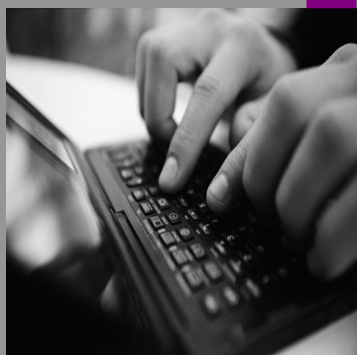


How-to Guide
SAP NetWeaver '04

How To ... Push data from BI to XI (including receiver examples)

Version 1.00 – May 2004

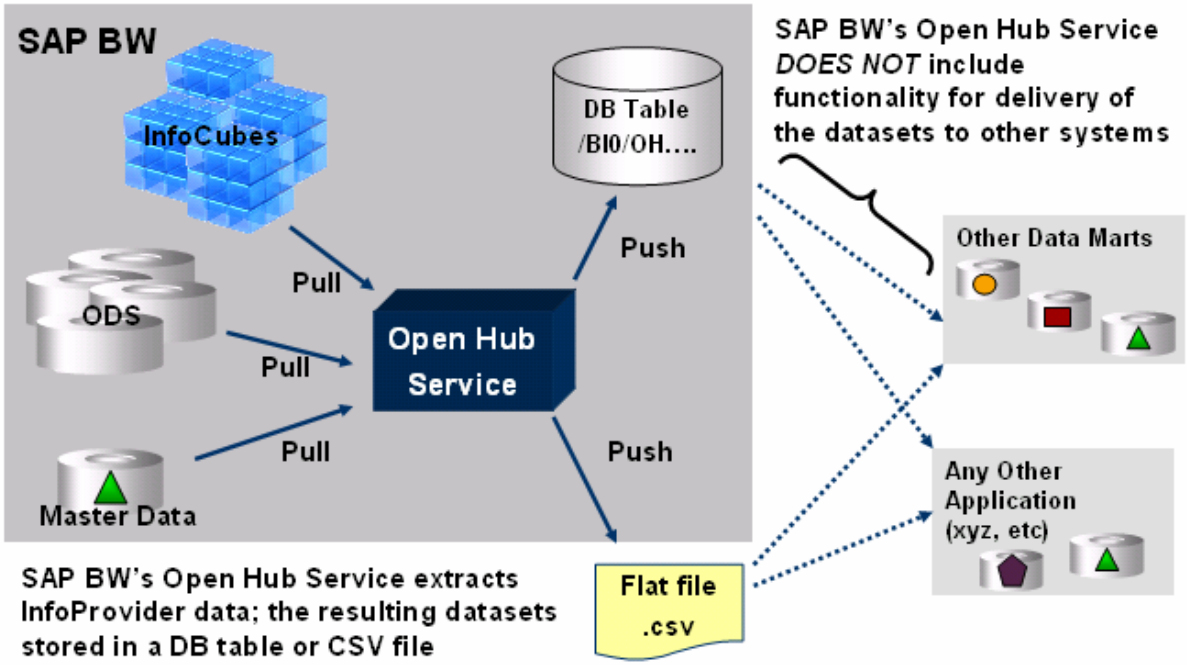
Applicable Releases:
SAP NetWeaver '04 (BW 3.5 and XI 3.0)

1 Business Scenario

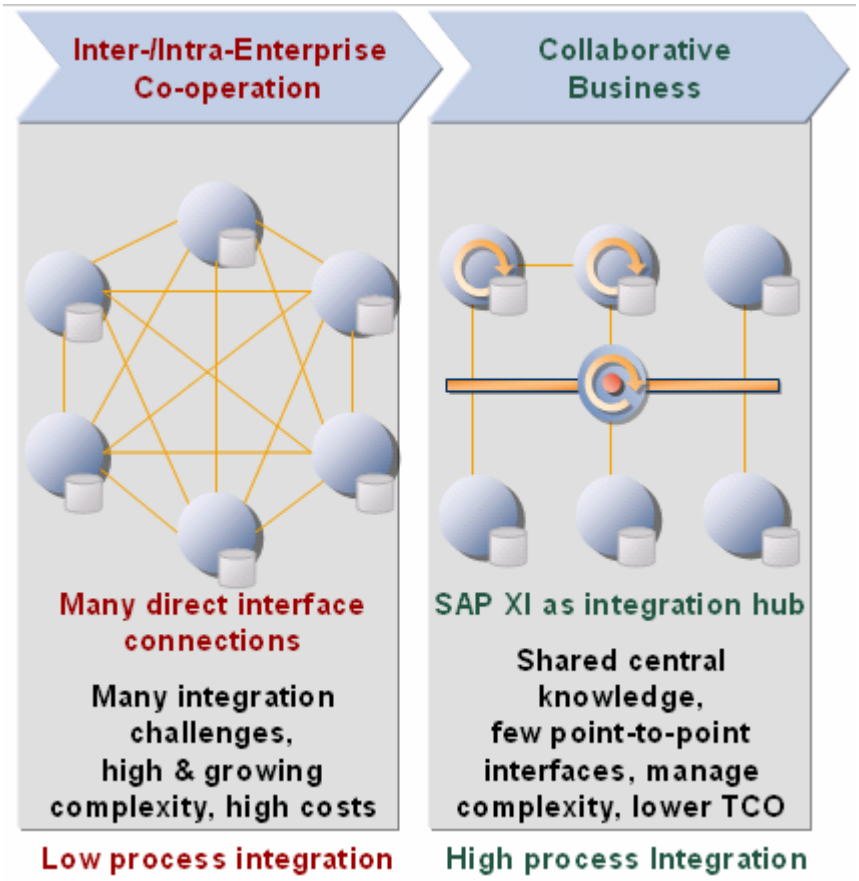
Working with the powerful analytical functionality of the SAP's Business Information Warehouse (SAP BW) typically involves accessing data stored in InfoProviders, where the data is kept in structures represented by database tables. These InfoProviders take the form of InfoCubes (multidimensional models), ODS (Operational Data Store) objects, or master-data bearing characteristics. SAP BW offers robust features and functionality for loading data from many different sources into the InfoProviders.

Sometimes projects encounter a requirement to obtain specific sets of data stored in SAP BW, in order to load those datasets into some other system. Introduced with SAP BW 3.0B, the Open Hub feature offers the ability to extract data stored in SAP BW InfoProviders, and the resulting datasets then can be then taken and loaded into any other system.

When data is staged for an external system using the Open Hub, the data is extracted in a controlled manner, and the resulting dataset is stored in one of two places, depending on customizing: in a specific database table (within the RDBMS where the SAP BW system runs), or written to a CSV file on the application server (where the BW instance runs). Amongst the key features of the Open Hub service is delta processing, where only the relevant changes to data are prepared into a dataset for use upstream. The Open Hub does not offer any mechanism to deliver the resulting datasets from the table or file to the ultimate receiving system (where the data is to be loaded). This is where we find the requirement that is addressed in this paper: specific sets of data are needed from the SAP BW system, for use by one or more other systems, and there exists a need to ensure the consistent delivery and upload of these datasets.



SAP Exchange Infrastructure (SAP XI) functions as a central integration hub, where interfaces and information exchange can be strategically managed. An effective deployment of SAP XI reduces the number of "peer-to-peer" connections in interface development and management. Instead, a significant number of the interfaces can be designed to be routed through the SAP XI system, decoupling interface dependencies, managing complexity, and ultimately lowering total cost of ownership.



SAP XI is designed to effectively facilitate messaging between systems, where specific datasets are exchanged. Inside a message, any dataset exchanged is held in a container known to XI only as the payload of the message. As SAP XI offers distinct advantages for managing interfaces in general, in this paper we present an example which utilizes SAP XI to manage the delivery of data from the SAP BW system Open Hub to several receiver systems.

Cost center planning data is used here as an example, where this planning data would updated into an InfoCube in SAP BW via the planning functions of BPS, and then that data is subsequently needed by other applications. The techniques described herein could be applied for any case where a specific type of dataset is extracted regularly from SAP BW utilizing the Open Hub service (not only planning applications). This paper demonstrates how SAP XI can be leveraged to manage the cross-system business processes involved in delivering the Open Hub datasets to multiple receivers, including sending an XML file and updating data in a database table.

2 Introduction

When sending data to a receiver system you like to insure that the data has been delivered properly. This paper offers an example (which follows the dataset delivery process) by sending a message back to SAP BW indicating successful delivery to a receiver.

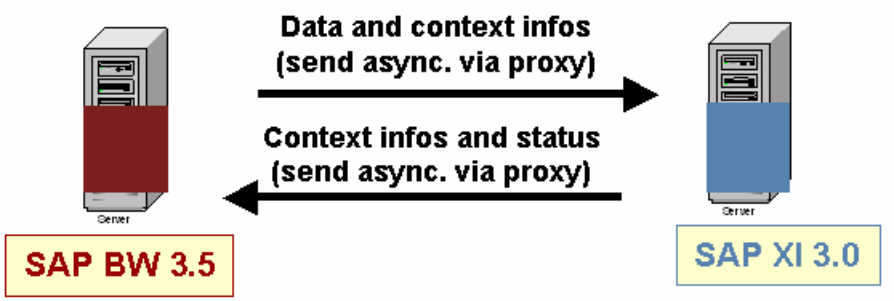
- Once the data has been sent to XI successfully, the BW processes are completed. In other words: XI has to deal with potential receiver problems, and the sender BW system is not aware of any problems.

  If you want to try a more basic scenario, where no delivery status information of any kind is sent back to the BW process chains framework, then simply omit the portions of the interface objects that are built for handling the return message. Additionally you can drop the ProcessID parts of the data type definitions.

- Once the data has been sent to XI the sending process enters a special state (the process chain is active and on hold) waiting for a call back from XI containing the error status.

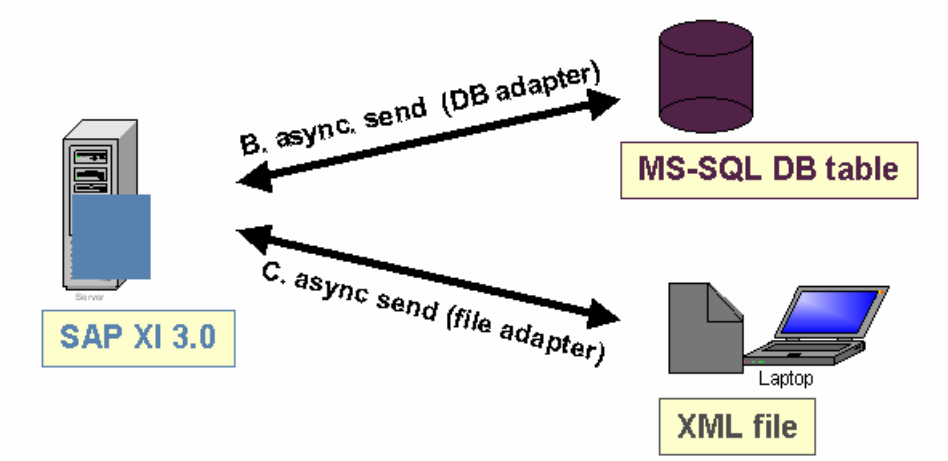
Once the transmission from XI to the receiver system has been done successfully (according to XI monitoring) the sample receivers *flat file* and *database* have stored the transaction data. The application check shown in this How to consists of sending feedback to SAP BW once the data is updated. In our implementation, a polling process is used, where either a file system directory or a database table is checked to ascertain delivery status. It would be possible to extend the scenario, by adding additional detailed handling for statuses of delivery failure.

The first part of this paper describes a technique that can be utilized to push data from SAP BW to SAP XI. This push includes a set of transaction data, to be sent to receiver system(s), as well as context information about the sending object in BW (which is a process in a process chain). The context information is needed in order to route delivery status information back to this sending process chain object in BW, allowing monitoring of delivery success.



The solution described will use the OpenHub to extract data into an intermediate storage (data base table). The Open Hub InfoSpoke extraction is run through a process chain. The next step of the chain utilizes a custom process type, which reads the staged data, adds context data, and then sends this entire enhanced dataset to XI using a proxy interface.
All customizing steps needed for the communication shown above will be described in chapter 3 (except triggering the return message as this requires a proper receiver system).

In chapter 4 you will find different examples for receiver systems and a detailed step-by-step instruction on how to configure them appropriately. It should be noted that these receiver systems become sender systems once they receive the dataset, in order to send a return message indicating successful delivery.



In order to determine status of the delivery, there must be an application existing on the receiving system to provide some handling of the receipt of the dataset. The receiver system examples included in this document utilize XI's adapters: the file adapter and database adapter. Upon successful receipt of a dataset, the adapter framework sends a success message, which is routed back to the process chain object. This success message is sent when either there is successful delivery of the XML file to a remote server, or upon successful update of the dataset into tables in a remote database, an MS-SQL system in the example.

3 Step By Step: Setup BW – XI communication

The example will begin in XI with the definition of interface objects for in-/and outbound communication with the BW system. These generic definitions will then be used to generate ABAP proxies in BW. The objects will establish the link between the two systems.
The use of proxies is an interesting feature of this example, as many people accustomed to working with different types of interfaces between SAP systems typically utilize RFC, ALE, IDoc, etc. This proxy interface between BW and XI is characteristic of techniques typically associated with the concept of building an Enterprise Services Architecture.
Once the technical infrastructure has been established for the interface between BW and XI, the proxies are bound to the process chain infrastructure by creating a custom process type. More specifically, ABAP code is implemented which reads datasets from the database table (to which the Open Hub has staged the dataset). In order to deliver the dataset from the staging table in the database to XI, the methods of the generated proxy are called, utilizing the configuration that defines the interface between the systems. The source code that performs the read and delivery is wrapped inside a custom process type, thus inheriting the general properties of the process chains framework (such as active monitoring of its status and writing a log).

The custom process type remains in a "wait" status within the process chain monitoring framework (but the object is not kept in runtime!). The provided example implements status return message handling, using an inbound proxy. The custom process type status is changed to "successful", upon receipt of a return message indicating the delivery status to the receiving system. Please note that in the samples shown the status of the custom process type will only change in case of success.

Now we create a process chain that combines the InfoSpoke (OpenHub technique) and the newly created infrastructure. Thus, the process chain in the BW system consists of a start process (scheduled to start at a specified date and time), an Open Hub InfoSpoke data extraction process, and a custom process type that reads and sends the staged data to XI.

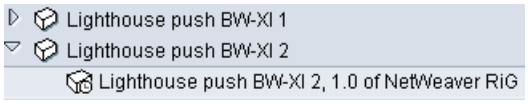
Note that this section of the paper deals with only the first half of the entire scenario, namely the portion where BW sends a dataset to XI. To construct a complete scenario, at least one receiver system, plus the message routing definitions must be defined; this portion will be described in chapter 4, where the configuration of several example receiver destinations is detailed.

As a starting point we assume that …
… a software component already exists. If this is not the case, please navigate to the System Landscape Directory and follow the steps described in chapter 5 Create a software component.
… the BW and the receiver systems are maintained as business system within the software landscape directory and assigned to the integration server.

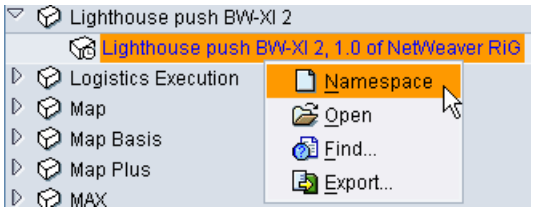
3.1 Define interface objects for BW-XI communication

1. Launch the overview of ***Exchange Infrastructure Tools*** (transaction SXMB_IFR on your XI system) and start the Integration Repository.

