

Version 9 Release 2

*A Guide to Graphical Data Mapping in
IBM Integration Bus*

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page 341.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright IBM Corporation 2014.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Using message maps	1
Chapter 2. Graphical Mapping overview	3
Graphical Data Mapping editor	5
Message maps	7
Submaps	10
Guidelines for developing reusable graphical data mapping assets	13
Considerations for mapping messages modeled in message sets	13
Chapter 3. Designing a message map	15
Input and output messages to a message map	17
Advanced XML schema structures valid in input and output messages	18
Substitution groups	19
Wildcards (xsd:any)	19
Derived types	19
List types (xs:list)	20
Union types (xs:union)	20
Chapter 4. Transforms (Mapping operations)	21
Choosing a transform to set the value of a simple type output element	23
Choosing a transform to set the value of a complex output element	25
Choosing a transform to map repeating elements	27
Selecting the indexes of input array elements	28
Choosing a transform to concatenate input data	29
Choosing a transform to perform an arithmetic operation	31
Choosing a transform to define a conditional mapping	31
Choosing a transform to map an input message to multiple output messages	32
Chapter 5. Handling nulls in message maps	35
Chapter 6. Using nested maps	37
Chapter 7. Configuring your workspace mapping preferences	39
Setting mapping preferences for your workspace	39
Setting mapping keyboard preferences for your workspace	41
Chapter 8. Creating a message map	45
Creating a message map	47
Creating a message map in the Application Development view	48
Creating a message map from a Mapping node	49
Creating a submap	50
Creating a submap by using the Graphical Data Mapping editor	51
Creating a submap in the Application Development view	52
Creating a submap by using the Submap transform	53
Converting a local map into a submap	54
Creating a message map programmatically	56
Creating a local map by using the Local map transform	57
Converting a submap into a local map	57
Creating a graphical data map in the Eclipse editor	59
Chapter 9. Editing message maps	61
Configuring the general properties of a message map	62
Adding input and output messages	63
Mapping xsd:any on an input or output message	64
Casting wildcards in a map	65
Casting a wildcard defined as <i>xsd:any</i> into a specific type for a SOAP message	65
Casting a base type to a derived type or extension type	68
Mapping input to output elements manually	70
Adding connections between input and output elements	70
Connecting multiple input elements to a transform	71
Mapping input to output elements automatically	73
Mapping by same name	75
Mapping by similar name	76
Examples of similarity values	77
Format of the synonym file	77
Algorithm used to match synonyms	80
Creating and using a synonym file	81
Selecting matches	82
Specifying a transform (mapping operation)	82
Configuring the properties of a transform	84
Defining an XPath conditional expression for a transform	85
Defining an XPath conditional expression for a structural transform (ForEach)	89
Choosing an XPath conditional expression that tests for a nil value in a transform	91
Grouping transforms into nested maps	92
Using content assist (Mapping syntax)	93
Deleting objects and transforms	94
Chapter 10. Advanced editing in a message map	95
Configuring the message map to include message assembly components	95
Choosing message assembly components to include in a message map	96

Choosing a mapping action	98
Customizing a message map to include a message assembly component	105
Configuring the properties of the input and the output message assembly to a message map	108
Mapping transport headers.	109
Mapping data in the local environment tree	111
Configuring the local environment tree	
Variables folder by using the Cast function	113
Adding database definitions to the IBM Integration Studio	116
Creating a data design project	117
Creating a database definition (.dbm file) by using the New Database Definition File wizard	118
Creating a database definition from scratch	120
Accessing integration node properties from a Mapping node	122
Accessing user-defined properties from a Mapping node	122

Chapter 11. Setting the value of an output element by using a transform or a function. 125

Setting the value of an output element to a simple data type.	126
Setting the value of an output element with a explicit data type	128
Setting the value of a simple output element to a default or fixed value.	130
Setting the value of a simple type element included in a complex type output structure to a default or fixed value	130
Creating a nil output element	132
Creating an empty output element	134
Initializing an output element by using the Assign transform	134
Initializing a simple or complex output element by using the Create transform.	135

Chapter 12. Copying a selected element of a repeating structure to a single output 139

Chapter 13. Copying some values of a repeating element when the input and output structures are the same. 141

Chapter 14. Copying some values of a repeating element when the input and output structures are different 143

Chapter 15. Splitting an input message into multiple identical output messages 145

Chapter 16. Mapping an input message into different output messages 151

Chapter 17. Using Java API classes for Custom Java mapping transforms . 155

Chapter 18. Applying mapping overrides 157

Chapter 19. Mapping database content 159

Selecting data from a table	159
Modifying data in a database by using mapping	161
Inserting data into a table	162
Updating data in a table.	163
Deleting data from a table	165
Data type considerations for mapping database content	167
Calling a stored procedure from a map.	168
Handling database exceptions in a graphical data map	171
Behavior when modifying database column values from optional source elements.	172

Chapter 20. Referencing message maps in your solution. 175

Referencing an existing message map from a Mapping node	175
Dynamically selecting a message map	176
Calling a submap	177

Chapter 21. Transforming a SOAP message in a message map 179

Mapping a SOAP message by using a conditional transform.	181
Mapping a SOAP message by using the Override function	183

Chapter 22. Creating or transforming a BLOB output message by using a graphical data map	185
---	------------

Chapter 23. Mapping from a BLOB message to an output message using a graphical data map	187
--	------------

Chapter 24. Troubleshooting graphical data maps.	189
---	------------

Chapter 25. Deploying message maps	191
---	------------

Chapter 26. Transform types in the Graphical Data Mapping editor	193
---	------------

Append	195
Assign.	198
Cast type (xs:type)	199
Concat	201
Convert	204
Create	204
Custom ESQL	208
Equivalent ESQL types and schema types	209
Equivalent ESQL and XPath mapping functions	210
Custom Java.	212
Custom XPath	214
Database Routine	216
Delete	216
Failure	217
For Each	218
Group	222
If, Else if, and Else	222
Insert	223
Join	223
Local map	225
Move	225
Normalize	226
Return.	226
Select	227
Submap	227
Substring.	227
Update	228
Built-in XPath transforms	229
fn:concat	231
fn:string-join.	234
fn:substring	237
fn:count	239
fn:sum.	241

Chapter 27. Scenario: Transforming SOAP messages by using a message map	245
--	------------

Introduction to the "Transforming SOAP messages by using a message map" scenario	245
Context	246
Technical solution	247
Implementing the solution	248
Creating the scenario initial configuration	249

Creating a message map to transform SOAP messages	250
Transforming elements in the Properties folder by using the Override function	254
Customizing a message map to include the local environment tree	257
Configuring the local environment tree Variables folder by using the Cast function	260
Configuring the message map to include the SOAP message	264

Chapter 28. Scenario: Using a message map to enrich a message with data from a database	289
--	------------

Introduction to the "Using a message map to enrich a message with data from a database" scenario	289
Context	290
Technical solution	291
Implementing the solution	292
Creating the scenario graphical data map configuration	292
Creating the scenario database configuration	294
Configuring a database in the IBM Integration Studio.	295
Configuring an integration solution to access database resources.	303
Adding database tables your message map	305
Configuring the Select transform in a message map	309
Handling database failures in a Select transform	312
Configuring a database to be available at run time	316

Chapter 29. Using or converting legacy resources into message maps	321
---	------------

Changes in behavior in message maps converted from legacy message maps	322
Behavior changes during the development phase	322
Behavior changes during the deployment phase	323
Behavior changes at run time	324
Planning the conversion of a legacy message map	324
Converting a message map from a .msgmap file to a .map file	325
Managing conversion warnings on converted legacy message maps.	327
Managing conversion errors on converted legacy message maps	330
Converting a legacy message map that includes transformations of the local environment tree or <i>xsd:any</i> elements	332
Converting a legacy message map that includes user-defined ESQL procedures.	333
Converting a legacy message map that includes ESQL mapping functions	334
Converting a legacy message map that includes calls to message map functions	336
Converting a legacy message map that includes relational database operations	337

Converting a legacy message map that is called
from an ESQL statement in a Compute node . . . 337
Replacing a WebSphere Message Broker Version 7
Mapping node 339

Notices 341
Programming interface information 343

Trademarks 343

Sending your comments to IBM . . . 345

Chapter 1. Using message maps

You can use a message map to graphically transform an input message into a required output message; to enrich the output message with data from a database; to dynamically set routing or destination control for the output message; and to modify data in a database system. You can use drag actions to make connections, select transforms, and build logic to transform your message data without programming.

About this task

A message map is the IBM® Integration Bus implementation of a graphical data map. It is based on XML schema and XPath 2.0 standards.

A message map offers the ability to achieve the transformation of a message without the need to write code. It provides a visual image of the transformation, and simplifies its implementation and ongoing maintenance.

You can use a message map to adopt any of the following integration requirements graphically:

- **Transform a message:** You can use a message map to graphically transform a message assembly, message body, and properties, according to the transforms and XPath functions defined in the message map. You can use the full set of XPath 2.0 expressions and functions to implement data calculations and manipulations in a message map. To define the input and output messages to a map, you can use a schema base message model, which defines the structure of the data and provides information about the data type.
- **Enrich a message with data available in an external database:** You can use a message map to enrich, or conditionally set the output message with data from a database table. The table data structure must be defined to the message map, and an SQL where clause can be used to select specific rows. The resulting row data is presented as an extra input in the message map, according to the database schema.
- **Modify data located in an external database.**
- **Route a message based on content:** You can use a message map to graphically route a message. You can modify the local environment tree to set a dynamic message destination.

Procedure

Read the following sections to learn how to design, create, configure, and troubleshoot a message map and its associated resources:

- Graphical data maps offer the ability to achieve the transformation of a message without the need to write code. Depending on the data transformation re-usability and manageability requirements, you can use a message map, a submap, a local map, or a legacy message map. For more information, see Chapter 2, “Graphical Mapping overview,” on page 3.
- You can create a graphical data map, a message map, a submap, or a legacy message map to transform a message. For more information, see Chapter 8, “Creating a message map,” on page 45.

- You can edit a message map by using the Graphical Data Mapping editor. For more information, see Chapter 9, “Editing message maps,” on page 61.
- You can use the Graphical Data Mapping editor to set the value of an output element by using an expression, a transform, or a function. For more information, see Chapter 11, “Setting the value of an output element by using a transform or a function,” on page 125.
- You can reference a message map during the development phase. You can also reference a message map dynamically at run time. For more information, see Chapter 20, “Referencing message maps in your solution,” on page 175.
- You can diagnose and solve problems that you encounter when you use a message map. For more information, see Chapter 24, “Troubleshooting graphical data maps,” on page 189.
- You can compile and deploy a legacy message map in IBM Integration Bus. However, to modify the legacy message map, you must first convert the legacy message map to a message map. For more information, see Chapter 29, “Using or converting legacy resources into message maps,” on page 321.

Chapter 2. Graphical Mapping overview

Graphical data maps offer the ability to achieve the transformation of a message without the need to write code. You can use a message map, a submap, a local map, or a legacy message map. To define the input and output messages to a map, you can use a schema base message model, which defines the structure of the data and provides information about the data type.

Message maps

A message map is the IBM Integration Bus implementation of a graphical data map. It is based on XML schema and XPath 2.0 standards.

A message map offers the ability to achieve the transformation of a message without the need to write code. It provides a visual image of the transformation, and simplifies its implementation and ongoing maintenance.

You can use a message map to graphically transform, route, and enrich a message. You can use a message map to modify data in a database system. You can use drag actions to make connections, select transforms, and build logic to transform your message data without programming.

For more information, see “Message maps” on page 7.

Submaps

A *submap* is a reusable form of message map.

You use a submap to reuse common data transformations. You define in the submap the mapping functions that transform a set of elements from the input object to the output object.

You can reuse submaps in other products that support graphical data maps.

Note: If you plan to reuse data transformations across different products, read “Guidelines for developing reusable graphical data mapping assets” on page 13.

For more information, see “Submaps” on page 10.

Local maps

A *local map* is a subset of data transformations between input elements and output elements that are part of a message map. You define a local map by creating a **Local map** transform in a message map.

A local map is not an independent resource. There is no physical file that is associated with a local map.

The scope of a local map is the message map. A local map is processed with the message map.

Local maps provide a way of breaking up a large message map into nested groups of mapping elements.

You can use local maps to simplify the overall message map presentation. You can structure complex data transformations into nested groups that are easier to manage and implement.

For more information, see “Local map” on page 225.

Legacy message maps

A legacy message map is a message map that is created as a .msgmap file in earlier versions of WebSphere® Message Broker Version 8, for example in WebSphere Message Broker Version 7.

Note: WebSphere Message Broker Version 8 introduces graphical data maps. These message maps replace the previous message map format, and are created as .map files.

You can compile and deploy a legacy message map in IBM Integration Bus. However, if you need to modify a legacy message map, you must first convert the legacy message map to a message map.

For more information, see Chapter 29, “Using or converting legacy resources into message maps,” on page 321.

Choosing a type of graphical data map

Use the following table to identify the type of map that you must create when you transform data graphically in the Graphical Data Mapping editor:

Table 1. Types of map based on design requirements

	Recommended use	Type of resource	Supported in IBM Integration Bus
Message map	Graphical data mapping	.map file	Yes
Submap	Reuse of common data transformations	.map file	Yes
Local map	Reduce complexity when you read and manage a message map	No file. It is embedded within a Message map	Yes
Legacy message map	Reuse of maps that are developed in earlier versions of IBM Integration Bus	.map file	Supported for compatibility with earlier releases of IBM Integration Bus. (See note below.)

Note: You can use a legacy message map, but you cannot modify it in IBM Integration Bus. These types of maps are maintained for compatibility with earlier versions of IBM Integration Bus.

Editing a graphical data map

You edit a message map or a submap in the Graphical Data Mapping editor.

The Graphical Data Mapping editor saves message maps as .map files.

For more information, see “Graphical Data Mapping editor” and Chapter 9, “Editing message maps,” on page 61.

Mapping operations

You can use transforms to map graphically your data in the Graphical Data Mapping editor.

For more information, see Chapter 4, “Transforms (Mapping operations),” on page 21.

XPath

In the Graphical Data Mapping editor, you can use XPath functions in any of the following ways:

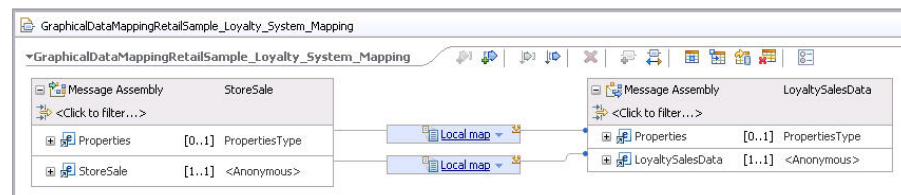
- You can define an XPath function to transform data by using a built-in XPath transform. For more information, see “Built-in XPath transforms” on page 229.
- You can define complex XPath expressions that combine multiple XPath functions to transform data by using the **Custom XPath** transform. For more information, see “Custom XPath” on page 214.
- You can define XPath expressions to set a condition on a transform or to filter an element in a repeating element. For more information, see “Defining an XPath conditional expression for a transform” on page 85.
- You can use XPath inline when you specify an argument to an XPath function. For more information, see “Built-in XPath transforms” on page 229.


Graphical Data Mapping editor

Use the Graphical Data Mapping editor to create and edit graphical data maps.

You can use the Graphical Data Mapping editor to take input (source) objects and transform them before you save the resulting output (target) objects.

Here is an example of the Graphical Data Mapping editor:



The input objects are shown on the left side of the canvas, and the output objects are shown on the right side. You can create connections between the input and output elements by clicking the grab handle , which is shown when you point to an element, and dragging the mouse to the element to which you want to connect. You can create connections either from left to right or from right to left.

You can use the Graphical Data Mapping editor to construct a graphical data map by using a wide variety of mapping transform functions. The transforms operate either from an input element to an output element on the canvas, or by directly setting the value of an output element.

You can use the functions either by using drag-and-drop, or from a menu action on the input or output element. The Graphical Data Mapping editor inserts the most appropriate mapping function on the newly created transform connection. For transforms that require multiple input connections, you can drag more connections onto the transform, and you can then select either a primary or supplementary connection mode. The Graphical Data Mapping editor adjusts the type of transform based on the new connections. When you create a connection, you can optionally change the type of transform function or start setting properties.

Some types of transforms involve complex input and outputs. They are edited by entering a nested view. The editor provides arrow buttons and breadcrumb navigation for nested transforms.

For information about the transforms that you can use in a graphical data map, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

The Graphical Data Mapping editor properties

The **Properties** pane displays the properties of the message map, its input and output elements, and transformations. For information about configuring the properties, see “Configuring the general properties of a message map” on page 62.

When you are working with the maximized view of the Graphical Data Mapping editor, you can bring the **Properties** pane back into focus in either of the following ways:

- Right-click the object and then select **Show in > Properties view**.
- Use the appropriate keyboard combination, which, by default, is **Alt+Shift+Q, R**. You can configure these keyboard shortcuts by selecting **Window > Preferences > General > Keys**.

Actions supported in the Graphical Data Mapping editor

Table 2. List of actions supported in the Graphical Data Mapping editor














Icon	Label	Action
	Add an input object	Adds an input object to the map by selecting a map input.
	Add an output object	Adds an output object to the map by selecting a map output.
	Sort mapping by source	Sorts the mapping objects so that the appearance of transforms is prioritized by the source.
	Sort mapping by target	Sorts the mapping object so that the appearance of transforms is prioritized by the target.
	Delete selected elements	Deletes the element or elements that are currently highlighted.
	Create a new transform	Creates a transform for the map.

Table 2. List of actions supported in the Graphical Data Mapping editor (continued)

Icon	Label	Action
	Auto map input to output	Automatically maps elements by using the Automap Wizard .
	Select rows from a database	Selects rows of data from a database table.
	Call a database routine	Calls a database routine to obtain the data to include in the map.
	Insert a row into a database table	Inserts one row into a database table.
	Update rows in a database table	Update one or more rows in a database table.
	Delete rows from a database table	Delete one or more rows in a database table.
	Show object preferences	Opens the Preferences window for the currently selected object.

Casting data structures

In the Graphical Data Mapping editor, you can use the **Cast** function to cast data structures.

Wildcards can be used to create a flexible message model that can be redefined when a more detailed definition is required. You define a wildcard as `xsd:any` in your schema.

When your message model schemas contain one or more wildcards, you can use the **Cast** function to redefine parts of the input or output model in a message map.

For more information, see “Casting wildcards in a map” on page 65.

Message maps

You can use a message map to graphically transform, route, and enrich a message. You can use a message map to modify data in a database system. You can use drag actions to make connections, select transforms, and build logic to transform your message data without programming.

A message map is the IBM Integration Bus implementation of a graphical data map. It is based on XML schema and XPath 2.0 standards.

You can use a message map to achieve the transformation of a message without the need to write code, providing a visual image of the transformation, and simplifying its implementation and ongoing maintenance.

Input and output data to a message map

In IBM Integration Bus, the logical tree structure is the internal representation of a message. It is also known as the message assembly.

When a message arrives to IBM Integration Bus, a parser is called. Each parser is suited for a particular type of message, and is known as a message domain. A parser converts the bit stream of an input message to its internal format. The data structure that you define in a message map for an input or an output message is the IBM Integration Bus internal representation of the message.

To configure the input and the output to a message map, you define an input message assembly and one or more output message assemblies.

- You can have 1 input to a message map.
- You can have multiple outputs in a message map.

For more information, see “Input and output messages to a message map” on page 17.

Supported message domains

The following message domains are supported when you use message maps:

- DFDL: You use this domain to manipulate general text or binary data streams including industry standards.
- XMLNSC: You use this domain to manipulate XML documents.
- SOAP: You use this domain to manipulate SOAP messages.
- DataObject: You use this domain to manipulate data without a stream representation.
- BLOB: You use this domain to manipulate messages with content that cannot be interpreted and subdivided into smaller sections of information.
- MRM: This domain is maintained for compatibility with earlier versions of IBM Integration Bus.

Message assembly components

You can configure a message map to include the following message assembly components:

- The Properties folder.
- The message tree transport headers: For more information, see “Mapping transport headers” on page 109.
- The local environment tree: For more information, see “Mapping data in the local environment tree” on page 111.
- The message body

Note: Other message assembly components, such as the Exception list tree, are passed on unchanged. They cannot be modified in a message map.

Message body

You define the message body by defining a message model in any of the following ways:

- Define a message model by using a predefined message model.
- Define a message model by using a user-provided message model.

You can use message models for any messages that you want to include in a mapping. You can select the message model from your existing message models in your application, integration service, or library when you create a message map.

The mapping facility supports message models that are provided in DFDL schema and XML schema files, or MRM message sets.

In a message map, input and output body objects are defined by reference to message models, which provide a definition of the message structure and type through the following components:

- Simple elements and attributes, which define the type, range, and default values
- Complex elements, which build the structure of the message
- Repeating simple or complex elements
- Other (embedded) messages

If your message model includes wildcards (`xsd:any`), you can use a **Cast** function to redefine these data elements to a global type or element from any message schema in your application. For more information, see “Casting wildcards in a map” on page 65.

Editing a message map

You edit a message map in the Graphical Data Mapping editor. You can create, modify, or delete a message map.

The Graphical Data Mapping editor saves message maps as `.map` files.

For more information, see Chapter 9, “Editing message maps,” on page 61.

Mapping behavior

By default, any message assembly component that is not included in the message map is copied from input to output unchanged.

To modify a message assembly component, you add the header or folder to the input and output message assembly in the message map, and provide transformations.

To delete a message assembly component, you add the component to the input message assembly, but not the output message assembly in the message map.

To create a message assembly component, you add the component only to the output message assembly in the message map.

Mapping operations

You can use any of the following transforms to map graphically your data in the Graphical Data Mapping editor:

- **Core mapping transforms:** You can use built-in structural and functional mapping operations to graphically construct the required message transformations to build the output message.
- **Custom transforms:** You can use custom transformations to define specialized transformations in XPath 2.0, Java™, or ESQL functions.
- **XPath functions** (`fn:functionName`): You can use XPath 1.0 and XPath 2.0 functions to transform data in a message map.
- **Database transforms:**
 - You can use the **Select** transform to query one or more database tables, and retrieve data that you can use in the message map. You can use the data to

set output element values, define conditions, or use as input to build other transforms conditions. Database tables can be set as extra outputs of a message map.

- You can use a **database routine** transform to call a stored procedure from a database, and retrieve data that you can use in the message map. You can use the data to set output element values, define conditions, or use as input to build other transforms conditions.

Note: Only IBM DB2[®] stored procedures are supported in IBM Integration Bus.

- You can use the **Insert** transform to add one new row of data, or multiple rows of data, into a database table.
- You can use the **Update** transform to modify a row of data, or multiple rows of data, in a database table.
- You can use the **Delete** transform to delete a row of data, or multiple rows of data, in a database table.
- **Cast** function: You can use the **Cast** function to cast schema types.

For more information, see Chapter 4, “Transforms (Mapping operations),” on page 21.

Local maps

You can use local maps as navigation aids. You view the map elements in a hierarchical way. Unlike submaps, local maps are not separate files and they are not reusable. They provide a way of breaking up a large map into nested groups of mapping elements and processing the complex elements of the whole message.

Deploying a message map

Message map files are deployed to the IBM Integration Bus run time environment to enable them to be run in a message flow.

When you build and deploy a BAR file for an integration solution, the message map files are automatically included.

When you deploy independent resources, the BAR file editor provides a resource category to allow message maps to be selected for deployment.

If your message map file is used by multiple solutions, you might store the map in a shared library. Shared libraries that are referenced by applications must be deployed with or before the applications that refer to them. You can deploy the shared libraries directly to the integration server before you deploy the applications. Alternatively, you can include the applications and shared libraries in a BAR file and deploy the BAR file. If you update the message map in a shared library, those changes are available automatically to all applications that refer to that shared library.

Submaps

You can use a **submap** to use the same mapping transformation in multiple message maps.

You use submaps to define a set of mapping functions that you can reuse in multiple message maps.

A **submap** references another map. It calls or invokes a map from the same file or another map file, which can be stored in a library, an application, an integration service, or an Integration project.

A submap can contain components of the message body only, such as global elements and global types. A submap does not contain Properties, message headers, or the Local Environment tree.

For more information, see “Creating a submap” on page 50.

When you use submaps, you must consider the following behavior:

- You can use a submap to define transformations between global elements or global types.
- A submap cannot be used to transform local anonymous complex types, that is, `xs:any` elements.
- A submap can be placed in any project that is visible to the main map that calls the submap.

Editing a submap

You edit a submap in the Graphical Data Mapping editor.

The Graphical Data Mapping editor saves submaps as `.map` files.

For more information, see “Graphical Data Mapping editor” on page 5.

Input and output data to a submap

To configure the input and the output to a submap, you must define one input message, and one output message.

The input message and the output message must be defined by a user-provided message model if it is to be transformed by using a message map.

You must have message models for any messages that you want to include in a mapping. You can select the message model from your existing message models in your application, integration service, or library when you create a message map. The mapping facility supports message models that are provided in DFDL schema and XML schema files, or MRM message sets.

In a message map, input and output objects are defined by reference to message models. They provide a definition of the message structure and type through the following components:

- Simple elements and attributes, which define the type, range, and default values
- Complex elements, which build the structure of the message
- Repeating simple or complex elements
- Other (embedded) messages

If your message model includes wildcards (`xsd:any`), you can use a **Cast** function to redefine these data elements to a global type or element from any message schema in your application. For more information, see “Casting wildcards in a map” on page 65.

Mapping operations

You can use any of the following transforms to map graphically your data in the Graphical Data Mapping editor:

- **Core mapping transforms:** You can use built-in structural and functional mapping operations to graphically construct the required message transformations to build the output message.
- **Custom transforms:** You can use custom transformations to define specialized transformations in XPath 2.0, Java, or ESQL functions.
- **XPath functions** (*fn:functionName*): You can use XPath 1.0 and XPath 2.0 functions to transform data in a message map.
- **Database transforms:**
 - You can use the **Select** transform to query one or more database tables, and retrieve data that you can use in the message map. You can use the data to set output element values, define conditions, or use it as input to build other transforms conditions. Database tables can be set as extra outputs of a message map.
 - You can use a **database routine** transform to call a stored procedure from a database, and retrieve data. You can use the data to set output element values, define conditions, or use as input to build other transforms conditions.

Note: Only IBM DB2 stored procedures are supported in IBM Integration Bus.

- You can use the **Insert** transform to add one new row of data, or multiple rows of data, into a database table.
- You can use the **Update** transform to modify a row of data, or multiple rows of data, in a database table.
- You can use the **Delete** transform to delete a row of data, or multiple rows of data, in a database table.
- **Cast function:** You can use the **Cast** function to cast schema types.

For more information, see Chapter 4, “Transforms (Mapping operations),” on page 21.

Starting a submap

A submap can be referenced from other message maps.

When you construct your transformation map, you create a submap to group part of the message transformation. The submap must be in a project visible to the main map that they are called from.

To start a submap, you define a **Submap** transform between the input object and the output object in your message map. The submap can then be used to enable reuse of common transformations for sections of, or the whole of, the message.

For more information, see “Calling a submap” on page 177.

Limitations

- A submap can provide callable mapping between global elements or global types from a message model.

- A submap cannot be used for local anonymous complex types. Anonymous complex types must be mapped within the main map, for example, by a local map.

Reusing a submap

You can use a submap to reuse common data transformations.

You can reuse submaps in other solutions, and in other products that support graphical data maps.

Note: If you plan to reuse a submap across different products, read “Guidelines for developing reusable graphical data mapping assets.”

Guidelines for developing reusable graphical data mapping assets

You can create graphical data maps (.map files) for reuse by other products that include the Graphical Data Mapping editor.

When you create graphical data maps that you want to reuse in other products, ensure that your graphical data maps conform to the following guidelines:

Build your reusable mapping structures as a submap

Many products use extra metadata to supplement the business message data, but a reusable transformation asset that is contained in a submap processes only the common business data.

Create a submap to contain the mapping structures that you want to reuse in other products, and then call this submap from a top-level mapping structure. You can use the Graphical Data Mapping editor to refactor a local map to a submap; see “Converting a local map into a submap” on page 54.

For more information about submaps, see “Creating a submap” on page 50.

Custom transforms can be only Custom Java and Custom XPath

Create custom transforms in your reusable submaps by using only **Custom Java** and **Custom XPath** transforms. Other custom transform types, such as Custom ESQL in IBM Integration Bus, are product-specific, and cannot be used.

Considerations for mapping messages modeled in message sets

You can use the Graphical Data Mapping editor to transform XML messages that are defined in XML Schema (.xsd files). Message Set modeling in IBM Integration Bus supports XML Schema extensions.

The Graphical Data Mapping editor supports both XML and text or binary messages that are modeled in IBM Integration Bus message sets, with the following considerations:

- Message sets provide facilities for defining message composition. When these extensions are used to redefine a wildcard in a message, they are not shown in the message map. The Graphical Data Mapping editor provides equivalent facilities for modeling choice in schema wildcards by using the Cast function. For more information about the Cast function, see “Mapping xsd:any on an input or output message” on page 64.

When you create a message map or convert a message map that includes a schema wildcard from a message set with XML Schema extensions, you must manually add a Cast function from the wildcard to the required schema element.

- The message map requires the message set schema (.xsdzip file) to be deployed to run your message map. If your existing message set is used for text and binary formats only, you can deploy your message map with only a .dictionary file in the BAR file. In this case, you must modify the message set to additionally set the XMLNSC domain support option, so it is added to a BAR with both a .dictionary file and .xsdzip file. If this option is not set, a warning is displayed in the Problems view, along with a quick fix action.

Chapter 3. Designing a message map

You can use a message map to graphically transform, route, or update an external system. For best performance and capability, you must design it to include the most appropriate transforms.

Procedure

Consider the following guidance to design a message map:

1. Design the data model of your input and output message per the solution requirements.

The function of a message map is driven by the data models that define the input and the output message structures of a map. At run time, the Graphical Data Mapping engine must account for all possible states of the data when executing the transformations you have defined in the map against the data models. You may have control or not over the data models in your solution. If you can influence the data model, these are some of the key points to consider:

- a. Set the cardinality of each element in a data model to specific values whenever possible.

When you define a logical model, you can configure the cardinality of each element by setting the **minoccurs** and the **maxoccurs** properties.

Avoid, whenever possible, configuring maximum flexibility unless actually required. Only set **minoccurs** to 0, when an element needs to be optional. Only set **maxoccurs** above 1 if the element will actually repeat.

- b. Define an element as nillable when you know that the application will need to handle out of bound value in the data.
- c. Enable schema validation when you are developing a map. Disable validation in your test and production systems unless the solution requires validation enabled.

Validation is the process of checking a message's structure, and optionally the values within, based on a description called a schema. The Mapping node relies on schema definitions, but it does not enforce them. If the input message does not conform to the schema being used, the output you expect to see might not be produced. During the development of your integration solutions, it is recommended that you enable validation. However, for other environments such as test or production, you should only leave validation on if your solution requires it because of the extra processing involved.

For more information, see [Validating messages](#).

2. Identify the type of message map.

- a. Use a message map to graphically transform, route, and enrich a message. You can use a message map to modify data in a database system.

For more information, see ["Message maps"](#) on page 7.

- b. Use a submap to define a set of mapping functions that you can reuse in multiple message maps.

For more information, see ["Submaps"](#) on page 10.

For example, the Mapping node invokes a map that deals with a message assembly. You can put the mapping for common parts of the message data into a submap to enable reuse.

3. Identify the input and output components to a message map.

- You can select an XML schema, DFDL schema, or message set to define the message body.
- You can choose any of the following message assembly components: the Properties tree, the local environment tree.
- You can add database tables.

You should only include a component when you need to read data or write data:

- To read elements of an input component, add the component to the input message assembly only. The Graphical Data Mapping editor passes to the output the input component unchanged.
- To modify elements of an input component, add the component to the input message assembly and to the output message assembly. Then, define transforms between its elements.
- To initialize an input component, that is, to create a new component in your output message, add the component only to the output message assembly.
- To add an input component, add the component to the output message assembly and populate at least one field. The Graphical Data Mapping editor creates a new output structure containing the results of your transformations.
- To delete an input component from the input message, add the component to the output message assembly and do not set any field.

For more information, see “Choosing a mapping action” on page 98.

4. For each output element, identify the transform and the input elements required to calculate its value.

When the transformation of an element from input to output becomes more than just a simple **Move**, or type conversion (**xs:type**), you can call on the full set of standard XPath 2.0 operators and functions to manipulate the data as required.

The Graphical Data Map editor offers the XPath functions as transform types in the pick list as well as in the content assist, “Ctrl-space” when editing expressions and conditions.

- “Choosing a transform to set the value of a simple type output element” on page 23
 - “Choosing a transform to set the value of a complex output element” on page 25
 - “Choosing a transform to map repeating elements” on page 27
 - “Choosing a transform to concatenate input data” on page 29
 - “Choosing a transform to perform an arithmetic operation” on page 31
 - “Choosing a transform to define a conditional mapping” on page 31
 - “Choosing a transform to map an input message to multiple output messages” on page 32
5. Define structure in your map so that you can have a single condition that applies to a group of transforms as opposed to having to repeat the condition on each transform because all the transforms are at the same level in the map.
 - a. Define a conditional expression for each transform to determine at run time whether the transform is applied or not.

For more information, see “Defining an XPath conditional expression for a transform” on page 85.

- b. Use structural transforms to enhance the readability and maintenance of your map. You obtain similar performance results whether all the transforms are in the main map or grouped into nested maps that are associated with structural transforms.

Other advantages when you use structural transforms are the following:

- 1) You can define a conditional expression that determines whether a nested map is applied at run time. For example if a set of child fields should only be mapped dependent on some attribute of the folder, place them all inside an **If** transform. The nested map is executed when the condition evaluates to **true**.
- 2) When you use a **Local Map** transform, you can convert the map to a submap if the need for reuse comes at a later stage. You can use an action in the Graphical Data Mapping editor to re-factor the **Local Map** into a **Submap**.
- 3) You can use the Auto map wizard to automatically create **Move** transforms from input to output elements based on some correlation of the names of input and output elements to create mappings.

For more information, see Chapter 6, “Using nested maps,” on page 37.

6. When you need to process data, rather than just move it from source to target, use custom transforms to define specialized transformations.
 - a. “Custom XPath” on page 214
 - b. “Custom Java” on page 212
 - c. “Custom ESQL” on page 208

From a performance point of view, it is recommended that you use **XPath** transforms or the **Custom XPath** transform as your first choice, then **Custom Java**. You can also use **Custom ESQL**.

Input and output messages to a message map

In a message map, you must define an input message and an output message. You can choose from a predefined message format, or a custom message format. You can cast `xsd:any` elements.

When a message arrives to IBM Integration Bus, a parser is called. Each parser is suited for a particular type of message, and is known as a message domain. A parser converts the bit stream of an input message to an internal format. A parser is also called when a logical tree that represents an output message is converted into a bit stream.

Note: The data structure that you define in a message map for an input or an output message is the IBM Integration Bus internal representation of the message.

Message domains

The following message domains are supported in a message map:

- DFDL
- XMLNSC
- SOAP
- DataObject
- BLOB
- MRM

Note: The MRM domain is supported for compatibility with legacy message maps.

Message assembly

In a message map, the *source message assembly* describes the input message and the *target message assembly* describes the output message.

A message assembly includes the properties tree, any relevant headers, the local environment tree, and the message body.

- When you create a top-level message map, only the Properties folder is initially included. A **Move** transformation from the input Properties folder to the output Properties folder is created by default where all input values are copied to the corresponding output values unchanged.
- The structure of the Properties folder, the transport headers, and the local environment tree are predefined in IBM Integration Bus.
- You can define the local environment tree **Variables** folder structure by using the **Cast** function.
- The input message body is defined by associating an input message model such as a DFDL schema, or an XML schema.
- The output message body is defined by associating an output message model such as a DFDL schema, or an XML schema.

You must set the **Output domain** property of the target message assembly to define the message domain in which an output message is to be built.

The message map uses the schema types of the output elements to create and set the elements of the output message tree.

Message models

The following message models are supported in a message map:

- Predefined message model
- Schema-based message model

You can select any of the following supplied message models as your input or your output message format:

- **SOAP_Domain_Msg** {}: You use this message model to handle messages in the SOAP domain.
- **BLOB** {}: You use this message model to handle messages in the BLOB domain.

For other supported message domains, you can select a schema-based message model.

- You use a schema-based message model when you have the message model in an XSD file. For more information on how to create a schema-based message model, see Modeling different data formats.

Advanced XML schema structures valid in input and output messages

You can use several advanced schema structures in graphical data maps.

You can use any of the following XML features in message models that are defined as inputs or outputs to the Graphical Data Mapping editor:

- “Substitution groups” on page 19

- “Wildcards (xsd:any)”
- “Derived types”
- “List types (xs:list)” on page 20
- “Union types (xs:union)” on page 20

Substitution groups

A *substitution group* is an XML schema feature that provides a means of substituting one element for another in an XML message.

- The element that can be substituted is called the **head** element.
- The **substitution group** is the list of elements that can be used in its place.

The head element and any mapped substitutions are shown by default in the Graphical Data Mapping editor. The mapped substitutions are listed beneath the head element.

You create mappings to or from members of substitution groups in the same way as you map other elements.

Wildcards (xsd:any)

You can use *wildcards* to create open content models. You can define `xsd:any` and `xsd:anyAttribute` wildcards.

Wildcards are characterized for the following attributes:

- The namespace attribute: You can use this attribute to specify the namespace that the elements or attributes that match the wildcard can come from.
- The processContents attribute: You can use this attribute to specify how the XML content matched by the wildcard is validated.

When you use wildcards in the Graphical Data Mapping editor, you must consider the following behavior:

- You can wire a `xs:any` or a `xs:anyAttribute` as the input or output of a **Submap** transform. Then, when you configure the **Submap** transform, you can define that input or output to be a particular type.
- A wildcard element can be instantiated only with another element.
- A wildcard attribute can be instantiated only with another attribute.
- You can use a global element or attribute as a wildcard replacement.

For more information, see “Mapping `xsd:any` on an input or output message” on page 64.

Derived types

A *derived type* is a data type that is related to another data type known as the base type or super type.

In a message map, you can cast a base type to a derived type or extension type. You can define transformations between subtypes of a data type.

When you use derived types in the Graphical Data Mapping editor, you must consider the following behavior:

- For an element of a specific type, the base type and the mapped derived types are shown by default. All attributes and elements of the base and derived types are displayed.
- You create mappings to or from a derived type and its elements in the same way that you map any base type and its elements.
- When you map an input element to an output element with a derived schema type or an extension schema type, the created output element is set with the relevant `xsi:type` attribute for that schema type.

List types (`xs:list`)

You can use `xs:list` to define a simple type element as a list of values of a specified data type. For example:

```
<xs:simpleType name='CustomerName'>
  <xs:list itemType='string' />
</xs:simpleType>
```

When you use `xs:list` in the Graphical Data Mapping editor, you must consider the following behavior:

- You map a simple type element that is defined as a list of values in the same way that you would map any other simple type attribute or element.

Union types (`xs:union`)

