MetaSolv Solution ™ 6.0.12+

Custom Extensions Developer's Reference

Sixth Edition
August 2008

METASOLV®
SOFTWARE

Copyright and Trademark Information

Document History

Edition	Date	Reason
First	December 2006	Initial release of custom extension functionality.
Second	January 2007	Re-wrote Chapter 4: Coding the extension logic. Completed Appendix B: Extensions sample code.
Third	March 2007	Added information regarding the three execution points that were new with 6.0.13. Changes were made to Chapter 3: Identifying an execution point, Appendix A: Supported execution points, and Appendix B: Extensions sample code
Fourth	June 2007	Updated Copyrights, About this Guide chapter, and customer portal references with Oracle.
Fifth	December 2007	Added information on new VCONDES Maintenance option. Added sample SelectComponentForVirtual.
Sixth	August 2008	Added information on new Email CLR/DLR/TCO option.

Contents

About this guide

This guide explains how to extend the MetaSolv Solution business logic layer with additional logic specific to your organization.

Audience

This guide is for individuals who are responsible for developing software to integrate an external application with MetaSolv Solution. This guide assumes the reader has a working knowledge of Oracle 9i, Windows XP Professional, BEA WebLogic Platform 8.1, and Java J2EE.

Additional information and help

To get additional information or help for MetaSolv Solution, refer to the following resources:

- Oracle E-Delivery—Provides access to product software and documentation.
  - Visit the E-Delivery Web site at http://edelivery.oracle.com.
  - Software and product documentation are contained in the Oracle Communications MetaSolv Solution 6.0 Media Pack.
  - Developer documentation is contained in the Oracle Communications MetaSolv Solution Developer Documentation Pack. Access to developer documentation requires a password.
- Oracle MetaLink—Provides access to software patches and a searchable Knowledge Base.
  - Visit the MetaLink Web site at https://metalink.oracle.com/, and log on using your User Name and Password.
  - Click the Patches & Updates tab to search for patches (efixes).
  - Click the Knowledge tab to search for technical bulletins, fixed issues, and additional product information. To narrow your search, click the Communication Apps link under Product Categories on the left side of the page.

Oracle Support

The preferred method of reporting service requests (SRs) is through MetaLink. MetaLink is available 24 hours a day, 7 days a week.

Although it is Oracle's preference that you use MetaLink to log SRs electronically, you can also contact Support by telephone. If you choose to contact Support by phone, a support engineer will gather all the information regarding your technical issue into a new SR. After the SR is assigned to a technical engineer, that person will contact you.

For urgent, Severity 1 technical issues, you can either use MetaLink or you can call Support. Oracle Support can be reached locally in each country. To find the contact information for your country, go to http://www.oracle.com/support/contact.html.

MetaSolv Solution documentation set

This guide is one book in a set of documents that helps you understand and use MetaSolv Solution. Figure 1 shows the complete documentation set.

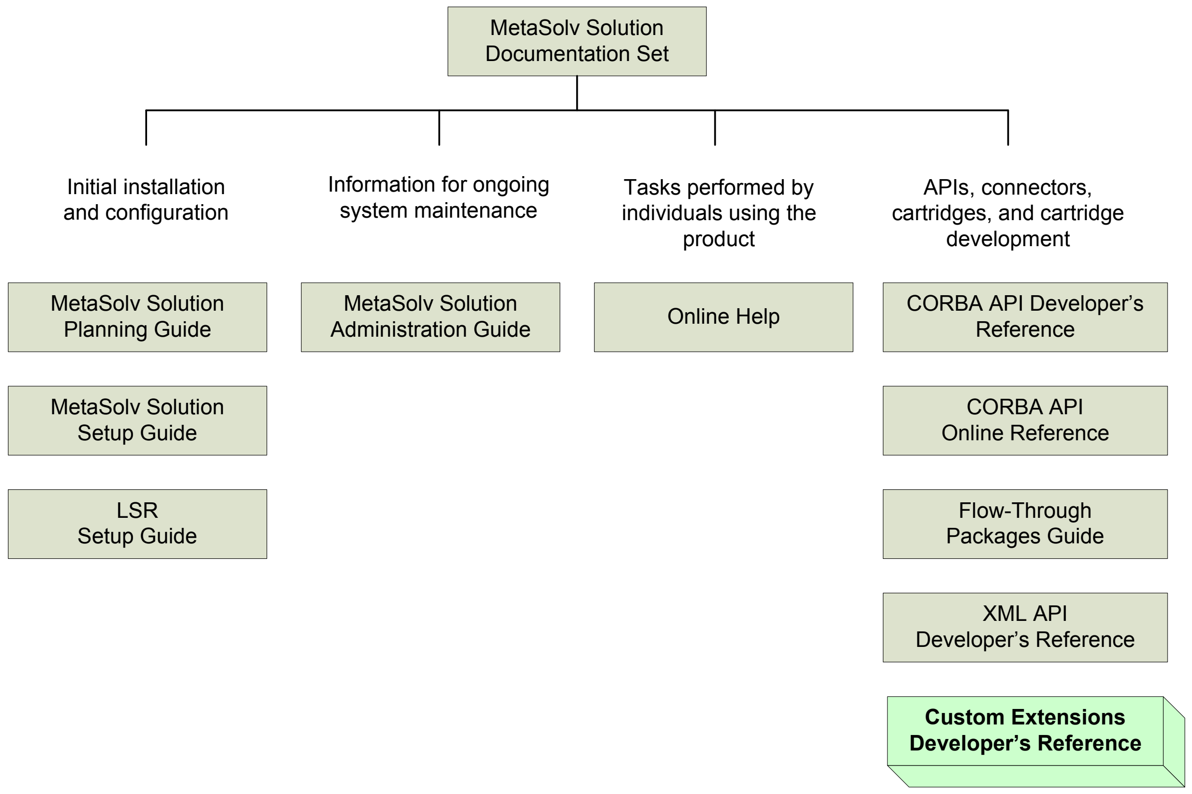
MetaSolv Solution Documentation Set
Initial installation and configuration
Information for ongoing system maintenance
Tasks performed by individuals using the product
APIs, connectors, cartridges, and cartridge development
MetaSolv Solution Planning Guide
MetaSolv Solution Setup Guide
LSR Setup Guide
MetaSolv Solution Administration Guide
Online Help
CORBA API Developer's Reference
CORBA API Online Reference
Flow-Through Packages Guide
XML API Developer's Reference
Custom Extensions Developer's Reference

Figure 1: MetaSolv Solution documentation set

MetaSolv Solution books are delivered in Portable Document Format (PDF). You can view a book online using Adobe Acrobat Reader.

To view a document

Locate the document on the Oracle E-Delivery or Oracle MetaLink Web site and do one of the following:

- Right-click the PDF file and select **Open** from the pop-up menu.
- Double-click the PDF file.

This action starts Acrobat Reader and opens the PDF document you selected. The following figure shows how a document appears in Acrobat Reader:

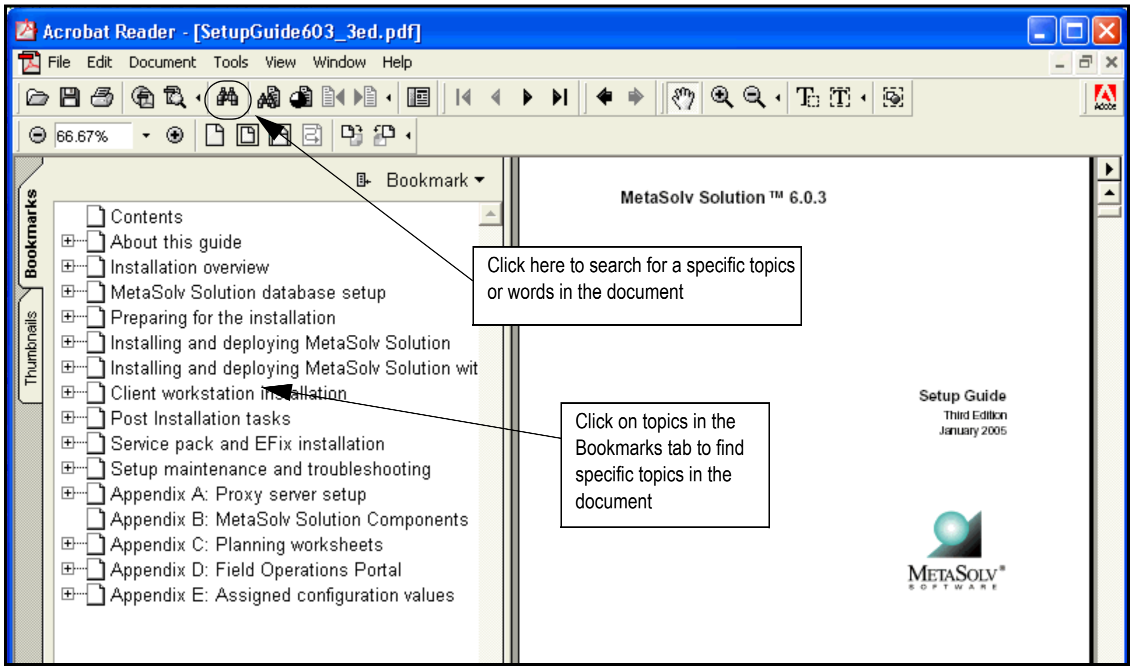
Acrobat Reader - [SetupGuide603_3ed.pdf]
File Edit Document Tools View Window Help
66.67%
Bookmark
Bookmarks
Thumbnails
Contents
About this guide
Installation overview
MetaSolv Solution database setup
Preparing for the installation
Installing and deploying MetaSolv Solution
Installing and deploying MetaSolv Solution wit
Client workstation installation
Post Installation tasks
Service pack and EFix installation
Setup maintenance and troubleshooting
Appendix A: Proxy server setup
Appendix B: MetaSolv Solution Components
Appendix C: Planning worksheets
Appendix D: Field Operations Portal
Appendix E: Assigned configuration values
MetaSolv Solution ™ 6.0.3
Setup Guide
Third Edition
January 2005
METASOLV SOFTWARE
Click here to search for a specific topics or words in the document
Click on topics in the Bookmarks tab to find specific topics in the document

Figure 2: Finding information in a PDF document

1

Extensions overview

This chapter provides basic information about custom extensions and how you can use them to invoke API calls and send messages that support your organization's business processes.