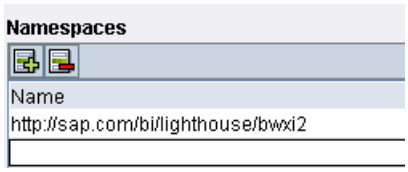
2. Select a software component. In our case we will use the ***Lighthouse push BW-XI 2***. If you haven't created a component so far please refer to chapter 5 Create a software component.



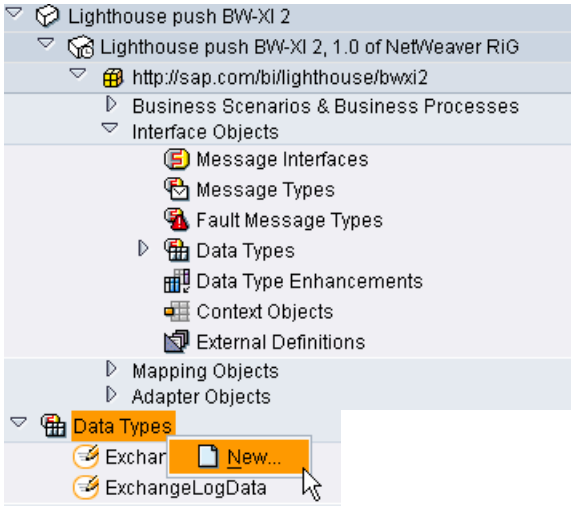
3. Now create a namespace.



4. In our example we will use http://sap.com/bi/lighthouse/bwxi2. It is recommended to replace the *sap.com* by your company URL, as the namespace has to be unique.

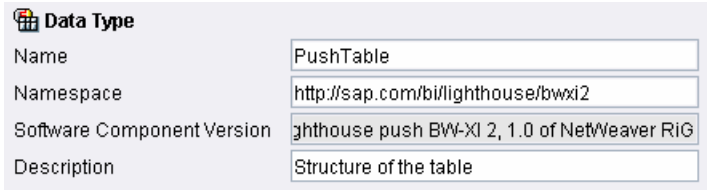


5. Once you have saved the namespace, you should see a list of similar to the one pictured.



6. Create a data type by right clicking on the node ***Data Types***.

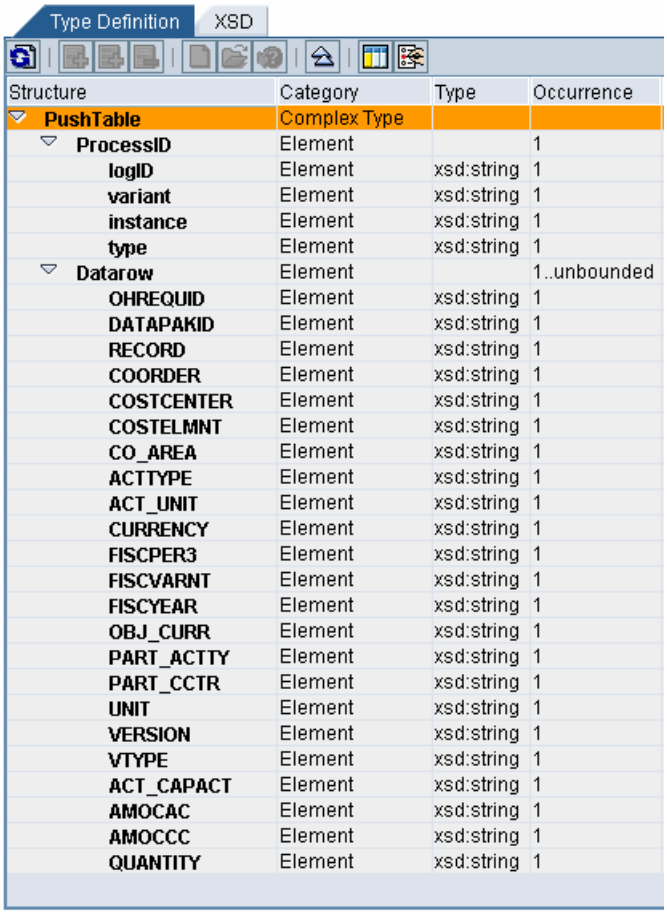
7. Fill the input fields as shown and press the create button.



8. Now we create the data structure ***PushTable*** that will be populated by BW when sending the data to XI.

It contains the row definition of the data to be sent and context information.

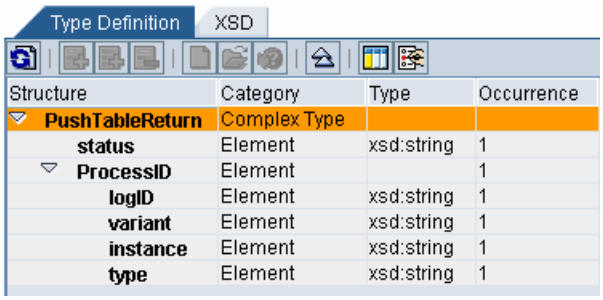
The context information is needed as we will initiate a call back to BW later. The items included under ProcessID represent context information that identifies the specific process chains object that sends the dataset. This context information holds the identification of the specific BW object that originally sent the dataset.



Structure	Category	Type	Occurrence
PushTable	Complex Type		
ProcessID	Element		1
logID	Element	xsd:string	1
variant	Element	xsd:string	1
instance	Element	xsd:string	1
type	Element	xsd:string	1
Datarow	Element		1..unbounded
OHREQUID	Element	xsd:string	1
DATAPAKID	Element	xsd:string	1
RECORD	Element	xsd:string	1
COORDER	Element	xsd:string	1
COSTCENTER	Element	xsd:string	1
COSTELMNT	Element	xsd:string	1
CO_AREA	Element	xsd:string	1
ACTTYPE	Element	xsd:string	1
ACT_UNIT	Element	xsd:string	1
CURRENCY	Element	xsd:string	1
FISCPER3	Element	xsd:string	1
FISCVARNT	Element	xsd:string	1
FISCYEAR	Element	xsd:string	1
OBJ_CURR	Element	xsd:string	1
PART_ACTTY	Element	xsd:string	1
PART_CCTR	Element	xsd:string	1
UNIT	Element	xsd:string	1
VERSION	Element	xsd:string	1
VTYPE	Element	xsd:string	1
ACT_CAPACT	Element	xsd:string	1
AMOCAC	Element	xsd:string	1
AMOCCC	Element	xsd:string	1
QUANTITY	Element	xsd:string	1

9. Repeat the steps for creating a data type for the new structure ***PushTableReturn***.

This structure is used to send delivery status information to the appropriate object in the BW process chains framework.



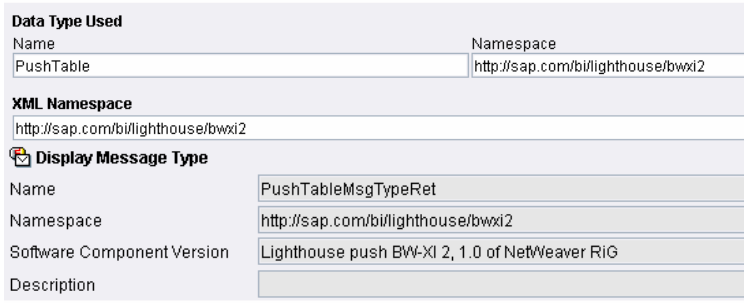
Structure	Category	Type	Occurrence
PushTableReturn	Complex Type		
status	Element	xsd:string	1
ProcessID	Element		1
logID	Element	xsd:string	1
variant	Element	xsd:string	1
instance	Element	xsd:string	1
type	Element	xsd:string	1

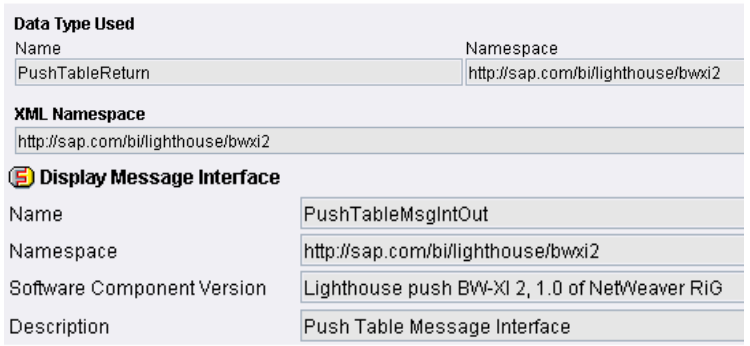
10. Based on the data types we create a message type. Right click on the node Message Types, fill the fields as shown and press create.

11. Enter the name ***PushTable*** and your namespace for the data type used.

Hint: use the value help or drag&drop to avoid mistyping.

12. Repeat the steps for creating a message type, this time for a new message type ***PushTableMsgTypeRet.***

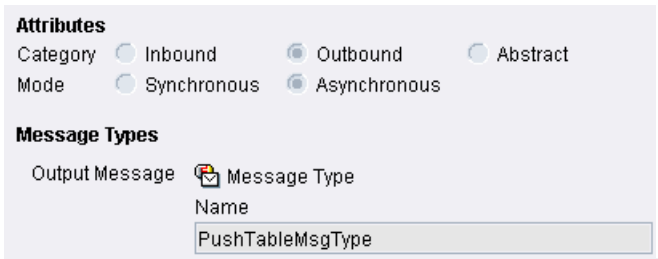




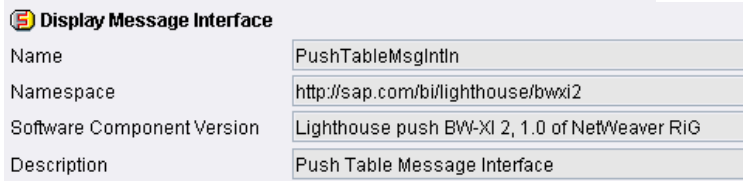
13. Right click on the node Message Interfaces and create a new Message Interface with the name ***PushTableMsgIntOut***. Complete the fields as shown and press create.

14. For attributes assign the category to outbound and set the mode to asynchronous.

Choose the message type ***PushTableMsgType***.

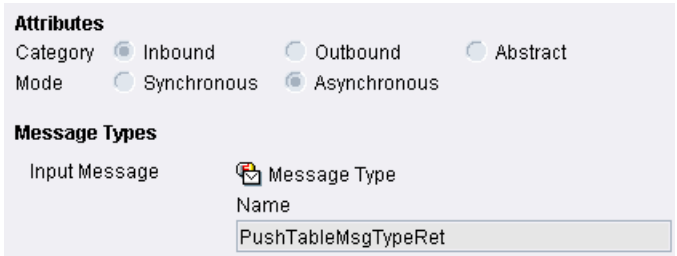


15. Right click on the node Message Interfaces and create a new Message Interface with the name ***PushTableMsgIntIn***. Complete the fields as shown and press create.

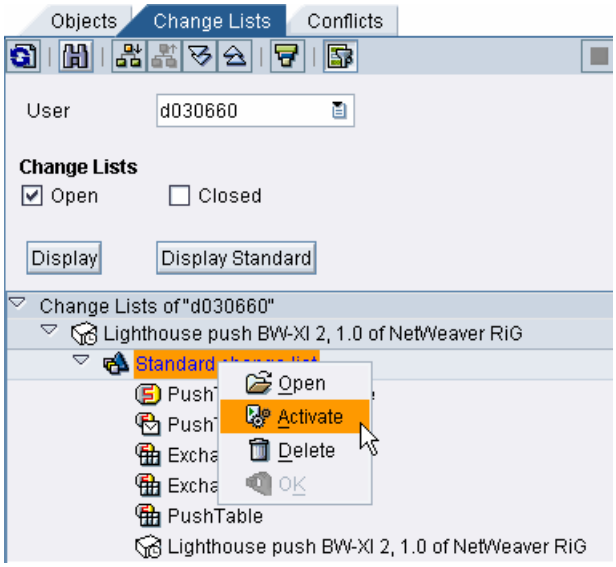


16. For attributes assign the category to outbound and the mode to asynchronous.

Choose the message type ***PushTableMsgTypeRet***.



17. Choose the tab *Change Lists* and activate the created objects.

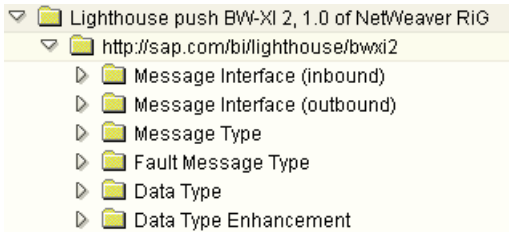


18. The Interface Objects for BW XI communication are set up now.

The next chapter describes configuration tasks to be performed in the BW system.

3.2 Create and implement ABAP-proxies for BW-XI communication

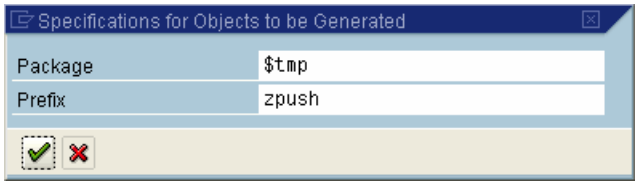
1. Start transaction SPROXY in your BW System and select the software component used in the previous step.



2. When expanding the tree you will find the message interfaces we created in XI (PushTableMsgIntOut and PushTableMsgIntIn).

Right click on the node ***PushTableMsgIntOut*** and choose ***create*** from the context menu.

3. Specify the package and a prefix that will be used as a name space of the generated proxy, interfaces and structures.



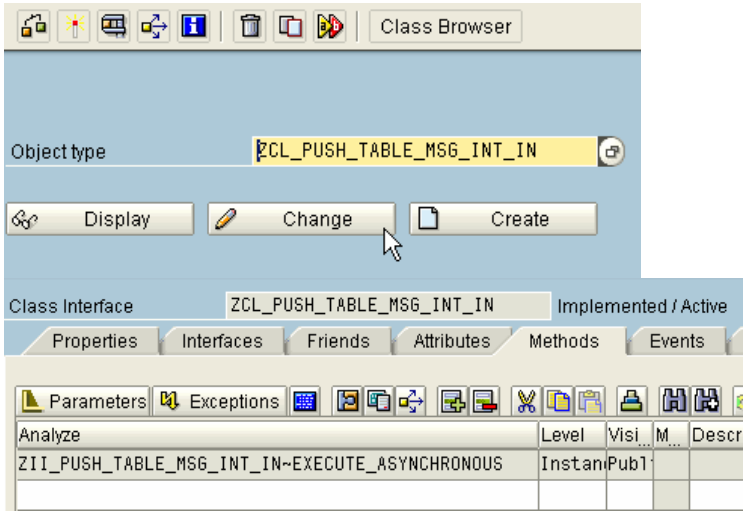
4. Activate the generated proxy and structures. You might receive a warning about the length of the name – you can ignore it – the system will shorten automatically.

5. Repeat the generation steps for ***PushTableMsgIntIn***.

6. The generation is now complete but the implementation of the inbound message interface PushTableMsgIntIn is outstanding.