You can use `xs:union` to define a simple type as a collection of values from specified simple data types. It allows a value to conform to any one of several different simple types. For example:

```
<xs:element name="zipUnion">
  <xs:simpleType>
    <xs:union memberTypes="USStateName StateID"/>
  </xs:simpleType>
</xs:element>
<xs:element name="USStateName">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="StateID">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

When you use `xs:union` in the Graphical Data Mapping editor, you must consider the following behavior:

- You map a simple type element that is defined as a collection of values in the same way that you would map any other simple type attribute or element.

Chapter 4. Transforms (Mapping operations)

In the Graphical Data Mapping editor, you can define mapping operations such as transforms, cast functions, or XPath 2.0 functions. Mapping operations define the transformation actions on input data, and set the result to the output element.

Mapping operations to transform graphically your data

You can use any of the following transforms to map graphically your data in the Graphical Data Mapping editor:

- **Core mapping transforms:** You can use built-in structural and functional mapping operations to graphically construct the required message transformations to build the output message. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.
- **Custom transforms:** You can define specialized transformations to build custom ESQL functions, XPath 2.0, or Java. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.
- **XPath functions:** You can use XPath 1.0 and XPath 2.0 functions to transform data in a message map.

All XPath 2.0 functions are supported in the form `fn:functionName`.

For more information about XPath, see the online document W3C XML Path Language (XPath) 2.0.

- **Database transforms:**
 - You can use the **Select** transform to query one or more database tables, and retrieve data. You can use the data in the message map to set output element values, define conditions, or use as input to build other transforms conditions. Database tables can be set as extra outputs of a message map. For more information, see “Selecting data from a table” on page 159.
 - You can use a **database routine** transform to call a stored procedure from a database, and retrieve data. You can use the data in the message map to set output element values, define conditions, or use as input to build other transforms conditions.

Note: Only IBM DB2 stored procedures are supported in IBM Integration Bus. For more information, see “Calling a stored procedure from a map” on page 168.

Mapping operations to modify data in a database

Database tables can be set as more outputs of a message map.

You can use any of the following transforms to modify data in a database:

- **Insert** transform: You use the **Insert** transform to add one new row of data, or multiple rows of data, into a database table. For more information, see “Inserting data into a table” on page 162.
- **Update** transform: You use the **Update** transform to modify a row of data, or multiple rows of data, in a database table. For more information, see “Updating data in a table” on page 163.

- **Delete** transform: You use the **Delete** transform to delete a row of data, or multiple rows of data, in a database table. For more information, see “Deleting data from a table” on page 165.
- **Database routine** transform: You use a **database routine** transform to call a stored procedure or user-defined function from a database to insert, delete, or update data in one database table. For more information, see “Calling a stored procedure from a map” on page 168.

Note: Only IBM DB2 stored procedures are supported in IBM Integration Bus.

At design time, you must have a database definition file (.dbm file) in an available Data Design project for each database that you want to access. A data definition file contains one connection per database system.

At run time, you must have a JDBC connection of Type 4 defined for each database that your message map uses. You must configure a JDBCProvider configurable service per database. The JDBCProvider service name for a runtime database must be the same name as the development database name that you use in your message map.

For more information, see “Modifying data in a database by using mapping” on page 161.

Cast function to define a schema type

In the Graphical Data Mapping editor, you can use the **Cast** function to cast schema types.

You must cast a schema in any of the following instances:

- Cast the value that you assign to an output element so it matches the output element schema definition type.
- Cast the value of an element that you use as a parameter to a function where the parameter is of a different type.
- Cast the value of an element that you use as a condition on a transform where the type is different.
- Cast the value of an element when you work with a database, and the types differ.

To cast a schema, you can use the *xs:castOperation* functions, where *castOperation* is the name of the cast function.

Mapping behavior driven by the type of operation

When the transformations in the message map are constructed, the values for output message elements can be derived from any of the following components:

- Input message elements, through any of the following mapping operations:
 - **Move**, **Convert**, and other built-in transforms in the Graphical Data Mapping editor.
 - XPath 2.0 functions (prefix fn:). All XPath 2.0 functions are supported by the Mapping node. For more information about XPath, see W3C XML Path Language (XPath) 2.0.
 - Database input by using a database **Select** transform.

- Schema type casts. For more information, see “Mapping xsd:any on an input or output message” on page 64.
- Extra functions, which allow multiple input values to produce the output value, such as **concat** and **join**.
- The result of database **Select**, **Insert**, **Update**, **Delete**, and **Database routine** transforms.
- Constant values, through an **Assign** operation that uses a supplied value.
- Custom functions, user-defined XPath, Java, or ESQL.

The logic to derive values can be simple or complex. In addition to the transformation operations that set an output value, structural transforms are provided to enable conditional statements, loops, and nesting of transform logic into local maps.

For information about the supported transform types, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Choosing a transform to set the value of a simple type output element

You can use different transforms, such as the **Assign** transform, the **Create** transform, the **Move** transform, or the **xs:type** transform, to set the value of an output element.

Procedure

Complete the questionnaire to identify the transform that you can use to set the value of an output element:

1. Do you want to set the value to a fixed value?
Use the **Assign** transform.
2. Do you want to initialize the output element, that is, do you want to create an empty structure?
Use the **Create** transform to initialize a string output element or a hexBinary output element.
3. Do you want to create a nil output element?
Use the **Create** transform and verify that the output element is defined as `nillable="true"` in the schema.
4. Do you want to set the output element as nil by using an input element with a value of nil?
Use the **Move** transform. Verify that the input element is defined as `nillable="true"` in the schema. Ensure that the value of the input element is nil.
5. Do you want to set the output element to a default value?
Use the **Create** transform and verify that the output element has a default value set in the schema.
6. Do you want to set the output element with input from a database column?
Use the **Select** transform to obtain the database input value. Then, in the nested map that is associated with the **Select** transform, use the **Move** transform to set the value of the output element.
7. Do you want to set the value of the output element with the value of an input element? Do the input element and the output element have the same type associated? Do you want to cast the input value to the type of the output value?

- When the input and output element have the same type, use the **Move** transform.
 - When the input and output element have different data types, use the **xs:type** transform.
8. Do you want to calculate the value of the output element by using the value of one or more input elements?

To set an output element with a string data type or hexBinary data type, use any of the following transforms:

- **Concat**
- **Normalize**
- **Append**
- **Substring**
- **fn:string-join**
- **Custom XPath**
- **Custom Java**
- **Custom ESQL**

To set up an output element with any other data type, use any of the following transforms:

- Any supported XPath functions, for example, **fn:round**
- **Custom XPath**
- **Custom Java**
- **Custom ESQL**

9. Do you want to apply the transform always? Do you want to apply the transform when a condition based on input data occurs?

Define a conditional expression for the transform you choose. This expression determines when the transform is applied. For more information, see “Defining an XPath conditional expression for a transform” on page 85.

Results

Table 3. Setting the value of a simple output element

	Number of input elements that are required to set the value of the output element	Transforms to set a string data type, or a hexBinary data type	Transforms to set other simple data types
Set the output element with a fixed value	0	Assign	Assign
Initialize the output element	0	Create	not valid option
Set the output element as nil	0	Create Condition: The output element must be defined as <code>nillable="true"</code> in the schema.	Create Condition: The output element must be defined as <code>nillable="true"</code> in the schema.

Table 3. Setting the value of a simple output element (continued)

	Number of input elements that are required to set the value of the output element	Transforms to set a string data type, or a hexBinary data type	Transforms to set other simple data types
Set the output element as nil by using a nil input element	1	Move Condition: The input element must be defined as <code>nillable="true"</code> in the schema.	Move Condition: The input element must be defined as <code>nillable="true"</code> in the schema.
Set the output element with a default value	0	Create Condition: The output element must have a default value set in the schema.	Create Condition: The output element must have a default value set in the schema.
Set the output value from a database table column	1..N	Select transform to obtain the database input value and Move transform to set the value	Select transform to obtain the database input value and Move transform to set the value
Copy the value of the input element to the output element (both elements have the same data type)	1	Move	Move
Copy the value of the input element to the output element (elements have different data types)	1	xs:type	xs:type
Calculate the value by using the values of multiple input elements	1..N	Concat, Normalize, Append, Substring, fn:string-join, Custom XPath, Custom Java, Custom ESQL	XPath fn: functions, Custom XPath, Custom Java, Custom ESQL

What to do next

Learn about the transforms. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Choosing a transform to set the value of a complex output element

You can use different transforms, such as **Move**, **Create**, **Assign**, to set the value of a complex output element.

Procedure

Complete the questionnaire to identify the transform that you can use to set the value of a complex output element:

1. Do you want to set the value to a fixed value?
Use the **Assign** transform.
2. Do you want to initialize the output element, that is, do you want to create an empty structure?
Use the **Create** transform.
3. Do you want to create a nil output element?
Use the **Create** transform and verify that the output element is defined as `nillable="true"` in the schema.
4. Do you want to set the output element as nil by using an input element with a value of nil?
Use the **Move** transform. Verify that the input element is defined as `nillable="true"` in the schema. Ensure that the value of the input element is nil.
5. Do you want to set the output element to a default value?
Use the **Create** transform and verify that the output element has a default value set in the schema.
6. Do you want to set the output element with input from a database column?
Use the **Select** transform to obtain the database input value. Then, in the nested map that is associated with the **Select** transform, use the **Move** transform to set the values of the output element.
7. Do you want to set the value of the output element with the value of an input element? Do the input element and the output element have the same type associated? Do you want to cast the input value to the type of the output value?
 - When the input and output element have the same type, use the **Move** transform.
 - When the input and output element have different data types, use the **xs:type** transform to set the value of each element.
8. Do you want to apply the transform always? Do you want to apply the transform when a condition based on input data occurs?
Define a conditional expression for the transform you choose. This expression determines when the transform is applied. For more information, see “Defining an XPath conditional expression for a transform” on page 85.

Results

Table 4. Setting the value of a complex output element

	Number of input elements that are required to set the value of the output element	Transforms to set a complex type
Set the output element with a fixed value	0	Assign
Initialize the output element	0	Create
Set the output element as nil	0	Create Condition: The output element is defined as <code>nillable="true"</code> in the schema.

Table 4. Setting the value of a complex output element (continued)

	Number of input elements that are required to set the value of the output element	Transforms to set a complex type
Set the output element as nil by using a nil input element	1	Move Condition: The input element must be defined as <code>nillable="true"</code> in the schema.
Set the output element with a default value	0	Create Condition: The output element has a default value set in the schema.
Set the output value from a database table column	1..N	Select transform to obtain the database input value and Move transform to set the value
Copy the value of the input element to the output element (both elements have the same data type)	1	Move
Copy the value of the input element to the output element (elements have different data types)	1	xs:type

What to do next

Learn about the transforms. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Choosing a transform to map repeating elements

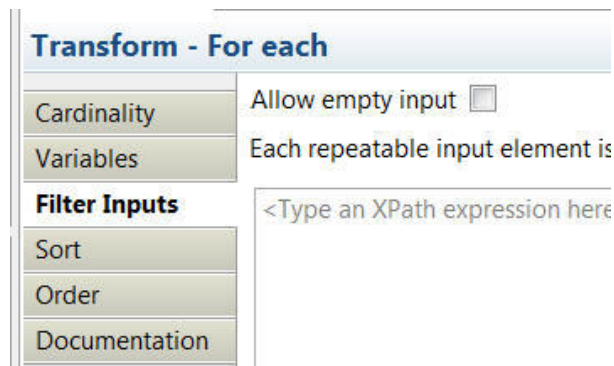
In the Graphical Data Mapping editor, you can use the **For Each** transform, the **Append** transform, the **Join** transform, XPath functions, the **Custom XPath**, and the **Custom Java** transform to map input and output arrays. You can use any of these transforms to choose the set of the elements in the array that you want to transform, and place the result in an output array or single element.

Procedure

Choose one of the following transforms to map repeating elements:

1. When you have repeating elements as input and output to a transform, you can use the **ForEach** transform to set the output array element.

The **For Each** transform iterates over one input array element, which can be either a simple type or a complex type, and enters a nested mapping in which you can provide transforms to populate an instance of the output from the input. You can configure the cardinality to filter which instances to process based on index. Also, you can provide a Boolean expression which will be applied to each instance to determine if it will be mapped.



Additionally, you can also set the **Allow empty input** option so that the Graphical Data Mapping editor enters the nested transform once when no matches occur. This can be used to implement an outer join by providing a supplementary input to the **For Each**.

For more information, see “For Each” on page 218.

2. When you have multiple inputs that are either simple type elements, complex type elements, or repeating elements from which you need to construct an array, you can use the **Append** transform to add instances to either a simple type output array or a complex type output array.

For more information, see “Append” on page 195.

3. When you have multiple inputs that are either simple type elements, complex type elements, or repeating elements as input to a transform, and you want to join these elements, you can use the **Join** transform to combine them into a single repeating output element. The output element can be an array or a single element. You configure an expression to control the match conditions for the join. The Graphical Data Mapping editor provides a link to create a simple match by index expression.

For more information, see “Join” on page 223.

4. You can use XPath functions to map from an array to a single element. For example, you can use **fn:string-join** to return a string created by concatenating multiple string arguments or **fn:sum** to return the sum of a repeating numeric element into a single element.

For more information, see “Built-in XPath transforms” on page 229.

5. When you want to map a particular instance from an array to a single output you can use a **Custom XPath** transform that has an XPath predicate expression to select a particular instance.

6. You can use a **Custom Java** transform to map an array by passing the array as an input or output parameter using a list of MbElement objects.

For more information, see “Custom Java” on page 212.

Selecting the indexes of input array elements

When you are transforming array elements in the Graphical Data Mapping editor, you can use the Cardinality page of the properties view to select the indexes of the input elements that you want the transform to operate over.

To specify the indexes, enter a value in the **Input array indexes** field of the Cardinality properties page for the transform.

The following table shows examples of the values that you can enter:

Table 5. Example values of input array indexes

Selected index elements	Value in input array indexes field
All indexes	* (or leave empty)
Only index 5	5
indexes 1 - 3	1:3
indexes 1, 3, and 5	1,3,5
indexes 2 and up	2:*
indexes 1, 3, 5 and up	1,3,5:*
indexes 2 - 8, but not 5	2:4,6:8
All indexes except 5	1:4,6:*

The indexes are 1-based, which means that the first element of the array is referenced as 1, the second element as 2, and so on. If the cardinality field is left blank for a specified array, all indexes are taken. If you have multiple levels of nested array elements, blank cardinality fields imply that all indexes are taken. Therefore, if the input element to a transform is `A/B[]/C[]`, where B and C are arrays with no indexes specified, all indexes are taken. This means that all C's part of B[1], all C's part of B[2], all C's part of B[3], and so on, are taken.

Choosing a transform to concatenate input data

In the Graphical Data Mapping editor, you can use a **Concat** transform, a **Custom XPath** transform, or an XPath function to define a mapping operation that sets the value of an output element by concatenating input data.

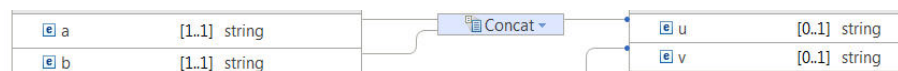
Procedure

Choose one of the following transforms to concatenate multiple input values and set the value of an output element:

- If your input elements are simple non-repeatable elements, and you want to create an output element that is the result of concatenating these input elements with a common delimiter, use the **Concat** transform. The delimiter is optional.

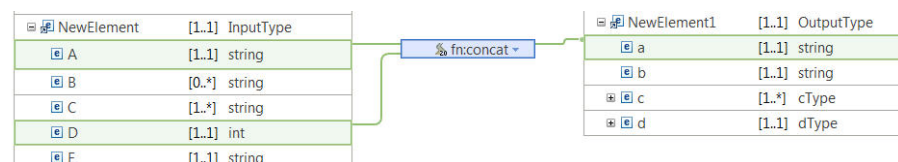
The **Concat** transform concatenates two or more simple non-repeatable elements from the input into a single string value in the output. You can specify a constant to define a prefix, a suffix, and multiple delimiters.

For more information, see “Concat” on page 201.



- If your input elements are simple non-repeating elements, and you want to create an output element that is the result of concatenating these input elements by using an XPath expression with different delimiters, use the **fn:concat** transform. The delimiters are optional.

For more information, see “fn:concat” on page 231.



- If the input is a repeatable simple element, and you want to create an output element that is the result of concatenating the sequence of input elements with a delimiter as optional, use the **fn:string-join** transform.

For more information, see “fn:string-join” on page 234.

The following figure shows the **fn:string-join** transform:

Transform - string-join

General Description: Takes an input sequence of strings and a separator and returns a string created by concatenating the members of the input sequence using the separator as a delimiter. See [XPath 2.0 Specification](#).

Cardinality [1..1]

Variables

Condition

Parameters:

Name	Type	Value
strings	xs:string[]	\$C
separator	xs:string	'xxxxx'

- Use the **Custom XPath** transform to define an XPath expression that concatenates multiple input elements into a simple output type.

The following figure shows a **Custom XPath** transform that concatenates some input elements in the array into a string element:

Transform - Custom XPath

General Description: `fn:string-join($C[fn:string-length() > 4], 'xxxxx')`

Namespaces

Cardinality [1..1]

- If you need to specify the matching criteria for joining or filtering input elements, use the **Join** transform. Then, define transforms between the input and output elements in the nested map associated to the **Join** transform.

For more information, see “Join” on page 223.

Transform - Join

General **Join Expression:** Add a boolean expression to specify the matching criteria for joining or filtering input array items. [More...](#)

Cardinality [1..1]

Variables [Create Join expression based on index](#)

Sort \$F-index = \$G-index

Choosing a transform to perform an arithmetic operation

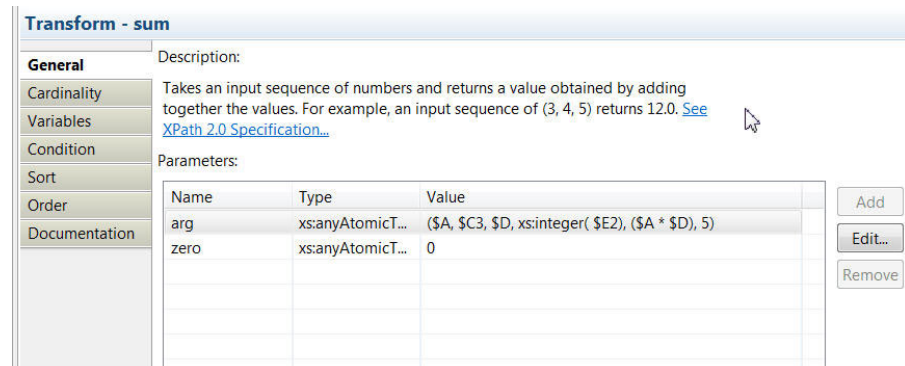
In the Graphical Data Mapping editor, use the `fn:sum` or a **Custom XPath** transform to implement an arithmetic operation.

Procedure

Choose one of the following transforms to implement an arithmetic operation to set the value of an output element:

- Use the `fn:sum` transform to calculate the sum of the values of multiple input elements.

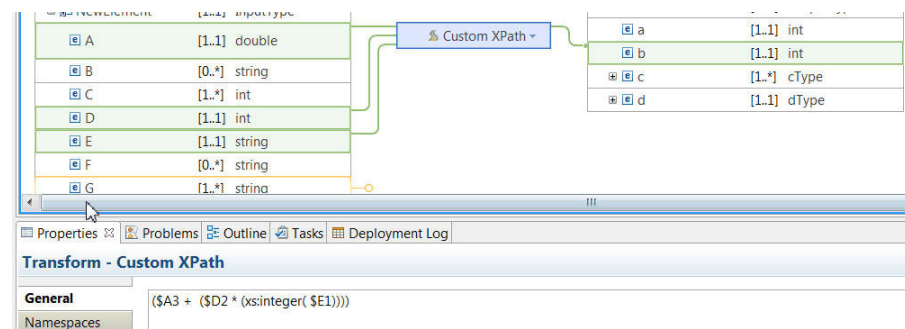
For more information, see “`fn:sum`” on page 241.



- Use the **Custom XPath** transform to implement any of the following arithmetic operations: addition (+), subtraction (-), multiplication (*), division (div), or any combination of these operations.

Note: You can only use as arguments of an arithmetic operation non-repeating simple type input elements.

For more information, see “Custom XPath” on page 214.



Choosing a transform to define a conditional mapping

In the Graphical Data Mapping editor, you can define conditional transformations by using a transform or by defining a conditional expression in the transform.

Procedure

Choose any of the following options to configure a conditional mapping in your map:

1. Use the **If** transform, the **Else if** transform, and the **Else** transform to control the flow of the mapping. For more information, see Configuring an If, Else if, and Else transform in a message map.

If, **Else if**, and **Else** operate as a group of conditional transforms. The condition is applied to the input element of the conditional transform. If the condition is satisfied, the nested transform within the conditional transform is run.

2. Use a **Custom Java** transform or a **Custom ESQL** transform to provide a condition function

For more information, see “Custom Java” on page 212 and “Custom ESQL” on page 208.

3. Define a conditional expression in a transform to determine when the transform is applied by configuring the **Condition** tab of the transform.

For more information, see “Defining an XPath conditional expression for a transform” on page 85.

Note: If you use a condition to optionally create one or more child output elements within a complex element, check that the condition is on the transform targeting the whole complex element, not just a child. If the conditional transform only targets a child, the folder element within the complex element will be created in the output before the condition is evaluated.

What to do next

For more information about mapping transforms, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Choosing a transform to map an input message to multiple output messages

You can create a map that takes a single input message and produces either multiple instances of an output message model, or one or more instances of different output message models.

About this task

When you create a map in the New Message Map wizard, you can only select a single input and a single output. However, you can use the **Add output** button in the Graphical Data Mapping editor to add additional outputs.

Note: You can only split an input message into multiple output messages in main maps. This is not valid in submaps.

Procedure

Choose any of the following options to split a message in your map:

- To produce multiple instances of a particular output message, you can use the **For Each** transform or the **Join** transform.

A typical use of this function is message splitting, in which an input batch message is divided into individual record messages.

When you run the map, a new message is propagated for each iteration of the **For Each** or **Join** transform.

For more information, see Chapter 15, “Splitting an input message into multiple identical output messages,” on page 145.

- To produce one or more instances of different output messages, you can use the **If/Else** transform.

When you run the map, a new message is propagated for each conditional transform that is applied.

For more information, see Chapter 16, “Mapping an input message into different output messages,” on page 151.

Chapter 5. Handling nulls in message maps

A message might contain fields that can carry a specific out-of-range value. This is distinct from the field being empty. Such values are termed *nil* or *null*, and the field is said to be *nillable* or *nullable*.

About this task

The logical message tree supports the concept of out-of-range values by using one of two techniques, depending on the data format:

1. For XML, the schema model allows for elements to be defined as nillable to indicate that they support an out-of-range value. An XML element in a document is identified as being nilled by having a `xsi:nil` attribute with the value `true`. The XMLNSC parser logical tree for a simple nilled element has an empty value and a child attribute set to `xsi:nil` with the value `true`.
2. For other text and binary messages modeled with a DFDL schema or MRM message set, elements can also be defined as nillable to indicate that they support an out-of-range value. In the message bit stream, a reserved value is identified to indicate the nullable state. The DFDL and MRM parsers logical tree for a nilled element have the value set to the special value NULL.

Procedure

When you map nilled values, consider the following behavior when using the **Move** transform, **Custom Java**, or **Custom ESQL** to set a target.

Source	Target	Comment
Logical Message Tree XML nillable element	XML nillable	Target created as nilled, has <code>xsi:nil</code> attribute 'true', if source has <code>xsi:nil</code> attribute 'true'.
Logical Message Tree XML nillable element	Non-XML nillable	Target created with NULL value, if source has <code>xsi:nil</code> attribute 'true'.
Logical Message Tree XML element set to NULL	XML or non-XML nillable	Target created with empty value. A Source XML having NULL value is not considered a nilled element.
Logical Message Tree non-XML	XML nillable	Target created as nilled, has <code>xsi:nil</code> attribute 'true', if source is NULL.
Logical Message Tree non-XML	Non-XML nillable	Target created with NULL value, if source is NULL.
Database nullable column	XML nillable	Target created as nilled, has <code>xsi:nil</code> attribute 'true', if source is SQL NULL.

Source	Target	Comment
Database nullable column	Non-XML nillable	Target created with NULL value, if source is SQL NULL.
Custom ESQL	XML nillable	Target created as nilled, has xsi:nil attribute 'true', if return is ESQL NULL
Custom ESQL	Non-XML nillable	Target created with NULL value, if return is ESQL NULL
Custom Java	XML nillable	Target created as nilled, has xsi:nil attribute 'true', if return is an MbElement with type set to "TYPE_UNKNOWN?" and a value of "null?" and a child element xsi:nil 'true'.
Custom Java	Non-XML nillable	Target created with value NULL, if return is an MbElement with type set to "TYPE_UNKNOWN" and a value of "null"

Definitions:

In the preceding table, *XML* means XMLNSC or XMLNS logical trees. The term *non-XML* means MRM or DFDL logical trees, and applies to all MRM physical formats, including MRM XML.

Chapter 6. Using nested maps

You can edit some types of transforms, involving complex inputs and outputs, by entering a nested view in the Graphical Data Mapping editor.

About this task

Structural transforms control how nested elements are displayed in the Graphical Data Mapping editor. These transforms control the display of nested elements, but they do not affect the data. You can use In and Out arrow buttons and breadcrumb navigation for the nested transforms.

The following transforms can contain nested graphical data maps:

- Local map
- Join
- Append
- ForEach
- If, Else if, and Else
- Database Routine

The elements in a nested map must be mapped in order for the transform to run.

A **Local map** is a navigation aid that you use to view the map elements in a hierarchical way. A local map can have one primary input and multiple supplementary inputs, which can be either a simple type or a complex type. The output can be either a single element or an array element, but it must be a complex type. The local map does not transform data; you must specify transforms for the input and output elements in the nested map.

You can use the **Join** transform to join elements from two or more inputs. The inputs can be arrays or single elements, which can be merged using nested transforms to create a single output. The target element can be an array or single element but must be a complex type.

The **Append** transform iterates over multiple inputs in the specified order to append data. This transform takes multiple inputs of either simple or complex types. The output must be an array of either a simple type or a complex type.

The **For each** transform contains a nested map, and it iterates over one input array element (either a simple type or a complex type). The elements in the nested map must be mapped, otherwise the transform has no effect.

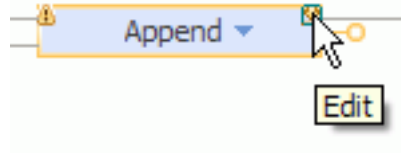
The **If**, **Else if**, and **Else** transforms enable you to control the flow of a mapping by setting conditions. If, Else if and Else operate as a group of conditional transforms, and the condition is applied to the input element of the conditional transform. If the condition is satisfied, the transform that is nested within the conditional transform is run.

The **Database Routine** transform contains a nested map to call a stored procedure or user-defined function from a database schema as input. Output from a **Database Routine** is optional, using the Return transform.

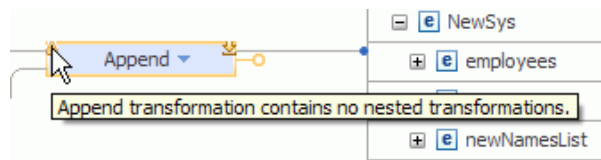
The following steps provide an example of how you can edit a nested map:

Procedure

1. Some transforms, such as **Local map** and **Append**, contain nested maps. If a nested map exists, an **Edit** icon is shown on the transform. Click the **Edit** icon to edit the nested map.



Nested maps must contain transforms, or else when the map is run, nothing happens. If a warning is displayed on the main map, you must edit the nested map.



2. When you open a nested map for some transforms such as **Append**, the nested map might also contain a nested map such as a **For each** transform:

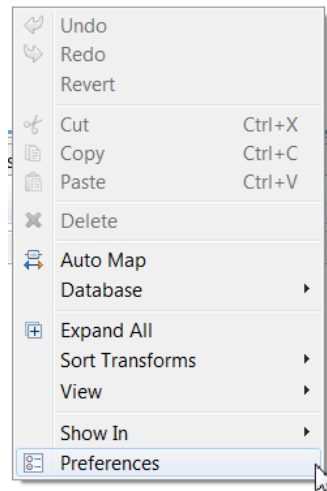


In this case, you cannot create a mapping in the first nested map. Click **Edit** to go down another level and you can then create your mapping, as shown:



What to do next

For more information about mapping transforms, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

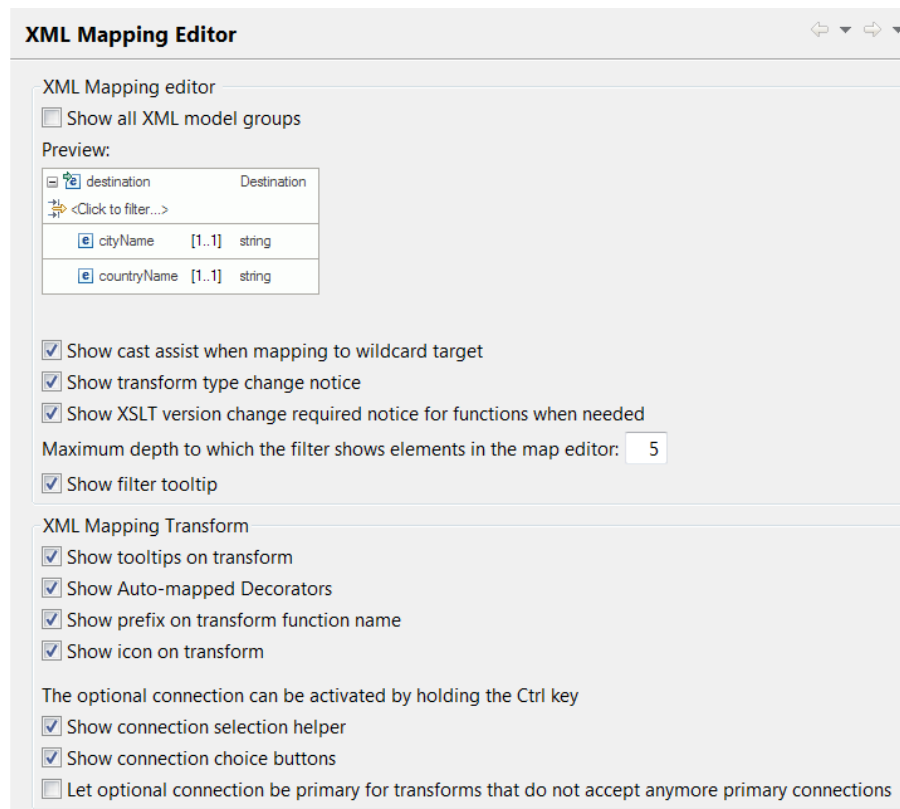


- Select **Window > Preferences > XML > XML Mapping Editor** in the main IBM Integration Studio menu.

- ▲ XML
 - > DTD Files
 - XML Catalog
 - > XML Files
 - > XML Mapping Editor
 - > XML Schema Files

Results

The Preferences wizard opens, and the **XML Mapping Editor** properties are displayed.



What to do next

Create a message map. For more information, see Chapter 8, “Creating a message map,” on page 45.

Setting mapping keyboard preferences for your workspace

You can set your keyboard preferences in the Graphical Data Mapping editor by specifying your choices in the workspace **Preferences** panel.

Procedure

Complete the following steps to change the default mapping keyboard commands, or to add new key combinations based on your usage specification:

1. Select **Window > Preferences > General > Keys** in the main IBM Integration Studio menu.

- ▾ General
 - > Appearance
 - Capabilities
 - > Compare/Patch
 - Content Types
 - > Editors
 - Keys

2. Identify the entries that refer to mapping. Enter Mapping in the **type filter text** box.

Keys

Scheme:

You can see the list of keys related to mapping.

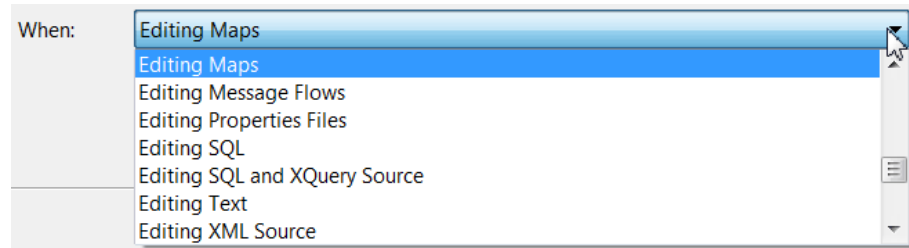
Keys

Scheme:

Command	Binding	When	Category
Add input	Ctrl+Shift+N, I	Editing Maps	Mapping
Add output	Ctrl+Shift+N, O	Editing Maps	Mapping
Add variable	Ctrl+Shift+N, V	Editing Maps	Mapping
Assign			Mapping
Assign transform			Mapping
Associate XML	Ctrl+Shift+X	Editing Maps	Mapping
Auto Map			Mapping
Change connection type			Mapping
Change connection type			Mapping
Collapse	-	Editing Maps	Mapping
Collapse	Ctrl+Shift+V, C	Editing Maps	Mapping
Collapse All			Mapping
Concat			Mapping
Copy	Ctrl+C	Editing Maps	Mapping
Create Connection			Mapping
Custom			Mapping
Cut	Ctrl+X	Editing Maps	Mapping
Expand	Ctrl+Shift+V, E	Editing Maps	Mapping

3. Optional: Select a command, for example, **Assign**.
4. Optional: For each command, complete the following steps to define the key:

- a. Enter the key combination in **Binding**.
- b. Select Editing Maps in the field **When**.



Results

You have created a new key command for the **Assign** transform that you can use when you edit a message map in your workspace.

What to do next

Change or add new commands to specify key actions when using the Graphical Data Mapping editor.

Create a message map. For more information, see Chapter 8, “Creating a message map,” on page 45.

Chapter 8. Creating a message map

You can create a graphical data map, a message map, a submap, or a legacy message map to transform a message without the need to write code, providing a visual image of the transformation, and simplifying its implementation and ongoing maintenance.

About this task

A message map is the IBM Integration Bus implementation of a graphical data map. It is based on XML schema and XPath 2.0 standards. You can use a message map to graphically transform, route, and enrich a message. You can use a message map to modify data in a database system. You can use drag actions to make connections, select transforms, and build logic to transform your message data without programming.

A *submap* is a reusable form of graphical data map. Submaps enable you to use a set of mapping functions in multiple graphical data maps to transform a common set of elements in the input object to the output object. You can use a submap to reuse common data transformations. You can reuse submaps in other products that support graphical data maps.

Note: If you plan to reuse data transformations across different products, read “Guidelines for developing reusable graphical data mapping assets” on page 13.

A *local map* is a subset of data transformations between input elements and output elements that are part of a graphical data map. You define a local map by creating a **Local map** transform in a message map. A local map is not an independent resource. There is no physical file associated with a local map. The scope of a local map is the message map. A local map is processed with the graphical data map. Local maps provide a way of breaking up a large graphical data map into nested groups of mapping elements. You can use local maps to simplify the overall graphical data map presentation. You can structure complex data transformations into nested groups that are easier to manage and implement.

A legacy message map is a message map created as a `.msgmap` file in earlier versions of WebSphere Message Broker Version 8, for example in WebSphere Message Broker Version 7. You can compile and deploy a legacy message map in IBM Integration Bus. However, if you need to modify a legacy message map, you must first convert the legacy message map to a message map. For more information, see Chapter 29, “Using or converting legacy resources into message maps,” on page 321.

Note: WebSphere Message Broker Version 8 introduces graphical data maps. These message maps replace the previous message map format, and are created as `.map` files.

Procedure

Choose any of the following options to create a graphical data map:

- Create a message map to transform you messages graphically. For more information, see “Creating a message map” on page 47.

Use this type of map to create a graphical data map in IBM Integration Bus. You can transform message assembly components, body data, and access a database to retrieve data or modify data, if needed.

- Create a graphical data map to share a map across different products that support the Graphical Data Mapping editor. For more information, see “Creating a graphical data map in the Eclipse editor” on page 59.

Use this type of map to create a graphical data map that you can use as a submap in IBM Integration Bus, or that you can use as a main map in other products that support the Graphical Data Mapping editor. You can only transform the body of a message. You cannot access a database when you use this type of map to enrich a message or to modify database content.

- Create a submap to reuse common transformation logic. For more information, see “Creating a submap” on page 50.

Use this type of map to create a graphical data map that you can use as a submap in IBM Integration Bus. You can only transform the body of a message. You can access a database to enrich a message or to modify database content.

- Create a local map to reduce complexity reading and managing graphical data maps. For more information, see “Creating a local map by using the **Local map** transform” on page 57.

Use this type of map to reduce the complexity of your maps and submaps. This type of map is local, and you cannot reuse it, unless you convert it to a submap.

Results

You can use the following table to identify the type of map that you must create when transforming data graphically in the Graphical Data Mapping editor:

Table 6. Types of map based on design requirements

	Recommended use	Type of resource	Supported in IBM Integration Bus
Message map	Transform messages graphically	.map file	Yes
Submap	Reuse of common data transformations	.map file	Yes
Local map	Reduce complexity reading and managing a Message map	No file. It is embedded within a Message map	Yes
Graphical data map	Share graphical data maps across different software products	.map file	Yes
Legacy message map	Solutions migrated from earlier versions of IBM Integration Bus	.map file	Supported for compatibility with earlier releases of IBM Integration Bus. (See note below.)

Note: You can use a legacy message map, but you cannot modify it in IBM Integration Bus. These type of maps are maintained for compatibility with earlier versions of IBM Integration Bus.

You can see the message map in the Application Development view displayed under a **Maps** category, and organized by namespace.

What to do next

After you create a graphical data map, edit the map, and define transformations between the input message and the output message. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a message map

You can create message maps either in the Application Development view of the IBM Integration Studio, or from a Mapping node in a message flow.

Before you begin

Before you create a message map, you must complete the following tasks:

1. Create an application, a library, an integration service, or an Integration project, as described in the following topics:
 - Creating an application
 - Creating a library
 - Creating an integration service based on a WSDL file or Creating an integration service from scratch
 - Creating an integration project
2. Create a message flow. For more information, see [Creating a message flow](#).
3. Define the message flow content that includes a Mapping node. For more information, see [Defining message flow content](#).

About this task

A message map is a graphical data map.

The physical representation of a message map is a `.map` file.

Procedure

You can use any of the following methods to start the New Message Map wizard and then create a message map:

- Create a message map in the Application Development view. For more information, see [“Creating a message map in the Application Development view”](#) on page 48.
- Create a message map from a Mapping node in a message flow. For more information, see [“Creating a message map from a Mapping node”](#) on page 49.

Results

A message map is created as a `.map` file.

In the Application Development view, the message map is displayed under a **Maps** category. Maps are organized by namespace.

What to do next

Edit the message map, and define transformations between the input message assembly and the output message assembly. For more information, see [“Mapping input to output elements manually”](#) on page 70.

Creating a message map in the Application Development view

You can create a message map for use in your message flows in the Application Development view with messages as input and output objects. Data from database tables can also be used as input to the message map.

Before you begin

Before you create a message map, you must complete the following tasks:

1. Create an application, a library, an integration service, or an Integration project, as described in the following topics:
 - Creating an application
 - Creating a library
 - Creating an integration service based on a wsdl or Creating an integration service from scratch
 - Creating an integration project
2. Create a message flow. For more information, see [Creating a message flow](#).
3. Define the message flow content that includes a Mapping node. For more information, see [Defining message flow content](#).

Procedure

To create a message map in the Application Development view, complete the following steps:

1. Choose one of the following options to start the New Message Map wizard:
 - In the IBM Integration Studio, click **File > New > Message Map**.
 - In the Application Development view, right-click the application, library, integration service, or Integration project where you want to create the message map. Then, click **New > Message Map**.

The New Message Map wizard opens.

2. On the **Specify a new message map file** pane, select **Message map called by a message flow**.

A message map is created that can be accessed from a Mapping node. A message map can contain components of a message body such as global elements and global types. A message map contains Properties, message headers, or the local environment.

3. Specify the **Container**, **Map name**, and broker schema for the message map. Click **Next**.

If your message map is likely to be used by multiple solutions, store it in a shared library.

4. On the **Select map inputs and outputs** pane, select your input type:
 - If you want to use a schema-defined element, expand the list of available input objects, and select the input objects that you want to use as inputs to the message map.

If necessary, use the **Filter map input names** field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: `objectname {namespace}`.

- If you want to use SOAP, JSON, or BLOB messages as input, expand **IBM supplied message models**, and select the message model type.
5. Select your mapping output type:

- If you want to use a schema-defined element, expand the list of available output objects, and select the output objects that you want to use as outputs for the message map.
If necessary, use the **Filter map output names** field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: `objectname {namespace}`.
 - If you want to use SOAP, JSON, or BLOB messages as input, expand **IBM supplied message models**, and select the message model type.
6. Optional: If you are creating a **Message map called by a message flow node**, click **Next** to specify the **Output domain** for the message map.
 7. Click **Finish** to create the message map.

Results

The new message map is created, and the Graphical Data Mapping editor opens with the selected sources and targets.

The message map is created as a `.map` file.

In the Application Development view, the message map is displayed under a **Maps** category. Maps are organized by namespace.

What to do next

Edit the message map, and define transformations between the input message assembly and the output message assembly. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a message map from a Mapping node

You can use a Mapping node to create a message map with messages as input and output objects. Data from database tables can also be used as input to the message map.

Before you begin

Before you create a message map, you must complete the following tasks:

1. Create an application, a library, an integration service, or an Integration project, as described in the following topics:
 - Creating an application
 - Creating a library
 - Creating an integration service based on a WSDL file or Creating an integration service from scratch
 - Creating an integration project
2. Create a message flow. For more information, see [Creating a message flow](#).
3. Define the message flow content that includes a Mapping node. For more information, see [Defining message flow content](#).

Procedure

To create a graphical data mapping (`.map`) file from a Mapping node:

1. In the Integration Development perspective, open your message flow.

2. Double-click a Mapping node that does not have a message map associated with it, or right-click the Mapping node and click **Open Map**. The New Message Map wizard opens.
3. On the **Specify a new message map file** pane, the type of message map that you want to create is selected as **Message map called by a message flow node**. This is a message map that can be accessed from a node.
4. Specify the **Container**, **Map name**, and **broker schema** for the message map, or use the values that have been entered for you by the wizard. Click **Next**.
If your message map is likely to be used by multiple solutions, store it in a shared library.
5. On the **Select map inputs and outputs** pane, select your input type:
 - If you want to use a schema-defined element, expand the list of available input objects, and select the input objects that you want to use as inputs to the message map.
If necessary, use the **Filter map input names** field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: objectname {namespace}.
 - If you want to use SOAP, JSON, or BLOB messages as input, expand **IBM supplied message models**, and select the message model type.
6. Select your mapping output type:
 - If you want to use a schema-defined element, expand the list of available output objects, and select the output objects that you want to use as outputs for the message map.
If necessary, use the **Filter map output names** field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: objectname {namespace}.
 - If you want to use SOAP, JSON, or BLOB messages as input, expand **IBM supplied message models**, and select the message model type.
7. On the **Select the domain to create the output** pane, specify the **Output domain** for the message map.
8. Click **Finish** to create the message map.

Results

The new message map is created, and the Graphical Data Mapping editor opens with the selected inputs and outputs.

A message map is created as a `.map` file.

In the Application Development view, the message map is displayed under a **Maps** category. Maps are organized by namespace.

What to do next

Edit the message map, and define transformations between the input message assembly and the output message assembly. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a submap

You can create a submap from scratch or after you define a **Submap** transform between an input object and an output object in a message map. You can also convert a local map into a submap.

About this task

You can use a submap to reuse common data transformations between input objects and output objects.

Procedure

Choose one of the following methods to create a submap:

- Create a submap from scratch. For more information, see “Creating a submap by using the **Submap** transform” on page 53.
Use this method to create a common transformation between an input object and an output object.
- Create a submap by using the Graphical Data Mapping editor. For more information, see “Creating a submap by using the Graphical Data Mapping editor.”
Use this method when you have a message map where you have identified a common transformation between an input object and an output object.
- Create a submap by converting a local map to a submap. For more information, see “Converting a local map into a submap” on page 54.
Use this method when you have a local map defined in your message map that you have identified as a common transformation between an input object and an output object.

What to do next

Edit the submap and define the transformation logic between the input and output elements in the submap. For more information, see “Specifying a transform (mapping operation)” on page 82.

Creating a submap by using the Graphical Data Mapping editor

Create a submap by defining a **Submap** transform between an input object and an output object in a message map.

About this task

A submap enables you to use the same piece of mapping function in multiple message maps.

Procedure

Complete the following steps to create a submap:

1. Create a connection between global input and output elements in a message map, and then select the **Submap** transform on the connection.
2. In the Properties view of the Submap transform, click **New** to create a submap. The New Message Map wizard is displayed.
3. In the New Message Map wizard, specify a new message map file. Complete the following steps:
 - a. Select **Submap called by another map** as the type of map that you want to create.
 - b. Select a container or create a new one. This is the project where the message map is created.

- c. Enter the map name.
 - d. Optional: Select **Use default broker schema**, or select a broker schema.
4. Click **Next**
 5. In the New Message Map wizard, verify that the map inputs and outputs that are selected are correct.
 6. Click **Next**.
 7. Click **Finish**.

Results

The new submap is displayed in the Graphical Data Mapping editor.

What to do next

Edit the submap. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a submap in the Application Development view

You can create a submap for use in your message flows in the Application Development view with messages as input and output objects. Data from database tables can also be used as input to the message map.

About this task

A submap enables you to use the same piece of mapping function in multiple message maps.

Procedure

Complete the following steps to create a submap in the Application Development view:

1. Choose one of the following options to start the New Message Map wizard:
 - In the IBM Integration Studio, click **File > New > Message Map**.
 - In the Application Development view, right-click the application, library, integration service, or Integration project where you want to create the message map. Then, click **New > Message Map**.

The New Message Map wizard opens.

2. On the **Specify a new message map file** pane, select **Submap called by another map**.

A submap is created that can be referenced from another message map. A submap can contain components of a message body such as global elements and global types. A submap does not contain Properties, message headers, or the local environment.

3. Specify the **Container**, **Map name**, and broker schema for the submap. Click **Next**.
4. On the **Select map inputs and outputs** pane, expand the list of available input objects, and select the input objects that you want to use as input to the submap. If necessary, use the **Filter map input names** field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: objectname {namespace}.
5. Expand the list of available output objects, and select the output object that you want to use as output for the submap. If necessary, use the **Filter map output**

names field to filter what is shown in the list of available objects. Each object in the list is displayed in the form: objectname {namespace}.

6. Click **Finish** to create the submap.

Results

The submap is created, and the Graphical Data Mapping editor opens with the selected sources and targets.

The submap is created as a .map file.

You can see the submap in the Application Development view displayed under a **Maps** category, and organized by namespace.

What to do next

Edit the submap, and define transformations between the input message assembly and the output message assembly. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a submap by using the Submap transform

Create a submap by using the Graphical Data Mapping editor. Create a submap by defining a **Submap** transform between an input object and an output object in a message map.

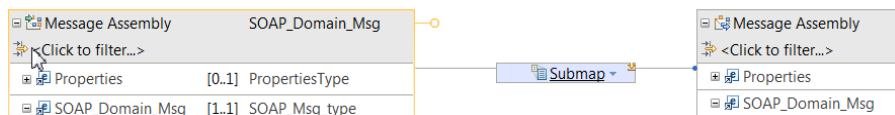
About this task

A submap enables you to use the same piece of mapping function in multiple message maps.

Procedure

Complete the following steps to create a submap:

1. Create a connection between global input and output elements in a message map, and then select the **Submap** transform on the connection: For example:



2. In the Properties view of the Submap transform, create a submap to use an existing submap:
 - To create a submap, click **New**. The New Message Map wizard opens.
 - Use an existing submap. Click **Browse**. A dialogue box will display the submaps that are available. Then, select a submap and click **OK**.
3. On the **Specify a new message map file** pane, the type of map that you want to create is selected as **Submap called by another map**. This is a message map that can be referenced from another message map. This is known as a submap and can contain components of a message body such as global elements and global types. A submap does not contain Properties, message headers, or the local environment tree. Click **Next**.
4. On the **Select map inputs and outputs** pane, the input and output objects of the submap have been pre-selected.

5. Click **Finish**. The new submap is displayed in the Graphical Data Mapping editor, and you can edit it in the same way that you would edit any graphical data map. For information about how to edit maps, see Chapter 9, “Editing message maps,” on page 61.

Results

The new submap is displayed in the Graphical Data Mapping editor.

What to do next

Edit the submap. For more information, see Chapter 9, “Editing message maps,” on page 61.

Converting a local map into a submap

Use the Graphical Data Mapping editor to convert a local map into a submap by using the **Refactor to submap** function.

About this task

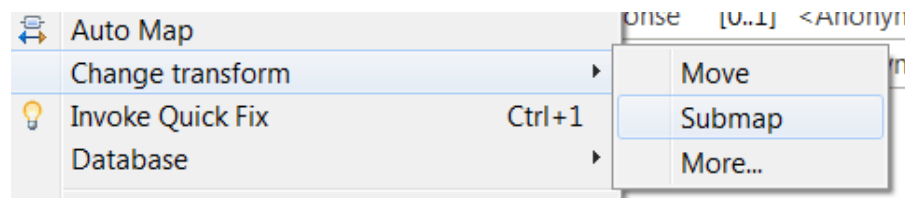
You can convert a local map into a submap so that the transformation logic can be reused by other graphical data maps.

To convert a local map into a submap, the following conditions must be true:

- The input and output elements to the local map must be global elements.
- There must be at least one transform configured in the local map.

If you want to convert a local map to an existing submap, you cannot use the **Refactor to submap** function. You must change the transform from a **Local map** transform to a **Submap** transform.

Note: If you have global elements defined as input and output objects to the local map, and you have not defined any transformations between the input and output objects, you cannot convert a local map into a submap. To create the submap, you can change the **Local map** transform for the **Submap** transform. The following figure shows the menu option that you can use to create a submap instead of a local map:

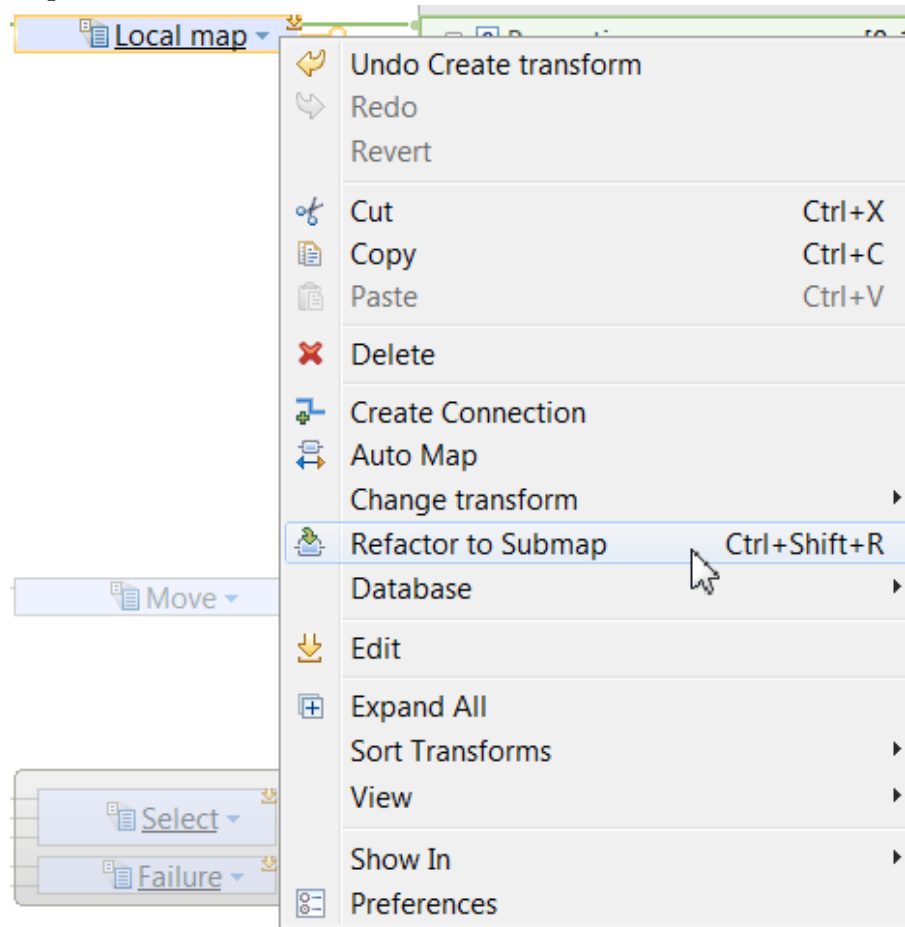


Procedure

To convert a local map into a submap, complete the following steps:

1. Right-click the **Local Map** transform in the graphical data map, and then select **Refactor to submap**.

The following figure shows the menu that opens when you right-click the **Local map** transform:



The

New Message Map wizard opens.

2. On the **Specify a new message map file** pane, the type of map that you want to create is selected as **Submap called by another map**. This is a message map that can be referenced from another message map. This is known as a submap and can contain components of a message body such as global elements and global types. A submap does not contain Properties, message headers, or the LocalEnvironment. Click **Next**.
3. On the **Select map inputs and outputs** pane, the input and output objects of the submap have been pre-selected.
4. Click **Finish**. The new submap is displayed in the Graphical Data Mapping editor, and you can edit it in the same way that you would edit any graphical data map. For information about how to edit maps, see Chapter 9, “Editing message maps,” on page 61.

Results

A submap is created, containing all the mappings from the local map.

What to do next

Edit the submap. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a message map programmatically

Use the Graphical Data Map Specification Language to create a message map programmatically.

About this task

The file `msl.xsd` provides the XML schema that describes the *Graphical Data Map Specification Language*, also known as **MSL**. This file is available in the Graphical Data Mapping component Version 1040, and later versions.

You can find the `msl.xsd` file inside the `com.ibm.msl.mapping.api_7.5.0.jar` jar file.

In IBM Integration Bus, you can find the jar file in any of the following directories:

- In a Windows platform installation: `C:\Program Files\IBM\IIB\10.0.n.0\server\ct\lib\`
- In a Linux platform installation: `install_dir/iib-10.0.n.0/server/ct/lib/`

Procedure

You must perform the following steps to programmatically generate a map file:

1. Create a template map in the Graphical Data Mapping editor.
Deploy and test your template map in the run time to confirm that the message transformation is correct.
2. Inspect the MSL XML content in the template map.
Use the MSL schema to identify the mapping constructs and the definition of the points of variation for the template map.
3. Develop the scripts and programs to generate the MSL XML for the new maps that you plan to generate programmatically.
When you use a development approach based on JAXB, the bindings file `msl_jaxb_bindings.xml` available in the `com.ibm.msl.mapping.api_7.5.0.jar` file provides the minimal required bindings.
4. Validate the syntax of each generated map file (`.map`) against the provided MSL schema `msl.xsd`.
5. Import each generated map file into your development environment. Then, check that all the referenced resources, such as `xsd` files, are imported into the relevant project types. Ensure the relevant builder is invoked to semantically validate each generated map file. Also, check using the Graphical Data Mapping editor that the transforms in your generated map are correct and free of error and warnings.
6. Package and deploy into the run time your programmatically generated maps. Then, test your application to confirm that the message transformation is correct.
Check that each generated map file is built into the relevant deployment artifact.

What to do next

Deploy and test the message map. For more information, see Chapter 25, “Deploying message maps,” on page 191 and Chapter 24, “Troubleshooting graphical data maps,” on page 189.

Creating a local map by using the Local map transform

Create a local map by using the Graphical Data Mapping editor. Create a local map by defining a **Local map** transform between an input object and an output object in a message map.

About this task

You can use a local map to break up a large map into nested groups of mapping elements and process the complex elements of the whole data object.

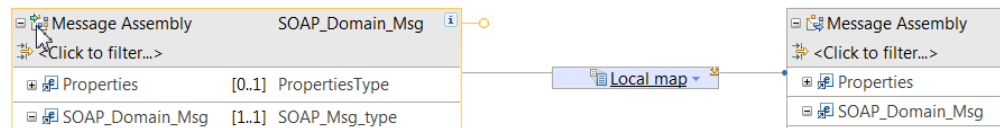
Local maps are a partial view of a larger map, rather than separate files.

A local map has only one element as input (either a simple type or a complex type), which can contain nested elements. The output can be either a single element or an array element, but it must be a complex type.

Procedure

Complete the following step to create a local map:

Create a connection between one input element and one output element in a message map, and then select the **Local map** transform on the connection: For example:



Results

The local map opens within the main map in the Graphical Data Mapping editor.

What to do next

Edit the local map. For more information, see Chapter 9, “Editing message maps,” on page 61.

Converting a submap into a local map

You can use the Graphical Data Mapping editor to change a submap into a local map by using the **Refactor from submap** function.

About this task

You can customize the logic of a submap in a graphical data map, without altering the submap logic, by converting the submap into a local map.

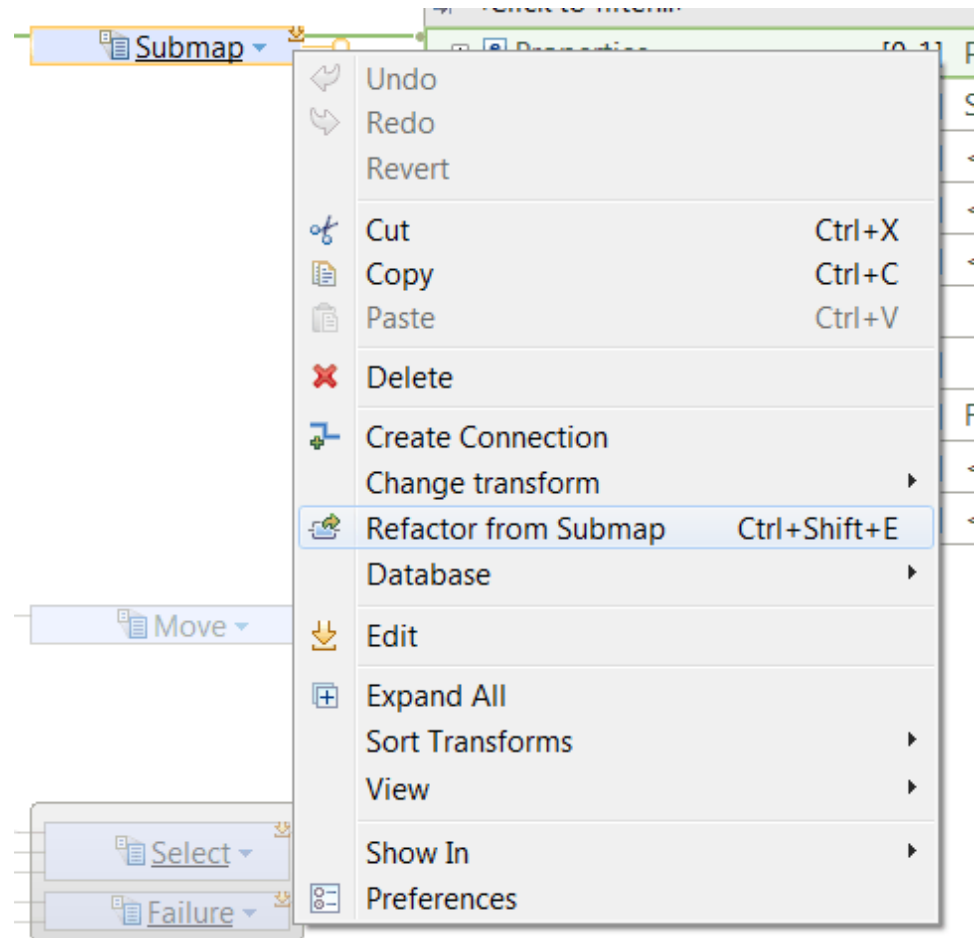
For example, you have a submap that contains common transformation logic that cannot be deleted. The submap is used by multiple graphical data maps. For one application, you need to specialize the submap, so rather than starting from scratch and creating a graphical data map, you can pull the submap logic into a main map by converting the submap to a local map. The action leaves the submap in place for ongoing use in other maps.

Procedure

To change a submap into local map, complete the following steps:

1. Right-click the Submap transform in the graphical data map, and then select **Refactor from submap**.

The following figure shows the menu that opens when you right-click the **Submap** transform:



2. Optional: Delete the map file corresponding to the submap that you have converted to a local map if the submap is not being used anywhere else in your solutions. For more information, see “Deleting objects and transforms” on page 94.

Results

A local map is created, containing all the mappings that were included in the original submap.

What to do next

Continue editing the graphical data map. For more information, see Chapter 9, “Editing message maps,” on page 61.

Creating a graphical data map in the Eclipse editor

Create a graphical data map in the Eclipse editor to transform data graphically so that you can reuse it in other products that support graphical data maps.

Before you begin

Before you create a reusable message map, you must complete the following tasks:

1. Create an application, a library, an integration service, or an Integration project, as described in the following topics:
 - Creating an application
 - Creating a library
 - Creating an integration service based on a WSDL file or Creating an integration service from scratch
 - Creating an integration project
2. Create a message flow. For more information, see [Creating a message flow](#).
3. Define the message flow content that includes a Mapping node. For more information, see [Defining message flow content](#).

If you plan to reuse the graphical data map in other products, read “Guidelines for developing reusable graphical data mapping assets” on page 13.

About this task

In IBM Integration Bus, you can create a graphical data map as a reusable message map. You handle a graphical data map as a submap.

Procedure

To create a graphical data map in the Eclipse editor, complete the following steps:

1. Create a new graphical data map, by selecting **File > New > Other > XML > XML Mapping**. Click **Next**.
The New XML Mapping wizard opens.
2. Select the parent folder for the new graphical data map and specify a file name. By default the new file is called `NewMAP.map`. Click **Next**.
The Input and Output Roots pane is displayed.
3. Select the input and output root files by clicking **Add** and then browsing to the required schema files.

Note: If you choose not to select the input and output schemas at this point, you can select **Next**, and the Graphical Data Mapping editor is displayed with no input or output objects. You can then select your input and output objects in the Graphical Data Mapping editor by using the **Add an input object** and **Add an output object** icons.

When you have specified your input and output objects, they are shown in the Graphical Data Mapping editor with the input object on the left side of the canvas, and the output object on the right side:

4. Click **Next**.
5. Click **Finish**.

What to do next

Edit the graphical data map, and define transformations between the input message and the output message. For more information, see Chapter 9, “Editing message maps,” on page 61.

Chapter 9. Editing message maps

You edit a message map in the Graphical Data Mapping editor.

Before you begin

Create a message map using the Graphical Data Mapping editor. For information about how to do this, see “Creating a message map” on page 47.

About this task

In the Graphical Data Mapping editor, you create a map, you define the input and output message models to the map, and the transformations that must be applied to create the output message with values from the input elements. When your input or output message model includes an **xsd:any** element, you must qualify this extension point by using the **Cast** function prior to defining transforms.

You use the Graphical Data Mapping editor to map (or connect) elements of input objects to elements of output objects. Then, for each mapping, you can create a **transform**, which performs an action on the data of the input element and puts the result in the output element. The input objects are on the left side in the Graphical Data Mapping editor, and the output objects are on the right.

You can define XPath conditional expressions on a transform. This expression determines whether the transform is applied. Use content assist to set the required parameters of the expression. For more information, see “Using content assist (Mapping syntax)” on page 93.

Procedure

You can do any of the following editing tasks in the Graphical Data Mapping editor:

1. Optional: Configure the general properties of a message map to define the Java, ESQL, and XSD resources that the map can refer to; to define your solution XML namespaces, and to add any documentation. For more information, see “Configuring the general properties of a message map” on page 62.
2. Redefine **xsd:any** elements in your message.
 - a. Use the **Cast** function to redefine parts of the input or output model in a graphical data map by using a schema model. For more information, see “Mapping **xsd:any** on an input or output message” on page 64. For more information, see “Mapping **xsd:any** on an input or output message” on page 64.
3. Map the input and output elements in any of the following ways:
 - a. Map elements manually. For more information, see “Mapping input to output elements manually” on page 70.

Use this method to select the input and output elements, create connections between them, and specify the required transforms.
 - b. Map elements automatically. For more information, see “Mapping input to output elements automatically” on page 73.

Use the Auto map wizard to map elements by examining the names of input and output elements to create the mappings.

4. Specify a transform, also known as a mapping operation. For more information, see “Specifying a transform (mapping operation)” on page 82.
5. Configure the properties of a transform. For more information, see “Configuring the properties of a transform” on page 84.
6. Delete objects and transforms. For more information, see “Deleting objects and transforms” on page 94.

What to do next

Configure the input and output message assembly in a message map by using the Graphical Data Mapping editor. For more information, see Chapter 10, “Advanced editing in a message map,” on page 95.

Configuring the general properties of a message map

You can configure the general properties of a message map in the Graphical Data Mapping editor.

About this task

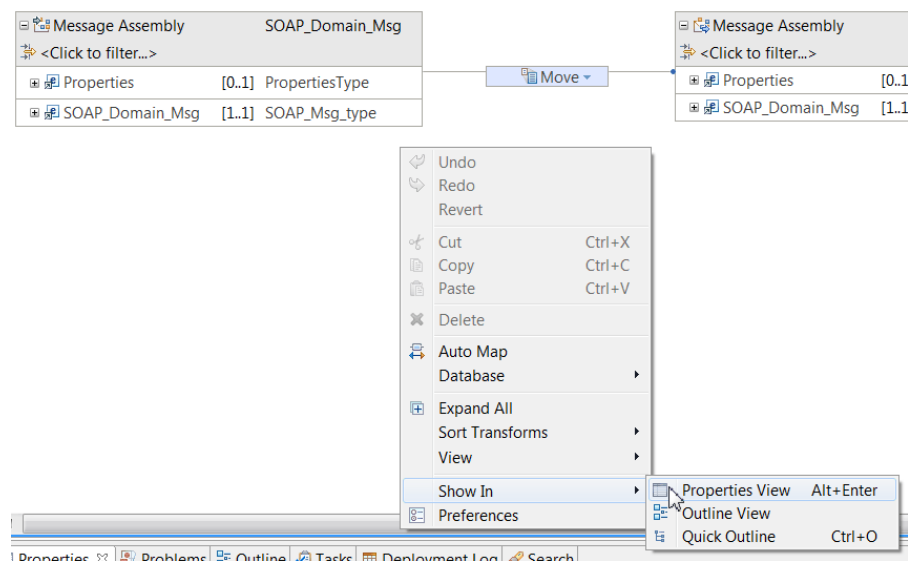
You can optionally configure the general properties of a message map:

- You can add documentation.
- You can configure prefixes for your XML namespaces.
- You can explicitly define references to Java, ESQL, and XSD resources that the map can refer. Note the Graphical Data Mapping editor will normally add these automatically while editing.

Procedure

Complete the following steps to configure the general properties of a map:

- Open the **Properties** tab of a map by using any of the following methods:
 - Select a map by focusing on the map canvas, rather than on an individual item in the map. Then, select the **Properties** tab.
 - Right-click the map canvas, and then select **Show in > Properties view**.



- Use the appropriate keyboard combination, which, by default, is Alt+Enter.

You can configure these shortcut keys by selecting **Window > Preferences > General > Keys**.

- In the **Properties** tab, add, modify, or remove resources:

1. Select the **General** tab to view the namespace where the map is available.
2. Select **Java Imports** to add or remove Java classes that Custom Java transforms can refer to in the message map.

When you include a Custom Java transform, an import is added to refer to the package qualified Java class, defining a prefix based on the class name. If you need to use custom Java only in condition or filter expressions, you can add Java imports to your Java class so that the class public static methods are available through content assist when you are composing an expression.

3. Optional: Select **ESQL Imports** to add or remove ESQL files that **Custom ESQL** transforms can refer to in the message map.

When you include a **Custom ESQL** transform, an import is added to refer to the ESQL file, defining a prefix based on the file name. If you need to use custom ESQL only in condition or filter expressions, you can add ESQL imports to your ESQL file so that the applicable modules are available through content assist when you are composing an expression.

4. Select **Scope** to add or remove XSD files that you can refer to in the message map.
5. Select **Cast** to obtain the list of input and output wildcard elements that are cast to a specific type or global element in the message map. You can remove any entry that is not required anymore.
6. Select **Namespaces** to add, edit, or remove user-defined namespaces in the message map.
7. Select **Documentation** to provide a description of the message map, or other relevant usage notes.

What to do next

Continue editing the map, and define transformations between the input message and the output message. For more information, see Chapter 9, “Editing message maps,” on page 61.

Adding input and output messages

Use the Graphical Data Mapping editor to add input and output messages to your message map.

Before you begin

Create a message map by using the Graphical Data Mapping editor. For more information, see “Creating a message map” on page 47.

About this task

You can add an input object to your message map by using the **Add an input object** icon in the toolbar of the Graphical Data Mapping editor:



Note: In IBM Integration Bus, you can only add one input object to a message map.

You can add an output object to your message map by using the **Add an output object** icon in the toolbar:



Note: In IBM Integration Bus, you can only add one output object to a message map.

What to do next

When your message map contains all of the required input and output objects, create the connections between them, as described in “Adding connections between input and output elements” on page 70.

Mapping xsd:any on an input or output message

You can use the **Cast** function to redefine parts of the input or output model in a graphical data map.

In your integration solution, you can create a generic message model, which you can later redefine to a specific model, by using a wildcard, defined as `xsd:any`.

You can redefine an `xsd:any` element in any of the following ways:

- Using the **Cast** function. You can redefine an `xsd:any` element by specifying the specific complex or global type defined in a particular schema file.
- Using a transform. You can define a transform, such as the **Submap** transform, and define the input and output `xsd:any` elements within the nested map of the transform.

Note: It is recommended to qualify an `xsd:any` element before you define any transforms in a main map. Alternatively, you can define a **Submap** between the `xsd:any` element and the output element, and then define the transforms within the nested map associated to the **Submap**.

For example, in IBM Integration Bus, a SOAP message is a common example of a generic model in which you are required to define the business data being exchanged through the SOAP protocol. The predefined SOAP message format defines only the structure of the SOAP envelope and allows you to redefine the Header and Body content.

Note: The `xsd:any` input element cannot be involved in a transformation when it is contained within a cast item group. You can either create transformations on the cast elements or remove all associated cast elements to directly transform the `xsd:any`.

Qualify `xsd:any` parts of a schema-based message model by using the **Cast** function

You can define a wildcard in a schema-based message model as an `xsd:any` element to create a flexible message model that can be redefined later.

You use the **Cast** function to redefine parts of the input or output model in a message map.

For example, you might have a base type of `AddressType`, and two derived types of `USAddressType` and `CanadianAddressType`. Using the **Cast** function in the Graphical Data Mapping editor, you can cast `AddressType` to `CanadianAddressType`.

For more information, see “Casting wildcards in a map.”

Qualify `xsd:any` parts by using a **Submap** transform

You can use the **Submap** transform to qualify an `xsd:any` element defined on the input message assembly, the output message assembly, or both.

You must specify the input element type and the output element type using a global type in the referenced submap of the **Submap** transform.

You qualify the `xsd:any` elements in the nested map by defining the input and output elements.

Casting wildcards in a map

Use the **Cast** function to redefine parts of the input or output model in a graphical data map.

About this task

Your message model schemas might contain one or more wildcards, defined as `xsd:any`. Wildcards can be used to create a flexible message model that can be redefined when a more detailed definition is required. The process of redefining is called a *cast*.

To cast a wildcard element by using the Graphical Data Mapping editor, complete the following steps:

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click the base element or wildcard (`xsd:any`) element that you want to cast, then select **Cast**. The Type Selection dialog opens.
2. In the Type Selection dialog, select the type that you want to cast to, and then click **OK**. The Type Selection dialog lists only those elements and types that are appropriate for the element that you want to cast, and that are contained in a referenced application or library. When casting a base element, the Type Selection dialog lists only derived types.

Results

Your element is cast to the type that you selected, and is displayed in the Graphical Data Mapping editor.

Casting a wildcard defined as `xsd:any` into a specific type for a SOAP message

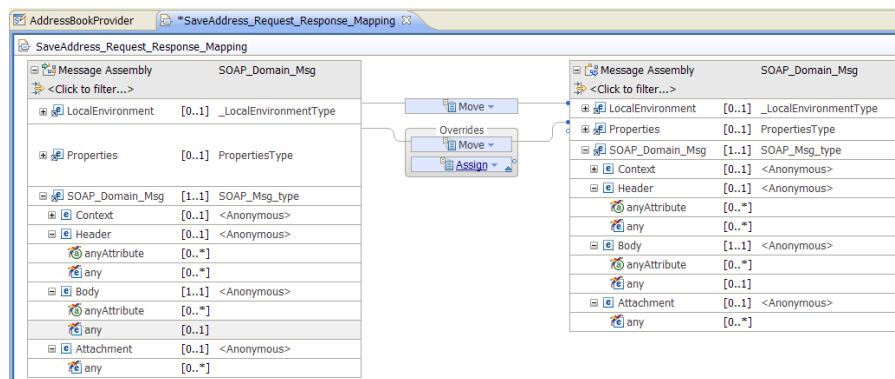
You can use the **Cast** function to redefine a wildcard element, that is, an element with type `xsd:any`, into a specific type. Each type is described by a schema.

About this task

You can transform a SOAP message that is defined by using the predefined format **SOAP_Domain_Msg**. This message type contains a Header, a Body, and an Attachment part. Each part contains an element that is named **any** to represent a wildcard, that is, an element of type *xsd:any*. The Header and Body sections also include an element that is named **AnyAttribute**. You can cast elements and attributes included in any of these SOAP sections by using the **Cast** function.

Note: When you transform a SOAP message, you cast the Body wildcard on the input side to the type of the request message for the SOAP operation. On the output side, you cast the Body wildcard to the type of the response message for the SOAP operation.

The following figure shows the message map in the Graphical Mapping Data editor after you create a message map to transform a SOAP message:

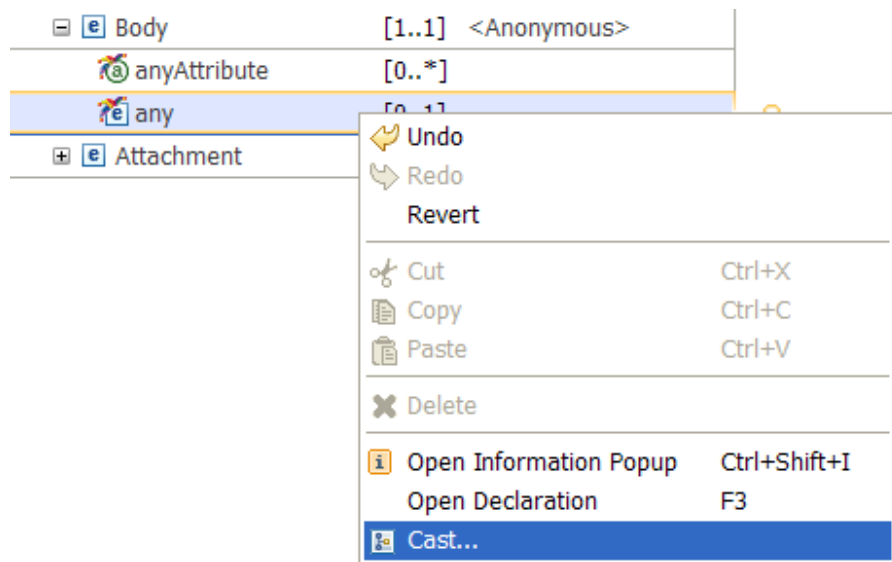


Procedure

To cast an element that is described as **any** or as **anyAttribute** in the message map, complete the following steps:

1. Right-click the element **any** or **anyAttribute** located in the section of your **SOAP_Domain_Msg** where you want to specify a type, and then select **Cast**.
 - To cast a SOAP header wildcard element, right-click **Header**, and then select **Cast**.
 - To cast a SOAP body wildcard element, right-click **Body**, and then select **Cast**.
 - To cast a SOAP attachment wildcard element, right-click **Attachment**, and then select **Cast**.

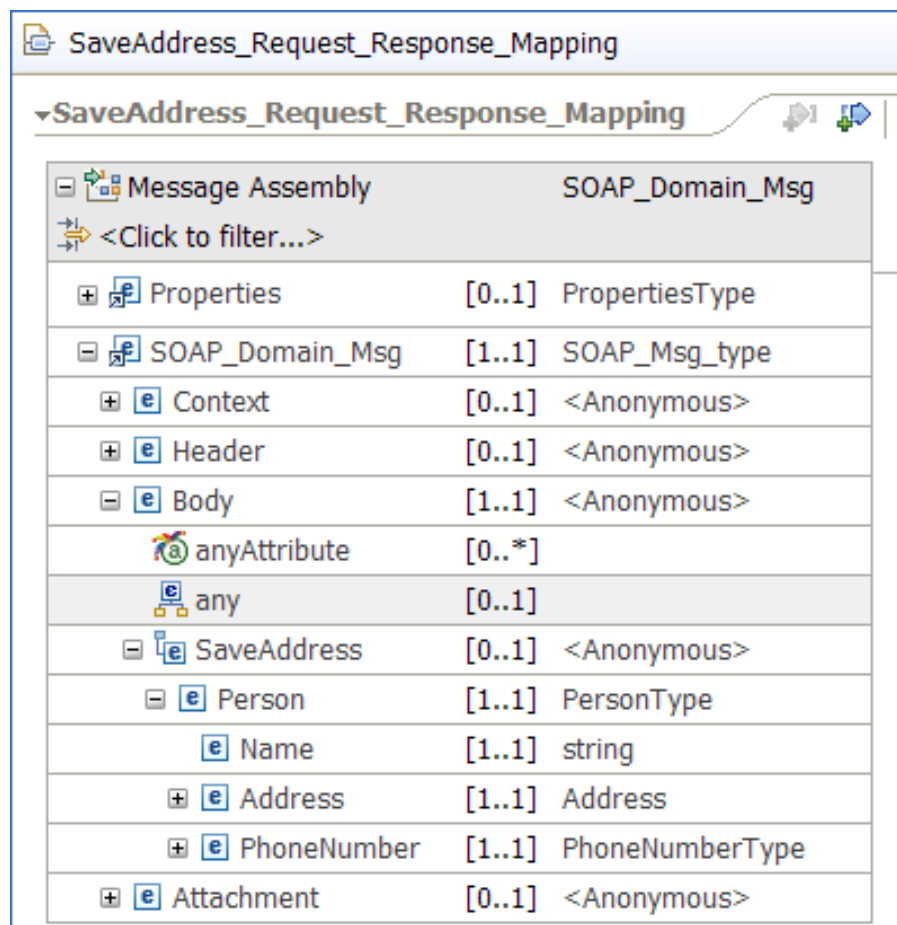
For example, to cast the Body section, right-click **Body**, and then select **Cast**.



2. In the Type Selection window, select a type.

The Type Selection window displays all the specific types that are available for selection. These types include the input and output elements that are defined in the WSDL file that describes your SOAP message.

In the example, the element **any** of the **SOAP_Domain_Msg** Body is redefined to the complex element **SaveAddress**.



Results

A wildcard element is redefined to a specific type.

What to do next

Define transformations between the input message assembly and the output message assembly. For more information, see “Specifying a transform (mapping operation)” on page 82.

Casting a base type to a derived type or extension type

In a message map, you can cast a base type to a derived type or extension type so that you can define transformations between subtypes of a data type.

Before you begin

- Model the schemas that correspond to the base type and the derive type.
- Cast an `xsd:any` element in your message map to a base type.

About this task

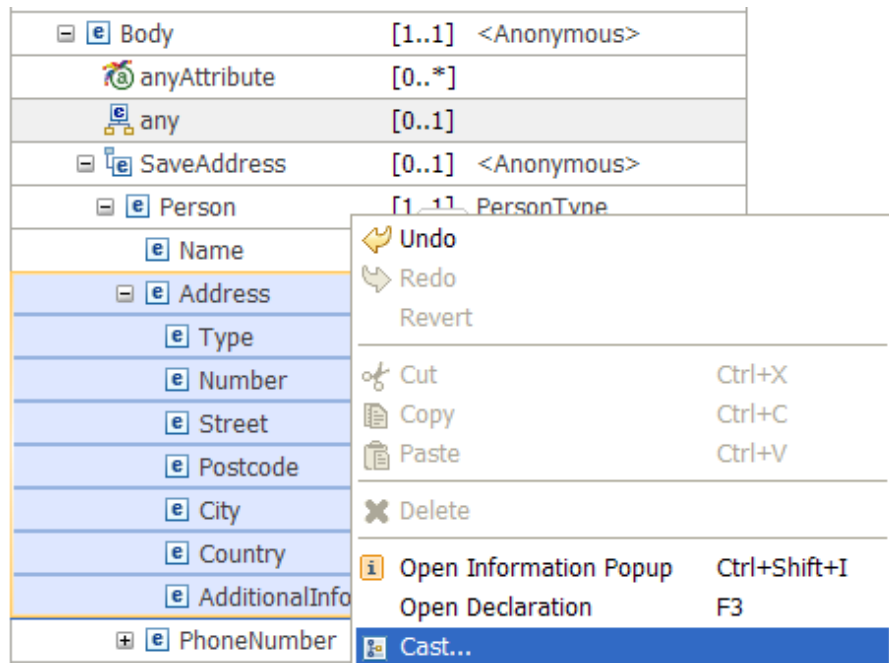
A *derived type* is a data type that is related to another data type known as the base type or super type.

For example, **Address** is the base type, and **USAddress**, **CanadianAddress**, and **UKAddress** are derived types of **Address**.

Procedure

To cast a base type to a derived type, complete the following steps:

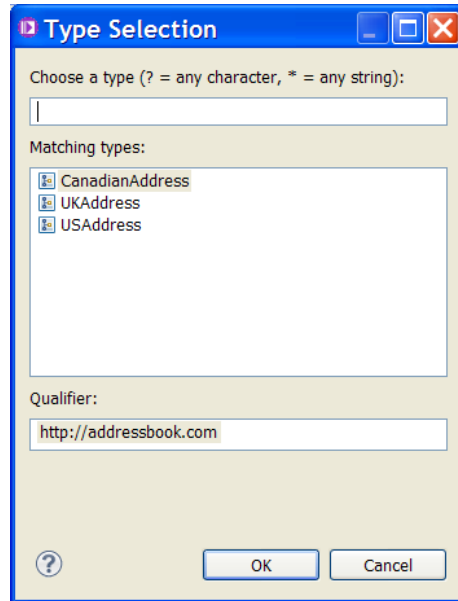
1. Select the base type.
For example, you can cast an `xsd:any` element to the **Address** base type.
2. Right-click the base type (**Address**), and then select **Cast**.



3. In the Type Selection window, choose a matching type, and then select **OK**.

The options available correspond to specific address types in the schema model that are modeled by using **Address** as the base type.

The following figure shows the Type Selection window that you get:



Results

The message map contains two entries, one for the base type and a second one for the derived type.

In the example, one entry corresponds to the base type **Address**. The other entry corresponds to an Address with the derived type **CanadianAddress**.

SaveAddress	[0..1]	<Anonymous>
Person	[1..1]	PersonType
Name	[1..1]	string
Address	[0..1]	Address
Type	[1..1]	string
Number	[1..1]	integer
Street	[1..1]	string
Postcode	[1..1]	string
City	[1..1]	string
Country	[1..1]	string
AdditionalInfo	[1..1]	string
Address	[0..1]	CanadianAddress
Type	[1..1]	string
Number	[1..1]	integer
Street	[1..1]	string
Postcode	[1..1]	string
City	[1..1]	string
Country	[1..1]	string
AdditionalInfo	[1..1]	string
province	[1..1]	string
postcode	[1..1]	string
PhoneNumber	[1..1]	PhoneNumberType

What to do next

Define additional transformations between elements in the message map. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Mapping input to output elements manually

Use the Graphical Data Mapping editor to map from input to output elements.

Before you begin

- Create a message map by using the Graphical Data Mapping editor. For more information, see “Creating a message map” on page 47.
- Add input and output objects to a message map. For more information, see “Adding input and output messages” on page 63.

About this task

Note: You can also use the Graphical Data Mapping editor to map automatically from input to output elements. For more information, see “Mapping input to output elements automatically” on page 73.

Procedure

To create manually the mappings between elements in a message map, complete the following tasks:

1. Add connections between the input and output objects. For more information, see “Adding connections between input and output elements.”
2. Optional: Connect multiple input elements to a transform. For more information, see “Connecting multiple input elements to a transform” on page 71.
3. Create the transforms. If the transform that you have selected is a type that is part of a nested transform (for example, Local map, Join, If), you must enter into that nested map and complete the transformation in it. For more information, see “Specifying a transform (mapping operation)” on page 82.
4. Set the properties for the transforms (for example, Condition, Cardinality, or Order).

What to do next

When you have added your input and output objects, created the connections between them, and specified your transforms, you can test your message map. For more information, see Chapter 24, “Troubleshooting graphical data maps,” on page 189.

Adding connections between input and output elements

Use the Graphical Data Mapping editor to create connections between the input and output objects in your message map.

Before you begin

- Read the following concept topics:
 - Chapter 2, “Graphical Mapping overview,” on page 3

- Create a message map by using the Graphical Data Mapping editor. For more information, see “Creating a message map” on page 47.
- Add input and output objects to your message map. For more information, see “Adding input and output messages” on page 63.

Procedure

You can create a connection in a message map by using one of the following methods:

- Move the cursor anywhere over the required element in the input object, then click and drag the connection over the required output element and drop. You can also create the connection from the opposite direction, by dragging the connection from the output element to the input element. You can also use the grab handle that appears when you hover over an element to click and drag the connection.
- Select the appropriate input or output object and right-click to display the context menu, then select **Quick Link**. An outline view of the opposite data structure is displayed; if the **Quick Link** action is invoked on an input object, a quick outline view of the output data structure is shown, and if the action is invoked on an output object, a quick outline view of the input data structure is shown. You can then use the quick outline view and its built-in filter to find and select the required element. When you have selected the required element, a transform is created in the Graphical Data Mapping editor.
- Select the required input element and right-click to display the context menu, then select **Create connection**. This method allows you to carry out other actions in the editor that require the use of mouse clicks (such as expanding output elements, or using the scroll bars) while a transform is being created.
- Some transformations, such as the **Assign** transform, require a connection to an output object, but not to an input object. You can create these transformations either by invoking the **Create Connection** action, or by clicking anywhere on the output object or on its **grab handle** icon and then dragging the connection onto the empty space between the input and output objects.

What to do next

When you have created the connection between your input and output objects, select the required transform as described in “Specifying a transform (mapping operation)” on page 82.

Connecting multiple input elements to a transform

To make additional input data available to the transform, you can specify a secondary connection to a transform. You must define the connection type as **supplement connection** for additional connections to a **Join** transform or to a **ForEach** transform. You must define the connection type as **primary connection** for additional connections to any of the other transforms.

Before you begin

- Read the following concept topics:
 - Chapter 2, “Graphical Mapping overview,” on page 3
- Create a message map by using the Graphical Data Mapping editor. For more information, see “Creating a message map” on page 47.
- Add input and output objects to your message map. For more information, see “Adding input and output messages” on page 63.

About this task

When you create additional input connections to a transform, you must choose the connection type.

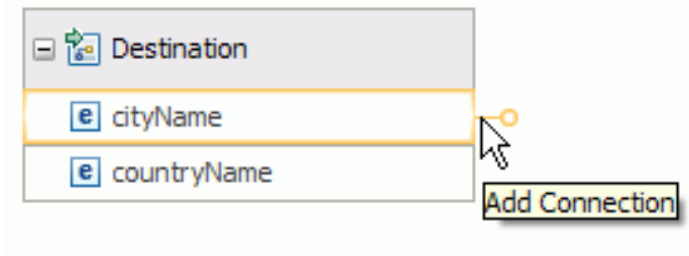
- *Primary connection:* You define a **primary connection** when you require multiple inputs to a transform.
- *Supplement connection:* You define a **supplement connection** to make additional sources of data available to a **Join** transform and to a **For Each** transform only.


You can also change the preferences of the Graphical Data Mapping editor. You can set the **Let optional connection be primary for transforms that do not accept anymore primary connections** preference.

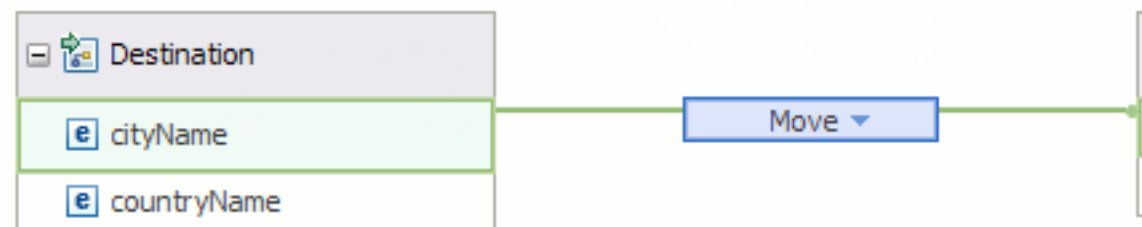
Procedure

The following steps show how to connect multiple input elements to a transform by using the drag and drop method:

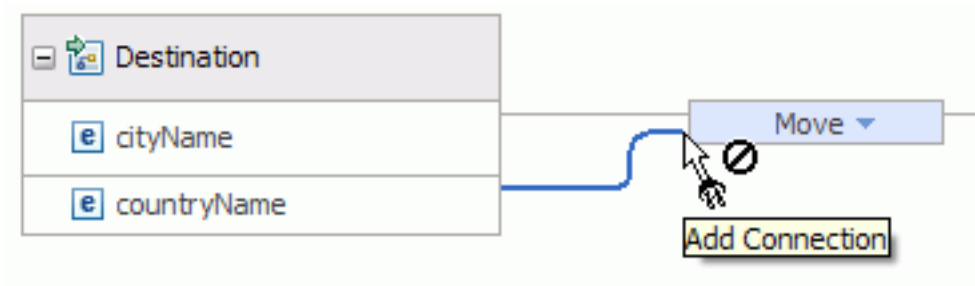
1. Move your cursor to the element of the input object that you want to map:



2. Click anywhere in the input element, or on its **grab handle** icon , and drag the connection to the output element. A connection is created between the two elements, and a transform is assigned, based on the number and type of input elements.



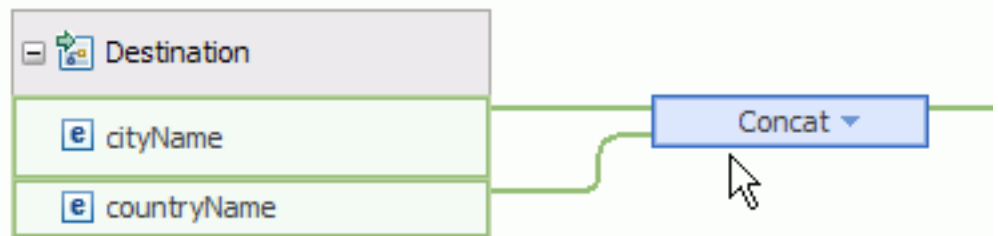
3. Create a new connection between a second input element and the transform.



When you define an additional connection between a second input element to an existing transform in your message map, you are prompted to specify if the connection is a primary connection or if it is a secondary connection (also known as a supplement connection).

4. Specify the type of the new connection as either a primary connection or a secondary connection (Supplement). If you specify a primary connection, the default transform changes to one that allows multiple primary inputs, such as Local Map, Concat, or Join.

For example:



You must choose **supplement connection** for additional connections to a **Join** transform, or to a **For Each** transform.

What to do next

When you have created the connection between your input and output objects, select the required transform as described in “Specifying a transform (mapping operation)” on page 82.

Mapping input to output elements automatically

Use the Auto map wizard to automatically create **Move** transforms from input to output elements based on some correlation of the names of input and output elements to create mappings.

About this task

You can use Auto map as a quick fix when you create a **Local map** or another nested mapping, and a warning or error marker is displayed to prompt you to complete the nested mapping.

Procedure

The following steps describe how to map from input to output elements by using the Auto map wizard:

1. In the Graphical Data Mapping editor, select the input and output elements that you want to map.
 - If you make no selections, the entire input and output objects in the current mapping level are processed by the Auto map wizard.
 - You can select groups of simple elements or complex elements. For complex elements, the descendants of the selected complex element can be mapped.
2. Click the Auto map icon. The Auto map wizard opens.
3. Choose the appropriate **Mapping scope** option:
 - **Map all simple descendants of the selected elements.** This option maps the descendants of the input element to the descendants of the output element that match each other; this option is selected by default. Optionally, select

Group transforms into nested maps. For more information about using nested maps, see “Grouping transforms into nested maps” on page 92.

- **Map the immediate children of the selected elements.** This option maps only the immediate child elements of the input element to the immediate child elements of the output element that match each other.
4. Specify how names are matched in the **Name Matching Options** section.
 - a. Select the **Case sensitive** option to set whether you want to match the case sensitivity of the name. This option is not selected by default.
 - b. Select the **Alphanumeric characters (Letters and digits only)** option to exclude special characters (for example & and -) from the name. This option is selected by default.

These two options are independent of each other, and you can select their values separately.

5. In the **Mapping Criteria** section, specify the mapping criteria for the matches between the input and output element names:
 - **Create transforms when the names of inputs and outputs are the same.** This option matches items of the same name, and is selected by default. Whether the two names are considered to be the same, depends on your selections for **Case sensitive** and **Alphanumeric characters (Letters and digits only)**. For example, if you use the default options for **Case sensitive** and **Alphanumeric characters (Letters and digits only)**, GIVEN_NAME and GivenName are considered to be a match. However, if you select **Case sensitive** and **Alphanumeric characters (Letters and digits only)**, the two names are considered to be identical only if they contain the same alphanumeric characters in the same order, and the characters are of the same case.

For more information, see “Mapping by same name” on page 75.

- **Create transforms when the names of inputs and outputs are more similar than.** With this option, you can specify how similar two names must be to create a mapping between them by varying the result from zero to 100 percent. The result is displayed and the default value is 60; see “Examples of similarity values” on page 77 for some examples of how similar words are matched to one another.

For more information, see “Mapping by similar name” on page 76.

- **Create transforms when the input and output names are matched to synonyms defined in a file.** With this option, you can create mappings for word pairs that are defined in a synonym file. A synonym file is a flat text file with file extension .txt or .csv. For further information about creating a synonym file from a Microsoft Excel spreadsheet, see “Creating and using a synonym file” on page 81.

For more information about the synonym file, see “Format of the synonym file” on page 77. For information about the methods that are used to match synonyms in a synonym file, see “Algorithm used to match synonyms” on page 80.

If the input and output names that you are using satisfy more than one of the following options, the order in which names are matched is:

- a. Same
- b. Synonym
- c. Similar

Any output that is matched during an earlier step is excluded from name matching in a later step.

6. Click **Finish** to start the auto mapping, or click **Next** to view a summary of the transforms that were found and to choose the transforms that you want to create. You can clear matches that are defined on the summary page. Sometimes an input might match to more than one output; use the summary page to review and choose which mappings to create. The auto mapping is performed, and the selected matched input and output elements are mapped through **Move** transforms. Each auto mapped **Move** transform is annotated to indicate that it was created as a result of auto mapping. Hover the mouse or right-click the annotated transform to accept or reject the automatically created transforms.

Mapping by same name

Learn about the rules that apply when you select in Auto map the **Create transforms when the names of inputs and outputs are the same** option.

About this task

When you select **Create transforms when the names of inputs and outputs are the same**, the following rules apply:

1. Any output field that has a fixed value is excluded in name matching. Any output that is already mapped, or under a container that is already mapped, is excluded from name matching.
2. If an input and an output have the same name, it is a match, regardless of the kind of, and XSD type of, the input and output. An element, an attribute, and a database column can all form a match if their names are the same.
3. XML namespaces are excluded from name matching. Therefore, `abc:something` and `xyz:something` are considered the same, as are `{http://www.abc.com}:something` and `{http://www.xyz.com}:something`.
4. When multiple inputs have the same name as one output, one mapping is created.

However, if you have multiple inputs with the same name as one output and you choose to map by the same name (or similar name) **and** to match descendants, an attempt is made first to match by path and name. If a match is found, one transform is made, and no further matches are considered.

5. When a single input has the same name as multiple outputs, multiple mappings are created, each for one input and one output.

However, if you have a single input with the same name as multiple outputs and you choose to map by the same name (or similar name) **and** to match descendants, an attempt is made first to match by path and name. If a match is found, one transform is made, and no further matches are considered.

6. When you select the **Map all simple descendants of the selected elements** option, the following steps are taken to match names:
 - a. Compare the relative path and item name of the selected input and output
 - b. Compare the item name without relative path

For example, assume you have the following input and output items:

- **Input:**

