook Point Activity](#) in the SAP ME application help.

Concepts

ActivityInterface

The technical implementation of an activity hook is a Java class that implements `com.visiprise.frame.service.ext.ActivityInterface<T>`. `ActivityInterface` contains only the `execute(<T> request)` method. The `execute()` method takes a parameterized type as its input and has no return value. Execution completes successfully if there are no exceptions thrown.

You can throw an exception from an activity hook if you want to indicate an error during hook execution. Since the activity hook executes in the scope of the activity transaction, no database changes will be committed.

The input to the activity hook `execute` method is specified as a generic type parameter `<T>`. The type is determined by the type of the parameter associated with the hook point. For example, an activity hook attached to the `POST_COMPLETE` hook point is passed a parameter of type `com.sap.me.production.PostCompleteHookDTO`.

Note: The “DTO” suffix is meant to indicate that the input is a Data Transfer Object, modeled after the DTO design pattern.

For a complete list of all hook points and their associated DTOs, see the [Hook Points](#).

Note: In some cases it may be advantageous to create an activity hook that performs a generic function, and thus can be assigned to more than one type of hook point. If you create such a hook, the input type may be different depending on the hook points it is designed to use. In this case, you will have to use `java.lang.Object` as the parameterized type so that your hook can accept the DTO type of any hook point.

Activity Hook Context

`com.sap.me.activity.HookContextSetterInterface` is an optional interface that your hook can implement if you need information about the hook execution context. If your hook implements this interface, it is passed a `com.sap.me.activity.HookContextInterface` object that contains context information. Context information includes the name of the activity being executed, activity options (referred to as activity rules in Activity Maintenance), and hook point user arguments.

Activity rules are represented as name-value pairs, for example:

Name	Value
NC_CODE	TTL_FAIL
RESOURCE_TO_HOLD	SDM

The name length is limited to 26 characters and the value length to 1024 characters.

Activity rules are passed to the activity hook through the `HookContextInterface`. Since it is possible to define more than one rule, the hook context returns a collection of rules instead of a single value.

Warning: The administrator has no way of knowing what the valid values for activity options should be. Details about valid values must be manually communicated to the SAP ME administrator.

User arguments are specified when the activity is assigned to a hook point. User arguments are assigned per hook point, and therefore can vary if the same activity hook is assigned to more than one hook point. This is in contrast to activity options, which are global to all hook points defined for a given activity.

The value of the user argument is string limited to 256 characters and is passed to the hook through the `com.sap.me.activity.HookContextInterface`.

Note: If multiple user arguments are needed for a given hook point then they can be included in the same user argument value (i.e using comma separated values such as val1,val2,val3). This means that it is the responsibility of the activity hook to parse the user argument string to recover individual argument values.

Note: Whenever possible, it is recommended to use activity rules instead of user arguments to pass parameters to your activity hook. User arguments are more error prone from the user point of view (they can be typed in incorrectly), and require special parsing within the hook. If you need to specify parameters at the hook point level, you can create a copy of your activity with its own rules and assign it to the hook point.

Warning: The administrator has no way of knowing what the valid values for user arguments should be. Details about valid values must be manually communicated to SAP ME users.

Tasks

Creating an Activity Hook

An activity hook is a type of business extension that allows you to customize the behavior of SAP ME activities (see [Activity Hook](#)).

To create an activity hook you need to create a service, that is similar to service extension (see [Service Extension](#)).

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a new Java class that implements `com.visiprise.frame.service.ext.ActivityInterface` (see [ActivityInterface](#)).
2. Add the activity hook class to the DC ASSEMBLY public part by right clicking the Java file and selecting `Development Component`→`Publish as File`
3. Specify the input DTO for your `execute` method.

Note: The input DTO depends on the type of hook point where it will be attached (see [Activity Hook](#)).

4. Optional: Implement `com.sap.me.activity.HookContextSetter` interface so you can add activity rules and user argument to your activity hook (see [Creating an Activity Hook](#)).
5. Configure your activity hook in `service-config.xml`, specify `Prototype` as parent, and save your entries (see [Service Descriptor](#)).

Example:

```
<import moduleId="com.sap.me.common" service="Prototype"/>
<service name="LogActivityHook"
class="com.acme.production.hook.LogActivityHook"
parent="Prototype"/>
```

6. Create an initial data load definition for the activity hook (see [Using Initial Data Load for Activity Hook Runtime Configuration](#)).

Using Initial Data Load for Activity Hook Runtime Configuration

The use of Initial Data Load to configure your new activity hook will simplify and speed up the configuration of the activity hook for the end user. Normally after deployment, the end user has to create an activity. Initial Data Load will do that automatically, when the end user loads (or reloads) initial data for the site.

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Procedure

1. Create IDAT entry using `ActivityConfigurationService.createActivity` XML to create your activity hook (see [Using Initial Data Load for Service Extension Configuration](#)).

2. Specify the input parameters for service invocation request:

- `activity`
ID of the new activity must be uppercase
- `description`
Description of the new activity
- `enabled`
Set to `true` so your activity will be enabled as soon as it is created.
- `executionType`
Set to `SERVICE`
- `classOrProgram`
Fully qualified service name for activity hook in the form of
`<moduleId>$<serviceName>`.
For example `com.acme.production.hook$LogActivityHook`.
- `visible`
Visible in Activity Manager and it is set to *false* because it is not the GUI activity
- (Optional) `activityOptionList`
Define activity rules here. Add as many `activityOptionList` items as many rules you have. There are two sub-items for `activityOptionList`:
 - `execUnitOption`
Activity rule name must be uppercase
 - `setting`
Activity rule value

References

Hook Points

For a functional overview of hook points, see [Hook Points](#).

Hook Point	Input Type
PRE_START POST_START	com.sap.me.production.StartHookDTO
PRE_COMPLETE	com.sap.me.production.CompleteHookDTO
POST_COMPLETE	com.sap.me.production.PostCompleteHookDTO
PRE_BATCH_START POST_BATCH_START	com.sap.me.production.StartHookDTO[]
PRE_BATCH_COMPLETE	com.sap.me.production.CompleteHookDTO[]
POST_BATCH_COMPLETE	com.sap.me.production.PostCompleteHookDTO[]
PRE_SIGNOFF	com.sap.me.production.SignoffHookDTO
POST_SIGNOFF	com.sap.me.production.PostSignoffHookDTO
PRE_DISP	com.sap.me.nonconformance.PreDispositionHookDTO
POST_DISP	com.sap.me.nonconformance.PostDispositionHookDTO
POST_ORDER_RELEASE	com.sap.me.demand.ShopOrderReleaseHookDTO
POST_PROD_CHANGE	com.sap.me.production.ChangeItemRouterHookDTO
POST_ORDER_CLOSE	com.sap.me.demand.ShopOrderCloseHookDTO
PRE_MERGE POST_MERGE	com.sap.me.production.MergeHookDTO
PRE_SERIALIZE POST_SERIALIZE	com.sap.me.production.SerializeHookDTO
PRE_SPLIT POST_SPLIT	com.sap.me.production.SplitHookDTO
PRE_RMA_SFC_RECEIPT POST_RMA_SFC_RECEIPT	com.sap.me.demand.AddSFCHookDTO

Activity Hook

Hook Point	Input Type
POST_CONTAINER_SAVE POST_CONTAINER_CLOSE PRE_PACKING_SFC PACKING_VALIDATION REOPEN_CONTAINER CALC_DIMENSIONS	com.sap.me.container.PackingHookDTO
ADD_COMPONENT	com.sap.me.demand.AssembleComponentHookDTO
REMOVE_COMPONENT	com.sap.me.demand.RemoveComponentHookDTO
POST_INV_RECEIPT	com.sap.me.inventory.InventoryHookDTO
POST_PL_ADD POST_PL_REMOVE	com.sap.me.demand.ProcessLotHookDTO
PARSE_COMPONENT PARSE_SLOT_COMPONENT	com.sap.me.production.ParseComponentHookDTO

Standard Hook Activities

For information about SAP ME standard hook point activities, see [Standard Hook Point Activities](#).

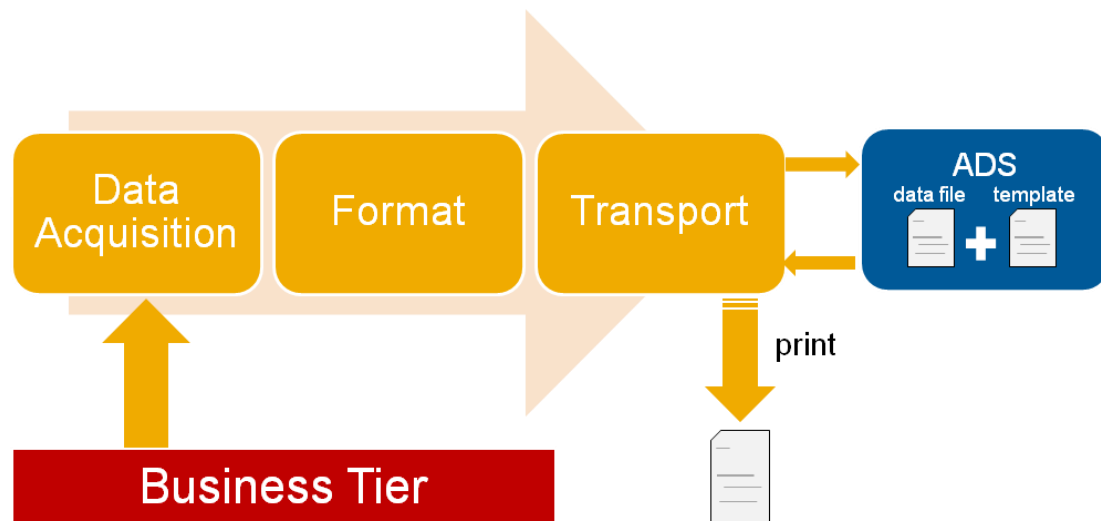
Document Printing

SAP ME Document Printing has many functional and administrative aspects that you should be familiar with before attempting to create a printing extension. For information about printing functionality, see [Printing of Production-Related Documents](#).

Concepts

Printing Workflow

Document Printing is based on a set of SAP ME business services that are executed to achieve the document printing workflow illustrated below:



The data acquisition step collects data according to the document options defined in Document Maintenance. The result of data acquisition is an XML string containing the collected data. The XML can be logged to the NetWeaver default trace so that you can view exactly what data is collected for a given document configuration (see [Binding SAP ME Data to XDP Templates](#) → *Creating a Sample SAP ME XML File*). This is especially important when considering whether the data you need is already provided, or whether you need to extend data acquisition to collect more data.

The format step accepts the acquired data XML and makes it available for further formatting. The SAP ME formatting service simply passes the XML through to the transport step since ADS does not require a particular schema or grammar for the XML. However, the transport service may be extended if you need to apply custom formatting.

The transport step creates a data file from the formatted XML and uses the ADS API to combine it with the document template created with Adobe LiveCycle Designer. This results in a printable file that is sent to the printer configured for the user's POD.

Printing Services

SAP ME provides printing services that implement each step of the printing workflow and are compatible with ADS. These services are assigned to a document using Document Maintenance.

Printing services may be extended or replaced with your own implementation. See the SAP ME [API Reference](#) for more information about SAP ME printing APIs.

Step	Interface
<i>Data Acquisition</i>	Java Interface: <code>com.sap.me.document.PrintingDataAcquisitionServiceInterface</code> Service: • Module: <code>com.sap.me.document</code> • Name: <code>PrintingDataAcquisitionService.</code>
<i>Format</i>	Java Interface: <code>com.sap.me.document.PrintingFormatServiceInterface</code> Service: • Module: <code>com.sap.me.document</code> • Name: <code>PrintingFormatAdsService</code>
<i>Transport</i>	Java Interface: <code>com.sap.me.document.PrintingTransportServiceInterface</code> Service: • Module: <code>com.sap.me.document</code> • Name: <code>PrintingTransportAdsService</code>

Tasks

Extending Printing Data Acquisition

While each step in the workflow may be customized by extending the underlying business service using a service extension, the most common customization scenario is to add more fields to the printed document by extending the data acquisition service and adding more data to the XML content.

Prerequisites

You have installed the Adobe Document Services usage type for your NetWeaver AS Java instance and performed the [basic ADS configuration](#).

Procedure

1. Create a new service extension (see [Creating a Service Extension](#)), assigning it to the POST execution point of the data acquisition service.
2. Add implementation to acquire the data you need and add it to the XML content (see the example below).
3. Create a sample XML data file and use it to create the ADS template using Adobe LiveCycle Designer (see [Binding SAP ME Data to XDP Templates](#) → *Creating a Sample SAP ME XML File*).
4. Configure the location of the ADS template using System Rule Maintenance and assign the template to the document configuration using Document Maintenance.
5. Use Document (Re)Print to test the extension and verify that the new fields are added to the printed document.

Creating a New Printing Service

You may also replace the transport service with your own custom service if you want to support document printing using a printing provider other than ADS. In most cases it is not necessary to replace the data acquisition or format service since the ADS data file format is generic XML. If your printing provider requires a specific XML structure or even a non-XML format, then you can create a service extension for the format service that transforms the XML into the format that you need.

Prerequisites

- You have determined whether the printing provider requires a format that is not compatible with the default XML format created by the default data acquisition service.

Procedure

1. Create a new formatting service by implementing `com.sap.me.document.PrintingFormatServiceInterface` and add the code necessary to format the XML content created by the default data acquisition service (see [Creating a Service](#)).
2. Create a new transport service by implementing `com.sap.me.document.PrintingTransportServiceInterface` and add the code necessary to create the printable document and send it to the printer.
3. Create an IDAT definition for the document.

Example

The following example illustrates a data acquisition service extension that adds SFC data to the XML printing content created by the default data acquisition service.

```
// Printing extensions cannot be asynchronous since their result may be
// consumed by the next print service in the printing work flow.
@ExecutionType(ActivityRuntimeType.SYNC)
public class SfcDataAcquisitionServiceExtension extends
ServiceExtension<PrintingDataAcquisitionRequest> {

    // Sfc data service injected via service config property.
    private SfcDataServiceInterface sfcDataService = null;
```

Document Printing

```

/**
 * ActivityInterface implementation required for the extension to be
 * executed.
 */
public void execute(PrintingDataAcquisitionRequest printingReq) throws
Exception {

    // The result DTO returned by the printing data acquisition service
    // contains the XML data content that we will add SFC data to.
    PrintingDataAcquisitionResponse printingResp =
    this.invocationContext.getResult();

    // Convert the XML data content to an XML document so that we can work
    // with it easier.
    Document doc = parseDefaultDataAcquisition(printingResp);

    // Use the injected SfcDataService to read the data for the SFC.
    FindSfcDataBySfcResponse findSfcDataBySfc = getSfcData(doc);

    // Add the SFC data to the document and convert back to an XML string
    // so that it is ready for the transport step.
    String xml = addSfcData(findSfcDataBySfc, doc);

    printingResp.setPrintContent(xml);
}

/**
 * Add the SFC data to the document.
 * @param findSfcDataBySfc
 * @param doc
 * @return XML with SFC data added
 * @throws JAXBException
 * @throws TransformerFactoryConfigurationError
 * @throws TransformerException
 */
private String addSfcData(FindSfcDataBySfcResponse findSfcDataBySfc,
Document doc) throws JAXBException,
TransformerFactoryConfigurationError,
TransformerException {

    Object obj = findSfcDataBySfc;
    if(findSfcDataBySfc.getClass().getAnnotation(XmlRootElement.class) ==
null) {
        String className = findSfcDataBySfc.getClass().getSimpleName();
        className = toLowerFirst(className);
        QName rootElement = new QName("http://www.sap.com/me/production",
className);
        obj = new JAXBElt(rootElement, findSfcDataBySfc.getClass(),
findSfcDataBySfc);
    }

    JAXBContext context =
JAXBContext.newInstance("com.sap.me.common:com.sap.me.production",
this.getClass().getClassLoader());

    Marshaller marshaller = context.createMarshaller();
    Node sfcDataNode = doc.createDocumentFragment();
    marshaller.marshal(obj, sfcDataNode);
    getSfcPrintingDataElt(doc).appendChild(sfcDataNode);

    Transformer transformer =
TransformerFactory.newInstance().newTransformer();
    StreamResult result = new StreamResult(new StringWriter());
    DOMSource source = new DOMSource(doc);

```

```

        transformer.transform(source, result);
        return result.getWriter().toString();
    }

    /**
     * Get the SFC data using the SFC reference from the XML.
     * @param doc
     * @return FindSfcDataBySfcResponse containing the SFC data
     * @throws BusinessException
     */
    private FindSfcDataBySfcResponse getSfcData(Document doc) throws
        BusinessException {
        Element sfcPrintElt = getSfcPrintingDataElt(doc);
        Element sfcRefElt =
            (Element) sfcPrintElt.getElementsByTagName("sfcRef").item(0);
        FindSfcDataBySfcRequest sfcDataRequest = new FindSfcDataBySfcRequest();
        sfcDataRequest.setSfcRef(sfcRefElt.getFirstChild().getNodeValue());
        return sfcDataService.findSfcDataBySfc(sfcDataRequest);
    }

    /**
     * Simple helper to retrieve the sfcPrintingData element from the XML
     * document.
     * @param doc
     * @return Element The sfcPrintingData element
     */
    private Element getSfcPrintingDataElt(Document doc) {
        Element sfcPrintElt =
            (Element) doc.getElementsByTagName("sfcPrintingData").item(0);
        return sfcPrintElt;
    }

    /**
     * Parse the data acquisition XML into an XML Document.
     * @param printingResp
     * @return The DOM Document
     * @throws ParserConfigurationException
     * @throws SAXException
     * @throws IOException
     */
    private Document
    parseDefaultDataAcquisition(PrintingDataAcquisitionResponse
        printingResp) throws ParserConfigurationException,
        SAXException,
        IOException {

        String printContent = printingResp.getPrintContent();

        DocumentBuilderFactory docFac = DocumentBuilderFactory.newInstance();
        DocumentBuilder docBuilder = docFac.newDocumentBuilder();

        ByteArrayInputStream is = new
            ByteArrayInputStream(printContent.getBytes());
        return docBuilder.parse(is);
    }

    /**
     * Setter required to set the SfcDataService configured in the service
     * configuration.
     * @param sfcDataService
     */
    public void setSfcDataService(SfcDataServiceInterface sfcDataService) {
        this.sfcDataService = sfcDataService;
    }

```

```
/**
 * Helper method to create the QName for the JAXB root element required
 * for marshalling.
 * @param className
 * @return The name of the class with the first character lower case
 */
private String toLowerFirst(String className) {
    return (className.length() > 0) ?
        Character.toLowerCase(className.charAt(0)) +
        className.substring(1) : className;
}
```

Production Operator Dashboard (POD) Plug-Ins

Concepts

POD plug-ins are reusable functional components with or without a corresponding visible user interface that can be defined within a [Production Operator Dashboard \(POD\)](#). The plug-ins can communicate and interact through events initiated by the user or other plug-ins.