7. Start transaction SE24, enter the class name ZCL_PUSH_TABLE_MSG_INT_IN and press the *change* button.

8. Double click on the method execute_asynchronous to invoke the ABAP editor.



9. Add the following coding to the method, and then activate the newly created source code.

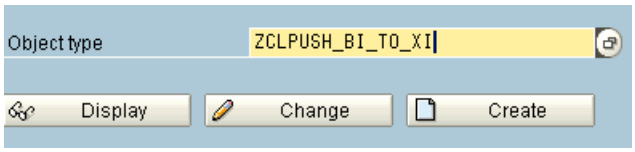
```
METHOD zii_push_table_msg_int_in~execute_asynchronous.

 DATA:
  l_logid    TYPE rspc_logid,
  l_type     TYPE rspc_type,
  l_variant  TYPE rspc_variant,
  l_instance TYPE rspc_instance,
  l_state    TYPE rspc_state,
  l_process_id type ZPUSH_TABLE_RETURN_PROCESS_ID.

 IF input-push_table_msg_type_ret-status = 'OK'.
  l_state = 'G'.
 ELSE.
  l_state = 'R'.
 ENDIF.
```

```
loop at input-push_table_msg_type_ret-process_id into
l_process_id.

 l_logid    = l_process_id-log_id.
 l_type     = l_process_id-type.
 l_variant  = l_process_id-variant.
 l_instance = l_process_id-instance.

 CALL FUNCTION 'RSPC_PROCESS_FINISH'
  EXPORTING
    i_logid          = l_logid
    i_type           = l_type
    i_variant        = l_variant
    i_instance       = l_instance
    i_state          = l_state.
*   i_dump_at_error     = 'X'.

endloop.

endmethod.
```

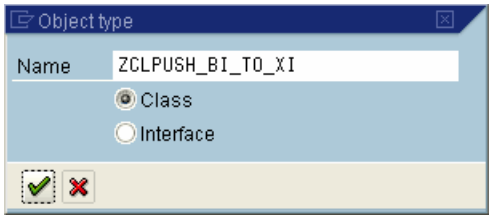
10. Activate the entire class.

3.3 Preparations for custom process type – class definition and implementation

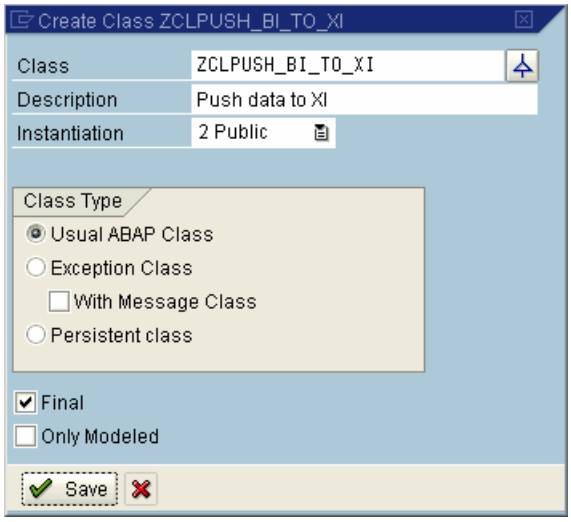
1. In the BW system start transaction SE24, enter the class name ZCLPUSH_BI_TO_XI and press *create*.



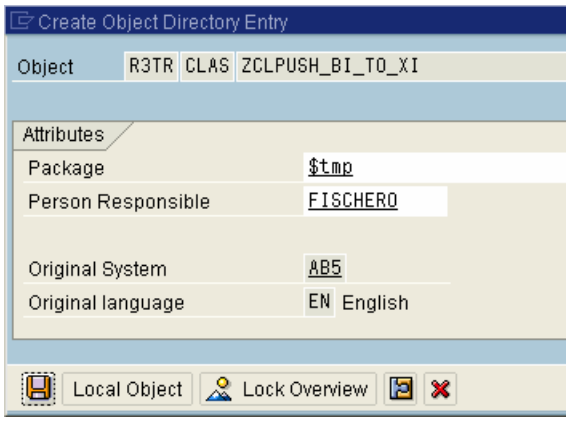
2. Choose object type *class*.



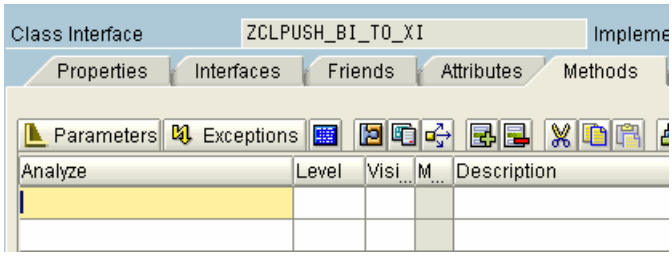
3. Complete the entries as shown and press save.



4. Choose a package.

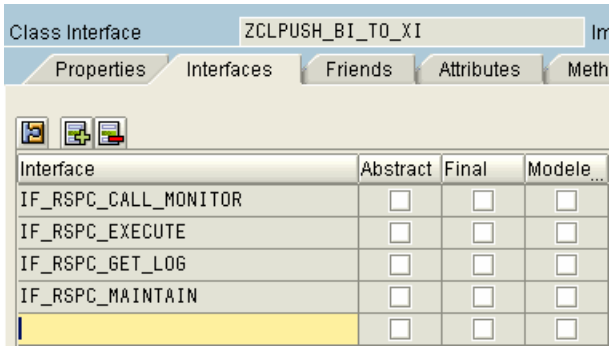


5. Choose the ***interfaces*** tab.

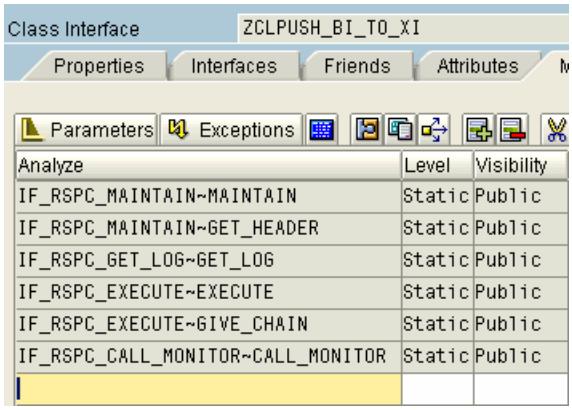


6. Add the interfaces
IF_RSPC_CALL_MONITOR
IF_RSPC_EXECUTE
IF_RSPC_GET_LOG
IF_RSPC_MAINTAIN
to the class.

These are interfaces are used to call methods of the process chains framework.



7. Now we have to implement these interfaces. Choose the ***methods*** tab and invoke the ABAP editor by double clicking in each method. Add the coding listed in chapter 5 Custom process type – source code for class zclpush_bi_to_xi.

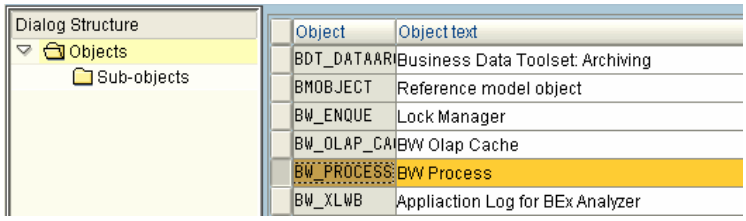


8. Activate the complete class.

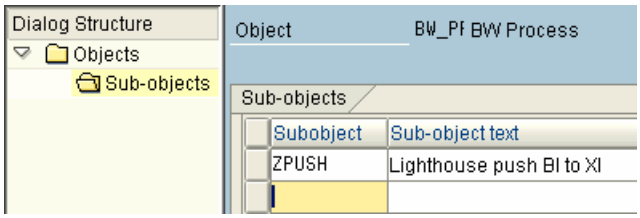
As the implementation uses the standard application log to store information, we have to create a sub object *zpush*.

9. Start transaction SLG0, accept the warning saying that the table is cross client.

Select the object *BW_PROCESS*.

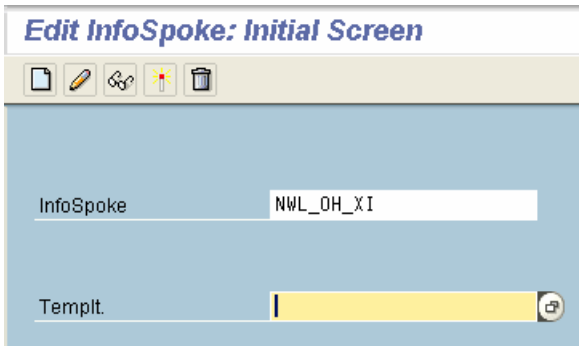


10. Double click on the node sub-objects and choose new entry. Create the new sub-object *ZPUSH* and then save.



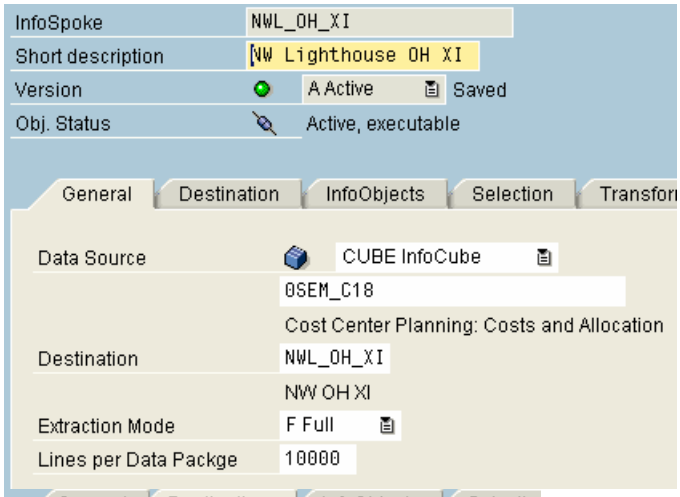
3.4 Create InfoSpoke

1. In the BW system, start transaction RSBO, enter the name NWL_OH_XI and press *create*.

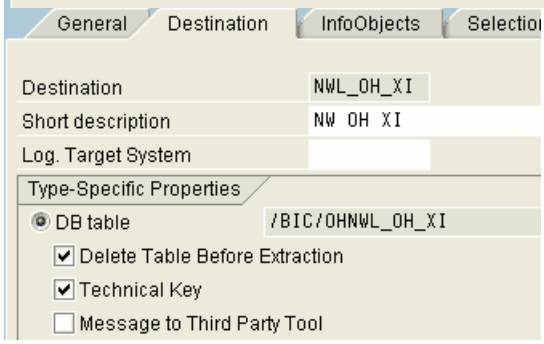


2. Complete the fields as shown.

In this example, we used the delivered InfoCube 0SEM_C18 as the DataSource. Note: this technique could be utilized for any InfoCube or ODS object, any object that can comprise an InfoSpoke.



3. In the example in this paper, we stage the data in a database table. Note that subsequent code refers to this specific database table name; any deviation in naming from the example must be adjusted in the code (for reading the DB table named */BIC/OHNWL_OH_XI*).



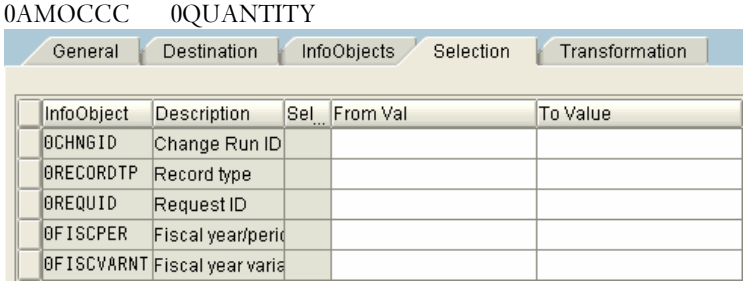
4. Go to tab *InfoObjects*.

The InfoObjects used in our example as listed here. The structure configuration for the dataset in XI must match the structural definition of the Open Hub InfoSpoke staging object

0COORDER 0COSTCENTER
0COSTELMNT 0CO_AREA
0ACTTYPE 0ACT_UNIT
0CURRENCY 0FISCPER3
0FISCVARNT 0FISCYEAR
0OBJ_CURR 0PART_ACTTY
0PART_CCTR 0UNIT
0VERSION 0VTYPE

0ACT_CAPACT 0AMOCAC
0AMOCCC 0QUANTITY

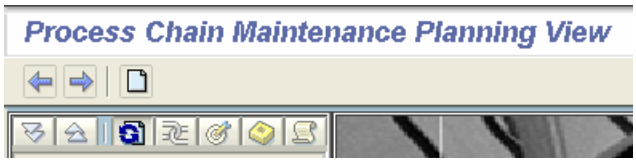
5. Depending on your sample data and data volume you can define some restrictions. We will continue without any restriction.



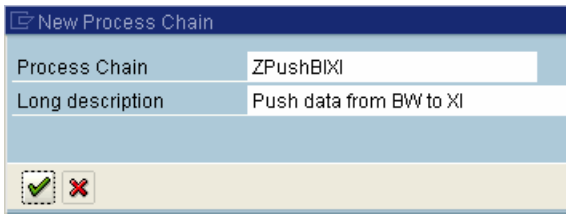
6. Save and activate the InfoSpoke.

3.5 Create custom process type and process chain

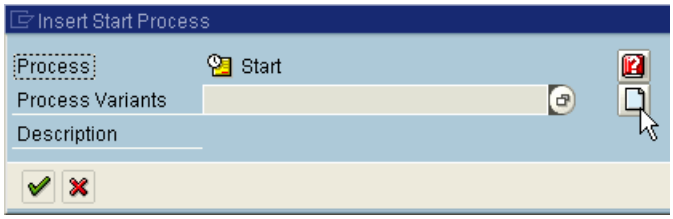
1. Start transaction RSPC and create a new process chain.



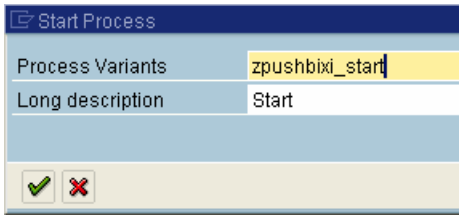
2. The name of the process chain is zpushbixi. Complete the fields as shown.



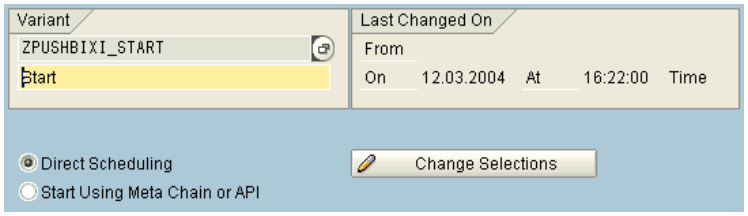
3. Create a new variant for the start process.



4. Specify the name of the process variant.

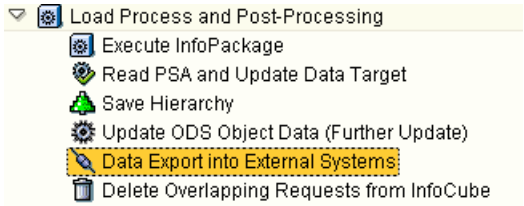


5. Set the scheduling option to immediate executing (for testing reasons) by pressing the button Change Selections.

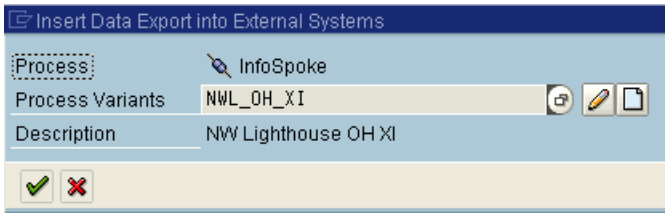


6. Save the settings and return to the process chain maintenance.

7. Expand the node Load Process and Post-Processing. Choose the type Data Export into External Systems.

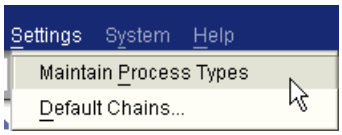


8. The variants reflect the name of existing InfoSpokes. Choose the variant NWL_OH_XI we have created previously.



9. The next step is to create a new custom process type, which reads the InfoSpoke staged dataset and transfers it to XI via proxy.