```
OldPurchaseOrder
  items
    item
      partNum
  partNum
```

- **Output:**

```
NewPurchaseOrder
  items
    item
      partNum
  resource
    partNum
```

If you select **Create transforms when the names of inputs and outputs are the same** when you have the inputs and outputs shown above, the relative paths of all the items are:

- Relative paths of the input items:

```
items/item/partNum
partNum
```

- Relative paths of the output items:

```
items/item/partNum
resources/partNum
```

During step a) `items/item/partNum` and `items/item/partNum` are matched.

During step b) `partNum` and `resources/partNum` are matched.

Inputs and outputs matched in a previous step are ignored in later steps.

When you select the **Map the immediate children of the selected elements** option, the only step taken to match names is to compare the item name without the relative path.

Mapping by similar name

Learn about the rules that apply when you select in Auto map the **Create transforms when the names of inputs and outputs are more similar than** option.

About this task

When you select **Create transforms when the names of inputs and outputs are more similar than**, the following rules apply:

1. Fixed value outputs and mapped outputs are excluded in name matching.
2. The *similarity* test is done using the name of an element, an attribute, or a database column regardless of its type. If an input and an output have the same name, it is a match, regardless of the kind of, and XSD type of, the input and output. An element, an attribute, and a database column can all form a match if their names are the same.
3. The similarity test applies in the same way to case sensitivity and alphanumeric characters as for **Mapping by same name**.
4. Namespace or namespace prefixes do not participate in the similarity test. XML namespaces are excluded from name matching. Therefore, `abc:something` and `xyz:something` are considered the same, as are `{http://www.abc.com}:something` and `{http://www.xyz.com}:something`.
5. When multiple inputs have the same name as one output, one mapping is created. However, if you have multiple inputs with the same name as one output and you choose to map by the same name (or similar name) **and** to match descendants, an attempt is made first to match by path and name. If a match is found, one transform is made, and no further matches are considered.
6. When you select **Map all simple descendants of the selected elements**, the following steps are taken to match names.

Inputs and outputs matched in a previous step are ignored in later steps:

- a. Compare the relative path and item name of the selected input and output
- b. Compare the item names without relative path

c. Compare similar item names without relative path

When you select the **Map the immediate children of the selected elements** option, the only step taken to match names is to compare similar item names without the relative path.

7. You can select the similarity threshold for two words to be considered similar.
8. You cannot use any other similarity algorithm.

Examples of similarity values

The following table lists words that are similar to one another, together with their similarity value, as a percentage:

Word1	Word2	Similarity value %
catalog	catalogue	85
anesthesia	anaesthesia	84
recognize	recognise	75
color	colour	66
theater	theatre	66
tire	tyre	33
intro	introduction	53
abbr	abbreviation	42
name	fullname	60
firstname	fullname	40
id	identification	14
NCName	Non colonized name	40
USA	United States of America	0
faq	frequently asked questions	0

Format of the synonym file

The Auto map facility allows you to create mappings between specific inputs and outputs by putting the names of the inputs and outputs in a file called the synonym file.

Synonyms, in the context of the synonym file, are groups of words that represent mappings that you want to create.

File type

A synonym file can reside anywhere in your file system, only if the encoding used in the synonym file is the same as that used by the Eclipse Toolkit system.

However, if the synonym file uses a specific encoding that is, or might be, different from the encoding of the Eclipse Toolkit, the file must reside in a project in the IBM Integration Studio.

If the synonym file is created outside the IBM Integration Studio, and uses a specific encoding, save the file under an IBM Integration Studio project and click **Refresh** to make the file visible in the navigator.

The synonym file uses Tab-separated or comma-separated files only. If you have written your mapping requirement in any external application, for example, Microsoft Word or Microsoft Excel, you must export the relevant data in a format that the synonym file supports.

Item names in the file

A synonym file contains the names of items to be mapped, without the path to the item or the namespace of the item.

For example, if you want to map `partNum` to `partNumber` in the following XML, you must put `partNum` in the synonym file, not `item/partNum`, `items/item/partNum`, or `purchaseOrder/items/item/partNum`.

```
<po:purchaseOrder xmlns:po="http://www.ibm.com">
  <items>
    <item>
      <partnum>100-abc</partnum>
      <productName>Acme Integrator</productName>
      <quantity>22</quantity>
      <USPrice>100.99</USPrice>
      <po:comment>Acme Integrator</po:comment>
      <shipDate>2008-12-01</shipDate>
    </item>
  </items>
</po:purchaseOrder>
```

Synonyms in the file can:

- Be case sensitive or not case sensitive
- Contain the entire mapping item name
- Have non-alphanumeric characters removed

Rows in the synonym file

In the synonym file, each row represents one group of names to be mapped between each other and each row must contain at least two names. Names within a row are separated by commas in `.csv` files, and by Tab characters in `.txt` files.

A synonym file can contain an optional special row at the top. This top row contains key words **Input**, **Output**, or **Input_Output**, separated by the same delimiter used in the remainder of the file. The top row is used to indicate whether the synonyms are to be used to match names in mapping the input or the output:

- If the first word in the top row is **Output**, the first name only, in each subsequent row is searched in the mapping output for name matching.
- If the second word in the top row is **Input**, the second name only, in each subsequent row is searched in the mapping input for name matching.
- If the third word in the top row is **Input_Output**, the third name only, in each subsequent row is searched in both the mapping input and mapping output for name matching.

The top row must not contain fewer key words than the maximum number of names in any row in the file.

If the top row contains any word other than **Input**, **Output**, or **Input_Output**, the top row is ignored and it is assumed that the top row is missing. If you omit the optional top row, every name in the synonym file is considered to be **Input_Output**; that is, any name found either in the mapping input or in the mapping output is matched.

If a synonym file contains two rows:

```
car          automobile
automobile   vehicle
```

car and *vehicle* are not considered to be synonyms.

In order to make all three words synonyms, your synonym file can have either of the following structures:

- One row with all three words -
car automobile vehicle
- Three rows -
car automobile
automobile vehicle
car vehicle

Special characters

You can write synonym files manually, or export them from another application; for example, Microsoft Excel.

Item names in synonym files reflect the application domain and do not have to match exactly the names in the XML schema or the relational database column.

For example, a synonym file might contain the row:

```
summer      l'été
```

As l'été does not conform to the XML NCName format, you could name the element l_été. If all the alphanumeric characters in the synonym file match those in the schema, you can use the file with the option **Alphanumeric characters (Letters and digits only)**.

Many mapping requirements are written in Microsoft Excel, and cells in a Microsoft Excel file might contain specific characters like double quotation marks, space, new line, comma, and so on. When such a Microsoft Excel file is saved as a Tab-separated or comma-separated file, they contain additional double quotation marks.

Two groups of synonyms in a synonym file are delimited either by a Line Feed (LF) character, or Line Feed followed by a Carriage Return (LFCR). A Carriage Return (CR) character by itself does not end a group of synonyms.

Leading and trailing space characters adjacent to the delimiter (comma or Tab character) are ignored. Blank rows, or rows that contain only space characters, are permitted and ignored in a synonym file.

Different editors might inject different space characters into a synonym file; spaces are not used to delimit synonyms, and spaces are ignored unless they are inside double quotation marks.

If a synonym contains a comma, a double quotation mark, a carriage return, or a leading or trailing space that is significant, the synonym must be enclosed in double quotation marks. A double quotation mark within a synonym is escaped with another double quotation mark. For example:

```
"comma,separated"  
"double"quote"  
"with<CR>  
  newline"  
" spaces "
```

When the synonym file is read by the Graphical Data Mapping editor, the double quotation marks at the beginning and end of the synonym are removed and the following data is stored in the synonym table:

```
comma,separated
double"quote
with<CR>newline
spaces
```

The Graphical Data Mapping editor reads a synonym file containing these special characters correctly, and you should select the **Alphanumeric characters (Letters and digits only)** option when using the synonym file.

Algorithm used to match synonyms

The way in which synonyms are matched by the Auto map function, to create mappings between specific inputs and outputs, follows a set of rules.

1. Fixed value outputs and mapped outputs are excluded in name matching. Any output field that has a fixed value is excluded in name matching. Any output that is already mapped, or under a container that is already mapped, is excluded from name matching.
2. The synonym matching is done by using the name of an element, an attribute, or a database column regardless of its type. If an input and an output have the same name, it is a match, regardless of the kind of, and XSD type of, the input and output. An element, an attribute, and a database column can all form a match if their names are the same.
3. The synonym matching of alphanumeric characters is not case-sensitive, and is identical to that used in "Mapping input to output elements automatically" on page 73.
4. Namespace or namespace prefixes do not participate in synonym matching. XML namespaces are excluded from name matching. Therefore, `abc:something` and `xyz:something` are considered the same, as are `{http://www.abc.com}:something` and `{http://www.xyz.com}:something`.
5. When multiple inputs have the same name as one output, one mapping is created. However, if you have multiple inputs with the same name as one output and you choose to map by the same name (or similar name) **and** to match descendants, an attempt is made first to match by path and name. If a match is found, one transform is made, and no further matches are considered.
6. If an input and an output have the same name, they are not considered a match under the option for synonyms. If you require a mapping for same-name inputs and outputs, you must also select the **Create transforms when the names of inputs and outputs are the same** option.
7. In addition to mapping synonyms, you might want to create mappings for some, but not all, same-name inputs and outputs. In this case, you have two options:
 - Clear **Create transforms when the names of inputs and outputs are the same**, and include the same-name inputs and outputs in the synonym file
 - Select **Create transforms when the names of inputs and outputs are the same**, and clear the unwanted mappings on the second page of the wizard.
8. When you select the **Map all simple descendants of the selected elements** option together with both same name and synonym mapping options, the following steps are taken to match names:
 - Compare the relative path and item name of the selected input and output
 - Compare the item name without relative path
 - Compare the item name without relative path to synonym

Inputs and outputs matched in a previous step do not participate in later steps.

9. When you select the **Map the immediate children of the selected elements** option together with both same name and synonym mapping options, the following steps are taken to match names:
 - a. Compare the item name without relative path
 - b. Compare the item name without relative path to synonym

Inputs and outputs matched in a previous step do not participate in later steps.

Creating and using a synonym file

You can use a synonym file to configure automatic mapping of input to output elements in the Auto map wizard. You can use either a Tab delimited .txt file or a comma delimited .csv synonym file. You can create a synonym file manually or generate a synonym file from the information that is contained in a Microsoft Excel spreadsheet.

Procedure

Complete the following steps to configure Auto map to use a synonym file that is created from the information that is contained in a Microsoft Excel spreadsheet:

1. Optional: Create a synonym file where the original mapping requirement is written in Microsoft Excel. The following set of instructions describe how to create a synonym file where the original mapping requirement is written in Microsoft Excel. If your original requirement is written in a table in Word, you must copy and paste the table into Microsoft Excel before you begin.
 - a. Select the section of the Microsoft Excel spreadsheet that you require. For example, if you have a Product that you want to map to a Part number, select that section of the spreadsheet.
 - b. Remove all columns from the spreadsheet, except the ones that contain the input field name and the output field name. You might have to edit some of the cells. For example, if your mapping instruction includes the phrase based on, remove this phrase.
 - c. If the input or output fields contain paths, remove the paths to leave only the short names of the item. However, it is helpful to sort the column before you remove the paths. Sorted path names can indicate which is the best input or output to select when you start the action. If all the interested inputs or outputs start with the same path prefix, you might consider selecting the lowest input (or output) node in the tree, which has that common path prefix.
 - d. Remove all rows that do not have an input field name and an output field name. For example, if you have an obsolete product that no longer has a part number and you have n/a in the input, remove that row.
 - e. Select the **Save As** function in Microsoft Excel to save the spreadsheet into a format that is supported by IBM Integration Studio. You can use either a Tab delimited .txt file or a comma delimited .csv file. A comma delimited file can be opened with Microsoft Excel. The file can also be opened in a text editor.
2. Create the mappings by using the synonym file. Select the options in the Auto map wizard that match your requirements. For example, select the default options of **Map all simple descendants of the selected elements** and **Alphanumeric characters (Letters and digits only)**.

When you choose these options, select **Create transforms when the input and output names are matched to synonyms defined in a file**.

If you want to map same-name inputs to outputs, and the synonym file does not contain rows with those names (for example a row with `car,car`), select the **Create transforms when the names of inputs and outputs are the same** option, in addition to the **Create transforms when the input and output names are matched to synonyms defined in a file** option.

You can select both **Create transforms when the names of inputs and outputs are the same** and **Create transforms when the names of inputs and outputs are more similar than**, in addition to **Create transforms when the input and output names are matched to synonyms defined in a file**, if your synonym file does not contain a row `color,color` and you want to map between them.

3. Click **Finish**.

Selecting matches

Use the Auto map wizard to select the mappings that you want to create.

About this task

When you have specified how you require the names to be matched on the initial panel of the Auto map wizard, and have selected **Next**, you see a panel that displays all the matches found.

You can now select the options that you require:

Procedure

1. Select a row in the **Transform Outputs** column that you want to change. Selecting a folder tree node results in the entire tree branch being selected or not selected.
2. Click **Edit** to start the **Select Transform Input** dialog.
3. To select a transform output, select the appropriate tree node check box. Conversely, to remove a mapping output, clear the appropriate tree node check box.
4. Ensure that you have selected only the number of matches that you require. The third column displays the number of transform inputs selected for each transform output. The cell has a value greater than one when the input of a transform contains several elements of certain names under various containers, and the input names match to the same output name.
5. Click **Finish** to complete the mapping process, or click **Back** to change the matches that you have set up. When you click **Finish**, you obtain a warning message if either, or both, of the following conditions apply:
 - a. More than a few inputs to map to the same output.
 - b. Many outputs for which you want to create mappings.

Specifying a transform (mapping operation)

Specify a transform, a cast function, or an XPath function between two or more elements by selecting from the list of available mapping operations that are shown on the connection.

Before you begin

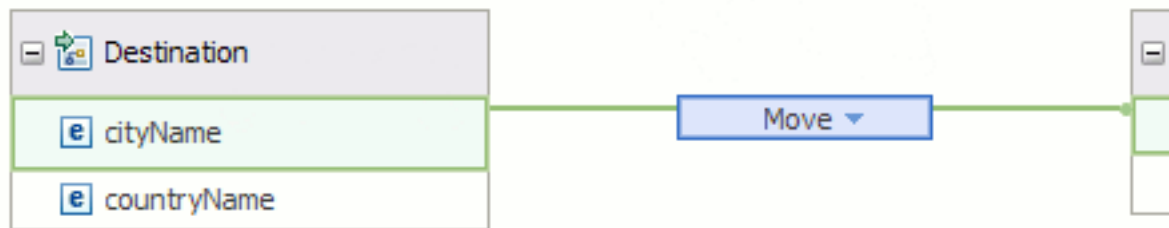
Learn about the different mapping operations. For more information, see Chapter 4, “Transforms (Mapping operations),” on page 21.

About this task

In the Graphical Data Mapping editor, you can use transforms, cast functions, and XPath 2.0 functions to run different actions on input data and move the result to the output element. You choose the appropriate mapping operation that is based on the result that you want to achieve.

When you create a connection between two or more elements, a transform is assigned, based on the number and type of input elements. You can then change the transform by choosing from a list of available transforms. If a particular transform type is not shown in the list, that transform is not valid for your input and output elements.

For example:



Note: Transforms that require multiple inputs such as **ForEach**, **Join**, or **Append** are not available in the list of available transforms until you wire two inputs to the transform.

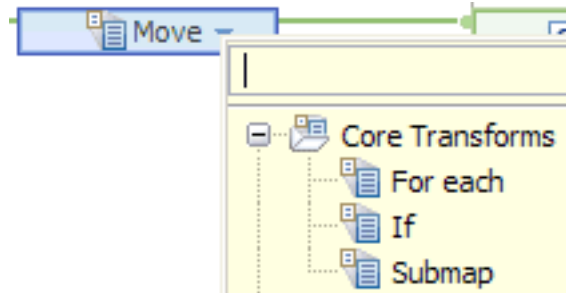
When you have a list of valid transform types, choose the appropriate transform:

- If you have a single array as input, with the same array type as output, and you want to move all elements to the output, use **Move**.
- If you have a single array as input, and you want to iterate over each element in the array (for example, you might want to remove some elements) use the **ForEach** transform and set the cardinality options.
- If you have multiple input elements, you can use the **Append** or the **Join** transforms. If you use the **Append** transform, the number of output elements is the total of the input elements. If you use **Join** transform, the number of output elements depends on the user expression added to specify the matching criteria for joining or filtering input items.

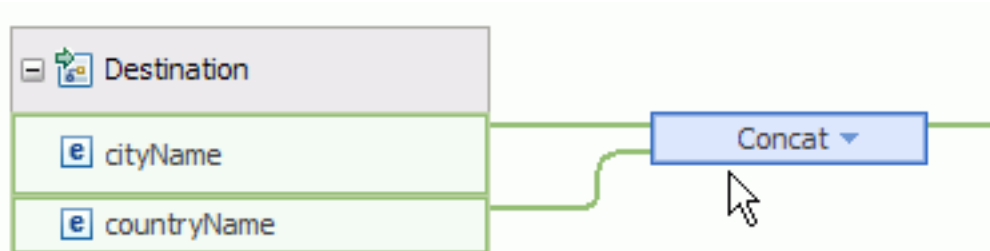
The following steps show how to change the transform that is selected, and also how to add more elements:

Procedure

1. Change the transform by clicking the arrow in the transform box, and selecting from the list of available transforms. For example:



2. If you create more primary connections between an input element and the transform, the transform type changes to one that allows multiple primary inputs, such as **Local Map**, **Concat**, or **Join**. For example:



What to do next

Configure the properties of the transform. For more information, see “Configuring the properties of a transform.”

Configuring the properties of a transform

Configure the properties of a transform to set the value of an output element; define a conditional expression that determines whether the transform is applied; define the indexes to use when the input element is a repeating structure; or reorder the way in which inputs to a transform are handled by the Graphical Data Mapping editor.

Procedure

Complete the following steps to configure the general properties of a transform:

- Open the **Properties** view of a transform by using any of the following methods:
 - Select a transform. Then, select the **Properties** tab.
 - Right-click the transform, and then select **Show in > Properties view**.
 - When the map is in full view, select **Alt+Enter**.
You can configure these keyboard shortcuts by selecting **Window > Preferences > General > Keys**.

Note: You can detach the **Properties** view from the map. This is very useful if you have a secondary workstation screen. In one workstation screen you can see the map, and in the other you can see the properties map.

- In the **Properties** tab, add, modify, or remove resources:
 - Select **Cardinality** to select the indexes of the input and output elements that you want this transform to operate over. For more information, see “Selecting the indexes of input array elements” on page 28.

- Select **Variables** to list the names of the input elements that are connected to the transform. Select **Edit** to change the name of a variable.

Note: When you change the name of a variable, the new name is the one you see when you use content-assist to create your expressions.

- Select **Condition** to define the XPath expression that must be evaluated against the input element before the transform is applied. If the condition evaluates to **true**, the transform is applied. For more information, see “Defining an XPath conditional expression for a transform.”
- Select **Filter** to define an expression that must be evaluated against each element on a repeating input element before the transform is applied. If the condition evaluates to **true**, the transform is applied for each input.
- Select **Sort** to sort the inputs to the transform by ascending order, descending order, case order, or data order. Then, complete the following steps:
 1. Select **Sort the inputs to this transform**.
 2. Add elements to the Sort by column.
 3. Select a sort method.
- Select **Order** to display the order of input connections to a transform. You can reorder them.
- Select **Documentation** to provide a description of the transform, or other relevant usage notes.

What to do next

Deploy and test the message map. For more information, see Chapter 25, “Deploying message maps,” on page 191 and Chapter 24, “Troubleshooting graphical data maps,” on page 189.

Defining an XPath conditional expression for a transform

You can define an XPath expression to set the conditional expression that determines whether a transform is applied in a message map. When the XPath expression evaluates to true, the transform is applied.

About this task

In the XPath expressions that you can use in a message map, the value obtained from a input element, by using the variable name, has an *effective Boolean value*.

The effective Boolean value is false for the following input elements:

- The input element is Boolean and its value is false.
- The input element is a sequence, and the sequence is empty.
- The input element is string and its value is the empty string "".
- The input element is a float or a decimal and the value is NaN (not a number).
- The input element is of a numeric type and its value is 0.

In any of these cases, the XPath expression can be formed from just the variable name that represents the input element.

You can also get a Boolean result from defining expressions that use variable names that represent inputs and constant values and any of the following operators :

- Logical operators such as and, or, not.

- Comparison operators such as =, !=, <, >, <=, >=.

Note: Always use content assist to select the variable name of the input elements that you use to define the XPath expressions. If you do not use content assist, you may be using an incorrect element name and your map will fail at run time.

Note: XPath 1.0 functions are valid XPath 2.0 expressions. You can use the XPath Expression Builder to generate simple XPath 1.0 expressions.

Procedure

Complete any of the following steps to set a conditional expression in a transform:

- For non-repeating elements, select the **Condition** property in the **Properties** tab, and enter an XPath expression.
- For repeating elements, select the **Filter Inputs** property in the **Properties** tab, and enter an XPath expression that will be applied to each instance of the repeating element.

For more information, see “Defining an XPath conditional expression for a structural transform (ForEach)” on page 89.

Results

The input element is evaluated against the condition. If the condition evaluates to true, the transform is applied to the input element.

Example

The following examples show how to define simple XPath conditional expression for a transform when you have non-repeating elements:

Example: XPath expression to check a Boolean input element

This example shows how to define the XPath expression for a Boolean element so that it evaluates to true. The expression depends on the value of the Boolean element.

The XML schema for the element is the following:

```
<element name="IsEmployee" type="boolean"></element>
```

When the value of a Boolean element is set to **false**, the XPath expression that you define is the following:

```
fn:not($IsEmployee)
```

When the value of the Boolean element is set to **true**, the XPath expression that you define is the following:

```
$IsEmployee
```

Example: XPath expression to check if the value of a numerical input element is greater than a constant value

This example shows how to define an XPath expression that evaluates to true when the value of a numerical element is greater than 200.

The XML schema for the element is the following:

```
<element name="BusinessUnit" type="int" ></element>
```

The XPath expression that you define is the following:

```
$BusinessUnit > 200
```

Example: XPath expression to check if a numerical input element has a specific value

This example shows how to define an XPath expression that evaluates to true when the numerical input element has a value of 0.

The XML schema for the element is the following:

```
<element name="QtyBooks" type="int" ></element>
```

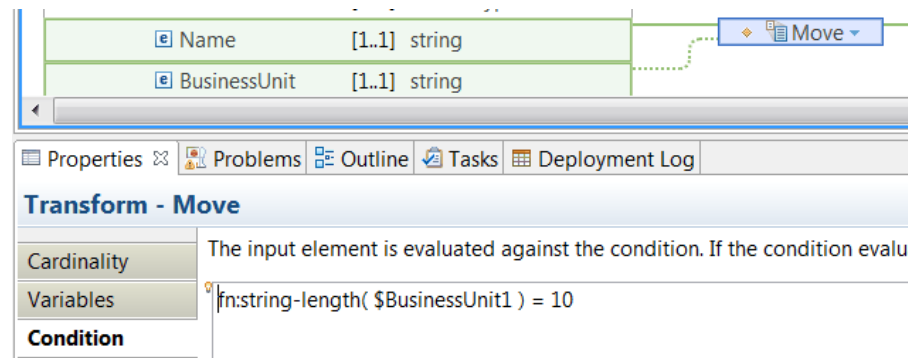
The XPath expression that you define is the following:

```
$QtyBooks = 0
```

Example: XPath expression to check the string length of an input element

This example shows how to define an XPath expression that evaluates to true when the length of a string is 10.

- The **Move** transform has a primary connection wired from the element **Name**.
- The **Move** transform has a secondary connection wired from the element **BusinessUnit**. This connection is supplementary because this element is not used to calculate the value of the output element.
- The **Move** transform should only execute if the length of the element **BusinessUnit** is 10.



The XML schema for the elements are the following:

```
<element name="Name" type="string" ></element>
```

```
<element name="BusinessUnit" type="string" ></element>
```

The XPath expression that you define is the following:

```
fn:string-length($BusinessUnit) = 10
```

Example: XPath expression to check if a string input element is set to the empty string

This example shows how to define an XPath expression that evaluates to true when a string element is empty.

To define an XPath expression that checks if a string element is empty, you must use the operator !=.

The XML schema for the element is the following:

```
<element name="Name" type="string" ></element>
```

The XPath expression that you define is the following:

```
$Name != ''
```

Example: XPath expression to check if a repeating element is empty

This example shows how to define an XPath expression that evaluates to true when a repeating element, which is referred to as a sequence, is empty.

The effective Boolean value of an empty sequence is false. You could use `fn:not($Address)`, but it is more readable to use `fn:empty()` explicitly.

The XPath function `fn:empty()` evaluates to true if a sequence is empty.

The XML schema for the element is the following:

```
<xsd:element form="qualified" name="Address" type="mqsistr:Address"
             maxOccurs="unbounded" minOccurs="0" />
```

The XPath expression that you define is the following:

```
fn:empty($Address)
```

Example: XPath expression to check if an optional input element is present

This example shows how to define an XPath expression that evaluates to true when an optional element is present.

Use the `fn:exists` XPath function if the input element is a boolean. Otherwise, you can use the effective Boolean value.

The XML schema for the element is the following:

```
<element name="BookName" type="string" maxOccurs="unbounded"
             minOccurs="0" ></element>
```

The XPath expression that you define is the following:

```
$BookName
```

Example: XPath expression to check if a complex input element has no content, that is, it is empty

To determine whether a complex element is empty, you must check for the presence of child elements or attributes.

Testing the effective Boolean value of elements or attributes that are present in a complex type will yield true. You can use the `fn:not` XPath function to invert a boolean.

The `fn:not` function accepts a sequence of items. The value that this function returns is true if any of the arguments is either a single Boolean value false, a zero-length string, the number 0 or NaN, or the empty sequence. Otherwise, it returns false.

This example shows how to define an XPath expression that evaluates to true when the complex input element has no children.

The XML schema for the elements are the following:

```
<complexType name="Address">
  <sequence>
    <element name="Type" type="string"/>
    <element name="Number" type="integer"/>
    <element name="Street" type="string"/>
    <element name="Postcode" type="string"/>
    <element name="City" type="string" />
    <element name="Country" type="string"/>
    <element name="AdditionalInfo" type="string"/>
  </sequence>
</complexType>
```

The XPath expression that you define to check for child elements being present in a complex element is the following:

```
fn:not($Address/* )
```

The XPath expression that you define to check for attributes being present in a complex element is the following:

```
fn:not($Address/@* )
```

The XPath expression that you define to check for elements and attributes being present in a complex element is the following:

```
fn:not($Address/*) and fn:not($Address/@* )
```

What to do next

Deploy and test the message map. For more information, see Chapter 25, “Deploying message maps,” on page 191 and Chapter 24, “Troubleshooting graphical data maps,” on page 189.

Defining an XPath conditional expression for a structural transform (ForEach)

Configure the **Filter Inputs** section in the **Properties** view of the **ForEach** transform to define the conditional expression that determines whether the transform is applied in a message map.

About this task

The **ForEach** transform can only have one **primary** input connection. Additional connections to the **ForEach** transform must be of type **Supplement**.

Procedure

Complete the following steps to define a conditional expression on a **ForEach** transform:

- Select **Allow empty input** if you want the **ForEach** transform to execute at least one.

If you select this option, the transformations that you define in the **ForEach** transform nested map will execute once regardless of the conditional expression.

- Define the XPath expression that determines whether the **ForEach** transform is applied in the map.

Note: Always use content assist to select the name of the input elements that you use to define the XPath expressions.

The conditional expression applies to all the indexes that you configure in the **Cardinality** tab of the **ForEach** transform properties view.

Results

The input element is evaluated against the condition. If the condition evaluates to true, the transform is applied to the input element.

Example

This example shows how to define an XPath expression that checks the value of a string element:

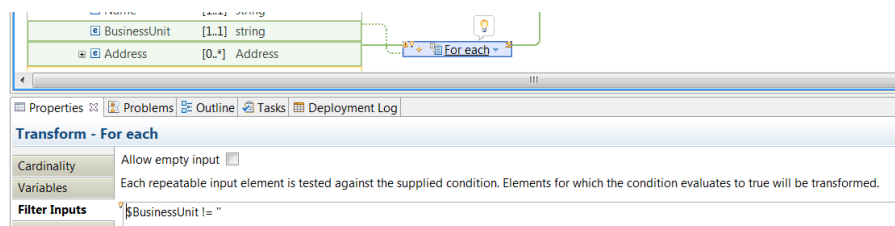
- The **ForEach** transform has a primary connection wired from the repeating element **Address**.
- The **ForEach** transform has a secondary connection wired from the mandatory element **BusinessUnit**. This element can be set to nil. This element is not used to calculate the value of the output element.
- The **ForEach** transform should only execute if the **BusinessUnit** element is not empty.

The XML schema for the mandatory element is the following:

```
<element name="BusinessUnit" type="string" nillable="true"></element>
```

The XML schema for the repeating element is the following:

```
<xsd:element form="qualified" name="Address" type="mqsistr:Address" maxOccurs="unbounded" minOccurs="0" />
```

The XPath expression that you define is the following:

```
$BusinessUnit != ''
```

What to do next

Deploy and test the message map. For more information, see Chapter 25, “Deploying message maps,” on page 191 and Chapter 24, “Troubleshooting graphical data maps,” on page 189.

Choosing an XPath conditional expression that tests for a nil value in a transform

Use the XPath functions **fn:empty**, **fn:nilled**, or **fn:exists** to test if an output element is set to nil. You can use these functions to define conditional expressions in a transform.

About this task

For example, you can use **fn:empty**, **fn:nilled**, or **fn:exists** as part of conditional expression that determines if a transform is applied. You can also use these transforms as part of the conditional expressions that you set in an **IF** transform.

Procedure

Choose any of the following XPath functions to determine the state of a source element:

XPath function	Usage	Source element	Output
fn:empty	Tests whether a set of elements is empty	A single XML or non-XML element that is present and nilled	false
fn:empty	Tests whether a set of elements is empty	A NULL value in a logical tree	false
fn:empty	Tests whether a set of elements is empty	A missing element, not present in the logical tree	true
fn:nilled	Tests whether an element is nilled	A nillable XML element	true only if the xsi:nil attribute is present and set to 'true' in the logical tree
fn:nilled	Tests whether an element is nilled	A nillable non-XML element	true only if the value in the logical tree is NULL

XPath function	Usage	Source element	Output
fn:exists	Tests whether an element exists	A single XML or non-XML element	true if the element is present in the logical tree, regardless of the nillable and nilled state
fn:exists	Tests whether an element exists	A NULL value in a logical tree	true
fn:exists	Tests whether an element exists	A missing element, not present in the logical tree	false

Example

This example shows an XPath expression that checks if an input element is nilled.

The XPath expression evaluates to true when an input element is not set to nil.

Use the `fn:nilled` XPath function to test whether the value of an input element has the `xsi:nil` attribute set.

Note: An XML element that has the `xsi:nil` attribute set is considered to be present.

The XML schema for the element is the following:

```
<element name="BookName" type="string" nillable="true" ></element>
```

The XPath expression that you define is the following:

```
fn:nilled( $BookName)
```

Grouping transforms into nested maps

Use the **Group transforms into nested maps** property to nest the transforms in a new **Local map** transform. The local map provides a way of displaying parts of a larger message map, enabling you to view the message map elements in a hierarchical way.

About this task

When the **Group transforms into nested maps** property is selected, the Auto map wizard attempts to place new transforms inside new message map transforms.

The Auto map wizard recursively analyzes the input and output elements of the new transforms, searching for a common input ancestor and a common output ancestor. If a common input ancestor is found for the input elements, and a common output ancestor is found for the output elements, a new **Local map** transform is created between them. Inside the new local map, when no more common ancestors can be found, the new transforms are created at this level. If there are still some common ancestors at a given level, the process is repeated.

For example, a map might contain the following input:

```
SomeInput
--Resources
---field1
---field2
---field3
---field4
```

and the following output:

```
SomeOutput
---Items
----field1
----field2
----field3
----field4
```

If the **Group transforms into nested maps** property is not selected, the Auto map wizard creates transforms between the input and output elements field1, field2, field3, and field4 at the current level of the map.

However, if the **Group transforms into nested maps** property is selected, the Auto map wizard creates a **Local map** transform between SomeInput/Resources and SomeOutput/Items at the current level of the map, and then creates transforms between the input and output elements field1, field2, field3, and field4 inside the local map.

Using content assist (Mapping syntax)

Use content assist to find the variable names for input and output objects, paths, and database elements.

Press **Ctrl+Spacebar** to display content assist, and use it to help you build your XPath statements and mapping expressions. Content assist provides the variable names that you need to use to reference elements in your XPath statements and expressions. The variable name assigned for the same element can vary between uses in different transforms, so always use content assist to obtain the correct variable name. Avoid copying and pasting expressions that include variable names.

You can use content assist for the following tasks:

- Making comparisons
- Performing arithmetic
- Creating complex conditions

The comparison operators are:

- = (equals)
- != (not equals)
- > (greater than)
- >= (greater than or equal)
- < (less than)
- <= (less than or equals)

The arithmetic operators are:

- + (plus)
- (minus)
- * (multiply)
- div (divide)

For information about XPath syntax, see W3C XML Path Language (XPath) 2.0.

Conditional operators (or and and) are supported; these are case-sensitive.

The following objects can be mapped:

- Message elements (defined in the schema for the input and output)
- Message assembly, comprising the properties tree and optionally the LocalEnvironment and transport headers
- Data from database tables

Database objects with names that do not conform to the XML NCName format

Some database objects have names that do not conform to the XML NCName format (for example, the name contains characters like #, or \$). If the database object name is used in SQL (for example, in the where clause of the **Select** transform), no action is required. If the database object is used in XPath (for example, in a Custom transform or a condition), use content assist, which adds the appropriate XPath-compliant expression.

Deleting objects and transforms

You can delete input objects, output objects, and transforms from a message map by using the Graphical Data Mapping editor.

Procedure

- To delete an input object or an output object from a message map, choose any of the following methods:
 - Select the appropriate object in the Graphical Data Mapping editor, and then click the **Delete selected elements** icon in the toolbar.
 - Select the object and then either press the delete key **Del**, or right-click and select **Delete** from the context menu.

You can delete multiple elements at the same time by marking them by using **Ctrl + click**, and then clicking the **Delete selected elements** icon.

- To delete a transform from a graphical data map, choose any of the following methods:
 - Select the transform and press the delete key (**Del**).
 - Select the transform, right-click on the transform, and then select **Delete** from the context menu.

What to do next

Continue editing the map, and define transformations between the input message and the output message. For more information, see Chapter 9, “Editing message maps,” on page 61.

Chapter 10. Advanced editing in a message map

You can configure the input and output message assembly in a message map by using the Graphical Data Mapping editor.

Procedure

1. Configure the general properties of the input and output message assembly by using the Graphical Data Mapping editor. For more information, see “Configuring the properties of the input and the output message assembly to a message map” on page 108.
2. Optional: Configure the message assembly to include any of the following components: the Properties tree, one or more message tree headers, the message tree body, and the local environment tree. For more information, see “Configuring the message map to include message assembly components.”
In IBM Integration Bus, the message assembly is the internal representation of a message. When you transform a message, you might need access to elements in a message assembly component or you might need to modify some of these elements in your message map.
3. Map graphically data available in the transport headers. For more information, see “Mapping transport headers” on page 109.
4. Map graphically data available in the local environment tree. For more information, see “Mapping data in the local environment tree” on page 111.
5. Configure a database to map or modify database content. For more information, see “Adding database definitions to the IBM Integration Studio” on page 116.

What to do next

Configure mapping transforms in your message map. For more information, see “Choosing a transform to set the value of a simple type output element” on page 23.

Configuring the message map to include message assembly components

In IBM Integration Bus, the message assembly is the internal representation of a message. When you transform a message, you might need access to elements in a message assembly component or you might need to modify some of these elements in your message map. You can configure a message map to include the following message assembly components: the message tree Properties tree, message tree headers, the message tree body, and the local environment tree.

About this task

When a message arrives to an application or to an integration service, it is received by an input node that you have configured in a message flow. Before the message can be processed by the message flow, the message must be interpreted by one or more parsers that create a logical tree representation from the bit stream of the message data. The logical tree is also known as the message assembly. The tree format contains identical content to the bit stream from which it is created, but it is easier to manipulate in the message flow.

Procedure

To include message assembly components into your message map, complete the following steps:

1. Identify the message assembly components that you need to add to your message map. For more information, see “Choosing message assembly components to include in a message map.”

You may need to include the local environment tree to use information provided in a variable or you may need to add a header to access transport specific information.

2. Identify whether you need to initialize, delete, or transform elements in components of the message tree or in the local environment tree. For more information, see “Choosing a mapping action” on page 98.

You can add different parts of the message tree to the map input, to the map output, or to both. You can also add the local environment tree. Depending on how you add a message assembly component, this component can be deleted, initialized, or transformed.

3. Configure the message map to include a message assembly component. For more information, see “Customizing a message map to include a message assembly component” on page 105.

To customize your message map to include more message assembly components, you must add message assembly components to the input message and to the output message, and then define transforms between them.

Results

You now have a message map that includes the message assembly components that you need to complete your message transformation.

What to do next

Define transforms between other message assembly components that you have included. For more information, see “Specifying a transform (mapping operation)” on page 82.

Choosing message assembly components to include in a message map

You can add the message tree components, that is, the Properties tree, the headers, and the message body into a message map. You can also add the local environment tree into a message map.

About this task

In IBM Integration Bus, the logical tree structure is the internal representation of a message. The logical tree structure is created by the parser when the message is received by an input node. It is also known as the Message tree and makes up part of the message assembly. The message assembly consists of four trees:

- Message tree: This tree includes the Properties folder, the message body, and headers.
- Environment tree
- Local environment tree: This tree includes multiple destination folders, and a variables folder.
- Exception list tree

When you create a message map, the Properties folder and the message body are automatically included in your Graphical Data Mapping editor.

Note: You can remove the properties folder and the message body in the message map if you only want to modify the local environment tree. This will accelerate your message transformation since the message properties and the message body will be copied over without the need to bring them into the transformation engine.

You cannot add the exception list tree to the message map. The exception list is included automatically, and the entire contents of the input exception list is retained in the output.

Procedure

Choose one or more message assembly components to include in a message map:

- **Header folders:** You can add one or more headers to a message map, in addition to the Properties folder and the message body.

When an input message is received by an input node, the input node invokes the correct parser for each header, and includes in the message tree the corresponding headers. You can then access these headers by using message maps.

The message tree always includes the following components:

- All the headers that are present in the message.
- The message body.
- The Properties folder. The Properties folder (sometimes referred as the Properties tree) is the first element of the message tree and holds information about the characteristics of the message. When the input node receives the input message, it creates and completes the Properties folder.

If you need to access information available in an element of a header or if you need to modify it, then you must add the header to the message map. For more information, see “Mapping transport headers” on page 109.

- **Local environment tree:** You can add the local environment tree to the message map. The local environment is divided into two parts:
 - Standard folders that are automatically defined for each of the destination folders available in IBM Integration Bus.
 - A variables folder that is added automatically. You can use the **Cast** function to include a variable into your message map.

The local environment tree stores variables that can be referred to and updated by message processing nodes that occur later in the message flow.

You can also use the local environment tree to define where a message is sent. The destination can be internal or external to the message flow.

IBM Integration Bus also stores information in the local environment tree in some circumstances, and references it to access destination values that you might have set.

If you need to access information available in an element of the variables folder or if you need to modify a variable, then you must add each individual variable to the message map. For more information, see “Configuring the local environment tree **Variables** folder by using the **Cast** function” on page 113.

Results

The following table summarizes the message assembly tree folders that you can include into your message map:

Table 7. Message assembly trees that can be included in a message map

Message assembly trees	Folders in a message assembly tree	Can be configured in a message map as an input to the map and as an output to the map?	Status in a message map
Message tree	Properties folder	Yes	Required
Message tree	Header folders	Yes	Optional
Message tree	Message body	Yes Note: You must cast parts of the SOAP message body to be able to define transforms between its input and output elements.	Required
Local environment tree	Variables folder	Yes (You must cast a variable to define transforms between its input and output elements.)	Optional
Local environment tree	Destination folders	Yes	Optional
Environment tree		No	
Exception list tree		No	

What to do next

Identify the configuration of the different message assembly components. For more information, see “Choosing a mapping action.”

Choosing a mapping action

You can add different parts of the message tree to the map input, to the map output, or to both. You can also add the local environment tree. Depending on how you add a message assembly component, this component can be deleted, initialized, or transformed.

Procedure

Identify the action that you want to achieve in your message map to find out how to add a message assembly component into the message map:

- To copy a message assembly component unchanged, do not include the component in the message map.

The mapping engine copies the local environment tree and any other headers and folders from input to output, unchanged, when they are not included in the message map.

The mapping engine handles the following components as independent units of transformation:

- The properties tree in the message tree
- The message body in the message tree
- Each header structure in the message tree. For example, the MQMD is treated as one unit, the MQRFH2 header is considered an independent unit, and so on.
- The local environment tree

If any of these units is not included in the message map, the mapping engine copies their contents unchanged.

Note: If the only transform that you define between an input map component and an output map component is the **Move** transform so that the component is copied over without any modification, you are recommended to remove the component from the message map. The map transformation will be more efficient since the mapping engine will only focus on the structures that do require change.

- To read elements of a message assembly component, add the component to the input message assembly only. The Mapping node passes it through unchanged
- To modify all the elements of a message assembly component, add the component such as the local environment tree to the input message assembly and to the output message assembly. Then, define transforms between each of its elements.
- To modify some elements of a message assembly component, add the component such as the local environment tree to the input message assembly and to the output message assembly. Define a **Move** transform for the entire component, that is, at folder level, and then specific transforms for each of the elements that you want to transform within an **Override** function. For more information, see “Transforming some elements of a message assembly component by using the **Override** function” on page 102.
- To initialize a message assembly component, that is, to create a new message assembly component in your output message, add the message assembly component only to the output message assembly. For more information, see “Initializing a message assembly component in the output message” on page 105.
- To add a message assembly component, add the message assembly component to the output message assembly and populate at least one field. The Mapping node creates a new output structure containing the results of your transformations.
- To delete a message assembly component from the input message, add the message assembly component to the output message assembly and do not set any field. For more information, see “Deleting a message assembly component from the output message” on page 104.

Results

The following table summarizes the mapping engine behavior when you add the local environment tree to your message map. The same behavior applies when you add any of the header folders, such as the **MQ headers > MQMD** folder, to your message map.

Table 8. Mapping engine behavior when adding the local environment to a message map for transformation

Input element	Output element	Transform defined between the input element and the output element	Mapper behavior
Local environment tree	Local environment tree	A transform operation such as Move is defined between the input local environment tree and the output local environment tree. Additional transforms are defined between some elements of the local environment tree within an Override function to change the value of those elements.	The input local environment tree is copied into the output local environment tree. The elements whose transforms are defined within the Override function have output values different from the input values based on the transformation. The elements outside the Override function maintain the same values in the output local environment tree.
Elements from other message assembly components, database elements whose values are obtained by doing a database read, or a combination of both	Local environment tree	A Move transform is defined between each parent input message assembly structure and its corresponding output message assembly structure. Additional transforms are defined between input elements and the output local environment tree. Note: If you do not define the Move transform between an input and an output message assembly structure from which you use values to set the local environment output elements, then you will lose the message assembly structure in the output message, although your map will perform the transformation correctly.	Each input message assembly structure is copied into its corresponding output message assembly structure. The output local environment variables are defined as per the additional transforms using values from the input elements.
Local environment tree	Local environment tree	None	Delete the original local environment tree. An empty local environment tree is created for the output message.

Table 8. Mapping engine behavior when adding the local environment to a message map for transformation (continued)

Input element	Output element	Transform defined between the input element and the output element	Mapper behavior
Local environment tree	None	None	Delete the original local environment tree. An empty local environment tree is created for the output message.
None	Local environment tree	None	Delete the original local environment tree. An empty local environment tree is created for the output message. You might populate some of the fields by using transforms between other input map components such as the message body and the new local environment structure.
Local environment tree	Local environment tree	A transform operation such as Move can be defined between one element from the input local environment tree and one element from the output local environment tree. The rest of the local environment elements do not have transforms defined that specify how to move the input value to the output value.	The output local environment tree is initialized and only the element that has the transform defined has a non default value set.
Local environment tree	Local environment tree	A transform operation such as Move is defined between the input local environment tree and the output local environment tree.	The input local environment tree is copied into the output local environment tree.

What to do next

Configure the message map to include the local environment tree or a message header. For more information, see “Customizing a message map to include a message assembly component” on page 105.

Transforming some elements of a message assembly component by using the Override function

You can use the **Move** transform to copy a complex type from the input message to the output message, while updating some of the child elements in the complex type by using the **Override** function. A message assembly component is described by a complex data structure.

About this task

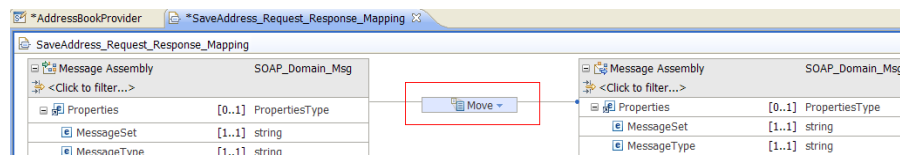
Note: You can only use the **Override** function to include **Move** transforms and **Assign** transforms.

Procedure

When you want to modify only some fields of a message assembly component, complete the following steps to transform the message assembly component:

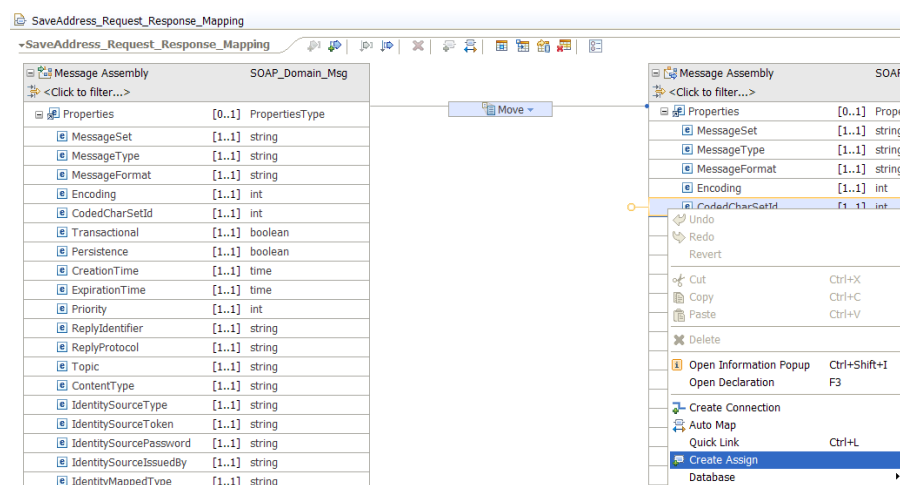
1. Required: Define a **Move** transform to copy the input message component into the output message component, that is, to copy an input complex data structure into an output complex data structure.

For example, the Properties tree has a **Move** transform defined automatically when you create a message map so that all elements in the Properties tree are copied to the output Properties tree structure. If you transform elements in the Local Environment tree, you must manually define the **Move** transform.

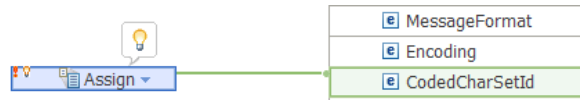


2. Add additional transforms between the input and output elements in the message assembly component.

For example, you need to change the encoding for the output message. You assign a different value to the **Encoding** element in the properties tree. Right-click the **Encoding** element, and then select the menu option **Create Assign**.

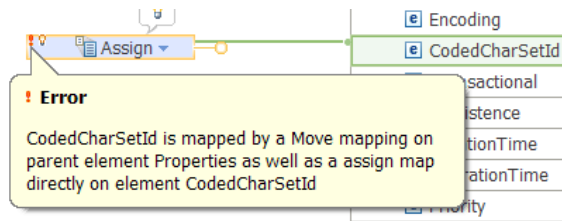


The **Assign** transform is defined and connected to the **Encoding** element in the output Properties tree.

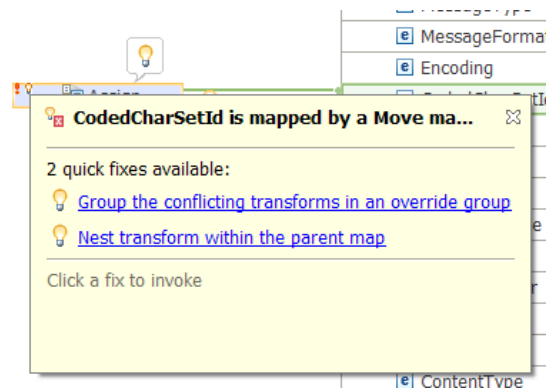


You get the following icons on the top left hand side of the transform:

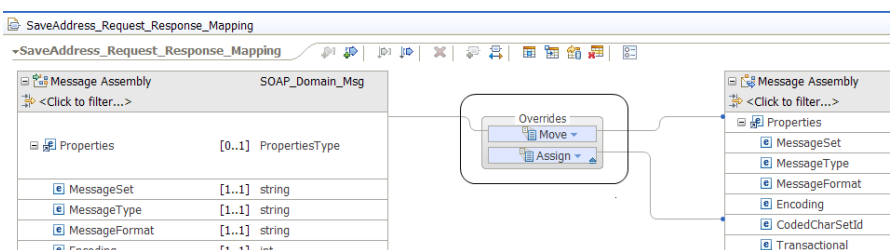
- An **Error** icon represented with a red exclamation mark. You can ignore this error and continue. You get the error because you have defined two transformations on an element and this is not allowed. By using the **Override** function, you fix the problem.



- A **suggestion** icon represented by a yellow light bulb. When you hover over the icon, you get the following pop-up window:



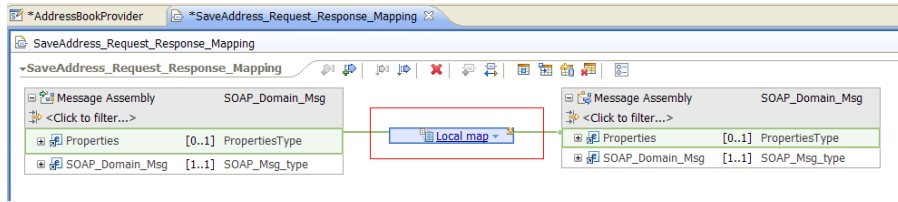
3. When you hover over the yellow light bulb, choose **Group the conflicting transforms in an override group**. This option is the recommended approach and allows you to maintain visibility of the transforms you have defined in the main transformation map.



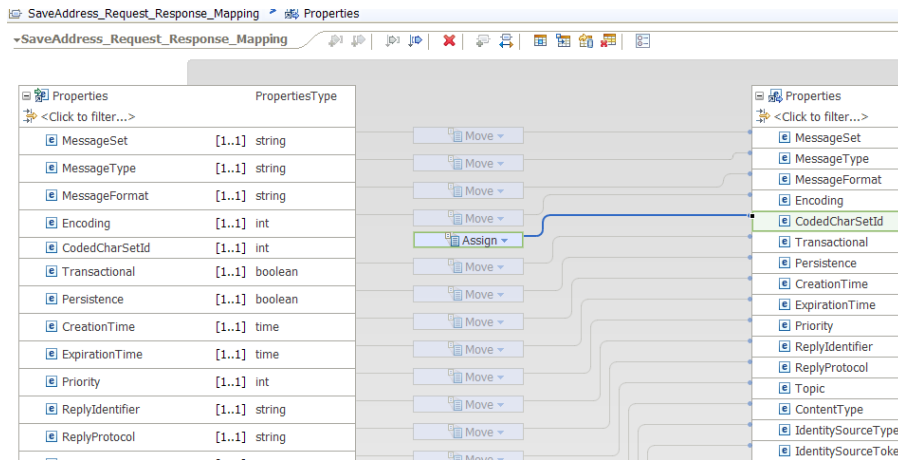
Results

You have transformed elements of a message assembly component by using the **Override** function.

Note: If you choose **Nest transforms within the parent map**, a **Local map** transform is defined between the input Properties tree and the output Properties tree.



The local map that is created contains a **Move** transform per element, with the exception of the **Encoding** element that has an **Assign** transform.



What to do next

Configure the message map body parts. For more information, see “Mapping input to output elements manually” on page 70 or “Mapping input to output elements automatically” on page 73.

Deleting a message assembly component from the output message

To delete a message assembly component from the output message, add the message assembly component to the output message and ensure that there are no mappings to it.

Procedure

To delete a message assembly component from the output message, complete any of the following steps:

- To delete the local environment tree, add the local environment tree to the output message, and do not set any fields.
- To delete the Properties tree, delete the **Move** transform between the input properties tree and the output properties tree.
- To delete a header, add the transport header folder to the output message, and do not set any fields.
- To delete a message body from the output message, add the message body to the output message, and do not set any fields.

What to do next

Configure the message map body parts. For more information, see “Mapping input to output elements manually” on page 70 or “Mapping input to output elements automatically” on page 73.

Initializing a message assembly component in the output message

To initialize a message assembly component in the output message, you add the message assembly component to the output message only. The input values are ignored. You define new values for the elements in the output message assembly structure.

About this task

When you initialize a message assembly structure, you ignore the incoming values and define new values on the output message assembly.

Procedure

To initialize different message assembly components, complete the following steps:

- Add the message assembly component to the output message assembly only.
 - To initialize the local environment tree, add the local environment tree to the output message only.
 - To initialize the Properties tree, delete the **Move** transform between the input properties tree and the output properties tree.
 - To initialize a header, add the transport header folder to the output message only.

For more information, see “Customizing a message map to include a message assembly component.”

- Use mapping transforms to set values for the elements in the new structure.

What to do next

Configure the message map body parts. For more information, see “Mapping input to output elements manually” on page 70 or “Mapping input to output elements automatically” on page 73.

Customizing a message map to include a message assembly component

To customize your message map to include more message assembly components, you must add message assembly components to the input message and to the output message, and then define transforms between them.

Before you begin

Check whether you need additional message assembly components to transform your input message. For more information, see “Choosing a mapping action” on page 98.

About this task

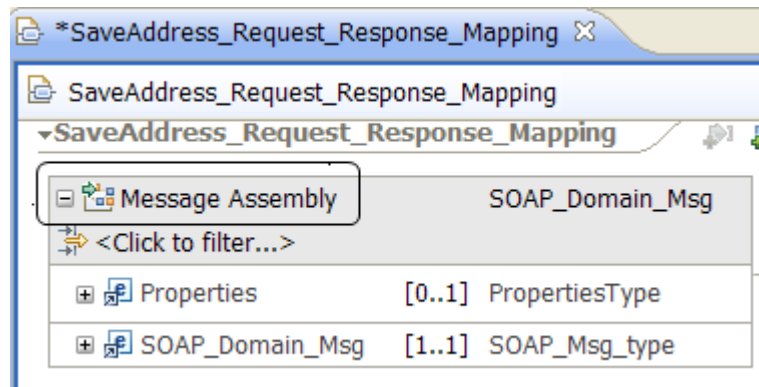
You configure additional message assembly components for a message map in the Graphical Data Mapping editor.

By default, when a message map is created, the only message assembly component that is configured automatically is the Properties tree. The input Properties tree is connected to the output Properties tree with a **Move** transform.

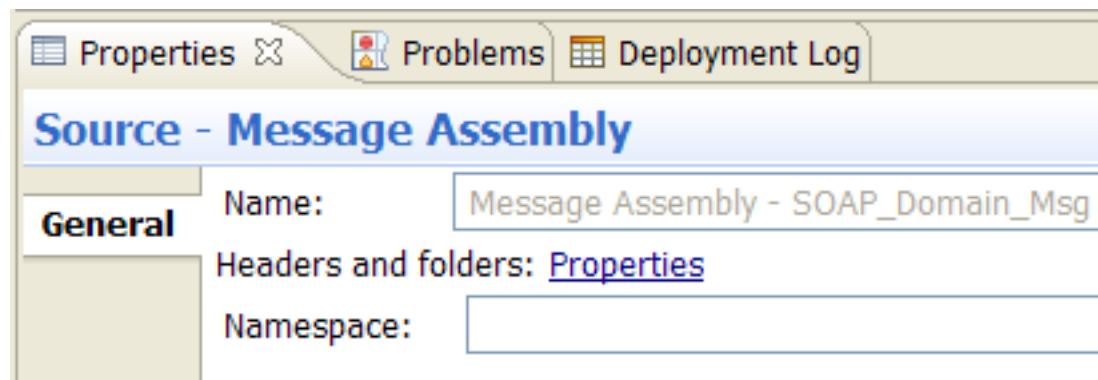
Procedure

To add a message assembly component in a message map, complete the following steps:

1. Open the message map in the Graphical Data Mapping editor.
2. Add a message assembly component such as the local environment tree to the input message.
 - Method 1:
 - a. Select **Message Assembly** .




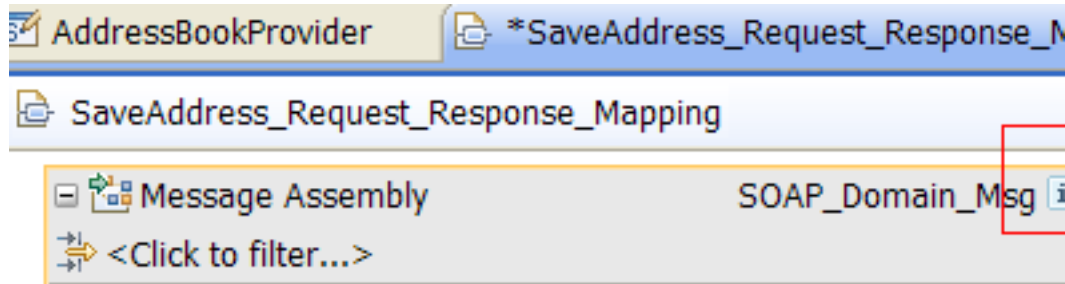
- b. In the Properties view, select the **General** tab.



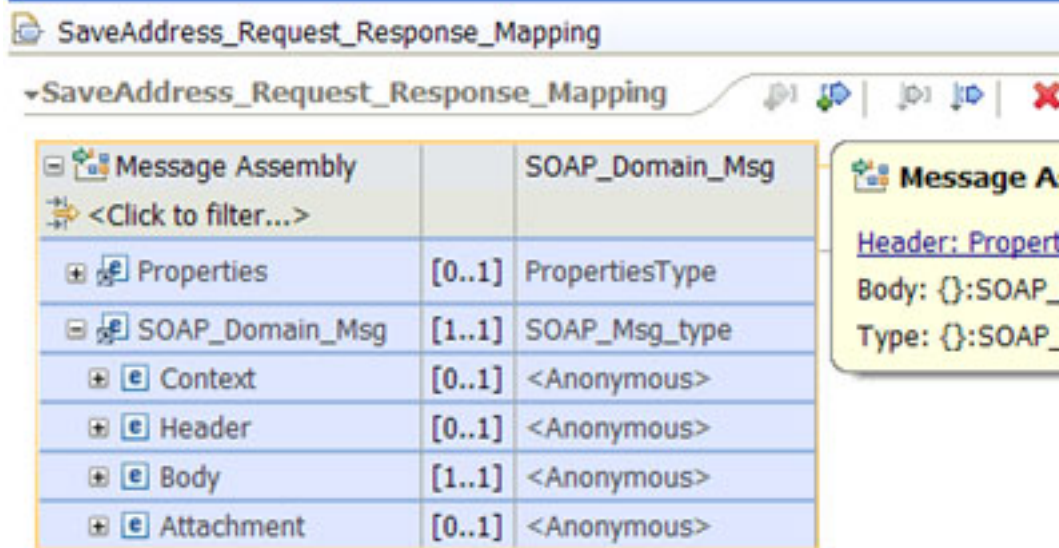
- c. Click **Properties**.

Note: If you have other structures included in your message assembly, the option that you can click includes all the different message assembly components that you have currently selected. For example, if you had the Properties tree and the local environment tree selected, you click **LocalEnvironment, Properties**.

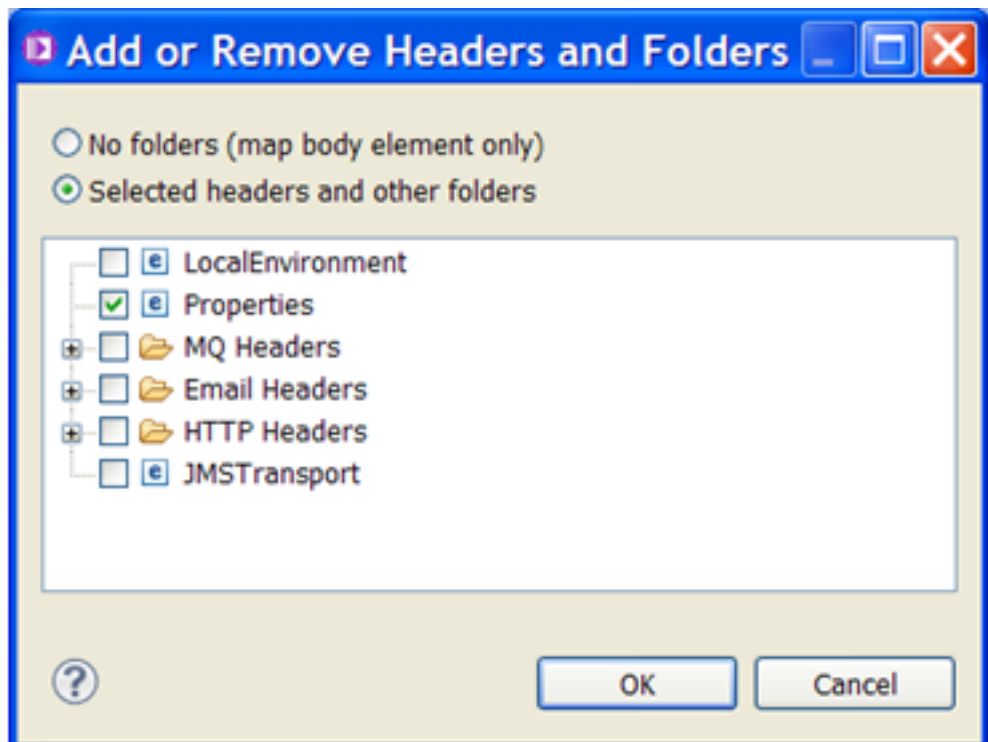
- Method 2:
 - a. Select the information  icon located by the input message body type.



b. Select Header: Properties.



3. In the Add or Remove Headers and Folders window, select one or more message assembly components, and then click OK.



What to do next

Define transforms between the input message assembly and the output message assembly. For more information, see “Specifying a transform (mapping operation)” on page 82.

Configuring the properties of the input and the output message assembly to a message map

You can configure the general properties of the input and output message assembly in a message map by using the Graphical Data Mapping editor.

About this task

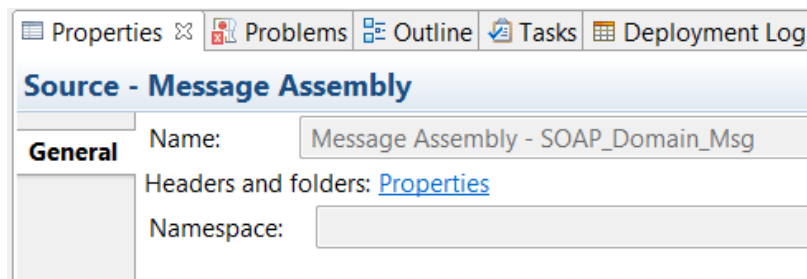
The input and output message assembly properties are located within the **General** tab. They provide information that is displayed for information only, and cannot be modified. In addition, you can configure the message assembly components that you might require to define your data transformations, and the message domain that defines the serializer to convert the logical tree structure into a bit stream.

Procedure

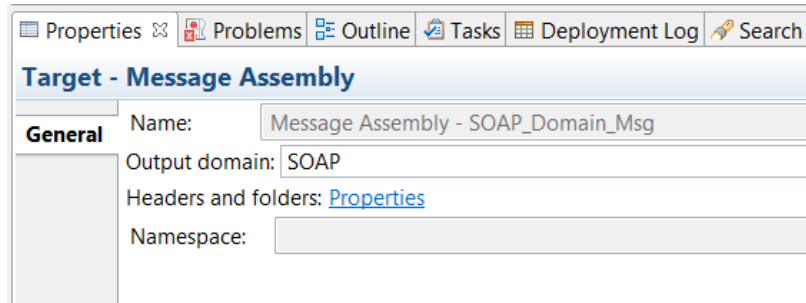
Complete the following steps to configure the general properties of the input and output message assembly to a message map:

- Configure the source message assembly components:
 1. Select the input message assembly in the map, and then select the **Properties** tab.

You can see the **Source - Message Assembly** properties:



2. Add or remove message assembly components to the message map source message assembly.
For more information, see “Customizing a message map to include a message assembly component” on page 105.
- Configure the target message assembly components:
 1. Select an output message assembly in the map, and then select the **Properties** tab.
You can see the **Target - Message Assembly** properties:



2. Specify the **Output domain**. This property defined the parser used to create the output message.
3. Add or remove message assembly components to the message map target message assembly.
For more information, see “Customizing a message map to include a message assembly component” on page 105.

What to do next

Define the map transformations, For more information, see “Specifying a transform (mapping operation)” on page 82.

Mapping transport headers

Use the Graphical Data Mapping editor to transform transport headers.

About this task

When you create a top level message map, only the Properties folder is initially included in the map, and a default transformation from input to output properties is created in a local map. You can then use the Message Assembly properties page in the Graphical Data Mapping editor to modify the transport headers that you might include in the map.

Depending on how you add a transport header to a message map, the component can be deleted, initialized, or transformed:

- To pass unchanged a transport header, do not add it to the message map.
- To read elements from a transport header, add it only to the input message assembly of the message map. The Mapping node passes it through unchanged.
- To modify elements in a transport header, add it to input message assembly and to the output message assembly, and provide transforms to copy and modify the header. The Mapping node deletes the input transport header, and creates a new output transport header containing the result of your transformations.
- To add a transport header to your message, add the header to the output message assembly and populate at least one field. The Mapping node creates a new output transport header containing the results of your transformations.
- To delete a transport header, add it to the output message assembly and do not set any field at all. The Mapping node deletes the transport header from the output message.
- MQ Headers
 - MQMD
 - MQCFH header with root element MQPCF
 - MQCIH

- MQDLH
- MQIIH
- MQMDE
- MQRFH
- MQRFH header with MQRFH2 or MQRFH2C parser
- MQRMH
- MQSAPH
- MQWIH
- SMQ_BMH
- Email Headers
 - EmailOutputHeader
- HTTP Headers
 - HTTPInputHeader
 - HTTPReplyHeader
 - HTTPRequestHeader
 - HTTPResponseHeader
- JMSTransport

Procedure

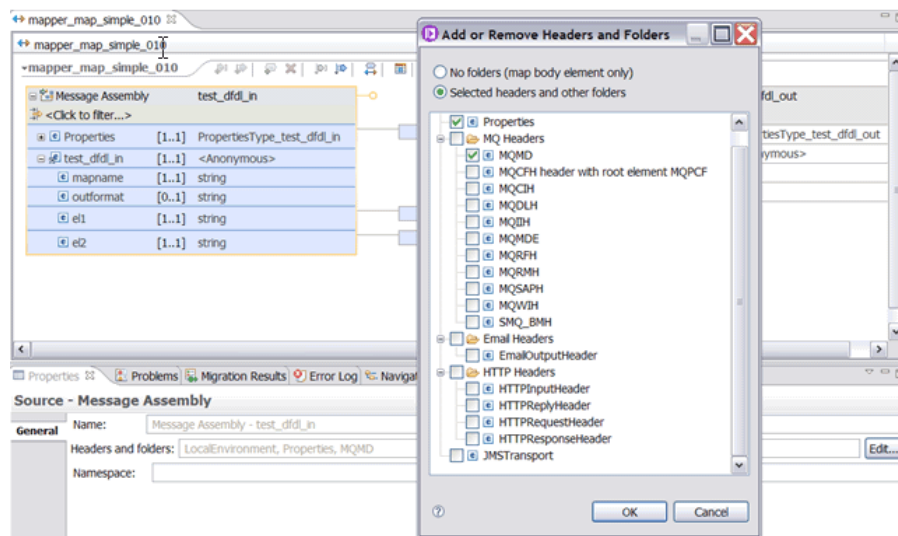
Complete the following steps to transform data in a transport header by using a message map:

1. Decide whether you need a transport header in the input message assembly, the output message assembly, or both. You may want to copy, initialize, or modify elements of a transport header. For more information, see “Choosing a mapping action” on page 98.

Note: The Mapping node copies the transport headers from input to output, unchanged, when they are not included in the message map.

2. Add a transport header to the input message assembly, the output message assembly, or both. For more information, see “Customizing a message map to include a message assembly component” on page 105.

For example, select **MQMD** to include MQMD headers in the input message assembly of the message map:



- Optional: Define a **Move** transform between the input transport header and the output transport header to copy all the input values onto the output values.

Note: If you need to modify only some fields in a transport header, you can use a **Move** transform to copy the transport header unchanged, and then use the **Override** function to modify the elements you must update. For more information, see Chapter 18, “Applying mapping overrides,” on page 157.

You can do this in any of the following ways:

- In the message map, right-click a transport header on the input message assembly, and select **Create Connection**. Move the mouse to the output local transport header, and click the transport header to define the **Move** transform.
 - In the message map, right-click a transport header on the input message assembly, and select **Quick Link**. A new window appears where you can select a transport header as the output element. Use this option when you have a long list of output elements. You can filter the list in Quick link too.
- Define transforms between the input transport header and the output transport header. For more information, see “Specifying a transform (mapping operation)” on page 82 and Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

What to do next

Define additional transformations between other elements in the message map. For more information about the available transforms, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Mapping data in the local environment tree

Use the Graphical Data Mapping editor to transform data graphically in the local environment tree.

Before you begin

Create a message map. For more information, see “Creating a message map” on page 47.

About this task

The local environment tree is a part of the logical message tree in which you can store information while the message flow processes the message. You use the local environment tree to store variables that can be referred to and updated by message processing nodes that occur later in the message flow. You can also use the local environment tree to define destinations (that are internal or external to the message flow) to which a message is sent.

When you add the local environment tree to a message map, you must provide transforms for all of its elements so that the input values of each element are not lost. You can copy the input field unchanged or modified by a transform. Many IBM Integration Bus nodes depend on information in the local environment tree being copied along the message flow.

The variables folder in the local environment tree is defined as *xsd:any*. When you add the local environment tree, you can see the structure of the destination folders with all its elements, and a **Variables** folder with a single element defined with a generic type.

Element	Type
Variables	[0..1] _LocalEnvironmentVariablesType
any	[0..*]

You manually define the elements that are included in the **Variables** folder. There is no predefined structure for the **Variables** folder. Each message flow node has an input and an output local environment tree. There is a **Variables** folder in the input local environment tree and a **Variables** folder in the output local environment tree.

When you create a top level message map, only the Properties folder is initially included in the map, and a default transformation from input to output properties is created in a local map.

You can then use the Message Assembly properties page in the Graphical Data Mapping editor to modify the message assembly components transformed in the map and include the local environment folders in the mapping. For more information, see “Customizing a message map to include a message assembly component” on page 105.

Procedure

Complete the following steps to transform data in the local environment tree by using a message map:

1. Decide whether you need the local environment tree in the input message assembly, the output message assembly, or both. You may want to copy, initialize, or modify LocalEnvironment elements. For more information, see “Choosing a mapping action” on page 98.

Note: The Mapping node copies the LocalEnvironment tree from input to output, unchanged, when they are not included in the message map.

2. Add the local environment to the input message assembly, the output message assembly, or both. For more information, see “Customizing a message map to include a message assembly component” on page 105.
3. Optional: Define a **Move** transform between the input local environment tree and the output local environment tree to copy all the input values onto the output values. Create a connection between the input local environment tree and the output local environment tree.

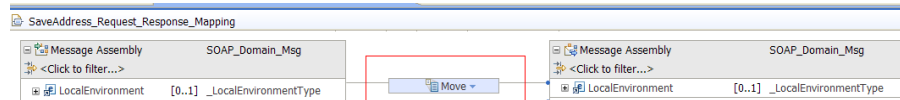
Note: If you need to modify only some fields in the local environment tree, you can use a **Move** transform to copy the local environment tree unchanged, and then use the **Override** function to modify the elements you must update. For more information, see Chapter 18, “Applying mapping overrides,” on page 157.

You can do this in any of the following ways:

- In the message map, right-click **LocalEnvironment** on the input message assembly, and select **Create Connection**. Move the mouse to the output local environment tree, and click **LocalEnvironment** to define the **Move** transform.

- In the message map, right-click **LocalEnvironment** on the input message assembly, and select **Quick Link**. A new window appears where you can select the output element **LocalEnvironment**. Use this option when you have a long list of output elements. You can filter the list in Quick link too.

The following figure shows graphically how the **Move** transform is defined between the input local environment tree and the output local environment tree.



4. Optional: Define the **Variables** folder in the local environment tree. The **Variables** folder contains an `xsd:any` element, that you can redefine by using a **Cast** function and a schema file describing the variables.
 - Define the **Variables** folder in the local environment tree by using the **Cast** function. For more information, see “Configuring the local environment tree **Variables** folder by using the **Cast** function.”
5. Define transforms between the input local environment tree and the output local environment tree. For more information, see “Specifying a transform (mapping operation)” on page 82 and Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

You must provide transforms to copy or modify all fields that are required on the output. Many message flow nodes depend on information in `LocalEnvironment` being copied along the message flow. If you need to modify only some fields, use a **Move** transform to copy `LocalEnvironment` and then use the **Override** function to modify the elements you must update. For more information, see Chapter 18, “Applying mapping overrides,” on page 157.

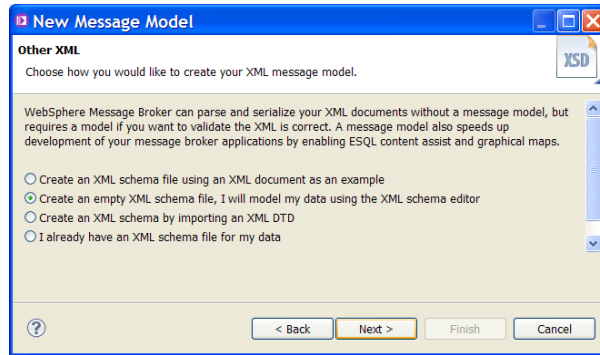
Configuring the local environment tree **Variables** folder by using the **Cast** function

You can use the **Cast** function to define variables in a message map that are defined in the local environment tree **Variables** folder.

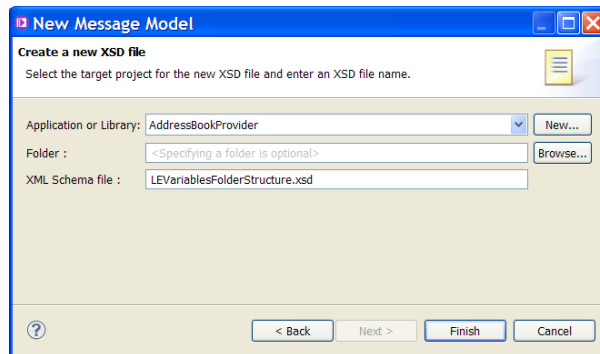
Procedure

Sometimes you need to use information passed in a variable in the local environment tree. In other instances, you might need to calculate the output value of a different element in the message body based on one of the local environment variables. You can also set variables in the output local environment, and use them for routing later in the message flow. To configure the local environment tree **Variables** folder so you can use its elements as part of your transformations, complete the following steps:

1. Create a schema file in your application, integration service, or library to define the elements of the local environment tree **Variables** folder and their type:
 - In the Application Development view, select **New > Message Model > Other XML**. Click **Next**.
 - Select **Create an empty XML schema file, I will model my data using the XML schema editor**, and then click **Next**.



- Create an XSD file *YourLExsdFileName.xsd*, where *YourLExsdFileName* is the name of the file that contains the local environment variables folder message model. Then, click **Finish**.



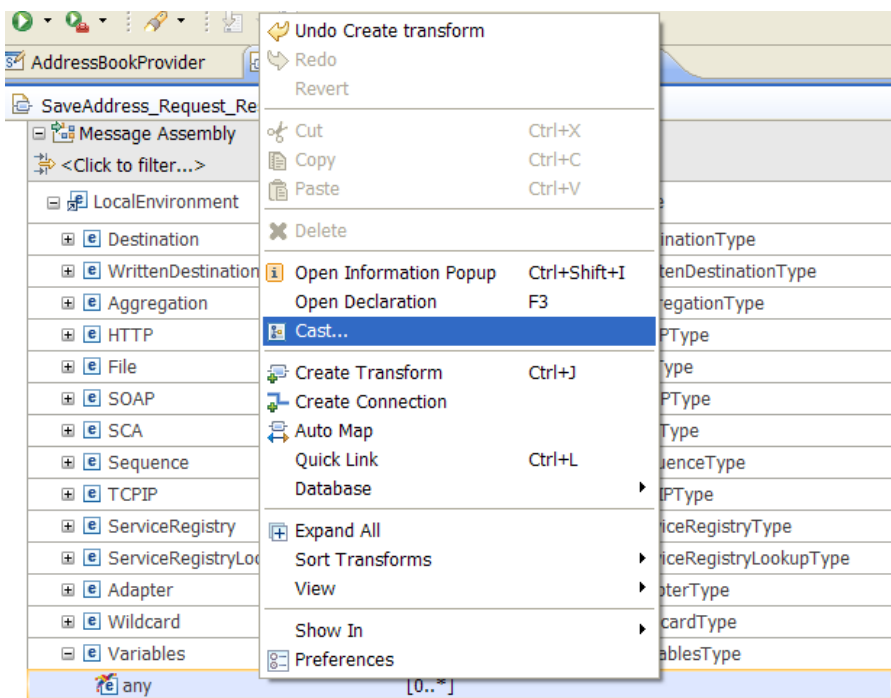
- The XSD file opens in a new tab where you use the XML Schema editor to define your variables and their types.

For example, you can define the following schema:

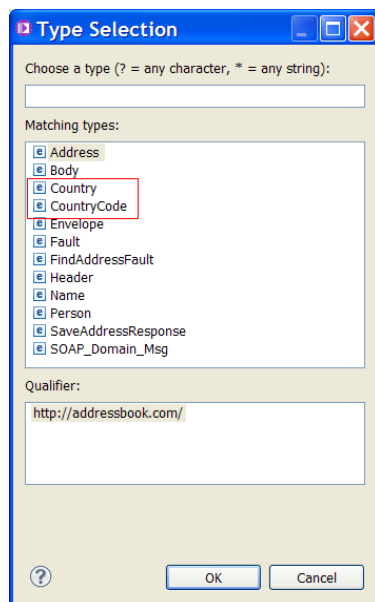
```
<?xml version="1.0" encoding="UTF-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Country" type="xsd:string"/>
  <xsd:element name="CountryCode" type="xsd:integer"/>
</xsd:schema>
```

Note: You can define the local environment variables in a message flow node by using ESQL or Java. Namespaces is not defined. For this reason, the schema is also defined without a namespace declaration.

2. Use the **Cast** function to define the local environment variables in the message map so they are visible under the **Variables** folder in the map. Complete the following steps to cast the **any** element to a variable and its type in the output local environment tree:
 - Right-click the **any** element, and then select **Cast**.



- In the Type Selection window, select a type, for example **Country**, and then click **OK**.



Results

You have defined one local environment variable that can be used by other message flow nodes in your message flow for routing or filtering.

You can see the element **Country** under the local environment **Variables** folder in the message map.

[-] [e] Variables	[0..1] _LocalEnvironmentVariablesType
[e] any	[0..*]
[e] Country	[0..*] string

Example

Another example:

If you set in an ESQL compute node two simple fields within the **Variables** folder of the Local Environment tree by using the following code:

```
SET Outputlocal environment.Variables.dec = 10.1;
SET Outputlocal environment.Variables.str = 'Some text';
```

To access these fields in a Mapping node by using the **Cast** function, you must create a schema file in your integration solution to define the elements and their type. Note that since the ESQL is not using any namespace to qualify these elements, the schema is also defined without a namespace declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="dec" type="xsd:decimal"/>
  <xsd:element name="str" type="xsd:string"/>
</xsd:schema>
```

Once the schema file is saved, you can then select the any element under the Variables section in the Local Environment tree, and use the context menu action **Cast** to add a Mapping cast for each of the elements "dec" and "str" that are required in the message map. For more information, see “Mapping xsd:any on an input or output message” on page 64.

What to do next

Define transforms between the input local environment tree and the output local environment tree. For more information, see “Specifying a transform (mapping operation)” on page 82 and Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Adding database definitions to the IBM Integration Studio

You must have a database definition (.dbm file), contained in a data design project, to create database mappings by using the Mapping node.

About this task

A database definition file holds the physical data model that details all the database resources, such as the schema, the tables, and other resources, that you need access to.

If you can connect to your database server by using the IBM Integration Studio, you can create a database definition when you create your database mapping. For more information, see Chapter 19, “Mapping database content,” on page 159.

If you cannot connect to your database server by using the IBM Integration Studio, you must create a database definition file from scratch before you can create your database mapping. For more information, see “Creating a database definition from scratch” on page 120.

You can also use database definitions in other nodes, such as the Compute node, to validate references to database sources and tables. You must include a data design project in an application, or reference it from an Integration project, before you can use the database definitions that the data design project contains.

Database definition files in the IBM Integration Studio are not automatically updated. If you modify your database, you must re-create the database definition files.

The following topics describe how to add a database definition to the IBM Integration Studio:

- “Creating a data design project”
- Creating a database definition by using the New Database Definition File wizard to connect to a database server.
- Create a database definition from scratch when it is not possible to connect to the database server by using the IBM Integration Studio.
- “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118
- “Creating a database definition from scratch” on page 120

Creating a data design project

Create a data design project to contain your database definition files.

About this task

A data design project is a specialized type of project where you store database definition files that hold information about database resources.

To create a data design project in the Integration Development perspective, complete the following steps:

Procedure

1. Click **File > New > Other**. A window opens in which you can select a wizard.
2. Expand **Data**, select **Data Design Project**, and click **Next**. The New Data Design Project wizard opens.
3. Enter a name for your data design project, and then click **Finish**. The Open associated perspective dialog is displayed.
4. Click **No**. Your data design project is created, and is displayed in the Application Development view, under **Independent resources**.

What to do next

After you create a data design project, you can complete the following tasks:

- Create a database definition by using the New Database Definition File wizard to connect to a database server; see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118.
- Create a database definition from scratch when it is not possible to connect to the database server by using the IBM Integration Studio; see “Creating a database definition from scratch” on page 120.
- Include your data design project in an application; see Adding a project to an application, integration service, or library.

- Reference your data design project from an Integration project; see Referencing resources in other libraries.

Creating a database definition (.dbm file) by using the New Database Definition File wizard

Use the New Database Definition File wizard to add database definitions to the IBM Integration Studio.

Before you begin

Note: Your database server must be configured to listen on a static port; dynamic ports are not supported with the New Database Definition File wizard. Contact your database administrator for information on how to verify port configuration.

About this task

A database definition file holds the physical data model that details all the database resources, such as the schema, the tables, and other resources, that you need access to.

If you can connect to your database server by using the IBM Integration Studio, you can create a database definition (.dbm file) when you create your database mapping. For more information, see Chapter 19, “Mapping database content,” on page 159.

If you cannot connect to your database server by using the IBM Integration Studio, you must create a database definition file from scratch before you can create your database mapping. For more information, see “Creating a database definition from scratch” on page 120.

You can also use database definitions in other nodes, such as the Compute node, to validate references to database sources and tables. You must include a data design project in an application, or reference it from an Integration project, before you can use the database definitions that your data design project contains.

Database definition files in the IBM Integration Studio are not automatically updated. If you modify your database, you must re-create the database definition files.

Complete the following steps to create a database definition (.dbm file) by using the New Database Definition File wizard:

Procedure

1. Click **File > New > Database Definition**. The New Database Definition File wizard is displayed.
2. Select an existing data design project, or click **New** to create a data design project.
3. From the **Database** drop-down list, select the type of database that you want to model and, if applicable, select the **Version**. Click **Next**.

Important: Ensure that you select a database from the list that is supported by IBM Integration Bus; you can use this wizard in a shell-share environment with

other Rational® products that support other databases or versions. For a list of databases supported by IBM Integration Bus, see IBM Integration Bus requirements.

If your database is supported by IBM Integration Bus, but is not included in the list of selectable databases, you might need to create a database definition from scratch; see “Creating a database definition from scratch” on page 120.

When using specific database servers, you might need to set the paths of your JDBC JAR files before you can use them with the New Database Definition File wizard. To set the paths of your JDBC JAR files:

- a. Click **Window > Preferences**. The Preferences window opens.
 - b. Expand **Data ManagementConnectivityDriver Definitions**.
 - c. In the Driver Definitions pane, select the database server that you want to connect to, and click **Edit...** The Edit Driver Definition window opens.
 - d. Select the **JAR List** tab, and click **Add JAR/Zip...**
 - e. Browse to the JDBC JAR file that was supplied with your database product, select the JAR file, and then click **Open**. The Edit Driver Definition window closes.
 - f. In the Preferences window, click **OK** to close the window, and return to the New Database Definition File wizard.
4. Select a connection to use from the list of existing connections, or click **New** to create a database connection. If you select to use an existing connection, the existing database definition is overwritten.
 5. If you selected to create a connection:
 - a. Optional: If you want to enter a custom value for the *Connection Name*, clear **Use default naming convention**.
 - b. Enter values for the Connection to the database, for example, *Database name*, *Host name* and *Port number*.
 - c. Enter values for the *User ID* and *Password* to connect to the database. Click **Test Connection** to verify the settings that you selected for your database. The default Port number for a DB2 database is 50000.
 - d. Click **Next**. If your data design project already contains a database definition for the database that you selected, and you want to overwrite this database definition, click **Yes** in the Confirm file overwrite window.

Tip: If you cannot connect to your database by using the New Database Definition File, you might need to create you database definition from scratch; see “Creating a database definition from scratch” on page 120.
 6. Alternatively, if you selected to use an existing connection:
 - a. To overwrite the existing database definition, click **Yes** in the Confirm file overwrite dialog.
 - b. Enter values for the *User ID* and *Password* to connect to the database, and then click **Next**.
 7. Select one or more database schemas from the list and click **Next**. When you create a database definition for use in a graphical data map called from a Mapping node, the database schema name that is displayed in the map is the one that you select here, but might be overridden by the corresponding JDBCProvider configurable service.
 8. Select the elements that you require on the Database Elements page. You can select any option, in addition to **Tables**, on the Database Elements page.
 - a. Select **Views** to see all the database views in the Data Project Explorer

- b. Select **Routines** to add stored procedures and user-defined functions to the database definition file.

If you select additional options, the database definition files that you create contain more information than the Compute, Database, or Mapping nodes require.

9. Click **Finish**.

Results

A new database definition (.dbm file) is added to your data design project. If you opened the New Database Definition File wizard while creating a database transform in the graphical data mapping editor, you are returned to the database transform wizard.

What to do next

Before you can use your database definition in a messaging solution, you must complete one of the following tasks:

- Include your data design project in an application; see Adding a project to an application, integration service, or library.
- Reference your data design project from an Integration project; see Referencing resources in other libraries.
- Create a corresponding JDBCProvider configurable service; see Configurable services properties.

Creating a database definition from scratch

You can create a database definition (.dbm file) from scratch. A database definition is required to create database mappings.

Before you begin

- You must have created a data design project; see “Creating a data design project” on page 117

About this task

A database definition file holds the physical data model that details all the database resources, such as the schema, the tables, and other resources, that you need access to.

You must have a database definition (.dbm file) contained in a data design project before you can create database mappings. You can also use database definitions in other nodes, such as the Compute node, to validate references to database sources and tables. You must include a data design project in an application, or reference it from an Integration project, before you can use the database definitions that your data design project contains.

Database definition files in the IBM Integration Studio are not automatically updated. If you modify your database, you must re-create the database definition files.

Complete the following steps to create a database definition (.dbm file) from scratch:

Procedure

1. Click **File > New > Other**. A window opens in which you can select a wizard.
2. Expand **Data**, select **Physical Data Model**, and click **Next**. The New Physical Data Model wizard opens.
3. Next to the *Destination folder* field, click **Browse...** A window opens in which you can select a parent folder for your database definition.
4. Select a data design project from the list, and then click **OK**.

Important: Ensure that you select a data design project as the parent folder for your database definition. Database definitions must be contained in a data design project before you can use them in your IBM Integration Bus messaging solutions.

5. In the *File name* field, enter a name to represent the database that you want to model. You do not need to select the **Database** or **Version**.

Important: Create a database definition only if your database is supported by IBM Integration Bus.

For a list of databases supported by IBM Integration Bus, see IBM Integration Bus requirements.

6. Select **Create from template**, then click **Next**.
7. In the Templates pane, select **Empty Physical Data Model**, then click **Finish**. Your empty database definition is created, is displayed in the Data Project Explorer view, and is opened in the Physical Data Model editor.
8. If the Data Project Explorer view is not open in your IBM Integration Studio, open it:
 - a. Click **Window > Show View > Other**. A window opens in which you can select a view.
 - b. Expand **Data Management**, select **Data Project Explorer**, and click **OK**. The Data Project Explorer view opens.
9. In the Data Project Explorer view, expand your database definition and select **Database**. The database properties are displayed in the Properties view.
10. In the Properties view, select the **General** tab. In the *Name* field, enter the name of your database. If you use this database definition with the Graphical Data Mapping editor, the *Name* is displayed as the name of the data source, and is used when creating the JDBC configurable service for the IBM Integration Bus runtime connection.
11. In the Data Project Explorer view, select **Schema**. The database schema properties are displayed in the Properties view.
12. In the Properties view, select the **General** tab. In the *Name* field, enter the name of your database schema. Database schemas are used only by the Mapping node, and only when calling a graphical data map that contains a database transform. For more information about mapping database content, see Chapter 19, "Mapping database content," on page 159.
13. In the Data Project Explorer view, right-click **Schema** and select **Add Data Object > Table**. A **Table** is created, and is displayed under your **Schema** in the Data Project Explorer view.
14. Define the columns in your **Table**:
 - a. In the Data Project Explorer view, right-click your **Table**, and select **Add Data Object > Column**. A **Column** is created, and is displayed under your **Table** in the Data Project Explorer view.

- b. Enter a name for your **Column**.
 - c. In the Properties view, select the **Type** tab to define the attributes of your column.
15. Repeat steps 13 on page 121 through 14 on page 121 for each table in your database, and then save your database definition.
16. Save your database definition, and close the Physical Data Model editor.

What to do next

Before you can use your database definition in a messaging solution, you must complete one of the following tasks:

- Include your data design project in an application; see Adding a project to an application, integration service, or library.
- Reference your data design project from an Integration project; see Referencing resources in other libraries.

Accessing integration node properties from a Mapping node

To obtain the value of an integration node property, call the appropriate function from a **Custom XPath** transform. These functions return a string with the value of the integration node property.

About this task

Procedure

To access integration node properties from a Mapping node, choose one of the following functions:

- Call **mb:getBrokerName()** to get the name of the integration node where the Mapping node that is executing the message map is running.
- Call **mb:getQueueManagerName()** to get the name of the default queue manager of the integration node where the Mapping node that is executing the message map is running.
- Call **mb:getNodeName()** to get the name of the Mapping node from the **Node name** property of the node in the message flow.
- Call **mb:getMessageFlowName()** to get the name of the message flow where the Mapping node that is executing the message map is running.
- Call **mb:getApplicationName()** to get the name of the application where the message flow that contains the Mapping node that is executing the message map is running.
- Call **mb:getLibraryName()** to get the name of the library where the message flow that contains the Mapping node that is executing the message map is running.

Accessing user-defined properties from a Mapping node

A Mapping node can access properties that you have associated with the message flow that contains the node.

About this task

To access these properties from a Mapping node, call the function **mb:getUserDefinedProperty("propertyname")** from a custom XPath mapping. The

function returns a string that contains the property value, regardless of the original type of the property.

Chapter 11. Setting the value of an output element by using a transform or a function

Use the Graphical Data Mapping editor to set the value of an output element by using an expression, a transform, or a function.

About this task

You can use a function or a transform to set the value of an output element, either by connecting the output element with an input element and then specifying a transform on the connection between them, or by specifying a transform directly on the output element. For information about creating connections and specifying transforms, see Chapter 9, “Editing message maps,” on page 61.

For information about all the functions and transforms that are available, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Transforms that support conditional control such as the **If** transform can use XPath 2.0 expressions, or invoke Java or ESQL functions.

Procedure

You can use any of the following mapping operations to map graphically your data in the Graphical Data Mapping editor

- **Core mapping transforms:** You can use built-in structural and functional mapping operations, which enable you to graphically construct the required message transformations to build the output message. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.
- **Custom transforms:** You can use custom transformations to build your own XPath 2.0, Java, or ESQL functions, which can be invoked to perform specialized transformations within the message map. For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.
 - Custom Java transform. For more information, see “Custom Java” on page 212.
 - Custom XPath transform. For more information, see “Custom XPath” on page 214.
 - Custom ESQL transform. For more information, see “Custom ESQL” on page 208.
- **XPath functions** (*fn:functionName*): You can use XPath 1.0 and XPath 2.0 functions to transform data in a message map. For more information about XPath, see the online document W3C XML Path Language (XPath) 2.0.

Note: XPath 1.0 functions are valid XPath 2.0 expressions. You can use the XPath Expression Builder to generate simple XPath 1.0 expressions.

- **Database transforms:**
 - You can use the **Select** transform to query one or more database tables, and retrieve data that you can use in the message map to set output element values, define conditions, or use as input to build other transforms conditions.

Database tables can be set as additional outputs of a message map. For more information, see “Selecting data from a table” on page 159.

- You can use a **database routine** transform to call a stored procedure or user-defined function from a database, and retrieve data that you can use in the message map to set output element values, define conditions, or use as input to build other transforms conditions.

Note: Only IBM DB2 stored procedures are supported in IBM Integration Bus. For more information, see “Calling a stored procedure from a map” on page 168.

What to do next

Define transformations to set the value of output elements in your output message:

- Use the **Assign** transform to set the value of an output element to a constant. For more information, see “Setting the value of an output element to a simple data type.”
- Use any of the **xs:type** transforms to cast the value of a simple type input element and set the value of a simple type output element. For more information, see “Setting the value of an output element with a explicit data type” on page 128.
- Use the **Create** transform to set the value of an output element that is defined as a simple type or as part of a complex data type. You use the default or the fixed value defined in the schema for that element. For more information, see “Setting the value of a simple output element to a default or fixed value” on page 130.
- Use the **Assign** transform or the **Create** transform to create an empty output element defined as a string or as a hexBinary element. For more information, see “Creating an empty output element” on page 134.
- Use the **Create** transform, the **Move** transform, the **Custom Java** transform, the **Custom ESQL** transform, or the return value of a database to create a null output element. For more information, see “Creating a nil output element” on page 132.
- Use the `fn:nilled` XPath function, and the `fn:exists` XPath function to define conditional expressions that determine whether an input element is nilled, or is present. For more information, see “Choosing an XPath conditional expression that tests for a nil value in a transform” on page 91.
- Use the `fn:empty` XPath function to define conditional expressions that determine whether an input element is empty. For more information, see “Creating an empty output element” on page 134.

Setting the value of an output element to a simple data type

In the Graphical Data Mapping editor, use the **Assign** transform to set the value of an output element to a constant.

About this task

You cannot use the value of an input element to set the value of an output element in an **Assign** transform.

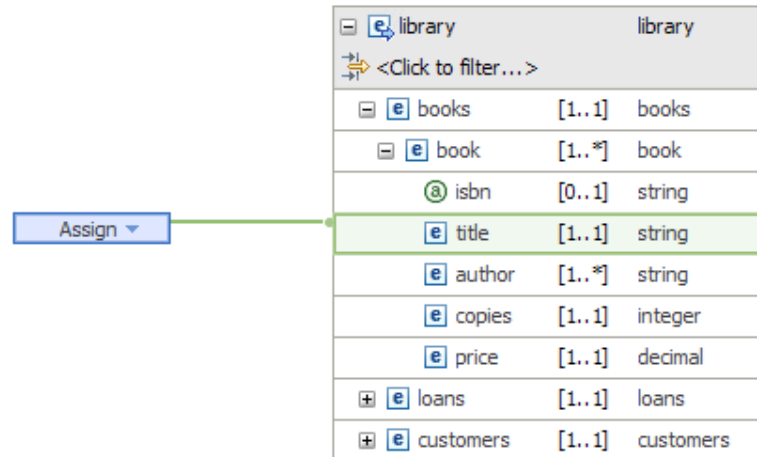
However, you can define **supplement** connections between input elements and the **Assign** transform. You can then use these input elements in a conditional expression that defines the condition under which the transform should be applied.

The output element is set to a constant value.

Procedure

Complete the following steps to set the value of an output element to a constant:

1. Create an **Assign** transform for the output element, by using either of the following methods:
 - Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
 - Right-click the output element and then select **Create Assign**.

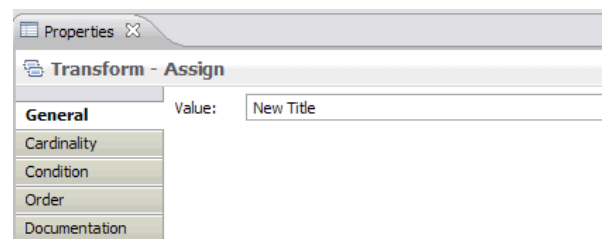


2. Enter the required constant into the **Value** field in the Properties panel for the **Assign**.

The constant value can be any simple data type value.

Note: Do not enclose string literals in single or double quotes; enter the required text in the entry field without quotes.

The following figure shows the Properties view of the **Assign** transform:



Results

If you do not specify a value, an empty element is created.

By default, the following values are set for an output element when you use the **Assign** transform:

Table 9. Default values set by using the **Assign** transform

Type	Default value
string	Empty string
dateTime	2002-01-01T11:00:00

Table 9. Default values set by using the **Assign** transform (continued)

Type	Default value
boolean	false
decimal	0.0
double	0.0
hexBinary	00
long	0
duration	P1Y
time	00:00:00
date	2002-01-01

Setting the value of an output element with an explicit data type

In the Graphical Data Mapping editor, you can use any of the **xs:type** transforms to cast the value of a simple type input element and set the value of a simple type output element.

About this task

For example, you want to assign a value with a specific data type to a target element that is defined as `xs:anySimpleType`. You can use the **xs:type** transform.

You can have zero or more input elements to an **xs:type** transform. However, you can only cast one of the input elements to set the value of an output element by using an **xs:type** transform.

You can use the input elements to build a conditional expression that determines if the **xs:type** transform is applied or not.

You must choose the **xs:type** transform according to the output element data type. For example, if you have an output element with a boolean data type, you must choose **xs:boolean** transform.

You can use any of the following `xs:any` transforms:

- **xs:NOTATION**: This function takes a primitive and casts it as notation.
- **xs:Qname**: This function takes a primitive and casts it as Qname.
- **xs:anyURI**: This function takes a primitive and casts it as anyURI.
- **xs:base64Binary**: This function takes a primitive and casts it as base64Binary.
- **xs:boolean**: This function takes a primitive and casts it as boolean.
You can use any of the following values: true or false.
- **xs:dateTime**: This function takes a primitive and casts it as dateTime.
- **xs:date**: This function takes a primitive and casts it as date.
- **xs:dayTimeDuration**: This function takes a primitive and casts it as dayTimeDuration.
- **xs:decimal**: This function takes a primitive and casts it as decimal.
- **xs:double**: This function takes a primitive and casts it as double.
- **xs:float**: This function takes a primitive and casts it as float.
- **xs:gDay**: This function takes a primitive and casts it as gDay.

- **xs:gMonthDay**: This function takes a primitive and casts it as gMonthDay.
- **xs:gMonth**: This function takes a primitive and casts it as gMonth.
- **xs:gYearMonth**: This function takes a primitive and casts it as gYearMonth.
- **xs:gYear**: This function takes a primitive and casts it as gYear.
- **xs:hexBinary**: This function takes a primitive and casts it as hexBinary.
- **xs:integer**: This function takes a primitive and casts it as integer.
- **xs:int**: This function takes a primitive and casts it as signed 32-bit integer.
- **xs:string**: This function takes a primitive and casts it as string.
- **xs:time**: This function takes a primitive and casts it as time.
- **xs:yearMonthDuration**: This function takes a primitive and casts it as yearMonthDuration.

Procedure

Complete the following steps to set the value of a output element by using an **xs:type** transform:

1. Add an **xs:type** transform, for example, an **xs:boolean** transform, to set the value of the output element, by using either of the following methods:
 - a. Create an **Assign** transform by using either of the following methods:
 - Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
 - Right-click the output element and then select **Create Assign**.
 - b. Use the drop-down, and select **Cast Function > xs:boolean**.
2. Enter true into the **Value** field in the Properties panel for the **xs:boolean** transform.

You can set the value to true or false.

The screenshot shows the XMLBeans IDE interface. At the top, a canvas displays a transform diagram with an input element 'CurrentDate' (type 'date') connected to an 'xs:boolean' transform, which is then connected to the 'ReturnStatus' output element (type 'boolean'). Below the canvas, the 'Properties - boolean' panel is open, showing the 'General' tab with the following details:

Description:	
Cardinality	Takes a primitive and casts it as a boolean.
Variables	Parameters:
Condition	
Sort	
Order	
Documentation	

Below the table, there is a table with columns 'Name', 'Type', and 'Value':

Name	Type	Value
primitive	xs:anyAtomicTy...	true

Buttons for 'Add', 'Edit...', and 'Remove' are visible to the right of the table.

3. Optional: Define a conditional expression to control when the transform should be applied .

You define a conditional expression by using an XPath expression.

For example, to evaluate the transform, the current date has to be a date after the date specified in the element **CurrentDate**.

This screenshot is similar to the previous one, but the 'Condition' field in the 'Properties - boolean' panel is now populated with the XPath expression: `$CurrentDate > f:current-date()`.

Setting the value of a simple output element to a default or fixed value

In the Graphical Data Mapping editor, use the **Create** transform to set the value of a simple output element. You use the default or the fixed value defined in the schema for that element.

About this task

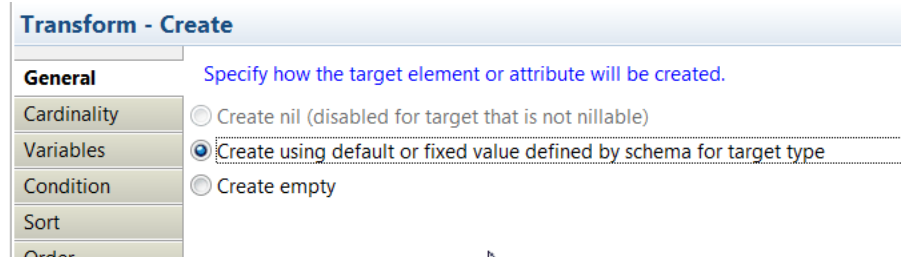
The **Create** transform creates an output element without the use of input data.

Note: You can only set the value of a simple output element to its default or fixed value when the schema includes a default value or a fixed value for the element.

Procedure

Complete the following steps to create a simple output element with its default or fixed value:

- Create an **Assign** transform by using either of the following methods:
 1. Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
 2. Right-click the output element and then select **Create Assign**.
- Use the drop-down, and select **Core transforms > Create**.
- In the Properties panel, select the **General** tab. Then, select **Create using default or fixed value defined by schema for target type**.



Setting the value of a simple type element included in a complex type output structure to a default or fixed value

In the Graphical Data Mapping editor, use the **Create** transform to set the value of an output element that is defined as part of a complex data type. You use the default or the fixed value defined in the schema for that element.

About this task

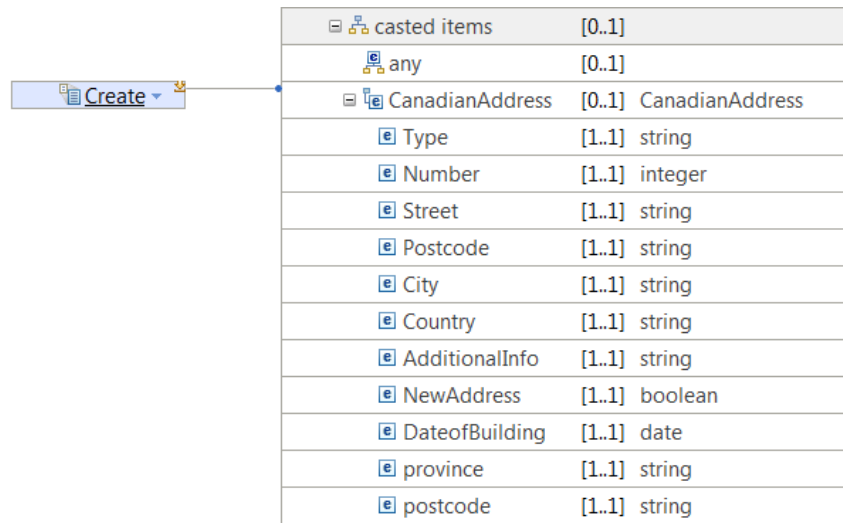
The **Create** transform creates an output element without the use of input data.

Note: You can only set the value of a simple output element to its default or fixed value when the schema includes a default value or a fixed value for the element.

Procedure

Complete the following steps to create a simple type element included in a complex type output structure with its default or fixed value:

- Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, a **Create** transform is created.



- Open the nested map associated to the **Create** transform. You can define how each target element or attribute will be created within the complex element. For more information, see “Setting the value of a simple output element to a default or fixed value” on page 130.

You can use in the **Create** transform nested map more **Create** transforms, **Assign** transforms, or any other mapping transforms that do not require an input. You can use this method to populate as many fields as required in the complex output structure.

Transform - Create

Specify how the target element or attribute will be created.

General

Cardinality Create nil (disabled for target that is not nillable)

Variables Create using default or fixed value defined by schema for target type

Condition Create empty

Creating a nil output element

In the Graphical Data Mapping editor, you can use the **Create** transform, the **Move** transform, the **Custom Java** transform, the **Custom ESQL** transform, or the return value of a database to create a nil output element.

Before you begin

When you map null values, consider the Graphical Data Mapping editor behavior. For more information, see Chapter 5, “Handling nulls in message maps,” on page 35.

Procedure

Choose any of the following methods to create a nil output element:

- Use the **Create** transform. Create a nil element without the use of input data.
- Use the **Move** transform. Copy an input element that is nil to an output nil element.
- Use the **Custom Java** transform. The Java method that you implement must return an MbElement that is nil.
- Use the **Custom ESQL** transform. Your ESQL code must return a NULL value.
- Use the database transform, and **Move** a nullable column to an output nillable element.
- Use the XPath function `iib:nullValue()` to set an XMLNSC element to nil, a JSON object to NULL and a JSON array to NULL.

Example

Example: Use the Create transform to create a simple type output element as nil

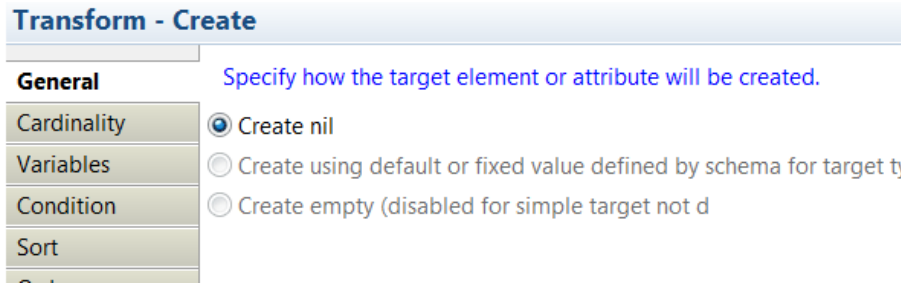
This example shows how to use the **Create** transform to create a simple type output element as nil in the Graphical Data Mapping editor.

You can use the **Create** transform to create an output element with `xsi:nil="true"`, also called a nil element, without the use of input data.

Note: The option to create a nil element is available only when the output element is nillable.

Complete the following steps to create a simple nil output element:

1. Add a **Create** transform to set the value of the output element.
 - a. Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
 - b. Select the drop-down, and select **Core transforms > Create**.
2. In the Properties panel, select the **General** tab. Then, select **Create nil**.