When customizing an existing plug-in or POD through configuration is not sufficient to fulfill your requirements, it is possible you may need to create a new plug-in. WPMF uses JavaServer Faces to provide a clean separation between behavior logic and presentation. JSF is a standard, server-side, user interface framework for building Java Web applications. JSF is based on a Model View Controller (MVC) architecture, which separates the application logic from the presentation of data in a user interface.

The Model is the application data which resides on the server and is exposed using Java classes which JSF refers to as Managed or Backing beans. WPMF Plug-ins are in most cases can be thought of as managed beans and reside on the server.

The Controller is the server-side JSF Engine which manages the JSF lifecycle that starts when the client makes an HTTP request and the server returns a response which contains the information required to update the view.

The View is the user interface which is composed of several JSF components (for example, text fields, tables, etc.). JSF manages components in a component tree which is rendered as html. The View is displayed in a Web Browser Client.

UI Plug-ins are defined as backing and/or managed beans and provides the properties and methods that are associated with the views client-side components. In addition there are Non-UI Plug-ins that are not associated directly with a UI but execute business logic on the server. They are able to communicate to other UI Plug-ins and the framework for providing feedback to the user. Plug-ins communicate through defined events and shared models. The framework defines a set of API's for implementing events and models that can be consumed by other plug-ins.

The next sections describe the steps for creating a new WPMF plug-in and view for use by a POD.

Tasks

Creating New POD Plug-Ins

POD plug-ins are reusable functional components that can be defined within the SAP ME POD. The Plug-ins can communicate and interact through events initiated by the user or other Plug-ins.

Prerequisites

You have created a Web DC for *POD Plug-in* extension type

Procedure

1. Add your custom POD plug-in(s) to your web project by creating the plug-in managed bean classes:
 - a. Add your new plugin classes to the web project source folder for example Java Resources: source. It is recommended to create unique package name(s) using tokens specifically related to your organization name to avoid name conflicts.
 - b. Enter the Superclass for the plug-in. This should be either one of the following classes:


```
com.sap.me.production.podclient.BasePodPlugin
or
com.sap.me.wpfm.Plugin
```
 - c. Enter any interfaces you wish your plug-in to implement.
 - d. Chose *Finish*.
 - e. If the plug-in does not contain a view, add code to the plug-in superclasses "execute()" method which will be invoked when the plug-in is run.

2. Create a view for your plug-in if applicable as follows:
 - a. Open the *Web Application* project and right-click on the *WebContent* folder
 - b. Create your views, the jsp file(s), associated with the plug-ins. Ensure the page encoding and content type of each view (jsp) has the following:

```
<%@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:subview id="myPluginView">
    ... other JSF components and your content...
</f:subview>
```

3. Add plug-in (Managed Bean) definitions to the *faces-config.xml* file as follows:
 - a. Create a *faces-config.xml* file in *Web Application/web/WEB-INF* folder if one does not already exist by either manually creating it or adding the JSF facet to your web project.
 - b. Edit *faces-config.xml* in *Web Application/web/WEB-INF* folder and add the managed bean definition for the new plug-in as the following examples show:

```
<managed-bean>
  <managed-bean-name>myPlugin</managed-bean-name>
  <managed-bean-class>com.acme.web.module.MyPlugin
</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>beanName</property-name>
    <value>myPlugin</value>
  </managed-property>
```

```

<managed-property>
  <property-name>backingBeanName</property-name>
  <value>myPluginBackingBean</value>
</managed-property>
</managed-bean>

<managed-bean>
<managed-bean-name>myPluginBackingBean
</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>myPluginBackingBean</value>
</managed-property>
<managed-property>
<property-name>assignedBeanName
</property-name>
<value>myPlugin</value>
</managed-property>
</managed-bean>

```

4. Check the content of the `web.xml` file under `Web Application/web/WEB-INF` folder and remove all servlet definitions and mappings to `Faces Servlet` and `javax.faces.webapp.FacesServlet` if it exists.
5. Build the web module DC and the ear DC containing the web module DC and deploy the ear.
6. Create new activities of type "java" for each of the new POD plug-ins using *Activity Maintenance* and according to WPMF requirements as outlined in the WPMF Developer Guide that is available with other SAP ME Guides in SAP Service Marketplace.

Note: These activities must also be defined as IDAT file(s) within the web DC project (see [Using Initial Data Load for Service Extension Configuration](#)).

7. Configure SAPME to use the newly defined plug-ins within new POD(s) by
 - a. Creating a new custom Pod from scratch or by saving an existing POD under a different name. Your new plug-in activities can now be added to this new POD according to *POD Maintenance* guidelines and definitions (see [POD Maintenance](#)).
 - b. Create a new activity of type `jsp` for launching the custom POD in the previous step. Ensure the *Class/Program* field content on the *Activity Maintenance* screen is defined as a fully qualified URL (see [Activity Maintenance](#)):
`%PROTOCOL%: // %SERVER%: %PORT% / custom_webcontext / com / sap / me / wpmf / client / template.jsf?WORKSTATION=custom_pod_name`
 The values of `%PROTOCOL%`, `%SERVER%` and `%PORT%` are automatically substituted at runtime.

`custom_webcontext` is the context of the custom web application created by editing `application.xml`. Please note this is not – **manufacturing**, which is the web context for ME core. The `custom_webcontext` is specified in the `application.xml` file of the custom EAR DC.

`custom_pod_name` is the custom pod definition created in POD Maintenance.

- c. Assign this activity to a proper activity group of your choice and give it proper permissions to make it visible in the SAPME Activity Manager and launch the POD from there.

Customizing SAP ME Core POD Plug-Ins

Existing SAP ME POD UI plugins can be modified to reflect your specific requirements. Since extending the existing POD plugins are not allowed and the `faces-config.xml` associated with those plugins are not exposed, the plugin UIs need to have a way of reflecting your requirements without access to those entities.

Prerequisites

1. You have created a DC for *POD Plug-in* extension type (see [Creating a Service DC](#)).
2. The reader has read/ has access to the WPMF Guide and has a basic working knowledge of Java Server Faces JEE technology.

Cosmetic Changes

You need to perform cosmetic changes by repositioning some UI components within the UI. The SDK exposes the JSPs associated with each plugin and you can modify the copy of the original JSP by repositioning the UI components.

New UI Elements

You need to add additional UI components to an existing UI. The copy of the exposed JSP for the given plugin can be modified to add those new elements. However, the data binding and label/title localization for the new elements should be provided by creating new managed bean(s) that support the new elements and new local bundles if required. The data bindings to the existing (core) UI components should never be modified.

Table Component Binding For Custom Fields

The existing UI for the plug-in contains a table and the table needs to be changed. For instance, the table needs to have additional custom columns and add data bindings for the new columns, and the columns widths need to be set, etc. This is a more complex change and needs an understanding of how WPMF and POD tables are configured but it does not require any changes to the JSP.

The tables defined in `wmpf` and consequently the POD are configured through a binding object which is of type `TableConfigurator` in general and specifically of type `PodClientTableConfigurator` for POD Tables. Therefore, this object can be modified to reflect the required changes. The approach outlined here is limited to the POD tables that use the `PodClientTableConfigurator` object directly. There are cases where a plugin table has been authored using a custom subclass of this object instead. For those cases, this approach should generally work but it is not guaranteed.

A `PodClientTableConfigurator` is assumed to work with SAPME *List Maintenance* and as such obtains some of its configuration meta-data from the list definitions. This meta-data is

not configurable and can not be changed to reconfigure the table as the purpose of this facility in SAP ME is to present a unified way of presentation and configuration. Therefore, custom fields should be added through SAP ME custom data facilities and must be reflected on the appropriate list configuration in order to become available to a table.

The list name for a given list category can be obtained from the POD definition. This list name is used internally by a `PodClient` API call to obtain the configuration object(s) for a given category. Once a reference to the configuration object is available, the object can be intercepted at the appropriate JSF Phase and the elements within it can be modified.

A standard JSF Phase Listener must be used to facilitate the interception at the JSF PRE-RENDER phase. This phase listener needs to be added to your local `faces-config.xml` file. Therefore, the custom developer needs to add a new phase listener that does the interception and adds required modification in that phase as shown in the example below:

```
package com.sapdev.me.wpmf.impl;
import java.util.Iterator;
import java.util.Map;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

import com.sap.me.production.podclient.PodClientTableConfigurator;
import com.sap.me.production.podclient.PodOptionType;
import com.sap.me.wpmf.LayoutManagerInterface;
import com.sap.me.wpmf.util.FacesUtility;
/**
 * A sample Phase Listener to conceptually show the process of modifying a
 * table
 * at run time.
 */
public class TableConfiguratorPhaseListener implements PhaseListener {
    public void afterPhase(PhaseEvent arg0) {
        // do nothing
    }
    public void beforePhase(PhaseEvent arg0) {
        PodClientTableConfigurator config =
        getTableConfigurators(PodOptionType.POD_WORKLIST.value());
        if (config != null) {
            config.setColumnWidth("SFC", "45em");
            config.configureTable();
            // Add the table to the list of components to be updated when the
            // request returns.
            FacesUtility.addControlUpdate(config.getTable());
        }
    }

    public PodClientTableConfigurator getTableConfigurators(String
listCategory) {
        PodClientTableConfigurator res = null;
```

Production Operator Dashboard (POD) Plug-Ins

```

        if (listCategory == null) {
            return res;
        }
        LayoutManagerInterface layoutManager = (LayoutManagerInterface)
FacesUtility.getSessionMapValue("layoutManagerBean");
        String listName = layoutManager.getPodOption(listCategory);
        if (listName == null) {
            return res;
        }
        Map<String, Object> sessionMap =
FacesContext.getCurrentInstance().getExternalContext().getSessionMap();
        Iterator<String> keyIter = sessionMap.keySet().iterator();
        while (keyIter.hasNext()) {
            Object sessionObj = sessionMap.get(keyIter.next());
            if (sessionObj == null)
                continue;
            if (sessionObj.getClass() !=
PodClientTableConfigurator.class)
                continue;
            PodClientTableConfigurator configurator =
(PodClientTableConfigurator) sessionObj;
            if (listCategory.equals(configurator.getListType()) &&
listName.equals(configurator.getListName())) {
                res = configurator;
                break;
            }
        }
        return res;
    }

    public PhaseId getPhaseId() {
        // This must be set to the phase specified below
        return PhaseId.RENDER_RESPONSE;
    }
}

```

faces-config.xml:

```

<?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">
    <lifecycle>
        <phase-
listener>com.sapdev.me.wpmf.impl.TableConfiguratorPhaseListener</pha
se-listener>
    </lifecycle>
</faces-config>

```

Browseable Input Fields

Concepts

SAP ME provides a data selection mechanism called “Browse” to ensure data value integrity for the property/attributes that are required to have a valid predefined value. This mechanism consists of a User Interface that displays a list of valid data to be selected and the supporting server side service(s) to provide the data. The Browse User Interface is defined by WPMF and the supporting services are either predefined by ME (Legacy Browse) or can be defined by the custom developer (New Browse). Once a Browse is properly identified or defined, it can be attached to an Input field, within a UI Plug-in, that supports lookup (for example, Input Field with F4 Lookup).

Browse Framework

The New Browse is defined as a service and therefore configurable declaratively through service definitions. The injection mechanism provided by the service framework enables a custom Browse service to use any of the Public API methods provided by core SAP ME services or any custom defined service as its backend engine.

Declaration of Browse Configuration

Declarative approach is recommended when there is a simple correspondence between API service method and browse use case. The good examples are 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.

Example:

```
<!-- The parent BrowseService is the core ME service for all browse
definitions. -->
<service name="SampleOperationBrowseService" parent="BrowseService">
<!-- The finder service is injected into this browse service -->
  <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 in the DTO returned by the finderMethod-->
  <property name="columnPropertyBindings">
    <entry key="PRIMARY" value="operation"/>
    <entry key="REVISION" value="revision"/>
    <entry key="OPERATION_TYPE" value="operationType"/>
  </property>
```

Browseable Input Fields

```
<!-- You can define a startegy 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]" />
    <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 binding 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 binding defined).
- `bundleService`
A valid bundle service name to be used for localizations of values. If not defined, then a service using the SAP ME `LocaleSepcificText` is used (optional).
- `sortColumn`
The column name which 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).

The framework provides optional off the shelf transformational and formatting services, based on a `StrategyInterface`, which can be used as bindings for applicable or desired columns/properties. Any other custom developed strategy service that implements the same interface can also be used.

Strategy Services

The following strategy based services have been made available as services and can be used for adavanced column bindings. The service implementation for all these services do implement `com.visiprise.frame.service.StrategyInterface`.

```
<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 PAPI response collection to `BrowseItem`. By default the `fromField` and `toField` equal to 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"
```

Browseable Input Fields

```
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.

Accommodating Legacy Browses

This framework also accommodates the Legacy Browses through a special parent service called the `LegacyBrowseService` that can be used as a facade to the legacy code. Defining legacy browses as services provides uniformity and enables us to use the features provided by the service framework.

These legacy Browses are part of the core SAP ME and can only be referred to as they are. They are identified by an ID, used by the Input field, which facilitates the lookup of the Browse backend code.

In order to reuse existing browses anywhere in application, the `LegacyBrowseService.java` was developed. It implements `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 may not be appropriate for your 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>
```

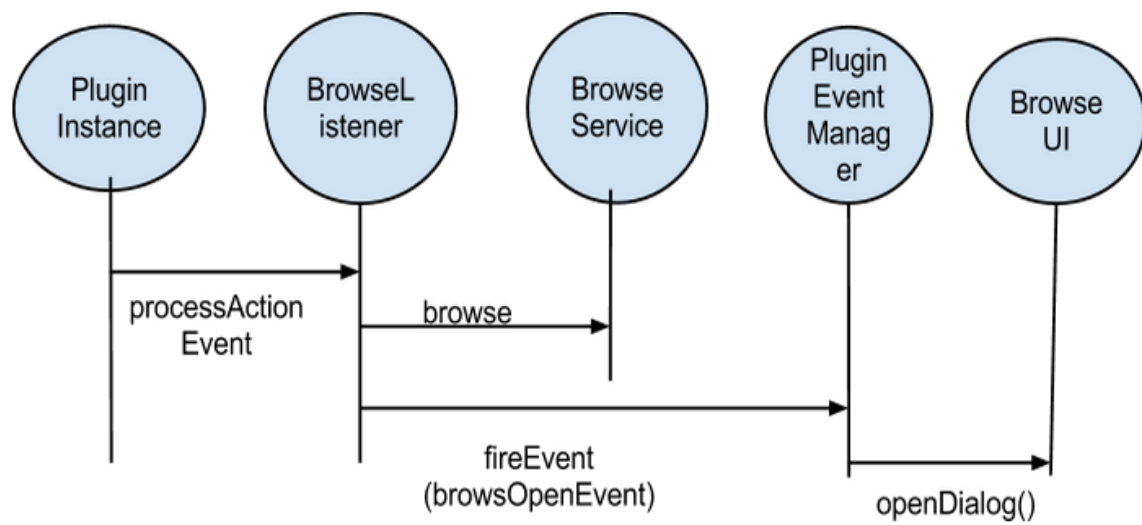
- `packagePrefix`
Indicates whether the browse is standard or custom. In most cases will be either "com.sap.me.browse." or "com.sap.me.browse.select."
- `browseId`
ID of the browse
- `searchField`
Indicator which field from `BrowseRequest` criteria map should be used as `SEARCH_VALUE`

WPMF and Browse Service Interaction

Any WPMF or POD plug-in 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:

Browseable Input Fields



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 plug-in author.

All the applicable attributes are defined below:

Attribute Name	Description
REQUESTING_PLUGIN_NAME	This is used to define the plugin bean name that is requesting a browse to be displayed. This is 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> searchMethodName(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 is used to define if the browse supports 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="extActInputField" width="16em"
        submitOnChange="false" submitOnFieldHelp="true"
value="#{simpleExtensionBean.extensionConfiguration.activityRef}"
title="#{gapiI18nTransformer['extensionTree.extensionActivity.TITLE']}">
```

Browseable Input Fields

```

        <f:attribute name="upperCase" value="true" />
        <f:actionListener
type="com.sap.me.wpmf.browse.BrowseListener"/>
        <f:attribute name="REQUESTING_PLUGIN_NAME"
value="newExtensionDialogPlugin" />
        <f:attribute name="SEARCH_METHOD_NAME"
value="newExtensionDialogPlugin.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="dfdfdf" />
        <f:attribute name="BROWSE_CALLBACK"
value="newExtensionDialogPlugin.processBrowseClosed" />
        <f:attribute name="BROWSE_DIALOG_WIDTH" value="20em" />
        <f:attribute name="BROWSE_DIALOG_HEIGHT" value="25em" />
    </sap:commandInputText>
</sap:panelGroup>

```

POD Completed and Nonconformed Values

These values are stored in session and not in PODSelectionModel. For customers using the custom plugin, ME core expects the values to be set in the session variable in following manner:

```

BigDecimal count = null;
BigDecimal sessionCount =
(BigDecimal)FacesUtility.getSessionMapValue("POD_COMPLETED_COUNT");
if (sessionCount != null && sessionCount.longValue() > 0) {
    count = sessionCount;
} else {
    count = getPodSelectionModel().getCompletedCount();
}

if (count == null) {
    count = BigDecimal.ZERO;
}
count = count.add(new BigDecimal(sfcSelectionList.size()));
getPodSelectionModel().setCompletedCount(count);
if (sessionCount != null && sessionCount.longValue() > 0) {
    FacesUtility.setSessionMapValue("POD_COMPLETED_COUNT", count);
}

BigDecimal sessionCount =
(BigDecimal)FacesUtility.getSessionMapValue("POD_NON_CONFORMED_COUNT");
if (sessionCount != null && sessionCount.longValue() > 0) {
    count = sessionCount;
} else {
    count = getPodSelectionModel().getNonconformedCount();
}

```

```
count);  
FacesUtility.setSessionMapValue("POD_NON_CONFORMED_COUNT",  
}
```

Tasks

Creating a Browse

SAP ME provides a data selection mechanism called "Browse". You can use a legacy Browse predefined in SAP ME, or you can create a new Browse (see [Browseable Input Fields](#)).