10. Choose from the main menu: Settings and *Maintain Process Types*.

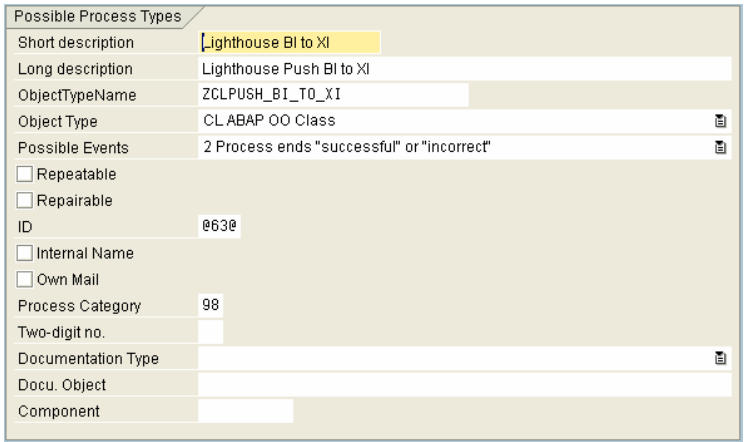


11. Press button *New Entries*.

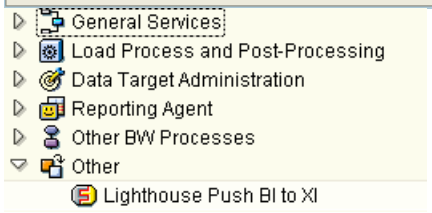
12. The technical name of new process type is ZPUSH.

Complete the entries as shown:
ObjectTypeName: ZCLPUSH_BI_TO_XI
ObjectType: CL ABAP 00 Class
PossibleEvents: Successful or incorrect

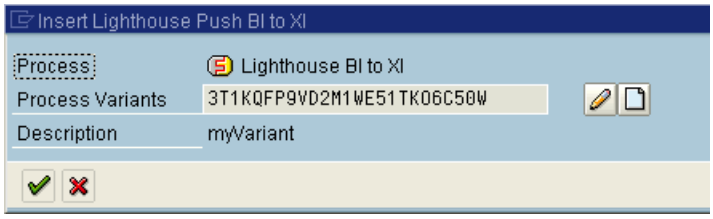
ID: @63@
Process Category: 98



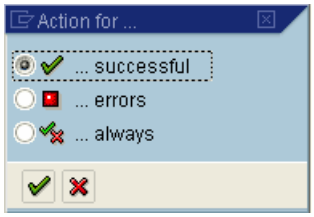
13. After saving the changes, you will see you new custom process type under *other.* Note that the icon was selected above as "ID" *@63@.*



14. Add the process type to the process chain, create a new variant and then continue.

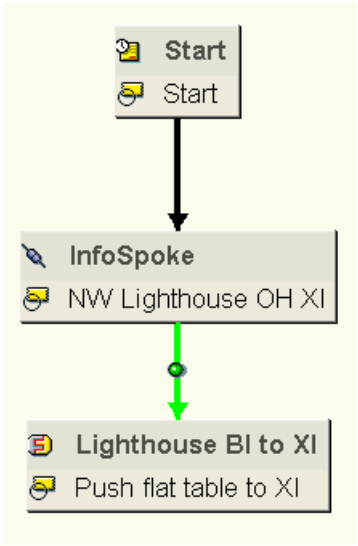


15. Draw the link between the InfoSpoke and the Custom process type. When releasing the mouse button the dialog shown on the right will come up.



The process type should only be executed if the InfoSpoke process ended successfully.

16. The resulting process chain should closely resemble the one here in the illustration.

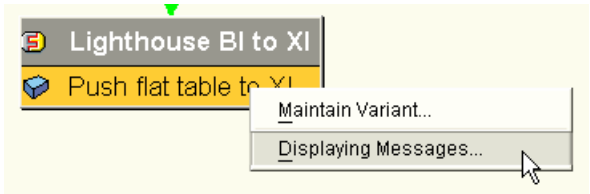


Don't forget to save and activate the process chain.

3.6 Testing the communication BW-XI

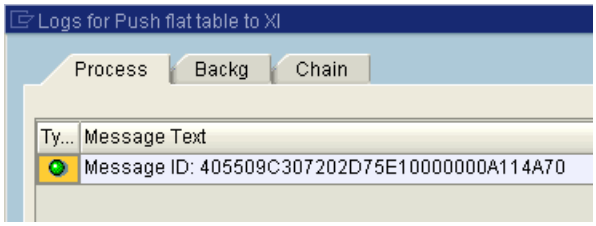
1. After successful execution of the process chain, we should see status information in the XI monitors.

2. Navigate to the process chain log view and choose your execution.

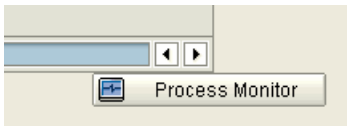


Choose *display messages* from the context menu of our newly created process type.

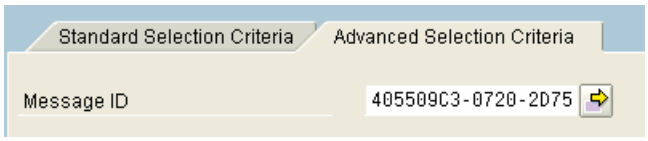
3. Choose the tab **Process**. The ID shown here is a XI message ID we can use to select the message created.



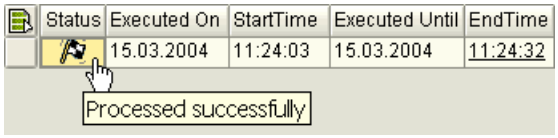
4. Press the button *Process Monitor* to start the transaction SXMB_MONI (on sender system side).



5. The message ID on tab ***Advanced Selection Criteria*** is automatically filled with the message ID.

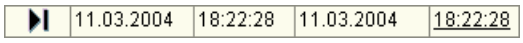


6. Everything is fine when you see a result line the black white flag shown.



7. In the XI system, the status of the message (where the dataset is contained) is different. This is because no receiver systems have been configured in XI yet.

No receiver could be determined



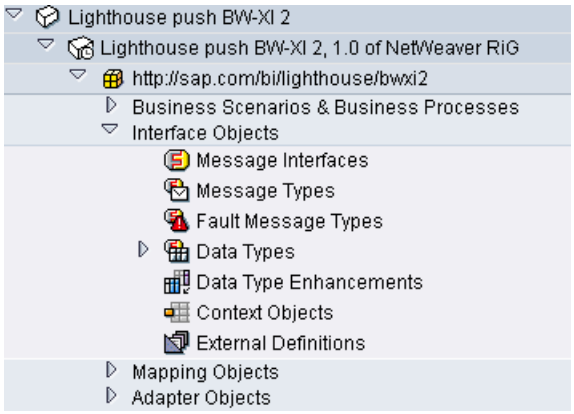
4 Setup BI – XI – receiver scenarios

4.1 Technical Adapter – Flat File

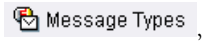
4.1.1 Create Interface Objects and Mapping for Flat File Receiver

1. Launch the overview of ***Exchange Infrastructure Tools*** (transaction SXMB_IFR on your XI system) and start the Integration Repository.

2. Select the software component used for BW XI. In our case this is the ***Lighthouse push BW-XI 2***.

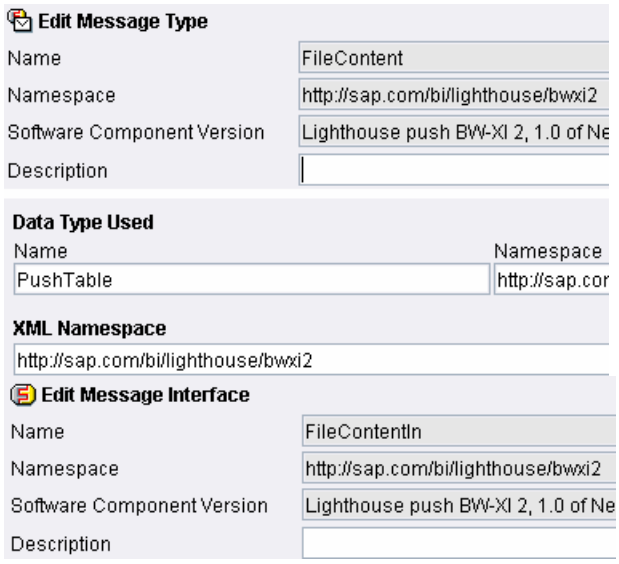


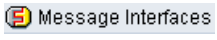
3. Based on the data type PushTable we now create the message type FileContent.

   Right click on the node 

, specify the name and choose create.

   Complete the fields as shown.



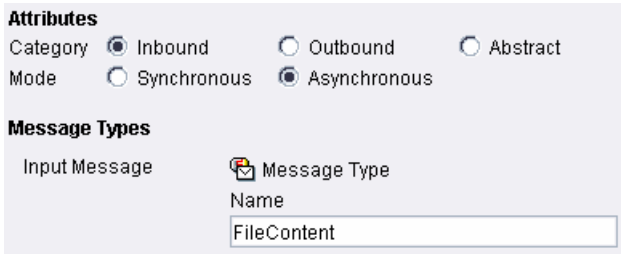
4. Right click on the node 

and create a new Message Interface with the name ***FileContentIn***.

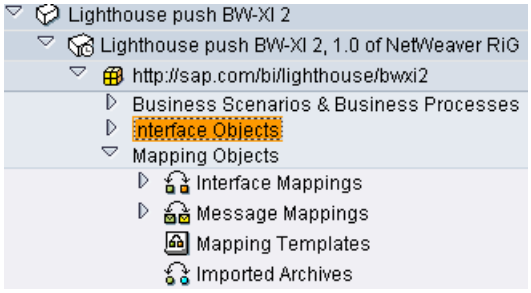
   Complete the fields as shown.

5. Set the category to Inbound and the mode to asynchronous.

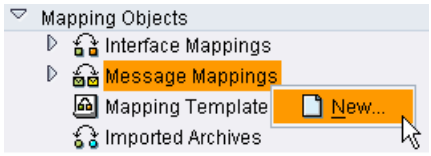
   Choose the message type ***FileContent.***



6. Now we will establish the mapping rules between the interfaces offered by the sender system (BW) and the by the receiver system (Lighthouse Receiver).



7. Right click on the Message Mapping and choose new.



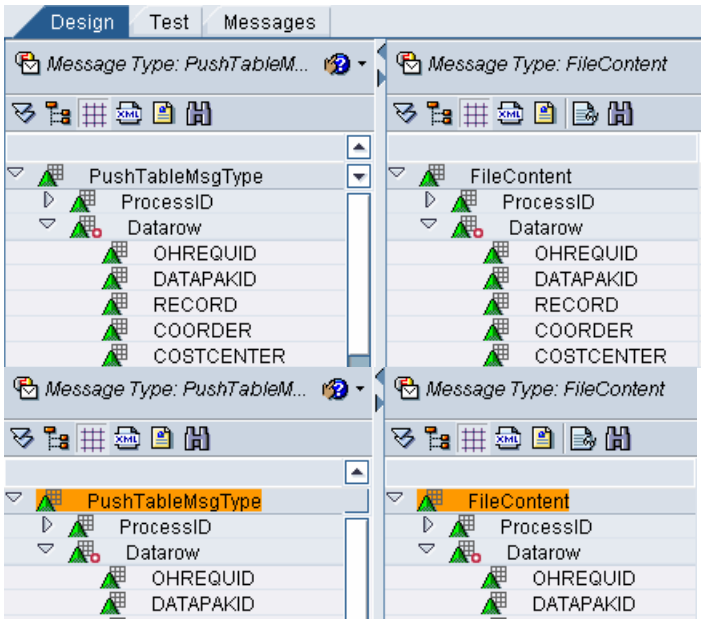
8. The name of the message mapping is ***PushTabletoFileContent***.

9. Add the message type ***PushTableMsgType*** on the left hand side and ***FileContent*** on the right hand side.

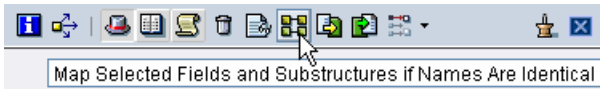
You can add by pressing or by dragging them from the tree.

10. Now will define the real mapping.

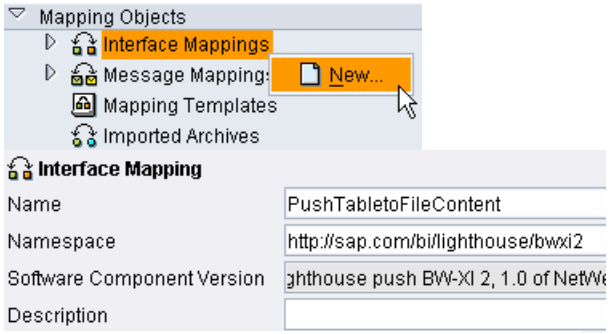
Select the root nodes PushTableMsgType and FileContent.



11. Press the button to map fields with identical names.

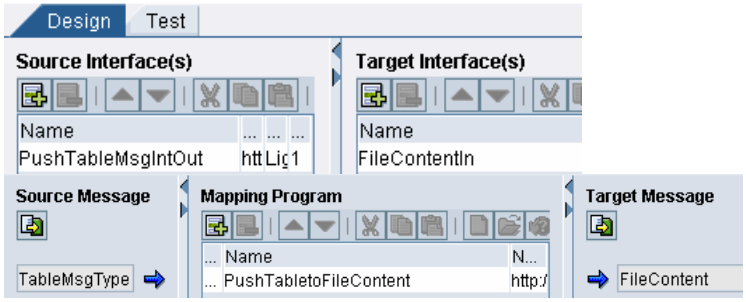


12. Right click on the Interface Mappings, choose new and specify the name to *PushTabletoFileContent*.



13. Choose *PushTableMsgIntOut* as source interface and *FileContentIn* as target.

Add the mapping program *PushTabletoFileContent*.



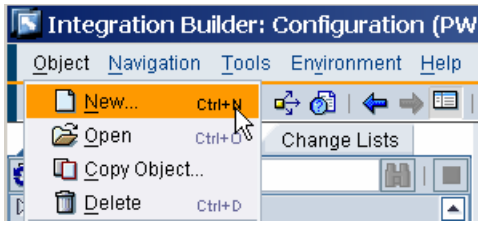
14. Choose the tab Change Lists and activate the created objects.

4.1.2 Create Directory Scenario for Flat File Receiver

The customizing created on XI side is generic so far. There are no links to real system available so far. This will be done now by creating a directory scenario.

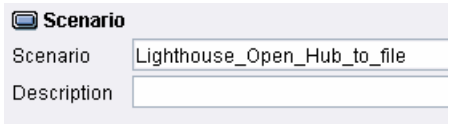
1. Launch the overview of *Exchange Infrastructure Tools* (transaction SXMB_IFR on your XI system) and start the Integration Directory.

2. In order to create a scenario choose Object from the main menu and select New.

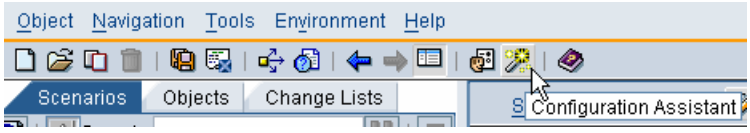


3. Choose ***Scenario*** and specify the name to Lighthouse_Open_Hub_to_File.

Save the scenario.