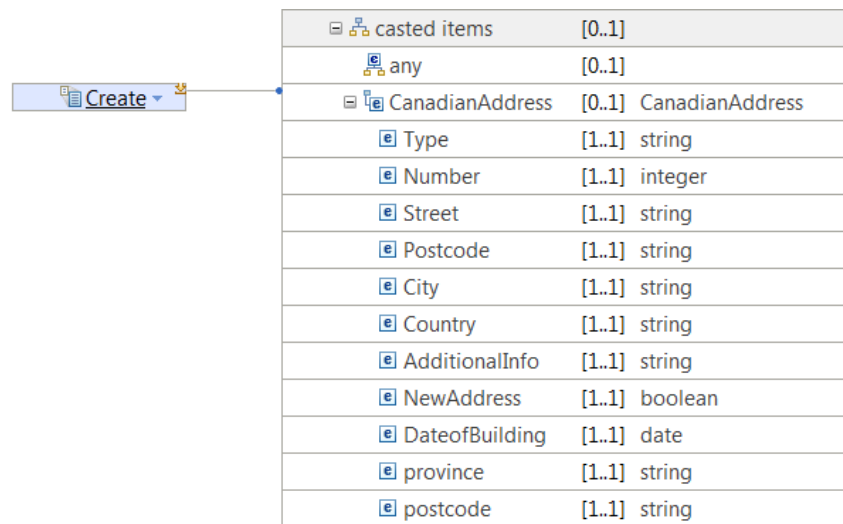
Example: Use the **Create** transform to create a complex type output element as nil

This example shows how to use the **Create** transform to create a complex type output element as nil in the Graphical Data Mapping editor.

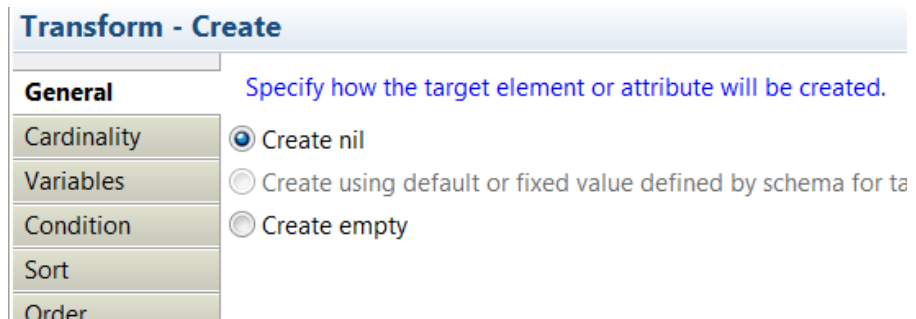
Note: The option to create a nil element is available only when the output element is nillable.

Note: The **Create** transform has a nested map, however, you cannot create or assign fields within a complex nil output element. You will get the following error: Cannot create or assign fields within nil target. To allow mapping, change parent Create transform to create empty instead of nil.

1. Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, a **Create** transform is created.



2. In the Properties panel, select the **General** tab. Then, select **Create nil**.



Creating an empty output element

In the Graphical Data Mapping editor, you can use the **Assign** transform or the **Create** transform to create an empty output element defined as a string or as a hexBinary element when you have no source to copy from.

About this task

You can define an empty output element when the following conditions are met:

- The data type of the element is string or hexBinary.
- The element is not nillable.

Procedure

Choose any of the following methods to create an empty output element:

1. Use the **Assign** transform to create an empty output element. For more information, see “Initializing an output element by using the **Assign** transform.”
2. Use the **Create** transform to create a simple or complex empty output element. For more information, see “Initializing a simple or complex output element by using the **Create** transform” on page 135.

Results

The following table lists the data types and the transforms that you can use to create an output element:

Table 10. List of transform types that you can use to create an empty element

Data type	Valid transforms
Simple data types: string, hexBinary	Assign, Create
Complex data types	Create

Initializing an output element by using the Assign transform

In the Graphical Data Mapping editor, use the **Assign** transform to create an empty output element.

About this task

There is no input element to an **Assign** transform.

You can only create an empty output element for the following output element data types:

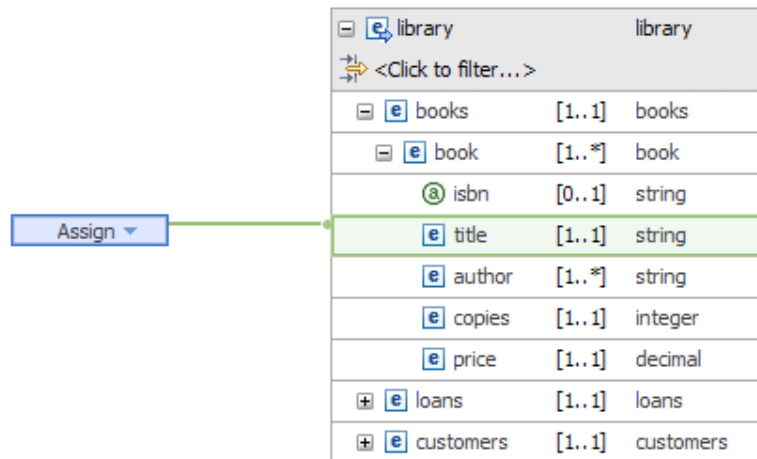
- string
- hexBinary

Procedure

Complete the following steps create an empty output element:

1. Create an **Assign** transform for the output element, by using either of the following methods:

- Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
- Right-click the output element and then select **Create Assign**.



2. Do not specify a value in the **Value** field in the Properties panel for the **Assign** transform.

Initializing a simple or complex output element by using the Create transform

In the Graphical Data Mapping editor, use the **Create** transform to create a simple or complex empty output element.

About this task

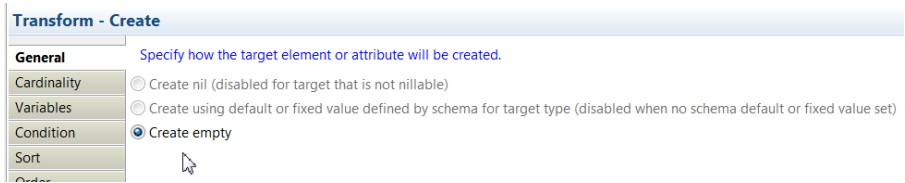
The **Create** transform creates an output element without the use of input data. You can create either simple or complex type output elements.

You can create a simple empty output element for elements defined as string or hexBinary.

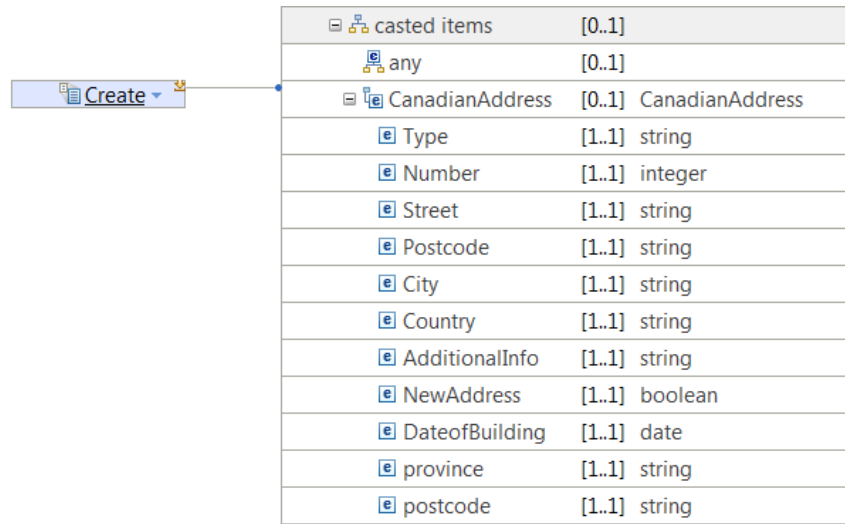
Procedure

Complete the following steps to create a simple or complex empty output element:

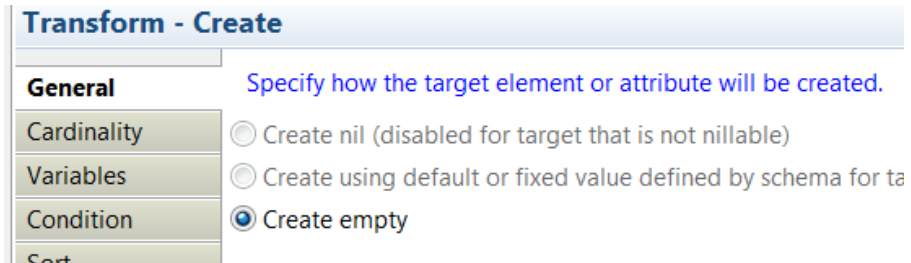
- For simple type output elements, complete the following steps:
 1. Add a **Create** transform to set the value of the output element, by using either of the following methods:
 - Create an **Assign** transform by using either of the following methods:
 - Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, an **Assign** transform is created.
 - Right-click the output element and then select **Create Assign**.
 - Use the drop-down, and select **Core transforms > Create**.
 2. In the Properties panel, select the **General** tab. Then, select **Create empty**.



- For complex type output elements, complete the following steps:
 1. Click the output element and drag the connector towards the input elements, releasing the connector onto a blank part of the canvas. When you release the connector, a **Create** transform is created.



2. In the Properties panel, select the **General** tab. Then, select **Create empty**.



3. Optional: Open the nested map associated to the **Create** transform. You can define how each target element or attribute will be created in a complex element.

You can use the **Create** transform to create an empty complex output element. then, you can enter the **Create** transform nested map and use more **Create** transforms, **Assign** transforms, or any other mapping transforms that do not require an input. You can use this method to populate as many fields as required in the complex output structure.

CanadianAddress		CanadianAddress
<Click to filter...>		
Type	[1..1]	string
Number	[1..1]	integer
Street	[1..1]	string
Postcode	[1..1]	string
City	[1..1]	string
Country	[1..1]	string
AdditionalInfo	[1..1]	string
NewAddress	[1..1]	boolean
DateofBuilding	[1..1]	date
province	[1..1]	string
postcode	[1..1]	string

Properties Problems Outline Tasks Deployment Log

Transform - Create

General Specify how the target element or attribute will be created.

Cardinality Create nil

Variables Create using default or fixed value defined by schema for target type

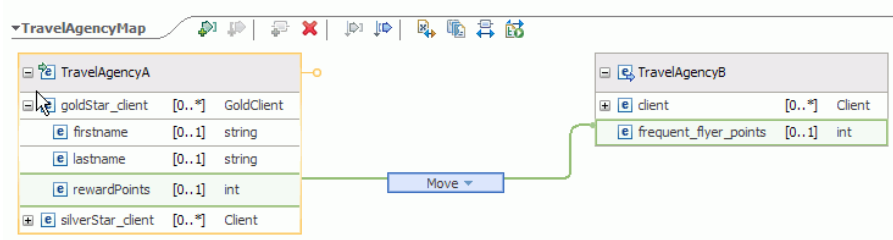
Condition Create empty (disabled for simple target not derived from built-in str)

Chapter 12. Copying a selected element of a repeating structure to a single output

In the Graphical Data Mapping editor, to copy one element from the input repeating element (array) to an output simple element, use the **Move** transform.

Procedure

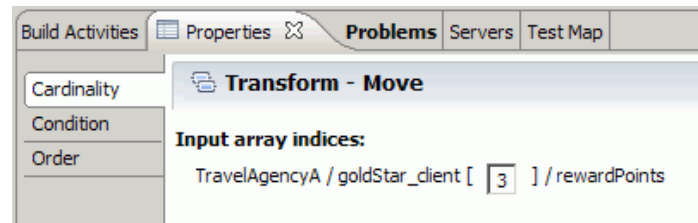
1. Define a **Move** transform between the input and the output elements.
The following figure shows the input and output elements connected by a **Move** transform:



2. Specify the index of the repeating structure that you must copy to the output.
 - a. Click the **Move** transform
 - b. Switch to the Properties view.
 - c. In the **Cardinality** properties page, enter a value in the **Input array indices** field.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.

For example, enter **3** to copy the third element in the repeating structure (array).



There is no option to set the cardinality of the output element, because it is a single element.

Chapter 13. Copying some values of a repeating element when the input and output structures are the same

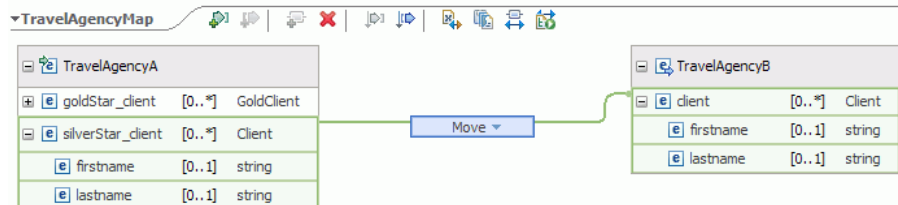
In the Graphical Data Mapping editor, to move selected elements of the input repeating element (array) to an output repeating element of the same type, select a **Move** transform between the elements, and then select the required elements in the **Cardinality** properties page.

Procedure

Complete the following steps to copy some elements of an input array to an output array when both repeating structures are identical:

1. Define a **Move** transform between the input and the output repeating structures.

The following figure shows the input and output repeating structures of type **Client** connected by a **Move** transform:

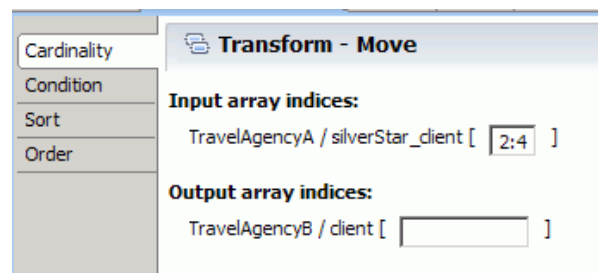


2. Specify the indexes that should be copied to the output repeating structure.
 - a. Click the **Move** transform
 - b. Switch to the Properties view.
 - c. In the **Cardinality** properties page, enter a value in the **Input array indices** field.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.

For example, to select elements 2, 3, and 4 of the array, set the cardinality to **2:4**.

The first index element has a cardinality of 1.



Chapter 14. Copying some values of a repeating element when the input and output structures are different

In the Graphical Data Mapping editor, use the **For Each** transform to iterate over an input repeating element. Then, to determine the value of the output elements, define transforms in the nested map associated with a **For Each** transform.

About this task

Use the **For Each** transform to move each element of an input array into an output array. The input and output arrays can be of the same type or of different types. For more information, see “For Each” on page 218.

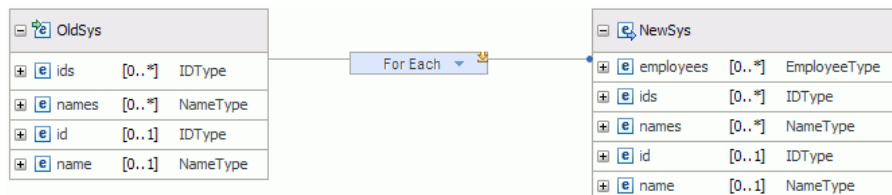
Iteration is performed over the input array.

The nested map associated to a **For Each** transform defines the type of transformations that will be performed for each iteration.

Procedure

Complete the following steps to copy some elements of an input array to an output array when the input and the output repeating elements have different types:

1. Define a **For Each** transform between the input and the output repeating structures.



2. Optional: Specify the indexes of the input repeating structure over which to iterate.

- a. Click the **For Each** transform
- b. Switch to the Properties view.
- c. In the **Cardinality** properties page, enter a value in the **Input array indices** field.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.

3. Edit the nested map by clicking the **For each** transform.
4. Define the transformations required to set the value of the output elements.
For more information, see Chapter 4, “Transforms (Mapping operations),” on page 21.

Chapter 15. Splitting an input message into multiple identical output messages

You can use a **For Each** transform or a **Join** transform wired to the head of the output message assembly to create a map that takes a single input message and produces multiple instances of an output message model. A typical use of this function is message splitting, in which an input batch message is divided into individual record messages.

About this task

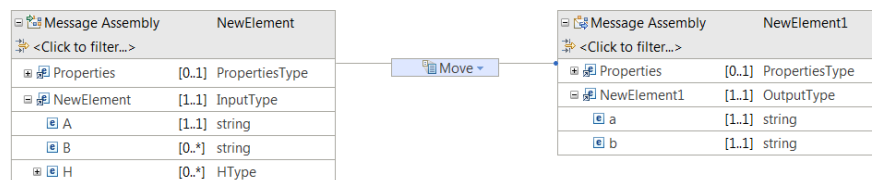
When the map runs, a new message is propagated for each iteration of the **For Each** or **Join** transform.

Procedure

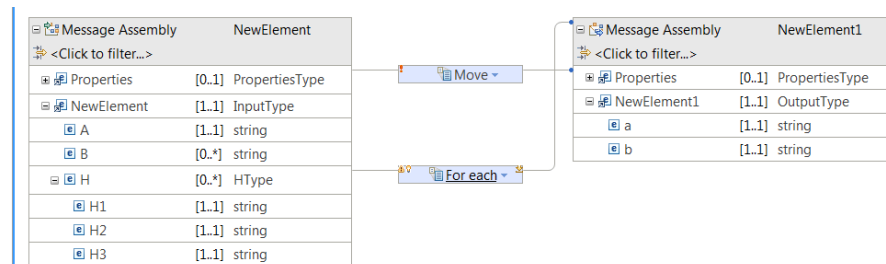
Complete the following steps to split a message into multiple output messages by using the **For Each** transform:

1. Create a map, and add one input element to your input message assembly, and one output element to your output message assembly.

The input message assembly must contain a repeating element. The values of the elements for each index are used to populate each output message instance.

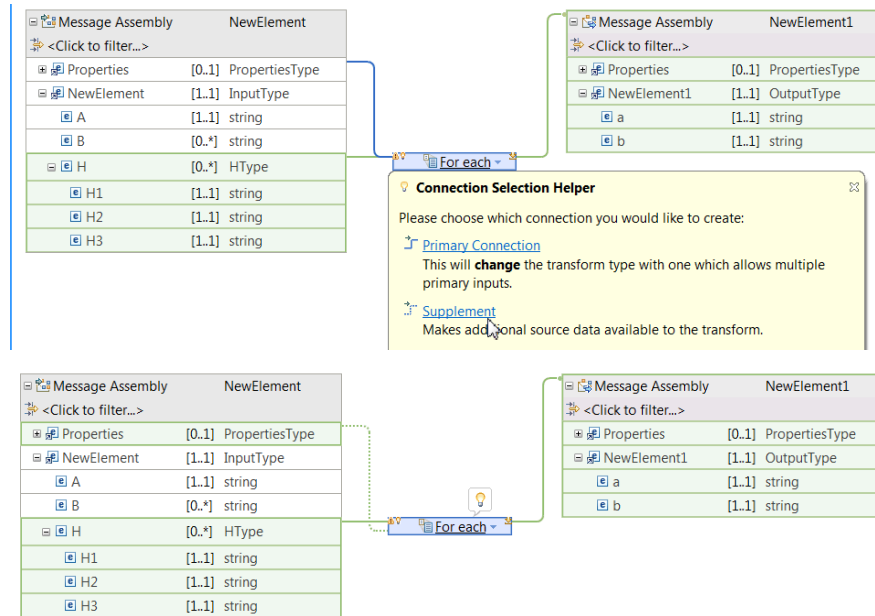


2. Define a **For Each** transform between the input element and the head of the output message assembly.

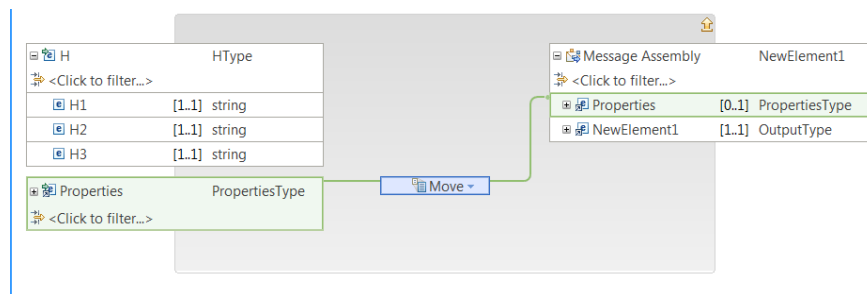


The **Move** transform between the input and the output Properties tree will display an error. Continue with the steps to remove the error.

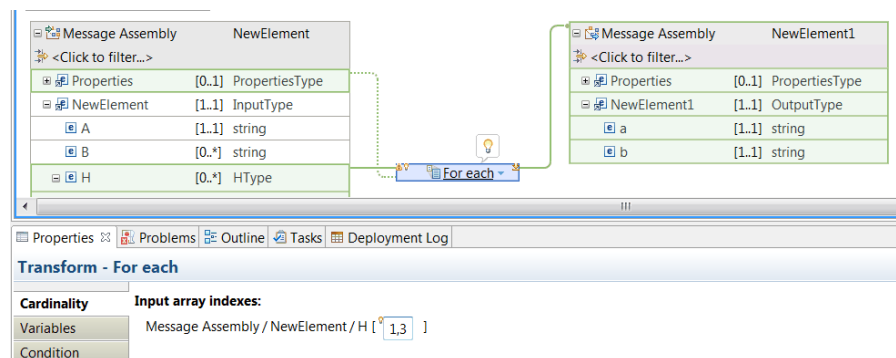
3. Delete the **Move** transform, and then connect with a supplement connection the input Properties tree to the **For Each** transform.



4. Select the **For Each** transform to open the nested map. Then, define a **Move** transform between the input and the output Properties tree.

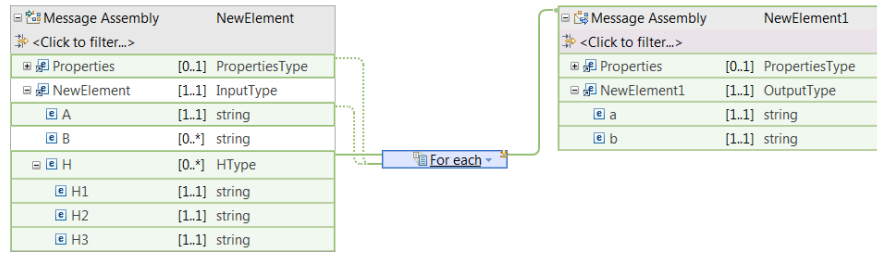


5. Optional: Return to the **For Each** transform, and configure the indexes of the **For Each** transform for which you want to generate an output message. If you need to create an output message for each index of the repeating structure, then continue with the next step. You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.



6. Optional: If you need additional information from the input message to populate output elements, define a supplement connection between each input element to the **For Each** transform.

When you connect additional input elements with supplement connections to a **For Each** transform, you can use these elements as part of your mapping operations within the nested map.



7. Select the **For Each** transform to open the nested map. Then, define the transformation logic inside the **For Each** nested map.
For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

Example

The following example shows how to define the mapping logic in a **For Each** transform after you complete steps 1 to 6:

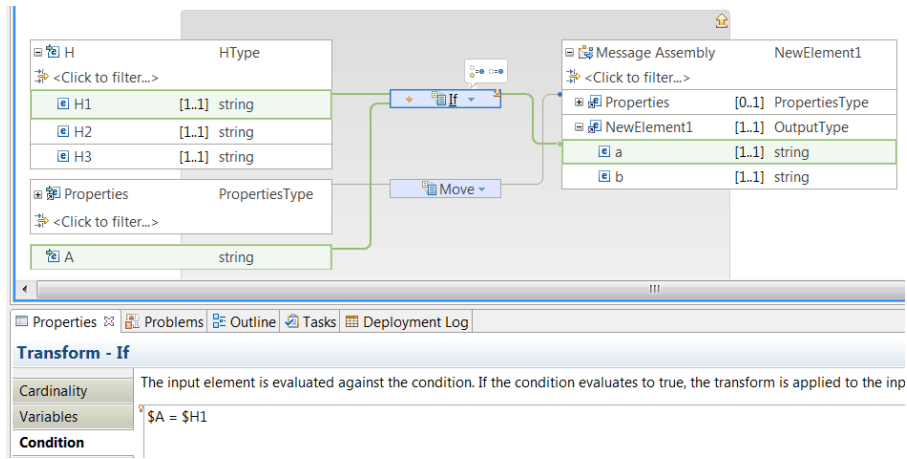
- Each instance of the **For Each** transform produces an output message.
- The repeating element **H** has three elements that are used to set the values of the output message, which only has two elements.
- An additional input element is needed as part of the transformation. This element is used as part of the conditional expression that determines when a set of transformations is applied.
- An **If** transform is used to model the conditional mapping requirements required to set the value of the output message elements for each index of the **For Each** transform.
- When the value of the input element **A** is equal to the value of the input element **H1**, then the **If** transform is applied. Otherwise, the transformation logic in an **Else** transform is applied.

Complete the following steps to define the transformation logic:

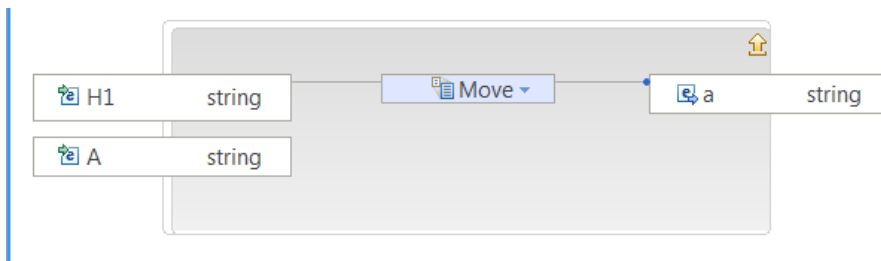
Open the **For Each** transform nested map.

Define an **If** transform between **H** and **a**.

- Define a primary connection between **H** and the **If** transform.
- Define a supplement connection between **A** and the **If** transform.
- Define a connection between the **If** transform and the output element **a**.

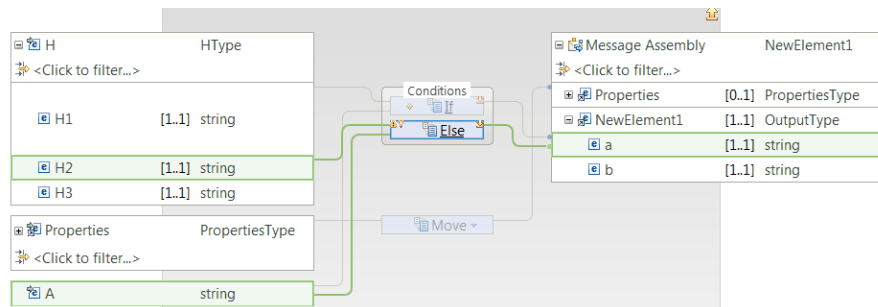


Define a **Move** transform between **H1** and **a**. This operation defines the transformation logic that the **If** transform performs.

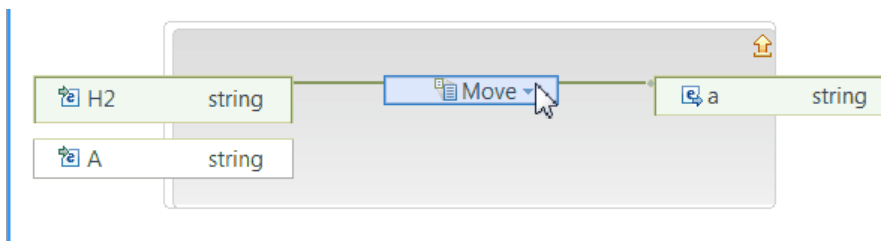


Add the **Else** transform:

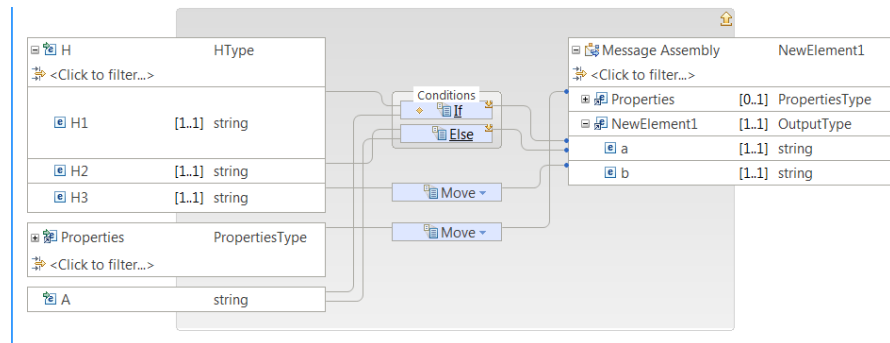
- Define a primary connection between **H** and the **Else** transform.
- Define a supplement connection between **A** and the **Else** transform.
- Define a connection between the **Else** transform and the output element **a**.



Define a **Move** transform between **H2** and **a**. This operation defines the transformation logic that the **Else** transform performs.



The following figure shows the **For Each** transform nested map after all the transformation logic has been implemented:



Chapter 16. Mapping an input message into different output messages

You can use the **If** transform to create a map that takes a single input message and produces a different output message based on the conditional expression that you define.

About this task

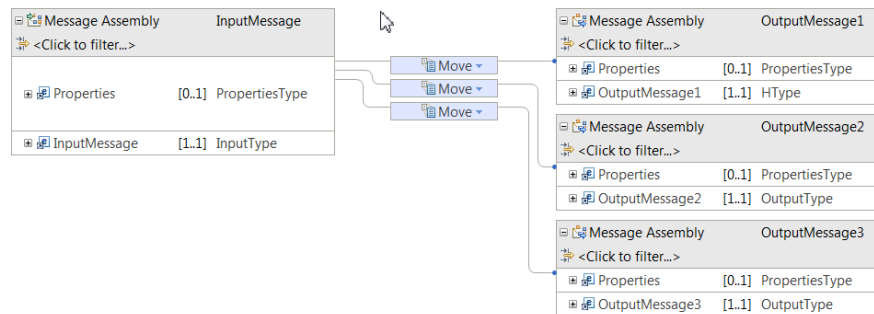
When you configure multiple output message assemblies, each output message assembly has its own properties.

You can configure each output message assembly independently of the others.

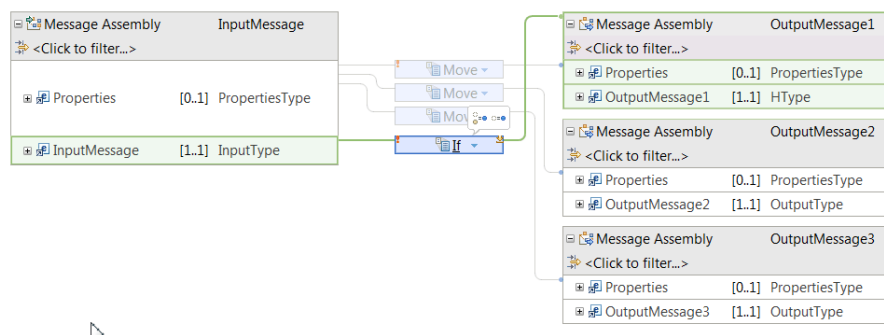
Procedure

Complete the following steps to split a message into different output messages by using the **If** transform:

1. Create a map, and add one input element to your input message assembly, and two or more output message assemblies.

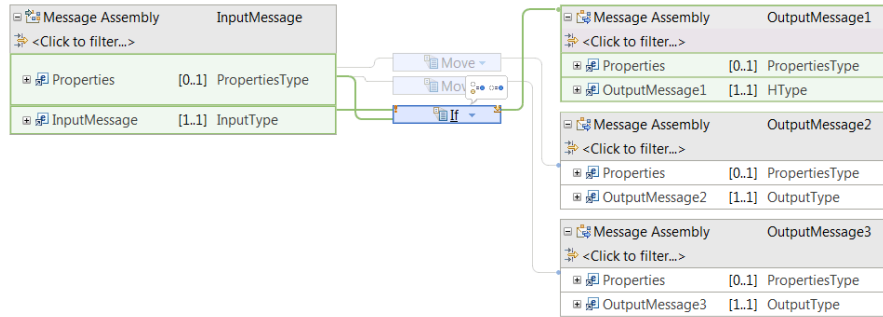


2. Define an **If** transform between the input element and one of the output message assemblies.

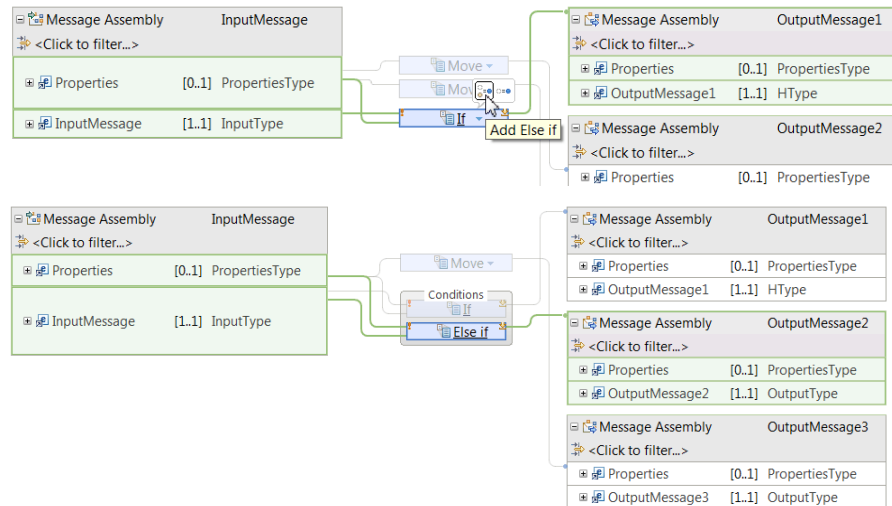


The **Move** transform between the input and the output Properties tree of the first message assembly will display an error. Continue with the steps to remove the error.

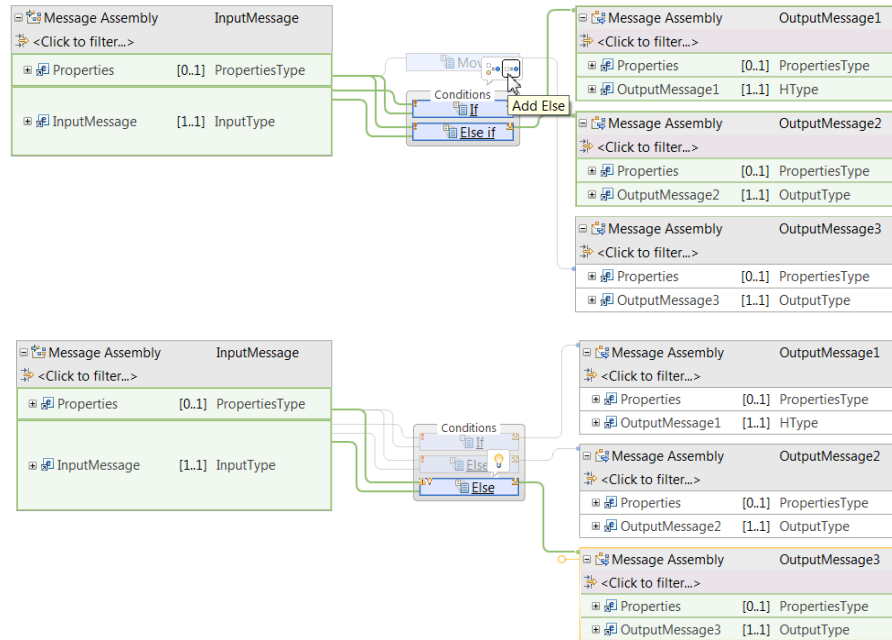
3. Delete the **Move** transform marked with an error, and then connect the input Properties tree to the **If** transform.



4. Optional: Add the **ElseIf** transform and complete the following steps to connect it to a different output message assembly:
 - a. Delete the **Move** transform that connects the Properties tree and the second output message assembly.
 - b. Define a connection between the Properties tree and the **ElseIf** transform.
 - c. Define a connection between the input message and the **ElseIf** transform.
 - d. Define a connection between the **ElseIf** transform and the second output message assembly.



5. Add the **Else** transform and complete the following steps to connect it to a different output message assembly:
 - a. Delete the **Move** transform that connects the Properties tree and the third output message assembly.
 - b. Define a connection between the Properties tree and the **Else** transform.
 - c. Define a connection between the input message and the **Else** transform.
 - d. Define a connection between the **Else** transform and the second output message assembly.



6. Define the conditional expression that determines when the **If** transform is applied and a message based on the first output message assembly is created.
 - a. Open the Properties view of the **If** transform.
 - b. Define an XPath expression in the **Condition** tab. Use content-assist. For more information, see “Defining an XPath conditional expression for a transform” on page 85.

Transform - If

Cardinality: The input element is evaluated against the condition. If the condition evaluates to true, the transform is applied to the input element.

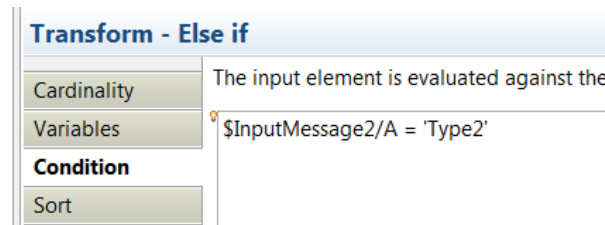
Variables: <Type an XPath expression here. Content assist available (Ctrl+space)>

Condition: `$InputMessage1/A = 'Type1'`

Sort: A : string, B : string [0..n], H : HType [0..n], H1 : string, H2 : string, H3 : string, Properties1 : PropertiesType

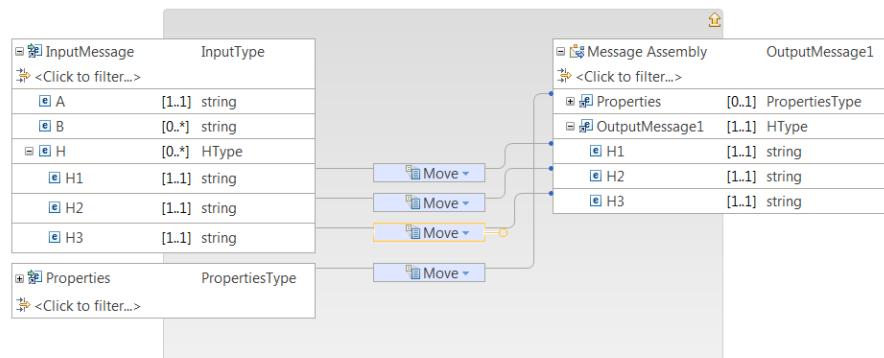
7. Define the conditional expression that determines when the **ElseIf** transform is applied and a message based on the second output message assembly is created.
 - a. Open the Properties view of the **If** transform.

- b. Define an XPath expression in the **Condition** tab. Use content-assist. For more information, see “Defining an XPath conditional expression for a transform” on page 85.



Note: When the conditional expression of the **If** transform and the **ElseIf** transform evaluate to **false**, the transformation logic defined for the **Else** transform is applied.

8. Select the **If** transform to open the associated nested map. Then, define transforms between the input and the output elements. Remember to connect the input Properties to the output properties tree with a **Move** transform.
- By default, a **Submap** transform is defined. You can choose to create a submap with your transformation logic, or delete the **Submap** transform and define locally your transformation logic.



For more information, see Chapter 26, “Transform types in the Graphical Data Mapping editor,” on page 193.

9. Select the **ElseIf** transform to open the associated nested map. Then, define transforms between the input and the output elements.
10. Select the **Else** transform to open the associated nested map. Then, define transforms between the input and the output elements.

Chapter 17. Using Java API classes for Custom Java mapping transforms

You can use the Java MbElement class for mapping inputs and outputs that are not simple types with a **Custom Java** transform.

To use the MbElement class, you must add the **plugin2.jar** file that includes the MbElement class in the **Java imports** property tab of the **Custom Java** transform. The **plugin2.jar** is available in the **classes** directory of the installation.

Alter the properties of your java project, and add the plugin2.jar to the java build path. Now you can import the MbElement class into your java source.

If your Java code is likely to be used by multiple solutions, store it in a shared library. You can store the Java code in the same shared library as a message map. Alternatively, you can store the Java code separately in a referenced shared library.

Mapping a single non-repeating element

When you map a single non-repeating element input to a single non-repeating element output, you can use a Java method with the following signature:

```
public static MbElement mbElMove(MbElement inEl) {
For example a Java method that simply copies a sub tree:
public static MbElement mbElMove(MbElement inEl) {
    MbElement outEl = null;
    try {
        outEl = inEl.copy();
        outEl.copyElementTree(inEl);
    } catch (MbException e) {
        throw (new RuntimeException(e));
    }
    return outEl;
}
```

Mapping a single repeating element

When you map a single repeating element input to an output repeating element, you can use a Java method with the following signature:

```
public static List<MbElement> customCompleTypeMove(List<MbElement> inEls)
```

For example:

```
public static List<MbElement> customCompleTypeMove(List<MbElement> inEls)
{
    List<MbElement> outEls = new ArrayList<MbElement>();
    try {
        Iterator<MbElement> i = inEls.iterator();
        while (i.hasNext()) {
            MbElement inEl = i.next();
            MbElement outEl = inEl.copy();
            // Do some processing of outEl
            outEls.add(outEl);
        }
    } catch (MbException e) {
```

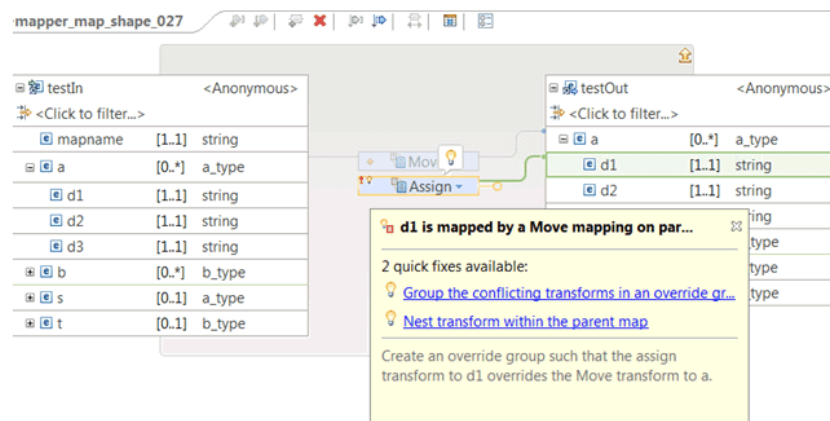
```
        throw (new RuntimeException(e));
    }
    return outEls;
}
```

Chapter 18. Applying mapping overrides

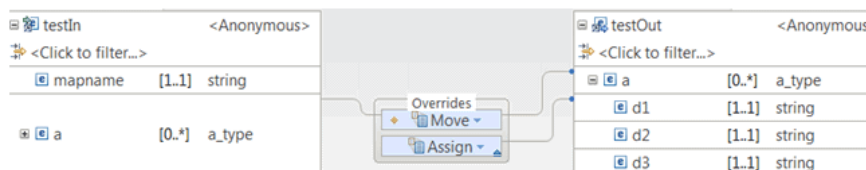
Transforms contained in a mapping overrides group are applied after a parent transform is applied. This function enables you to copy a complex type from the input to the output object, while updating some of the child elements in the complex type.

About this task

You can create a graphical data map that performs a simple copy of a complex type, by wiring a **Move** transform from the input to the output object. You can also add additional transforms that have an output element that is a simple type within the complex type. As the parent is already mapped, the Graphical Data Mapping editor provides you with the option of creating a quick fix to create an overrides group:



The transforms contained in the overrides group are applied after the parent transform is applied:



Chapter 19. Mapping database content

Use a graphical data map to map or modify database content.

About this task

You can use database content as input data for a graphical data map transform. You can use graphical data map transforms to modify database content, or call a stored procedure from a database to map the returned data. For each database transform in your graphical data map, the Graphical Data Mapping editor uses a database definition file (.dbm file) to determine the name and structure of the database that you want to access. When your graphical data map has been deployed, the IBM Integration Bus runtime component connects to the database used in each database transform by using a JDBCProvider service that you configured with the same name as the database.

The following topics guide you through the steps that are involved in mapping database content:

- “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118
- “Selecting data from a table”
- “Modifying data in a database by using mapping” on page 161
- “Calling a stored procedure from a map” on page 168
- Enabling JDBC connections to the databases

Optionally, you can use JDBC connection pooling with your JDBCProvider service to manage your database resources. For more information about JDBC connection pooling, see Using a JDBC connection pool to manage database resources used by an integration server.

To learn more about mapping database content, see the following samples:

- Graphical Data Mapping Retail
- Graphical Data Mapping Loyalty Data Warehousing

Note: You can view information about samples only when you use the information center that is integrated with the IBM Integration Studio or the online information center. You can run samples only when you use the information center that is integrated with the IBM Integration Studio.

Selecting data from a table


To map an output element from a database table, use the Graphical Data Mapping editor to retrieve the relevant rows from the database and then populate the output elements with values from the database.

Before you begin

You must complete the following tasks:

- Create a graphical data map by using the Graphical Data Mapping editor. For information, see “Creating a message map” on page 47.

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, click the **Select rows from a database** icon. . If you include a Select transform within a ForEach nested transform, the IBM Integration Bus runtime component issues one SQL select to the database for each iteration of the ForEach transform.
2. In the "New database select" wizard, select the database, table, and column from which you want to select data. To add a database definition file, or to discover a new database by connecting to a database server, click **Add database...** For more information, see "Creating a database definition (.dbm file) by using the New Database Definition File wizard" on page 118.
3. In the **SQL where clause** field, use supported SQL to specify the criteria for selecting the rows from the selected column of your database table.
Build a supported SQL statement by dragging items from the **Table columns** and **Operators** panes to the **SQL where clause** field.
To include values in your SQL statement, drag items from the **Available inputs for column values** pane to the **SQL where clause** to add them as parameters, or type literal values such as 'abc' or 123 directly in the **SQL where clause**.
Parameters from the **SQL where clause** are listed in the XPath expression table. You can edit the XPath expressions to refine the input, for example to add a specific array index for a dragged repeating field. A default **SQL where clause** is created for you, which selects all rows in your selected database table.

Note: If you edit the text of the **SQL where clause** directly, take care to:
 - ensure the case of your table and column names match that of your database.
 - avoid the use of double-quotes around table and column names.
 - only use the supported SQL keywords that are presented in the Operators pane.
4. Optional: Select **Treat warning as error**. If this option is selected, the first SQL operation that results in a warning from the selected database raises an exception.

Important: Database warnings are vendor-specific. For more information about database warnings, see the documentation for your database product.
5. Click **OK**. A **Select** transform, is created, and the data that you selected is displayed in the graphical data map.
6. Connect the **Select** transform to the required output object in the map. The **ResultSet** input to the **Select** transform is a repeating structure that contains one instance for each row that is selected by your configured **SQL where clause**.
7. Click the **Select** transform to further define the transform. A nested map is created, in which you can select the specific transforms that are required for the input and output elements.

What to do next

- If you want exceptions that are returned from the database server when the SQL operation is run to be handled by the map, instead of having such exceptions stop the map and being reported, you can add a Failure transform into the transform group; see "Handling database exceptions in a graphical data map" on page 171.
- Set up a JDBC connection to the database that you want to access; see [../com.ibm.etools.mft.doc/ah61300_.dita](#).

Modifying data in a database by using mapping

Use the Graphical Data Mapping editor to insert, update, or delete rows of data in a database table.

About this task

You can use database transforms in your graphical data maps to insert new rows of data, or to update or delete existing rows of data, in your database tables. For each database transform in your graphical data map, the Graphical Data Mapping editor uses a database definition file (.dbm file) to determine the name and structure of the database that you want to access. You can start the wizard to create a database definition file when you create a database transform in a graphical data map.

If you connect elements from an input object to database columns within a database transform in your graphical data map, every input message that is processed by your map at run time must include those elements. If a message is either missing an element that is connected to a database column, or does not provide a valid value for that database column, an exception is raised when the message is processed by the map. Input elements that you connect to nullable database columns must provide either a valid value, or the NULL value. For more information about null values, see XMLNSC empty elements and null values.

When you add a database Insert, Update or Delete transform to a graphical data map, the transform is displayed as an additional output target to which you can connect input objects. When your map is run, a database transform calls a single operation on the configured database server. If you connect a repeating input element to the database transform, the Graphical Data Mapping editor moves the database transform inside a nested "For Each" transform from the repeating input.

A database Insert, Update, or Delete transform are created as a transform group, comprising the database operation and a Return transform. The database operation transform for Insert and Update are nested transforms in which the individual mapping to the database table columns are made. The Return transform is an optional transform that allows a nested mapping to be entered if the database operation is successful. If you do not want to use the Return transform, you can delete it from the transform group. If you must provide some mapping for when a failure is returned from the database operation, you can add a Failure transform into the transform group. The Failure transform provides a nested transform that is entered if the database system returns a failure.

If the insert, update, or delete is conditional on a test result, you can change the Insert, Update, or Delete transform to an If transform. Before you change the transform, ensure that the Insert, Update, or Delete transform is not part of a transform group. Remove any Return or Failure transform then select an If transform in place of the Insert Update or Delete. The Insert, Update, or Delete transform is moved into the nested mapping of the If transform. You can then add any required Return and Failure transforms.

The following topics describe how to modify data in a database table:

- "Inserting data into a table" on page 162
- "Updating data in a table" on page 163
- "Deleting data from a table" on page 165

Inserting data into a table

Use the Graphical Data Mapping editor to insert data into a database table.

Before you begin


You must complete the following task:

- Create a graphical data map by using the Graphical Data Mapping editor. For information, see “Creating a message map” on page 47.

About this task

To insert a row, or multiple rows, into a database table by using the Graphical Data Mapping editor, complete the following steps:

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click the canvas, and select **Database > Insert into table**. Alternatively, select a schema element as an input object, and then click the **Insert a row into a database table** icon.  The Insert wizard is displayed.
2. In the **Database** field, select the database that you want to modify. To add a database definition file, or to discover a new database by connecting to a database server, click **Add database...** For more information, see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118. To use a different database name at run time, you can override this value by setting the **databaseName** property of the JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.
3. In the **Schema** field, select the database schema that you want to use to build the transform. To use a different database schema at run time, you can override this value by setting the **databaseSchemaNames** property of the JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.
4. In the **Table** field, select the table that you want to modify.
5. Optional: Select **Treat warning as error**. If this option is selected, the first SQL operation that results in a warning from the selected database raises an exception.

Important: Database warnings are vendor-specific. For more information about database warnings, see the documentation for your database product.

6. Click **OK**. An **Insert** transform and a **Return** transform are created as a transform group, and are displayed in your graphical data map. The **Return** transform is an optional transform type. If you do not need to use the **Return** transform, you can delete it from your graphical data map.
7. Optional: To replace a **Return** transform that you deleted from your graphical data map, right-click your **Insert** transform and select **Database > Utilize return**.
8. In the Graphical Data Mapping editor, connect input objects to the **Insert** transform to define the content of your inserted row.
 - Connect a non-repeating element to the **Insert** transform to insert a single row into the selected database table.
 - Connect one or more repeating elements to the **Insert** transform to insert multiple rows into the selected database table. To connect multiple

repeating elements, select your repeating elements, then right-click the **Insert** transform and select **Create Connection**.

If you connect a single repeating element, the **Insert** transform is nested inside a **For Each** transform. If you connect multiple repeating elements, the **Insert** transform is nested inside a **Join** transform. In either case, the nested transform opens so you can continue to edit your **Insert** transform.

9. Click the **Insert** transform to create connections to the columns in your inserted row, and to further define the transform.
10. Optional: If you need to provide handling for the connected source element being Missing, Empty or Nil, you can set a Database Policy. See “Behavior when modifying database column values from optional source elements” on page 172.
11. Optional: Connect the **Return** transform to implement a nested mapping that is called if the **Insert** operation was completed successfully.
12. Optional: Click the **Return** transform to further define the transform. A nested map is created, in which you can select the specific transforms that are required for the input and output elements.

What to do next

- If you want exceptions that are returned from the database server when the SQL operation is run to be handled by the map, instead of having such exceptions stop the map and being reported, you can add a Failure transform into the transform group; see “Handling database exceptions in a graphical data map” on page 171.
- Set up a JDBC connection to the database that you want to access; see `../com.ibm.etools.mft.doc/ah61300_.dita`.

Updating data in a table

Use the Graphical Data Mapping editor to update data in a database table.

Before you begin

You must complete the following task:

- Create a graphical data map by using the Graphical Data Mapping editor. For information, see “Graphical Data Mapping editor” on page 5.

About this task

To update a row of data, or multiple rows of data, in a database table by using the Graphical Data Mapping editor, complete the following steps:

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click the canvas, and select **Database > Update Table**.

Alternatively, click the **Update a row in a database table** icon.  The New Database Table Update wizard is displayed.

2. In the **Database** field, select the database that you want to modify. To add a database definition file, or to discover a new database by connecting to a database server, click **Add database...** For more information, see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118. To use a different database name at run time, you can override this value by setting the **databaseName** property of the

JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.

3. In the **Schema** field, select the database schema that you want to use to build the transform. To use a different database schema at run time, you can override this value by setting the **databaseSchemaNames** property of the JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.
4. In the **Table** field, select the database table that you want to modify.
5. Optional: Select **Treat warning as error**. If this option is selected, the first SQL operation that results in a warning from the selected database raises an exception.

Important: Database warnings are vendor-specific. For more information about database warnings, see the documentation for your database product.

6. In the **SQL where clause** field, use supported SQL to specify the criteria for selecting rows from your database table.

Build a supported SQL statement by dragging items from the **Table columns** and **Operators** panes to the **SQL where clause** field.

To include values in your SQL statement, drag items from the **Available inputs for column values** pane to the **SQL where clause** to add them as parameters, or type literal values such as 'abc' or 123 directly in the **SQL where clause**.

Parameters from the **SQL where clause** are listed in the XPath expression table. You can edit the XPath expressions to refine the input, for example to add a specific array index for a dragged repeating field. A default **SQL where clause** is created for you, which selects all rows in your selected database table.

Note: If you edit the text of the **SQL where clause** directly, take care to:

- ensure the case of your table and column names match that of your database.
 - avoid the use of double-quotes around table and column names.
 - only use the supported SQL keywords that are presented in the Operators pane.
7. Optional: Select **Insert when a row does not exist** if you want to insert a new row in your database table when no existing row meets the criteria of your **SQL where clause**. If this option is selected, the map checks the "number of rows updated" return from the database server for the Update SQL operation. If the "number of rows updated" is zero, the map issues an insert SQL operation. For the insert operation to succeed, your **Update** transform must explicitly provide valid values for all mandatory database columns. If you want a row that is inserted in this way to use different values to those that are provided by your **Update** transform, consider adding a conditional **Insert** transform inside the **Return** transform.
 8. Click **OK**. An **Update** transform and a **Return** transform are created as a transform group, and are displayed in your graphical data map. The **Return** transform is an optional transform type that provides a nested mapping that is entered only if the associated **Update** was successful. If you do not need to use the **Return** you can delete it from your graphical data map.
 9. Optional: To replace a **Return** transform that you deleted from your graphical data map, right-click your **Update** transform and select **Database > Utilize return**.

10. In the Graphical Data Mapping editor, connect input objects to the **Update** transform to define the content of your updated row. If you connect a repeating element, the **Update** transform is nested inside a **For Each** transform, and this nested transform is opened so that you can continue to edit your **Update** transform.
11. Click the **Update** transform to create connections to the columns in your updated row, and to further define the transform.
12. Optional: If you need to provide handling for the connected source element being Missing, Empty or Nil, you can set a Database Policy. See “Behavior when modifying database column values from optional source elements” on page 172.
13. Optional: Connect the **Return** transform to implement a nested mapping that is called if the **Update** operation was completed successfully. The nested **Return** transform provides a built-in input, "NumberOfRowsUpdated", and additional inputs can be connected.
14. Optional: Click the **Return** transform to further define the transform. A nested map is created, in which you can select the specific transforms that are required for the input and output elements.

What to do next

- If you want exceptions that are returned from the database server when the SQL operation is run to be handled by the map, instead of having such exceptions stop the map and being reported, you can add a Failure transform into the transform group; see “Handling database exceptions in a graphical data map” on page 171.
- Set up a JDBC connection to the database that you want to access; see [../com.ibm.etools.mft.doc/ah61300_.dita](http://com.ibm.etools.mft.doc/ah61300_.dita).

Deleting data from a table

Use the Graphical Data Mapping editor to delete data from a database table.

Before you begin

You must complete the following task:


- Create a graphical data map by using the Graphical Data Mapping editor. For information, see “Creating a message map” on page 47.

About this task

To delete a row of data, or multiple rows of data, from a database table by using the Graphical Data Mapping editor, complete the following steps:

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click the canvas, and select **Database > Delete from Table**.

Alternatively, click the **Delete a row from a database table** icon.  The New Database Table Delete From wizard is displayed.

2. In the **Database** field, select the database that you want to modify. To add a database definition file, or to discover a new database by connecting to a database server, click **Add database...** For more information, see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118. To use a different database name at run time, you can override this value by setting the **databaseName** property of the

JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.

3. In the **Schema** field, select the database schema that you want to use to build the transform. To use a different database schema at run time, you can override this value by setting the **databaseSchemaNames** property of the JDBCProvider configurable service that connects to your database; see Setting up a JDBC provider for type 4 connections.
4. In the **Table** field, select the database table that you want to modify.
5. Optional: Select **Treat warning as error**. If this option is selected, the first SQL operation that results in a warning from the selected database raises an exception.

Important: Database warnings are vendor-specific. For more information about database warnings, see the documentation for your database product.

6. In the **SQL where clause** field, use supported SQL to specify the criteria for selecting the rows that you want to delete from your database table.

Build a supported SQL statement by dragging items from the **Table columns** and **Operators** panes to the **SQL where clause** field.

To include values in your SQL statement, drag items from the **Available inputs for column values** pane to the **SQL where clause** to add them as parameters, or type literal values such as 'abc' or 123 directly in the **SQL where clause**.

Parameters from the **SQL where clause** are listed in the XPath expression table. You can edit the XPath expressions to refine the input, for example to add a specific array index for a dragged repeating field. A default **SQL where clause** is created for you, which selects all rows in your selected database table.

Note: If you edit the text of the **SQL where clause** directly, take care to:

- ensure the case of your table and column names match that of your database.
 - avoid the use of double-quotes around table and column names.
 - only use the supported SQL keywords that are presented in the Operators pane.
7. Click **OK**. A **Delete** transform and a **Return** transform are created as a transform group, and are displayed in your graphical data map. The **Return** transform is an optional transform that provides a nested mapping. It is entered only if the associated **Delete** was successful. If you do not need to use the **Return** transform, you can delete it from your graphical data map.
 8. Optional: To replace a **Return** transform that you deleted from your graphical data map, right-click your **Insert** transform and select **Database > Utilize return**.
 9. Optional: Connect the **Return** transform to implement a nested mapping that is called if the **Delete** operation was completed successfully.
 10. Optional: Click the **Return** transform to further define the transform. A nested map is created, in which you can select the specific transforms that are required for the input and output elements.

What to do next

- If you want exceptions that are returned from the database server when the SQL operation is run to be handled by the map, instead of having such exceptions

stop the map and being reported, you can add a Failure transform into the transform group; see “Handling database exceptions in a graphical data map” on page 171.

- Set up a JDBC connection to the database that you want to access; see `../com.ibm.etools.mft.doc/ah61300_.dita`.

Data type considerations for mapping database content

Data type handling using the Graphical Data Mapping editor to read or modify data in a database table requires consideration of the type of Database server that will be connected to from the runtime. The map may require to make explicit type casts, in order to avoid mapping node exceptions or database server exceptions being thrown.

The data types of the database columns, shown at map design time in the Graphical Data Mapping editor, are provided by the database definition file. You can use Cast transform or custom transforms, such as XPath, to ensure that data from elements mapped to the database columns are of the correct type.

When the map is executed in the IBM Integration Bus runtime, the JDBC Providers configurable service determines the database to connect to. This must be defined in the runtime. See `../com.ibm.etools.mft.doc/ah61300_.dita`.

The broker runtime attempts to query the connected database system, in order to obtain the data type of the target columns. This is so that required type casts take place before passing data in SQL statements. If there is no valid typecast between the type of the presented value and the type defined by the Database metadata in the IBM Integration Bus runtime, an IBM Integration Bus runtime exception is thrown by the Mapping node that is executing the map.

Note: Not all database servers supported by IBM Integration Bus provide querying of table meta data in a way IBM Integration Bus can currently process. IBM Integration Bus cannot currently obtain table metadata when connected to the following database server types:

- Microsoft_SQL_Server
- Oracle
- Sybase_JConnect6_05
- solidDB®

When using these types of database systems, the IBM Integration Bus runtime cannot perform casting. The data element values are passed to the database server in the type they are presented, without any casting being performed. This can result in the Database System rejecting the value and throwing a Database Exception. This is in contrast to an IBM Integration Bus runtime exception, where it is thrown as a Mapping node exception.

The resulting types are determined depending on how the input element is wired to the database transform:

- Column values set via Move transforms from an message tree element are passed as its given type when it is a base SQL type. For example: Integer, otherwise as character string formatted as per the IBM Integration Bus `getValueAsString() MbElement` method.
- Column values set via custom XPath, Java or ESQL functions are passed as the type returned by the function.
- Column values set via **Assign** transform will always be passed as character string. If you require a specific type to be assigned, you must use a **Cast**

transform of the appropriate `xs` type constructor. For example, to assign the value 1 to an Integer type column, use the `xs:int()` **Cast** transform and set a value of '1' instead of an **Assign** transform.