Prerequisites

1. You have read WPMF Guide and have a basic working knowledge of Java Server Faces technology
2. You have created a service for Browse extension type (see [Creating a Service DC](#)).

Procedure

Creating a New Browse

1. Define a new Browse service in your `service-config.xml`, ensure all prerequisite services are imported (for example, `BrowseService`), and save your entries (see [Declaration of Browse Configuration](#)).
 - Reference [API Reference](#) to determine the SAP ME business service and finder method that fits your Browse needs.
 - Reference [API Reference](#) for the DTO returned by your finder method to determine the property names that can be used for your column bindings.
 - Ensure each column has either a simple binding to a DTO property name or it has a strategy service binding that uses that DTO property to do the desired formatting or transformation.
 - Configure any optional Browse definitions (for example, sort column, bundle service) that fit your needs.
2. Configure your F4 lookup capable input field within your plug-in using your new Browse service as described in [WPMF and Browse Service Interaction](#).

Creating a Legacy Browse

1. Reference [Legacy Browse IDs and Types](#) for the list of available Browsers and choose `browseId` and `type` of Legacy Browse that fits your needs.
2. Configure your F4 lookup capable input component within your plug-in using your `browseId`.

For more information about configuration requirements and options, see WPMF Guide.

Session Timeouts

Note that since custom web application runs in its own web context it has its own http session. The ME core timeouts are documented in the SAP Note # 1825121. Timeouts of Core Standalone activities are documented in SAP Note # 2049640.

Please note that while testing timeout behavior it is important to keep the ME Core Maintenance UIs, PODs, ME Core Standalone UI – DC and NC Standalone and As-Built Configuration and Custom Apps in the own browser UIs. This is because they all can have independent timeout settings and opening all of them in the same browser session using multiple IE tabs can lead to unpredictable behavior.

Currently, custom web timeouts are set to default of 30 minutes. This can be overridden in NetWeaver Administrator.

References

Legacy Browse IDs and Types

```
#
# Provides information describing the Browse class to use to configure the
browse
#
# &browse.browseId = Browse ID (i.e. "WorkCenter" for
WorkCenterBrowse.java")
# &browse.type = Location ("custom" or "standard")
#
# Where &browse is browse ID assigned to the BrowseControl
#
# Note: "custom" or "standard" is used on the backend "BrowseFactory" class
to
# denote whether the "*Browse" class is found in one of the following
packages:
#
# "custom" = "com.sap.me.common.browse.server.custom"
# "standard" = "com.sap.me.common.browse.server.standard"
#
#####
#####
#

WORK_STATION.browseId=WorkStation
WORK_STATION.type=standard
NC_CODE.browseId=NCCode
NC_CODE.type=standard
WORK_CENTER.browseId = UserWorkCenter
WORK_CENTER.type = standard
USER.browseId = User
USER.type = standard
END_UNIT.browseId = EUNByWorkCenter
END_UNIT.type = standard
NC_CODE_BY_OPER.browseId = NCCode
```

```
NC_CODE_BY_OPER.type      = custom
COMPONENT_CUSTOM.browseId = ComponentCustom
COMPONENT_CUSTOM.type     = custom
VENDOR_FOR_COMPONENT.browseId = VendorForComponent
VENDOR_FOR_COMPONENT.type = standard
SFC_FOR_COMPONENT.browseId = SFCForComponent
SFC_FOR_COMPONENT.type     = standard
KIT_SFC_FOR_COMPONENT.browseId = KitSFCForComponent
KIT_SFC_FOR_COMPONENT.type = custom
INVENTORY_FOR_COMPONENT.browseId = InventoryForComponent
INVENTORY_FOR_COMPONENT.type = standard
KIT_INVENTORY_SFC_FOR_COMPONENT.browseId = KitInventorySFCForComponent
KIT_INVENTORY_SFC_FOR_COMPONENT.type = custom
INVENTORY_SFC_FOR_COMPONENT.browseId = InventorySFCForComponent
INVENTORY_SFC_FOR_COMPONENT.type = standard
LOCATION_SFC_COMPONENT.browseId = LocationSFCComponent
LOCATION_SFC_COMPONENT.type = custom
COMPONENT.browseId = Component
COMPONENT.type = standard
COMPONENT_ITEM.browseId = Component
COMPONENT_ITEM.type = custom
INVENTORY_ID.browseId = InventoryForComponent
INVENTORY_ID.type = standard
VENDOR.browseId = Vendor
VENDOR.type = standard

INVENTORY_TSM.browseId = InventoryTsm
INVENTORY_TSM.type = standard

ITEM.browseId = ItemDefault
ITEM.type = standard

ITEM_ASSEMBLY.browseId = Item
ITEM_ASSEMBLY.type = standard

ALT_COMPONENT.browseId = AlternateComponentForReqComponent
ALT_COMPONENT.type = custom

REASON_CODE.browseId = ReasonCodeByCategoryEnabled
REASON_CODE.type = standard

OPERATION.browseId = Operation
OPERATION.type = standard

OPERATION_ASSEMBLY.browseId = OperationAssembly
OPERATION_ASSEMBLY.type = standard

OPERATION_CUSTOM.browseId = OperationCustom
OPERATION_CUSTOM.type = custom
```

References

```
RESOURCE.browseId = Resource  
RESOURCE.type = standard
```

```
RESOURCE_BY_OPERATION.browseId = ResourceByOperation  
RESOURCE_BY_OPERATION.type = custom
```

```
SFC.browseId = SFCSimple  
SFC.type = standard
```

```
SHOP_ORDER.browseId = ShopOrderDefault  
SHOP_ORDER.type = standard
```

```
TOOL_NUMBER.browseId = ToolNumber  
TOOL_NUMBER.type = standard
```

```
TOOL_NUMBER_LOG.browseId = ToolNumberLog  
TOOL_NUMBER_LOG.type = custom
```

```
TOOL_GROUP.browseId = ToolGroup  
TOOL_GROUP.type = standard
```

```
WORK_CENTER_STANDARD.browseId = WorkCenter  
WORK_CENTER_STANDARD.type = custom
```

```
OPERATION_STEP_BY_SFC.browseId = OperationStepBySfc  
OPERATION_STEP_BY_SFC.type = custom
```

```
USER_GROUP.browseId = UserGroup  
USER_GROUP.type = standard
```

```
USER_GROUP_FOR_USER.browseId = UserGroupForUser  
USER_GROUP_FOR_USER.type = standard
```

```
CORRECTIVE_ACT_CODE.browseId = CorrectiveActCode  
CORRECTIVE_ACT_CODE.type = standard
```

```
LABOR_CHARGE_CODE.browseId = LaborChargeCode  
LABOR_CHARGE_CODE.type = standard
```

```
TOOL_NUMBER_BY_GROUP.browseId = ToolNumberByGroup  
TOOL_NUMBER_BY_GROUP.type = custom
```

```
INCIDENT_NUMBER_BY_INCIDENT.browseId = IncidentNumber  
INCIDENT_NUMBER_BY_INCIDENT.type = custom
```

```
ROOT_CAUSE_OPERATION.browseId = ValidRootCauseOperation  
ROOT_CAUSE_OPERATION.type = custom
```

```
REF_DES.browseId = RefDes  
REF_DES.type = standard
```

```
COMPONENT_SFC.browseId = ComponentSFC
COMPONENT_SFC.type      = custom

DATA_FIELD_VALID_VALUES.browseId = DataFieldValidValues
DATA_FIELD_VALID_VALUES.type      = standard

REASON_CODE_CUSTOM.browseId      = ReasonCodeCustom
REASON_CODE_CUSTOM.type          = custom

REASON_CODE_BY_CATEGORY.browseId = ReasonCodeByCategory
REASON_CODE_BY_CATEGORY.type      = custom

WORK_CENTER_CUSTOM.browseId = WorkCenter
WORK_CENTER_CUSTOM.type     = custom

ACTIVITY_POD.browseId = ActivityPOD
ACTIVITY_POD.type     = standard

ACTIVITY_POD_SELECTION.browseId = ActivityPODSelection
ACTIVITY_POD_SELECTION.type     = standard

EXTENSION_ACTIVITY.browseId = ExtensionActivity
EXTENSION_ACTIVITY.type     = standard

USER_WITHOUT_STATUS.browseId = UserWithoutStatus
USER_WITHOUT_STATUS.type     = custom
```

Initial Data Load

SAP ME provides initial data load (IDAT) for loading configuration data into the system from XML files. Configuration data includes SAP ME core configuration data, such as Activities, Operations, or NC Codes, as well as custom data you create. For example, each service extension and activity hook extension requires an Activity configuration before it can be executed by SAP ME.

Concepts

IDAT Execution

IDAT is triggered during the SAP ME installation process and by the end user from *Site Maintenance* when they create a site or reload data for an existing site (see [Site Maintenance](#)). Custom configuration data is always loaded after SAP ME core configuration data has been loaded. SAP ME core data is loaded even if an error occurs during the load of custom data.

Transaction Boundaries

Transaction boundaries of IDAT are defined by the XML file. All data defined within a single file is loaded within a single transaction. Any error that occurs during the processing of a single file will result in rollback of all data defined in the file and will abort the entire IDAT process. When a file is processed without error its data is permanently committed to the database.

Invocation Order and Dependent Data

The order in which individual IDAT files are processed cannot be configured and is therefore undefined. This means that you should not create data in one file that depends on the data created in another file. If one configuration data object depends on another, then both objects must be defined in the same IDAT file. For example, a service extension configuration depends on the definition of a service extension activity. You could place the service extension activity definition in `Activity.xml` and the service extension configuration in `ServiceExtension.xml`. However, IDAT may attempt to load `ServiceExtension.xml` before `Activity.xml`, which would result in an error since the service configuration references an activity that does not yet exist.

IDAT Service Descriptor Configuration

SAP ME IDAT initialization incorporates a discovery process that locates all IDAT files packaged within a DC. The discovery process is initiated by a special SAP ME service called `ConfigDataLoaderParent` that is configured in the service descriptor. Each DC that contains IDAT files must include this service configuration (configuration details are described in the procedure below).

The `ConfigDataLoaderParent` service requires that IDAT files are packaged according to the following Java package structure:

```
<module_id>.idat
```

where `module_id` is the module ID assigned to the service descriptor. For example, the following `service-config.xml` would load files packaged under `com.acme.production.idat`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-config SYSTEM "http://www.visiprise.com/dtd/service-
config.dtd">
<service-config moduleId="com.acme.production">
  <import moduleId="com.sap.me.plant" service="ConfigDataLoaderParent"/>
  <service name="IDATLoader" parent="ConfigDataLoaderParent"/>
</service-config>
```

Note: Even though IDAT files are XML files, they must be packaged as Java source just as you would Java class files (.java) or resource bundles.

For more information about service descriptor configuration, see [Service Descriptor](#).

IDAT XML Content

You can use the *View Input/Output* feature in *Service Extension Maintenance* to generate an XML template for the configuration data you want to load. The template can be copied into the XML file and populated with configuration data values.

IDAT content is the XML representation of SAP ME configuration service `create` methods (see [Create Methods](#)). You can load any type of configuration data, such as operations, materials, and activities. You can create more than one type of object as well as created several instances of the same type of object.

For example, this IDAT creates two activities and an extension configuration:

```
<common:serviceInvocationRequests
  xmlns:activity="http://www.sap.com/me/activity"
  xmlns:common="http://www.sap.com/me/common">

  <common:serviceInvocationRequest
    xmlns:activity="http://www.sap.com/me/activity"
    xmlns:common="http://www.sap.com/me/common">
    <moduleId>com.sap.me.activity</moduleId>
    <serviceName>ActivityConfigurationService</serviceName>
    <methodName>createActivity</methodName>

    <requests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:type="activity:ActivityConfiguration">
      <common:activity>LOGGER_EXT</common:activity>
      <common:description>Logger extension</common:description>
      <activity:enabled>true</activity:enabled>
      <activity:executionType>SERVICE</activity:executionType>
      <activity:classOrProgram>
        com.acme.production.serviceextension$LoggerServiceExtension
      </activity:classOrProgram>
      <activity:visible>false</activity:visible>
      <activity:activityOptionList>
        <activity:execUnitOption>
          ACTIVITY_RULE_DEFAULT
```

```

        </activity:execUnitOption>
        <common:setting>Some value</common:setting>
    </activity:activityOptionList>
</requests>
<requests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="activity:ActivityConfiguration">
    <common:activity>LOG_HOOK</common:activity>
    <common:description>Activity hook logger</common:description>
    <activity:enabled>true</activity:enabled>
    <activity:executionType>SERVICE</activity:executionType>
    <activity:classOrProgram>
        com.acme.production.hook$LogActivityHook
    </activity:classOrProgram>
    <activity:visible>false</activity:visible>
    <activity:activityOptionList>
        <activity:execUnitOption>
            LOG_INPUT_DTO
        </activity:execUnitOption>
        <common:setting>true</common:setting>
    </activity:activityOptionList>
</requests>
</common:serviceInvocationRequest>

<common:serviceInvocationRequest>
    <moduleId>com.sap.me.activity</moduleId>
    <serviceName>ExtensionConfigurationService</serviceName>
    <methodName>createExtension</methodName>
    <requests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="activity:ExtensionConfiguration">
        <activity:extensionName>EXT1</activity:extensionName>
        <activity:extensionPointId>
            com.sap.me.production$AssemblyService#assembleComponents
        </activity:extensionPointId>
        <activity:activityRef>
            <reference type="ACTIVITY">
                <activity>LOGGER_EXT</activity>
            </reference>
        </activity:activityRef>
        <activity:sequence>0</activity:sequence>
        <activity:enabled>true</activity:enabled>
    </requests>
    <activity:extensionInvocationType>PRE</activity:extensionInvocationType>
</common:serviceInvocationRequest>
</common:serviceInvocationRequests>

```

Troubleshooting

The loading of IDAT files is handled by SAP ME framework and it provides the following tracing locations to troubleshoot any possible issues during loading:

- com.sap.me.plant.impl.ConfigDataLoader

- `com.sap.me.common.impl.XmlInvocationService`

You must set the logging level to `Info` or `finer` to enable logging.

Tasks

Creating Configuration Data with Initial Data Load

Prerequisites

You have created a service descriptor in the Java DC that will contain your IDAT files (see [Service Descriptor](#)).

Procedure

1. Add the following entries to the service descriptor:

```
<import moduleId="com.sap.me.plant"
service="ConfigDataLoaderParent"/>
<service name="ConfigDataLoader" parent="ConfigDataLoaderParent"/>
```

2. Create a Java package for your IDAT files (see [IDAT Service Descriptor Configuration](#)).
3. Create an empty XML file in the `<module_id>.idat` package.
Note: You can choose any name for your XML file, but it must have `.xml` extension.
4. Add the IDAT XML file to the DC ASSEMBLY public part by right clicking the file and selecting `Development Component`→`Publish as File`
5. Log on to SAP ME and open *Service Extension Maintenance*.
6. Choose *Search* and select the service and method used to create your configuration data object.
7. Choose *View Input/Output* → *Input* and copy the input XML displayed into the XML file you created in the `<module_id>.idat` package. Enter element values and save your entries.
Note: You can add multiple service invocation requests to the same XML file. These invocation requests can refer to more than one service.

Database Schema

Concepts

Customizing the Database

When you create a SAP ME extension, you may sometimes need to add a new database table to store your data. You may also need to alter your data when you upgrade to a new release (see [Data Migration](#)). You can use NetWeaver Configuration Wizard to customize the database during installation and upgrade by packaging database scripts in your SC.

Please note that using CTC this way to execute custom database scripts, is only meant for basic SQL statements. CTC wizard is not a replacement for DB vendor tools like SQL Plus or SQL Server Management Studio. CTC parses the SQL file line by line and executes each statement using JDBC driver. Therefore it may be more convenient to store the SQL file in NWDI using the package structure below but then execute the SQL outside of CTC using a third part DB vendor tool like SQL Plus or SQL Server Management Studio. This will allow the full flexibility of using SQL scripting to customize the database.

Configuration Wizard searches your SC for a specific directory structure containing database scripts (see *Script DC Directory Structure*). When database scripts are located, Configuration Wizard executes them in the order of WIP, ODS, then GODS. Configuration Wizard executes and commits all SQL commands and logs the status of PASSED, WARNING, or FAILED for each.

Note: You only create directories that are required for your scenario and target environment. For example, you may omit the `database\upgrade\sqlserver` directory if you do not have upgrade scripts to run on Microsoft SQL Server.

For upgrades, Configuration Wizard executes all scripts in one upgrade directory before moving to the next upgrade directory. For example, if you have scripts associated with SAP ME 5.2 to SAP ME 6.0 upgrade as well as scripts for SAP ME 6.0 to SAP ME 6.1 upgrade, Configuration Wizard executes them in the following order:

Example: Upgrade

Root Directory: `database\upgrade\<database vendor>\`

Order	Directory	User
1.	<code>upgrade52to60\wip</code>	WIP database user
2.	<code>upgrade52to60\ods</code>	ODS database user
3.	<code>upgrade52to60\gods</code>	GODS database user
4.	<code>upgrade60to61\wip</code>	WIP database user
5.	<code>upgrade60to61\ods</code>	ODS database user

Order	Directory	User
6.	upgrade60to61\gods	GODS database user

You can include multiple database scripts under each directory. When multiple scripts are present in a directory then they are executed in alphabetical order. For this reason, you can use a naming convention for your script files to stipulate order of execution alphabetically. For example:

- EXE_101_ScriptOne.sql
- EXE_102_ScriptTwo.sql
- EXE_103_ScriptThree.sql

You can package multiple DCs containing custom database scripts within a single SC. Configuration Wizard executes all scripts in one DC before executing scripts from the next DC in the SC. Execution order among multiple DCs in a SC is not guaranteed.

Configuration Wizard supports execution of database scripts from multiple custom SCs. At runtime, Configuration Wizard executes all scripts from one SC then prompts for the next SC.

Data Migration

When SAP ME is upgraded to a new version, it is sometimes necessary to alter database data so it is compatible with the target version. This process is known as data migration and is incorporated into the SAP ME upgrade process.

Data migration is accomplished using standard SQL scripts. For example, suppose your extension uses `CUSTOM_WIP_TABLE`. When you upgrade your extension, you need to modify a value in `CUSTOM_DATA` column. To do this, you provide `UPDATE` statements in your SQL script.

Example - MS SQL Server

```
UPDATE CUSTOM_WIP_TABLE
  SET  CUSTOM_DATA='NewValue'
  WHERE CUSTOM_DATA='OriginalValue'
go
```

Example - Oracle

```
UPDATE CUSTOM_WIP_TABLE
  SET  CUSTOM_DATA='NewValue'
  WHERE CUSTOM_DATA='OriginalValue'
);
```

For more information about how to package your SQL scripts, see [Customizing the Database](#).

Tasks

Creating Database Schema

When you create a SAP ME extension, you may sometimes need to add a new database table or alter your data when you upgrade to a new release. If you package your database scripts in a predefined directory structure within your SC, Configuration Wizard can execute them during installation (see [Customizing the Database](#)).

Prerequisites

You have created a DC for *Database Schema* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a new source folder in your DC with the following structure **com/<vendor_id>**.
2. Right-click on `this folder` folder and choose `New → Folder`. Repeat this step recursively to create the directory structure for your custom database scripts (see [Script DC Directory Structure](#)).
3. Add your database scripts under each appropriate folder in the structure.
4. Add the database file to the DC ASSEMBLY public part by right clicking the file and selecting `Development Component → Publish as File`

Reference

Script DC Directory Structure

Database Scripts directory structure for SQL Server (new installation):

com/vendorID/database/newinstall/sqlserver/sqlgods
com/vendorID/database/newinstall/sqlserver/sqlods
com/vendorID/database/newinstall/sqlserver/sqlwip

Database Scripts directory structure for SQL Server (upgrade):

com/vendorID/database/upgrade/sqlserver/upgrade52to60/gods
com/vendorID/database/upgrade/sqlserver/upgrade52to60/ods
com/vendorID/database/upgrade/sqlserver/upgrade52to60/wip
com/vendorID/database/upgrade/sqlserver/upgrade60to61/gods
com/vendorID/database/upgrade/sqlserver/upgrade60to61/ods
com/vendorID/database/upgrade/sqlserver/upgrade60to61/wip
com/vendorID/database/upgrade/sqlserver/upgrade61to150/gods
com/vendorID/database/upgrade/sqlserver/upgrade61to150/ods
com/vendorID/database/upgrade/sqlserver/upgrade61to150/wip
com/vendorID/database/upgrade/sqlserver/upgrade150to151/gods
com/vendorID/database/upgrade/sqlserver/upgrade150to151/ods
com/vendorID/database/upgrade/sqlserver/upgrade150to151/wip
com/vendorID/database/upgrade/sqlserver/upgrade151to1511/gods
com/vendorID/database/upgrade/sqlserver/upgrade151to1511/ods
com/vendorID/database/upgrade/sqlserver/upgrade151to1511/wip
com/vendorID/database/upgrade/sqlserver/upgrade1511to1513/gods
com/vendorID/database/upgrade/sqlserver/upgrade1511to1513/ods
com/vendorID/database/upgrade/sqlserver/upgrade1511to1513/wip

Migrate from 15.1.3 to 15.1.5:

com/vendorID/database/upgrade/sqlserver/upgrade1513to1514/gods
com/vendorID/database/upgrade/sqlserver/upgrade1513to1514/ods
com/vendorID/database/upgrade/sqlserver/upgrade1513to1514/wip

Migrate from 15.1.5 to 15.2:

com/vendorID/database/upgrade/sqlserver/upgrade1515to152/gods
com/vendorID/database/upgrade/sqlserver/upgrade1515to152/ods
com/vendorID/database/upgrade/sqlserver/upgrade1515to152/wip

Migrate from 15.2 to 15.2.3:

com/vendorID/database/upgrade/sqlserver/upgrade152to1521/wip
com/vendorID/database/upgrade/sqlserver/upgrade152to1521/ods
com/vendorID/database/upgrade/sqlserver/upgrade152to1521/odsdt

Database Scripts directory structure for Oracle (new installation):

com/vendorID/database/newinstall/oracle/oracle gods
com/vendorID/database/newinstall/oracle/oracleods
com/vendorID/database/newinstall/oracle/oraclewip

Database Scripts directory structure for Oracle (upgrade):

com/vendorID/database/upgrade/oracle/upgrade52to60/gods
com/vendorID/database/upgrade/oracle/upgrade52to60/ods
com/vendorID/database/upgrade/oracle/upgrade52to60/wip
com/vendorID/database/upgrade/oracle/upgrade60to61/gods
com/vendorID/database/upgrade/oracle/upgrade60to61/ods
com/vendorID/database/upgrade/oracle/upgrade60to61/wip
com/vendorID/database/upgrade/oracle/upgrade61to150/gods
com/vendorID/database/upgrade/oracle/upgrade61to150/ods
com/vendorID/database/upgrade/oracle/upgrade61to150/wip
com/vendorID/database/upgrade/oracle/upgrade150to151/gods
com/vendorID/database/upgrade/oracle/upgrade150to151/ods
com/vendorID/database/upgrade/oracle/upgrade150to151/wip
com/vendorID/database/upgrade/oracle/upgrade151to1511/gods
com/vendorID/database/upgrade/oracle/upgrade151to1511/ods
com/vendorID/database/upgrade/oracle/upgrade151to1511/wip
com/vendorID/database/upgrade/oracle/upgrade1511to1513/gods
com/vendorID/database/upgrade/oracle/upgrade1511to1513/ods
com/vendorID/database/upgrade/oracle/upgrade1511to1513/wip

Migrate from 15.1.3 to 15.1.5:

com/vendorID/database/upgrade/oracle/upgrade1513to1514/gods
com/vendorID/database/upgrade/oracle/upgrade1513to1514/ods
com/vendorID/database/upgrade/oracle/upgrade1513to1514/wip

Migrate from 15.1.5 to 15.2:

com/vendorID/database/upgrade/oracle/upgrade1515to152/gods
com/vendorID/database/upgrade/oracle/upgrade1515to152/ods
com/vendorID/database/upgrade/oracle/upgrade1515to152/wip

Migrate from 15.2 to 15.2.3:

com/vendorID/database/upgrade/oracle/upgrade152to1521/wip
com/vendorID/database/upgrade/oracle/upgrade152to1521/ods
com/vendorID/database/upgrade/oracle/upgrade152to1521/odsdt

Database Scripts directory structure for HANA:

com/vendorID/database/newinstall/hana/hanawip
com/vendorID/database/newinstall/hana/hanaarc
com/vendorID/database/newinstall/hana/hanaarcdt

Migrate from 15.1 to 15.1.1:

com/vendorID/database/upgrade/hana/upgrade151to1511/wip
com/vendorID/database/upgrade/hana/upgrade151to1511/ods

Migrate from 15.1.1 to 15.3:

com/vendorID/database/upgrade/hana/upgrade1511to1513/wip
com/vendorID/database/upgrade/hana/upgrade1511to1513/ods

Migrate from 15.1.3 to 15.1.5:

com/vendorID/database/upgrade/hana/upgrade1513to1514/wip
com/vendorID/database/upgrade/hana/upgrade1513to1514/ods

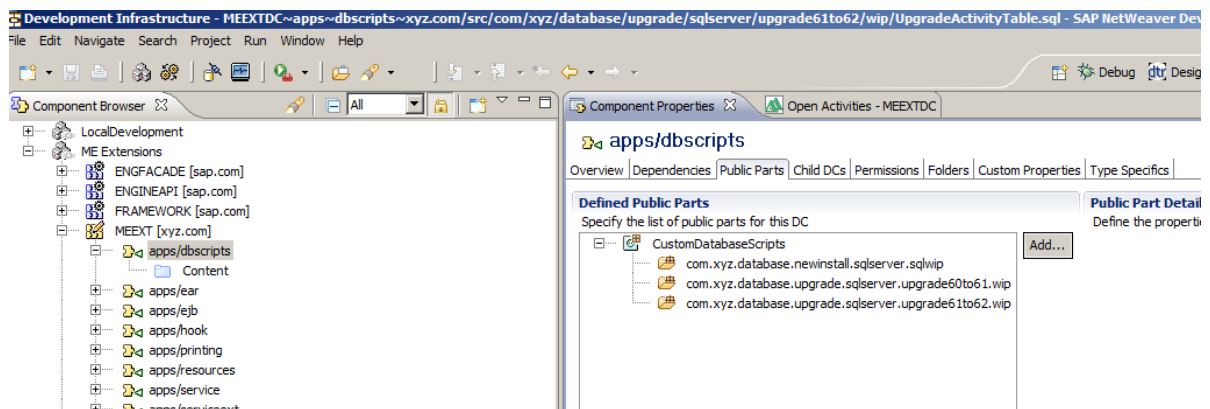
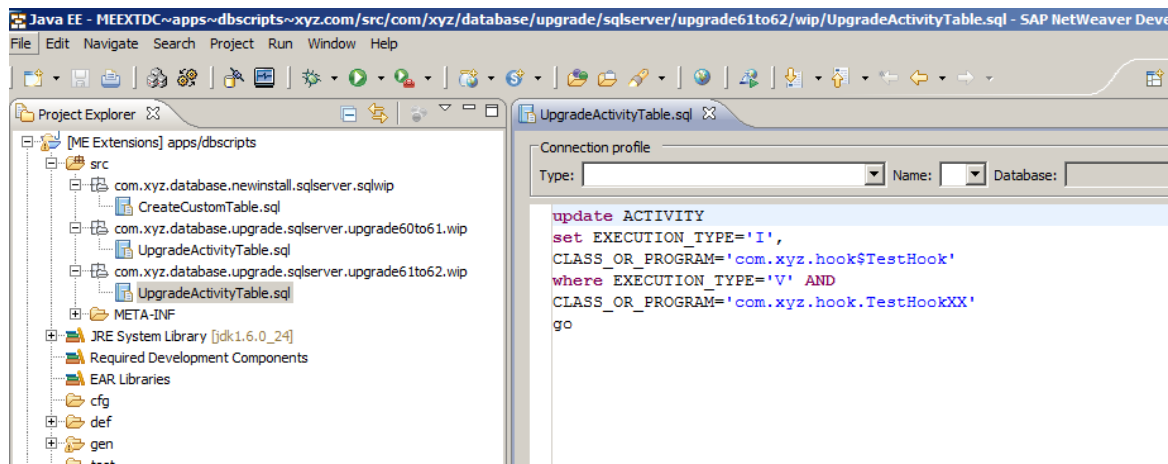
Migrate from 15.1.5 to 15.2:

com/vendorID/database/upgrade/hana/upgrade1515to152/wip
 com/vendorID/database/upgrade/hana/upgrade1515to152/ods
 com/vendorID/database/upgrade/hana/upgrade1515to152/odsdt

Migrate from 15.2 to 15.2.3:

com/vendorID/database/upgrade/hana/upgrade152to1521/wip
 com/vendorID/database/upgrade/hana/upgrade152to1521/ods
 com/vendorID/database/upgrade/hana/upgrade152to1521/odsdt

For example from ME 6.1 to ME 15.0 using the following structure in NWDS



Additional Rules Regarding Custom SQL Scripts

Custom SQL Scripts are executed inside SAP CTC execution engine and hence have a few coding conventions and rules unlike other SQL Scripts that are executed inside a database tool like SQLPlus or MS SQL Server Management Studio.

1. Vendor ID used in the package structure should adhere to the following format:

com/mycompany/database/newinstall/oracle/oraclewip.

2. If you have stored procedures in your scripts:
 - a. If Oracle procedures or synonyms are in the script file, the procedure needs to be separated by the delimiters <carriage_return> then / and then <carriage_return> and filename should contain either **Synonym_ or Functions.**
 - b. We cannot mix simple database script DML with stored procedures in case of Oracle.
3. Simple scripts need to be delimited by ; and <carriage_return> for Oracle and **go** and <carriage_return> for SQLServer.
4. The SQL scripts you are writing for MECTC should be able to rerun multiple times without an error.

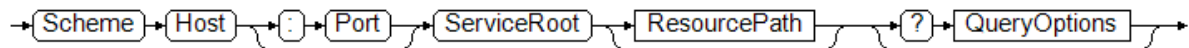
The following database error codes are treated as warnings that the CTC execution will not stop if any of these warning messages are thrown from the scripts. Any other database errors will be treated as CTC errors and execution of CTC scripts will stop.

HANA error codes	Oracle error codes	SQL Server error codes
288	955	2714
349	942	1913
259	2024	3701
289	1430	15248
261	904	4924
505	957	2705
328	1442	1750
329	1451	2627
321	1	
322	2261	
506		

272		
308		

OData

The Open Data Protocol enables creating REST based services where these services and their associated resources can be identified using Uniform Resource Identifiers (URIs). A URI used by an OData service has up to three significant parts: the service root URI, resource path and query string options as shown below.



The SAPME ODATA services expose a subset of the ME public APIs through these URIs, to be consumed by clients capable of transporting and receiving data over HTTP protocol. These services enable such clients to interact with ME over the web in the more intuitive and easier OData protocol compared to the traditional SOAP protocol.

This section describes the ways by which developers can customize the ME OData module by either extending core ME OData services or by creating new OData services that use ME PAPI methods.

Security and Site Management

All OData Services provided by SAPME are built on top of standard ME security and need authentication using a valid user name and password. The only difference is the use of Basic Authentication instead of Form based Authentication.

All OData services require operating within a specific site. Therefore, an OData request must pass a required **SITE** parameter whose value sets the desired site during the execution of the given service.

If the site value is missing, or an invalid site value, or a site value that is not valid for the current user is specified, the request fails with the appropriate error message.

Available Services

All SAPME OData services are contained within a web application with the reserved context name of **manufacturing-odata**. These services are then divided into categories within which one or more service(s) are exposed.

Similarly, the custom OData services are implemented using a custom Web DC module that can expose the OData services as part of custom webcontext. This context is defined in the EAR DC's application.xml deployment descriptor file.

The different categories are mapped to corresponding service roots within the URL. All the resource and data model definitions within a given service root are available using the reserved OData resource path **\$metadata**.

Java Docs

The Java Documentation for **com.sap.me.odata** module is available along with the standard SAP ME Core Java documentation on SAP Service Marketplace. OData documentation should be available from SAP ME 15.0 SP01 Patch 01 release. Note that `com.sap.me.odata.service.test` module is not to be used.

References

Document Title	URL
What is OData	OData Specification
Olingo OData Library 2.0.0 (Java) - For OData Version 2.0	Olingo OData Lib
ME OData Service Feature Guide	ME OData Features Guide

General Guidelines

All OData services whether new or modified must be built on top SAPME API layer and direct access of SAP ME data through SQL must be avoided. All new or modified services must be defined using a new OData Schema namespace and EntityContainer(s) as appropriate. The Schema namespace **"SAPME"** and container name **"ME"** are reserved and should not be used.

Basic Concepts and Steps

An OData Service implemented through SAP OData library must define a Service Factory, an EDM Provider and a Processor. The Service Factory is a class that creates the EDM Provider and the Processor to be used. This Service Factory itself is then referenced by a [servlet](#) servicing the OData service requests. This is how an OData request knows what EDM definitions and what processor need to be selected for servicing it. The Service Factory must always inherit from SAPME core service factory.