About custom extensions

A custom extension enables you to extend MetaSolv Solution functionality with additional business logic specific to your organization. In other words, extensions provide the ability to make calls to external systems and to send e-mail and JMS messages at predefined execution points, over and above the functionality supported by the MetaSolv Solution application and APIs. Beginning with MetaSolv Solution 6.0.12, there are ten supported execution points that provide the ability to extend logic. Future releases will bring additional supported execution points, which will allow you to extend logic from additional places.

You can develop custom extensions that simply send data to another system, or that both send and receive data. An extension that sends data, and does not expect a response from an external system, is defined as asynchronous. An example of an asynchronous extension is an e-mail message. You might choose to develop an asynchronous extension to send an e-mail when a particular process or event occurs in MetaSolv Solution.

An extension that sends data, and expects a response from an external system, is defined as synchronous. An example of an execution point that can be used to develop a synchronous extension is Assign Queues. You might choose to develop a synchronous extension that executes a custom java class when a particular process occurs in MetaSolv Solution. The java class executes as its own transaction, separate from the process that initiated it.

Developing a custom extension involves several tasks. These tasks, listed below, appear in a conceptual order to help you understand extensions. In reality, these tasks would probably be performed by different people, and at varying times.

1. Define the extension.
2. Identify execution points.
3. Code the extension logic.

Extensions

The first step in developing a custom extension is to define the extension in the GUI. The extension name that you define is the name of the java class that will contain your custom logic.

Execution points

The second step in developing a custom extension is to define the point at which you want the custom extension logic to execute; that is, the process or action that will trigger the invocation of your custom code. You define this execution point by identifying three key pieces of information:

- Building block
- Process point
- Action type

Building block

A building block type is a predefined item in MetaSolv Solution, such as a gateway event, with which you can associate an extension. Building blocks further describe building block types. For example, using the building block type of Gateway Event enables you to associate an extension with a gateway event. You then further define this item by selecting the building block of All Gateway Events. This means you can associate the extension with all gateway events, as opposed to specific events.

Process point

A process point describes general processing that takes place in MetaSolv Solution, for example, gateway event maintenance. To continue with the example used for building blocks, you can associate a process point of GW (gateway) Event Maintenance with the extension. This means the extension logic will be triggered when MetaSolv Solution processes some type of gateway event maintenance.

Like building blocks, process points are predefined by MetaSolv.

Action type

An action type is a specific task or process that takes place in MetaSolv Solution. When you associate an action type with an extension, you are identifying the specific action that triggers the extension logic to execute for a particular extension. To conclude the previous example, you can associate the action type of GW (gateway) Event Failed with the extension. This means the extension logic will be triggered when MetaSolv Solution processes a gateway event and it fails to successfully complete.

Like building blocks and process points, action types are predefined by MetaSolv.

Extension logic

The next step in developing a custom extension is to code a free-form java class that provides additional functionality to support your business processes. As examples, you can code a java class to:

- Make calls to external systems
- Send e-mail notifications
- Send JMS messages
- Invoke other MetaSolv Solution API calls

Invocation methods

This section is not listed as a step in the "About custom extensions" section of this chapter because identifying the execution points is what defines the invocation method(s). Therefore, this is not actually a step that you need to perform. However, it is important to understand the information contained in this section, therefore, it is included in the overview because it addresses, at a high level, how custom extension logic is invoked. Specific information regarding invocations for supported execution points is included in "Appendix A: Supported execution points".

After you define the extension, associate the execution point, and code the logic for your custom extension, it will be invoked from one or more of the places listed below. The invocations are dependent upon the execution points associated with your extension.

- MetaSolv Solution graphical user interface (GUI)
- XML API clients
- CORBA API clients
- Polling servers running on the appserver

The following diagram shows the architecture of MetaSolv Solution and how the various system components interact to support custom extension functionality.

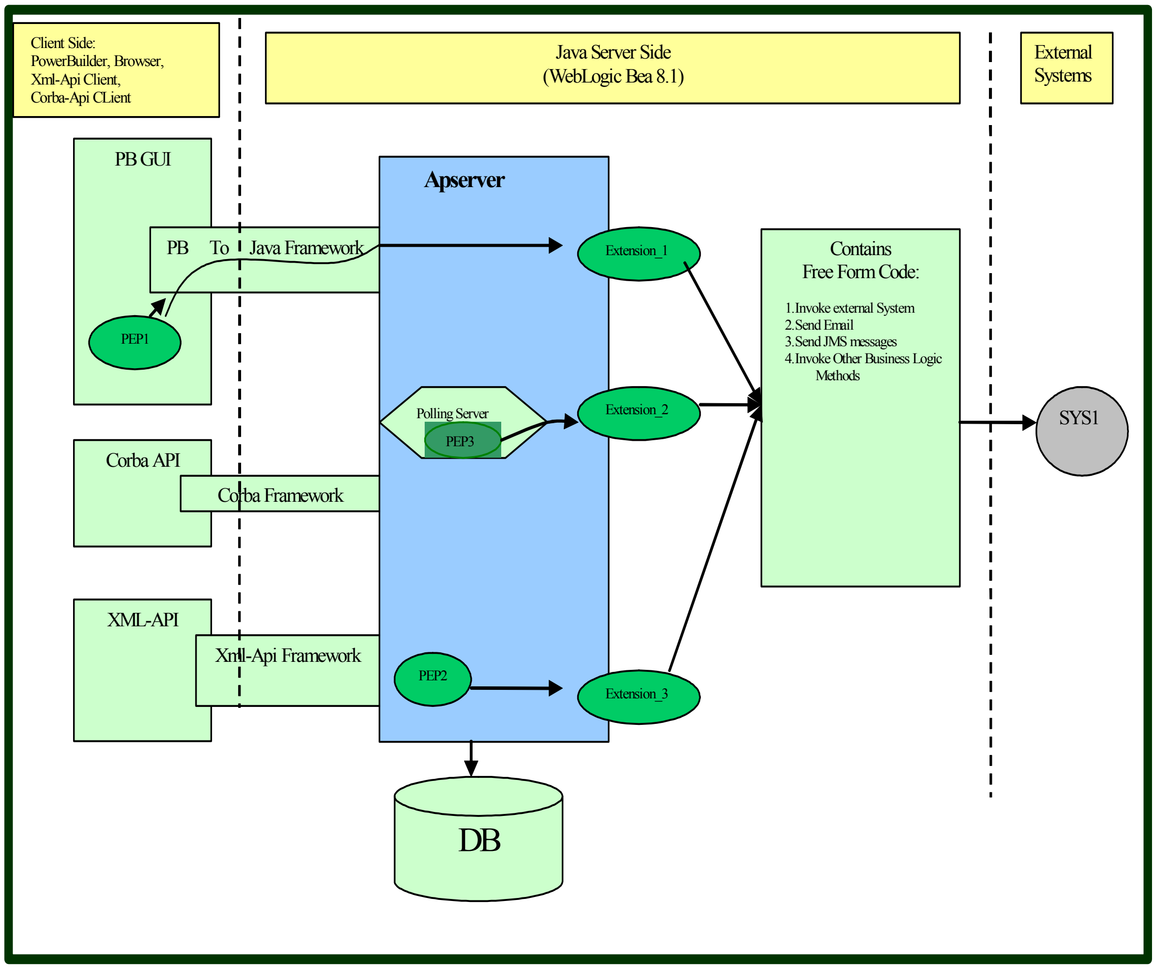

Figure 3: Architecture supporting extension functionality

MetaSolv Solution GUI

You can invoke extension logic through the GUI when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a user assigning a jeopardy code to a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

XML API clients

You can invoke extension logic through a call to an XML API method when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a third party calling the addTaskJeopardyRequest method to assign a jeopardy code to a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

CORBA API clients

You can invoke extension logic through a call to a CORBA API method when the specified action, defined by an execution point (combination of building block, process point, and action type), occurs. For example, a third party calling the deleteTaskJeopardy method to remove a jeopardy code from a task is a specific action that can invoke an extension, if that action is defined as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of Assign Jeopardy.

Polling servers

You can invoke extension logic through polling servers as well. These servers, which need to be configured in the gateway.ini file, are listed on the following page. For detailed information regarding these configurations, refer to the section “Additional configurations” in Chapter 2.

Polling servers can invoke extension logic if the action of the polling server is defined as an execution point. For example, a task that is defined as a system task with a task execution point of Ready, will automatically be picked up by the System Task Server when the task status becomes Ready. If the task completion logic that runs on the server fails, extension logic can be invoked if it defines that as an execution point. Specifically, you would choose the execution point that combines the building block type of Task Type, process point of Task Maintenance, and action type of System Task Failure.

Polling servers and supported execution points

The following polling servers can invoke an extension that is defined with the specified execution point(s). For specific information regarding the supported execution points mentioned here, refer to "Appendix A: Supported execution points".