When using values in **Where** clauses for **Select**, **Update** and **Delete**, the types are determined by:

- Literal values are typed according to standard SQL syntax, such as quote character strings, unquoted numbers and so on.
- Values set via XPath path reference to a message tree element are passed as its given type when it is a base SQL type. For example: Integer, otherwise as character string formatted as per the IBM Integration Bus `getValueAsString()` `MbElement` method.

Calling a stored procedure from a map

Use the Graphical Data Mapping editor to call a stored procedure by using the Database Routine transform.

Before you begin

Before you start:

You must complete the following task:

- Create a graphical data map by using the Graphical Data Mapping editor. For information, see “Creating a message map” on page 47.

About this task

You can use the Database Routine transform to call a stored procedure from a database.

Note: Only IBM DB2 stored procedures are supported in IBM Integration Bus v9.0.


The Database Routine transform acts as a nested mapping, into which you can wire inputs to construct mappings to set the input parameter values for the stored procedure call. The routine input parameters are displayed as outputs of the Database Routine nested mapping. You cannot wire any output from the Database Routine transform.

Use the Return transform within the Transform group of the Database Routine to map any output parameters, return values, or **ResultSets** produced by calling the Database Routine. You can wire any additional inputs that you want to map when the Database Routine call completes successfully. The output parameters, any return value, and any returned Result sets you defined for the Database Routine are provided as inputs in the nested Return mapping. You can wire the Return to a single or multiple sibling output elements to enable the returned data to be mapped to the map output.

Stored procedure parameters of type **Array** are not supported in database calls from a graphical data map.

Procedure

Using the Graphical Data Mapping editor, complete the following steps:

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click the canvas, and select **Database > Call Database Routine**.
 - Alternatively, select a schema element (or elements) as an input parameter value. Optionally, select a schema element (or elements) as an output value.
 - You can also use drag-and-drop to create the Database Routine transform. Connect the input object to the output object, and a transform is automatically created. Select the transform, and choose **Database Routine** from the **Transforms** list.
 - You can also click the **Call a stored procedure in a database** icon. 

The Database Routine wizard is displayed.

2. In the **Database** field, select the database that you want to call the routine from. To add a database definition file, or to discover a new database by connecting the IBM Integration Studio using JDBC to a database server, click **Add database....** For more information, see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118. You must select the **Routines** option during the discovery process. To use a different database name at run time than the name used in the IBM Integration Studio, you can override this value by setting the **databaseName** property of the JDBCProvider configurable service that defines how to connect to your database; see Setting up a JDBC provider for type 4 connections.
3. In the **Schema** field, select the database schema that defines the stored procedure that you want to call from the map. To call the stored procedure from a different database schema at run time, you can override this value by setting the **databaseSchemaNames** property of the JDBCProvider configurable service that defines how to connect to your database; see Setting up a JDBC provider for type 4 connections.
4. In the **Routine** field, select the stored procedure that you want to call into the Database Routine transform in the map.
 - a. Optional: If the selected routine can return a value, a **Return value** check box is displayed. If you want to make the return value available for mapping in the Return transform, check this box.

Note: The selected **Routine** then populates the following locked fields:

- **Type:** states whether the type of the selected **Routine** is a stored procedure .
- **Parameters:** details of the parameter names, mode, and type for the selected **Routine**.
- **Max ResultSets:** if the Database definition file for the **Routine** provides this information, states the maximum Result sets. Otherwise, it is blank.

These fields can be displayed by clicking **Display Routine parameter details...**

5. Optional: Select **Treat warning as error**. If this option is selected, and calling the Database Routine in the configured runtime database returns an SQL warning, it is handled as if the database is raising an exception. If the Failure transform is present, then it enters its nested mapping. Otherwise, the map execution stops, and an exception is raised from the Mapping node that is running the map.

Important: Database warnings are vendor-specific. For more information about database warnings, see the documentation for your database product.

6. Optional: If the selected **Routine** can return Result sets, and you want to map values from them, you must define their order and column contents.

- a. If your Database definition file defined the number of Result sets, then **Result sets list** is pre-populated with one **ResultSet**, and displays the maximum number that can be returned. Use the **Add** and **Delete** buttons to populate the **Result sets list**, up to any maximum number defined in the database definition file. You must order the Result sets as they are defined in the Database Routine code. If you only want to map data from, for example, the second result set, you must still include the first result set in the table because they are accessed by their positional order.
 - b. Select each Result set, and use the check boxes in **Available table columns** to add column definitions to the Result set to match what the Database Routine returns. You only need to define the Result set columns that you want to be available for mapping.
7. Click **OK**. The Database Routine, and its grouped Return transform are displayed in your graphical data map. If you made any selections in the mapping input/output, all selected inputs are wired into the Database Routine, and outputs are wired to the Return transform. If you made no selections, then the new transform appears in the map unconnected.
 8. Provide any required values for **IN** and **INOUT** mode parameters for the Database Routine. The input parameters for the selected stored procedure are displayed as outputs in the nested Database Routine transform.
 - a. Connect the required input elements to the Database Routine, and within the nested map provide transforms to set a value for each parameter. The Database Routine is entered only once, making one call to the database system. You must set cardinality for any repeating elements that are connected into the Database Routine transform, or use a function transform to provide a single value to the parameter.

Note: If any parameter is not given a value, the database server might return an exception if it cannot provide a default value. If the resulting output value of the transforms setting the parameters is not the correct type for the Database Routine, or the content is invalid, for example, exceeding a maximum length, a database exception might occur.
 9. Provide any required mapping for the output elements from data that is returned from the Database Routine. This data can include **OUT** and **INOUT** mode parameters, optional **Routine** return values, and one or more Result sets. The **Routine** outputs are displayed as inputs in the nested Return transform.
 - a. Optional: Connect any additional input elements that you might require merged with the Database Routine data to the Return transform. Connect the Return transform to one or more output elements of the map. Provide transforms within the nested Return map to set the connected outputs from the provided Database Routine output values.

What to do next

Next:

- If you want exceptions that are returned from the database server when the Database Routine is called to be handled by the map, instead of having such exceptions stop the map and being reported, you can add a Failure transform into the transform group; see “Handling database exceptions in a graphical data map” on page 171.
- Set up a JDBC connection to the database that you want the run time to call the Database Routine into; see ../com.ibm.etools.mft.doc/ah61300_.dita.

- If you want to modify the available columns in the Result sets for mapping, use the Properties view of the Database Routine transform, and then update the Return nested mapping.

Handling database exceptions in a graphical data map

Add a Failure transform to your graphical data map to handle exceptions that might be raised as a result of a database transform.

About this task

If you want the map to handle exceptions that are returned from the database server when the SQL operation is run, instead of such exceptions stopping the map and being reported, you can add a Failure transform to the transform group. The Failure transform is an optional transform in each of the database transform groups, and can be added or removed as required. If an exception is raised by the configured database server, and you have not configured a corresponding Failure transform, the map operation is stopped.

To add a Failure transform to a graphical data map by using the Graphical Data Mapping editor, complete the following steps:

Procedure

1. With a graphical data map (.map) file open in the Graphical Data Mapping editor, right-click a **Select**, **Insert**, **Update**, **Delete**, or **Database Routine** transform, and then select **Database > Handle Failure**. A **Failure** transform is created, and is displayed in your graphical data map.
2. Connect the **Failure** transform to specify how any exceptions from the database transform are processed when the map is run. If the Failure transform is present in the graphical data map, and is connected to one or more output objects, then the exception is caught and processed by the Failure transform. Database transforms additionally have a **Treat warnings as exceptions** option.

•

Important: If the Failure transform is present in the graphical data map, but is not connected, then the exception is caught by the Failure transform, and is ignored.

- If the **Failure** transform has been deleted from the graphical data map, then the exception is handled by the Mapping node in your message flow, and is handled in the same way as other message flow exceptions.
3. Click the **Failure** transform to open the nested map and further define the transform.

Results

You have added and configured a **Failure** transform into your graphical data map. If you want the failure to cause the execution of the map to stop when the database transform receives an SQL exception, remove the **Failure** from the transform.

What to do next

Before you deploy a graphical data map that contains database transforms, you must complete the following task:

- Set up a JDBC connection to the database that you want to access. For more information, see ../com.ibm.etools.mft.doc/ah61300_.dita.

Behavior when modifying database column values from optional source elements

When updating or inserting database columns, you can define different behaviors for a missing, empty, or nil source.

Behavior with no Database Policy

When Inserting or Updating data into a database column by connecting a transform which is defined as optional in the schema model, you might want to consider the behavior for the possible source input states: Missing, Empty, or Nil. The behavior can be default or customized by enabling a Database Policy.

Table 1 defines the source states and the behavior without a Database Policy enabled.

Tables 2 and 3 define the behavior of enabling a Database Policy to check the source state and to then take a specific configured action.

Table 11. Behavior with no Database Policy on transforms linked to a column in an Insert or Update operation.

Source state	Definition	Behavior
Missing Source	The input document does not contain the source element.	The column will not be passed in the SQL statement sent to the database server. The outcome is determined by the definition of the target column in the database: <ul style="list-style-type: none"> • If the column is defined with a default value, this value is set by the database system. • If the column is defined as nullable, and no default is defined, the column is set to null by the database system. • If the column is defined as not nullable, and no default is defined, the database will return a SQL exception.
Empty Source	The input document contains the source element, but that source is empty.	IBM Integration Bus passes the value returned by "getValue" for the source element as the parameter value for the column in the SQL statement sent to the database. For example, an element of the String type will return the empty String value, so the target database column would be set with an empty string, "".
Nil Source	The input document contains the expected source, and it is nil.	The value returned by "getValue" is set to NULL.

Behavior for Insert with an enabled Database Policy

When Inserting data into a database column, you can enable a database policy on each transform mapping a single value from a source element. This allows you to choose one of the following actions for each of the input source states: Missing, Empty, or Nil.

Table 12. Behavior with a Database Policy enabled on transforms linked to a column in an Insert operation.

Actions for source state	Behavior
Exclude column from database operation	Insert the database default value for the column. The column is excluded from the SQL statement sent to the database. This option is only enabled if the target database column has a default value defined in the database model from the associated .dbm file.
Insert the empty String value ""	This option is only enabled if the target database column is defined as any character string type in the database model from the associated .dbm file.
Set to NULL	This option is only enabled if the target database column is defined as nullable in the database model from the associated .dbm file.
Throw a map error	Produces a map error: <ul style="list-style-type: none">• Missing: BIP3970• Empty: BIP3971• Nil: BIP3972 For more information, see: BIP3000-3999: Built-in nodes.

Behavior for Update with an enabled Database Policy

When Updating data in a database column, you can enable a database policy on each transform mapping a single value from a source element. This allows you to choose one of the following actions for each of the input source states: Missing, Empty, or Nil.

Table 13. Behavior with a Database Policy enabled on transforms linked to a column in an Update operation.

Actions for source state	Behavior
Exclude column from database operation	The column is excluded from the SQL statement sent to the database. The value of the column currently in the database is not changed.
Set to the empty String value ""	This option is only enabled if the target database column is defined as a character string type in the database model from the associated .dbm file.
Set to NULL	This option is only enabled if the target database column is defined as nullable in the database model from the associated .dbm file.

Table 13. Behavior with a Database Policy enabled on transforms linked to a column in an Update operation (continued).

Actions for source state	Behavior
Throw a map error	Produces a map error: <ul style="list-style-type: none">• Missing: BIP3970• Empty: BIP3971• Nil: BIP3972 For more information, see: BIP3000-3999: Built-in nodes.

Chapter 20. Referencing message maps in your solution

You can reference a message map and a submap during the development phase. You can also reference a message map dynamically at runtime.

About this task

Message maps and submaps are resources that can be used more than once in your solutions. For example, a message map can be used by one or more Mapping nodes in a message flow.

Procedure

Choose any of the following options to reference a message map or a submap in your solution:

- Reference a message map: You can reference a message map by configuring the **Mapping routine** property in a Mapping node. For more information, see “Referencing an existing message map from a Mapping node.”
- Dynamically reference a message map at runtime: You can define a message map dynamically at runtime by setting the local environment **MappingRoutine** element in your message flow before the message reaches the Mapping node where the message map needs to be used. For more information, see “Dynamically selecting a message map” on page 176.
- Reference a submap: You can call a submap from a graphical data map in the Graphical Data Mapping editor by using the **Submap** transform. For more information, see “Calling a submap” on page 177.

Referencing an existing message map from a Mapping node

A message map can be used by one or more Mapping nodes in a message flow. You can reference a message map by configuring the **Mapping routine** in a Mapping node.

About this task

The Mapping node invokes a map-based transform.

The input to the Mapping node is the input message assembly that is propagated from the upstream node.

The output of the Mapping node is the new message assembly that is created by the mapping operations and propagated from the output terminal of the Mapping node.

Procedure

To reference an existing message map from a Mapping node, complete the following steps:

1. In the Application Development view, double-click the message flow that contains the Mapping node that you want to modify.
The message flow opens in the Message Flow editor.

2. In the Message Flow editor, click the Mapping node that you want to modify.
The Mapping node properties are displayed in the Properties view.
3. In the Properties view, select the **Basic** tab.
4. Next to the **Mapping routine** field, click **Browse**. The Data Transformation Map Selection window opens.
5. From the list in the Data Transformation Map Selection window, select the message map that you want to reference from your selected Mapping node, and click **OK**.
Message maps are listed in the format `{BrokerSchemaName}:MapName`.
`{default}` indicates that no broker schema is used by the message map.
6. Save and close your message flow.

Results

Your Mapping node references the message map that you selected.

Dynamically selecting a message map

To dynamically assign a message map to a Mapping node at runtime, you must pass the new map name in the local environment tree. You must define the new map name in the **MappingRoutine** element. The value you set in the **MappingRoutine** element overrides the map name that is set in the **Mapping routine** property of the Mapping node.

About this task

You can create, deploy, and run a message flow that invokes a different message map at a Mapping node.

You can override the mapping routine that is used to transform a message instance by specifying a new mapping routine in the local environment **MappingRoutine** element. You must specify the new mapping routine in the local environment tree that is upstream of the Mapping node that you need to modify.

The mapping routine qualified name that is provided in the **MappingRoutine** element must be defined in a map file that has to be deployed to the integration node in a BAR file where the message flow is deployed.

You can use any of the following message flow nodes to set the value of the local environment **MappingRoutine** element:

- Mapping node
- Compute node
- JavaCompute node

Procedure

To override at runtime the message map configured during development in a Mapping node, you must complete the following steps:

1. Required: Reference a message map in the Mapping node. This is the default message map executed by the Mapping node. For more information, see “Referencing an existing message map from a Mapping node” on page 175.
You must configure the name of the mapping routine that contains the statements to execute against the database or the message tree in the Mapping

node property **Mapping Routine**. By default, the name that is assigned to the mapping routine is:

{default_broker_schema}:DefaultMsgFlowName_MappingNodeName, where *default_broker_schema* is the broker schema where the message flow is located, and *DefaultMsgFlowName_MappingNodeName* is the name of the message flow concatenated with the name of the Mapping node.

2. In your message flow, add a new Mapping node located before the Mapping node where you want to assign dynamically a message map. Then complete the following configuration steps in the new Mapping node:
 - a. Add the local environment tree to the input message assembly.
 - b. Add the local environment tree to the output message assembly.
 - c. Optional: Add database tables if you require information available in an external database.
 - d. Configure a **Move** transform between the input local environment tree and the output local environment tree.
 - e. Configure a transform to set the output local environment tree **MappingRoutine** element. You can also add a condition to the transform.

You must define the value for **LocalEnvironment > Mapping > MappingRoutine**.

Message Assembly	SOAP_Domain_Msg
<Click to filter...>	
LocalEnvironment	[0..1] _LocalEnvironmentType
Destination	[0..1] _LocalEnvironmentDestinationType
WrittenDestination	[0..1] _LocalEnvironmentWrittenDestinationType
Aggregation	[0..1] _LocalEnvironmentAggregationType
DecisionServices	[0..1] _LocalEnvironmentDecisionServicesType
HTTP	[0..1] _LocalEnvironmentHTTPType
File	[0..1] _LocalEnvironmentFileType
SOAP	[0..1] _LocalEnvironmentSOAPType
SCA	[0..1] _LocalEnvironmentSCAType
Sequence	[0..1] _LocalEnvironmentSequenceType
TCPIP	[0..1] _LocalEnvironmentTCPIPType
ServiceRegistry	[0..1] _LocalEnvironmentServiceRegistryType
ServiceRegistryLookupProperties	[0..1] _LocalEnvironmentServiceRegistryLookupType
Adapter	[0..1] _LocalEnvironmentAdapterType
Wildcard	[0..1] _LocalEnvironmentWildcardType
Variables	[0..1] _LocalEnvironmentVariablesType
CICS	[0..1] _LocalEnvironmentCICSType
FTE	[0..1] _LocalEnvironmentFTEType
Email	[0..1] _LocalEnvironmentEmailType
CD	[0..1] _LocalEnvironmentCDType
JMS	[0..1] _LocalEnvironmentJMSType
Mapping	[0..1] _LocalEnvironmentMappingType
MappingRoutine	[0..1] string
Database	[0..1] _LocalEnvironmentDatabaseType

A light bulb will appear on the left hand side of the transform.

- f. Click the yellow light bulb, and then select **Group the conflicting transforms in an override group**.

Results

You have configured your message flow to dynamically assign a message map to a Mapping node. In the first Mapping node, you have defined the logic to set the local environment **MappingRoutine** value. In the Mapping node where you want to dynamically assign a message map, you have defined a message map.

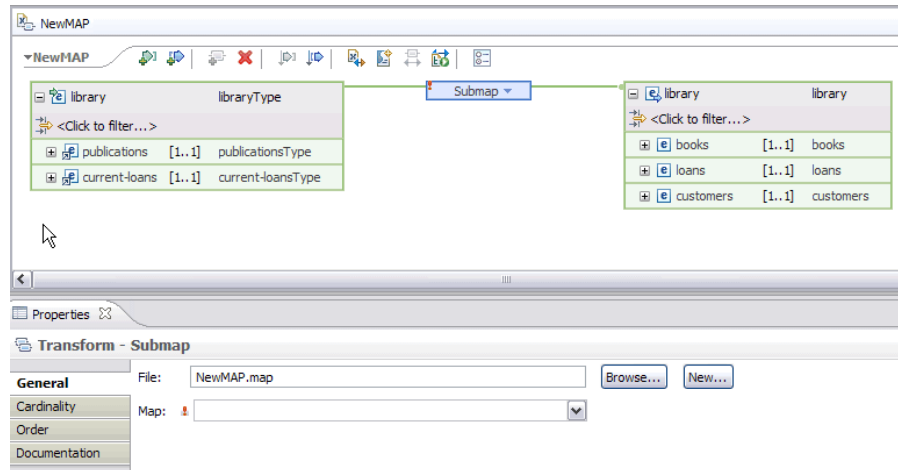
Calling a submap

You can call a submap from a graphical data map in the Graphical Data Mapping editor by using the **Submap** transform.

Procedure

Complete the following steps to call a submap from another graphical data map:

1. Create a connection between global input and output elements in a graphical data map, and then select the **Submap** transform on the connection: For example:



2. Click **Browse** in the Properties view of the **Submap** transform. The Submap Selection wizard is displayed where the dialogue box will display the submaps that are available.
3. Select a submap, and click **OK**.
You can choose to display only the valid maps that can be selected, by clicking **Show only applicable maps**.

Results

The submap is displayed in the Graphical Data Mapping editor, and you can edit it in the same way that you would edit any graphical data map. For information about how to edit maps, see Chapter 9, “Editing message maps,” on page 61.

Chapter 21. Transforming a SOAP message in a message map

In IBM Integration Bus, a SOAP message is described by a generic model that includes the SOAP *Envelope* and optionally *Attachments*. You define your SOAP message parts in a message map by using the **Cast** function.

About this task

A SOAP message consists of an *Envelope* and optionally *Attachments*. The envelope contains a SOAP header and a SOAP body. A SOAP body can include SOAP faults.

In IBM Integration Bus, when you use SOAP nodes, a SOAP message is described by a generic model.

In addition to the standard SOAP parts, the SOAP message generic model includes a *Context* part that includes contextual information about the current SOAP message that is processed. This part is the only one in a message map whose structure is included automatically. You must define the other SOAP message parts manually by using the **Cast** function.

The following table compares the SOAP message structure with the IBM Integration Bus SOAP message generic model:

Table 14. Comparison between the SOAP message structure and the IBM Integration Bus SOAP message representation

Standard SOAP message parts	Status	IBM Integration Bus SOAP message parts	IBM Integration Bus Status
		Context	Required
SOAP header (part of the SOAP envelope)	Optional	Header (part of the SOAP_Domain_Msg)	Optional
SOAP body (part of the SOAP envelope)	Required	Body (part of the SOAP_Domain_Msg)	Required
SOAP faults (part of the SOAP body)	Optional	Fault (part of the Body)	Optional
SOAP Attachments	Optional	Attachment (part of the SOAP_Domain_Msg)	Optional

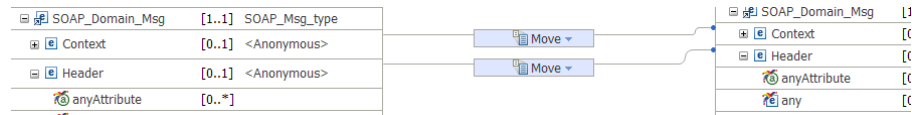
Procedure

Complete the following steps to configure the **SOAP_Domain_Msg** when the Mapping node is connected directly from a SOAPInput node with no SOAPExtract node:

1. Define the transformation for the **Context** object by using one of the following approaches:
 - Copy the **Context** object by using the **Move** transform.
 - Copy the **Context** object by using the **Move** transform, and then use the **Override** function if you want to modify a few elements with an **Assign** transform or a **Move** transform.
 - Define transforms for all the **Context** elements that you want to maintain in the output object.

2. Define the transformation for the **Header** object by using one of the following approaches:

- Copy the **Header** object by using the **Move** transform.



- Copy the **Header** object by using the **Move** transform, and then use the **Override** function if you want to modify a few elements with an **Assign** transform or a **Move** transform.
- Define transforms for all the **Header** elements that you want to maintain in the output object.

You define SOAP header parts by using the **Cast** function. You can cast attributes and other header parts. Then, define transforms between the input elements and the output elements in each header part.

The SOAP Header element contains application-specific information, including attributes that define how you process the SOAP message.

3. Define the transformation for the **Body** object.

You define SOAP body parts by using the **Cast** function. You can cast attributes and other body parts. Then, define transforms between the input elements and the output elements in each body part.

Complete the following steps to define the SOAP body parts and their transformations:

- Cast attributes that are defined as *xsd:any* into a specific type. Then, define a transform between the input attribute and the output attribute.
- Cast wildcard elements that are defined as *xsd:any* into a specific type.
- Cast a base type element to a derive type element. A derive type element is also known as an extension type element.

In a message map, you cast a base type to a derive type or extension type so that you can define transformations between subtypes of a data type. For example, addresses are represented differently for different countries. You might want to map addresses from different countries into a common complex structure for addresses.

- Define transforms between the input and output body elements.

4. Define the transformation for the **Attachment** object by using one of the following approaches:

- Copy the **Attachment** object by using the **Move** transform.
- Copy the **Attachment** object by using the **Move** transform, and then use the **Override** function if you want to modify a few elements with an **Assign** transform or a **Move** transform.

You define SOAP attachment parts by using the **Cast** function.

Example

Complete the steps in any of the following use cases to learn how to configure the **SOAP_Domain_Msg** when the Mapping node is wired directly from a SOAPInput node with no SOAPExtract node:

- Configure a SOAP message when you use a conditional transform to map an input element to an output element. For example, you create and configure the **If**, **Else if**, and **Else** transform to control the flow of the mapping between elements that are defined as a specific or a derive type in the input and output

message assembly by setting conditions. For more information, see “Mapping a SOAP message by using a conditional transform.”

- Configure a SOAP message to transform some input elements to output elements. Use the **Override** function, **Assign** transform, and **Move** transform. For more information, see “Mapping a SOAP message by using the **Override** function” on page 183.

What to do next

Define more transforms between the input **SOAP_Domain_Msg** and the output **SOAP_Domain_Msg**. For more information, see “Specifying a transform (mapping operation)” on page 82.

Mapping a SOAP message by using a conditional transform

You define SOAP message parts in a message map by using the **Cast** function and then you define transforms between its elements. To map between elements that are defined as specific or derived types, you might want to define some conditional transforms. You can configure the **If**, **Else if**, and **Else** transform to control the flow of the mapping between elements.

About this task

When you use an **If**, **Else if**, and **Else** transform between your **SOAP_Domain_Msg** input object and **SOAP_Domain_Msg** output object, you must manually configure each element in the **SOAP_Domain_Msg**. You must map each element in the **SOAP_Domain_Msg** input object to the corresponding output object so that you do not lose the information of the element.

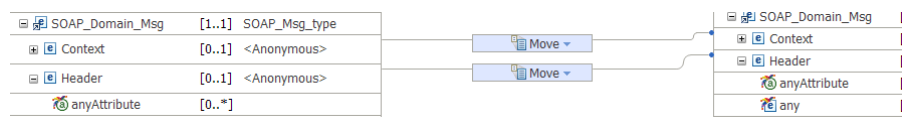
Note: Elements that are part of the input object and do not have a transform that is defined to an output object are deleted from the output structure and their value is lost.

Procedure

Complete the following steps to configure the **SOAP_Domain_Msg** when the Mapping node is wired directly from a SOAPInput node with no SOAPExtract node:

1. Define the transformation for the **Context** object.

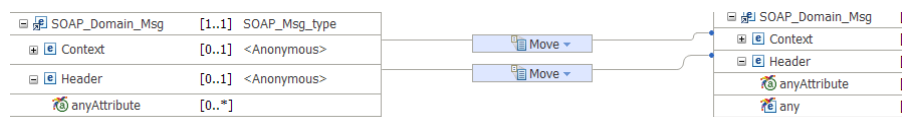
Copy the **Context** object by using the **Move** transform. For more information, see “Specifying a transform (mapping operation)” on page 82.



2. Define the transformation for the **Header** object.

You can define a **Move** transform between the input **Header** object and the output **Header** object to copy the structure and all its elements.

Copy the **Header** object by using the **Move** transform.



3. Define the transformation for the **Body** object. Define SOAP Body parts by using the **Cast** function. You can cast attributes and other body parts. Then, define transforms between the input elements and the output elements in each body part. For more information, see “Casting wildcards in a map” on page 65.
 - a. Cast attributes that are defined as *xsd:any* into a specific type. Then, define a transform between the input attribute and the output attribute.
 - b. Cast wildcard elements that are defined as *xsd:any* into a specific type.
 - c. Cast a base type element to a derive type element. A derive type element is also known as an extension type element.

In a message map, you cast a base type to a derive type or extension type so that you can define transformations between subtypes of a data type. For example, addresses are represented differently for different countries. You might want to map addresses from different countries into a common complex structure for addresses.
 - d. Create and configure the **If, Else if, and Else** transform to control the flow of the mapping between elements that are defined as a specific or a derive type in the input and output message assembly by setting conditions.
4. Optional: Define the transformation for the **Attachment** object. Copy the **Attachment** object by using the **Move** transform.

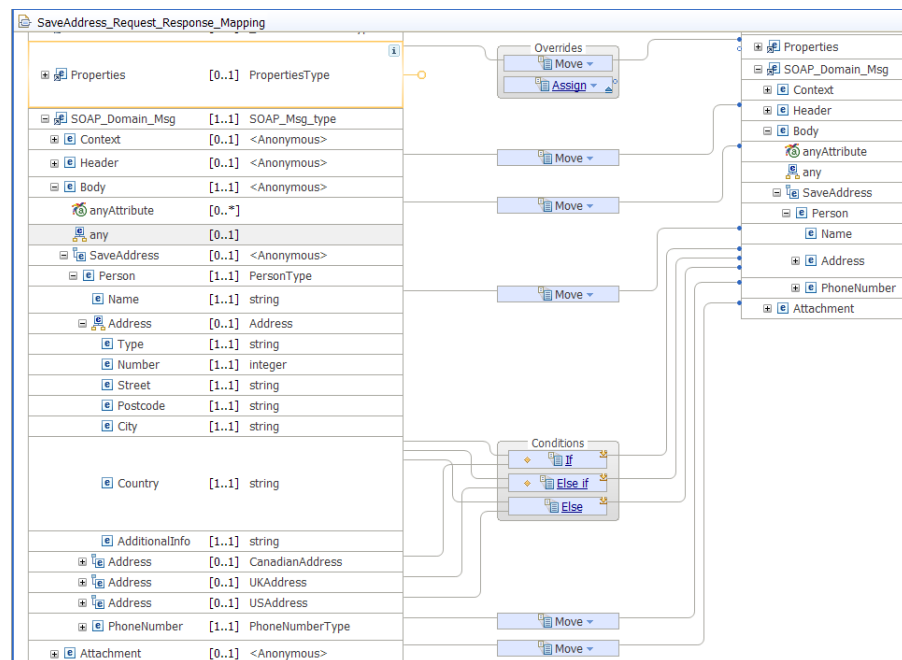
You define SOAP attachment parts by using the **Cast** function.

Results

You have a message map that transforms a SOAP message. The message map contains a nested map that uses the **If, Else if, and Else** transform.

Example

The following figure shows a message map after you complete the previous steps to transform a SOAP message:



What to do next

Define more transforms between the input **SOAP_Domain_Msg** and the output **SOAP_Domain_Msg**. For more information, see “Specifying a transform (mapping operation)” on page 82.

Mapping a SOAP message by using the Override function

In a SOAP message, you can use the **Override** function to copy a complex type from the input message to the output message, while you update some of the child elements in the complex type with an **Assign** transform, and a **Move** transform.

Before you begin

You define your SOAP message parts in a message map by using the **Cast** function. For more information, see “Casting wildcards in a map” on page 65.

About this task

Note: You can include **Move** transforms and **Assign** transforms within an **Override**.

Procedure

Complete the following steps to configure the **SOAP_Domain_Msg** when the Mapping node is wired directly from a SOAPInput node with no SOAPExtract node:

1. Define a **Move** transform to copy all the elements in the input **SOAP_Domain_Msg** message unchanged.
2. Change the value of at least one element by using an **Assign** transform or a **Move** transform.
3. Apply a quick fix to configure the **Override** function. Select the option **Group the conflicting transforms in an override group**.

Results

You transformed the SOAP message by using the **Override** function.

What to do next

Define more transforms between the input **SOAP_Domain_Msg** and the output **SOAP_Domain_Msg**. For more information, see “Specifying a transform (mapping operation)” on page 82.

Chapter 22. Creating or transforming a BLOB output message by using a graphical data map

Use the Graphical Data Mapping editor to create or transform a message using the BLOB message domain.

About this task

When you create a new map, the BLOB message is available under the "IBM supplied message models" category in the map input and output selection tree. If you select a BLOB message as the map output, the output domain in the output message assembly is automatically set to BLOB. For more information about creating a new map, see "Creating a message map from a Mapping node" on page 49.

The BLOB message provides a single element value of type `xsd:hexBinary`. When you create a BLOB message output, you must wire a transform to the value element in the BLOB message, which sets the `hexBinary` data that you require in the BLOB message.

When you transform a BLOB message, you must wire the value element of the input BLOB message into a transform that accepts an `xsd:hexBinary` type. For example, you could use a Move transform to set the BLOB binary data into an XML output message element that has either an `xsd:hexBinary` or `xsd:base64Binary` type.

If you are transforming only the Message Assembly headers and folders (for example, Properties and Transport headers), you can pass the message body data through unmodified by using the BLOB domain and a Move transform from the BLOB input message to the BLOB output message.

Chapter 23. Mapping from a BLOB message to an output message using a graphical data map

You can use the Graphical Data Mapping editor to transform a message in the BLOB message domain.

About this task

When you create a new map, the BLOB message is available under the "IBM supplied message models" category in the map input and output selection tree. Select the BLOB message as the map input, and then select your required output message and set its domain in the output message assembly (for example, XMLNSC).

For more information about creating a new map, see "Creating a message map from a Mapping node" on page 49.

The BLOB message provides a single element value of type `xsd:hexBinary`. When you transform a BLOB message, you must wire the value element of the input BLOB message into a transform that accepts an `xsd:hexBinary` type. For example, you could use a Move transform to set the BLOB binary data into an XML output message element with either an `xsd:hexBinary` or `xsd:base64Binary` type.

Chapter 24. Troubleshooting graphical data maps

Diagnose and solve problems that you encounter when using your graphical data maps.

About this task

The Mapping node reports the running of graphical data map scripts as detailed user trace events.

The user trace events report the entry and completion of each transform in a map, and the setting of values in the map output.

The collection and review of IBM Integration Bus user trace enables you to troubleshoot transformation logic that you build in your graphical data maps.

user trace enables you to troubleshoot transformation logic that you build in your graphical data maps.

The following topics describe how to diagnose problems by using user trace:

- Debugging with user trace
- Starting user trace
- Checking user trace options
- Changing user trace options
- Stopping user trace
- Retrieving user trace

Chapter 25. Deploying message maps

By default, message map files are deployed in BAR files as a part of an application, integration service, or library providing an integration solution. You can also deploy a map as an independent resource if you are managing your message flows that way. If you change a message map, you must redeploy your integration solution, or independent message flows.

About this task

IBM Integration Bus prepares message maps for execution on deployment instead of when the first message is flowed through the Mapping node.

This behavior has the following advantages:

- There is no drop in performance from initializing a message map when the first message is flowed through the node.
- The message map and its dependencies, such as any referenced message models, are resolved and validated during deployment to ensure that the message map runs successfully on first message.
- The message map syntax is validated during deployment to ensure that the message map runs successfully on first message.
- When IBM Integration Bus is restarted, the message map syntax and its dependencies are validated before the message flow can be restored.

Note: To avoid a deployment failure, you must include all the message map dependencies, referenced schemas, ESQL modules, Java classes, and other resources in your BAR file. You must resolve any message map static errors such as an invalid XPath expression. If these requirements are not met, you receive a BIP message that reports the map generation failure.

Procedure

When you deploy message maps, the behavior of IBM Integration Bus is as follows:

- Behavior when you deploy or redeploy a BAR file:
 1. All message maps are validated to ensure that all the map dependencies can be resolved at run time. This validation step checks that the referenced message models such as XML schema files, DFDL schema files, and message set files, and the referenced submaps can be resolved.
 2. The message maps and their dependencies are generated to an executable form. This step also checks that the contents of the map and submaps are valid, and that they have no errors such as an invalid XPath expression.
 3. If the message maps and their dependencies are valid and can be successfully generated, they are persisted in both the deployed and generated forms to the configuration store. Otherwise the deployment is aborted and you receive a BIP message that reports the map generation failure.
- Behavior after deployment:

1. Background processing is started to compile the generated message maps to Java byte code so that they can benefit from JIT optimization. The Java byte code for each map is persisted on completion of the compilation.
- Behavior when the first message flows runs after deployment or redeployment:
 1. If the background compilation processing is complete, the Mapping node executes the java byte code and the JIT optimization starts.
 2. If the background compilation processing is not complete, the Mapping node runs the map initially in the generated form until the compilation is complete. Then, JIT optimization starts.
- Behavior for any subsequent messages that flow after deployment or redeployment:
 1. The prepared message map runs. If the background processing is completed, the JIT optimization starts.
- Behavior when you restart IBM Integration Bus, that is, when you restart an integration node, an integration server, an integration solution, or a message flow:
 1. If the compiled Java byte code for the map is available, it is loaded and the Mapping node will run this code as soon as the first message is processed. Then, JIT optimization will start.
 2. If the generated form for the map is available and loaded, background processing is started to compile the generated map to Java byte code so that they can benefit from JIT optimization. The Java byte code for each map is persisted on completion of the compilation.
 3. If neither the generated or the compiled map code are available, the map is processed in the same way as when you deploy or redeploy a BAR file.

Chapter 26. Transform types in the Graphical Data Mapping editor

In the Graphical Data Mapping editor, you can map elements and attributes between the input and output objects. You can apply a transform to the mapping that specifies the action to be performed on the input data. The result of the transform is stored in the output element.

The following table shows the standard mapping transforms that are provided by the Graphical Data Mapping editor:

Table 15. Core mapping transforms in the Graphical Data Mapping editor:

Transform	Description
"Assign" on page 198	Sets a value in the output element. There is no input element. Column values set via Assign transform will always be passed as character string.
"Setting the value of a simple output element to a default or fixed value" on page 130	Sets a specific value type in the output element. Cast can also move and convert an input element to become a specific value type in the output element.
"Concat" on page 201	Creates a string concatenation that allows you to retrieve data from two or more entities and link them into a single result.
"Convert" on page 204	Copies the input element to the output element and changes the type. The transform takes a single simple input and creates a single simple output with a different type.
"Create" on page 204	Creates an empty element, a nil element, or a simple type element by using a default value that is based on the element's type.
"Custom ESQL" on page 208	Enables you to enter your own ESQL code to be used in the transform.
"Custom Java" on page 212	Enables you to enter your own Java code to be used in the transform.
"Custom XPath" on page 214	Enables you to enter your own XPath expressions to be used in the transform..
"Move" on page 225	Copies data from the input element to the output element.
"Normalize" on page 226	Normalizes the input string by removing white space such as spaces, tabs, and returns, and moves the resulting normalized string to the output element.
"Substring" on page 227	Extracts information as required, and moves the extracted string to the output element.
Task	Describes a manual task or point of concern that might need to be reviewed or resolved before a message map can be used in your solution.

Table 15. Core mapping transforms in the Graphical Data Mapping editor: (continued)

Transform	Description
“Built-in XPath transforms” on page 229	All XPath 2.0 functions are supported, in the form fn:<function_name>.

The following table shows the database transforms that are provided by the Graphical Data Mapping editor:

Table 16. Database transforms in the Graphical Data Mapping editor:

Transform	Description
“Database Routine” on page 216	Calls a stored procedure or user-defined function from a database.
“Delete” on page 216	Deletes one or more rows in a database table that is matched by a Where clause.
“Failure” on page 217	Enables the map to take-on error handling for any exceptions that are raised by the database server in a database transform, instead of having such exceptions stop the map and be reported.
“Insert” on page 223	Inserts a row into a database table.
“Return” on page 226	Enables extra processing after a successful Insert, Update, or Delete database operation, or Database Routine call. Provides the results from the database operation or call as inputs.
“Select” on page 227	Retrieves data from rows in a database table, so that the data can be used as input in a message map.
“Update” on page 228	Updates one or more rows in a database table that is matched by a Where clause with a single set of data values.

In addition to the core mapping transforms, several structural transforms are provided. The structural transforms control how nested elements are displayed in the Graphical Data Mapping editor, but they have no effect on the data itself. The structural transforms are described in the following table:

Table 17. Structural mapping transforms in the Graphical Data Mapping editor:

Transform	Description
“Append” on page 195	Appends occurrences of an output array in the order of the inputs.
“For Each” on page 218	Iterates over an input array element (either a simple type or a complex type).
“Group” on page 222	Takes a single input array and produces a set of nested output arrays that collate elements of the input array.
“If, Else if, and Else” on page 222	Enables you to control the flow of the mapping by setting conditions.
“Join” on page 223	Joins elements from two or more inputs.
“Local map” on page 225	Provides a hierarchical view of element transforms in the message map.

Table 17. Structural mapping transforms in the Graphical Data Mapping editor: (continued)

Transform	Description
“Submap” on page 227	References another map. It calls a map from this or another map file, which can be stored in a library, an application, an integration service, or an Integration project.

Append

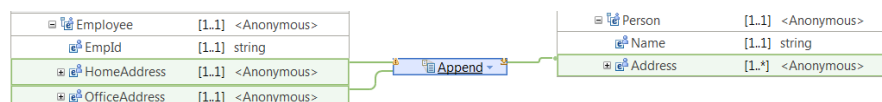
You can use the **Append** transform to create an output array in the order of the inputs.

Overview

The **Append** transform is not available in the list of available transforms until you wire at least two inputs to a transform.

To build an output array of N elements where you have N inputs available in a flat structure, you wire each input to the **Append** transform.

For example, the following figure shows two flat structures, *HomeAddress* and *OfficeAddress* wired into an **Append** transform to create the *Address* array.



To build an output array of N elements where you have less than N unique inputs to wire into the **Append** transform, you can wire the same input multiple times.

Inputs to the Append transform

The **Append** transform takes multiple inputs of either simple type or complex type.

You wire inputs to the **Append** transform as primary connections.

You can wire the same input into an **Append** transform more than once if you need to produce more than one array instance from a single input.

The **Append** transform iterates over multiple inputs in the specified order to append data. The order of inputs into the transform is recognized, and is set in the **Order** property page.

When you connect a repeating input, each instance adds an extra occurrence to the output array. You can use the **Cardinality** property page on the transform to specify a subset of indexes the transform should process. The first index element is 1.

Inputs of single, non-repeating elements are also allowed. Each single input element adds an extra occurrence to the output array.

The **Append** transform provides a nested transform for each input in the output array.

The nested transforms are performed for each input sequentially, producing occurrences of the output array. First over all elements in the first input, then over all elements in the second input.

Output of the Append transform

The output of an **Append** transform can be a simple type array or a complex type array.

The output array size is the sum of the input elements wired to the **Append** transform.

Note: You can define any number of inputs to the **Append** transform. The Graphical Data Mapping editor does not validate the number of inputs. You must ensure that the number of wired inputs to the **Append** transform correspond to the value configured in the **Maximum occurrence** property of the output array.

Order of the inputs

By default, the order of the inputs to the **Append** transform is the order in which you wire the inputs.

You can modify the order by reordering the inputs in the **Order** tab of the transform properties.

Transform - Append

Cardinality

Variables

Condition

Sort

Order

Documentation

Inputs:

Input	Parameter
HomeAddress : <Anonymous>	
OfficeAddress : <Anonymous>	

Reorder

Outputs:

Parameter	Output
	Address : <Anonymous>

Reorder

Define a conditional expression

You can define **supplement** connections between input elements and the **Append** transform. You can then use these input elements in a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

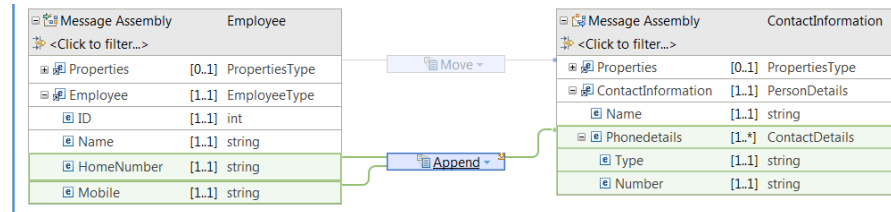
For more information, see “Configuring the properties of a transform” on page 84.

Example

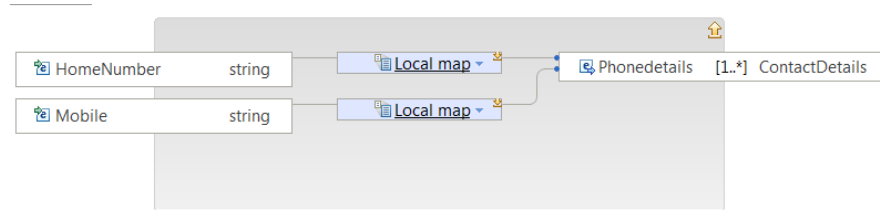
This example shows how to create an output array with two indexes by using the **Append** transform.

You connect two input elements (**HomeNumber** and **Mobile**) to the **Append** transform. These inputs will be used to set the value of multiple indexes in the output repeating element **Phonedetails**.

You also define a connection from the **Append** transform to the repeating element **Phonedetails**.

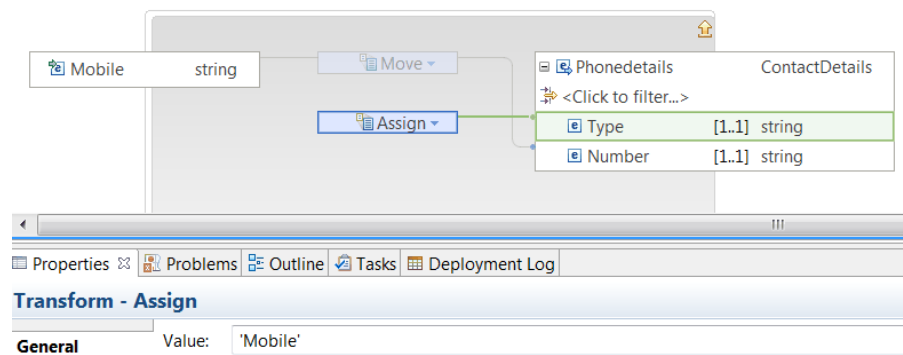


When you open the nested map associated to the **Append** transform, you get two **Local Map** transforms, one per input element.

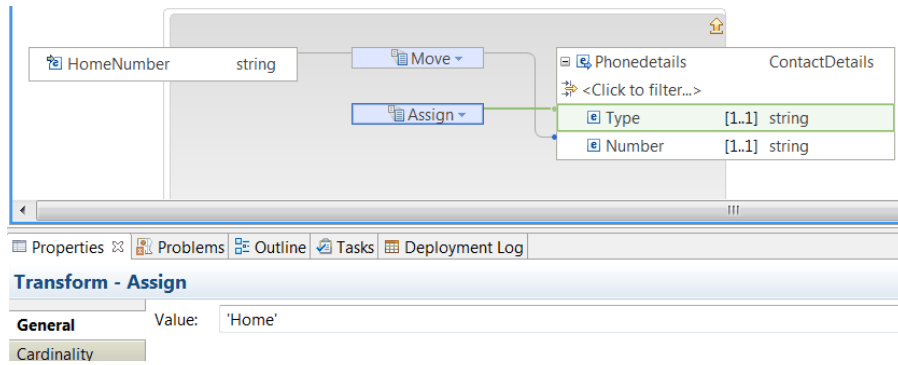


Inside each **Local Map** transform, you define the transformation logic that sets the output values for one index of the output repeating structure **Phonedetails**.

The following figure shows the nested map associated with the **Local Map** transform for the input element **Mobile**:

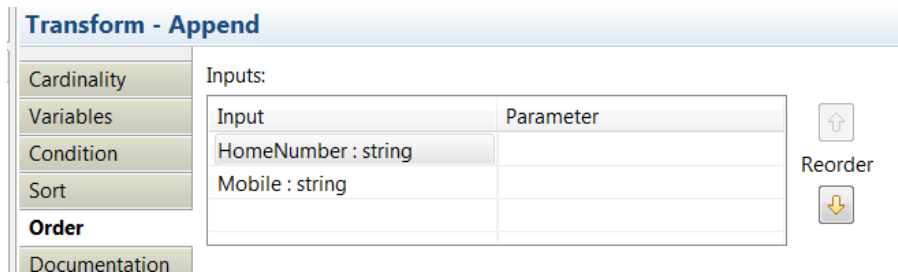


The following figure shows the nested map associated with the **Local Map** transform for the input element **HomeNumber**:



You can configure the order in which the indexes are created. You set the order in which indexes are created in the **Order** tab of the Properties view of the **Append** transform.

In this example, the first index contains the information of the home number. The second index contains the information of the mobile number.



Assign

You can use the **Assign** transform to set the value of an output element to a constant or fixed value.

Overview

The **Assign** transform sets a value that can be a fixed value or it can be the result of a function that has no input; for example, a current date function.

You cannot use the value of an input element to set the value of an output element with the **Assign** transform.

The output element is set to a constant value.

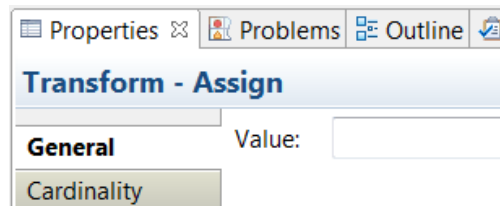
The output element must be a simple type element.

For more information about assigning a value to an output element, see “Setting the value of an output element to a simple data type” on page 126.

Assign a value

You must assign a value in the **Value** field that is located in the **General** tab of the **Assign** transform properties view.

If you do not specify a value, an empty element is created.



Define a conditional expression

You can define **supplement** connections between input elements and the **Assign** transform. You can then use these input elements in a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.



When you define supplement connections, you can configure the following properties to define how those inputs are displayed in the message map:

- **Sort:** You can sort the inputs to the transform by ascending order, descending order, case order, or data order.
- **Order:** You can display the order of input connections to a transform. You can reorder them.

For more information, see “Configuring the properties of a transform” on page 84.

Default values

The following table lists the default values set when you use the **Assign** transform:

Table 18. Default values set by using the **Assign** transform

Type	Default value
string	Empty string
dateTime	2002-01-01T11:00:00
Boolean	false
decimal	0.0
double	0.0
hexBinary	00
long	0
duration	P1Y
time	00:00:00
date	2002-01-01

Cast type (xs:type)

You can use an **xs:type** transform to cast the value of a simple element to a specific data type.

Overview

For example, you might want to assign a value with a specific data type to a target element that is defined as `xs:anySimpleType`.

When you use an `xs:type` transform, you can have zero or more input elements.

- You cast one of the input elements to set the value of an output element.
- You can use any of the input elements to build an XPath conditional expression that determines whether the `xs:type` transform is applied or not.

You must choose the `xs:type` transform according to the output element data type. For example, if you have an output element with a Boolean data type, you must choose `xs:boolean` transform.

For more information about casting a specific value type to an output element, see “Setting the value of an output element with a explicit data type” on page 128.

Assign a value

You can set a fixed value or define an XPath expression in the **Value** field that is located in the **General** tab of a `xs:type` transform properties view.

Transform - boolean

Description: Takes a primitive and casts it as a boolean.

Cardinality:

Variables:

Parameters:

Name	Type	Value
primitive	xs:anyAtomicType	"

Buttons: Add, Edit..., Remove

To define an XPath expression, you click **Edit**. Then, you can use content-assist to enter the expression. As part of the expression, you can use any input elements for which you define connections to the transform.

Define a conditional expression

You can define multiple connections between input elements and an `xs:type` transform. You can then use these input elements in a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

When you define multiple input connections, you can configure the following properties to define how those inputs are displayed in the message map:

- **Sort**: You can sort the inputs to the transform by ascending order, descending order, case order, or data order.
- **Order**: You can display the order of input connections to a transform. You can reorder them.

For more information, see “Configuring the properties of a transform” on page 84.

xs:type transforms

You can use any of the following xs:any transforms:

- **xs:NOTATION**: This function takes a primitive and casts it as notation.
- **xs:Qname**: This function takes a primitive and casts it as a qualified name.
- **xs:anyURI**: This function takes a primitive and casts it as anyURI.
- **xs:base64Binary**: This function takes a primitive and casts it as base64Binary.
- **xs:boolean**: This function takes a primitive and casts it as boolean.
You can use any of the following values: true Or false.
- **xs:dateTime**: This function takes a primitive and casts it as dateTime.
- **xs:date**: This function takes a primitive and casts it as date.
- **xs:dayTimeDuration**: This function takes a primitive and casts it as dayTimeDuration.
- **xs:decimal**: This function takes a primitive and casts it as decimal.
- **xs:double**: This function takes a primitive and casts it as double.
- **xs:float**: This function takes a primitive and casts it as float.
- **xs:gDay**: This function takes a primitive and casts it as gDay.
- **xs:gMonthDay**: This function takes a primitive and casts it as gMonthDay.
- **xs:gMonth**: This function takes a primitive and casts it as gMonth.
- **xs:gYearMonth**: This function takes a primitive and casts it as gYearMonth.
- **xs:gYear**: This function takes a primitive and casts it as gYear.
- **xs:hexBinary**: This function takes a primitive and casts it as hexBinary.
- **xs:integer**: This function takes a primitive and casts it as integer.
- **xs:int**: This function takes a primitive and casts it as signed 32-bit integer.
- **xs:string**: This function takes a primitive and casts it as string.
- **xs:time**: This function takes a primitive and casts it as time.
- **xs:yearMonthDuration**: This function takes a primitive and casts it as yearMonthDuration.

Concat

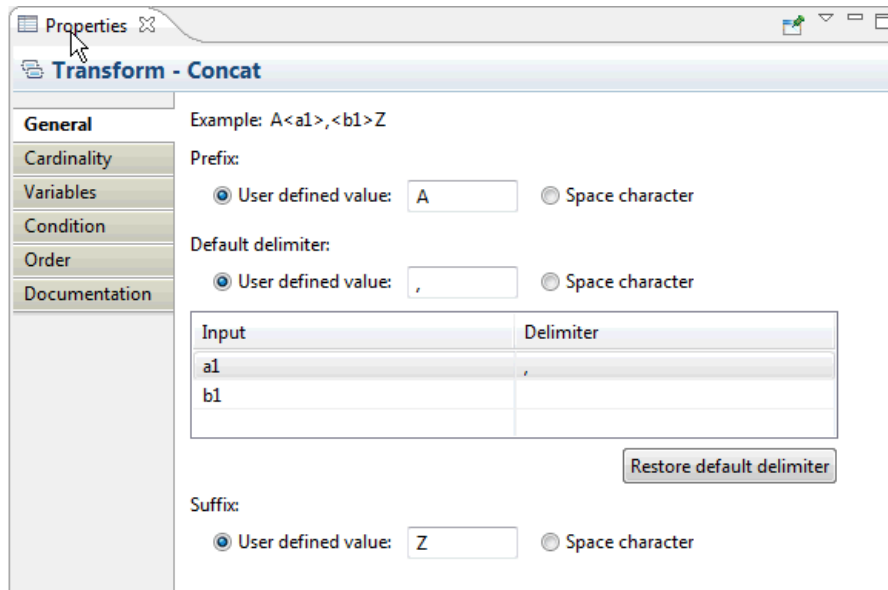
You can use a **Concat** transform to concatenate data from two or more simple elements into a string output element.

Overview

The **Concat** transform concatenates two or more simple inputs into a string output element.

When you configure the **Concat** transform, you can specify a prefix, a suffix, and a delimiter through the Properties page:

You can specify an alphanumeric character to be the delimiter between the strings,. You can also use a string prefix and a string suffix.



For example, you can concatenate the strings from the elements `firstname` and `lastname`, and specify a space as the delimiter, a prefix of `Mr.` , and a comma as the suffix, with the following result: `Mr. firstname lastname`,

When can you use the Concat transform?

You can use the **Concat** transform when the following premises apply:

- You want to concatenate data from two or more single type inputs.
- The input types to the **Concat** transform can be any simple or primitive data types.

Note: Simple type input elements that are not of type `xs:string` will be cast to `xs:string`.

- All the inputs to the **Concat** transform, that are connected as primary connections, are used to calculate the value of the output string element.
- You might need to define a prefix.
- You might need to define the same delimiter between input values.
- You might need to define a suffix.

You cannot select and use the **Concat** transform when any of the following criteria on inputs applies:

- One of the inputs to the **Concat** transform is a complex type element.
- One of the inputs is a repeating element, that is, the input element cardinality is set to `[1..*]` or `[0..*]`.

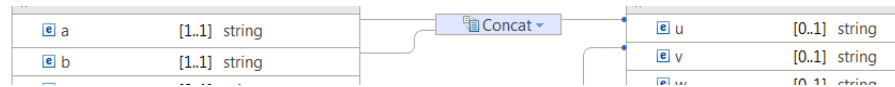
Inputs to the Concat transform

The **Concat** transform takes multiple simple type elements.

You wire inputs to the **Concat** transform as primary connections.

You can wire the same input into a **Concat** transform more than once.

The **Concat** transform concatenates input data in order. The order of inputs into the transform is recognized, and is set in the **Order** property page.



Order of the inputs

By default, the order of the inputs to the **Concat** transform is the order in which you wire the inputs.

You can modify the order by reordering the inputs in the **Order** tab of the transform properties.

Define when the transform is applied at run time

You can use any of the input elements to the **Concat** transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

Warning error

By default, you cannot connect a repeating simple element to a **Concat** transform. However, if you have a map where you have defined a **Concat** transform, and you change the cardinality of one of the input elements so it becomes a repeatable simple type, the **Concat** transform will show a warning.

The warning message is the following:

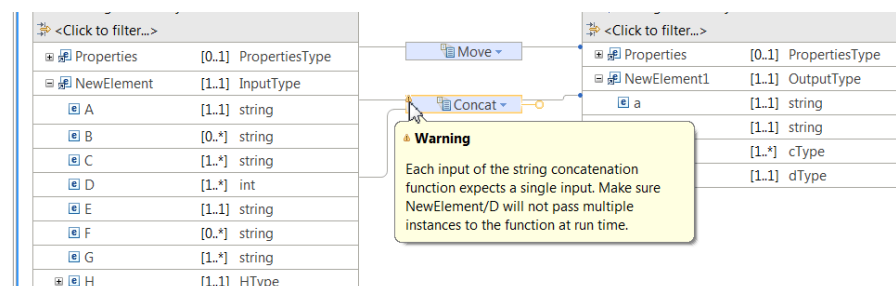
Each input of the string concatenation function expects a single input. Make sure {0} will not pass multiple instances to the function at run time.

The warning is displayed because at least one input is not of single type.

When a **Concat** transform has such a warning, the run time behavior is the following:

- If the input XML has no more than one instance of the repeatable input, the **Concat** transform produces the expected result at run time.
- If the input XML has more than one instance of the repeatable input, the **Concat** transform results in a run time exception.

Example for a repeating single input element to a Concat transform:



Convert

The **Convert** transform implements a schema type cast on the input to match the output. If the type cast cannot be performed on the input instance value, an exception is thrown and the map processing stops.

Use the **Convert** transform to move a simple input element to an output element, when both elements are defined in the relevant schema models with different types. For example, you might have an input element with type `xsd:int` and an output element with type `xsd:decimal`. The **Convert** transform changes the input with an `int` type to an output with a `decimal` type.

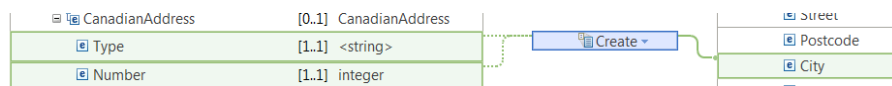
Create

You can use the **Create** transform to initialize an output element, set an output element to nil, or set an output element to fixed value defined by the schema.

Overview

For simple types, you cannot use the value of an input element to set the value of an output element in a **Create** transform.

However, you can define zero or more **supplement** connections between input elements and the **Create** transform. You can then use these input elements in a conditional expression that defines the condition under which the transform is applied.



You can use the **Create** transform to create the following types of output element:

- An empty element. For more information, see “Initializing a simple or complex output element by using the **Create** transform” on page 135.
- An element with `xsi:nil="true"`, also called a nil element. For more information, see “Choosing an XPath conditional expression that tests for a nil value in a transform” on page 91.
- A simple type element with a default value that is specified by the schema. For more information, see “Setting the value of a simple output element to a default or fixed value” on page 130.
- A complex element with child elements populated by using the **Assign** transform or the **Move** transform from supplemental wired inputs.

Set the value of a string type output element

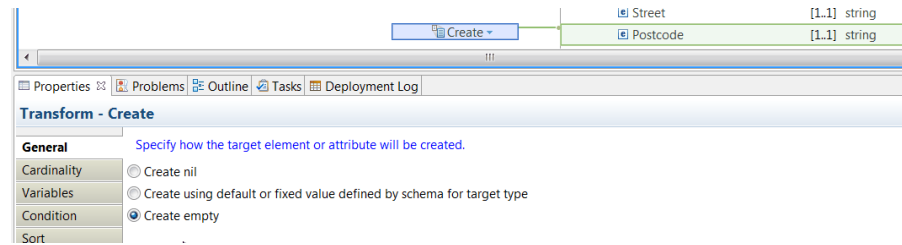
By default, the option **Create empty** is preselected. You must choose this option to initialize a single type output element.

For example, for an element that is defined in the schema model as `<element name="Postcode" type="string" nillable="true" default="30567" />`, you can choose any of the following options:

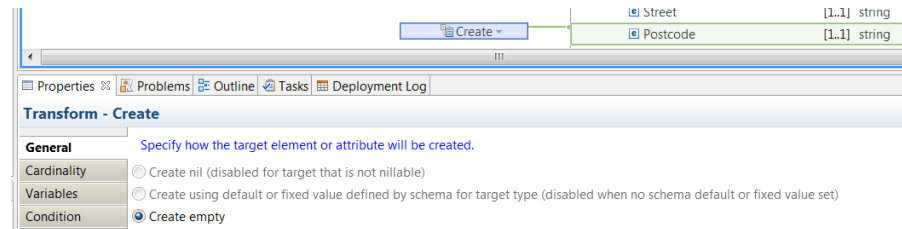
- Create nil
- Create by using a default or a fixed value that is defined by schema for target type

- Create empty

The following figure shows the properties view for the **Create** transform defined for that element:



For example, if the output element is defined in the schema model as `<element name="Postcode" type="string" nillable="false" />`, you can choose to create the output element as empty.



The option to create a nil element is available only when the target is nillable.