```
public class ProductionServiceFactory extends
com.sap.me.odata.service.MEODataServiceFactory {

    public ODataService createService(ODataContext ctx) throws
    ODataException {
        //always call super first to enable site Handling
        super.createService(ctx)
        EdmProvider edmProvider = new ProductionEdmProvider();
        ODataSingleProcessor singleProcessor = new
        ProductionSingleProcessor()
        return createODataSingleProcessorService(edmProvider,
        singleProcessor);
    }

}

public class ProductionSingleProcessor extends ODataSingleProcessor
{
    ...
}
```

```

    ...
}

```

New Services without Access to CORE EDMs

New Services can be defined by creating an EDM Provider, Processor, and a Service Factory. The processor simply calls SAPME API methods within the appropriate handlers that must be implemented per Olingo OData Library requirements. See [Olingo OData2.0 Documentation](#).

```

public class ProductionEdmProvider extends EdmProvider {

    protected final static String LOCAL_NAMESPACE = "Vendor";
    protected final static String LOCAL_CONTAINER_NAME = "Acme"
    ...
    ...
}

```

New Services with Access to CORE EDMs

In this case the EDM Provider, the Processor and the ServiceFactory must inherit from the core SAPME counterparts within the core ME OData service modules. Note that the core OData services are not be confused with core PAPI services. These core OData services cannot be extended using Service Extensions.

```

public class ProductionEdmProvider extends

    com.sap.me.odata.service.production.ProductionEdmProvider {

    protected final static String LOCAL_NAMESPACE = "Vendor";
    protected final static String LOCAL_CONTAINER_NAME = "Acme";
}

public class ProductionSingleProcessor extends

    com.sap.me.odata.service.production.ProductionSingleProcessor {
    ...
    ...
}

```

Let's look at an example of a requirement that needs to add some custom properties to an SFC while hiding other SFC properties provided by ME core EDM definitions. This new SFC can be defined within our new EDM provider within the Vendor schema namespace and ACME container name.

```

public class ProductionEdmProvider extends

    com.sap.me.odata.service.production.ProductionEdmProvider {

```

```

protected final static String LOCAL_NAMESPACE = "Vendor";
protected final static String LOCAL_CONTAINER_NAME = "Acme";

    /*
    * The local Sfc entity type and Sfc set definitions
    */
private static final FullQualifiedName SFC_ENTITY_TYPE = new
FullQualifiedName(LOCAL_NAMESPACE, "Sfc");
private static final String SFC_SET = "Sfcs";

@Override
public List<Schema> getSchemas() throws ODataException {
    // lets create a new schema for the vendor.
    List<Schema> schemas = super.getSchemas();
    Schema acmeSchema = new Schema();
    acmeSchema.setNamespace(LOCAL_NAMESPACE);

    //create a new container list for the new schema and add
    a new Vendor
    List<EntityContainer> entityContainers = new
ArrayList<EntityContainer>();
    EntityContainer entityContainer = new EntityContainer();
    entityContainer.setName(LOCAL_CONTAINER_NAME);
    entityContainers.add(entityContainer);
    acmeSchema.setEntityContainers(entityContainers);

    //Create and add the entity types
    List<EntityType> entityTypes = new
ArrayList<EntityType>();
    entityTypes.add(getEntityType(SFC_ENTITY_TYPE));
    acmeSchema.setEntityTypes(entityTypes);
    List<EntitySet> entitySets = new ArrayList<EntitySet>();
    entitySets.add(getEntitySet(LOCAL_CONTAINER_NAME,
SFC_SET));
    entityContainer.setEntitySets(entitySets);

    // add the new schema to the list available schemas.
    schemas.add(acmeSchema);
    return schemas;
}

@Override
public EntityContainerInfo getEntityContainerInfo(String name)
throws ODataException {
    if (LOCAL_CONTAINER_NAME.equals(name)) {
        return new
EntityContainerInfo().setName(LOCAL_CONTAINER_NAME);
    } else {

```

```

// return the ME Core
ContainerInfo
    return super.getEntityContainerInfo(name);
}

@Override
public EntitySet getEntitySet(String entityContainer, String
name)
    throws ODataException {
    if (LOCAL_CONTAINER_NAME.equals(entityContainer)) {
        if (SFC_SET.equals(name)) {
            return new EntitySet().setName(name)
                .setEntityType(this.SFC_ENTITY_TYPE);
        } else {
            return null;
        }
    }

    // return the entity set definition
    from CORE ME
    return super.getEntitySet(entityContainer, name);
}

/**
    This the EDM definition for the new SFC Entity that shows a subset
    of properties and has
    additional custom properties.
 */
@Override
public EntityType getEntityType(FullQualifiedName edmFQName)
    throws ODataException {
    if (LOCAL_NAMESPACE.equals(edmFQName.getNamespace())) {
        if
(SFC_ENTITY_TYPE.getName().equals(edmFQName.getName())) {
            EntityType sfcEntityType = new EntityType();
            List<Property> properties = new
ArrayList<Property>();

            Property prop = new
SimpleProperty().setName("sfc").setType(EdmSimpleTypeKind.String).setFacets(
new Facets().setNullable(false).setMaxLength(128));
            properties.add(prop);

            prop = new
SimpleProperty().setName("site").setType(EdmSimpleTypeKind.String).setFacets
(new Facets().setNullable(false).setMaxLength(6));
            properties.add(prop);

            prop = new
SimpleProperty().setName("active").setType(EdmSimpleTypeKind.Boolean).setFac
ets(new Facets().setNullable(true));
            properties.add(prop);

```

```

        prop = new
SimpleProperty().setName("inQueue").setType(EdmSimpleTypeKind.Boolean).setFa
cets(new Facets().setNullable(true));
        properties.add(prop);
        prop = new
SimpleProperty().setName("qty").setType(EdmSimpleTypeKind.Decimal).setFacets
(new Facets().setNullable(true));
        properties.add(prop);
        prop = new
SimpleProperty().setName("udf1").setType(EdmSimpleTypeKind.String).setFacets
(new Facets().setNullable(false));
        properties.add(prop);
        prop = new
SimpleProperty().setName("udf2").setType(EdmSimpleTypeKind.Double);
        properties.add(prop);
        sfcEntityType.setProperties(properties);
        //Keys
        List<PropertyRef> keyProperties = new
ArrayList<PropertyRef>();
        String[] ids = {"sfc", "site"};
        for (String idFieldName : ids) {
            keyProperties.add(new
PropertyRef().setName(idFieldName));
        }
        Key key = new Key().setKeys(keyProperties);
        sfcEntityType.setKey(key);
        sfcEntityType.setName(SFC_ENTITY_TYPE.getName());
        return sfcEntityType;
    } else {
        return null;
    }
} else {
    // return the SFC Entity
    definition provided by CORE ME
    return super.getEntityType(edmFQName);
}
}
}

```

We also need to create a Processor that will actually do the work and processes an incoming request and sends a response back to the caller.

```

public class ProductionSingleProcessor extends

com.sap.me.odata.service.production.ProductionSingleProcessor {

    private static final String SIMPLE_12213 =
"12213.simple";

    @Override

```

```

        public ODataResponse readEntitySet(GetEntitySetUriInfo uriInfo,
String contentType)
        throws ODataException {
            EdmEntitySet entitySet;
            if (
uriInfo.getEntityContainer().getName().equals(ProductionEdmProvider.L
OCAL_CONTAINER_NAME)) {
                if (uriInfo.getNavigationSegments().size() == 0)
{
                    entitySet = uriInfo.getStartEntitySet();
                    if ("Sfcs".equals(entitySet.getName())) {
                        String srchBy =
uriInfo.getCustomQueryOptions().get("Sfc
");
                        return
EntityProvider.writeFeed(contentType, entitySet,
                        getAcmeSfcs(srchBy),
EntityProviderWriteProperties.serviceRoot(
                        getContext().getPathInfo().getServiceRoot()).build());
                    } else {
                        throw new
ODataNotImplementedException();
                    }
                } else {
                    throw new ODataNotImplementedException();
                }
            } else {

                // return the SFcs that COE ME returns
                return super.readEntitySet(uriInfo, contentType);
            }
        }
    }

```

```

/**
 * Obtain the ACME vendor Sfcs that has additional attributes.
 * @param sfcMask
 * @return
 * @throws ODataException
 */
@Override
protected List<Map<String, Object>> getAcmeSfcs(String sfcMask)
    throws ODataException {
    // call the super method to obtain
all the SFC properties provided by CORE.
    List<Map<String, Object>> res = super.getSfcs(sfcMask);
    for (Map<String, Object> record : res) {
        if (record == null) continue;
    }
}

```

```

        Set<Entry<String, Object>> entries =
record.entrySet();
        Iterator<Entry<String, Object>> it =
entries.iterator();
        while ( it.hasNext() ) {
            Entry<String, Object> entry = it.next();
            if ( entry.getKey().equals("sfc")      ||
                entry.getKey().equals("site")     ||
                entry.getKey().equals("active")   ||
                entry.getKey().equals("inQueue")  ||
                entry.getKey().equals("qty")
            ) {
                continue;
            } else {
                it.remove();
            }
        }
        // populate values for your own
        additional properties.
        for (Map<String, Object> record : res) {
            if ( record == null) continue;
            record.put("udf1", "hello there");
            record.put("udf2", Double.valueOf(12.50));
        }
        return res;
    }
}

```

If this service is deployed, one can see the EDM definitions by invoking the OData \$metadata request. The output of such a request, as shown below, shows the new schema namespace, the container name, and the new SFC definition within them. As you can see, the SAPME schema definitions are still available and can be used as before.

```

<edmx:Edmx xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx"
Version="1.0">
    <edmx:DataServices
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
m:DataServiceVersion="1.0">
        <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
Namespace="SAPME">...</Schema>
        <Schema xmlns="http://schemas.microsoft.com/ado/2008/09/edm"
Namespace="Vendor">
            <EntityType Name="Sfc">
                <Key>
                    <PropertyRef Name="sfc"/>
                    <PropertyRef Name="site"/>
                </Key>
                <Property Name="sfc" Type="Edm.String" Nullable="false"
MaxLength="128"/>
            </EntityType>
        </Schema>
    </edmx:DataServices>
</edmx:Edmx>

```

New Services with Access to CORE EDMs

```
<Property Name="site" Type="Edm.String" Nullable="false"
MaxLength="6"/>
<Property Name="active" Type="Edm.Boolean" Nullable="true"/>
<<Property Name="udf2" Type="Edm.Double"/>
</EntityType>
<EntityContainer Name="Acme">
  <EntitySet Name="Sfcs" EntityType="Vendor.Sfc"/>
</EntityContainer>
</Schema>
</edmx:DataServices>
```

In order to invoke the new SFC service, the entity set name must be fully qualified with the container name it is defined within as shown.

http://host:port/acme-odata/Production.svc/Acme.Sfcs?Sfc=2&SITE=*

The response from this request will be a list of SFCs showing the limited number of properties and the additional properties as shown.

```
<entry>
  <id>
    http://localhost:50000/acme-
    odata/Production.svc/Acme.Sfcs(sfc='20',site='%2A')
  </id>
  <title type="text">Sfcs</title>
  <updated>2014-07-23T18:37:53.821-04:00</updated>
  <category term="Vendor.Sfc"
  scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme"/>
  <link href="Acme.Sfcs(sfc='20',site='%2A') " rel="edit" title="Sfc"/>
  <content type="application/xml">
    <m:properties>
      <d:sfc>20</d:sfc>
      <d:site>*</d:site>
      <d:active m:null="true"/>
      <d:inQueue m:null="true"/>
      <d:qty>1</d:qty>
      <d:udf1>hello there</d:udf1>
      <d:udf2>12.5</d:udf2>
    </m:properties>
  </content>
</entry>
<entry>
  .....

```

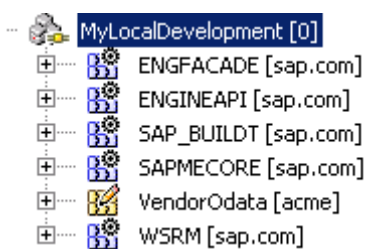
If the container name is not specified the default container name (ME) will be used and the resulting SFC entries will be what is defined and provided by core.

Creating and Deploying Extensions

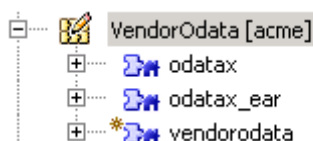
The SDK relies on NWDI processes for creating SAPME extensions in general and OData extension falls in the same category. For an OData Extension, a development configuration with access to the following software components is needed.

Note that while creating the Web DC module (odatax) you should not include the me-buildplugin as a dependency. The inclusion of the buildplugin will trigger the post-processing of the deployment descriptor web.xml file and might include other resources not needed for the OData services. Therefore, the recommendation is to keep the odata web context and the regular custom web context separate.

Software Component Configuration



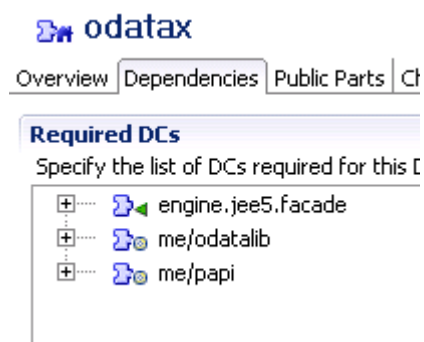
You also need to create a software component to house your J2EE Web and EAR DCs.



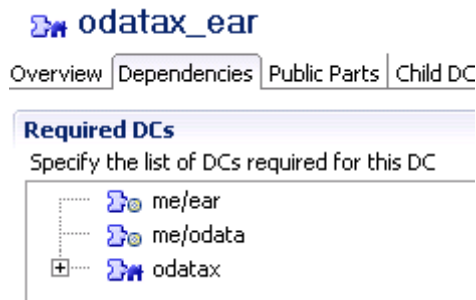
In this case, they are named odatax and odatax_ear respectively.

DC Configuration

The web DC (odatax) requires the Dcs listed below as required dependencies.



The ear DC (odatax_ear) needs to have the DCs below as required dependencies.



Web Project Configuration

The web project generated by the web DC needs to have its web.xml to minimally include the following. The highlighted areas are user defined and are here as just an example.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID"
version="2.5">
  <display-name>MyLocalDevelopment~odatax~acme</display-name>
  <listener>
    <listener-
class>com.sap.me.odata.GlobalizationProviderListener</listener-class>
  </listener>
  <listener>
    <listener-class>com.sap.me.frame.AppStartupListener</listener-class>
  </listener>
  <filter>
    <filter-name>ODataEntranceFilter</filter-name>
    <filter-class>com.sap.me.odata.OdataEntranceFilter</filter-class>

    <init-param>
      <param-name>FILTER_LIST</param-name>
      <param-value> <!-- VALUES MUST BE COMMA DELIMITED -->
        com.sap.me.odata.DefaultCredentialsSetterFilter,
        com.sap.me.common.client.impl.LocaleGetterPropagationFilter,
        com.sap.me.odata.OdataServiceFactoryFilter
      </param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>ODataEntranceFilter</filter-name>
    <url-pattern>/Production.svc/*</url-pattern>
  </filter-mapping>
  <servlet>
    <servlet-name>AcmeProductionServlet</servlet-name>
```

```
<servlet-  
class>org.apache.cxf.jaxrs.servlet.CXFNonSpringJaxrsServlet</servlet-class>  
<init-param>  
  <param-name>javax.ws.rs.Application</param-name>  
  <param-  
value>org.apache.olingo.odata2.core.rest.app.ODataApplication</param-value>  
</init-param>  
<init-param>  
  <param-name>org.apache.olingo.odata2.service.factory</param-  
name>  
  <param-  
value>com.acme.odata.service.production.ProductionServiceFactory</param-  
value>  
</init-param>  
</servlet>  
  
<servlet-mapping>  
  <servlet-name>AcmeProductionServlet</servlet-name>  
  <url-pattern>/Production.svc/*</url-pattern>  
</servlet-mapping>  
  
<security-constraint>  
  <web-resource-collection>  
    <web-resource-name>AcmeODataExtension</web-resource-name>  
    <description>Vendor OData Extension</description>  
    <url-pattern>/Production.svc/*</url-pattern>  
  
  </web-resource-collection>  
  <auth-constraint>  
    <description>Need this Access</description>  
    <role-name>SAP_ME_USER</role-name>  
    <role-name>SAP_ME_READONLY</role-name>  
    <role-name>SAP_ME_ADMINISTRATOR</role-name>  
  </auth-constraint>  
  <user-data-constraint>  
    <description>no description</description>  
    <transport-guarantee>NONE</transport-guarantee>  
  </user-data-constraint>  
</security-constraint>  
  
<security-role>  
  <role-name>SAP_ME_USER</role-name>  
</security-role>  
<security-role>  
  <role-name>SAP_ME_READONLY</role-name>  
</security-role>  
<security-role>
```

Web Project Configuration

```

        <role-name>SAP_ME_ADMINISTRATOR</role-name>
    </security-role>
    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>/manufacturing</realm-name>
    </login-config>
</web-app>

```

The SAP NetWeaver specific web deployment descriptor (web-j2ee-engine.xml) should always minimally contain the following:

```

<web-j2ee-engine xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="web-j2ee-engine.xsd">
    <spec-version>2.4</spec-version>
    <security-role-map>
        <role-name>SAP_ME_USER</role-name>
        <server-role-name>SAP_ME_USER</server-role-name>
    </security-role-map>
    <security-role-map>
        <role-name>SAP_ME_READONLY</role-name>
        <server-role-name>SAP_ME_READONLY</server-role-name>
    </security-role-map>

    <login-module-configuration>
        <login-module-stack>
            <login-module>
                <login-module-
name>EvaluateTicketLoginModule</login-module-name>
                <flag>SUFFICIENT</flag>
            </login-module>
            <login-module>
                <login-module-
name>BasicPasswordLoginModule</login-module-name>
                <flag>REQUISITE</flag>
            </login-module>
            <login-module>
                <login-module-
name>CreateTicketLoginModule</login-module-name>
                <flag>OPTIONAL</flag>
            </login-module>
        </login-module-stack>
        <security-policy-domain>/manufacturing</security-policy-domain>
    </login-module-configuration>
</web-j2ee-engine>

```

You also have to edit the application-j2ee-engine.xml in the EAR DC to include the following references. Note that if you are including both regular custom web customization DC and also custom odata service DC in your EAR DC, you should not include the **me/ear** reference-target as this is included automatically by the me-buildplugin.

```

<?xml version="1.0" encoding="UTF-8"?>
<application-j2ee-engine xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="application-j2ee-engine.xsd">

```

```
<reference reference-type="hard">
<reference-target provider-name="sap.com" target-
type="application">me/ear</reference-target>
</reference>
<reference reference-type="hard">
<reference-target provider-name="sap.com" target-
type="application">me/odata</reference-target>
</reference>
<provider-name>acme</provider-name>
</application-j2ee-engine>
```

General Development Aspects

Building MII Extensions

Disclaimer

We recommend that you use SAP MII extensions and transactions to support small, lightweight customizations **when deployed with SAP ME**.