4. Invoke the Configuration Assistant by pressing .



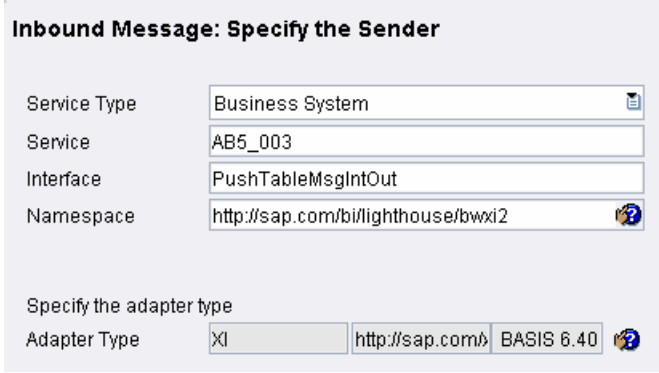
5. Now we create the configuration to send a message from BW to our receiver system lighthouse_Receiver2.

Choose internal communication.

6. The incoming Message is initiated by BW system.

The business system that is linked to our BW system has the name AB5_005.

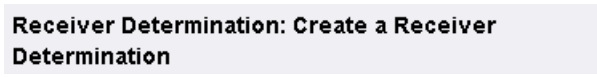
Service: AB5_003 (Business System defined in SLD)
Interface: PushTableMsgIntOut



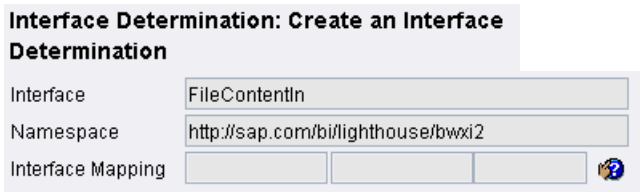
7. The receiver is our system lighthouseReceiver2 and will use the adapter type XI.

8. Just continue.

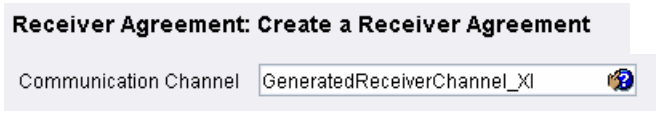
9. Just continue.



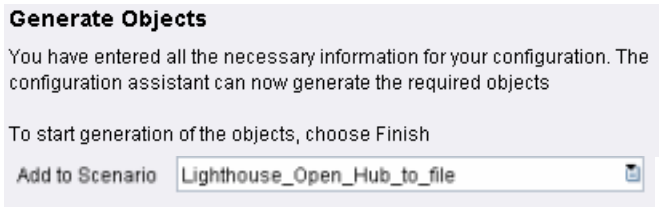
10. Select the interface mapping *PushTabletoFileContent.*



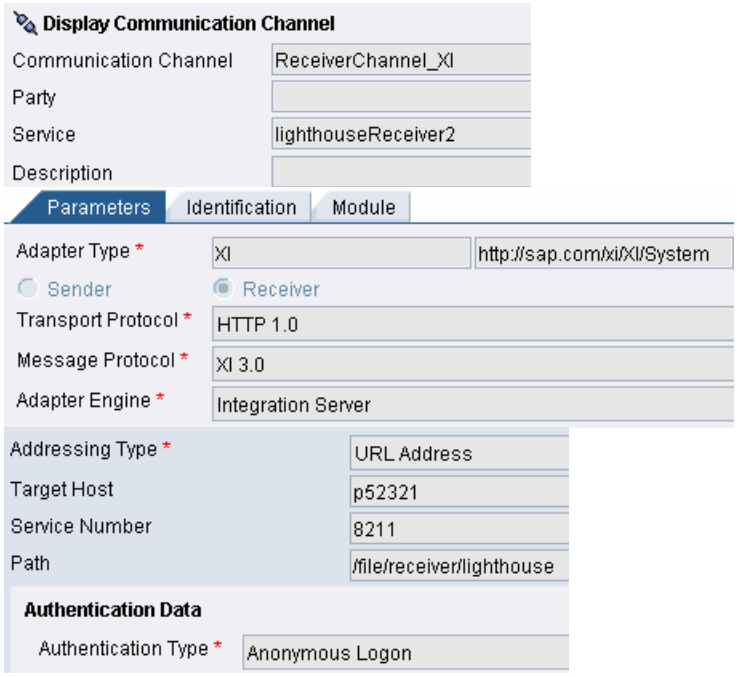
11. Select the standard XI communication channel.



12. Now the wizard will generate all required objects.



13. Customize the communication channel for lighthouseReceiver2.



14. Choose the tab Change Lists and activate the created objects.

4.1.3 Set up the flat file receiver system

1. Install the technical adapter on your receiver system.

The name is TechnicalAdapter.sda.
It is part of the NetWeaver 04' installation CD/download.
You can find Details in the current documentation \Adapter Engine\Installation

2. Add the servlet.jar into the folder *|tech_adapter*

3. Start the technical adapter.

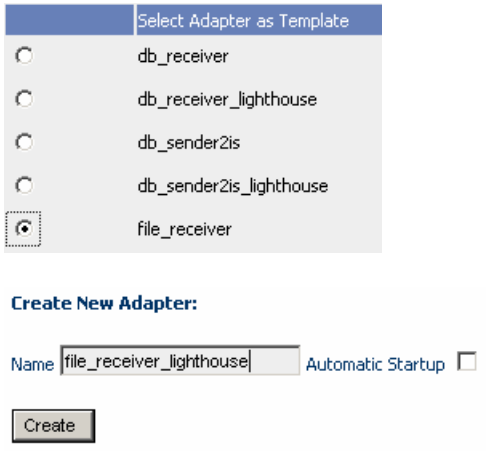
Execute the program run_adaper.bat that can be found in the directory J2SE\tech_adapter.

4. Start the adapter customizing.

Use URL http://localhost:8200/adapter.html to enter adapter customizing.
Username: SAP, Password: Init

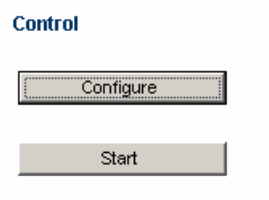
5. Click on the link 'Add new adapter'.

6. Specify the name of the adapter ***file_receiver_lighthouse*** based on the template ***file_receiver***.

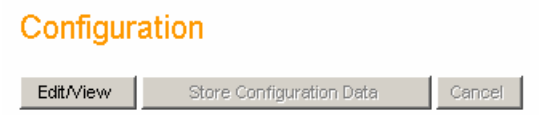


7. You will see now the new adapter created. Click on the name to navigate to the detail setup screen.

8. Press button configure. You see now a text editor on the right hand side.



9. Press button Edit/View, add the settings as shown and store the configuration data.



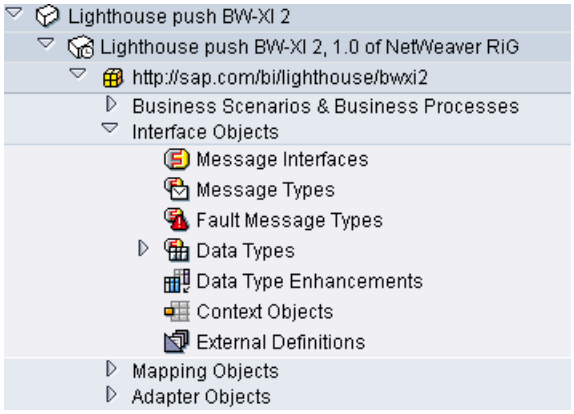

```
## File adapter java class
classname=com.sap.aii.messaging.adapter.ModuleXMB2File
version=30
mode=XMB2FILE
##Adress for XMB endpoint
XI.httpPort=8211
XI.httpService=/file/receiver/lighthouse
##File Adapter specific parameters
file.createDir=1
file.targetDir=c:/DataReceiver
file.targetFilename=lighthouse_output.xml
#file.writeMode=append
file.writeMode=addCounter
file.counterMode=immediately
file.counterSeparator=_
file.counterFormat=00000
file.counterStep=1
```

10. Restart the adapters to activate the changes.

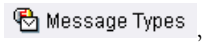
4.1.4 Create Interface Objects and Mapping for Flat File Sender

1. Launch the overview of ***Exchange Infrastructure Tools*** (transaction SXMB_IFR on your XI system) and start the Integration Repository.

2. Select the software component used for BW XI. In our case this is the ***Lighthouse push BW-XI 2***.

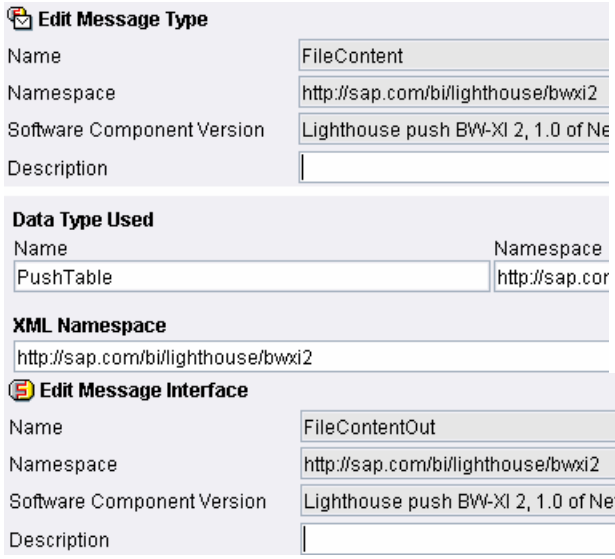


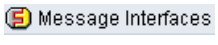
3. Based on the data type PushTable we now create the message type FileContent.

Right click on the node 

, specify the name and choose create.

Complete the fields as shown.



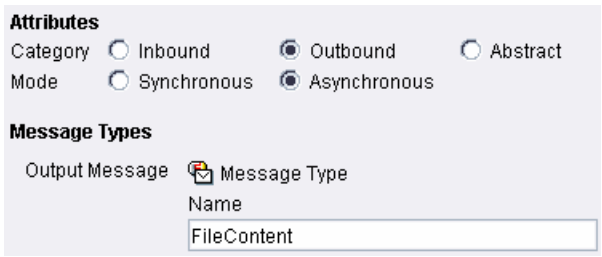
4. Right click on the node 

and create a new Message Interface with the name ***FileContentOut***.

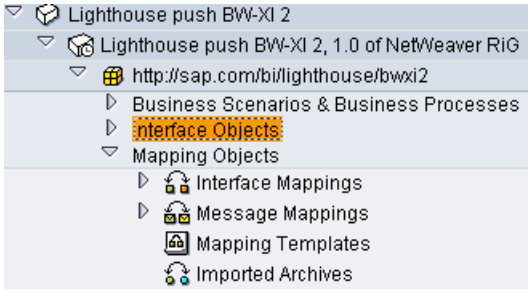
Complete the fields as shown.

5. Set the category to Outbound and the mode to asynchronous.

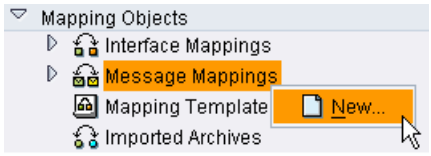
Choose the message type ***FileContent.***



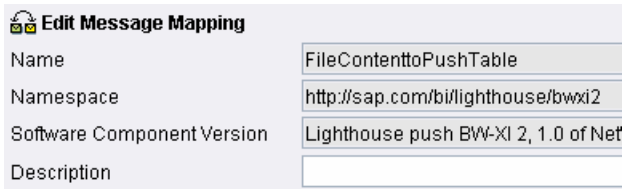
6. Now we will establish the mapping rules between the interfaces offered by the sender system (BW) and the by the receiver system (Lighthouse Receiver).



7. Right click on the Message Mapping and choose new.



8. The name of the message mapping is ***FileContenttoPushTable***.

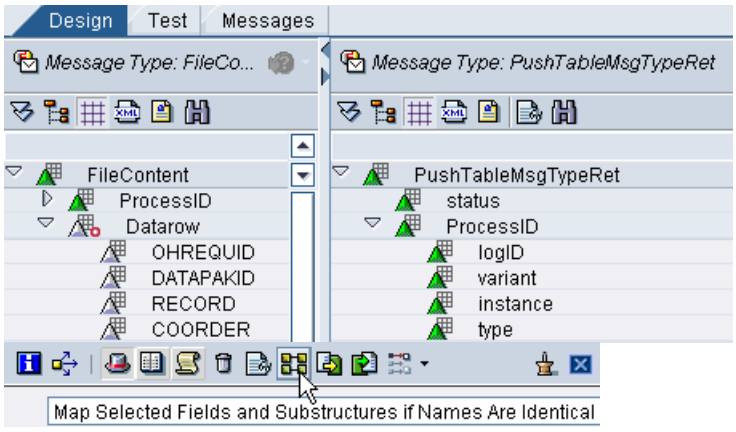


9. Choose the message type ***FileContent*** on the left hand side and ***PushTableMsgTypeRet*** on the right hand side.

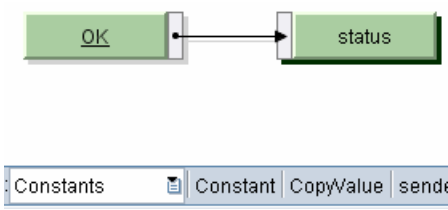
Map the fields with identical names as described above.
You can add by pressing or by dragging them from the tree.

10. Now will define the real mapping.

Select the root nodes FileContent and PushTableMsgTypeRet and press the button to map fields with identical names.



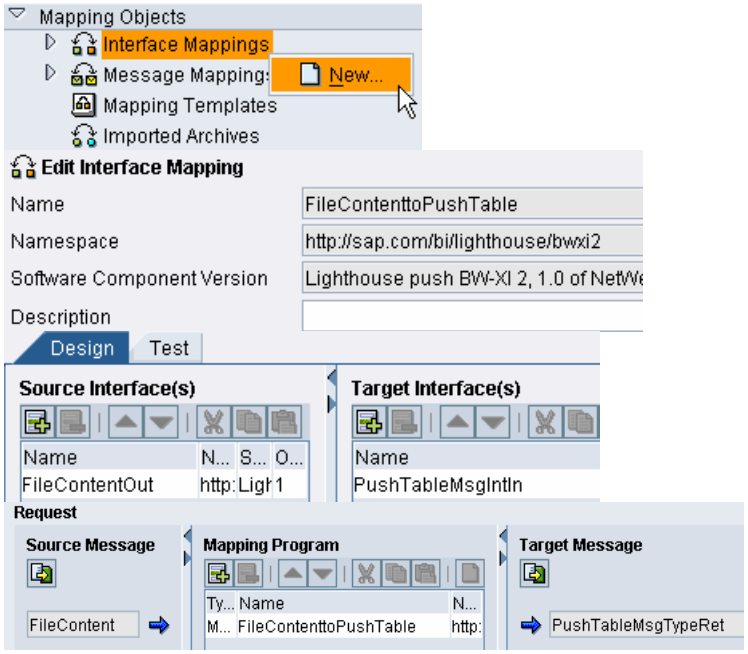
11. Double click on the status node. Choose constants from the drop down box, set the constant to the value OK and link it to the status box.



12. Right click on the Interface Mappings, choose new and specify the name to *FileContenttoPushTable*.

13. Choose *FileContentOut* as source interface and *PushTableMsgIntIn* as target.

Add the mapping program *FileContenttoPushTable*.



14. Choose the tab Change Lists and activate the created objects.

4.1.5 Create Directory Scenario for Flat File Sender

Now we will create the scenario sending the file via XI to the BW system.
As this file contains context information of the BW push process, this is only needed if you need the call back.
In general we go the same steps as shown in chapter Create Directory Scenario for Flat File Receiver.
Hence only the entries for the wizard steps are shown.