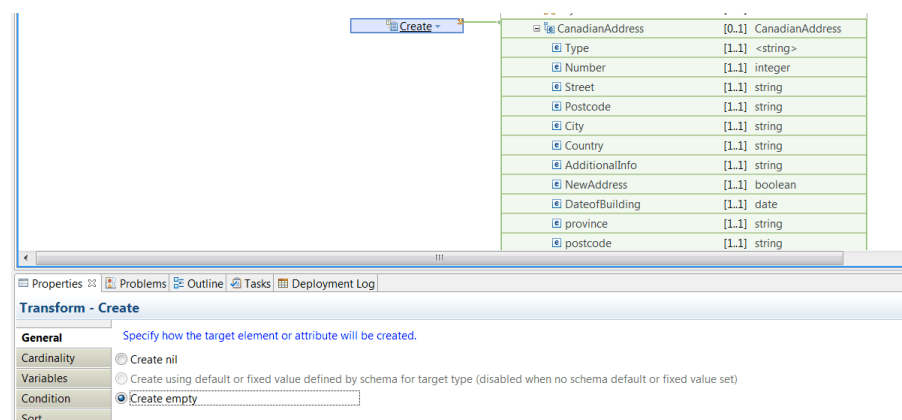
The option to create an element with the default value is only available when the target has a default value that is defined in the schema file.

Set the value of a complex type output element

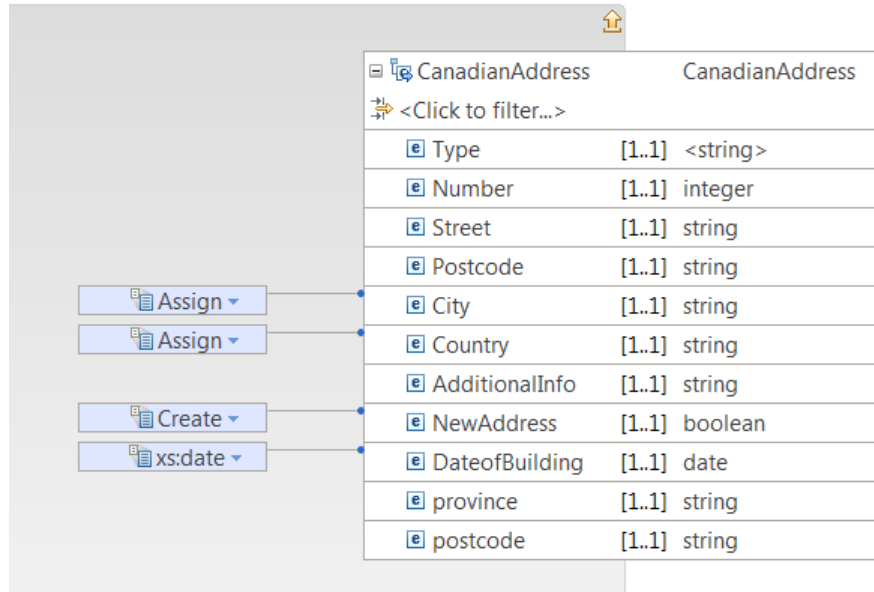
By default, the option **Create empty** is preselected. You must choose this option to initialize a complex output element.

Note: You can initialize or set to nil complex output elements.

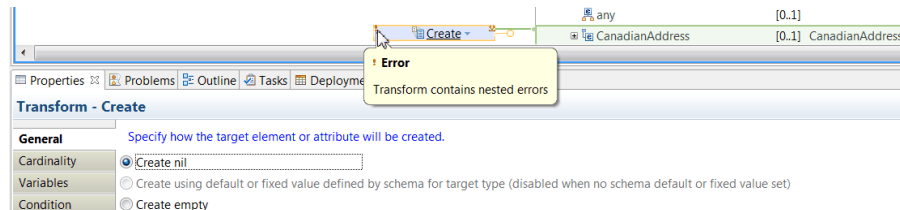
When you define a **Create** transform to initialize a complex type output element, a nested map is created containing the complex structure elements. You configure each element within the nested map individually, that is, you initialize, set to nil, or set to a default value each individual element.



When you initialize a complex type, you can use the nested map that is associated with the **Create** transform to provide values for the child element. You can define and configure each element by using transforms that do not require an input, for example the **Assign** transform, an **xs:type** transform, or the **Create** transform. You can also wire a supplemental input to move input values to the child elements.



When you define a **Create** transform to set to nil a complex type output element, a nested map is created containing the complex structure elements. You can get the error Transform contains nested errors if you try to configure any element within the nested map individually.



Specify how a simple data type element or attribute is created

You must specify how an output element or an attribute is created in the **General** tab of the **Create** transform properties view.

The following table lists simple data types with their default options preselected when you define a **Create** transform. It lists the options that you can choose to set the value of an output element of that type:

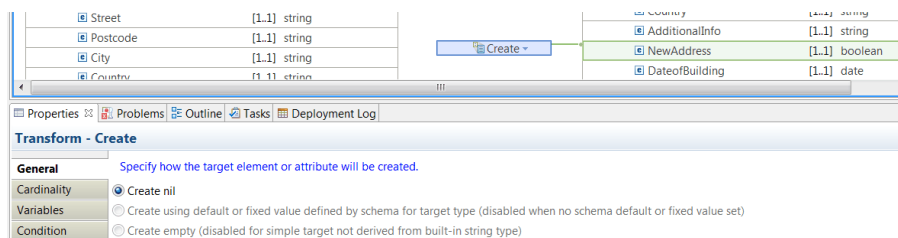
Table 19. Configuration choices available when you define a **Create** transform to set the value of an output element

Data type	Default option	Output element can be set to empty	Output element can be set to nil (the element is defined as <i>nillable="true"</i>)	Output element can be set to fixed or default value (the <i>default</i> property is set in the schema)
String	Create empty	valid	valid	valid
Integer	Create nil	not valid	valid	valid
Boolean	Create nil	not valid	valid	valid
date	Create nil	not valid	valid	valid
dateTime	Create using default or fixed value defined by schema for target type	not valid	not valid	valid
double	Create using default or fixed value defined by schema for target type	not valid	not valid	valid
float	Create using default or fixed value defined by schema for target type	not valid	not valid	valid
hexBinary	Create empty	valid	valid	valid
int	Create using default or fixed value defined by schema for target type	not valid	not valid	valid
time	Create using default or fixed value defined by schema for target type	not valid	not valid	valid

For example, for the following schema definition of an output element:

```
<element name="NewAddress" type="boolean" nillable="true" />
```

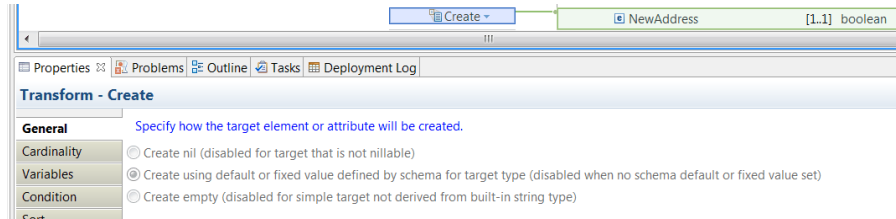
When you define a **Create** transform to set the value of a Boolean element, you can create the element as nil.



For example, for the following schema definition of an output element:

```
<element name="NewAddress" type="boolean" nillable="false" />
```

When you define a **Create** transform to set the value of a Boolean element, you can specify a fixed value, and the choice is preselected.



Define a conditional expression

You can define **supplement** connections between input elements and the **Create** transform. You can then use these input elements in a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

When you define supplement connections, you can configure the following properties to define how those inputs are displayed in the message map:

- **Sort:** You can sort the inputs to the transform by ascending order, descending order, case order, or data order.
- **Order:** You can display the order of input connections to a transform. You can reorder them.

For more information, see “Configuring the properties of a transform” on page 84.

Custom ESQL

You use the Custom ESQL transform to call your own ESQL code from a graphical data map.

In the Graphical Data Mapping editor, select **Custom ESQL** from the **Custom Transforms** list. You can then use the transform properties to select ESQL code that is stored in your workspace. When you select the ESQL route, the Parameters table **Name** and **Type** columns are populated. You must then select an input element or XPath expression in the **Value** column for each parameter. You can use the content assist in the **Value** column to help you to assign the required element, literal, or XPath expression.

The following topics contain further information about ESQL types and functions:

- “Equivalent ESQL types and schema types” on page 209
- “Equivalent ESQL and XPath mapping functions” on page 210

The ESQL file that contains the referenced ESQL module must be visible for mapping to be selectable. Ensure the application, library, or project references are set to make the ESQL file accessible to the map. When you deploy the map, ensure that the ESQL file is also deployed, and that **Compile and in-line resources** is not selected.

Requirements for ESQL modules called from a graphical data map

The following requirements apply to ESQL modules that are called from a graphical data map:

- The syntax of an ESQL procedure is shown in CREATE PROCEDURE statement. The procedures that can be called from a Custom ESQL transform in a graphical data map must conform to the following requirements:
 - Only IN parameters are allowed
 - A RETURN is required
- The input and return data types must be simple scalars; ESQL reference data types are not supported.
- An ESQL module with no inputs can be used to assign to an output element.
- Each input parameter to the ESQL module can be taken from an input element that is wired into the custom ESQL transform or specified as a constant.
- The ESQL must not include SQL calls to a data source. The Graphical Data Mapping editor provides facilities to include database operations in the map. For more information, see Chapter 19, “Mapping database content,” on page 159.

Equivalent ESQL types and schema types

When you need to process the data to set a target value, you can implement a function in XPath Java or ESQL. If you choose to use ESQL functions you can invoke them from your map in a custom ESQL transform.

The following table shows the equivalence between schema types and ESQL types:

XML Schema simple type	ESQL data type in message tree
anyURI	CHARACTER
base64Binary	BLOB
boolean	BOOLEAN
byte	INTEGER
date	DATE
dateTime	TIMESTAMP
dayTimeDuration	INTERVAL
decimal	DECIMAL
double	FLOAT
duration	INTERVAL
ENTITIES	Repeating set of elements in the tree, each of type CHARACTER
ENTITY	CHARACTER
float	FLOAT
gDay	DATE
gMonth	DATE
gMonthDay	DATE
gYear	DATE
gYearMonth	DATE
hexBinary	BLOB

XML Schema simple type	ESQL data type in message tree
ID	CHARACTER
IDREF	CHARACTER
IDREFS	Repeating set of elements in the tree, each of type CHARACTER
int	INTEGER
integer	DECIMAL
language	CHARACTER
long	INTEGER
Name	CHARACTER
NCName	CHARACTER
negativeInteger	DECIMAL
NMTOKEN	CHARACTER
NMTOKENS	List of CHARACTER
nonNegativeInteger	DECIMAL
nonPositiveInteger	DECIMAL
normalizedString	CHARACTER
NOTATION	CHARACTER
positiveInteger	DECIMAL
QName	CHARACTER
short	INTEGER
string	CHARACTER
time	TIME
token	CHARACTER
unsignedByte	INTEGER
unsignedInt	INTEGER
unsignedLong	DECIMAL
unsignedShort	INTEGER
yearMonthDuration	INTERVAL

Equivalent ESQL and XPath mapping functions

You can implement some data mapping functions either by using XPath functions or by supplying equivalent ESQL functions in a Custom ESQL transform.

The following functions in the ESQL language have equivalent XPath functions built into the Graphical Data Mapping editor. You can invoke these functions directly without having to write ESQL modules in an ESQL file to be called from the map:

Table 20. Equivalent ESQL and XPath functions

ESQL function	XPath function
EXTRACT YEAR FROM	fn:year-from-date fn:year-from-dateTime

Table 20. Equivalent ESQL and XPath functions (continued)

ESQL function	XPath function
EXTRACT MONTH FROM	fn:month-from-date fn:month-from-dateTime
EXTRACT DAY FROM	fn:day-from-date fn:day-from-dateTime
EXTRACT HOUR FROM	fn:hours-from-dateTime fn:hours-from-duration fn:hours-from-time
EXTRACT MINUTE FROM	fn:minutes-from-dateTime fn:minutes-from-duration fn:minutes-from-time
EXTRACT SECOND FROM	fn:seconds-from-dateTime fn:seconds-from-duration fn:seconds-from-time
EXTRACT DAYS FROM	fn:days-from-duration
EXTRACT MONTHS FROM	fn:months-from-duration
CURRENT_DATE	fn:current-date
CURRENT_TIME	fn:current-time
CURRENT_TIMESTAMP	fn:current-dateTime
LOCAL_TIMEZONE	fn:implicit-timezone
ABS (ABSVAL)	fn:abs
CEIL (CEILING)	fn:ceiling
FLOOR	fn:floor
LEFT	fn:substring
CONTAINS	fn:contains
ENDSWITH	fn:ends-with
LENGTH	fn:string-length
LOWER (LCASE)	fn:lower-case
REPLACE	fn:replace
RIGHT	fn:substring
STARTSWITH	fn:starts-with
SUBSTRING ... FROM	fn:substring
SUBSTRING ... BEFORE	fn:substring-before
SUBSTRING ... AFTER	fn:substring-after
SUBSTRING ... FROM ... FOR	fn:substring(fn:substring(...), \$for)
SUBSTRING ... BEFORE ... FOR	fn:substring(fn:substring-before(...), \$for)
SUBSTRING ... AFTER ... FOR	fn:substring(fn:substring-after(...), \$for)
TRANSLATE	fn:translate
UPPER (UCASE)	fn:upper-case

Table 20. Equivalent ESQL and XPath functions (continued)

ESQL function	XPath function
FIELDNAME	fn:local-name
FIELDNAMESPACE	fn:namespace-uri
CARDINALITY	fn:count
EXISTS	fn:exists

Custom Java

You can use the Custom Java transform to enter Java code in a message map.

In the Graphical Data Mapping editor, you can use the Custom Java transform to enter your own Java code. You can then use the transform properties to select a Java class, and a Java method. The Java class must be available in a Java project in your workspace. You can use the content assist in the **Values** column to help you assign the required elements of the source schema.

Consider the Graphical Data Mapping editor behavior when you use the Custom Java transform:

- When you include a Custom Java transform, an import is added to refer to the package qualified Java class, defining a prefix based on the class name.
- If you need to use custom Java only in condition or filter expressions, you can add Java imports to your Java class so that the class public static methods are available through content assist when you are composing an expression.
- The Java class that you provide to the map must have a static method that returns the appropriate type for the value of the output element, and takes parameters of the appropriate type for the wired inputs.

For example, the following Java method could be used in a Custom Java transform that had three input elements, of types a `xs:string`, `xs:decimal` and `xs:boolean` and the output element being a `xs:decimal`:

```
public static BigDecimal calSomething(String memType, BigDecimal stdCost, boolean flag) {
    BigDecimal actualCost = stdCost;
    if (flag & memType.startsWith("gold")) {
        BigDecimal discRate = new BigDecimal(0.9);
        actualCost = actualCost.multiply(discRate);
    }
    return actualCost;
}
```

Mappings between the Schema type, the Java type, and the IBM Integration Bus message assembly type

The following table shows the mappings between the Schema type, the Java type, and the IBM Integration Bus message assembly type:

Table 21. Mapping the Schema type, Java type, and message tree element type

Schema type	Java type	IBM Integration Bus message assembly tree element type
<code>xs:anyURI</code>	<code>java.lang.String</code>	CHARACTER
<code>xs:base64Binary</code>	<code>byte[]</code>	BLOB

Table 21. Mapping the Schema type, Java type, and message tree element type (continued)

Schema type	Java type	IBM Integration Bus message assembly tree element type
xs:boolean	boolean, java.lang.Boolean	BOOLEAN
xs:byte	byte, java.lang.Byte	INTEGER
xs:date	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:dateTime	javax.xml.datatype.XMLGregorianCalendar	TIMESTAMP
xs:dayTimeDuration	javax.xml.datatype.Duration	INTERVAL
xs:decimal	java.math.BigDecimal	DECIMAL
xs:double	double, java.lang.Double	FLOAT
xs:duration	javax.xml.datatype.Duration	INTERVAL
xs:float	float, java.lang.Float	FLOAT
xs:gDay	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:gMonth	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:gMonthDay	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:gYear	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:gYearMonth	javax.xml.datatype.XMLGregorianCalendar	DATE
xs:hexBinary	byte[]	BLOB
xs:int	int, java.lang.Integer	INTEGER
xs:integer	java.math.BigInteger	DECIMAL
xs:long	long, java.lang.Long	INTEGER
xs:negativeInteger	java.math.BigInteger	DECIMAL
xs:nonNegativeInteger	java.math.BigInteger	DECIMAL
xs:nonPositiveInteger	java.math.BigInteger	DECIMAL
xs:normalizedString	java.lang.String	CHARACTER
xs:positiveInteger	java.math.BigInteger	DECIMAL
xs:short	short, java.lang.Short	INTEGER
xs:string	java.lang.String	CHARACTER
xs:time	javax.xml.datatype.XMLGregorianCalendar	TIME
xs:unsignedByte	Short	INTEGER
xs:unsignedInt	Long	INTEGER
xs:unsignedLong	java.math.BigInteger	DECIMAL
xs:unsignedShort	Int	INTEGER
xs:yearMonthDuration	javax.xml.datatype.Duration	INTERVAL

Processing arrays and complex types

You can create Custom Java transforms to process input and outputs that are arrays or complex types. These cannot be converted from a Schema type to a defined Java type shown in the previous table.

In IBM Integration Bus, your Java class must use the **MbElement** class to reference the input or output element. For information about using Java and the MbElement class, see Chapter 17, “Using Java API classes for Custom Java mapping transforms,” on page 155.

Custom XPath

You can use the **Custom XPath** transform to provide a data value for a simple target element, or values for a repeating simple target element by using an XPath expression.

Overview

You define the XPath expression in the **General** tab of the Properties view.

You can use context assist by pressing **Ctrl+Space** while constructing the XPath expression. For more information, see “Using content assist (Mapping syntax)” on page 93.

When to use the Custom XPath transform

You can extend built-in transformation functions by using custom XPath expressions. For more information about XPath, see XPath tutorial or W3C XML Path Language (XPath) 2.0.

You can use the **Custom Xpath** transform to perform any of the following mappings:

- Implement an arithmetic operation, such as add, subtract, or multiply. For more information, see “Choosing a transform to perform an arithmetic operation” on page 31.
- Implement complex XPath expressions.
- To obtain the value of an integration node property. For more information, see “Accessing integration node properties from a Mapping node” on page 122.
- To access user-defined properties. For more information, see “Accessing user-defined properties from a Mapping node” on page 122.

Inputs

You can connect any input element to a **Custom XPath** transform.

Outputs

You can use a **Custom XPath** transform to set the data value of a simple target element, or to set the values for a repeating simple target element.

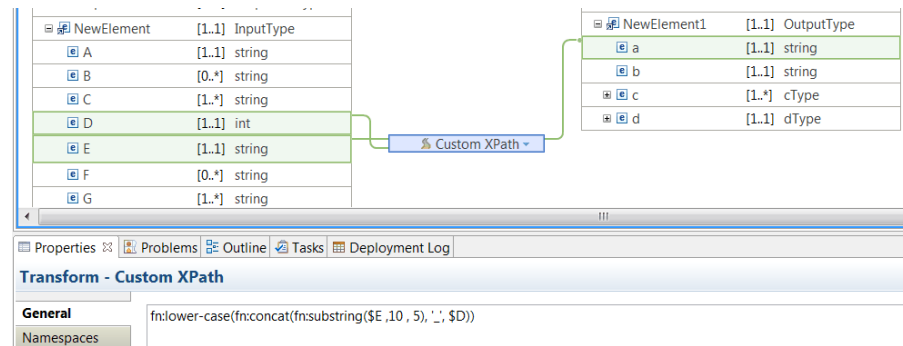
Note: The **Custom XPath** transform returns values, not elements. For this reason, you cannot populate a complex structure from a **Custom XPath** transform.

Example

The following figure shows a **Custom XPath** transform that performs the following calculation:

1. Uses the **fn:substring** XPath function to obtain part of the information provided in element **E**.

2. Uses the **fn:concat** XPath function to concatenate the result of the **fn:substring** function, with the delimiter `_` and the element **D**.
3. Uses the **fn:lower-case** XPath function to put in lower case the result of the **fn:concat** function.



If you input the following message:

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>A1</A>
  <B>B1</B>
  <C>Field_1</C>
  <D>1000</D>
  <E>CUSTOMER_AREA1</E>
  <F>Optional1</F>
  <G>Optional1</G>
</NewElement>
```

you get the following output value:

```
<NewElement1>
  <a>area1_1000</a>
</NewElement1>
```

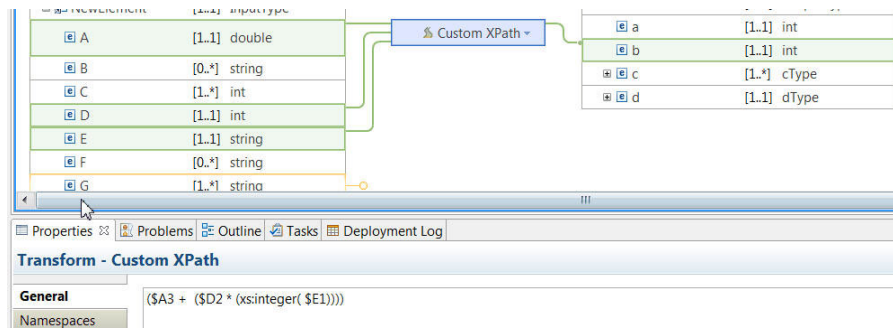
Example: How to perform an arithmetic operation

This example shows how to perform an arithmetic operation by using a **Custom XPath** transform.

The transform has 3 inputs, that are used as part of the calculation that determines the value of the output element **b**.

Note: You can only use as arguments of an arithmetic operation non-repeating simple type input elements.

The operations consists on adding the value of the input element **A** to the value of calculating **D** times **E**. The value of **E** is of type string. The value of **E** is cast to an integer.



When you transform the following message, the value of **b** is **120012**.

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>10</A>
  <C>2</C>
  <C>10</C>
  <D>1000</D>
  <E>120</E>
  <F>Optional1</F>
  <G>Optional1</G>
  <H>
    <H1>A</H1>
    <H2>10</H2>
    <H3>20</H3>
  </H>
</NewElement>
```

Database Routine

You can use the Database Routine transform to call a stored procedure in a database, setting the parameter values by mapping input elements.

The definition of the stored procedure used during the creation of your Database Routine transform is specified in a database definition file (.dbm file) that you select from the Data Design project.

You create the information that is provided in the database definition file by running discovery against your database server. For more information, see “Creating a database definition (.dbm file) by using the New Database Definition File wizard” on page 118. The information available varies depending on the vendor of your database server. Some information, such as the content of any Result Sets returned, is not provided. If you want to map data from any columns in these Result sets, you must define them in the Database Routine setup.

For more information, see “Calling a stored procedure from a map” on page 168.

Limitations

- Only IBM DB2 stored procedures are supported database calls from a graphical data map.
- Stored procedure parameters of type **Array** are not supported in database calls from a graphical data map.

Delete

The **Delete** transform deletes one or more rows of data from a database table.

Use the **Delete** transform to delete one or more rows of data that match a configured **Where** clause from a database table. For more information, see “Deleting data from a table” on page 165.

The database table structure needed to create your Delete transform is specified in a database definition file (.dbm file) that you select from the Data Design project used to create the message map. The database definition can be discovered by connecting to a data server from the tooling.

When your message map has been deployed to an integration server, and your message map is run, the database server that is used by the IBM Integration Bus runtime component to process your Delete transform is specified by the JDBCProvider configurable service. The JDBCProvider service name must have the same name as the database name that is specified in your Delete transform during development. For more information, see Enabling JDBC connections to the databases.

Failure

Use the **Failure** transform to handle exceptions that might be raised by the configured database server when your message map runs SQL statements to implement the action of a database transform.

If you want the message map to handle exceptions that are returned from the database server when the SQL operation is run, instead of such exceptions stopping the message map and being reported, you can add a **Failure** transform to the transform group.

The **Failure** transform is an optional transform that can be added or removed as required.

The **Failure** transform does not perform any transformation. You must transform the input and output elements within the nested map.

Handling database warnings

When you create a database transform, select **Treat warnings as exceptions** if you want the database warnings to be treated as errors.

If this option is selected, the first SQL operation that results in a warning from the selected database raises an exception.

Run time behavior

The way that database exceptions are handled is determined by the configuration of the corresponding **Failure** transform in a message map:

1. If the **Failure** transform is present in the message map, and is connected to one or more output objects, then the exception is caught and processed by the **Failure** transform.
2. If the **Failure** transform is present in the message map, but is not connected, then the exception is caught by the **Failure** transform, and is ignored.
3. If the **Failure** transform has been deleted from the message map, then the exception is handled by the Mapping node in your message flow, and is handled in the same way as other message flow exceptions.

- If you want the message map to stop running when the database transform receives an SQL exception, remove the **Failure** transform. For more information, see “Handling database exceptions in a graphical data map” on page 171.

For Each

You can use the **For Each** transform to iterate over one input array element, which can be either a simple type or a complex type repeating structure, and set the value of an output element.

Overview

The **For Each** transform contains a nested map. The elements in the nested map must be mapped, otherwise no action is performed when the transform runs.

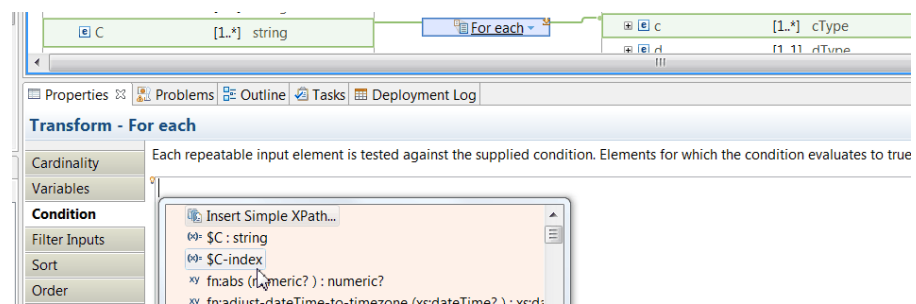
You can configure the **For Each** transform to execute once when the input array is empty or there are input elements that match your input selection criteria. You must set the **Allow Empty input** condition in the **Filter Inputs** property tab.

Inputs

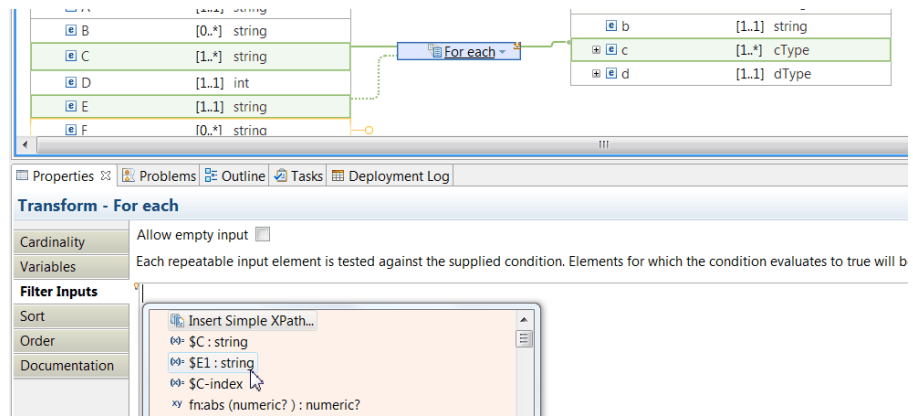
You can define multiple inputs to a **For Each** transform.

- The **For Each** transform can only have one **primary** input connection, and the input must be repeatable.
 - The Graphical Data Mapping editor counts the indexes from **1** to **N** for each processed input.
 - When the input array element is empty or no inputs match a provided filter condition, the Graphical Data Mapping editor sets the index variable to **0**.
 - The Graphical Data Mapping editor provides a variable that contains the index value of each iteration of the **For Each** transform. The name is as follows: *\$VarName-index*, where *VarName* is the name of the repeating element.

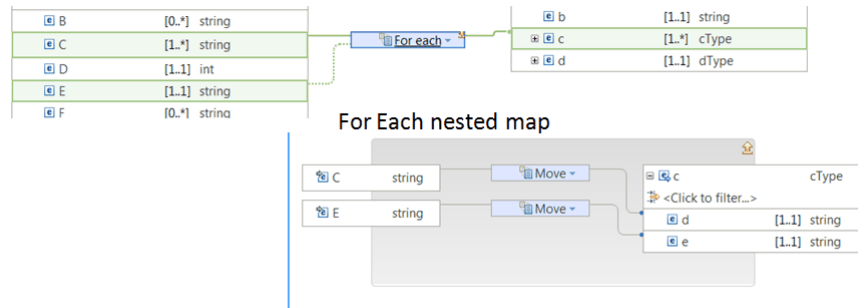
Note: Always use content assist to get the element name assigned by the Graphical Data Mapping editor to the repeating element.



- Additional connections to the **For Each** transform must be of type **Supplement**. You can use these inputs in any of the following ways:
 - You might want to create a conditional expression based on the value of the input. You can use it to define the condition that determines whether the **For Each** transform is applied. You can use it to determine which indexes to apply as part of the transformation.



- You might want to pass an additional non-repeating element into the nested transform. This input is available in the mapping of each iteration executed by the **For Each** transform.



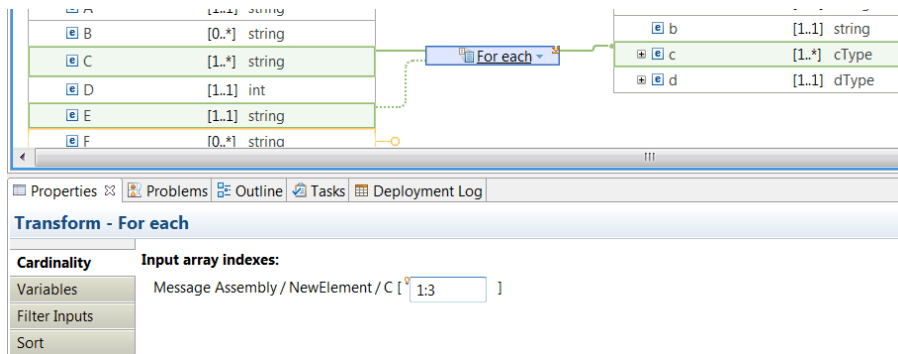
- You can configure the **Cardinality** property page on the **For Each** transform to indicate which index of an input array to iterate over.
- You can configure the **Filter Inputs** property page to specify the matching criteria for filtering input repeating elements.

Cardinality

The **Cardinality** property determines the inputs participating in the **For Each** operation.

The first index element is 1.

You configure the **Cardinality** tab in the Properties view to specify the indexes that will be processed by the transform. For more information, see “Selecting the indexes of input array elements” on page 28.



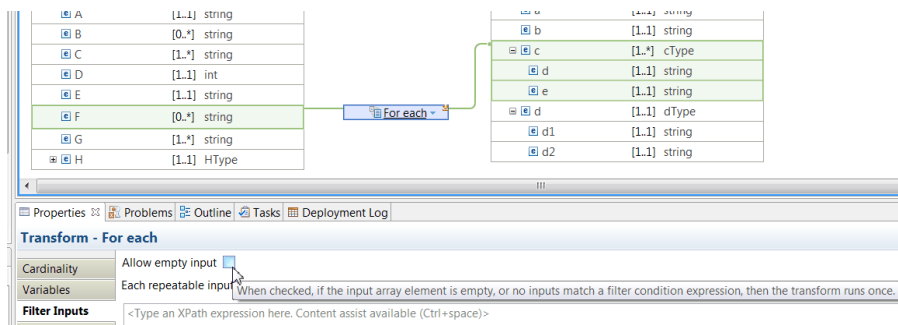
Filter Inputs

Configure the **Filter Inputs** property tab to specify an XPath expression that determines which instances of the repeating input will be processed in the nested mapping. Each element of the repeatable input will be tested against the condition. The transform will run for those elements that satisfy the condition.

You can use context assist by pressing **Ctrl+Space** while constructing the XPath expression. For more information, see “Using content assist (Mapping syntax)” on page 93.

Note: To obtain the exact name of the variables associated with the inputs, use content assist.

Enable **Allow Empty input** when the input array element is empty or no inputs match a provided filter condition, so that the transform is still entered exactly once. In this case, the primary input in the nested transform will be missing, and the index variable will be zero.



Outputs

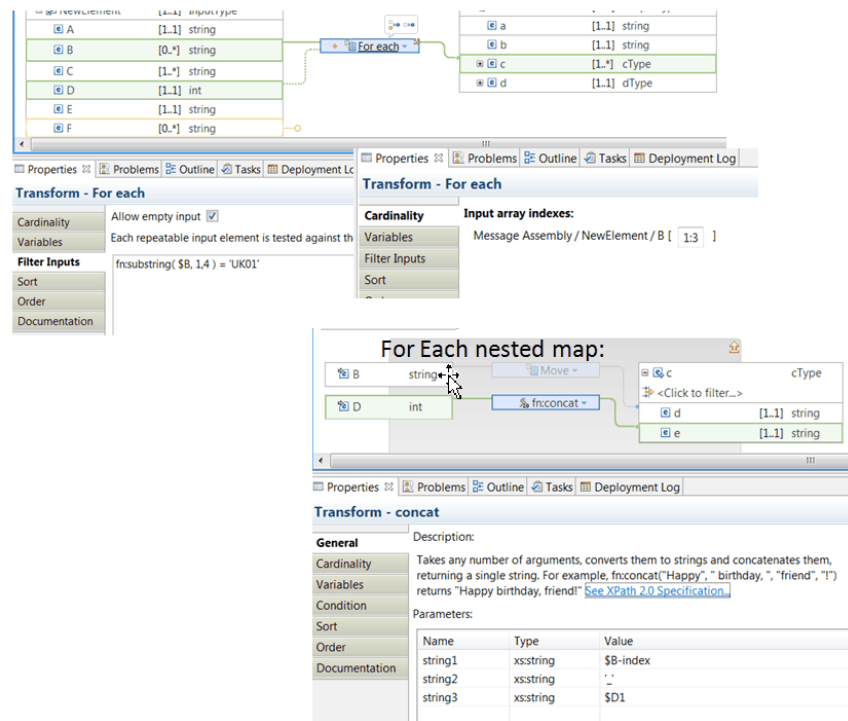
The output element of a **For Each** transform can be a simple element, or a complex element, that can be a repeating element or a non-repeating element.

The output array size is equal to the input array size, minus any elements that are filtered out from the cardinality property page.

Example

In this example, the **For Each** transform only runs for the first three elements of the repeating element. The rest of the elements in the array are not considered. If the first four characters of the element B in each element of the array start with

UK01, then the transformation inside the nested map is executed. Inside the nested map, the `fn:concat` transform calculates the value of element `e` based on the input element index and the input element `D`.



When you run the following message through the **For Each** transform,

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>A1</A>
  <C>Field_1</C>
  <C>Field_2</C>
  <C>Field_3</C>
  <D>1000</D>
  <E>CUSTOMER_AREA1</E>
</NewElement>
```

you get the following output:

```
<NewElement1>
  <c>
    <d/>
    <e>0_1000</e>
  </c>
</NewElement1>
```

Note: If the repeating element is empty, then the nested map is executed once because the option **Allow Empty input** is selected.

When you run the following message through the **For Each** transform,

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>A1</A>
  <B>UK011234567</B>
  <B>B2</B>
  <B>UK019999999</B>
  <B>UK01xxxxxxx</B>
```

```
<C>Field_1</C>
<C>Field_2</C>
<C>Field_3</C>
<D>1000</D>
<E>CUSTOMER_AREA1</E>
</NewElement>
```

you get the following output:

```
<NewElement1>
  <c>
    <d>UK011234567</d>
    <e>1_1000</e>
  </c>
  <c>
    <d>UK019999999</d>
    <e>3_1000</e>
  </c>
</NewElement1>
```

Group

The **Group** transform produces a set of nested output arrays by collating the input array.

Overview

The **Group** transform takes the following types of inputs and outputs:

- The input must be a single repeating complex type, such as an array or a list.
- The output must be a repeating complex type, that is formed of a nested simple type and a complex type, such as a nested array or lists.

The **General properties** page of the **Group** transform provides the **Group by** selection, from which you can select a simple type element. The value of the simple type element is used to collate occurrences of the repeating input.

The **Group** transform provides a nested transform for mapping each collated input array occurrence to the output nested array structure.

The nested transforms are performed sequentially for each collated group of input elements that have matching values in the element that you have selected to group by in the **General Properties**.

For example, you might use the **Group** transform to map a single list of company employee records into a nested list of employees by department. In this case, the input would be the repeating array input of **Employee** complex types. You would then select the **Department** element of the **Employee** input complex type for the **Group by** property. The output would be the repeating array input of **Department** complex types. The **Department** complex type would contain a **DepartmentName** simple type element, and a repeating nested array element of **DepartmentEmployees** complex types. The nested mapping of the **Group** transform would allow you to map the values of the "Employee" complex type to a "Department" complex type.

If, Else if, and Else

The **If, Else if, and Else** transform enables you to control the flow of the mapping by setting conditions.

The **If, Else if and Else** transform operates as a group of conditional transforms. The condition is applied to the input element of the conditional transform. If the condition is satisfied, the transform that is nested within the conditional transform is run.

- For each conditional transform in the group, enter a condition in the **Conditions** tab in the Properties view. The condition must be in the form of an XPath expression.
- To change the order in which the conditions are evaluated, select the conditional group and click the **Order** tab and use the up and down arrows.
- Double-click the conditional transform (for example, **If**) to create the mapping that will execute for the condition.

The elements in the nested map must be mapped in order for the transform to run.

Insert

The **Insert** transform inserts a row to a database table.

Use the **Insert** transform to insert a row into a database table. For more information, see “Inserting data into a table” on page 162.

The database server that is used during the creation of your **Insert** transform is specified in a database definition file (.dbm file) that you select from the Data Design project used to create the message map.

When your message map has been deployed to an integration server, and your message map is run, the database server that is used by the IBM Integration Bus runtime component to process your **Insert** transform is specified by the JDBCProvider configurable service. The JDBCProvider service name must have the same name as the database name that is specified in your **Insert** transform during development. For more information, see Enabling JDBC connections to the databases.

Join

You can use a **Join** transform to join elements from two or more inputs into an output element.

Overview

You provide a **Join expression** to determine which data to transform.

You define the mapping from the joined instances of the input arrays to an output target in the nested map. For more information, see Chapter 6, “Using nested maps,” on page 37.

The **Join** transform implements an inner-join.

If you want to implement an outer-join, use the **For Each** transform with the option **Allow Empty input** enabled.

Inputs

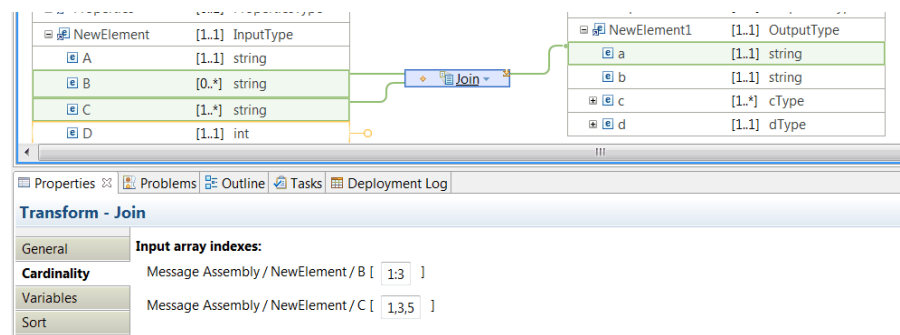
The inputs to a **Join** transform can be repeating simple elements or complex type elements, which can be merged using nested transforms to create an output.

- You can configure the **Cardinality** property page of each input to indicate which index of an input array to iterate over.
- You can configure the **Order** property page to define the order of iteration over the input elements.
- You can define a **Join expression** to specify the matching criteria for joining or filtering input repeating elements. The join expressions determines the size of the output element.

Cardinality

The **Cardinality** property determines the inputs participating in the join operation.

You configure the **Cardinality** property tab of a **Join** transform to define which indexes are used from each input array in the join operation. For more information, see “Selecting the indexes of input array elements” on page 28.



Order of the inputs

By default, the order of the inputs to the **Join** transform is the order in which you wire the inputs.

You can modify the order by reordering the inputs in the **Order** tab of the transform properties.

Output

The output element can be a simple element or a complex type element that can be repeating or not repeating.

When your output element is a repeating element, the **Join expression** determines the size of the output element.

For example, if you have a repeating input element of size M, and a repeating input element of size N to a **Join** transform, the output element size is calculated as follows:

- If you click the option **Create Join expression based on index**, the size of the output element is the minimum value of M and N.
- If you do not define a **Join expression** or you do not select the option **Create Join expression based on index**, the size of the output element is $M \times N$.
- If you define a **Join expression**, the size of the output element is determined by the expression.

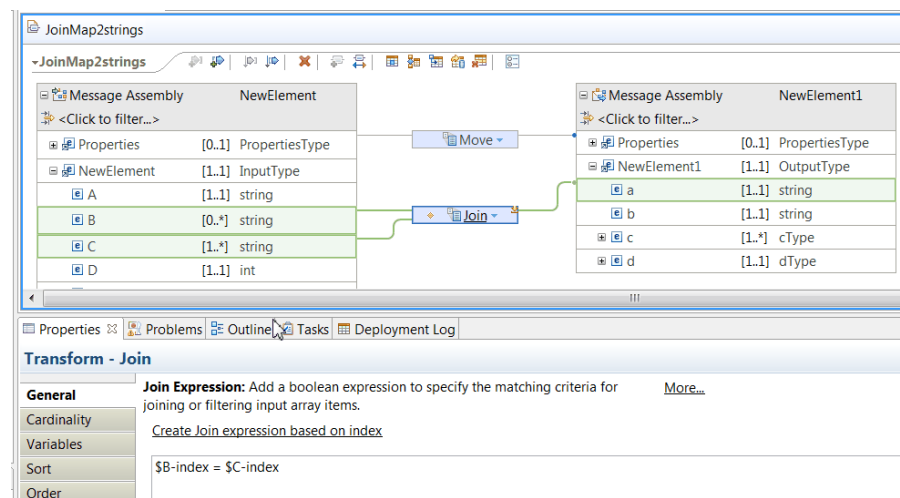
Example 1: To join the first element of the first array with the first element of the second array, you set a join condition that matches based on index. You click **Create Join expression based on index**.

Example 2: If you do not specify a **Join expression**, the join matches the first element of the first array with all elements of the second array, and then it matches the second element of the first array with all elements of the second array, and so on.

Join expression

You can define a **Join expression** to specify the matching criteria for joining or filtering input array elements. This expression is an XPath expression.

- You can use content assist to create the XPath expression. For more information, see “Using content assist (Mapping syntax)” on page 93.
- You can select the option **Create Join expression based on index**. This option requires two repeatable inputs to create the Join expression.



Local map

A **local map** is a transform that provides a hierarchical view of element transforms in a message map.

You can use local maps to break up a large map into nested groups of mapping elements and process the complex elements of the whole data object.

Local maps are a partial view of a larger map, rather than separate files.

A local map has only one element as input (either a simple type or a complex type), which can contain nested elements. The output can be either a single element or an array element, but it must be a complex type.

The local map does not transform data; you must specify transforms for the input and output elements in the nested map.

Move

You can use the **Move** transform to copy data from one input element to one output element.

Overview

You can define the **Move** transform between single simple elements or between complex type elements:

You can use the Move transform between complex type input and output elements only if the input and output have the same type, or if the type of the input is derived from the type of output; for example, if the input element's type is `USAddress` and the output type is `Address`.

Normalize

The **Normalize** transform normalizes the input string by removing whitespace such as spaces, tabs, and returns, and moves the resulting normalized string to the output element.

For example, it can be used to remove multiple occurrences of white space characters before a comparison of data is done.

The **Normalize** transform is functionally equivalent to the XPath 2.0 `fn:normalize-space()` function. For more information about XPath functions, see the online document *W3C XML Path Language (XPath) 2.0*.

Return

Use the **Return** transform to report the number of rows that are modified by a database transform.

The **Return** transform is called if the operation that is specified in the database transform is successful.

Use the **Return** transform in your message map to specify a nested mapping that is called if a database transform was completed successfully.

The **Return** transform provides an in-built input that provides an integer value of the number of rows that are modified by the related database transform:

Database transform	Name of input provided by Return transform	Description
Insert	NumberOfRowsInserted	The Return transform is called each time that an insert operation is successful, and reports the number of rows inserted as 1.
Update	NumberOfRowsUpdated	Number of rows updated depends on the WHERE clause
Delete	NumberOfRowsDeleted	Number of rows deleted depends on the WHERE clause

For more information, about database transforms, see Chapter 19, "Mapping database content," on page 159.

Select

Use the **Select** transform to retrieve data from rows in a database table, so that the data can be used in a message map.

For more information, see “Selecting data from a table” on page 159.

When you design your message map in the IBM Integration Studio, the database server that is used during the creation of your **Select** transform is specified in the database definition file (.dbm file) that you selected from the Data Design project used to create the message map.

When your message map has been deployed to an integration server, and your message map is run, the database server that is used by the IBM Integration Bus runtime component to process your **Select** transform is specified by the JDBCProvider configurable service. The JDBCProvider service name must be the same name as the database name that is specified in your **Select** transform during development. For more information, see Enabling JDBC connections to the databases.

Submap

A **submap** enables you to use the same piece of mapping function in multiple graphical data maps.

A **submap** references another map. It calls or invokes a map from the same file or another map file, which can be stored in a library, an application, an integration service, or an Integration project.

When using submaps, consider the following points:

- A **submap** can provide callable mapping between global elements or global types from a message model.
- A **submap** cannot be used for local anonymous complex types. These must be mapped within the main map, for example, by a local map.
- Submaps must be placed in the same application, library, integration service, or project. Alternatively, submaps can be placed in a project or library that is visible to the main map(s) that they are called from.

For more information, see “Creating a submap” on page 50.

Substring

In the Graphical Data Mapping editor, you can use the **Substring** transform to set the value of an output element to a substring of the original input value. You use this transform to extract a subset of characters that are separated by a delimiter based on a position that you indicate with an index.

Overview

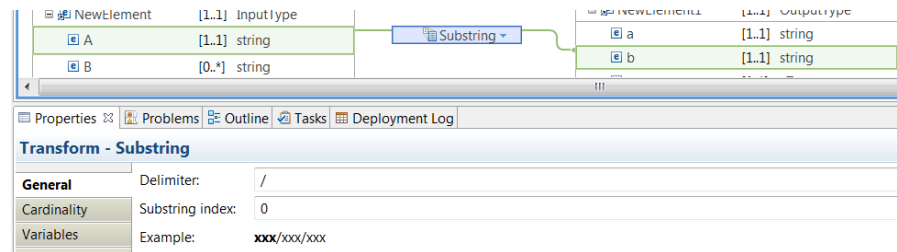
The **Substring** transform uses a delimiter and 0-based index to determine what text to extract from the incoming source string.

Based on the specified delimiter, the source string is divided into sections. The index is used to identify which section of the divided string you want to use.

For example, you can pass the following input string: '123/124/125/126'. You can configure the **Delimiter** property with the value '/'.

- If you set the **Substring index** property to 0, the transform returns the value '123'.
- If you set the **Substring index** property to 1, the transform returns the value '124'.
- If you set the **Substring index** property to 3, the transform returns the value '126'.

By default, the **Substring index** has a value of 0, that indicates that the first section will be used.

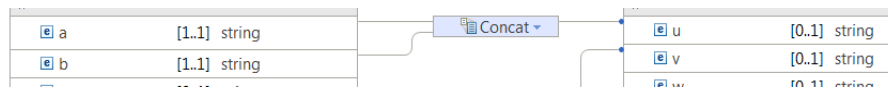


Inputs to the Substring transform

The **Substring** transform takes multiple input elements.

You wire one simple type input to the **Substring** transform as primary connection. This input contains the input value from which you need to extract part of it.

You can wire more inputs as supplementary connections. These other inputs are used to define the conditional expression that determines if a transform is applied.



Define when the transform is applied at run time

You can use any of the input elements to the **Substring** transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

Update

The **Update** transform modifies one or more rows of data in a database table.

Use the **Update** transform to modify one or more rows of data that match a configured **Where** clause from a database table. Optionally, if no rows of data match your configured **Where** clause, the **Update** transform can insert a row of data. For more information, see “Updating data in a table” on page 163.

The **Update** transform performs a single update SQL operation on the configured database server, so the inputs that you connect to your **Update** transform must provide a single set of data values. If you connect a repeating element to your

Update transform, the Graphical Data Mapping editor moves the **Update** transform into a nested **For Each** transform.

The database server that is used during the creation of your **Update** transform is specified in a database definition file (.dbm file) that you select from the Data Design project used to create the map.

When your message map has been deployed to an integration server, and your message map is run, the database server that is used by the IBM Integration Bus runtime component to process your **Update** transform is specified by a JDBCProvider configurable service. The JDBCProvider service must have the same name as the database that is specified in your **Update** transform during development. For more information, see *Enabling JDBC connections to the databases*.

Built-in XPath transforms

In the Graphical Data Mapping editor, you can use built-in XPath functions to transform data.

Overview

The Graphical Data Mapping editor supports XPath functions that allow you to manipulate graphically string values, numeric values, date and time comparison, and more.

The XPath functions are grouped in the following categories:

- String functions
- Boolean functions
- Math functions
- Date and time functions
- QName functions
- Node functions
- List functions
- Diagnostic functions

For more information about XPath, see *XPath tutorial* or *W3C XML Path Language (XPath) 2.0*.

All XPath 2.0 functions are supported in the form `fn:<function_name>`.

For more information on the following XPath functions, see the following topics:

- “fn:concat” on page 231
- “fn:string-join” on page 234

When you need to define complex XPath expressions, use the **Custom XPath** transform. For more information, see “Custom XPath” on page 214.

Inputs versus arguments

When you use an XPath transform, you must differentiate between inputs to the transform and arguments required to run the XPath function represented by the XPath transform.

Arguments are the data elements required in the calculation of an XPath function.

An argument can be a literal expression, a constant, an input element, a custom XPath expression, or a combination of multiple input elements.

You can have any number of inputs to an XPath transform. You use these inputs to define the arguments of the XPath function.

In the following figure, the XPath transform has two inputs. Each input is used as an argument of the **fn:concat** transform:

Name	Type	Value
string1	xs:string	\$A1
string2	xs:string	\$E1

Define a conditional expression

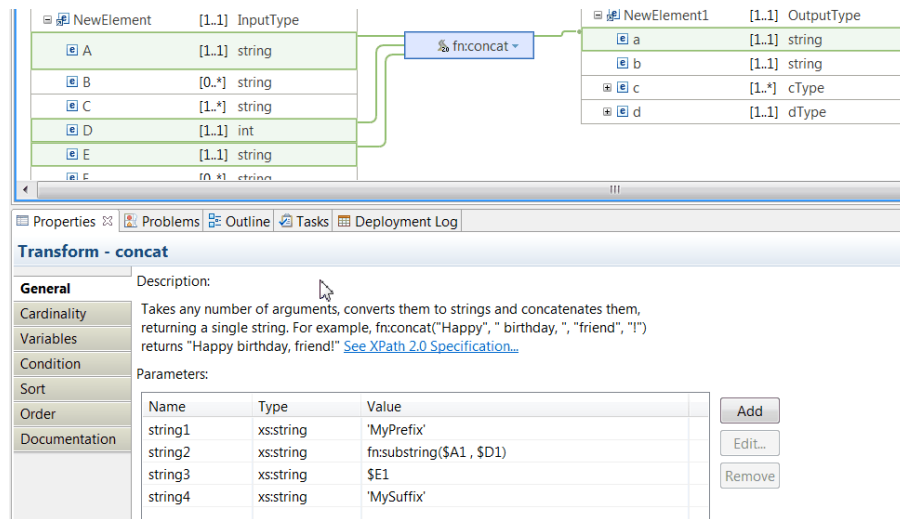
You can use any of the input elements to an XPath transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

Example

This example shows how to use the **fn:concat** transform to concatenate multiple input elements and set the value of a string element by using the **fn:concat** function.

The arguments to the XPath function include a prefix, two elements, and one suffix. One of the arguments is defined by an XPath function that requires data from two inputs. One of the arguments is set with data from an input. The prefix and the suffix are literals.



When you run the following message and transform the data with the **fn:concat** transform:

```
<NewElement>
  <A>My FieldA</A>
  <B>B1</B>
  <C>Field_1</C>
  <D>4</D>
  <E>FIELD_E</E>
</NewElement>
```

You obtain the following result:

```
<NewElement1>
  <a>MyPrefixFieldAFIELD_EMySuffix</a>
</NewElement1>
```

Troubleshooting

BIP3946E:

BIP3946E: The map script generation for QName {1} has failed, with the following details: {2}

You get **BIP3946E** when you try to deploy a map that contains an invalid XPath expression. Check the description provided by {2} to find out which expression is not a valid.

fn:concat

In the Graphical Data Mapping editor, you can use the **fn:concat** transform to create a string output element that is the result of concatenating simple input elements.

Overview

The XPath 2.0 function **fn:concat**(arg1, arg2, ...,) takes two or more arguments, converts them to their string representation, and concatenates them, returning a single string.

The **fn:concat** transform is the representation of the **fn:concat** XPath function in the Graphical Data Mapping editor.

You can have any number of inputs to the **fn:concat** transform. These inputs are used to define the arguments of the **fn:concat** function.

For more information about XPath, see XPath tutorial or W3C XML Path Language (XPath) 2.0.

When to use the fn:concat transform

You can use **fn:concat** to concatenate input elements.

You can configure the arguments to the **fn:concat** transform to represent a prefix, a suffix, and delimiters.

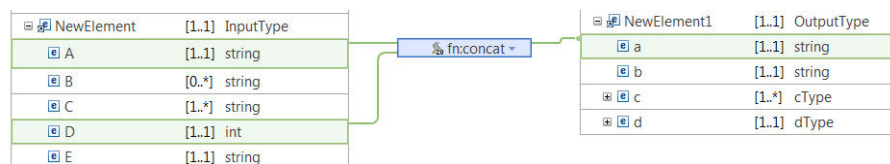
- You can use the first argument as the prefix.
- You can use the last argument as the suffix.
- You can insert a delimiter as an argument between any two arguments you want to delimit.

Inputs to the transform

You can connect simple elements to the **fn:concat** transform.

Simple input elements that are not of type **xs:string** will be cast to **xs:string**.

The following figure shows two inputs connected to the **fn:concat** transform:



Note: The Graphical Data Mapping editor does not display an error or a warning on a **fn:concat** transform when the input to the transform is not a single type element, or is repeatable.

- If the input XML has no more than one instance of the repeatable input, the **fn:concat** transform produces the expected result at run time.
- If the input XML has more than one instance of the repeatable input, the **fn:concat** transform results in a run time exception.

Arguments of the XPath function

The **fn:concat** function has two or more non-repeating arguments.

Note: A run time failure occurs complaining that a sequence is not valid if you do not ensure that each argument is non-repeating.

You define the number and the order of the arguments to the **fn:concat** function in the **General** tab of the **fn:concat** transform properties view.

You can use the **Add** button on **General** tab to create additional arguments:

- You can add a literal value entered in single quotes.
- You can add a Custom XPath expression.

In the following figure, the **fn:concat** transform has two inputs. Each input is used as an argument of the **fn:concat** function:

The screenshot shows a graphical data mapping editor. On the left, a source table 'NewElement' has columns A (string), B (string), C (string), D (int), E (string), and F (string). On the right, a target table 'NewElement1' has columns a (string), b (string), c (cType), and d (dType). A central 'fn:concat' transform has two input lines connecting to elements A and E. Below the diagram is a 'Transform - concat' panel with a description and a parameters table.

Name	Type	Value
string1	xs:string	\$A1
string2	xs:string	\$E1

In the following figure, the **fn:concat** transform has three inputs, whilst the **fn:concat** function only requires two arguments. Two of the inputs are used to define one of the arguments of the **fn:concat** transform:

The screenshot shows a graphical data mapping editor. On the left, a source table 'NewElement' has columns A (string), B (string), C (string), D (int), E (string), and F (string). On the right, a target table 'NewElement1' has columns a (string), b (string), c (cType), and d (dType). A central 'fn:concat' transform has three input lines connecting to elements A, E, and F. Below the diagram is a 'Transform - concat' panel with a description and a parameters table.

Name	Type	Value
string1	xs:string	fn:substring(\$A,\$D)
string2	xs:string	\$E

Define a conditional expression

You can use any of the input elements to the **fn:concat** transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

Example

This example shows how to use the **fn:concat** transform to concatenate multiple input elements and set the value of a string element by using the **fn:concat** function.

The arguments to the XPath function include a prefix, two elements, and one suffix. One of the arguments is set by an XPath function base on calculations from data from two inputs. One of the arguments is set with data from an input. The prefix and the suffix are literals.

The screenshot shows the Graphical Data Mapping editor. At the top, there are two input tables. The first table, 'NewElement', has columns for element name and type: A (string), B (string), C (string), D (int), E (string), and F (string). The second table, 'NewElement1', has columns for element name and type: a (string), b (string), c (cType), and d (dType). A central 'fn:concat' transform is connected to these inputs. Below the diagram is the 'Transform - concat' configuration panel. It includes a description: 'Takes any number of arguments, converts them to strings and concatenates them, returning a single string. For example, fn:concat("Happy", " birthday, ", "friend", "!") returns "Happy birthday, friend!" See XPath 2.0 Specification...'. The parameters table is as follows:

Name	Type	Value
string1	xs:string	'MyPrefix'
string2	xs:string	fn:exists(\$A1) and fn:exists(\$D1)
string3	xs:string	\$E1
string4	xs:string	'MySuffix'

When you run the following message and transform the data with the **fn:concat** transform:

```
<NewElement>
  <A>My FieldA</A>
  <B>B1</B>
  <C>Field_1</C>
  <D>4</D>
  <E>FIELD_E</E>
</NewElement>
```

You obtain the following result:

```
<NewElement1>
  <a>MyPrefixtrueFIELD_EMySuffix</a>
</NewElement1>
```

fn:string-join

In the Graphical Data Mapping editor, you can use the **fn:string-join** transform to create an output element that is the result of concatenating a sequence of input elements with a delimiter as optional.

Overview

The XPath 2.0 function `fn:string-join(arg1, arg2)` takes two arguments, converts them to their string representation, and concatenates them, returning a single string.

The **fn:string-join** transform is the representation of the **fn:string-join** XPath function in the Graphical Data Mapping editor.

You can have any number of inputs to the **fn:string-join** transform. These inputs can be used to define the arguments of the **fn:string-join** function.

Inputs to the transform

You can connect one or more inputs to the **fn:string-join** transform. These inputs are used to define the arguments of the **fn:string-join** function.

- One of the primary inputs must be a repeatable simple element. This input is used to define the first argument of the **fn:string-join** function. It defines the sequence of elements that you want to concatenate.

Note: When the type of the repeating elements is different from **xs:string**, the Graphical Data Mapping editor will cast the element to a **xs:string** type.

- One of the primary inputs can be a simple type element. This input is used to define the second argument of the **fn:string-join** function. It defines the delimiter used between the elements that you want to concatenate.

Note: If the simple type is not of type **xs:string**, the transform will fail at run time with **BIP3945E**. Cast the element to a string if you want to use a non-string input element as your delimiter. For more information, see “Cast type (xs:type)” on page 199.

The screenshot shows the Graphical Data Mapping editor interface. At the top, there are two input tables. The first table, 'NewElement', has columns for element name, cardinality, and type. It contains elements A (string), B (string), C (string), and D (int). The second table, 'NewElement1', has columns for element name, cardinality, and type. It contains elements a (string), b (string), c (cType), and d (dType). A central box labeled 'fn:string-join' is connected to elements C and D. Below the tables is a 'Transform - string-join' properties view. It includes a 'General' tab with a description: 'Takes an input sequence of strings and a separator and returns a string created by concatenating the members of the input sequence using the separator as a delimiter. See XPath 2.0 Specification...'. It also has a 'Parameters' table with columns for Name, Type, and Value. The parameters are: strings (xs:string[]), separator (xs:string), and Value (\$C1, xs:string(\$D1)). There are 'Add' and 'Edit...' buttons next to the parameters table.

Name	Type	Value
strings	xs:string[]	\$C1
separator	xs:string	xs:string(\$D1)

- More inputs can be connected to the **fn:string-join** transform. You can use these inputs as additional information to the transform. For example, you can use these inputs as arguments of the XPath expression that you can define in the **Condition** tab to determine whether the transform is applied at run time.

Arguments of the XPath function

You define the arguments to the **fn:string-join** function in the **General** tab of the **fn:string-join** transform properties view.

- The first argument is a sequence of elements.
- The second argument is a delimiter. The delimiter is optional. If the separator defined for the **fn:string-join** function is a zero-length string, no delimiter is used.

In the following figure, the **fn:string-join** transform has one input. The input is used to define the first argument of the **fn:string-join** function. No delimiter is configured.

Transform - string-join

General Description: Takes an input sequence of strings and a separator and returns a string created by concatenating the members of the input sequence using the separator as a delimiter. [See XPath 2.0 Specification...](#)

Cardinality: [1..1] string

Variables: strings: xs:string [] \$C

Condition: separator: xs:string ""

Parameters:

Name	Type	Value
strings	xs:string []	\$C
separator	xs:string	""

The following figure shows the `fn:string-join` transform with one repeating element as input, and a separator. To add the delimiter, you must select the **General** tab in the Properties view, click **Edit**, and then enter the value.