Please do not use SAP MII transactions to orchestrate massive business logic using multiple SAP ME API calls. Such a design might work in low volume development or test environments but may be unstable and cause severe performance problems in high volume production environments. The SAP ME SDK should be used to implement Java extensions to realize more complex scenarios **for optimal performance and scalability**.

Finally, use a test environment for simulation before deploying any customizations into production. This is to ensure the design meets the scalability and performance requirements. If you are in doubt of the complexity of your solution, please contact your **SAP ME consultants or certified partner for support**.

Localization

Concepts

Localization Services

SAP ME provides localization services to manage three types of properties, including:

- Text: Includes all localized text except user error messages
- Error Messages: User error messages defined in business service exceptions
- Resources: URLs to static web content such as images and HTML pages

You can add new properties of each type by creating custom resource bundles and configuring the appropriate localization service according to the property type (see [Creating a Resource Bundle](#)).

You can also override SAP ME core properties by creating a new property in your custom bundle with the core property key.

See [Property Key Names](#) for information about creating and overriding properties.

Localization API

In most cases localization is performed automatically by the SAP ME localization services. Localization is automatically performed for WPMF based user interfaces, including field labels, user messages and other text presented on the screen.

There are special cases where you need to programmatically localize text. For example, if you create a web service that calls a business service then you will need to localize error messages whenever the business service throws an exception.

See `com.sap.me.extension.Localization` in SAP ME API Javadoc for more information about programmatically localizing text.

You can use the core framework class `GlobalizationService` to create an instance of `DateTimeInterface`. Here is an example that creates a `DateTimeInterface` in GMT timezone.

```
DateGlobalizationServiceInterface dateServiceInvariant =  
(DateGlobalizationServiceInterface)  
GlobalizationService.getInvariantService(KnownGlobalizationServices.  
DATE);  
DateTimeInterface dateTimeInterfaceInvariant =  
dateServiceInvariant.createDateTime();
```

Use the following code to create a `DateTimeInterface` object in the timezone of ME SITE.

```
GlobalizationService.getUserService(KnownGlobalizationServices.DATE)
```

Property Key Names

When you create a new text or resource property, you must ensure the key name is unique so it does not conflict with another key. To avoid naming conflicts, you can use a naming convention for your property key that includes your vendor ID in the property key name. For example,

```
<vendor_ID>.button_start.label
```

Since each software vendor has a unique vendor ID, this will insure that there are no property name collisions between vendors.

You create error messages in a Java resource bundle using the error number and the following format:

```
<ErrorNumber>.simple=<ErrorMessage>.
```

For example,

```
20996.simple=My new error message.
```

To override a SAP ME core property, you must use the core property key name for your property. To assist with this, SAP ME provides a mechanism to display the property key name in SAP ME user interface. These on-screen keys are activated by using the NetWeaver Config Tool to set the Java system property as follows:

```
-Ddisplay.bundle.keys=true
```

Once this system property is enabled, SAP ME displays message keys for labels, buttons, texts and exceptions.

Image Property URLs

Localization

If you want to replace a SAP ME core image, you must construct your image URL relative to SAP ME web application context. For example, if your web application context is "vendorapplication" and you want to override the SAP ME company logo with your company logo, you would create the following property:

```
sapLogo.default.IMAGE=../../vendorapplication/com/acme/resources/images/new_logo.gif
```

Tasks

Creating a Resource Bundle

You must create a Java resource bundle and service configuration to make your localized properties available at run-time. You may create a bundle to support any language and variant and are not limited to those supported by SAP ME.

Prerequisites

You have created a DC for *Localization* extension type (see [Creating a Service DC](#)).

Procedure

1. Create Java resource bundle in your DC.
2. Create an entry in `service-config.xml` for the SAP ME localization service associated with your property type:

Property Type	Service Type	Service Configuration Entry
label, button, title	Text	<pre><service name="ExtensionTextResourceBundle" parent="Prototype" class="com.sap.me.frame.globalization.LocaleSpecificTextResourcesGlobalizationExtensionService"> <property name="resourceName" value="com.acme.resources.VendorBundle"/> </service></pre>
font, image, help topic	Resources	<pre><service name="ExtensionResourcesResourceBundle" parent="Prototype" class="com.sap.me.frame.globalization.LocaleSpecificResourceResourcesGlobalizationExtensionService"> <property name="resourceName" value="com.acme.resources.VendorResourcesBundle"/> </service></pre>

Property Type	Service Type	Service Configuration Entry
error message	Error Messages	<pre><service name="ExtensionErrorResourceBundle" parent="Prototype" class="com.sap.me.frame.globalization.LocaleSpecificErrorResourcesGlobalizationExtensionService"> <property name="resourceName" value="com.acme.resources VendorErrorBundle"/> </service></pre>

Reference

[Error Handling](#)

Error Handling

Concepts

This section provides a high level overview of base application exception handling so that you know what to expect when your service extension throws an exception.

Exceptions thrown from your extension are propagated to and handled by the base application. Base application exception handling consists of the following steps:

- The transaction is marked for rollback
- If the exception is a runtime exception, then the user is redirected to the global ME error page
- If the exception is a checked exception, then the application displays an error message

All SAP ME checked exceptions extend `com.sap.me.frame.domain.BusinessException`. This base class accepts an error number which is a unique ID that correlates an error message with the exception.

Caution: Error numbers must be globally unique across the SAP ME application and all customizations. You can use an error number in the 20000-29999 range as these are reserved for custom development and will not conflict with SAP ME core error numbers.

Note: You are responsible for localizing the error message using the [Localization API](#) when you create a new exception for a user interface that is not based on the WPMF framework.

Error Message Replaceable Parameters

You can include replaceable parameters in your error message definition.

For example, to display a replaceable parameter named “SHOP_ORDER” you can define an entry in your resource bundle as follows:

```
20202.simple=Resource %SHOP_ORDER% does not exist
```

To enable parameter replacement, you create the `getShopOrder()` getter method in `InvalidShopOrderException` to provide the value for the `SHOP_ORDER` message parameter:

```
public class InvalidShopOrderException extends BusinessException {
    private String shopOrder;
    public InvalidShopOrderException (String shopOrderRef) {
        super(id);
        this.shopOrder = shopOrder;
    }
    public String getShopOrder() {
        return shopOrder;
    }
}
```

```
}
```

Tasks

Creating an Exception

You can create a new exception to implement custom error handling.

Prerequisites

- You have created a Java DC
- You have created your resource bundle (see [Creating a Resource Bundle](#))

Procedure

1. [Add your message](#) to the error message resource bundle (see [Error Handling](#)).
Example: 20201.simple=Serial Number Mismatch
2. Create a Java class that extends `BusinessException` and implements at least one `BusinessException` constructor.

Example:

```
public class VendorExtendedException extends BusinessException
{
    public SerialNumberMismatchException(int errorNumber) {
        super(errorNumber);
    }
}
```

3. Add your exception class to the `ASSEMBLY` public part of your DC

Overriding an Error Message

You can extend an exception to override its error message.

Prerequisites

You have created a Java DC.

Procedure

1. Add your message to the error message resource bundle (see [Error Handling](#)).
Example: 20996.simple=Message type is disabled. Use %NEW_TYPE% instead.
2. Create a Java class that extends an existing SAP ME exception.

Example:

```
public class MessageTypeDisabledExtendedException extends
MessageTypeDisabledException {
    private String newType = "NEW_FREE_FORM_TYPE";

    public MessageServiceExtendedException() {
        super("");
    }
}
```

Error Handling

```
public String getNewType() {
    return newType;
}

public void setNewType(String newType) {
    this.newType = newType;
}
}
```

3. Create a new service extension and configure it to execute at the `EXCEPTION` execution point (see [Creating a Service Extension](#)).

Set your exception into the invocation context (this replaces the original exception):

Example:

```
public class VendorExtension implements ActivityInterface,
InvocationContextSetter {
    private InvocationContext invocationContext;

    @Override
    public void execute(Object argument) throws Exception {
        invocationContext.setException(new
        MessageTypeDisabledExtendedException());
    }

    @Override
    public void setInvocationContext(InvocationContext
        invocationContext) {
        this.invocationContext = invocationContext;
    }
}
```

4. Add the exception class to the DC ASSEMBLY public part by right clicking the Java file and selecting `Development Component`→`Publish as File`.

Security

References

[Integrating Security Functions in SAP NetWeaver AS Java](#)

[Secure Programming - Java](#)

SAP ME 15.2 Security Guide

Logging

SAP ME application runs on the SAP NetWeaver AS Java platform and takes advantage of logging and tracing capabilities provided by the platform.

To access logging APIs you must add the following Build Time dependency to your DC:

```
ENGFACADE[sap.com] → tc/bl/logging/api
```

For more information about the logging and tracing capabilities of SAP NetWeaver AS Java platform, see [Developing Logging-Enabled Applications](#).

Customization Checklist

The following is a list of some best practices that are to be followed while customizing SAP ME using NWDS.

Don'ts:

- Do not hide exceptions, runtime errors. Catch {} section should not hide errors.
- If you 'open' user transaction always 'close' it, think about situations that runtime error happened and transaction must be closed.
- Do not change core SAP ME tables. Do not add or delete columns, do not alter column types and length. If you do so, you might have to manage migration and upgrade process. Besides, depending on the change in some cases the default core CTC scripts might not work.
- Domain services and other internal classes like com.sap.me.frame.SystemBase that are not part of SAP ME Public API (PAPI) are considered unsupported API as they can change between SAP ME versions. Only Public API (PAPI) is supported and guaranteed to remain backwards compatible. Never use domain services to update/delete/create records at ME database; reads are fine.
- Do not use LSF internal components. Do not overload pages with LSF containers; the recommended maximum per page is low.
- Do not try to play with XA transactions; there are many problems with JDBC drivers provided by DB vendors.
- Do not rely on SAP ME internal cache solution. It is unsupported.
- Watch out for Custom queries that return a large number or rows. If possible place a row limit in custom queries.
 - Or use same query framework that ME CORE uses. That way row limit parameter shown in System settings is applicable to custom query too. Note that ME CORE query framework is unsupported.
- Do not ignore custom SCA deployment errors while deploying using NWDS. Look at the appropriate log file and try to understand the reason for the error. Fix the deployment error before proceeding to test your customization.

Dos:

- Use only SAP ME Supported API.
- Always try to do Proof-Of-Concept project first to see that all method outputs contain required parameters and all additional supplement transactions are executed (for instance there is a validation along with update method).
 - Build and test in small increments.
- Always add traces to your custom code, it will help to debug problems during Production.
- If you create custom tables, do not forget to add indexes.

- Test your custom solution for Performance and Scalability before Deployment; functional test alone is not enough.
- Review your JavaScript code. Script tend to sometimes cause slow GUI so a quick look at script written will be useful.
- Any customization for ODS related needs to be looked at closely due to complexity.
- Custom hooks
 - During performance testing try with and without the hooks.
 - Generic hook(for example, hook can be executed at all the hook points) needs to be monitored/questioned for functionality.
- During code review keep a spreadsheet of tables where you expect large set of data.
- If working with Production tables always think about Archiving strategy in the long run.
- New configuration objects needs to be checked to see whether Audit Logging is needed.
- Complex customization around SAP ME Labor and SAP ME Time-based Assembly functionality need to be carefully reviewed.
- PAPI services use standard JEE container-managed transactions with a transaction attribute of the "REQUIRES" type. If you implement a custom API service as defined in the *Creating a Service* section of this guide, you can place your custom ME WIP database access code along with SAP ME PAPI calls to ensure that they both participate in the same database transaction.
- Always keep the version of SAPMECORE that is deployed on local SAP NetWeaver in sync with the version of SAPMECORE in NWDI that is used by local NWDS. If they go out of sync you will get unpredictable results during deployment or execution of your customizations including POD Plugins.

Specific Examples

The following examples are provided as a starting point for customers with specific customization requests. Note that the complete source code is provided below and that due to minor changes in the API between version of SAPMECORE and WPMF libraries modifications to these examples might be needed .

Automatic POD Refresh

The following example illustrates automatic POD Worklist Refresh. Note that polling as mentioned in the example below is not the optimum way to do a refresh from a performance perspective. The SAPME core team is working on functionality that will enable automatic server-side updates to POD. This new functionality will likely be available in the next release of SAPMECORE.

Procedure

1. Create a package called "com.<vendorid>.pod.refresh" in your Web DC. Replace <vendorid> with your vendor ID.
2. Copy `ExampleConfiguratorFilter.java` and `RefreshPlugin.java` files provided in Example 5 folder into this package in Web DC. This class will be called by the WPMF framework with your first click on your custom POD to initialize the POD. It has a reference to the `customHeader.jsp` file that you will create in step 5.
3. Update your `faces-config.xml` in Web DC to include the following elements. Make sure to change <vendorid> to denote your vendor ID.
4. Create a folder called `com/<vendorid>/pod/refresh` in your Web DC WebContent folder. This folder will contain the jsp in step 5. Replace <vendorid> with your vendor ID.
5. Copy `customHeader.jsp` into the folder created in step 4. Note that Java is a case-sensitive language, so folder and file names should exactly match. The `customHeader.jsp` file contains the "*refreshTrigger*" `<sap:poll>` control that calls the `RefreshPlugin.processRefresh` method every 10 seconds. Inside this method we update the custom POD worklist
6. Deploy your EAR DC.
7. Create a custom Work Center POD called "workCenterPod" in *POD Maintenance*. The `ExampleConfiguratorFilter.java` checks for this POD name in the code. Create appropriate activity in *Activity Maintenance* for this POD.
8. Create a link for this custom POD in Activity Manager by following the instructions in this chapter.
9. Create 2 web sessions in two separate Internet Explorer windows. Session 1 could be core POD without the automatic refresh custom code above. Session 2 is the custom Work Center POD with 10 second automatic worklist refresh.
10. Retrieve SFCs in both PODs.
11. Complete an SFC in session 1.
12. After 10 seconds you will see that this SFC is taken out of the queue in session 2 automatically during a refresh.
13. Optionally, set breakpoints in your code to see the execution trace.