1. Complete the information for the sender service.

   Sender Service: lighthouseReceiver2
   Sender Interface: FileContentOut
   Namespace: http://sap.com/bi/lighthouse/bwxi2

2. Complete the information for the receiver service.

   Receiver Service: AB5_003
   Receiver Interface: PushTableMsgIntIn
   Namespace: http://sap.com/bi/lighthouse/bwxi2

3. Specify the message mapping.

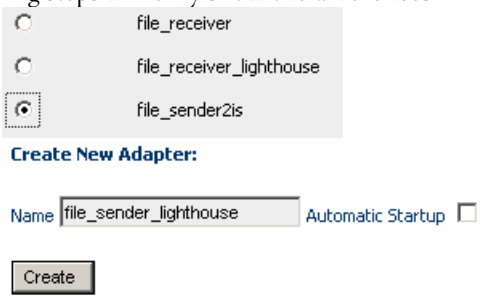
   The name of the message mapping is ***FileContenttoPushTable***.

4. Choose the tab Change Lists and activate the created objects.

4.1.6 Set up the flat file sender system

In order to set up the flat file sender system we have to do basically the things as described above (section Set up the flat file receiver system). Hence the following steps will only show the differences

1. Create a new adapter with the name ***file_sender2is_lighthouse*** that is based on the template adapter ***file_sender2is***.



2. Adapt the configuration as shown.

Please keep in mind, that you have to adjust some entries according to your system environment:
- XI.TargetURL
- XI.InterfaceNamespace

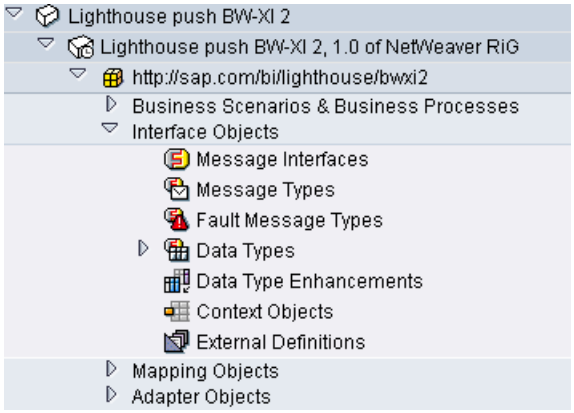
```
## file adapter java class
classname=com.sap.aii.messaging.adapter.ModuleFile2XMB
version=30
mode=FILE2XMB
## Integration Engine address and document settings (example, see docu)
XI.TargetURL=http://pwdf0321:50031/sap/xi/engine?type=entry
XI.User=XIAPPLUSER
XI.Password=XIPASS
XI.Client=100
XI.QualityOfService=EO
XI.SenderParty=
XI.SenderService=lighthouseReceiver2
XI.ReceiverParty=
XI.ReceiverService=
XI.Interface=FileContentOut
XI.InterfaceNamespace=http://sap.com/bi/lighthouse/bwxi2
##File Adapter specific parameters (example, see docu)
file.type=BIN
file.sourceDir=c:/DataReceiver
file.sourceFilename=*.*
file.processingMode=archiveWithTimeStamp
file.archiveDir=c:/DataReceiver_archive
file.pollInterval=200
```

4.2 Technical Adapter – Data Base

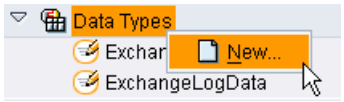
4.2.1 Create Interface Objects and Mapping for DB Receiver

1. Launch the overview of ***Exchange Infrastructure Tools*** (transaction SXMB_IFR on your XI system) and start the Integration Repository.

2. Select the software component used for BW XI. In our case this is the ***Lighthouse push BW-XI 2***.



3. Create a data type by right clicking on the node ***Data Types***.



4. Fill the input fields as shown and press the create button.

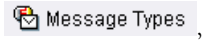
5. Now we create the data structure ***PushDB*** that contains the transaction data delivered by BW and the SQL statement to perform the update in the database.

The highlighted line contains the structure of the transaction data as shown in the second screen shot.

Structure	Category	Type	Occurrence
PushDB	Complex Type		
Data	Element		1
PrimCostData	Element		1
action	Attribute	xsd:string	required
access	Element	PushDataRow	1..unbounded
ProcessID	Element		1
PrimCostSenderContext	Element		1
action	Attribute	xsd:string	required
access	Element		1
processed	Element	xsd:string	1
logID	Element	xsd:string	1
variant	Element	xsd:string	1
instance	Element	xsd:string	1
type	Element	xsd:string	1

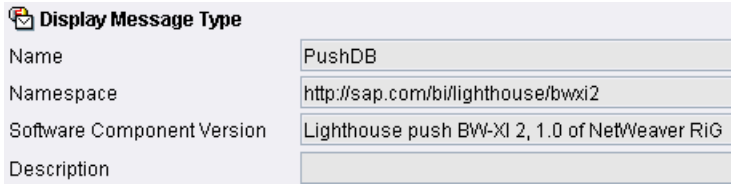
access	Element	PushDataRow	1..unbounded
OHREQUID	Element	xsd:string	1
DATAPAKID	Element	xsd:string	1
RECORD	Element	xsd:string	1
COORDER	Element	xsd:string	1
COSTCENTER	Element	xsd:string	1
COSTELMNT	Element	xsd:string	1
CO_AREA	Element	xsd:string	1
ACTTYPE	Element	xsd:string	1
ACT_UNIT	Element	xsd:string	1
CURRENCY	Element	xsd:string	1
FISCPER3	Element	xsd:string	1
FISCVARNT	Element	xsd:string	1
FISCYEAR	Element	xsd:string	1
OBJ_CURR	Element	xsd:string	1
PART_ACTTY	Element	xsd:string	1
PART_CCTR	Element	xsd:string	1
UNIT	Element	xsd:string	1
VERSION	Element	xsd:string	1
VTYPE	Element	xsd:string	1
ACT_CAPACT	Element	xsd:string	1
AMOCAC	Element	xsd:string	1
AMOCCC	Element	xsd:string	1
QUANTITY	Element	xsd:string	1

6. Based on the data type PushDB we now create the message type PushDB.

Right click on the node 

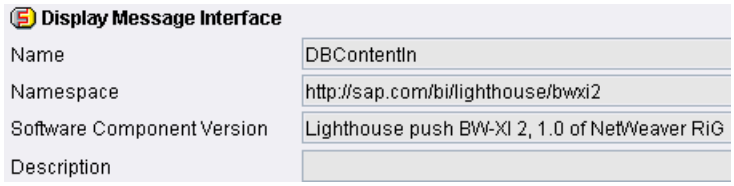
, specify the name and choose create.

Complete the fields as shown.



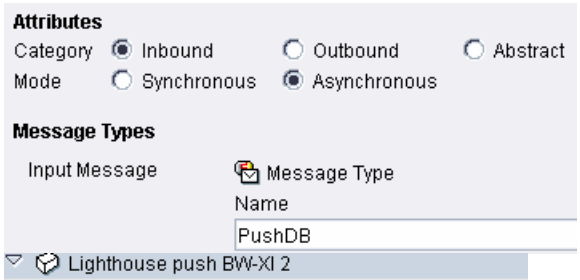
7. Right click on the node Message Interfaces and create a new Message Interface with the name ***DBContentIn***.

Complete the fields as shown.

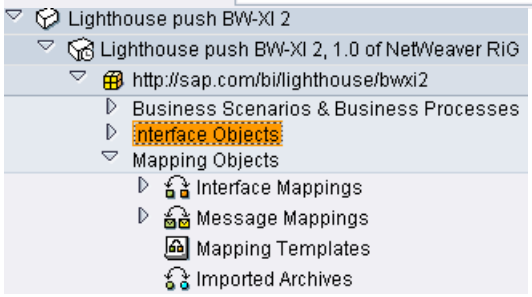


8. Assign the category to Inbound and the mode to asynchronous.

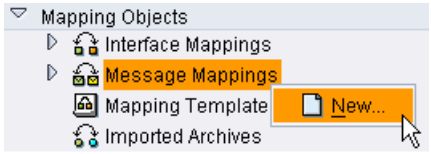
Choose the message type ***PushDB***.



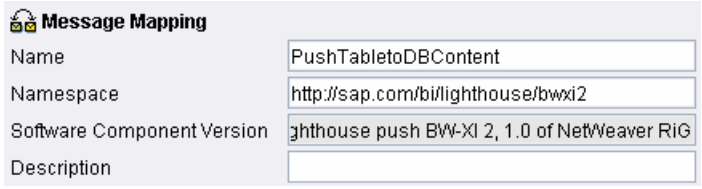
9. Now we will establish the mapping rules between the interfaces offered by the sender system (BW) and the by the receiver system (Lighthouse Receiver).



10. Right click on the Message Mapping and choose new.

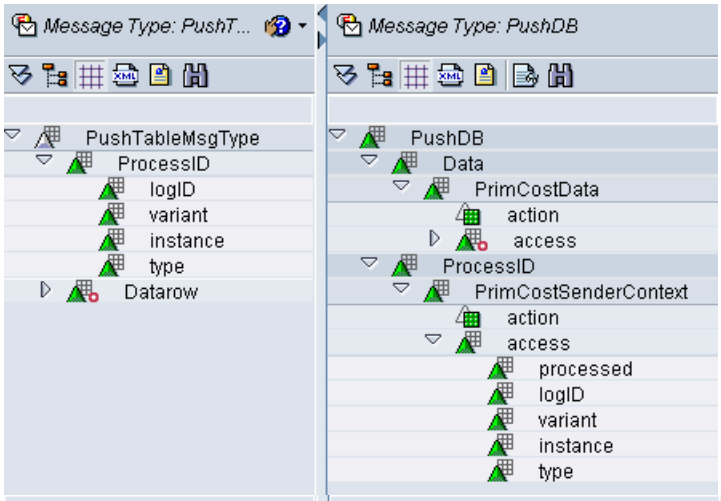


11. The name of the message mapping is ***PushTabletoDBContent***.



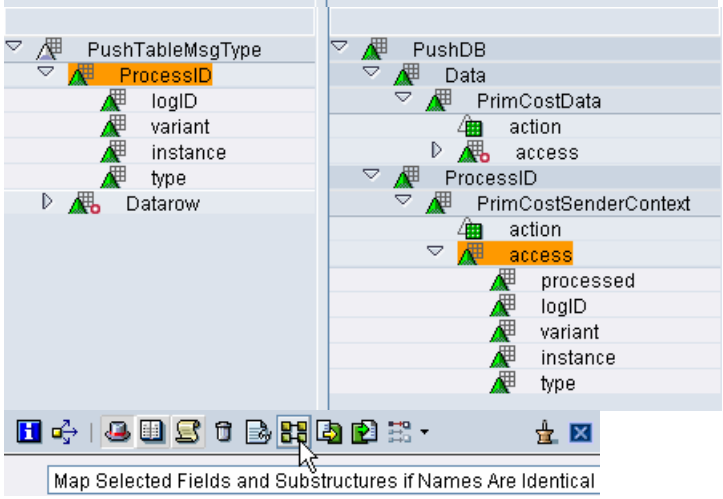
12. Add the message type ***PushTableMsgType*** on the left hand side and ***DBContent*** on the right hand side.

You can add by pressing or by dragging them from the tree.



13. Now will define the real mapping.

Select the nodes ***ProcessID*** and ***access***.



14. Press the button to map fields with identical names.

15. Now we have to specify some constant values.

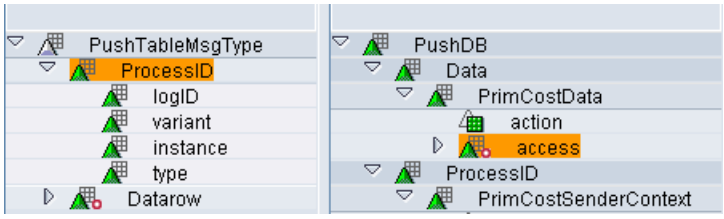
Double click on the ***action*** node (underneath the node PrimCostSenderContext).

Choose constants from the drop down box, set the constant to the value INSERT and link it to the @action box.

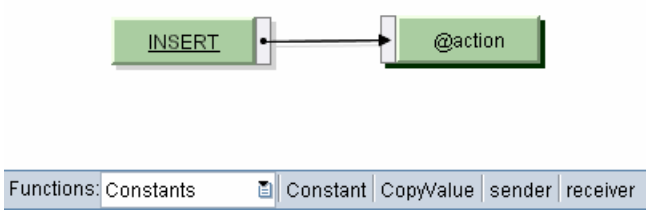
16. Double click on the ***processed*** node (underneath the node PrimCostSenderContext).

Choose constants from the drop down box, set the constant to the value N and link it to the processed box.

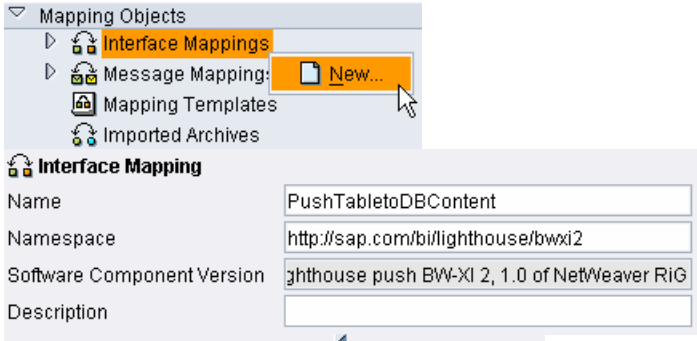
17. Select the nodes ***Datarow*** and ***access*** (underneath the node PrimCostData) and map fields with identical names.



18. Double click on the ***action*** node. Choose constants from the drop down box, set the constant to the value INSERT and link it to the @action box.

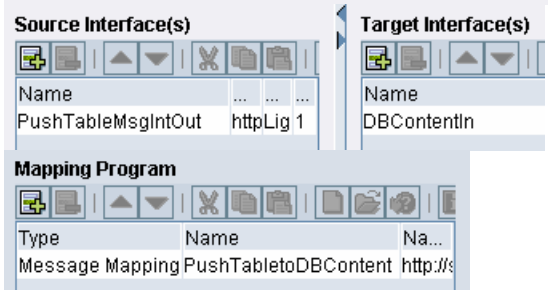


19. Right click on the Interface Mappings, choose new and specify the name to ***PushTabletoDBContent***.



20. Choose ***PushTableMsgIntOut*** as source interface and ***DBContentIn*** as target.

Add the mapping program ***PushTabletoDBContent***.



21. Choose the tab Change Lists and activate the created objects.

4.2.2 Create Directory Scenario for DB Receiver

Now we will create the scenario sending the data via XI to a database.
In general we go the same steps as shown in chapter **Create Directory Scenario for Flat File Receiver**. Hence only the entries for the wizard steps are shown.

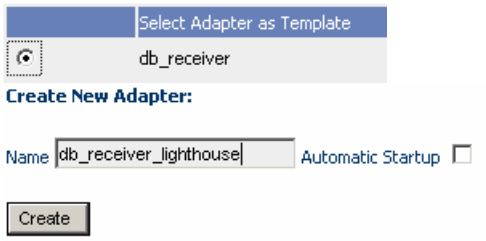
1.	Complete the information for the sender service.	Sender Service: AB5_003 Sender Interface: PushTableMsgIntOut Namespace: http://sap.com/bi/lighthouse/bwxi2
2.	Complete the information for the receiver service.	Receiver Service: lighthouseReceiver2 Receiver Interface: DBContentIn Namespace: http://sap.com/bi/lighthouse/bwxi2
3.	Specify the message mapping.	The name of the message mapping is ***PushTabletoDBContent***.
4.	Choose the tab Change Lists and activate the created objects.	

4.2.3 Set up the DB Receiver System

In order to set up the DB receiver system we have to do basically the things as described in section **Set up the flat file receiver system**. Hence the following steps will only show the differences.