- Background Processor*
  - System Task Failure
- Gateway Event Server
  - Gateway Event Failure
- Integration Server
  - Gateway Event Failure
  - Late Task
  - Potentially Late Task
- System Task Server
  - System Task Failure

*The Background Processor is not a Java based polling server. Rather, it is a PowerBuilder application that runs in the background.

2

Defining an extension

This chapter explains how to define a custom extension in the GUI. Online Help for defining an extension is available in the form of the help topics listed below.

Open the online Help and type the following window or procedure names in the Search field:

- Extensions window
- Extension Summary window
- Extension Parameters window
- Opening the Extension Summary window
- Creating a new Extension
- Editing an existing Extension
- Deleting an existing Extension
- Associating an Execution Point to an Extension
- Disassociating an Execution Point from an Extension
- Editing an Extension Parameter
- Filtering the Extensions list

Defining an extension in the GUI

For specific GUI instructions on how to define the extension, refer to the online Help procedures Creating a new Extension, Associating a Process Point to an Extension, and Editing an Extension Parameter.

Type of extension

When defining an extension, you must select the Type from a drop-down. The following types display in the drop-down, which is defaulted to Logic.

- Logic

  Logic is the only type of extension that is supported at this time. Logic extensions define associated execution points that, when triggered, invoke the custom extension logic java class defined by the extension name.

- Viewable

  Viewable extensions are not supported at this time.

Name of extension

When defining an extension, you must define the name of the extension. The name of the extension is the name of the java class that will be invoked when an associated execution point is triggered. When naming your extension, be sure to follow java class naming standards such as starting with an upper case letter, using upper and lower case letters to distinguish words, no spaces, etc. Also, do not include the .java file type extension in the name of the extension. For example, if you are defining an extension to call the java class MySpecificLogic.java, name the extension MySpecificLogic.

Execution Mode

When defining an extension, you must select the Execution Mode from a drop-down. The following execution modes display in the drop-down, which is defaulted to Synchronous.

- Synchronous

  A synchronous extension will execute and return specified data. The calling process must wait for the extension to finish before continuing.

- Asynchronous

  An asynchronous extension will execute and not return any data, allowing processing to continue without waiting for the extension to finish.

Associating an execution point with an extension

When defining an extension, you must associate one or more execution points with the extension. Execution points are predefined combinations of a building block, process point, and action type. These execution points have "hooks" in the code that, when triggered, invoke the extension java class. Refer to Chapter 3 for more details on execution points.

Defining the extension parameters

When defining an extension, the parameter IDs and their corresponding default names are displayed on the Extension Parameters window. The types of extension parameters are predefined for each execution point, such as String, int, etc. The corresponding default parameter names may be edited so that is has meaning to your particular usage of it.

Configuring an extension

Gateway.ini configuration

To enable custom extensions, the following changes must be made in the gateway.ini file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/gateway directory. Specifying the CLASSPATH tells the framework where to find your custom extension java class, which must reside in the path specified in the gateway.ini.

1. Save a copy of the gateway.ini file.
2. Open the original gateway.ini file for editing.
3. Add the following line at the end of [Custom] section within the file. (If your gateway.ini file does not have the [Custom] section, you will need to add it.)

◆ For Windows operating systems

CLASSPATH=<MSLV_HOME>/<SERVER_NAME>/appserver/samples/
customExtension;

◆ For Unix operating systems

CLASSPATH=<MSLV_HOME>//<SERVER_NAME>//appserver//samples//
customExtension;

Additional configurations

Additional information regarding configuration requirements for using custom extensions can be found in the MetaSolv Solution Setup Guide. Look for the following note located in "Chapter 5: Upgrading MetaSolv Solution" of the Setup Guide, which will guide you to the section that covers these minor configuration changes.

You will need to manually modify the loggingconfig.xml file and integration.xml file or you will receive an error on your appserver console. Additionally, if using Custom Extensions, you will need to manually modify the gateway.ini file. Detailed information regarding the changes to these files is located in "Appendix H: Configuration Files". For information on Custom Extensions, refer to the Custom Extensions Developer’s Reference.

Custom Extensions are new with 6.0.12. If you have performed a full installation of 6.0.12, these configurations are already in place. The configurations covered in the Setup Guide only need to be set up if you have upgraded from a previous release. (Note: Regarding step 3 above, a full installation will put the classpath for custom extesnions in the gateway.ini file, but you will still need to specify the correct path to your server. For an upgrade, you will need to add the classpath for custom extensions to gateway.ini file as welll as specify the correct path to your server.)

Invoking an extension

Certain execution points are invoked by polling servers, as covered in section "Polling servers and supported execution points" located in Chapter 1. Three of these servers are Java based servers that need to run as part of the appserver. This is accomplished by configuring the gateway.ini file to define the appropriate servers within the [Servers] section as follows:

- Gateway Event Server

  ```
  EVENTPROC=MetaSolv.eventServer.S3Startup
  ```

- Integration Server

  ```
  INTEGRATIONSERVER=com.mslv.integration.integrationServer.S3Startup
  ```

- System Task Server

  ```
  SYSTEMTASKSERVERPROC=com.mslv.core.api.internal.WM.systemTaskServer.S
  ystemTaskServer
  ```

The remaining server, the Background Processor, is not part of the appserver and, therefore, is not configured through the gateway.ini file. To start the background processor, execute the jmaster.exe located in the MSS directory.

3

Identifying an execution point

This chapter explains how to identify an execution point. Once identified, execution points are then associated with an extension, as covered in the previous chapter. Online Help for identifying execution points is available in the form of the help topics listed below.

Open the online Help and type the following window or procedure names in the Search field:

- Execution Point Search and Results window
- Execution Points window
- Searching for an Execution Point
- Toggling between Execution Point Search and Results
- Filtering the Execution Points list

Component options

As you recall from the overview in Chapter 1, an execution point is defined by a combination of three components—its building block, process point, and action type. MetaSolv predefined a number of options for each of these components, along with the combinations of options that represent valid execution points. This section describes the options that are available for each component, as well as the first MetaSolv Solution release in which they are supported.

Building block options

Building blocks are grouped into building block types. Both building blocks and building block types are MetaSolv defined data. The following table lists building block types in the order that they appear in the drop-down list on the Execution Point Search window.

In M6.0.12 and M6.0.13, the building block type drop-down list includes Network Element, Equipment, and Order, which are not listed in following table because they are not applicable to custom extensions. Execution points created from building block types Network Element, Equipment or Order are not allowed to be associated with an extension. To avoid confusion, in M6.0.14 the building block types of Network Element, Equipment, and Order were removed from the drop-down list.

Table 1: Building block type options

Building block type	Supported release
Connection	M6.0.13
Task Type	M6.0.12
Gateway Event	M6.0.12

The following table lists the building blocks defined by MetaSolv that can be used with extensions. The building blocks available for selection depend on the building block type chosen. The building block ID, an Oracle generated number, is included in the information because it is part of the data that will be passed from an execution point to an extension java class.

Table 2: Building block options

Building block	Building block ID	Supported release
All Task Types	1001	M6.0.12
All Gateway Events	1002	M6.0.12
[specific task type*]	[depends on task type*]	M6.0.13
All Connections	409	M6.0.13

*Specific task types are user defined data stored on the TASK_TYPE table. To support the Complete Task execution point for individual task types in M6.0.13, the building block id field (ms_bb_id) was added to the TASK_TYPE table. A row is inserted into the TASK_TYPE table when a new task type is created in Work Management. However, the ms_bb_id field is not populated with the row insertion, rather, it is populated when the task is selected from the Name dropdown field on the Execution Point Search window. Note that the Name dropdown field will list all task types when Task Type is selected in the Building Block Type dropdown.

This docucment does not provide the building block ids for each task type because they are based on user data. Building block ids are not displayed in the application, therefore, they must be manually looked up on the TASK_TYPE table.

Process point options

The following table lists the process points defined by MetaSolv that can be used with extensions. The process points available for selection depend on the building block chosen. The process point ID, an Oracle generated number, is included in the information because it is part of the data that will be passed from an execution point to an extension java class.

Table 3: Process point options

Process point	Process point ID	Supported release
Task Generation	1	M6.0.12
Task Maintenance	101	M6.0.12
GW Event Maintenance	102	M6.0.12
PCONDES Maintenance	103	M6.0.13
VCONDES Maintenance	105	M6.0.15

Action type options

The following table lists the action types defined by MetaSolv that can be used with extensions. The action types available for selection depend on the process point chosen.The action type ID, an Oracle generated number, is included in the information because it is part of the data that will be passed from an execution point to an extension java class.

Table 4: Action type options

Action type	Action type ID	Supported release
Generate	32	M6.0.12
Assign Jeopardy	41	M6.0.12
Reject	42	M6.0.12
Assign Queues	43	M6.0.12
Change Completion Date	44	M6.0.12
System Task Failure	45	M6.0.12
Late	46	M6.0.12
Potentially Late	47	M6.0.12
GW Event Failed	51	M6.0.12
Provision Plan Default	52	M6.0.12
Complete	53	M6.0.13
Select Component or Element	54	M6.0.13
Select Port Address	55	M6.0.13

Component combinations

As explained in each of the previous component sections, there are dependencies between the components. Specifically, action types are dependent on process points, which are dependent on building blocks, which are dependent on building block types.

The following table shows the valid combinations that result from these dependencies. For example, if you choose a building block type of Task Type, your only choice of building block is All Task Types. (This will change in future releases. Currently, the building block type and the building block components happen to have a one-for-one relationship. However, this will change when more specific building blocks such as DD or DLRD are defined.) If you then choose the process point of Task Generation, your only action type choices will be Generate or Provision Plan Default.