Sample Code

customHeader.jsp

```
<%@page pageEncoding="UTF-8"%>
<%@ page language="java"%>
<%@ taglib prefix="fh" uri="http://java.sun.com/jsf/html"%>
<%@ 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/sapme/ui/icons"%>

<f:subview id="headerView">

    <sap:pageHeader id="headerContainer" >

        <f:facet name="header">
            <h:outputText id="pageHeaderTitle"
value="#{frameworkManagerBean.podTitle}" />
        </f:facet>

        <f:facet name="headerRightAlignedArea">

            <sap:panelGrid cellBackgroundDesign="transparent" width="100%"
cellHalign="start" cellValign="middle" columns="4">

                <sap:panelGroup backgroundDesign="transparent" padding="0 20
0 0">

                    <h:commandButton id="home"
image="#{icon:podHeaderUrl('home')}}"

alt="#{gapiI18nTransformer['Home.button.TEXT']}"

title="#{gapiI18nTransformer['Home.button.TEXT']}"

actionListener="#{frameworkManagerBean.reloadAction}">
                        <sap:ajaxUpdate />
                    </h:commandButton>
                </sap:panelGroup>

                <sap:panelGroup backgroundDesign="transparent" padding="0 20
0 0" rendered="#{frameworkManagerBean.logoutRendered}">

                    <h:commandButton id="mainLogout"
rendered="#{frameworkManagerBean.logoutRendered}"

image="#{icon:podHeaderUrl('logout')}}"

alt="#{gapiI18nTransformer['podSfcSelect.logout.TOOLTIP']}"
```

```

title="#{gapiI18nTransformer['podSfcSelect.logout.TOOLTIP']}">
    <sap:ajaxUpdate onevent="return window.doLogout()" />
    </h:commandButton>
</sap:panelGroup>

    <sap:panelGroup backgroundDesign="transparent" padding="0 20
0 0">
        <h:commandButton id="mainHelp"
image="#{icon:podHeaderUrl('help')}"
alt="#{gapiI18nTransformer['toolTip.help.TEXT']}"
title="#{gapiI18nTransformer['toolTip.help.TEXT']}"
actionListener="#{frameworkManagerBean.mainHelpAction}">
            <sap:ajaxUpdate />
            </h:commandButton>
        </sap:panelGroup>

        <sap:panelGroup backgroundDesign="transparent" padding="0 20
0 0" rendered="#{rtedMessagePlugin.displayIcon}">
            <h:commandButton id="rtedLink"
rendered="#{rtedMessagePlugin.displayIcon}"
image="#{rtedMessagePlugin.rtemImageUrl}"
alt="#{rtedMessagePlugin.toolTip}"
title="#{rtedMessagePlugin.toolTip}"
actionListener="#{rtedMessagePlugin.openRtemAction}">
                <sap:ajaxUpdate />
                </h:commandButton>
            </sap:panelGroup>

        </sap:panelGrid>
    </f:facet>

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

    <sap:panelGrid cellBackgroundDesign="transparent" width="100%"
height="0px" cellHalign="start" cellValign="top" columns="1">

        <sap:panelGroup backgroundDesign="transparent" width="100%"
height="0px">

            <sap:poll id="timeTrigger"
rendered="#{rtedMessagePlugin.enableRTEDTimeTrigger}"

interval="#{rtedMessagePlugin.pollingPeriod}"

actionListener="#{rtedMessagePlugin.pollRTEMState}" >

```

```

        <sap:ajaxUpdate
render="#{sap:toClientId('headerContainer')}"
                                onevent="window.startPolling();" />
        </sap:poll>

        <sap:poll id="refreshTrigger"
rendered="#{refreshPlugin.enableTrigger}"

interval="#{refreshPlugin.pollingPeriod}"

actionListener="#{refreshPlugin.processRefresh}" >
        </sap:poll>
    </sap:panelGroup>

</sap:panelGrid>

</sap:pageHeader>
</f:subview>

```

ExampleConfiguratorFilter.java

```

package com.abc.pod.refresh;

import com.sap.me.frame.Utills;
import com.sap.me.wpmf.application.ConfiguratorFilterInterface;
import com.sap.me.wpmf.PodConfiguration;
import com.sap.me.wpmf.PodLayoutConfiguration;

/**
 * filter to modify POD Configuration prior to initialization
 */
public class ExampleConfiguratorFilter implements
ConfiguratorFilterInterface {

    /**
     * Method called to allow modification of PodConfiguration object
     * prior to initialization of POD.
     *
     * @param podConf PodConfiguration object
     */
    public void modifyPodConfiguration(PodConfiguration podConf) {
        String podName = podConf.getPodName();

        // optional check to limit to specific POD's
        if (!Utills.isBlank(podName) /*&&
podName.equalsIgnoreCase("workCenterPod")*/) {

```

Automatic POD Refresh

```

        PodLayoutConfiguration layoutConf =
podConf.getLayoutConfiguration();

        // overrides core templateHeader.jsp for this POD

layoutConf.setHeaderAreaUrl("/com/abc/jsp/pod/refresh/customHeader.jsp");

    }
}
}

```

RefreshPlugin.java

```

/* (c) Copyright 2008 SAP AG. All rights reserved. */
package com.abc.pod.refresh;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.faces.event.ActionEvent;

import com.sap.me.common.CustomValue;
import com.sap.me.common.ObjectReference;
import com.sap.me.customdata.CustomDataServiceInterface;
import com.sap.me.customdata.ReadCustomDataRequest;
import com.sap.me.customdata.SaveCustomDataRequest;
import com.sap.me.demand.SFCBOHandle;
import com.sap.me.frame.ServiceLocator;
import com.sap.me.frame.domain.BusinessException;
import com.sap.me.frame.service.CommonMethods;
import com.sap.me.productdefinition.OperationBOHandle;
import com.sap.me.productdefinition.OperationConfigurationServiceInterface;
import com.sap.me.production.podclient.BasePodPlugin;
import com.sap.me.production.podclient.PodSelectionModelInterface;
import com.sap.me.production.podclient.RefreshWorklistEvent;
import com.sap.me.production.podclient.RefreshWorklistListenerInterface;
import com.sap.me.production.podclient.SfcChangeEvent;
import com.sap.me.production.podclient.SfcChangeListenerInterface;
import com.sap.me.production.podclient.SfcSelection;
import com.sap.me.wpmf.MessageType;
import com.sap.me.wpmf.util.MessageHandler;

public class RefreshPlugin extends BasePodPlugin {

    private static final long serialVersionUID = 1L;

```

```

// Set default polling frequency to 10 sec.
private String pollingPeriod = "10";
private boolean enableTrigger = true;

public RefreshPlugin() {
}

/**
 * Method to enable or disable refreshing.
 * @return boolean true to do POD refresh, else false
 */
public boolean isEnabledTrigger() {
    return enableTrigger;
}

/**
 * Action method to perform refresh processing
 * @param event ActionEvent
 */
public void processRefresh(ActionEvent event) {
    RefreshWorklistEvent e = new RefreshWorklistEvent(this);
    fireEvent(e, RefreshWorklistListenerInterface.class,
"processRefreshWorklist");
}

/**
 * gets the polling frequency.
 * @return polling period in seconds
 */
public String getPollingPeriod() {
    return pollingPeriod;
}
}

```

faces-config.xml

```

<?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">

    <managed-bean>
        <managed-bean-name>podConfiguratorFilterBean</managed-bean-name>

```

```
<managed-bean-  
class>com.abc.pod.refresh.ExampleConfiguratorFilter</managed-bean-class>  
  <managed-bean-scope>session</managed-bean-scope>  
</managed-bean>  
  
<managed-bean>  
  <managed-bean-name>refreshPlugin</managed-bean-name>  
  <managed-bean-class>com.abc.pod.refresh.RefreshPlugin</managed-bean-  
class>  
  <managed-bean-scope>session</managed-bean-scope>  
</managed-bean>  
</faces-config>
```

Assemble Component Plug-In Service Extension

Background

Currently POD plug-ins do not provide a core mechanism for extending them per customer needs. The only way a client can extend the operation of specific plugins is to rewrite core plug-ins with their extensions as custom plugins via custom PODs through the SDK. This approach can be impractical, time consuming, and complicated, especially for those core plugins that have a complex set of rules and operational requirements. Therefore, we need a simpler approach. Note that this extension mechanism is available from SAP ME 15.0 SP03.

Operational Specification

The core Assemble Component plug-in is the first plugin to provide for a customer defined service to be invoked as part of its post processing within the same transactional context of the plugin operation. This service should not be confused with the extension mechanism provided at the API level. This service is only at the POD level and is invoked by the given POD plug-in after the API call and its extension(s) have been invoked and completed.

The core Assemble Component plugin checks for the existence of a service with a reserved fully qualified name and if it finds such a service, invokes it within the same transactional context of the API call(s) performed by the plug-in. The service implementation must also ensure that POD integrity is preserved and the PODs' internal structures are updated as needed.

Service Requirements and Naming

The service must adhere to a core contract by implementing `com.sap.me.production.client.AssemblePluginExtensionInterface`.

Since the service is customer defined, it must ensure that any POD internal structures, such as the POD Selection Model, are updated and any events to notify other plug-ins are also fired as needed. The service has a reserved name that must be defined as `com.vendor.assembly$AssemblyPluginExtensionAssemblyPluginExtension`.

Here `<vendor>` is any valid vendor name chosen by the customer.

Once the implementation and service definition are completed these artifacts can become part of the system's runtime using the standard SDK extended application process.

```
/**
 * An interface defining the contract for a post component assembly
extension object executed
 * from the Pods plugin layer.
 *
 * @author I821888
 *
 */
public interface AssemblePluginExtensionInterface {
```

Assemble Component Plug-In Service Extension

```

/**
 * Contract method, implemented by a customer, to do any post
processing needed after
 * the assembly process is done.
 * @param plugin The component Assembly plug-in.
 * @param assembleRequest The request submitted for component
assembly
 * @param notAssembled A list of those components failing to be
assembled and the reason.
 * This list should be checked to determine
what needs to be done, if anything.
 * @throws BusinessException Exception thrown in case of any
possible failure from an API call.
 */
public void execute(Plugin plugin, GroupAssembleComponentsRequest
assembleRequest, List<InvalidAssemblyIdentifier> notAssembled) throws
BusinessException;
}

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-config SYSTEM "http://www.visiprise.com/dtd/service-
config.dtd">
<service-config moduleId="com.vendor.assembly">

    <import moduleId="com.sap.me.common" service="Prototype" />

    <service name="AssemblyPluginExtension"

        interface="com.sap.me.production.client.AssemblePluginExtensionInterf
ace"

            class="<the implementation class>
            parent="Prototype">

        </service>
    </service-config>

```

SDK Process

Since this service runs as part of a POD plug-in, a web container is needed to house the service. If you do not already have a Web application DC, one needs to be created with all the required dependencies. The custom service implementation class and the service definition contained in a `service-config.xml` must be part of, and exposed by, this web container. Once this web application is deployed on a given server, the extended service becomes available to all running PODs that are using the Assemble Component plug-in. Undeploying the extended application will revert the Assemble Component behavior to normal.

Sample Code

The example here is an implementation based on a request by one of our customers. The requirement is to change the SFC name (reserialize) as components are added to a given SFC.

AssemblePluginExtensionImpl.java

```
package com.acme.production.client;

/**
 * This is a sample implementation of an assemble plugin extension that
 * assumes you are adding components to one SFC at a time.
 * Note that this object has to be bound as a JSF managed bean with the
 * reserved name of "assembleComponentsPluginExtension".
 * @author I821888
 *
 */
public class AssemblePluginExtensionImpl implements
    AssemblePluginExtensionInterface {

    private SfcStateServiceInterface sfcStateService;
    private SplitSerializeServiceInterface splitSerializeService;

    /**
     * Default constructor
     */
    public AssemblePluginExtensionImpl() {
        super();
    }

    @Override
    public void execute(Plugin plugin,
        GroupAssembleComponentsRequest assembleRequest,
        List<InvalidAssemblyIdentifier> notAssembled)
        throws BusinessException {

        String operRef = assembleRequest.getOperationRef();
        String resourceRef = assembleRequest.getResourceRef();
        List<AssemblyComponent> comps =
            assembleRequest.getComponentList();
        List<SfcIdentifier> sfcs = assembleRequest.getSfcList();
        String inputSfcRef = sfcs.iterator().next().getSfcRef();
        SfcConfiguration sfcConf =
            this.getSfcStateService().readSfc(inputSfcRef);
    }
}
```

Assemble Component Plug-In Service Extension

```

        // add your own filter criteria based on the component list,
        attached data entries, and the notAssembled list for serializing the sfc
        here.

        if ( sfcConf != null ) {
            SfcKeyData newSfcKeyData = serializeSfc(sfcConf, operRef,
resourceRef);

            // now we need to update the Pod selections to ensure
            everything else in the Pod will know about the change.
            List<SfcSelection> podSfcSelections = ((BasePodPlugin)
plugin).getPodSelectionModel().getSfcs();
            for ( SfcSelection podSfcSelection : podSfcSelections ) {
                if ( podSfcSelection == null) continue;
                SfcKeyData sfcKey = podSfcSelection.getSfc();
                if ( sfcKey != null &&
sfcConf.getSfc().equals(sfcKey.getSfc())) {
                    podSfcSelection.setSfc(newSfcKeyData);

                    podSfcSelection.setInputId(newSfcKeyData.getSfc());
                }
            }
            // you should always fire these two events to refresh the other
            plugins and the worklist(either oper or workcenter).
            plugin.fireEvent(new SfcChangeEvent(this),
SfcChangeListenerInterface.class, "processSfcChange");
            plugin.fireEvent(new RefreshWorklistEvent(this),
RefreshWorklistListenerInterface.class, "processRefreshWorklist");
        }

    }

    /**
     * Do the actual serialization of an SFC at the given oper and
resource
     * @param sfcConf
     * @param operRef
     * @param resourceRef
     * @return
     * @throws BusinessException
     */
    private SfcKeyData serializeSfc(SfcConfiguration sfcConf , String
operRef, String resourceRef) throws BusinessException {
        List<SfcLocation> locationList = sfcConf.getSfcLocationList();
        Iterator<SfcLocation> it = locationList.iterator();
        while(it.hasNext()) {
            SfcLocation loc = it.next();
            if(! loc.getLocationState().equals(LocationState.READY)) {
                it.remove();
            }
        }
        List<SerializeSfcDetail> newSfcList = new
ArrayList<SerializeSfcDetail>();
        // serialize SFC with new names

```

```

        SerializeSfcDetail newSFC = new SerializeSfcDetail();
        newSFC.setSfc(sfcConf.getSfc() + "-X");
        if ( locationList != null && !locationList.isEmpty()) {

            newSFC.setLocation(locationList.iterator().next().getLocation());
        }

        newSfcList.add(newSFC);
        List<SerializeSfcRequest> serializeSfcRequests = new
LinkedList<SerializeSfcRequest>();
        SerializeSfcRequest request = new SerializeSfcRequest();
        request.setSfcRef(sfcConf.getSfcRef());
        request.setResourceRef(resourceRef);
        request.setOperationRef(operRef);
        request.setAllowSplitOnDoneSfc(Boolean.TRUE);
        request.setNewSfcList(newSfcList);
        serializeSfcRequests.add(request);
        Collection<SerializeSfcResponse> responses =
getSplitSerializeService().serializeSfcs(serializeSfcRequests);
        SerializeSfcResponse response = responses.iterator().next();
        SfcKeyData newSfcKeyData =
getSfcStateService().findSfcKeyDataByRef(new
ObjectReference(response.getNewSfcRef()));
        return newSfcKeyData;
    }

    /**
     * Obtain a reference to the SfcStateService.
     * @return
     */
    protected SfcStateServiceInterface getSfcStateService() {
        if ( sfcStateService == null) {
            sfcStateService =
Services.getService("com.sap.me.production", "SfcStateService");
        }
        return sfcStateService;
    }

    /**
     * Obtain a reference to the SplitSerialize service.
     * @return
     */
    protected SplitSerializeServiceInterface getSplitSerializeService() {
        if ( splitSerializeService == null) {
            splitSerializeService =
Services.getService("com.sap.me.production", "SplitSerializeService");
        }
        return splitSerializeService;
    }
}

```

service-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE service-config SYSTEM "http://www.visiprise.com/dtd/service-
config.dtd">
<service-config moduleId="com.vendor.assembly">

    <import moduleId="com.sap.me.common" service="Prototype" />

    <service name="AssemblyPluginExtension"

        interface="com.sap.me.production.client.AssemblePluginExtensionInterf
ace"
            class="com.acme.production.client.AssemblePluginExtensionImpl"
            parent="Prototype">

                parent="Prototype">
            </service>
</service-config>
```

Extension Migration from SAP ME Legacy SDK

Note that NWDS-based extensibility was added to ME starting from version 6.1. These steps are necessary for migration from Legacy SDK to NWDS for ME 6.1, ME 15.0/15.1/15.2. If you are already using NWDS for custom code development for ME, no explicit migration steps are necessary to move to ME 15.2. However, we recommend that once you update SAP ME Core to a newer version, you have to test the existing code to make sure it works as expected. The version of the SAP ME Core that is used for compiling your extensions in NWDS must be the same as the version of SAP ME Core that is deployed in production.

Tasks

Migrating Global Configuration

This section describes migration of the Global Configuration project of the previous version of SAP ME SDK. The Global Configuration project contains application descriptors, IDAT files and property files.

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Migrating Application Descriptors

Optional: If you have references declared in the `application-j2ee-engine.xml` of your Global Configuration project, you must create the same dependencies in the `application-j2ee-engine.xml` of your *Java EE Enterprise Application* DC.

Note: In the previous version of SAP ME SDK the `application.xml` descriptor was populated automatically by the build system. You do not have to migrate it.

Migrating IDAT files

In SAP ME 15.0 the concept of IDAT files has been changed and now these files are packaged within `SAPMECORE.SCA`. There is no way to add a “custom” folder to SAP ME core with your own IDAT files. Instead, a new mechanism to load data has been created and is described in [Using Initial Data Load for Service Extension Configuration](#).

You must perform these steps for every IDAT file in the IDAT folder of the Global Configuration project of the previous version of SAP ME SDK:

1. Identify which business object is created by this IDAT file.

Note: In most cases the business object can be determined just by looking at the file name. Sometimes you will need to look at the SAP ME Database Guide document to match the file name with the object.

For example: `OperationBO.idat` is for creating Operation business objects, and `ItemBO.idat` is for creating Material business objects.

2. In the SAP ME API documentation, find the service in the public API responsible for creating the business object. The service will have a `create` method to create the business object.

Tasks

For example, the Operation business object can be created via the `OperationConfigurationService` service, and the Material business object can be created using the `ItemConfigurationService` service.

3. Follow the steps in [Using Initial Data Load for Service Extension Configuration](#) and create one XML request per record in your IDAT file. All requests may be placed in the same XML file.

Migrating Properties

1. Copy all property files in the *Properties* folder of the Global Configuration project of the previous version of SAP ME SDK to your DC.

Note: The folder where you copy the property files in the DC should be under source folder. It is strongly recommended to specify the package for it, for example, `"com.acme.resources"`.

2. For each property file copied, follow the corresponding section in [Creating a Service](#) to create service entry in the `service-config.xml`.

Migrating Supporting Classes and Resources

This section describes migration of the Supporting Classes & Resources project from the previous version of SAP ME SDK. The classes in this project are used in other SAP ME SDK projects, like Activity Hook, Service Plugin, EJB, etc.

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Procedure

1. Copy Java classes from the Supporting Classes & Resources project of the previous version of SAP ME SDK to your DC.
2. Include your Java classes in the `ASSEMBLY` public part of your DC.
3. Optional: If your Java classes reference the legacy SAP ME classes, add a `Build Time` dependency to `SAPMELEGACY[sap.com]/me/legacy` public part "api" to your DC.
4. Optional: If there are other DCs that need your Java classes for compilation,
 - a. Add your Java classes to the `COMPILATION` public part of your DC.
 - b. Set the `COMPILATION` public part as `Build Time` dependency for all DCs that need your Java classes.

For example, there may be business interfaces for services that you want to provide to a Java DC (where service implementation is located), an EJB Module DC (where EJB implementation is located) and a Web Module DC (where service and EJB are used).

Migrating Business Process Extensions

This section describes steps you need to perform to migrate activity hooks, printing services, service plug-ins and collaboration plugins from SDK to NWDI.

Prerequisites

You have created a DC for *Service* extension type (see [Creating a Service DC](#)).

Migrating Activity Hook

1. From the Activity Hook project in the previous version of SAP ME SDK, copy your activity hook Java class to your DC in the new SAP ME SDK.
2. Add the activity hook Java class copied above to the `ASSEMBLY` public part of your DC.

Tasks

3. Optional: If your activity hook Java class references any legacy SAP ME classes, you must add a built time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part “api” to your DC.
4. For the activity hook Java class copied above create a service definition in the `service-config.xml` the by extending the `Prototype` service as the parent (see [Service Descriptor](#)).

For example:

```
<service name="LogActivityHook" parent="Prototype"
class="com.acme.production.hook.LogActivityHook"/>
```

Note: You may need to add a statement to import the `Prototype` service into your service configuration descriptor if it has not been added already:

```
<import moduleId="com.sap.me.common" service="Prototype"/>
```

5. Update the SAP ME system so it recognizes this new activity as a service instead of a Java class. This may be done by either changing the activity in *Activity Maintenance* or by utilizing database script.
 - a. In *Activity Maintenance* retrieve the activity where the activity hook class is declared.
Change the *Type* field value from *Java Class* to *Service*.
Change the *Class/Program* field from the fully qualified Java class name to the fully qualified service name. For example instead of `com.acme.production.hook.LogActivityHook` it should be `com.acme.production.hook$LogActivityHook`
 - b. Create a database script to update the following fields in the `ACTIVITY` table:
 - `EXECUTION_TYPE` - needs to be changed from `JAVA ("V")` to `SERVICE("I")`
 - `CLASS_OR_PROGRAM` - needs to be changed from the fully qualified Java class name to the fully qualified service name.

For more information, see [Creating Database Schema](#).

Here is an example of a database script to update the SAP ME `ACTIVITY` table for the activity hook described above:

```
update ACTIVITY
set EXECUTION_TYPE='I',
    CLASS_OR_PROGRAM='com.acme.production.hook$LogActivityHo
ok'
where EXECUTION_TYPE='V' AND
    CLASS_OR_PROGRAM='com.acme.production.hook.LogActivityHo
ok'
```

Note: Localization and IDAT migration are covered in [Migrating Global Configuration](#).

Migrating Print Plug-In

1. Copy the Java class for your printing service from the Print Plugin project in the previous version of SAP ME SDK to your DC in the new SAP ME SDK.
2. Add the Java class copied above to the `ASSEMBLY` public part of your DC.

3. Add a build time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part "api" to your DC.
4. For the Java class copied above create a service definition in the `service-config.xml` file by extending `Prototype` as a parent (see [Service Descriptor](#)).

For example:

```
<service name="CustomPrintingPluginDA" parent="Prototype"
class="com.acme.document.CustomPrintingPluginDA"/>
```

Note: You may need to add the following statement to import the `Prototype` service into your service configuration descriptor if it has not been added already:

```
<import moduleId="com.sap.me.common" service="Prototype"/>
```

5. Update the SAP ME system to reflect the above changes. This may be done by either changing the printing integration in *Document Maintenance* or by utilizing a database script.
 - a. In *Document Maintenance*, retrieve the document where the printing service is used. Switch to the *Print Integration* tab. Replace the print plugin type from the fully qualified Java class name to the fully qualified service name.
For example instead of
`com.acme.document.CustomPrintingPluginDA`
it should be
`com.acme.document$CustomPrintingPluginDA`
 - b. Create a database script to update one of the following fields in the SAP ME `DOCUMENT` table depending on the type of print plug-in you are migrating:
 - `DATA_ACQUISITION_CLASS` - for print plug-ins that implement the `PrintingPluginDataAcquisitioner` interface or one of its super classes.
 - `FORMATTER_CLASS` - for print plug-ins that implement the `PrintingPluginFormatter` interface or one of its super classes.
 - `TRANSPORT_CLASS` - for print plug-ins that implement the `PrintingPluginTransporter` interface or one of its super classes.

For more information see [Creating Database Schema](#).

Here is an example of a database script to update `DOCUMENT` table for the data acquisition print plugin mentioned above:

```
update DOCUMENT
set
DATA_ACQUISITION_CLASS='com.acme.document$CustomPrintingPlugin
DA'
where
DATA_ACQUISITION_CLASS='com.acme.document.CustomPrintingPlugin
DA'
```

Migrating Collaboration Plug-In

1. Copy the Java class for your collaboration plugin from the Collaboration Plugin project of the previous version of SAP ME SDK to your DC in the new SAP ME SDK.
2. Add the Java class copied above to the `ASSEMBLY` public part of your DC.
3. Add a build time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part "api" to your DC.

Tasks

4. For the Java class copied above create a service definition in the `service-config.xml` file by extending the `Prototype` service as a parent (see [Service Descriptor](#)).
For example:

```
<service name="SFCCustomValueDA" parent="Prototype"
class="com.acme.production.collaboration.SFCCustomValueDA"/>
```

Note: You may need to add a statement to import the `Prototype` service into your service configuration descriptor if it has not been added already:

```
<import moduleId="com.sap.me.common" service="Prototype"/>
```
5. Update the SAP ME system to reflect the above changes. This may be done by either updating the activity in *Activity Maintenance* or by utilizing a database script.
 - c. In *Activity Maintenance* retrieve the activity for the collaboration plug-in. Change the *Class/Program* field from the fully qualified Java class name to the fully qualified service name.
For example instead of

```
com.acme.production.hook.LogActivityHook
```

it should be

```
com.acme.production.hook$LogActivityHook
```
 - d. Create a database script to update the `CLASS_OR_PROGRAM` field in the SAP ME `ACTIVITY` table. The value needs to be changed from the fully qualified Java class name to the fully qualified service name.

For more information, see [Creating Database Schema](#).

Here is an example of a database script to update existing SAP ME `ACTIVITY` table for the collaboration plugin mentioned above:

```
update ACTIVITY
set
CLASS_OR_PROGRAM='com.acme.production.collaboration$SFCCustomV
alueDA'
where
CLASS_OR_PROGRAM='com.acme.production.collaboration.SFCCustomV
alueDA'
```

Migrating Service Plug-In

1. Copy your service plugin Java class from the Service Plugin project of the previous version of the SAP ME SDK to your DC in the new SAP ME SDK.
2. Add the Java class copied above to the `ASSEMBLY` public part of your DC.
3. Optional: If your Java class references any legacy SAP ME classes, then you must add a build time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part "api" to your DC.
4. For the Java class, copy any service definition(s) from the old SDK's `service-config.xml` to the `service-config.xml` in your DC. Also copy all necessary import and referenced `service` entries.

Migrating POD Plug-Ins

This section describes how to migrate POD plug-ins into the new SAP ME SDK.

Prerequisites

You have read WPMF Plug-In Writer's Guide and have a basic working knowledge of JSF.
You have created a DC for *POD Plug-in* extension type (see [Creating a Service DC](#)).

Procedure

1. Create a J2EE dynamic web project using this web DC.
Add your custom pod plug-in(s) to the web project by copying the existing pod plug-in managed bean classes to the the *Web Application* project's `Java websrc` folder. If possible keep the packaging of each plug-in the same for minimum impact on the other definitions.
2. Copy the views (jsp files) of your existing plugins plug-ins as follows:
 - a. Open the *Web Application* project and right click on the `web` folder
 - b. If you have not yet created a vendor specific directory structure, choose *New* → *Folder* and create a directory structure according to existing folder structure of your plug-in views.
3. Add plug-in (managed bean) definitions to the `faces-config.xml` file as follows:
 - a. Edit `faces-config.xml` in `Web Application/web/WEB-INF` folder.
 - b. Copy the managed bean definition for the existing plugins from your existing `faces-config.xml` to this project's `faces-config.xml` file. Check the file for any name conflicts with any previous contents in the file. If you have changed any of the package names, reflect those changes in `faces-config.xml`.
4. Build the web development DC and the ear DC containing the web DC and deploy the ear.
5. Perform steps 7 and 8 in [Creating New POD Plug-Ins](#) if applicable.

Migrating EJBs

This section describes migration of standard Java EE EJBs from SAP ME SDK to NWDI.

Prerequisites

You have migrated classes from the Supporting Classes & Resources project (see [Migrating Supporting Classes and Resources](#)).

Procedure

1. Create a new *Java EE EJB Module* DC.
2. Add a build time dependency to the `COMPILATION` public part from the DC where you migrated supported classes and resources to your *EJB Module* DC.
3. Copy the Java classes from the EJB project of the previous version of SAP ME SDK to your *EJB Module* DC.
4. Optional: If any of your Java classes reference the legacy SAP ME classes, add a build time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part “api” to your DC.
5. Open `ejb-j2ee-engine.xml` in the EJB project of the previous version of SAP ME SDK and copy the contents within the `ejb-j2ee-engine` tag into `ejb-j2ee-engine.xml` of your *EJB Module* DC.
6. Create a new `ejb-jar.xml` in your *EJB Module* DC.
7. Open the `ejb-jar.xml` from the EJB project of the previous version of SAP ME SDK and copy the contents within the `ejb-jar` tag into `ejb-j2ee-engine.xml` of your *EJB Module* DC. If the descriptor does not validate you will need to resolve all errors to comply with the EJB 3.0 descriptor XSD schema.

Migrating Web Services

This section describes the migration of Web Services from the previous release of the SAP ME SDK to NWDI.

Prerequisites

You have migrated classes from the *Supporting Classes & Resources* project (see [Migrating Supporting Classes and Resources](#)).

Procedure

1. Copy the web service Java sources to the `source` folder of *Java EE Web Module* DC:
 - Java sources from the `src` folder of the *Web Services* SDK Project
 - Generated Java sources from the `generated-src` folder of the *Web Services* SDK Project.
2. Optional: If any of your Java classes reference the legacy SAP ME classes, add a build time dependency to `SAPMELEGACY[sap.com]/me/legacy` public part “api” to your DC.
3. Copy all the WSDL files from the `web/WEB-INF` folder of the *Web Services* SDK Project. Put the WSDL files into the `WebContent/WEB-INF` folder of *Java EE Web Module* DC. If the WSDL files are located within subfolders, keep the subfolder structure when copying the files.

Note: You no longer use web descriptors to secure your web services. For information about securing web services, see [Configuring WSDL Security](#).

Migrating Web Service Clients

This section describes migration of web service clients from SAP ME SDK to NWDI. The best way to migrate the web service client is to recreate the web service proxy by following the NWDS documentation.

Procedure

1. Follow the instructions in the SAP NWDS online help to create web service proxies (see [Consuming Web Services](#)).
2. Copy the Java sources that use the web service proxy from the previous version of SAP ME SDK to the DC where you are planning to use them.

Migrating the Web Extension Application

This section describes the migration of images and work instructions from the Web Extension Application of SAP ME SDK to NWDI.

Prerequisites

You have migrated classes from the *Global Configuration* project (see [Migrating Global Configuration](#)).

Procedure

Copy the `images` and `workinstructions` folders from the `web` folder of the *Web-Ext Application* project to the *Java EE Web Module* DC.

Migrating the Custom Servlets

If you are migrating custom Servlets from SAP ME SDK to NWDS, you will need to add the following mapping to your `web.xml` so that you can access `SITE` and `USER_ID` values using `com.sap.me.frame.service.CommonMethods` class.

```
<filter-mapping>
    <filter-name>MeEntranceFilter</filter-name>
    <url-pattern>/CustomServletController1</url-pattern>
    <url-pattern>/CustomServletController2</url-pattern>
</filter-mapping>
```

Where `MeEntranceFilter` refers to pre-configured ME CORE `com.sap.me.frame.web.MeEntranceFilter`; `CustomServletController1` and `CustomServletController2` are custom Servlets built using SDK.

Migrating SAP ME Login Module

If you are migrating from SAP ME 5.2 or SAP ME 6.0 and used the SAP ME login module to secure your web services then this section will help you to discontinue its use. If you are unsure as to whether your web services use the login module, check the `web-j2ee-engine.xml` deployment descriptor in the *Web Services* SDK project for the following element:

```
<login-module-name>MfgLoginModule</login-module-name>
```

Removing the Login Module and Migrating Web Service Implementations

In each web service method that uses a business service API, add a parameter for the `site` value and use the corresponding `Service.getService()` method with the `site` parameter.

Perform the following steps to remove the login module dependency from your web service:

1. Add a field for `site` to your request object, including a setter and getter method.
There is no need to add a field for the user since it will be automatically propagated by SAP NetWeaver as a JEE Principal object.

Tasks

2. Change all usages of

```
Services.getService(String module, String service)
RunAsServiceLocator.getService(ServiceReference svcRef, String
site)
```

to

```
Services.getService(String module, String service, String
site)
```

3. Optional: Add SAP NetWeaver security annotations for setting authentication level and transport security. For example, HTTP and HTTPS basic authentication is supported by the following annotations:

HTTP

```
@AuthenticationDT(authenticationLevel =
AuthenticationEnumsAuthenticationLevel.BASIC)
@AuthenticationRT(AuthenticationMethod = "sapsp:HTTPBasic")
```

HTTPS

```
@AuthenticationDT(authenticationLevel =
AuthenticationEnumsAuthenticationLevel.BASIC)

@AuthenticationRT(AuthenticationMethod = "sapsp:HTTPBasic")
@TransportGuaranteeRT(TLSType="sapsp:HTTPS")
```

For more information about web service authentication, see [Setting Authentication Level](#).

Reference

DC Configuration by Extension Type

Extension Type	<i>Business Service</i>
DC Type	Java
Public Part Configuration	There are two public part configurations required. Purpose: ASSEMBLY Suggested name: <code>services</code> Description: Includes service interfaces, implementation classes, and the service descriptor. Purpose: COMPILATION Suggested name: <code>service-api</code> Description: Includes only service interfaces for compiling dependent DCs.
Build Time Dependencies	SC: SAPMECORE[sap.com] • DC: <code>/me/papi</code> ◦ Public Part: <code>api</code>

Extension Types	<i>Service Extension</i>
DC Type	Java
Public Part Configuration	Purpose: ASSEMBLY Suggested name: <code>service-extension</code> Description: Includes service extension implementation classes and the service descriptor.
Build Time Dependencies	SC: SAPMECORE[sap.com] • DC: <code>/me/papi</code> ◦ Public Part: <code>api</code>

Extension Types	<i>Activity Hook</i>
DC Type	Java

Public Part Configuration	Purpose: ASSEMBLY Suggested name: activity-hook Description: Includes activity hook implementation classes and the service descriptor.
Build Time Dependencies	SC: SAPMECORE[sap.com] • DC: /me/papi ◦ Public Part: api

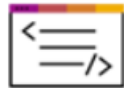
Extension Type	<i>POD Plug-in</i>
DC Type	Web Module
Public Part Configuration	Purpose: ASSEMBLY
Build Time Dependencies	SC: SAPMECORE[sap.com] • DC: /me/papi ◦ Public Part: api • DC: /me/lsf/libs ◦ Public Part: api • DC: /me/wpmf/libs ◦ Public Part: api • DC: /me/buildplugin-resources ◦ Public Part: def ◦ Public Part: RESOURCES

Extension Type	<i>Database Schema</i>
DC Type	Java
Public Part Configuration	Purpose: COMPILATION Required public part name for Database Script Files: CustomDatabaseScripts
Build Time Dependencies	None

API Reference

SAP ME 15.2 Javadoc is available at URL, at Development section:

https://help.sap.com/viewer/product/SAP_MANUFACTURING_EXECUTION/15.2/en-US



Development

[Javadocs for SAP Manufacturing Execution](#)

[Database Guide for SAP Manufacturing Execution](#)

PDF

[Database Quick Reference Guide for SAP Manufacturing Execution](#)

[Object Models Guide for SAP Manufacturing Execution](#)

[View All](#)

Included within the Javadoc archive is the API summary report (*APISummary.html*). The purpose of the API summary report is to provide a quick reference listing of all SAP ME modules, business service interfaces, and interface methods, allowing you to quickly survey what business services are available and ascertain their connectivity status.