1.	Install the JDBC driver for your database on the same system where the technical adapter is running.	For MS SQL you can download the JDBC driver here: http://www.microsoft.com/downloads/details.aspx?FamilyID=9f1874b6-f8e1-4bd6-947c-0fc5bf05bf71&DisplayLang=en

2. Launch the technical adapter and create a new adapter with the name ***db_receiver_lighthouse*** that is based on the template adapter ***db_receiver***.



3. Adapt the configuration as shown.

Please keep in mind, that you normally have to adjust some entries according to your system environment:

- db.jdbcDriver:
this is the class name for the standard Microsoft JDBC driver for MS SQL data base

- db.connectionURL:
Specify the host, user, password and database name.

- db.table
It is assumed that we have a data base table with the name PrimCostData containing the same columns as mentioned in the definition of the data type pushtable/datarow.

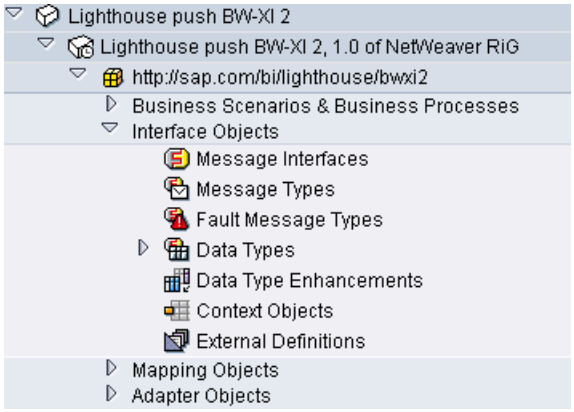
```
## jdbc adapter java class
classname=com.sap.aii.messaging.adapter.ModuleXMB2DB
version=30
mode=XMB2DB_XML
##Adress for XMB endpoint
XI.httpPort=8221
XI.httpService=/db/receiver/lighthouse
##DB Adapter specific parameters
db.jdbcDriver=com.microsoft.jdbc.sqlserver.SQLServerDriver
db.connectionURL=jdbc:microsoft:sqlserver://<hostname>;user=
<user>;password=<pass>;DatabaseName=<name of the data
base>;
db.table=PrimCostData
```

4. Start the new adapter.

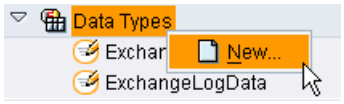
4.2.4 Create Interface Objects and Mapping for DB Sender

1. Launch the overview of ***Exchange Infrastructure Tools*** (transaction SXMB_IFR on your XI system) and start the Integration Repository.

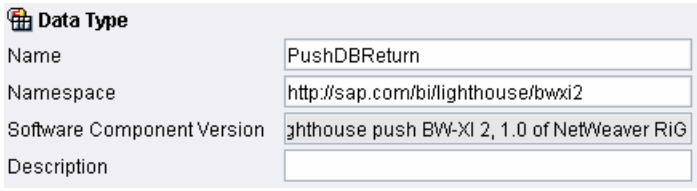
2. Select the software component used for BW XI. In our case this is the ***Lighthouse push BW-XI 2***.



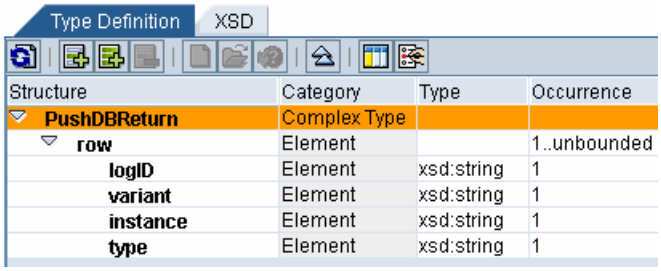
3. Create a data type by right clicking on the node ***Data Types***.



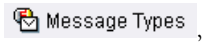
4. Fill the input fields as shown and press the create button.



5. Now we create the data structure ***PushDBReturn*** that contains the return information needed to adjust the BW process chain status.

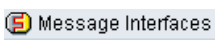


6. Based on the data type PushDBReturn we now create the message type PushDBReturn.

Right click on the node 

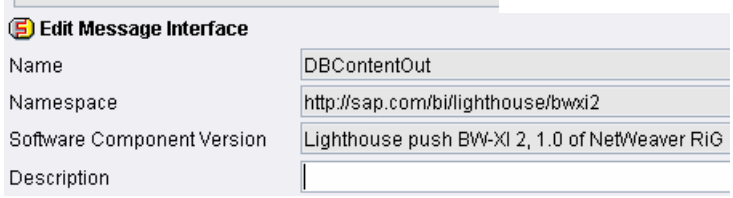
, specify the name and choose create.

Attention: the XML Namespace has to be empty!

7. Right click on the node 

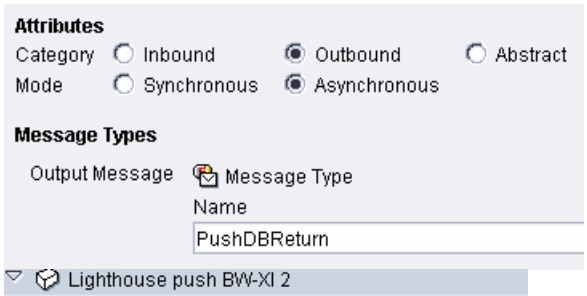
and create a new Message Interface with the name ***DBContentOut***.

Complete the fields as shown.

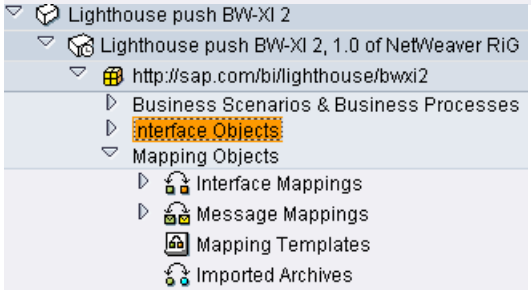


8. Assign the category to Outbound and the mode to asynchronous.

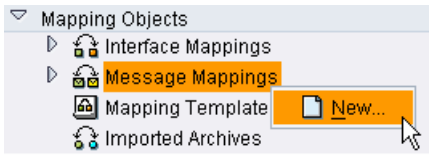
Choose the message type ***PushDBReturn***.



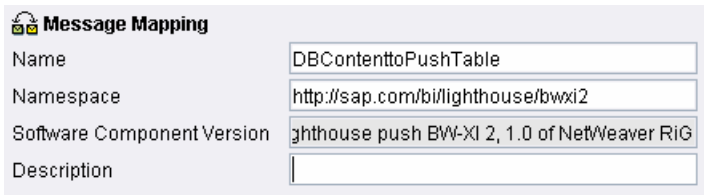
9. Now we will establish the mapping rules between the interfaces offered by the sender system (BW) and the by the receiver system (Lighthouse Receiver).



10. Right click on the Message Mapping and choose new.

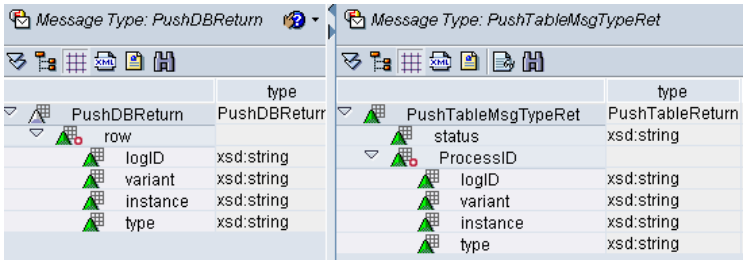


11. The name of the message mapping is ***DBContenttoPushTable.***



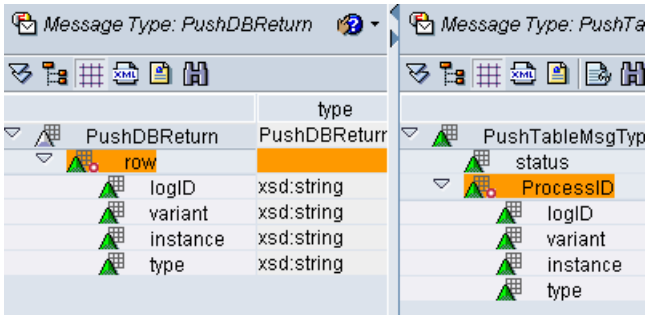
12. Add the message type ***PushDBReturn*** on the left hand side and ***PushTableMsgTypeRet*** on the right hand side.

You can add by pressing or by dragging them from the tree.

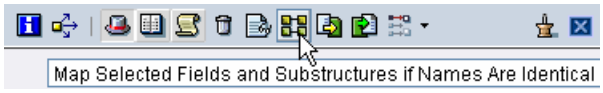


13. Now will define the real mapping.

Select the nodes ***ProcessID*** and ***access***.



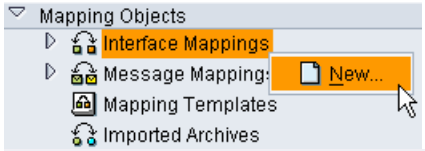
14. Press the button to map fields with identical names.



15. Double click on the ***status*** node.

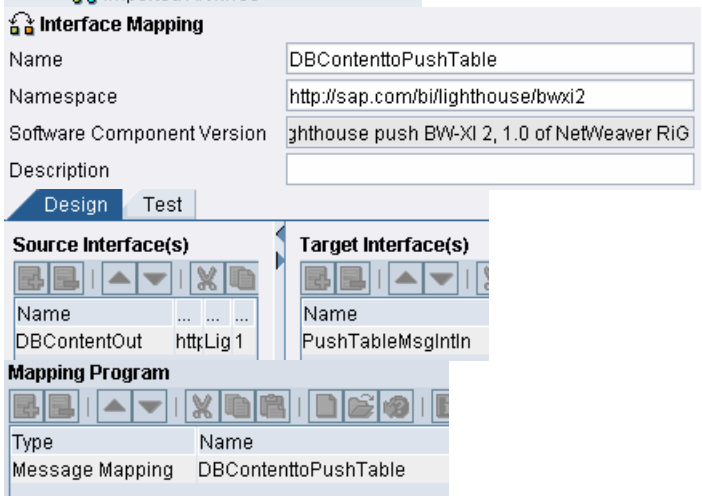
Choose constants from the drop down box, set the constant to the value OK and link it to the status box.

16. Right click on the Interface Mappings, choose new and specify the name to ***DBContenttoPushTable***.



17. Choose ***DBContentOut*** as source interface and ***PushTableMsgIntIn*** as target.

Add the mapping program ***DBContenttoPushTable***.



18. Choose the tab Change Lists and activate the created objects.

4.2.5 Create Directory Scenario for DB Sender

Now we will create the scenario sending the context information of the process chain via XI back to the BW system. This is only needed if you need the call back.
In general we go the same steps as shown in chapter **Create Directory Scenario for Flat File Receiver**. Hence only the entries for the wizard steps are shown.

1. Complete the information for the sender service.

   Sender Service: lighthouseReceiver2
   Sender Interface: DBContentOut
   Namespace: http://sap.com/bi/lighthouse/bwxi2

2. Complete the information for the receiver service.

   Receiver Service: AB5_003
   Receiver Interface: PushTableMsgIntIn
   Namespace: http://sap.com/bi/lighthouse/bwxi2

3. Specify the message mapping.

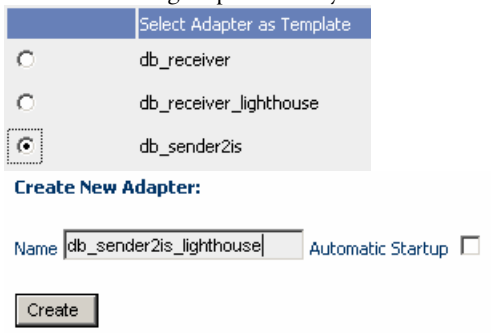
   The name of the message mapping is ***DBContenttoPushTable***.

4. Choose the tab Change Lists and activate the created objects.

4.2.6 Set up the DB Sender System

In order to set up the DB sender system we have to do basically the things as described in section **Set up the flat file receiver system**. Hence the following steps will only show the differences.

1. Create a new adapter with the name ***db_sender2is_lighthouse*** that is based on the template adapter ***db_sender2is***.



2. Adapt the configuration as shown.

   Please keep in mind, that you have to adjust some entries according to your system environment:

   - XI.TargetURL
   - XI.InterfaceNamespace

   - db.jdbcDriver and db.connectionURL
   Refer to the db_receiver_lighthouse for more details

   - db.processDBSQLStatement
   It is assumed that a data base table PrimCostSenderContext exist. It has to contain the columns processed, logID, variant, instance and type.

   - db.confirmDBSQLStatement
   Once the return message with the context information is read and transferred to XI, the adapter will set the status of processed from N to X. This is needed to avoid double sending.

```
## jdbc adapter java class
classname=com.sap.aii.messaging.adapter.ModuleDB2XMB
version=30
mode=DB2XMB
## Integration Engine address and document settings
XI.TargetURL=http://pwdf0321:50031/sap/xi/engine?type=entry
XI.User=XIAPPLUSER
XI.Password=XIPASS
XI.Client=100
XI.QualityOfService=EO
XI.SenderParty=
XI.SenderService=lighthouseReceiver2
XI.ReceiverParty=
XI.ReceiverService=
XI.Interface=DBContentOut
XI.InterfaceNamespace=http://sap.com/bi/lighthouse/bwxi2
##DB Adapter specific parameters
db.jdbcDriver=com.microsoft.jdbc.sqlserver.SQLServerDriver
db.connectionURL=jdbc:microsoft:sqlserver://<host>;user=<user
>;password=<pass>;DatabaseName=<data base name>;
db.processDBSQLStatement=select logID, variant, instance, type
from PrimCostSenderContext where processed<>'X'
db.confirmDBSQLStatement=update PrimCostSenderContext SET
processed = 'X'
db.pollInterval=600
db.documentName=PushDBReturn
```

3. Start the new adapter.

5 Appendix

5.1 Custom process type – source code for class zclpush_bi_to_xi

The class implements the following interfaces:
IF_RSPC_CALL_MONITOR
IF_RSPC_EXECUTE
IF_RSPC_GET_LOG
IF_RSPC_MAINTAIN

5.1.1 IF_RSPC_MAINTAIN~MAINTAIN

METHOD if_rspc_maintain~maintain .

```
DATA: l_id TYPE sysuuid_25.
```

```
 IF i_variant IS INITIAL.
  CALL FUNCTION 'RSSM_UNIQUE_ID'
   IMPORTING
    e_uni_idc25 = l_id.

  e_variant = l_id.
  e_variant_text = 'myVariant'.
 ENDIF.

ENDMETHOD.
```

5.1.2 IF_RSPC_MAINTAIN~GET_HEADER

```
method IF_RSPC_MAINTAIN~GET_HEADER .

 e_variant_text = 'Push flat table to XI'.

endmethod.
```

5.1.3 IF_RSPC_GET_LOG~GET_LOG

```
method IF_RSPC_GET_LOG~GET_LOG .

 CALL METHOD CL_RSPC_APPL_LOG=>DISPLAY
  EXPORTING
   I_TYPE         = 'ZPUSH'
   I_VARIANT      = i_variant
   I_INSTANCE     = i_instance
   I_NO_DISPLAY   = 'X'
  IMPORTING
   E_T_MSG        = e_t_messages
  EXCEPTIONS
   INTERNAL_FAILURE = 1
   others         = 2.

 IF SY-SUBRC <> 0.
  MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
          WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
 ENDIF.

endmethod.
```