Table 5: Valid Combinations

Building block type	Building block	Process point	Action type	Supported release
Task Type	All Task Types	Task Generation	Generate	M6.0.12
			Provision Plan Default	M6.0.12
		Task Maintenance	Assign Jeopardy	M6.0.12
			Reject	M6.0.12
			Assign Queues	M6.0.12
			Change Completion Date	M6.0.12
			System Task Failure	M6.0.12
			Late	M6.0.12
			Potentially Late	M6.0.12
			Complete	M6.0.13
	[specific task type]	Task Maintenance	Complete	M6.0.13
Gateway Event	All Gateway Events	GW Event Maintenance	GW Event Failed	M6.0.12

Building block type	Building block	Process point	Action type	Supported release
Connections	All Connections	PCONDES Maintenance	Select Component or Element	M6.0.13
			Select Port Address	M6.0.13
		VCONDES Maintenance	Select Component or Element	M6.0.15

4

Coding the extension logic

This chapter covers information regarding coding the extension java class. Sample code, which is provided with your installation of M6.0.12 or higher, has concrete code examples of extension java classes. Refer to “Appendix B: Extensions sample code” for detailed information about the sample code.

Inheriting from the extension framework

All extension java classes must extend the extension framework through the class ExtensionRoot, located in the package com.metasolv.custom.common.extension. Extending the extension framework is necessary to access the data passed from the execution point. Therefore, all new extension java classes should contain the following lines of code, or some derivation of them:

```
import com.metasolv.custom.common.extension.ExtensionRoot

public class MyExtension extends ExtensionRoot
```

A derivation of the code could be that the extension java class directly, or indirectly, extends ExtensionRoot. For example, all of the sample source code extends SampleExtensionRoot rather than ExtensionRoot. That is because SampleExtensionRoot extends ExtensionRoot, adding a middle layer to the inheritance that provides common functionality used by all the sample classes. You may wish to create a similar class, or even use the SampleExtensionRoot class, depending on what you are developing.

Note that all of the sample source code implements the class Extension. This is really not necessary because ExtensionRoot implements Extension. Therefore, by inheritance, any class that extends ExtensionRoot implements Extension.

Accessing data passed from the execution point

Overview

Extension java classes cannot define input parameters. Rather, data passed from the execution point can be accessed by the extension java class through the extension framework. Specifically, the class ExtensionRoot defines the following methods:

```
protected final Policy getPolicy()

protected final Entity[] getParameter()
```

Note that while these methods are defined as protected, they are available to the extension java class because it inherits from the class in which the methods are defined (ExtensionRoot). From these two methods, the following data can be retrieved:

- Execution mode

  The execution mode tells you if the execution point that invoked the extension class is defined as synchronous or asynchronous. This information was entered in the GUI when defining the extension.

- Execution point

  The execution point tells you the point at which the extension class was invoked. This information is passed in the form of building block ID, process point ID, and action type ID. The unique combination defines a specific execution point such as Assign Queues or Reject Task.

- Execution point data

  The execution point data is the specific data that is associated with each supported execution point. This information is passed in the form of a name/value pair array. Refer to "Appendix A: Supported execution points" for the specific data that is passed from each execution point.

Class details

Policy class

As mentioned in the overview section above, the method getPolicy() returns Policy. However, it actually returns an instance of the class PlugInPolicy, which extends Policy. Therefore, you can caste the returned Policy to PlugInPolicy, which makes an instance of the class PlugInPolicy available to the extension java class.

The class PlugInPolicy defines the following methods:

```
public String getExecutionMode();

public PlugInExecutionPoint getExecutionPoint();
```

Calling the method getExecutionMode() from the extension java class returns a String that indicates if the execution mode is synchronous or asynchronous. Calling the method getExecutionPoint() returns an instance of the class PlugInExecutionPoint.

The class PlugInExecutionPoint defines the following methods:

int getBuildingBlock();

int getProcessPoint();

int getActionType();

Calling these methods returns the combination of building block ID, process point ID, and action type ID that defines an execution point, as described in "Appendix A: Supported execution points".

Entity class

As mentioned in the overview section above, the method getParameter() returns an Array of Entity classes. Another class, ExtensionData, extends the class Entity. Since ExtensionData is a child of Entity, Entity can be casted to ExtensionData. Casting Entity to ExtensionData makes the Array of ExtensionData available to the extension java class.

The class ExtensionData defines the following method:

```
public NameValuePair[]  getNameValuePairs()
```

Calling this method from the extension java class returns an Array of NameValuePair classes. The name/value pairs represent the specific data that is defined for each supported execution point, as described in "Appendix A: Supported execution points".

Finally, the class NameValuePair defines the following methods:

```
public String getName();
public String[] getValue();
```

Calling these methods returns the String name and the String values. It is important to note that all value data is of type String.

Appendix A: Supported execution points

The preceding chapters described what custom extensions are and how to create them. As mentioned earlier, MetaSolv predefined the components used to define execution points—the building blocks, process points, and action types. This means there are specific execution points that are available for your use.

In addition to predefining "Component combinations" associated with each execution point, MetaSolv developed functionality that supports the invocation of a custom extension java class for each valid combination. This functionality includes:

- "Hooks" that are triggered by an execution point. These "hooks" call the extension framework, which determines what extension class to invoke based on which extensions the execution point is associated with.
- Parameters for each execution point. The parameters are used to pass data that is pertinent to the execution point to the extension class. This data is then available to the extension class and can be used to code your specific business logic.

The supported execution points are listed in the following table. The execution points are grouped by building block, and ordered alphabetically. The number of supported execution points correlates to the number of valid component combinations, and the execution point names correlate to the action type of each valid combination.

Table 6: Supported Execution Points

Building block	Execution Point	Supported release
Connections	Select Port Address	M6.0.13
	Select Component or Element for Physical Connection	M6.0.13
	Select Component or Element for Virtual Connection	M6.0.15
Gateway Events	Gateway Event Failure	M6.0.12
Tasks	Assign Queues	M6.0.12
	Assign Task Jeopardy	M6.0.12
	Change Task Completion Date	M6.0.12
	Complete Task	M6.0.13
	Generate Tasks	M6.0.12
	Late Task	M6.0.12
	Potentially Late Task	M6.0.12
	Provisioning Plan Default	M6.0.12
	Reject Task	M6.0.12
	System Task Failure	M6.0.12

This appendix provides detailed information for each supported execution point, which includes:

- A brief description of the execution point.
- A business example of how you might use the execution point.
- The options you should choose when searching for the execution point to associate it with an extension.
- The data that is sent from the execution point to the extension java class, and, in the case of a synchronous call, the data that is returned from the extension java class to the execution point.*
- How the extension java class is invoked by the execution point, whether it is by the GUI, XML APIs, CORBA APIs, or polling servers.

* The data is housed in an Array of name/value pairs. All value data in the name/value pair is of type String.

Execution Points

Select Port Address

MetaSolv Solution provides the ability to automatically design physical connections through the PCONDES task. This execution point enables you to extend logic that is triggered when the PCONDES task is executed, either manually from the gui or automatically from the System Task Server. The extension logic enables you to select the appropriate port address to use in the physical design of the connection. It executes prior to the existing PCONDES auto-provisioning logic. If a port address is successfully selected by the extension logic, the existing PCONDES auto-provisioning logic is bypassed. If a port address is not selected by the extension logic, the existing PCONDES auto-provisioning logic still executes.

Business example

You enter a PSR order and assign a provisioning plan that defines the PCONDES task as a system task. The PCONDES task is used to automatically design physical connections. When the status of the PCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate port address is selected for the design of the physical connection.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 7: Select Port Address execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	PCONDES Maintenance(103)
Action Type	Select Port Address Element(55)

Data passed / Data returned

This is a recommended synchronous call, therefore data should be returned from the extension java class.

The data that is passed to the extension java class includes:

Table 8: Select Port Address name/value pair input data

Data name	Data value
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
End user location ID	endUserLocationId
Rate code	rateCode
Network system component ID	nsCompId*
Network system ID	nsId*
Network system component key Array	nsCompKey* (String Array comprised of nsCompId and nsId)

* Pass nsCompId and nsId, or pass an Array of nsCompKeys; do not pass both sets of data. If the input data is comprised of the Array of nsCompKeys, custom logic can be written to select which component id will be used. Having this option of input data allows for you to customize your extension code to account for things like load balancing between different elements. For example, if there are three valid elements from which to choose, custom code can select the element which has the most or least capacity available, depending on your specific business requirements.

The data that is returned by the extension java class is as follows:

Table 9: Select Port Address name/value pair return data

Data name	Data value
Equipment ID	equipmentId
Port Address Sequence	portAddrSeq

GUI invocation

From the Work Queue window within Work Management, select a PCONDES task, right-click and select Auto Provision from the pop-up menu. The extension logic executes prior to the existing PCONDES auto provision logic. If a port address is successfully selected by the extension logic, then the existing PCONDES auto provision logic is bypassed. However, if a port address is not selected by the extension logic, the existing PCONDES auto provision logic still executes.

XML API invocation

The Select Port Address execution point is not triggered by the XML API.

CORBA API invocation

The Select Port Address execution point is not triggered by the CORBA API.

Additional invocations

- This execution point can also be triggered by the System Task Server for cases where the PCONDES task is defined as a System Task.

  For this to occur, the System Task Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Select Component or Element for Physical Connection

MetaSolv Solution provides the ability to automatically design physical connections through the PCONDES task. This execution point enables you to extend logic that is triggered when the PCONDES task is executed, either manually from the GUI or automatically from the System Task Server. The extension logic enables you to select the appropriate component or element to use in the physical design of the connection. It executes prior to the existing PCONDES auto-provisioning logic. If a component or element is successfully selected by the extension logic, the existing PCONDES auto-provisioning logic is bypassed. If a component or element is not selected by the extension logic, the existing PCONDES auto-provisioning logic still executes.

Business example

You enter a PSR order and assign a provisioning plan that defines the PCONDES task as a system task. The PCONDES task is used to automatically design physical connections. When the status of the PCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate component or element is selected for the design of the physical connection.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 10: Select Component or Element execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	PCONDES Maintenance(103)
Action Type	Select Component or Element(54)

Data passed / Data returned

This is a recommended synchronous call, therefore data should be returned from the extension java class.

The data that is passed to the extension java class includes:

Table 11: Select Component or Element name/value pair input data

Data name	Data value
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
End user location ID	endUserLocationId

The data that is returned by the extension java class is as follows:

Table 12: Select Component or Element name/value pair return data

Data name	Data value
Network system component key Array	nsCompKey (String Array comprised of nsCompId and nsId)

GUI invocation

From the Work Queue window within Work Management, select a PCONDES task, right-click and select Auto Provision from the pop-up menu. The extension logic executes prior to the existing PCONDES auto provision logic. If a component or element is successfully selected by the extension logic, then the existing PCONDES auto provision logic is bypassed. However, if a component or element is not selected by the extension logic, the existing PCONDES auto provision logic still executes.

XML API invocation

The Select Component or Element execution point is not triggered by the XML API.

CORBA API invocation

The Select Component or Element execution point is not triggered by the CORBA API.

Additional invocations

- This execution point can also be triggered by the System Task Server for cases where the PCONDES task is defined as a System Task.

  For this to occur, the System Task Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Select Component or Element for Virtual Connection

MetaSolv Solution provides the ability to automatically design virtual connections through the VCONDES task. This execution point enables you to extend logic that is triggered when the VCONDES task is executed, either manually from the GUI or automatically from the System Task Server. The extension logic enables you to select the appropriate component or element to use in the virtual design of the connection. It executes prior to the existing VCONDES auto-provisioning logic. If a component or element is successfully selected by the extension logic, the existing VCONDES auto-provisioning logic is bypassed. If a component or element is not selected by the extension logic, the existing VCONDES auto-provisioning logic still executes.

Business example

You enter a PSR order and assign a provisioning plan that defines the VCONDES task as a system task. The VCONDES task is used to automatically design virtual connections. When the status of the VCONDES task becomes Ready, the System Task Server processes the task. The extension logic executes and, based on the selection logic in the extension and the information on the order, the appropriate component or element is selected for the design of the virtual connection.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 13: Select Component or Element execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Connections (409)
Process Point	VCONDES Maintenance (105)
Action Type	Select Component or Element (54)

Data passed / Data returned

This is a recommended synchronous call, therefore data should be returned from the extension java class.

The data that is passed to the extension java class includes:

Table 14: Select Component or Element name/value pair input data

Data name	Data value
Document number	documentNumber
Circuit design ID	circuitDesignId
Service item ID	servItemId
Connection Spec	nstCompTypeConId
Network Configuration Type	nstConfigTypeId
Component Type	networkComponentType

The data that is returned by the extension java class is as follows:

Table 15: Select Component or Element name/value pair return data

Data name	Data value
Network system component key Array	nsCompKey (String Array comprised of nsCompId and nsId)

Returned data validation

The data returned by the VCONDES Maintenance - Select Component custom extension must adhere to certain rules. All components (NS_ID/NS_COMP_ID combination) must pass the following validation logic:

- The NS_COMP_ID must exist in the database.
- The component type of the returned NS_COMP_ID must match the networkComponentType input parameter.
- The NS_ID must exist in the database.
- The network configuration type of the returned NS_ID must match the nstConfigTypeId input parameter.

GUI invocation

From the Work Queue window within Work Management, open the Service Request Virtual Connections window by double-clicking a VCONDES task and then select Auto Provision from the Options menu. The extension logic executes prior to the existing VCONDES auto provision logic. If a component or element is successfully selected by the extension logic, then the existing VCONDES auto provision logic is bypassed. However, if a component or element is not selected by the extension logic, the existing VCONDES auto provision logic still executes.

XML API invocation

The Select Component or Element execution point is not triggered by the XML API.

CORBA API invocation

The Select Component or Element execution point is not triggered by the CORBA API.

Additional invocations

This execution point can also be triggered by the System Task Server for cases where the VCONDES task is defined as a System Task. For this to occur, the System Task Server must be configured to run on the appserver. Refer to the section “Invoking an extension” located in Chapter 2 for specific configuration information.

Gateway Event Failure

MetaSolv Solution provides the ability to change the status of a gateway event to Error. This execution point enables you to extend logic that will execute after the gateway event status change has completed. This execution point is asynchronous so the continuation of the Gateway Event Server process will not be jeopardized.

Business example

You entered an order and assigned a provisioning plan with a task that has an auto-complete gateway event associated with it. When the task becomes Ready, the gateway event automatically fires, but fails. The gateway event status is set to "Error", and the extension logic executes and sends an e-mail notification to the appropriate person regarding the failed gateway event.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 16: Gateway Event Failure execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Gateway Events (1002)
Process Point	GW Event Maintenance (102)
Action Type	GW Event Failed (51)

Data passed

This is required to be an asynchronous call. Data cannot be returned from the extension java class.

The data passed to the Gateway Event Failure extension depends on the gateway event type. There are four types of gateway events listed below. Table 17 shows all the data inputs, but these will vary based on gateway event type.

- Service Request or Order Type
- Service Item or Item Level Type
- Equipment Type
- Design Type

The data passed to the extension java class includes:

Table 17: Gateway Event Failure data value input by event type

data value	Order Type	Item Level Type	Equipment Type	Design Type
documentNumber	yes	yes	no	no
taskId	yes	yes	no	no
taskType	yes	yes	no	no
gatewayName	yes	yes	yes	yes
gatewayEventType	yes	yes	yes	yes
gatewayEventId	yes	yes	yes	yes
gatewayEventName	yes	yes	yes	yes
gatewayEventVersion	yes	yes	yes	yes
serviceItemId	yes	yes	no	no
errorText	yes, if exists	yes, if exists	yes, if exists	yes, if exists

GUI invocation

The Gateway Event Failure execution is not triggered by the GUI.

XML API invocation

The XML API method through which the java class extension is invoked is:

Order Management > updateOrderManagementRequest*

*The updateOrderManagementRequest method defines several choices of input structures. The invocation is applicable only when the input structure chosen is TaskGWEventValue.

CORBA API invocation

The CORBA API method through which the java class extension is invoked is:

Work Management > updateGWEvent

Additional invocations

- This execution point is triggered by the Gateway Event Server.

  For this to occur, the Gateway Event Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

- This execution point is triggered by the Integration Server.

  For this to occur, the Integration Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Assign Queues

MetaSolv Solution provides the ability to assign a provisioning plan to an order. A provisioning plan defines tasks, and assigns work queues to the tasks within the provisioning plan. This execution point enables you to extend logic in the way the work queues are assigned to tasks within a provisioning plan when tasks are generated for an order.

Business example

You built provisioning plans and assigned default work queues to the tasks in every plan. However, for a specific task type, you would like to do the following:

- Assign it to the ABC queue at certain hours of the day, depending on the workload.
- Assign it to the XYZ queue at certain hours of the day, depending on the workload.
- Send an e-mail notification to notify the owner of each work queue when a task is assigned to them.

You can use the assign queues execution point to extend logic to accomplish those tasks.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 18: Assign Queues execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Assign Queues (43)

Data passed / Data returned

This is a recommended synchronous call, therefore data should be returned from the extension java class.

The data that is passed to the extension java class includes:

Table 19: Assign Queues name/value pair input data

Data name	Data value
Document number	documentNumber
Task type Array	taskType
Task number Array	taskId

The data that is returned by the extension java class is as follows:

Table 20: Assign Queues name/value pair return data

Data name	Data value
Work queue ID Array*	workQueueId

*The work queue ID Array is returned in the same order as the input Arrays of task types and corresponding task numbers.

GUI invocation

After you assign a provisioning plan to an order, you click the Queues button to assign the tasks to the appropriate work queues. The execution point is triggered when you click the Queues button on the Task List tab of the Tasks window.

When you click the Queues button, the task list is sent to the extension. The data received back populates the Work Queue field for each task. This logic overrides the default work queues that were assigned to the provisioning plan when it was established. However, you can still select a different work queue for any or all tasks, should you need to do so after the extension logic executes.

XML API invocation

The XML API method through which the java class extension is invoked is:

OrderManagement - > assignProvisionPlanProcedureRequest

CORBA API invocation

The CORBA API method through which the java class extension is invoked is:

WorkManagement -> generateAndSaveTasks

Assign Task Jeopardy

MetaSolv Solution provides the ability to add, change, and delete jeopardy information for tasks. This execution point enables you to extend logic that will execute when jeopardy information on a task changes (in the form of add, change, or delete).

Business example

You assigned a provisioning plan and, from your Work Queue, set up a jeopardy code on a task. The task ends up going into jeopardy. When the jeopardy status changes, the extension logic executes and sends an e-mail notification to the appropriate person regarding the task jeopardy status.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 21: Assign Task Jeopardy execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Assign Jeopardy (41)

Data passed

This is a recommended asynchronous call, therefore no data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 22: Assign Task Jeopardy name/value pair input data

Data name	Data value
Document number	documentNumber
Task type	taskType
Task number	taskId
Work Queue ID	workQueueId

GUI invocation

From the Work Queue, select a task, right-click, and select Jeopardy Status from the pop-up menu. This opens the Task Jeopardy Codes window where jeopardy codes can be added, changed, or deleted. Click OK or the Apply button to trigger the Task Jeopardy execution point.