5.1.4 IF_RSPC_EXECUTE~EXECUTE

```
METHOD if_rspc_execute~execute.

 TYPE-POOLS: rssm.

 CONSTANTS:
  c_type_name TYPE rspc_type VALUE 'ZPUSH'.

 DATA: l_t_msg        TYPE rs_t_msg.
```

```
 DATA: l_s_msg         TYPE rs_s_msg.
 DATA: l_uid           TYPE sysuuid_25.
 DATA: l_r_appl_log    TYPE REF TO cl_rspc_appl_log.
 DATA: l_error.
 DATA: l_idx           TYPE sytabix.

*---------------------------------------------------------------------

*  call function 'RSSM_SLEEP_DEBUG'
*    EXPORTING
*       I_SECONDS = 20.

* Create instance of this process (process prefix + GUID25)...
 CALL FUNCTION 'RSSM_UNIQUE_ID'
   IMPORTING
     e_uni_idc25 = l_uid.
 CONCATENATE 'ZPUSH' l_uid INTO e_instance.

 CALL METHOD cl_rspc_appl_log=>create
   EXPORTING
     i_type      = c_type_name
     i_variant   = i_variant
     i_instance  = e_instance
   RECEIVING
     r_r_appl_log = l_r_appl_log.

* ---------------------------------------------------------------------
* This report will read the db-table /BIC/OHNWL_OH_XI
* and push the results to XI-system currently connected to AB5

* 1. Fill the interface structures of the ABAP proxy.
* ---------------------------------------------------------------------
* - it is assumed that the names of the XI structures are equal to the
*   BW InfoObject without the first number.
*   (reason: XI/XML doesn't allow leading number)

 DATA:
     lt_data   TYPE zpushdbpush_table_datarow_tab,
     ls_data   TYPE zpushdbpush_table_datarow,
     ls_applid TYPE zpushdbpush_table_process_id,
     l_db_line TYPE /bic/ohnwl_oh_xi,
     l_db_tab  TYPE TABLE OF /bic/ohnwl_oh_xi.

 SELECT * FROM /bic/ohnwl_oh_xi INTO TABLE l_db_tab.

 LOOP AT l_db_tab INTO l_db_line.
   CLEAR ls_data.
   MOVE-CORRESPONDING l_db_line TO ls_data.
   APPEND ls_data TO lt_data.
 ENDLOOP.

 ls_applid-log_id   = i_logid.
 ls_applid-variant  = i_variant.
```

```
 ls_applid-instance = e_instance.
 ls_applid-type     = c_type_name.

* 2. Call XI async using an ABAP-proxy
* ---------------------------------------------------------------
* - based on message interface definition the ABAP proxy and
*   needed structures can be generated automatically.
*   (Use transaction SPROXY to initiate/regenerate)
*
* - The acknoledgement feature is only supported by some adapters.
*   Change constant to enable the acknoledgement request.
*
* - You can determine the ID of the message - this is helpful
*   to find your request in the monitors (transaction SXMB_MONI)

 CONSTANTS:
    c_ack_enabled TYPE c VALUE ' '.

 DATA:
* Definition for error handling
      lr_ai_system_fault TYPE REF TO cx_ai_system_fault,
      l_errortext       TYPE string,
      l_errorcode       TYPE string,

* Definitions for proxy and proxy interface
      lr_push  TYPE REF TO zpushdbco_push_table_msg_int_o,
      ls_out   TYPE zpushdbpush_table_msg_type,

* Definitions for acknowlegment
      ls_status      TYPE prx_ack_status,
      lr_ack         TYPE REF TO if_ws_acknowledgment,
      ls_req_detail   TYPE prx_ack_request_details,
      lr_async_messaging TYPE REF TO if_wsprotocol_async_messaging,
      l_cnt          TYPE i,

* Definitions for message ID determination
      lr_message_id_protocol TYPE REF TO if_wsprotocol_message_id,
      l_message_id TYPE sxmsmguid.

* add data table to the interface structure
 ls_out-push_table_msg_type-datarow = lt_data.
 ls_out-push_table_msg_type-process_id = ls_applid.

 TRY.
    CREATE OBJECT lr_push.

  CATCH cx_ai_system_fault INTO lr_ai_system_fault.
    l_errortext = lr_ai_system_fault->errortext.
    l_errorcode = lr_ai_system_fault->code.

    CLEAR l_s_msg.
    l_error = 'X'.
    l_s_msg-msgty = 'E'.
```

```
    l_s_msg-msgid = 'UPF'.
    l_s_msg-msgno = '001'.
    l_s_msg-msgv1 = l_errorcode.
    l_s_msg-msgv2 = l_errortext.
    APPEND l_s_msg TO l_t_msg.

  ENDTRY.

  TRY.

* Enable aknowledgement if required
    IF NOT c_ack_enabled IS INITIAL.

      lr_async_messaging ?= lr_push->get_protocol( if_wsprotocol=>async_messaging ).
      lr_async_messaging->set_acknowledgment_requested(
if_wsprotocol_async_messaging=>co_complete_acknowledgment ).

*     As alternative to complete acknowledgement:
*     -------------------------------------------
*     - check out other constants of if_wsprotocol_async_messaging
*     - set manually by:
*       lr_req_detail-application_ok = abap_true.
*       lr_async_messaging->set_acknowledgment_requested( details = lr_req_detail ).

    ENDIF.

    lr_message_id_protocol ?= lr_push->get_protocol( if_wsprotocol=>message_id ).

    CALL METHOD lr_push->execute_asynchronous
      EXPORTING
        output = ls_out.

   CATCH cx_ai_system_fault INTO lr_ai_system_fault.
    l_errortext = lr_ai_system_fault->errortext.
    l_errorcode = lr_ai_system_fault->code.

    CLEAR l_s_msg.
    l_error = 'X'.
    l_s_msg-msgty = 'E'.
    l_s_msg-msgid = 'UPF'.
    l_s_msg-msgno = '001'.
    l_s_msg-msgv1 = l_errorcode.
    l_s_msg-msgv2 = l_errortext.
    APPEND l_s_msg TO l_t_msg.

  ENDTRY.

* is needed here if not done later by the system
  COMMIT WORK.

* read message ID
  l_message_id = lr_message_id_protocol->get_message_id( ).
```

```
 CLEAR l_s_msg.
 l_s_msg-msgty = 'I'.
 l_s_msg-msgid = 'UPF'.
 l_s_msg-msgno = '001'.
 l_s_msg-msgv1 = 'The message send has the ID: '.
 APPEND l_s_msg TO l_t_msg.

 l_s_msg-msgv1 = 'MSGGUID'.
 l_s_msg-msgv2 = l_message_id.
 APPEND l_s_msg TO l_t_msg.

* read acknowledgement status
 IF NOT c_ack_enabled IS INITIAL AND l_error IS INITIAL.

  DO 20 TIMES.
   l_cnt = l_cnt + 1.

   TRY.
     lr_ack = cl_proxy_access=>get_acknowledgment( l_message_id ).
     ls_status = lr_ack->get_status( ).

    CATCH cx_ai_system_fault INTO lr_ai_system_fault .
     IF lr_ai_system_fault->code = cx_xms_syserr_proxy=>co_id_no_ack_arrived_yet.
       sy-subrc = 1.
     ELSE.
       l_errortext = lr_ai_system_fault->errortext.
       l_errorcode = lr_ai_system_fault->code.
       CLEAR l_s_msg.
       l_error = 'X'.
       l_s_msg-msgty = 'E'.
       l_s_msg-msgid = 'UPF'.
       l_s_msg-msgno = '001'.
       l_s_msg-msgv1 = l_errorcode.
       l_s_msg-msgv2 = l_errortext.
       APPEND l_s_msg TO l_t_msg.
     ENDIF.
   ENDTRY .

   IF ls_status IS INITIAL.
    CLEAR l_s_msg.
    l_s_msg-msgty = 'I'.
    l_s_msg-msgid = 'UPF'.
    l_s_msg-msgno = '001'.
    l_s_msg-msgv1 = l_cnt.
    l_s_msg-msgv2 = ' . try:'.
    l_s_msg-msgv3 = 'Acknowledgement NOT YET arrived!' .
    l_s_msg-msgv4 = sy-uzeit.
    APPEND l_s_msg TO l_t_msg.
   ELSE.
    EXIT.
   ENDIF.
   WAIT UP TO 5 SECONDS.
  ENDDO.
```

```
  CLEAR l_s_msg.
  l_s_msg-msgty = 'I'.
  l_s_msg-msgid = 'UPF'.
  l_s_msg-msgno = '001'.
  l_s_msg-msgv1 = l_cnt.
  l_s_msg-msgv2 = ' . try:'.
  IF ls_status IS INITIAL.
    l_s_msg-msgv3 = 'Acknowledgement Never arrived!' .
    l_s_msg-msgv4 = sy-uzeit.
    APPEND l_s_msg TO l_t_msg.
  ELSE.
    l_s_msg-msgv3 = 'Acknowledgement arrived at:' .
    l_s_msg-msgv4 = sy-uzeit.
    APPEND l_s_msg TO l_t_msg.
*     acknowledgment_status
    CLEAR l_s_msg.
    l_s_msg-msgty = 'I'.
    l_s_msg-msgid = 'UPF'.
    l_s_msg-msgno = '001'.
    l_s_msg-msgv1 = 'Status:'.

    IF ls_status-ack_status = if_ws_acknowledgment=>co_stat_ack_app_ok.
      l_s_msg-msgv2 = 'Application Status: OK'.
    ELSEIF ls_status-ack_status = if_ws_acknowledgment=>co_stat_ack_sys_error_final.
      l_s_msg-msgv2 = 'System Status: ERROR'.
    ELSEIF ls_status-ack_status = if_ws_acknowledgment=>co_stat_ack_app_error_final.
      l_s_msg-msgv2 = 'Application Status: ERROR'.
    ELSE.
      l_s_msg-msgv2 = 'Unexpected Status: ERROR'.
    ENDIF.
  ENDIF.
ENDIF.

LOOP AT l_t_msg INTO l_s_msg.
  CALL METHOD l_r_appl_log->add_message
    EXPORTING
      i_msgty = l_s_msg-msgty
      i_msgid = l_s_msg-msgid
      i_msgno = l_s_msg-msgno
      i_msgv1 = l_s_msg-msgv1
      i_msgv2 = l_s_msg-msgv2
      i_msgv3 = l_s_msg-msgv3
      i_msgv4 = l_s_msg-msgv4.
ENDLOOP.

CALL METHOD l_r_appl_log->save
  EXCEPTIONS
    internal_failure = 1
    OTHERS           = 2.

FREE l_r_appl_log.
```

```
 IF l_error = 'X'.
   e_state = 'R'.
 ELSE.
   e_state = 'A'.
   e_hold  = 'X'.
 ENDIF.


ENDMETHOD.
```

5.1.5 IF_RSPC_EXECUTE~GIVE_CHAIN

```
method IF_RSPC_EXECUTE~GIVE_CHAIN .

endmethod.                    "
```

5.1.6 IF_RSPC_CALL_MONITOR~CALL_MONITOR

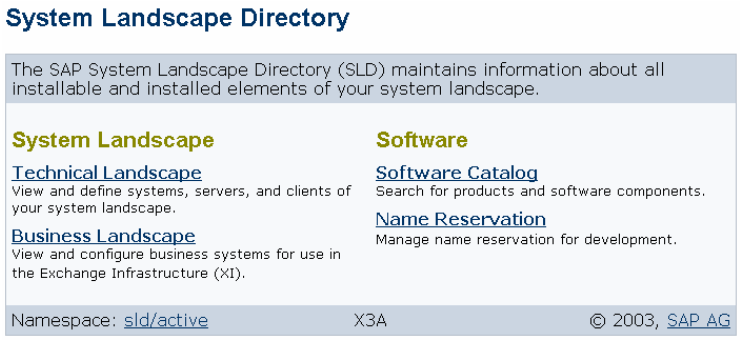
```
method IF_RSPC_CALL_MONITOR~CALL_MONITOR .

type-pools: rs.

data: l_t_messages TYPE RS_T_MSG,
     l_s_messages type rs_s_msg.

  CALL METHOD CL_RSPC_APPL_LOG=>DISPLAY
    EXPORTING
      I_TYPE         = 'ZPUSH'
      I_VARIANT      = i_variant
      I_INSTANCE     = i_instance
      I_NO_DISPLAY   = 'X'
    IMPORTING
      E_T_MSG        = l_t_messages
    EXCEPTIONS
      INTERNAL_FAILURE = 1
      others         = 2.
  IF SY-SUBRC <> 0.
    MESSAGE ID SY-MSGID TYPE SY-MSGTY NUMBER SY-MSGNO
            WITH SY-MSGV1 SY-MSGV2 SY-MSGV3 SY-MSGV4.
  ENDIF.

  read table l_t_messages into l_s_messages
      with key msgv1 = 'MSGGUID'.

  if sy-subrc = 0.
    submit RSXMB_SELECT_MESSAGES and return
         with msgguid EQ l_s_messages-msgv2.
  else.
    submit RSXMB_SELECT_MESSAGES via selection-screen and return.
  endif.
endmethod.
```

5.2 Create a software component

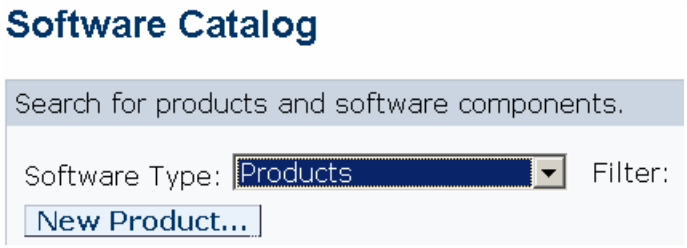
The example shown will create the software component **Lighthouse Scenario** based on the product **Lighthouse Receiver**. Feel free to change the names according to your needs.

1. Choose System Landscape Directory

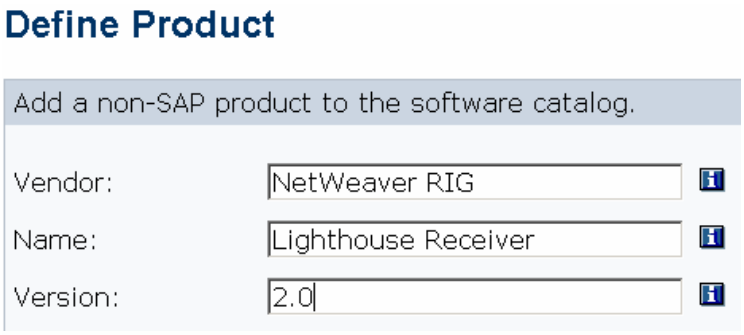
2. Choose Software Catalog.



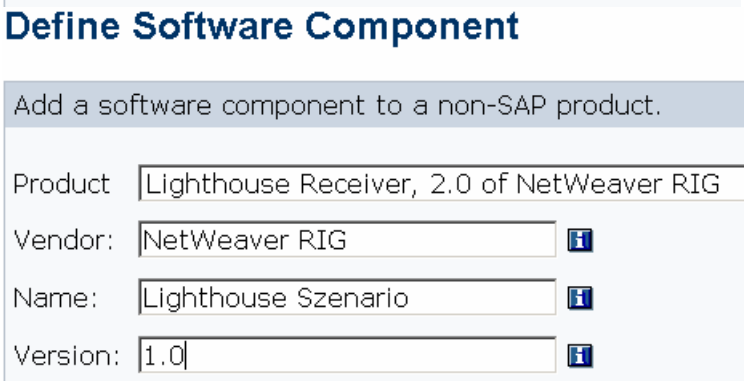
3. Press New Product.



4. Complete entries as shown and press create.



5. Complete entries as shown and press create. That's it.



6. Before you can use this new software component, you have to start the integration builder.

Choose from the main menu ***tools*** ➔ ***retrieve from SLD*** ➔ ***Import Software Component Versions***.