XML API invocation

The XML API method through which the java class extension is invoked is:

OrderManagement > addTaskJeopardyRequest

CORBA API invocation

The CORBA API methods through which the java class extension is invoked are:

Work Management > addTaskJeopardy
Work Management > deleteTaskJeopardy
Work Management > updateTaskJeopardy

Change Task Completion Date

MetaSolv Solution provides the ability to change a task due date. This execution point enables you to extend logic that will execute when a task due date is changed.

Business example

You entered an order, assigned a provisioning plan, and then supplemented the order to change the due date. The extension logic executes and sends an e-mail notification to the appropriate person regarding the task due date change.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 23: Change Task Completion Date execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Change Completion Date (44)

Data passed

This is a recommended asynchronous call, therefore no data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 24: Change Task Completion Date name/value pair input data

Data name	Data value
Document number	documentNumber
Task Type	taskType
Task number	taskId
Work Queue ID	workQueueId
New revised completion date	newRevisedCompletionDate

GUI invocation

From the Work Queue, select a task, right-click, and select Service Request Tasks from the pop-up menu. This opens the Tasks window > Task List tab where task due dates can be changed. Click OK or the Apply button to trigger the Task Due Date Change execution point, which only executes if any task due dates were actually changed.

Additionally, you can supplement an order to bring up the Tasks window where task due dates can be changed.

XML API invocation

The XML API method through which the java class extension is invoked is:

Order Management > processSuppOrder

CORBA API invocation

The Change Task Completion Date execution point is not triggered by the CORBA API.

Complete Task

MetaSolv Solution provides the ability to complete a task assigned to an order. This execution point enables you to extend logic that will execute when a task completes, either manually from the gui or automatically from the System Task Server.

Business example

You entered a PSR order and assigned a provisioning plan comprised of three tasks. The second task is defined as an execution point and associated to an extension. When the task completes, the extension logic executes and sends an e-mail notification to the appropriate person regarding the task completion.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 25: Complete Task execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001) -- or -- [specific task type] (dynamic)
Process Point	Task Maintenance (101)
Action Type	Complete (53)

Data passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension java class, it must indicate success or failure.

The data passed to the extension java class includes:

Table 26: Complete Task name/value pair input data

Data name	Data value
Document number	documentNumber
Task number	taskId

GUI invocation

From the Work Queue window within Work Management, select a task, right-click and select Complete from the pop-up menu. The extension logic will execute after the task completion logic runs successfully, but before the commit. If the task completion logic fails, the extension logic will not execute. If the extension logic fails, the task will not complete and a rollback will occur.

XML API invocation

The XML API method through which the java class extension is invoked is:

Order Management > updateOrderManagementRequest*

*The updateOrderManagementRequest method defines a choice of input structures. To complete a task, use the input structure CompleteTaskProcedureValue.

CORBA API invocation

The CORBA API methods through which the java class extension is invoked are:

Work Management > completeTask

Work Management > completeTaskOnDate

Additional invocations

◆ This execution point can also be triggered by the System Task Server for cases where the task is defined as a System Task.

  For this to occur, the System Task Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Generate Tasks

MetaSolv Solution provides the ability to generate tasks for an order. This execution point enables you to extend logic that will execute after tasks are generated. Order management also provides the ability to split a PSR order, a process that also generates tasks for the new order created as a result of the split. This execution point also enables you to extend logic that will execute after tasks are generated as a result of a split.

Business example

You entered a PSR order and assigned a provisioning plan. Two of the service items on the order are delayed, and you split the order so the remaining items can be completed. When the order is split, tasks are generated for the new order that is created as a result of the split. The extension logic executes and sends an e-mail notification to the appropriate person regarding the tasks being generated due to the split. Both the original order and the split order information is made available to the extension.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 27: Generate execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Generation (1)
Action Type	Generate (32)

Data passed

This is a recommended asynchronous call, therefore no data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 28: Generate Tasks name/value pair input data

Data name	Data value
Document number	documentNumber
Split document number	splitDocumentNumber

Note that the document will always be passed to the extension java class, but the split document number may or may not be passed, depending on what triggered the task generation. If a split order triggered the task generation, then the split document number, in addition to the document number, will be passed to the extension java class.

GUI invocation

From the Product Service Request window within Order Management, select Options from the menu bar, and then select Task Generation Maintenance from the pop-up menu. This opens the Tasks window > Plan Selection tab. Select a provisioning plan from the drop-down list. Click the Task List tab, and select work queues for each task. Click OK or the Apply button to trigger the Generate execution point, which happens immediately following the creation of the tasks for the order.

XML API invocation

The Generate Tasks execution point is not triggered by the XML API.

CORBA API invocation

The Generate Tasks execution point is not triggered by the CORBA API.

Late Task

MetaSolv Solution considers a task late when the current GMT date is greater than the revised completion date on the task. This execution point enables you to extend logic that will execute when a task becomes late.

This execution point enables you to extend logic that will execute when a task becomes late. Note that this execution point will be triggered only once when the task is determined to be late. It may be triggered again if the revised completion date is updated on the task. There are new fields on the Task table that indicate if an extension has been invoked.

At the point you define this extension, there could be a large number of late tasks already existing in the database. Invoking this extension for each of these tasks can affect system perofrmance. You can manage the system load by modifying the setup values in the integration.xml file. The maxThreads should always be set to 1. However, the queueMaxCapacity can be lowered and the dbPollingInterval increased to allow breaks in the system processing so the late task extensions can be invoked. The following excerpt from the integeration.xml file illustrates this concept:

```
<LateTaskExtensionEvent event_name="LateTaskExtensionEvent">
       <maxThreads>1</maxThreads>
       <queueMaxCapacity>100</queueMaxCapacity>
       <dbPollingInterval>5</dbPollingInterval>
</LateTaskExtensionEvent>
```

Business example

You entered an order and assigned a provisioning plan. One of the tasks becomes late. The extension logic executes and sends an e-mail notification to the appropriate person regarding the late task.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 29: Late Task execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Late (46)

Data passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension java class, it must indicate success or failure.

The data passed to the extension java class includes:

Table 30: Late Task name/value pair input data

Data name	Data value
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId
Organization for employee	organizationName
Employee name	employeeName
Error text for failure	errorText

GUI invocation

The Late Task execution point is not triggered by the GUI.

XML API invocation

The Late Task execution point is not triggered by the XML API.

CORBA API invocation

The Late Task execution point is not triggered by the CORBA API.

Additional invocations

- This execution point is triggered by the Integration Server.

  For this to occur, the Integration Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Potentially Late Task

MetaSolv Solution provides the ability to define the potentially late window of time for each task type. MetaSolv Solution considers a task potentially late when the revised completion date on the task, minus the time defined as the potentially late window, is less than the current GMT date. This comparison will take into account the calendar that is set up by the organization. The calendar relationship is determined from the task's work queue, which is then associated with an employee, and each employee is associated with organization. For an organization, the calendar may reflect non-work days, which would be considered in determining if a task was potentially late.

This execution point enables you to extend logic that will execute when a task becomes potentially late. Note the following regarding the Potentially Late Task execution point:

- This execution point will be triggered only once when the task is determined to be potentially late. It may be triggered again if the revised completion date is updated on the task. There are new fields on the Task table that indicate if an extension has been invoked.
- If the potentially late server event is disabled during the window of time for a potentially late task, and the task passes from a potentially late task to a late task, the potentially late execution point trigger will not execute. When the server event is enabled, and the task is now late, then the Late Task execution point will be triggered.

Business example

You entered an order and assigned a provisioning plan with a task that defines a potentially late window. The task becomes potentially late. The extension logic executes and sends an e-mail notification to the appropriate person regarding the potentially late task.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 31: Potentially Late Task execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Potentially Late (47)

Data passed

This is required to be a synchronous call because existing logic must know if the extension logic executed successfully before continuing. While no task related data needs to be returned from the extension java class, it must indicate success or failure.

The data passed to the extension java class includes:

Table 32: Potentially Late Task name/value pair input data

Data name	Data value
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId
Organization for employee	organizationName
Employee name	employeeName
Error text for failure	errorText

GUI invocation

The Potentially Late Task execution point is not triggered by the GUI.

XML API invocation

The Potentially Late Task execution point is not triggered by the XML API.

CORBA API invocation

The Potentially Late Task execution point is not triggered by the CORBA API.

Additional invocations

- This execution point is triggered by the Integration Server.

  For this to occur, the Integration Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

Provisioning Plan Default

MetaSolv Solution provides the ability to assign a provisioning plan to an order. This execution point enables you to extend logic to default the appropriate provisioning plan to an order, rather than having to specify a particular provisioning plan.

Business example

You built provisioning plans and assigned default work queues to the tasks in every plan. An extension could be added for defaulting a provisioning plan, allowing you to put logic around the default. For example, you can reduce the number of errors that are made in assigning a provisioning plan to an order by basing the assignment on specific data. Additionally, when the extension logic executes, you can send an e-mail notification to the appropriate person regarding the defaulted provisioning plan.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 33: Provision Plan Default execution point

Field name	Option
Execution Mode	Synchronous
Building Block	All Task Types (1001)
Process Point	Task Generation (1)
Action Type	Provision Plan Default (52)

Data passed / Data returned

This is a recommended synchronous call, therefore data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 34: Provisioning Plan Default name/value pair input data

Data name	Data value
Document number	documentNumber
Organization	organization
Jurisdiction	jurisdiction
Service type group	serviceTypeGroup
Order status	status

The data returned by the extension java class is as follows:

Table 35: Provisioning Plan Default name/value pair return data

Data name	Data value
Provision plan ID	provisionPlanId

GUI invocation

From a Service Request window (ISR, PSR, etc.) within Order Management, select Options from the menu bar, and then select Task Generation Maintenance from the pop-up menu. This opens the Tasks window > Plan Selection tab. The Provisioning Plan Default execution point will be triggered just prior to the Tasks window being displayed. If custom logic is executed, and a valid provisioning plan is returned from the extension, that plan will automatically be populated in the drop-down list and the display will proceed to the Task Gantt tab. The user may return to the Plan Selection tab to change the selected plan.

XML API invocation

The Provisioning Plan Default execution point is not triggered by the XML API.

CORBA API invocation

The Provisioning Plan Default execution point is not triggered by the CORBA API.

Reject Task

MetaSolv Solution provides the ability to reject a task. This execution point enables you to extend logic that will execute when a specified task is rejected.

Business example

You assigned a provisioning plan and, from your Work Queue, reject a task. The extension logic executes and sends an e-mail notification to the appropriate person regarding the rejected task.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 36: Reject Task execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	Reject (42)

Data passed

This is a recommended asynchronous call, therefore no data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 37: Reject Task name/value pair input data

Data name	Data value
Document number	documentNumber
Task type	taskType
Task number	taskId
Work Queue ID	workQueueId
Previous task status	priorTaskStatus
Reject reason	note

GUI invocation

From the Work Queue, select a task, right-click, and select Reject Task from the pop-up menu. This opens the Reject Task window where you select, from a list of predecessor tasks, the task that will be set back to Ready status. All tasks between the initial selection and this second selection (tasks in that provisioning plan for that order) will be set back to Pending status. Click OK to trigger the Reject Task execution point. A list of affected tasks will be sent to the extension.

XML API invocation

The Reject Task execution point is not triggered by the XML API.

CORBA API invocation

The CORBA API method through which the java class extension is invoked is:

Work Management > rejectTask

System Task Failure

MetaSolv Solution provides the ability to define a task as a system task. This indicates that the task's completion logic will automatically run on the System Task Server when the task becomes Ready or when the task start date is reached. However, the system task's completion logic may fail. When a system task cannot be completed, the System Task Server rolls back the transaction, transfers the task to the Exception queue, and logs information to the Server Log table. The server log entries associated with a task can be viewed from the work queue by selecting the task, and then clicking the Server Log tab. Tasks are not completed if a gateway event is in error or if a why-missed code cannot be defaulted.

This execution point enables you to extend logic that will execute when a system task fails to complete. This execution point is asynchronous so that the continuation of the System Task Server process will not be jeopardized.

Business example

You entered an order and assigned a provisioning plan with a system task. The task becomes Ready, the System Task Server picks up the task and attempts to complete it, but fails. The extension logic executes and sends an e-mail notification to the appropriate person regarding the failed system task.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options when searching for an execution point to associate with the extension:

Table 38: System Task Failure execution point

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (1001)
Process Point	Task Maintenance (101)
Action Type	System Task Failure (45)

Data passed

This is a recommended asynchronous call, therefore no data should be returned from the extension java class.

The data passed to the extension java class includes:

Table 39: System Task Failure name/value pair input data

Data name	Data value
Document number	documentNumber
Task number or identifier	taskId
Task type	taskType
Work queue identifier	workQueueId
Error text for failure	errorText or note

GUI invocation

The System Task Failure execution point is not triggered by the GUI.

XML API invocation

The System Task Failure execution point is not triggered by the XML API.

CORBA API invocation

The System Task Failure execution point is not triggered by the CORBA API.

Additional invocations

- This execution point is triggered by the System Task Server.

  For this to occur, the System Task Server must be configured to run on the appserver. Refer to the section "Invoking an extension" located in Chapter 2 for specific configuration information.

- This execution point is triggered by the Background Processor.

  For this to occur, the Background Processor must be running. Refer to the section "Invoking an extension" located in Chapter 2 for specific information on how to run the Background Processor.

Email CLR/DLR/TCO

MetaSolv Solution provides the ability to perform a process from the connection print window. This execution point enables you to extend logic that activates upon clicking of the OK button on the print window after closing the email recipient's window. In order to open the email recipient's window, in the Preference window, set the "Enable HTML Email" option to true and select the Email option in the Print window.

You can modify the sample code to fit the email protocol used at a customer site. The sample extension uses the ByteArrayDataSource method available in the mailapi.jar. The mailapi.jar is from JAVAMail4.1by Sun Microsystems. The sample email extension exists in the SendEmailAttachment folder.

If required, download the mailapi.jar from Sun Microsystems website. After downloading, you can include the jar in the CLASSPATH of the appserver environment.

Business example

You can use this custom extension in several ways. One possible use of this extension is to retrieve the saved HTML files from the database and email the files to the appropriate recipients. Other possibilities include displaying the HTML files on an Intranet or providing access to the HTML files from other applications. The HTML attachment exists as a CLOB in the Email_Job_Attachment table.

Execution point definition

When defining the extension in the MetaSolv Solution application, choose the following options while searching for an execution point to associate with the extension:

Select Component or Element execution point

Table 40:

Field name	Option
Execution Mode	Asynchronous
Building Block	All Task Types (409)
Process Point	Task Maintenance (104)
Action Type	Email(56))

Data passed / Data returned

As this is an asynchronous call, therefore extension java class does not return data.

The data that is passed to the extension java class includes:

Email input data

Table 41:

Data Name	Data Value
Job Id	jobid

GUI invocation

From the Connection print window, select the Email checkbox and click OK. The execution occurs on the Print window but the logic will wait till the user clicks OK on the Recipient window and the Recipient window closes. If the user clicks Cancel on the Recipients window, the extension does not execute.

XML API invocation

The Email execution point is not triggered by the XML API.

CORBA API invocation

The Email execution point is not triggered by the CORBA API.

Additional invocations

This execution point is not triggered anywhere else.

B

Appendix B: Extensions sample code

This appendix covers information regarding sample code that is provided with your installation of M6.0.12 or higher.

Using sample code as a reference for best practices

This section covers information regarding best practices for writing java classes to extend the MetaSolv Solution application logic. The best practices are covered in the form of referencing the provided sample code. The sample code demonstrates how to throw an exception, send an e-mail notification, and call a CORBA API method from an extension class.

Exception handling

The mss_ext_samples.jar contains the class SampleExtensionException.java. This class provides sample code that throws an exception from an extension class. The result of an extension class throwing an exception is an entry in the appserverlog.xml file that shows the error text provided by the extension class. No error is shown to the user.

Below is a sample of the message text logged to the appserverlog.xml when this class executes:

```
PlugInReturn object returned from Extension contained errors:
Testing Extension Exception - Sample Error Message
processPoint 101 ActionType 46 BuildingBlock 1001 Caller USER.
```

E-mail notification

The mss_ext_samples.jar contains the class ExtensionFrameworkOneWayTest.java. This class provides sample code that sends an e-mail notification from an extension class.

CORBA API Invocation

The mss_ext_samples.jar contains the class InvokeCorbaAPIExtension.java. This class provides sample code that invokes a CORBA API method from an extension class. The sample code calls the CORBA API method getOrganization, which is defined in the TaskCompletionSubsession of the Work Management CORBA API.

Running the sample code

The extensions sample code provides concrete examples of how to code specific logic in the extension java class such as error handling, sending an e-mail notification, and making an API call. When executed, the sample code also provides concrete examples of the outcome of these actions. You can define any of the sample classes as an extension in the GUI, associate an execution point with the extension, and then trigger the execution point to invoke the sample class extension and see the outcome.

The extension sample code provided with your installation of MetaSolv Solution is listed below, including the first release in which it's supported. All sample code related files are located in the mss_ext_samples.jar file. The installer copies the mss_ext_samples.jar file to your <M6Home>/appserver/samples directory.

- For a full installation, the mss_ext_samples.jar contents are extracted into the appropriate path under your <M6Home> directory. The appropriate path for each file is identified by the path specified in the .jar file.
- For an upgrade, you must manually extract the contents of the mss_ext_samples.jar into the appropriate path under your <M6Home> directory. The appropriate path for each file can be identified by the path specified in the .jar file.

Table 42: Sample code

Sample code	Supported release
AssignWorkQueues	M6.0.12
ProvPlanDefault	M6.0.12
ExtensionFrameworkOneWayTest	M6.0.12
SampleExtensionException	M6.0.12
InvokeCorbaAPIExtension	M6.0.12

For each sample, you will find the following file types in the mss_ext_samples.jar file. (The only exception is the InvokeCorbaAPIExtension sample, which does not have a supporting .xml file because there is no input data needed for this sample.)

- .java—the extension java source file
- .class—the corresponding compiled java class file
- .xml—the supporting xml file that defines sample input data and sample configuration data that is passed to the extension logic

For example, you will find the following three files that support the AssignWorkQueues sample in the mss_ext_samples.jar file:

- AssignWorkQueues.java
- AssignWorkQueues.class
- AssignWorkQueues.xml

AssignWorkQueues

The AssignWorkQueues sample is provided to show extension logic that assigns specific work queues, and uses a synchronous example. The sample logic shows how to return the specific data that the Assign Queues execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point will be logged for your viewing.

Perform the following steps to run the AssignWorkQueues sample code:

1. Through the GUI, define a synchronous extension with the name AssignWorkQueues.
2. Through the GUI, associate the Assign Queues execution point with the extension by searching for the following criteria:
   - Building Block—All Task Types
   - Process Point—Task Maintenance
   - Action Type—Assign Queues
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Look at the AssignWorkQueues.xml file to understand what the expected results should be in step 7. Specifically, the AssignQueues.xml file defines four tasks and the corresponding work queues to which the tasks will be assigned. The work queues will be returned by the AssignWorkQueues extension logic.
6. Through the GUI, trigger the execution point by assigning work queues.
7. Verify the outcome by looking in the GUI, and by looking in the appserserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

ProvPlanDefault

The ProvPlanDefault sample is provided to show extension logic that defaults a provisioning plan, and uses a synchronous example. The sample logic shows how to return the specific data that the Provisioning Plan Default execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point will be logged for your viewing.

Perform the following steps to run the ProvPlanDefault sample code:

1. Through the GUI, define a synchronous extension with the name ProvPlanDefault.
2. Through the GUI, associate the Provisioning Plan Default execution point with the extension by searching for the following criteria:
   - Building Block—All Task Types
   - Process Point—Task Generation
   - Action Type—Provision Plan Default
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure that the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Look at the ProvPlanDefault.xml file to understand what the expected results should be in step 7. Specifically, the ProvPlanDefault.xml file defines a specific provisioning plan ID that will be returned by the ProvPlanDefault extension logic.
6. Through the GUI, trigger the execution point by assigning a provisioning plan to an order.
7. Verify the outcome by looking in the GUI, and by looking in the appserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

ExtensionFrameworkOneWayTest

The ExtensionFrameworkOneWayTest sample is provided to show extension logic that sends an e-mail notification. When the sample code is executed, it shows the outcome of this action, and the notification will be logged for your viewing. This sample also shows:

- How to read an XML file and determine what execution point invoked it.
- How to send an e-mail notification.
- How to read the input name/value pair Array and put that data into an e-mail.

Perform the following steps to run the ExtensionFrameworkOneWayTest sample code:

1. Through the GUI, define an extension with the name ExtensionFrameworkOneWayTest.
2. Through the GUI, associate an execution point with the extension by searching for criteria such as:
    - Building Block—All Task Types
    - Process Point—Task Maintenance
    - Action Type—Assign Jeopardy
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure that the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Look at the ExtensionFrameworkOneWayTest.xml file to understand what the expected results should be in step 7. Modify the data, such that the e-mail recipient is a valid address that can check for the mail notification, and the SmtpServerKey value is valid for your location.
6. Through the GUI, trigger the execution point that was selected in step 2.
7. Verify the outcome by looking in designated e-mail inbox, and by looking in the appserserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/ appserver/logs directory.

SampleExtensionException

The SampleExtensionException sample is provided to show extension logic that sends an e-mail notification and throws an exception. The code will always throw an exception. When the sample code is executed, it shows the outcome of this action in the form of the e-mail notificaiton, and in the form of a logged error if the extension is defined as synchronous. Note the following:

- If the extension is defined as asynchronous, the extension framework will not log an error, and only the e-mail notification will occur.
- If the extension is defined as synchronous, the extension framework will log an error to the log file, in addition to the e-mail notificaiton being sent.

Perform the following steps to run the SampleExtensionException sample code:

1. Through the GUI, define a synchronous extension with the name SampleExtensionException.
2. Through the GUI, associate an execution point with the extension by searching for criteria such as:
    - Building Block—All Task Types
    - Process Point—Task Maintenance
    - Action Type—Assign Jeopardy
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure that the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Look at the SampleExtensionException.xml file to understand what the expected results should be in step 7. Modify the data, such that the e-mail recipient is a valid address that can check for the exception notification, and the SmtpServerKey value is valid for your location.
6. Through the GUI, trigger the execution point that was selected in step 2.
7. Verify the outcome by looking in the appserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

InvokeCorbaAPIExtension

The InvokeCorbaAPIExtension sample is provided to show how to code CORBA API calls in the extension logic. When the sample code is executed, it also shows the outcome of this action. Specifically, the sample calls the CORBA API mehtod getOrganization(), so the organization will be logged for your viewing.

Perform the following steps to run the InvokeCorbaAPIExtension sample code:

1. Through the GUI, define an extension with the name InvokeCorbaAPIExtension.
2. Through the GUI, associate an execution point with the extension by searching for criteria such as:
   - Building Block—All Task Types
   - Process Point—Task Maintenance
   - Action Type—Assign Jeopardy
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure that the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Through the GUI, trigger the execution point that was selected in step 2.
6. Verify the outcome by looking in the appserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

SelectComponent

The SelectComponent sample is provided to show extension logic that selects a component or element, and uses a synchronous example. The sample logic shows how to return the specific data that the Select Component or Element execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point will be logged for your viewing.

This sample is different from the others in that it is very specific in its function. Other samples are open-ended and can apply to several execution points. This sample code calls specific methods to accomplish the component selection. Java documentation is provided in the sample code to give you additional information about the methods that the sample code calls.

Perform the following steps to run the SelectComponent sample code:

1. Through the GUI, define a synchronous extension with the name SelectComponent.
2. Through the GUI, associate the Select Component or Element execution point with the extension by searching for the following criteria:
   - Building Block—All Connections
   - Process Point—PCONDES Maintenance
   - Action Type—Select Component or Element
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Through the GUI:
   - Set up a DSLAM network location.
   - Add a network element of type DSL Multiplexer to the DLSLAM network location.
   - Add a DSL card with an available port matching the rate code of the ordered service to the DSL Multiplexer.
   - Enter a PSR order with an end user location that has the same zip code as the DSLAM network location.
   - On the PSR order, add a service to the end user location that can be auto provisioned.
   - Assign a provisioning plan to the order that defines the PCONDES task.
6. Through the GUI, trigger the execution point by completing the PCONDES task.
7. Verify the outcome by looking in the GUI, and by looking in the appserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

SelectPort

The SelectPort sample is provided to show extension logic that selects a port address, and uses a synchronous example. The sample logic shows how to return the specific data that the Select Port Address execution point is expecting. When the sample code is executed, it also shows the outcome of this action. Specifically, the data that was passed back to the execution point will be logged for your viewing.

This sample is different from the others in that it is very specific in its function. Other samples are open-ended and can apply to several execution points. This sample code calls specific methods to accomplish the port selection. Java documentation is provided in the sample code to give you additional information about the methods that the sample code calls.

Perform the following steps to run the SelectComponent sample code:

1. Through the GUI, define a synchronous extension with the name SelectPort.
2. Through the GUI, associate the Select Port Address execution point with the extension by searching for the following criteria:
   - Building Block—All Connections
   - Process Point—PCONDES Maintenance
   - Action Type—Select Port Address
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Ensure the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
5. Through the GUI:
   - Set up a DSLAM network location.
   - Add a network element of type DSL Multiplexer to the DLSLAM network location.
   - Add a DSL card with an available port matching the rate code of the ordered service to the DSL Multiplexer.
   - Enter a PSR order with an end user location that has the same zip code as the DSLAM network location.
   - On the PSR order, add a service to the end user location that can be auto provisioned.
   - Assign a provisioning plan to the order that defines the PCONDES task.
6. Through the GUI, trigger the execution point by completing the PCONDES task.
7. Verify the outcome by looking in the GUI, and by looking in the appsserverlog.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/logs directory.

SelectComponentForVirtual

The SelectComponentForVirtual sample is provided to show extension logic that selects a component or element for a virtual connection using a synchronous call. The sample logic reads the expected values (NS_ID and NS_COMP_ID) from the corresponding XML file, but shows how to return the data that the Select Component or Element execution point is expecting. When the sample code is executed, it shows the outcome of this action by logging the input parameters to the console.

The SelectComponentForVirtual sample is provided to show extension logic that selects a component or element for a virtual connection using a synchronous call. The sample logic reads the values (NS_ID and NS_COMP_ID) from the corresponding XML file. Even though the sample logic uses values from an XML file instead of performing actual logic to retrieve those values, it does demonstrate how to format the return data as required by the calling method. When the sample code is executed, it shows the outcome of this action by logging the input parameters to the console.

Perform the following steps to run the SelectComponentForVirtual sample code:

1. Through the GUI, define a synchronous extension with the name SelectComponentForVirtual.
2. Through the GUI, associate the Select Component or Element execution point with your newly created extension by searching for the following criteria:
   - Building Block-All Connections
   - Process Point-VCONDES Maintenance
   - Action Type-Select Component or Element
3. Ensure the gateway.ini entry that defines the sample code path reflects the correct location of the sample files extracted from the mss_ext_samples.jar file.
4. Navigate to the SelectComponentForVirtual.xml file in the <MSLV_HOME>/ <SERVER_NAME>/ appserver/samples/customExtension/xml directory. The keys in this file represent the desired Network System (NS_ID) and Component (NS_COMP_ID) for the virtual connection to be provisioned to. This file will be read by the custom extension in Step 6 and therefore you must modify these key values to represent the actual corresponding data in your database.
5. Ensure the logging level is set correctly by checking the loggingconfig.xml file, located in the <MSLV_HOME>/<SERVER_NAME>/appserver/config directory.
6. Through the GUI:
   - Enter a PSR order with one or more virtual connections.
   - Assign a provisioning plan to the order that defines the VCONDES task.
   - Open the Service Request Virtual Circuits window by opening the VCONDES task.

- Select one or more connections and then select Auto Provision from the Options menu.
- Verify the outcome by looking in the GUI, and by looking in the <SERVER_NAME>.mss.xml file, located in the <MSLV_HOME>/ <SERVER_NAME>/appserver/logs directory.