Transform - string-join

General Description: Takes an input sequence of strings and a separator and returns a string created by concatenating the members of the input sequence using the separator as a delimiter. [See XPath 2.0 Specification...](#)

Cardinality: [1..1] string

Variables: strings: xs:string [] \$C

Condition: separator: xs:string 'xxxxx'

Parameters:

Name	Type	Value
strings	xs:string []	\$C
separator	xs:string	'xxxxx'

Cardinality

The **Cardinality** property determines the elements (also known as indexes) in the repeating input element that are processed by the `fn:string-join` transform.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.

Transform - string-join

General **Input array indexes:**

Cardinality Message Assembly / NewElement / C [1,5,7]

Control the instances to be concatenated

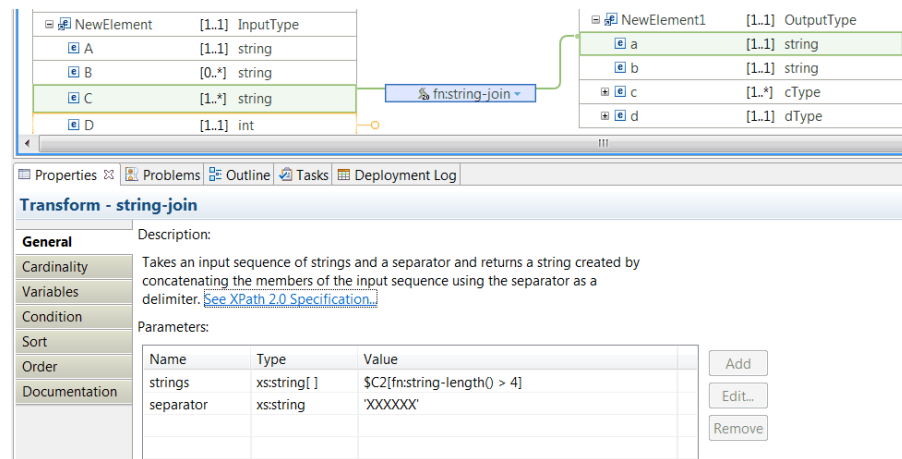
When the instances that you need to concatenate need to be calculated at run time based on a condition, you can filter dynamically the input indexes by defining a

custom XPath conditional expression for the first argument named **strings** in the **General** tab. The expression determines which elements of the repeating structure are applied at run time.

For example, to calculate the value of the output element **a**, you must concatenate the elements in the repeating structure **C** whose length is greater than 4. You can use the following XPath expression:

```
$C2[fn:string-length() > 4]
```

Note: Always use content-assist to build your XPath expressions. You must use the element names used by the Graphical Data Mapping editor.



Define when the transform is applied at run time

You can use any of the input elements to the **Concat** transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

fn:substring

In the Graphical Data Mapping editor, you can use the **fn:substring** transform to set the value of an output element to a substring of the original input value. You must define the start position, and optionally, the number of characters that you need to extract.

Overview

The **fn:substring** XPath 2.0 function takes two arguments, an input string and a 1-based number, to return a part of the original string, beginning from the position indicated. You can also specify a third optional parameter as a number, to indicate the end position to compose the resulting string.

The following function call `fn:substring("12345", 2, 3)` returns "234".

The following function call `fn:substring("12345", 2)` returns "2345".

The `fn:substring-before()` and `fn:substring-after()` functions are variations of the `fn:substring()` function.

- Use the `fn:substring-before(arg1, arg2)` function when you need the part of `arg1` that occurs before `arg2` occurs in it. For example, `substring-before('1234567/CustomerID', '/')` returns `1234567`.
- Use the `fn:substring-after(arg1, arg2)` function when you need the part of `arg1` that occurs after `arg2` occurs in it. For example, `substring-after('1234567/CustomerID', '/')` returns `CustomerID`.

For more information about XPath, see XPath tutorial or W3C XML Path Language (XPath) 2.0.

Inputs to the transform

The `fn:substring` transform takes as input one simple type element. This element is used to define the first argument of the `fn:substring` function.

Note: If the simple type is not of type `xs:string`, the transform will fail at run time with **BIP3945E**. Cast the element to a string if you want to use a non-string input element. For more information, see “Cast type (`xs:type`)” on page 199.

Arguments of the XPath function

You define the arguments to the `fn:substring` function in the **General** tab of the `fn:substring` transform properties view.

- The first argument is a string element. You can define a literal expression, a constant, an input element, or a custom XPath expression.
- The second argument is named **startLocation** and specifies the a start position.
- The third argument is named **length**, is optional, and specifies the number of characters that you need to select.

In the following figure, the `fn:substring` transform has one input. The input is used to define the first argument of the `fn:substring` function.

The screenshot shows the Eclipse IDE interface for configuring a transform. At the top, a diagram illustrates the transform flow: an input element 'A' (type string) is connected to the 'fn:substring' transform, which then produces two output elements, 'a' and 'b' (both type string). Below the diagram, the 'Transform - substring' properties view is open, showing the 'General' tab. The description states: 'Takes a string and two integers, and returns the portion of the string beginning at the position indicated by the first integer and continuing for the number of characters indicated by the second integer. The first character of the string is located at position 1, not position 0. For example, `fn:substring("123456", 2, 4)` returns "2345". See [XPath 2.0 Specification...](#)'

The parameters table is as follows:

Name	Type	Value
sourceString	xs:string	\$A3
startLocation	xs:double	3
length	xs:double	4

Define when the transform is applied at run time

You can use any of the input elements to the `fn:substring` transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

For more information, see “Configuring the properties of a transform” on page 84.

fn:count

In the Graphical Data Mapping editor, you can use the **fn:count** transform to set the value of an output element to the total number of elements in the input element.

Overview

The XPath 2.0 function `fn:count((arg1, arg2, arg3, ...))` takes a list of elements and returns the total number of elements.

The **fn:count** transform is the representation of the **fn:count** XPath function in the Graphical Data Mapping editor.

You can have any number of inputs to the **fn:count** transform. These inputs can be used to define the arguments of the **fn:count** function.

Inputs to the transform

You can connect one or more inputs to the **fn:count** transform. These inputs are used to define the argument of the **fn:count** function.

In the following figure, there are 4 inputs to the **fn:count** transform.

The screenshot shows the Graphical Data Mapping editor interface. On the left, a table lists input elements: A (string, [1..1]), B (string, [0..*]), C (string, [1..*]), D (int, [1..1]), E (string, [1..1]), F (string, [0..*]), and G (string, [1..*]). A central box labeled 'fn:count' has four green lines connecting it to elements A, C, D, and E. On the right, a table lists output elements: a (string, [1..1]), b (string, [1..1]), c (cType, [1..*]), d (string, [1..1]), e (string, [1..1]), d (dType, [1..1]), d1 (int, [1..1]), and d2 (string, [1..1]). Below the diagram is the 'Transform - count' properties view. The 'General' tab is active, showing a description: 'Returns the number of items in the input value. See XPath 2.0 Specification'. The 'Parameters' section contains a table with one row: Name 'item', Type 'item[]', and Value '(\$G,\$F,\$A,\$D)'. There are 'Add' and 'Edit' buttons next to the parameters table.

When you run the following message, the value of **d1** is **4**.

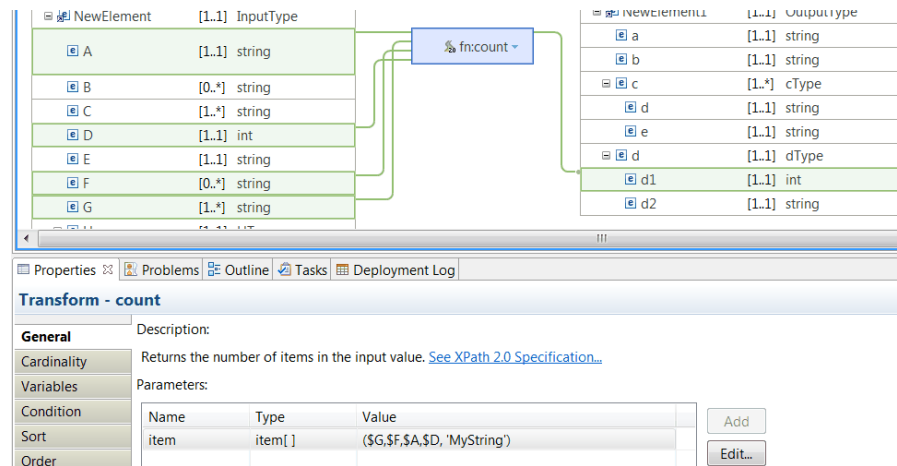
```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>10</A>
  <C>100</C>
  <C>1000</C>
  <C>10000</C>
  <D>1000</D>
  <E>112</E>
</NewElement>
```

Arguments of the XPath function

You define the argument to the **fn:count** function in the **General** tab of the **fn:count** transform properties view.

You can define a literal expression, a constant, an input element, or a custom XPath expression as the argument.

In the following figure, there are 4 inputs to the `fn:count` transform. **A** is only considered if the string length is greater than 4. A literal expression has been added as an additional element in the argument.



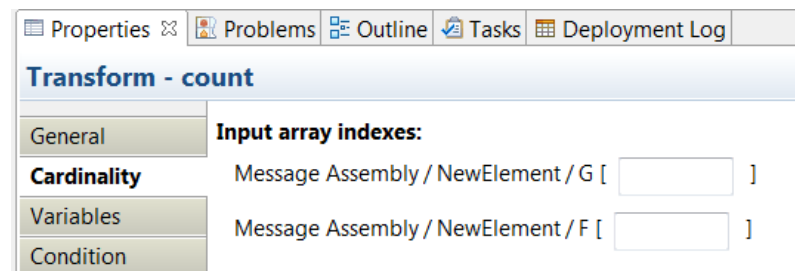
When you run the following message, the value of **d1** is 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>10</A>
  <C>100</C>
  <C>1000</C>
  <C>10000</C>
  <D>1000</D>
  <E>112</E>
</NewElement>
```

Cardinality

The **Cardinality** property determines the elements (also known as indexes) in each repeating input element that is processed by the `fn:count` transform.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.



Define when the transform is applied at run time

You can use any of the input elements to the `fn:count` transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied. You define this expression in the **Condition** tab of the transform properties.

For more information, see “Configuring the properties of a transform” on page 84.

fn:sum

In the Graphical Data Mapping editor, you can use the **fn:sum** transform to set the value of an output element to a numeric type that is the result of the sum of all the values in a sequence. You can also use the **fn:sum** transform to set the value of an output element to the sum of durations in a sequence.

Overview

You can use the XPath 2.0 function `fn:sum(arg1, arg2)` to sum numeric or duration values in a sequence.

The **fn:sum** transform is the representation of the **fn:sum** XPath function in the Graphical Data Mapping editor.

You can have any number of inputs to the **fn:sum** transform. These inputs can be used to define the arguments of the **fn:sum** function.

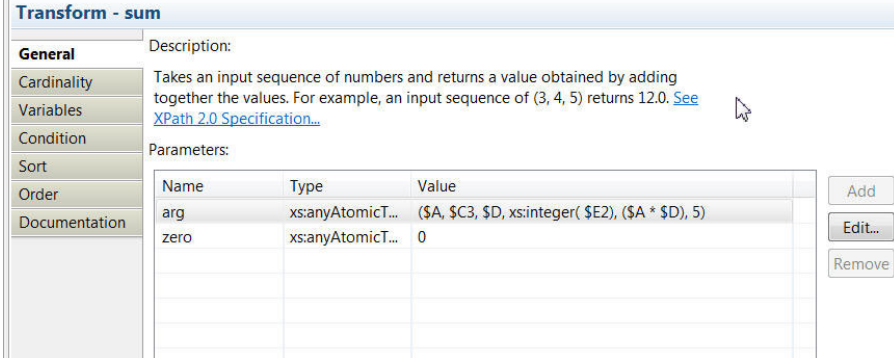
Inputs to the transform

You can connect one or more inputs to the **fn:sum** transform. These inputs are used to define the arguments of the **fn:sum** function.

Arguments of the XPath function

You define the two arguments to the **fn:sum** function in the **General** tab of the **fn:sum** transform properties view.

- The first argument (**arg**) contains any number of numeric and untyped values, or any number of duration values.
- The second argument (**zero**) contains the default value returned by **fn:sum** when the sequence is empty. You can set this value to the integer 0, a duration of zero seconds, or any other atomic value.



Transform - sum

General Description: Takes an input sequence of numbers and returns a value obtained by adding together the values. For example, an input sequence of (3, 4, 5) returns 12.0. [See XPath 2.0 Specification...](#)

Parameters:

Name	Type	Value
arg	xs:anyAtomicT...	(\$A, \$C3, \$D, xs:integer(\$E2), (\$A * \$D), 5)
zero	xs:anyAtomicT...	0

Buttons: Add, Edit..., Remove

You must consider the following points when you define the arguments to the **fn:sum** function:

- Untyped values are cast to a double type element.
- You can edit the sequence specified for **arg** and then add literal values, cast untyped values, and add more values that are the result of arithmetic operations.
- If **arg** is an empty sequence, and **zero** is set, then **fn:sum** returns the value specified by **zero**.

- When you sum durations, you must ensure that all the elements in the sequence have the same type, that is, all values are of type **xs:yearMonthDuration** or all values are of type **xs:dayTimeDuration**.

Note: If an input is not of a numeric type such as **xs:int**, the transform will fail at run time. Cast the input element to a numeric type. For more information, see “Cast type (xs:type)” on page 199.

Cardinality

The **Cardinality** property determines the elements (also known as indexes) in the repeating input element that are processed by the **fn:sum** transform.

You can configure the **Input array indexes** section to select specific instances of the input array. For more information, see “Selecting the indexes of input array elements” on page 28.

Define when the transform is applied at run time

You can use any of the input elements to the **fn:sum** transform to define a conditional expression that defines the condition under which the transform is applied. If the condition evaluates to **true**, the transform is applied.

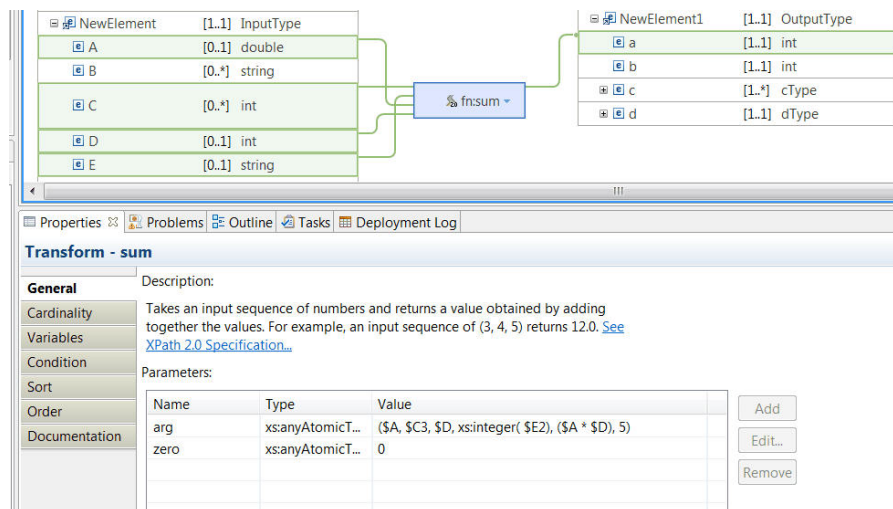
For more information, see “Configuring the properties of a transform” on page 84.

Example: Calculate the value of a numeric output element

In the following figure, the **fn:sum** transform has four inputs and the **fn:sum** function has two arguments.

The input **E** is of type string and is cast to an integer type.

The second argument **zero** is set to the value of multiplying two input elements.



The **fn:sum** transform returns **11147** for the following input message:

```
<?xml version="1.0" encoding="UTF-8"?>
<NewElement>
  <A>10</A>
  <C>2</C>
```



```
<C>10</C>  
<D>1000</D>  
<E>120</E>  
</NewElement>
```

Chapter 27. Scenario: Transforming SOAP messages by using a message map

In IBM Integration Bus, you can use message maps to transform SOAP messages. You can set functions in a graphical data map to transform a SOAP input message into a required SOAP output message.

WebSphere Message Broker Version 8.0 introduces graphical data maps. These message maps replace the previous message map format, and are created as *.map files. You can continue to use maps that were created in versions earlier than WebSphere Message Broker Version 8.0. However, if you need to modify any of your legacy maps, you must convert these legacy message maps into *.map message maps.

This scenario shows how to create a message map that transforms a SOAP message, and how to apply transforms.

If you want to try out the scenario, you can either use your own integration solutions, or set up a copy of the sample provided with the scenario.

Introduction to the "Transforming SOAP messages by using a message map" scenario

You can create a message map that transforms an existing SOAP message. You can configure the map properties and define data transformations between simple elements, complex elements, and repeating elements. Review the topics in this section to understand what is covered in this scenario, the situations in which a business might want to follow the scenario, and an overview of the solution that is proposed by the scenario.

About this task

WebSphere Message Broker Version 8.0 introduces graphical data maps. These message maps replace the previous message map format, and are created as *.map files. If you migrate from an earlier version of WebSphere Message Broker Version 8.0, you can continue using your legacy maps. However, if you need to modify any of your legacy maps, you must convert these legacy message maps into *.map message maps. For more information about converting maps, see *Converting a message map from a .msgmap file to a .map file*.

Message maps are based on XML schema and XPath 2.0 standards. You can use these maps to transform and enrich messages in your integration solution. These maps offer the ability to achieve the transformation without the need to write code, providing a visual image of the transformation, and simplifying its implementation and ongoing maintenance.

IBM Integration Bus supports SOAP 1.1 and SOAP 1.2 messages. A SOAP message is encoded as an XML document, consisting of an **Envelope** element, which contains an optional **Header** element, and a mandatory **Body** element. The **Fault** element, contained in the **Body** element, is used for reporting errors.

Message maps provide a simple way to transform SOAP messages since they are messages encoded as an XML document.

Although using message maps is very intuitive, you might need help creating a map that transforms SOAP messages due to the multi part structure of a SOAP message. This scenario explains how to create a message map that transforms a SOAP message, how to configure the map properties, how to define transformations between the different parts of a SOAP message, and how to define transformations between simple elements, and complex elements.

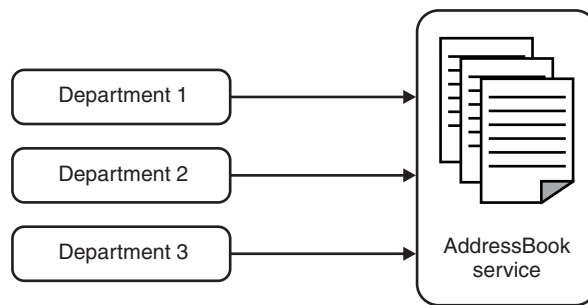
Read the following topics to understand the scenario and the concepts the scenario is intended to demonstrate:

- “Context”
- “Technical solution” on page 247
- “Implementing the solution” on page 248

Context

This scenario explains how to create a message map that transforms a SOAP message, how to configure the map properties, and how to define transformations between the different parts of a SOAP message.

Your company has implemented an AddressBook service that is used by different departments in different countries in your organization. This service allows your employees to obtain a client's mail address or to save a new client's mail address.



The company uses IBM Integration Bus to develop and manage a number of integration solutions that transform and communicate data between source and target systems. In order to make the service reusable by multiple applications, you design an application responsible for the transformation of the different data formats between the requesting application and the AddressBook service. The AddressBook service is a SOAP based service that stores a new address or returns an address to the user. You use a message map to define how to transform the SOAP message based on the operation that your user requests.

In IBM Integration Bus, you can transform SOAP messages by using any of the following methods:

- Transform data by using a message map in a Mapping node.
- Transform data by using ESQL in a Compute node.
- Transform data by using an XSL transform in a XSLTransform node.
- Transform data by using Java in a JavaCompute node.
- Transform data by using the PHP scripting language.
- Transform data by using .NET applications on Windows integration nodes.

In IBM Integration Bus, you have the following choices to implement a visual transformation:

- You can use a Mapping node to transform the incoming message, create new output messages, and interact with information in a database by using a graphical data map.
- You can use the XSLTransform node to transform the incoming XML message by using an XSL style sheet.

This scenario demonstrates how to transform SOAP messages by using a message map that you create, configure and assign to a Mapping node.

Technical solution

To complete the scenario and successfully transform message data, you must create a message map and customize it based on your SOAP message and transformation requirements. In this scenario, you use the SOAP domain to parse your SOAP message.

You configure a message domain on an input node such as a SOAPInput node to define the parser that IBM Integration Bus uses to parse a message. IBM Integration Bus supplies a range of parsers to parse and write messages in different formats.

IBM Integration Bus supports SOAP 1.1 and SOAP 1.2 messages.

Depending on the message domain that you configure in your input node, you might have to consider the differences between SOAP 1.1 and SOAP 1.2 when transforming SOAP messages.

- If you receive a SOAP message through a SOAPInput node, the SOAP parser handles SOAP 1.1 or SOAP 1.2 automatically. The SOAP domain uses a common logical tree format that is independent of the exact format of the web service message. For details of the SOAP tree format, see SOAP tree overview.
- If you receive a SOAP message through an HTTPInput node, the XMLNSC parser handles your SOAP 1.1 or SOAP 1.2 message differently. When you create a message map, you must be aware of the SOAP version, and configure the correct SOAP 1.1 or SOAP 1.2 schema when you create and configure your graphical data map.

Depending on the nodes that you use when you model your message flow or your service operation, and the message domain you configure, you must use a different schema model:

- If you use the SOAP nodes excluding the SOAPExtract node, you must map the **SOAP_Domain_Msg** in the SOAP domain.
- If you use the SOAP nodes including the SOAPExtract node, and the Mapping node is wired after a SOAPExtract node, you must map the schema associated with your operation in the XMLNSC domain. You use the SOAPExtract node to remove SOAP envelopes, allowing just the body of a SOAP message to be processed.
- If you use HTTP nodes or MQ nodes, you must map the SOAP 1.1 or the SOAP 1.2 schema as the root model of the map in the XMLNSC domain.

The following table summarizes the different types of nodes and domains that you can use to map a SOAP message and the schema that you must use when you use a message map to transform a SOAP message.

Table 22. Schemas to use when transforming a SOAP message

Message domain		Schema to configure in a message map
SOAP	SOAP nodes	SOAP_Domain_Msg
XMLNSC	SOAP nodes including the SOAPEExtract node where the SOAPEExtract node is modeled before the Mapping node	Schema associated with the SOAP operation
XMLNSC	HTTP nodes	SOAP 1.1 or 1.2 schema as the root model of the map
XMLNSC	MQ nodes	SOAP 1.1 or 1.2 schema as the root model of the map

Use this scenario to learn how to create a message map that transforms a SOAP message in a message flow where the Mapping node is connected directly from a SOAPInput node with no SOAPEExtract node. For more information, see “Implementing the solution.”

Implementing the solution

You can transform a SOAP message by using a message map. You can use message maps to route, transform and enrich existing messages in your integration solution.

Before you begin

To start the scenario, create the initial configuration. For more information, see “Creating the scenario initial configuration” on page 249.

Procedure

Complete the following steps to create a message map that transforms a SOAP message:

1. Create a message map. For more information, see “Creating a message map to transform SOAP messages” on page 250.
2. Configure the **Override** function to transform some elements of the Properties folder. For more information, see “Transforming elements in the Properties folder by using the **Override** function” on page 254.
3. Customize the message map to include the local environment tree. For more information, see “Customizing a message map to include the local environment tree” on page 257.
4. Configure the local environment tree variables folder by using the **Cast** function. For more information, see “Configuring the local environment tree **Variables** folder by using the **Cast** function” on page 260.
5. Configure the message map to include your SOAP message. For more information, see “Configuring the message map to include the SOAP message” on page 264.

In IBM Integration Bus, a SOAP message is described by a generic model. For more information, see SOAP tree overview.

In addition to the standard SOAP parts, the IBM Integration Bus SOAP message generic model includes a *Context* part that includes contextual information about the current SOAP message being processed. This is the only

part in a message map whose structure is included automatically. You must define the other SOAP message parts manually by using the **Cast** function. You must customize the message map to include your SOAP envelope and SOAP attachments.

Results

You have a message map that transforms your SOAP message.

Creating the scenario initial configuration

This scenario was developed by using a sample initial configuration. Follow the instructions to set up the sample to try out the scenario in the same way as it was originally developed.

Before you begin

- Download a copy of the `AddressBookProviderInitialConfiguration.zip` from the IBM Integration Community.
- Download a copy of the `AddressBookProviderFinalConfiguration.zip` from the IBM Integration Community to set up the final scenario configuration and see the result of following the steps that are documented in the scenario.
- Make sure you have access to a IBM Integration Bus runtime environment and a IBM Integration Studio installation with the default configuration deployed. For more information on installing IBM Integration Bus components, see [Installing in the IBM Integration Bus information center](#).

Procedure

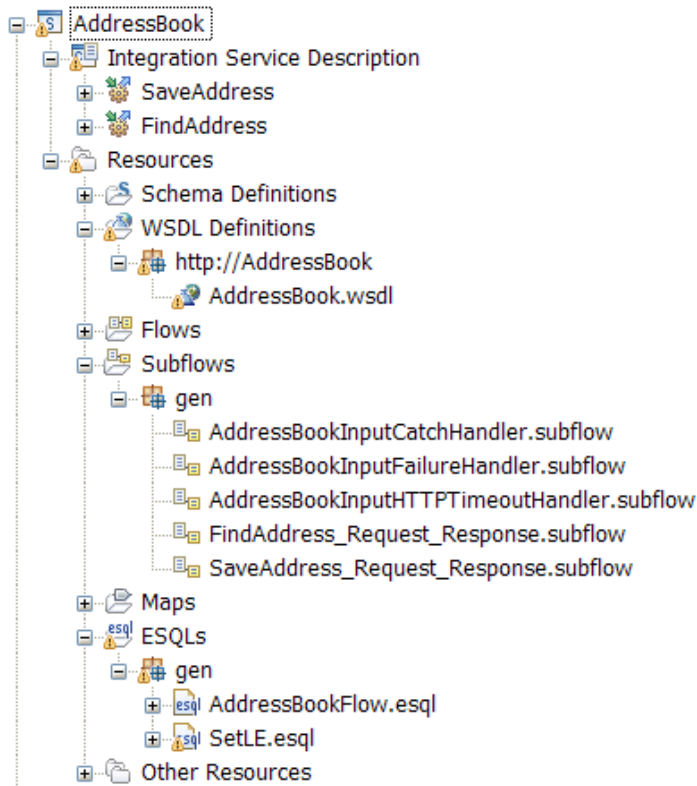
Complete the following steps to set up the sample initial configuration that was used to develop the scenario:

1. Install IBM Integration Studio. For more information, see [Installing](#) .
2. Import the `AddressBookProviderInitialConfiguration.zip` file:
 - a. Click **File** > **Import**. The Import wizard opens.
 - b. Expand **Other**, click **Project Interchange**, then click **Next**.
 - c. Specify the location of `AddressBookProviderInitialConfiguration.zip`.
 - d. Specify the location of the open workspace.
 - e. Select the projects that you want to import into your workspace. For this scenario, select all projects. Then, click **Finish**.

Results

You imported the scenario source files.

In the **Application Development** view, you should see the following resources:



What to do next

Follow the steps for “Creating a message map to transform SOAP messages.”

Creating a message map to transform SOAP messages

Create a message map with a SOAP message as input and a SOAP message as output.

About this task

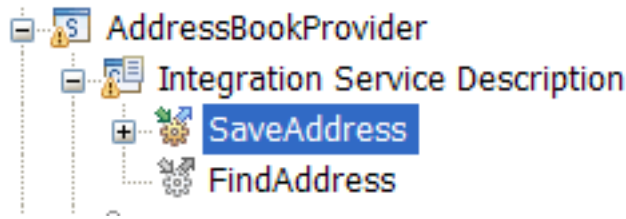
This scenario demonstrates how to create a message map within one operation of a service.

If you want to use your own application, you can follow the same steps. The difference is that you create the map in a message flow or subflow within the application or library referenced by the application.

Procedure

Complete the following steps to create a map in the IBM Integration Studio:

1. Start the New Message Map wizard.
 - a. Identify the **SaveAddress** operation.



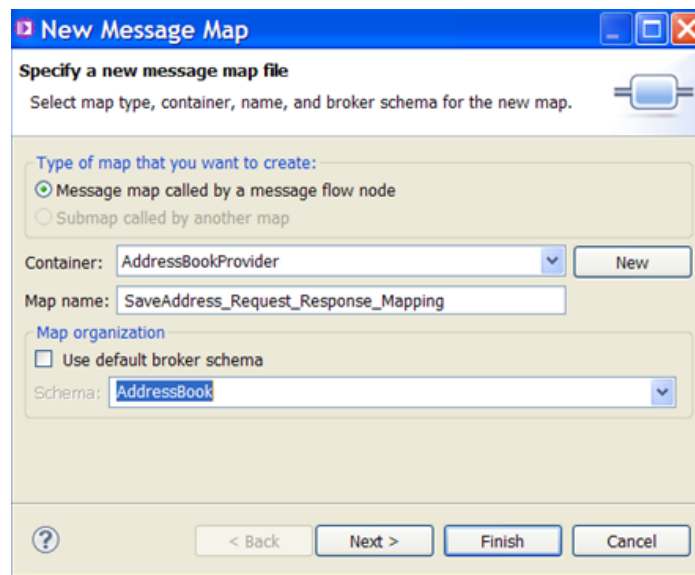
- b. Double-click the **SaveAddress** operation and drag and drop a Mapping node.
- c. In the Mapping node properties, select the **Description** tab, and enter **Normalize_AddressBook_Transform** as the **Node name**.
- d. Connect the **Normalize_AddressBook_Transform** Mapping node between the two nodes where the message transformation is required.



- e. Double-click the **Normalize_AddressBook_Transform** Mapping node to start the New Message Map wizard.
2. Optional: Edit the *Map name* field and enter your map name.
You can keep the default name provided by IBM Integration Bus.
In the scenario, the map name that you use is the default name **SaveAddress_Request_Response_Mapping.map**.
 3. Enter the broker schema name **AddressBook** in the field *Schema* to create a new broker schema.

To organize your resources, and to define the scope of resource names to ensure uniqueness, you create broker schemas. For more information on how to create a broker schema in the IBM Integration Studio, see [Creating a broker schema](#).

After you enter **AddressBook** as the name of the broker schema, the window looks as follows:



4. Click **Next**.
5. Select the map inputs and outputs.

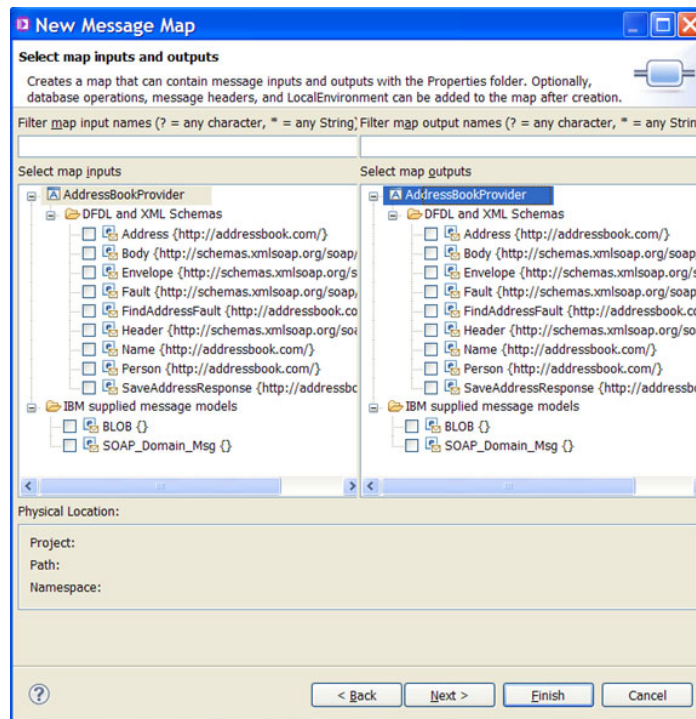
- a. Select the map input **SOAP_Domain_Msg{}**.
- b. Select the map output **SOAP_Domain_Msg{}**.

In the scenario, you have a SOAPInput node that produces a SOAP_Domain_Msg. A Mapping node is connected to the SOAPInput node, and receives as input a **SOAP_Domain_Msg** message.

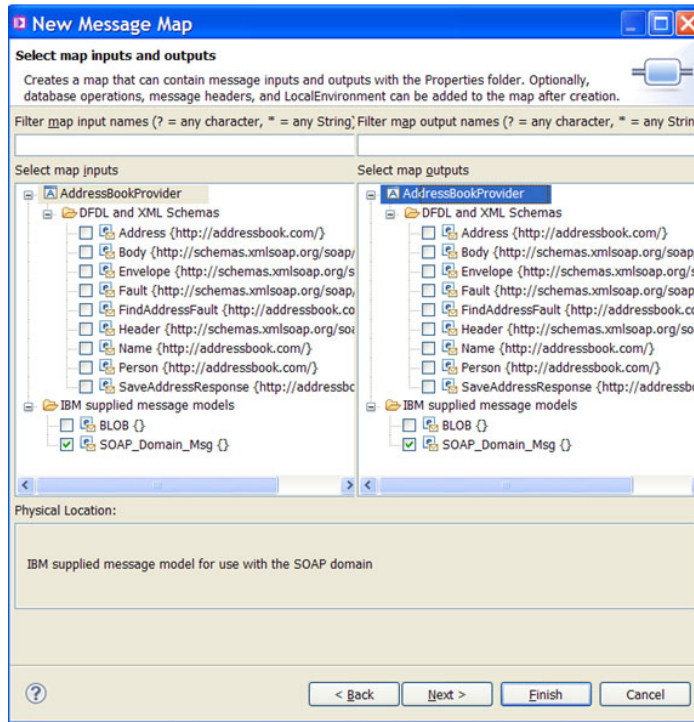
Note: In IBM Integration Bus, you can choose from multiple inputs and multiple outputs in a message map. However, you can only select one input and one output for a message map.

- If you use a SOAPInput node, you must map the **SOAP_Domain_Msg** in the SOAP domain.
- If you use a SOAPInput node followed by a SOAPExtract node, you must map the schema associated with your operation in the XMLNSC domain. You use the SOAPExtract node to remove SOAP envelopes, allowing just the body of a SOAP message to be processed.
- If you use HTTP nodes or MQ nodes, you must map the SOAP 1.1 or the SOAP 1.2 schema as the root model of the map in the XMLNSC domain.

The following figure shows you the options you have as potential map inputs and map outputs in the scenario:



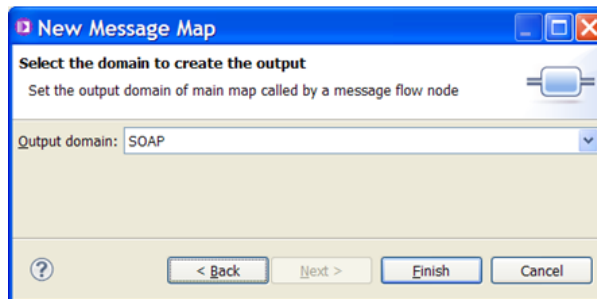
The following figure shows what the Select map inputs and outputs window looks like after you have selected your map input and map output for a SOAP message transformation:



6. Click **Next**.
7. Select the output domain **SOAP**.

Note: The only domain option available is the **SOAP** domain.

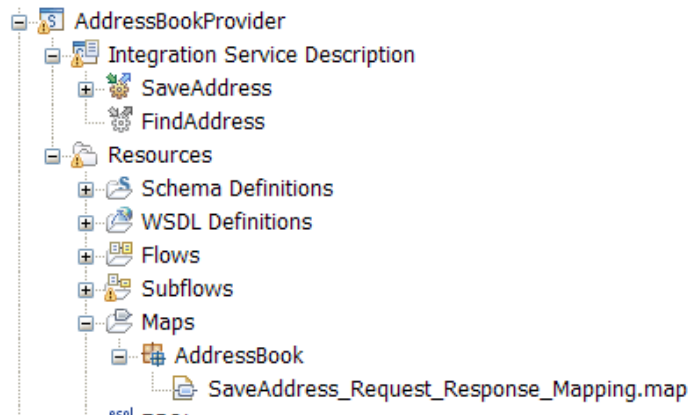
The following figure shows what the New Message Map - Select the domain to create the output window looks like after you have selected the domain.



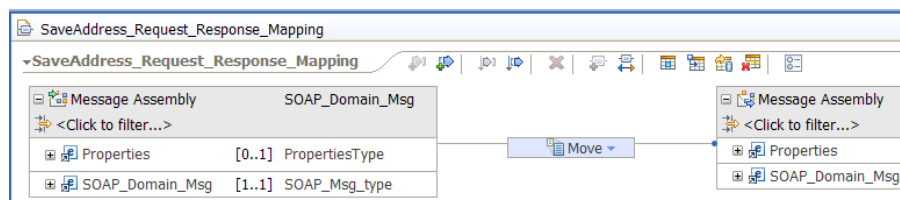
8. Click **Finish**.

Results

The message map **SaveAddress_Request_Response_Mapping.map** is created in the Application Development view, within the folder **Maps** located under your **AddressBookProvider** service project. The map is created under the **AddressBook** schema.



The map opens in the Graphical Data Map editor. The following figure shows what the map looks like when it is first opened.



What to do next

Configure the Properties folder. For more information, see “Transforming elements in the Properties folder by using the **Override** function.”

Transforming elements in the Properties folder by using the **Override** function

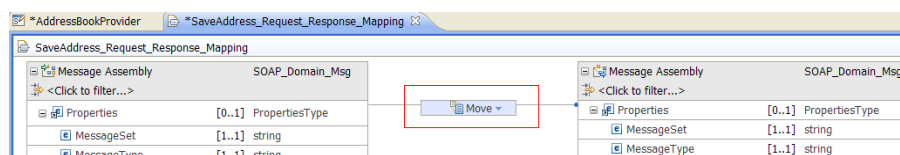
You can use the **Override** function to copy a complex type from the input message to the output message, while updating some of the child elements in the complex type. A message assembly component is described by a complex data structure.

Before you begin

Create a message map. For more information, see “Creating a message map to transform SOAP messages” on page 250.

About this task

The Properties folder has a **Move** transform defined automatically when you create a message map so that all elements in the Properties folder are copied to the output Properties folder structure. The following figure shows the message map that you have created previously:



Note: You can only use the **Override** function to include **Move** transforms and **Assign** transforms.

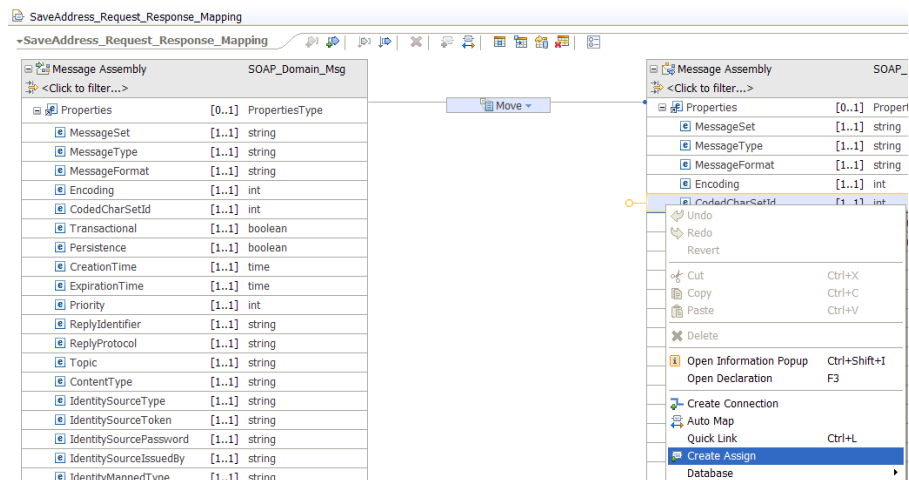
In the scenario, you define an **Assign** transform to change the value of the `CodedCharSetId` element in the `Properties` folder from UTF-16 to UTF-8. Support for Universal Transformation Format (UTF)-16 encoding is required by the WS-I Basic Profile 1.0. UTF-16 is a unicode encoding scheme using 16-bit values to store Universal Character Set (UCS) characters. UTF-8 is the most common encoding that is used on the Internet and UTF-16 encoding is typically used for Java and Windows product applications. For more information on the values that you can set for the `CodedCharSetId` element, see Supported code pages.

Procedure

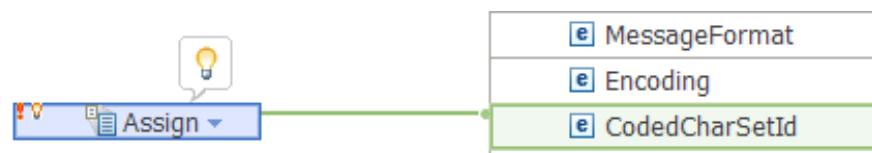
Complete the following steps to modify the `CodedCharSetId` element of the properties folder:

1. Right-click the `CodedCharSetId` element, and then select the menu option **Create Assign**.

The following figure shows the message map with the options you can choose from when you right-click the element `CodedCharSetId`.

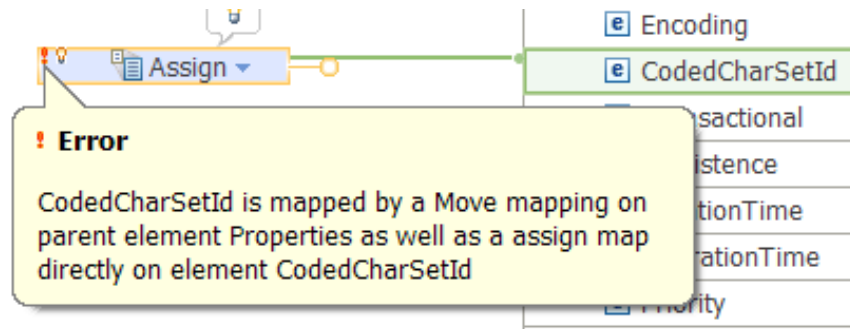


The **Assign** transform is defined and connected to the `CodedCharSetId` element in the output Properties folder.

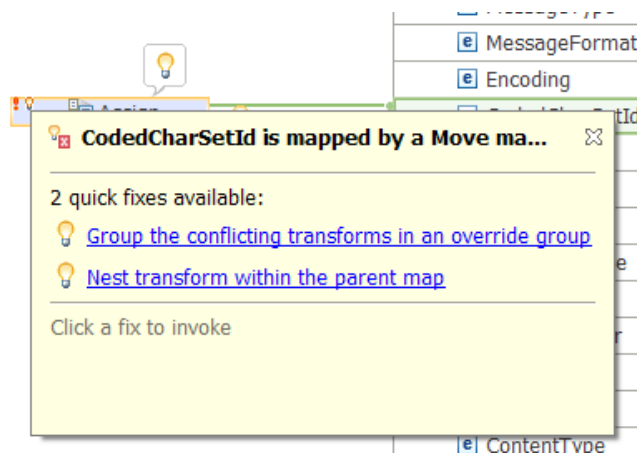


You get the following icons on the top left hand side of the transform:

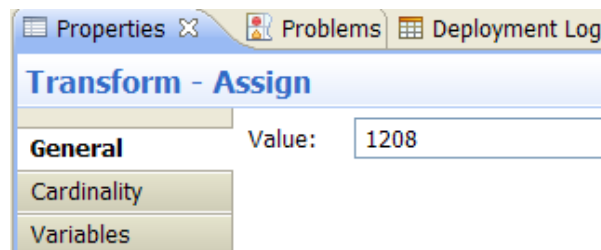
- An **Error** icon represented with a red exclamation mark. You can ignore this error and continue. You get the error because you have defined two transformations on an element and this is not allowed. By using the **Override** function, SOAP, you fix the problem.



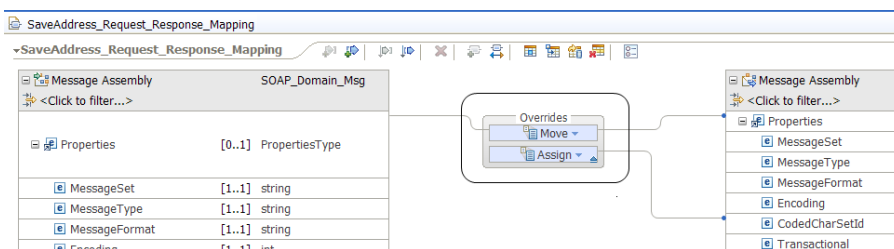
- A **suggestion** icon represented by a yellow light bulb. When you hover over the icon, you get the following pop-up window:



2. Set the value of the **CodedCharSetId** to 1208. This is the value for UTF-8. In the **Assign** transform properties tab, you set the value in the **General** tab. You set the element *Value* to 1208.



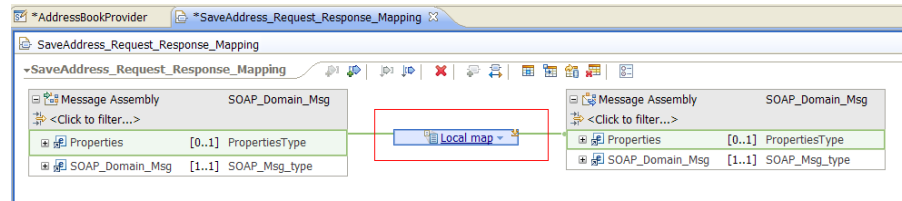
3. When you hover over the yellow light bulb, choose **Group the conflicting transforms in an override group**. This option is the recommended approach and allows you to maintain visibility of the transforms you have defined in the main transformation map.



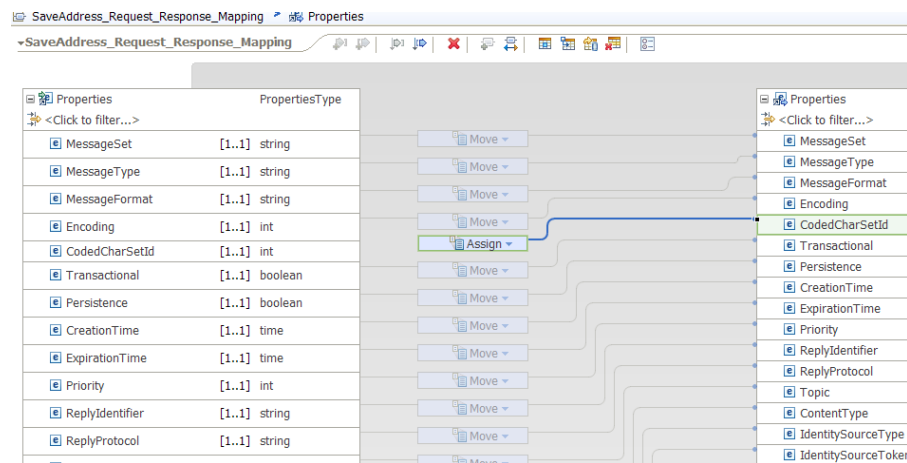
Results

You have transformed elements of the Properties folder by using the **Override** function.

If you choose **Nest transforms within the parent map**, a **Local map** transform is defined between the input Properties folder and the output Properties folder.



The local map that is created contains a **Move** transform per element, with the exception of the **CodedCharSetId** element that has an **Assign** transform.



What to do next

Configure the message map to include the local environment tree. For more information, see “Customizing a message map to include the local environment tree.”

Customizing a message map to include the local environment tree

To customize your message map to include the local environment tree, you must add the local environment tree to the input message and to the output message, and then define transforms between them.

Before you begin

1. Create a message map. For more information, see “Creating a message map to transform SOAP messages” on page 250.
2. Define transformations between elements of the Properties folder. For more information, see “Transforming elements in the Properties folder by using the **Override** function” on page 254.

About this task

By default, when a message map is created, the only message assembly component that is configured automatically is the Properties folder. The input Properties folder is connected to the output Properties folder with a Move transform. It is also possible to map other message assembly components such as transport headers and the local environment tree.

In this scenario you configure the local environment tree as an additional component for a message map in the Graphical Data Mapping editor.

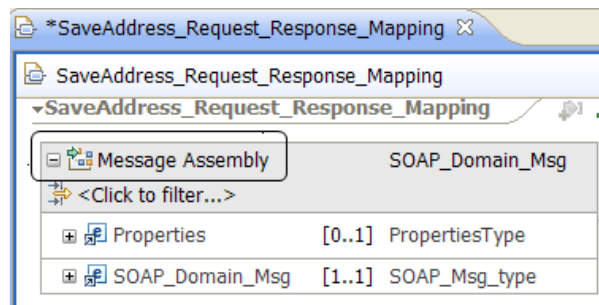
Procedure

To configure the local environment tree in a message map, complete the following steps:

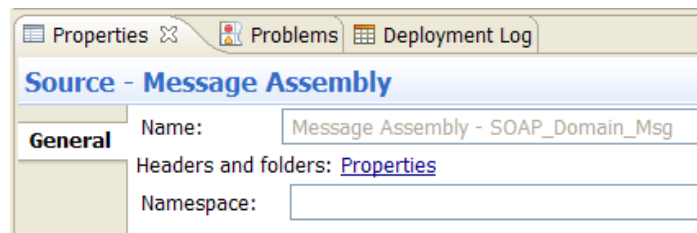
1. Open the message map in the Graphical Data Mapping editor.
2. Add the local environment tree to the input message.

- Method 1:

- a. Select **Message Assembly**.



- b. In the Properties view, select the **General** tab.

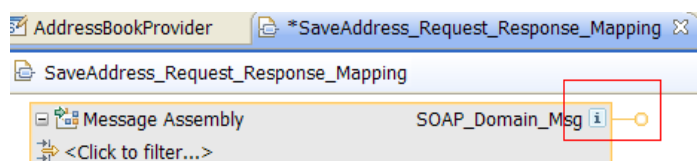


- c. Click **Properties**.

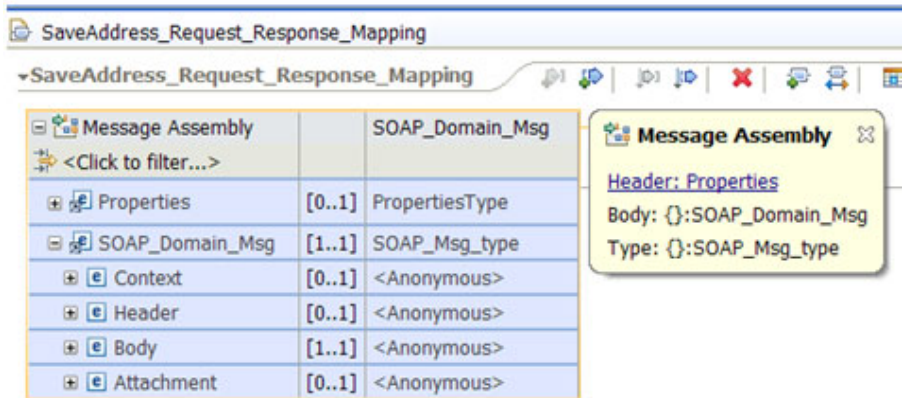
Note: If you have other structures included in your message assembly, the option that you can click includes all the different message assembly components that you have currently selected. For example, if you had the Properties tree and the local environment tree selected, you click **LocalEnvironment, Properties**.

- Method 2:

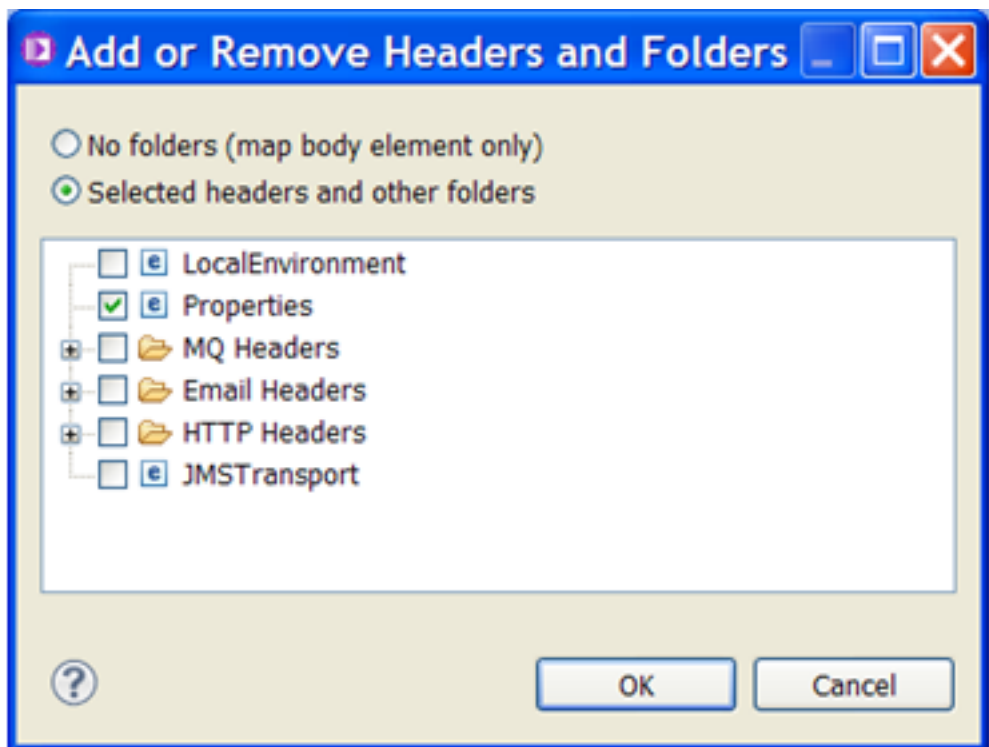
- a. Select the information **i** icon located by the input message body type.



- b. Select **Header: Properties**.



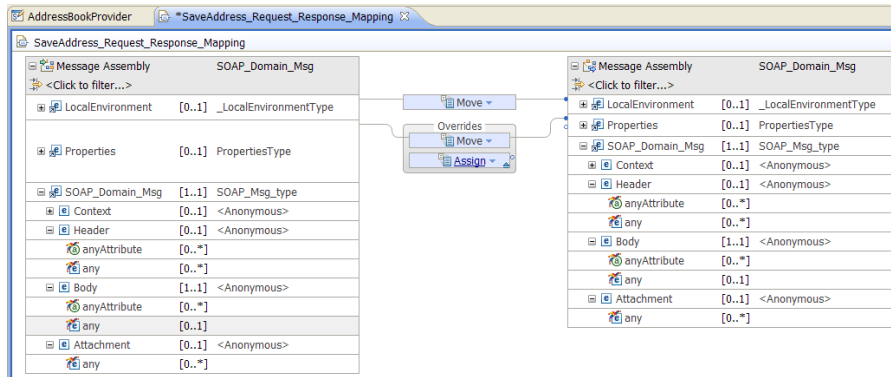
- In the Add or Remove Headers and Folders window, select **LocalEnvironment**, and then click **Ok**.



- Follow the previous steps to add the local environment tree to the output message.
- Define a **Move** transform between the input local environment tree and the output local environment tree. You can add other transforms. For more information, see Specifying a transform and Transform types in the Graphical Data Mapping editor.

Results

The following figure shows the message map in the Graphical Mapping Data editor after you create a message map to transform a SOAP message and configure the local environment tree:



What to do next

Add variables defined in the local environment tree variables folder, see “Configuring the local environment tree **Variables** folder by using the **Cast** function.”

Configuring the local environment tree **Variables** folder by using the **Cast** function

You can use the **Cast** function to define variables in a message map that are defined in the local environment tree **Variables** folder.

Before you begin

Customize the message map to include the local environment tree. For more information, see “Customizing a message map to include the local environment tree” on page 257.

About this task

The local environment tree is a part of the logical message tree in which you can store information while the message flow processes the message. You use the local environment tree to store variables that can be referred to and updated by message processing nodes that occur later in the message flow. You can also use the local environment tree to define destinations (that are internal or external to the message flow) to which a message is sent.

When you add the local environment tree to a message map, you must provide transforms for all of its elements so that the input values of each element are not lost. You can copy the input field unchanged or modified by a transform. Many IBM Integration Bus nodes depend on information in the local environment tree being copied along the message flow.

The variables folder in the local environment tree is defined as *xsd:any*. When you add the local environment tree, you can see the structure of the destination folders with all its elements, and a **Variables** folder with a single element defined with a generic type.

Variables	[0..1] _LocalEnvironmentVariablesType
any	[0..*]

You manually define the elements that are included in the **Variables** folder. There is no predefined structure for the **Variables** folder. Each message flow has its own local environment tree **Variables** folder. For this reason, if you want to access any of these elements within your message map, you must define each element that you want to use in the message map by using the **Cast** function.

Note:

- You can use the **Cast** function to explicitly define other elements in the message map message assembly.
- In IBM Integration Bus, the local environment tree predefines other folders to reflect the data created and used by IBM Integration Bus nodes.

In this scenario, you create an element called **Country** under the local environment **Variables** folder to be used by other nodes later in the message flow for routing.

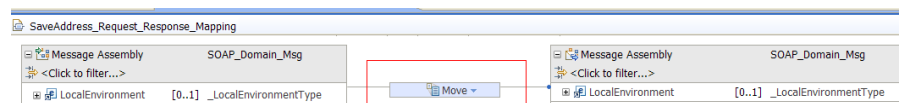
Procedure

To add the Country element to the local environment tree Variables folder complete the following steps:

1. Define a **Move** transform between the input local environment tree and the output local environment tree. Create a connection between the input local environment tree and the output local environment tree. You can do this in any of the following ways:
 - In the message map, right-click the input local environment tree, and select **Create Connection**. Move the mouse to the output local environment tree, and click local environment to define the **Move** transform.
 - In the message map, right-click the input local environment tree, and select **Quick Link**. A new window appears where you can select the output element local environment. Use this option when you have a long list of output elements. You can filter the list in Quick link too.

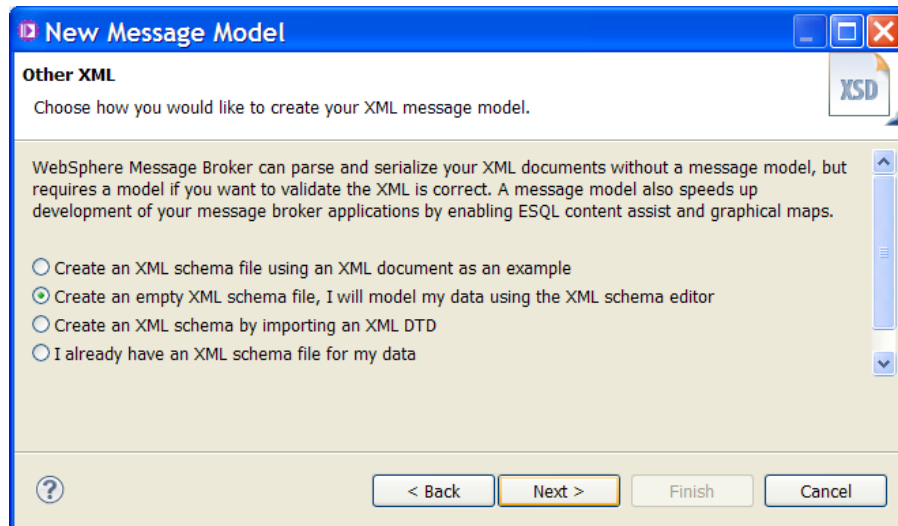
If you need to modify only some fields in the local environment tree, you can use a **Move** transform to copy the local environment tree unchanged, and then use the **Override** function to modify the elements you must update.

The following figure shows graphically how the **Move** transform is defined between the input local environment tree and the output local environment tree.

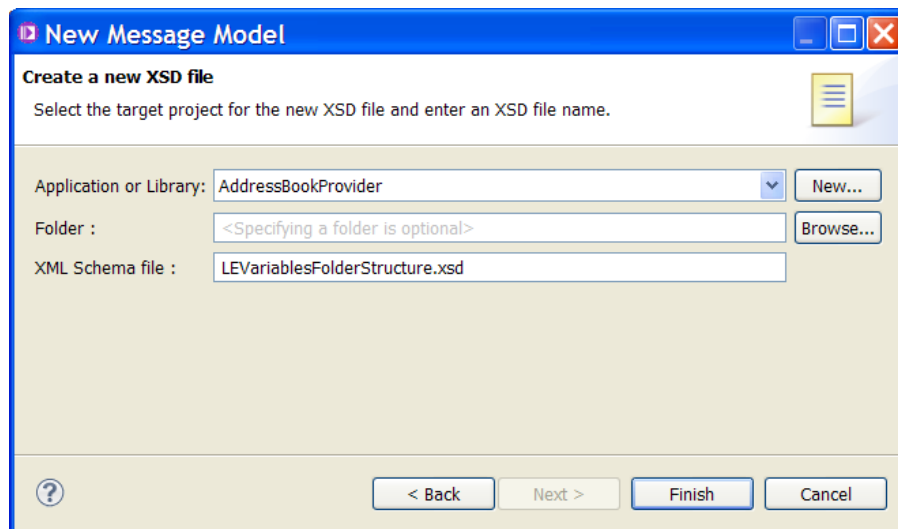


All the input values are copied onto the output values.

2. Create a schema file in your application, service, or library to define the elements of the local environment tree **Variables** folder and their type:
 - In the Application Development view, select **New... > Message Model... > Other XML**. Click **Next**.
 - Select **Create an empty XML schema file, I will model my data using the XML schema editor**, and then click **Next**.



- Create the XSD file **LEVariablesFolderStructure.xsd** within the project **AddressBookProvider**. Then, click **Finish**.



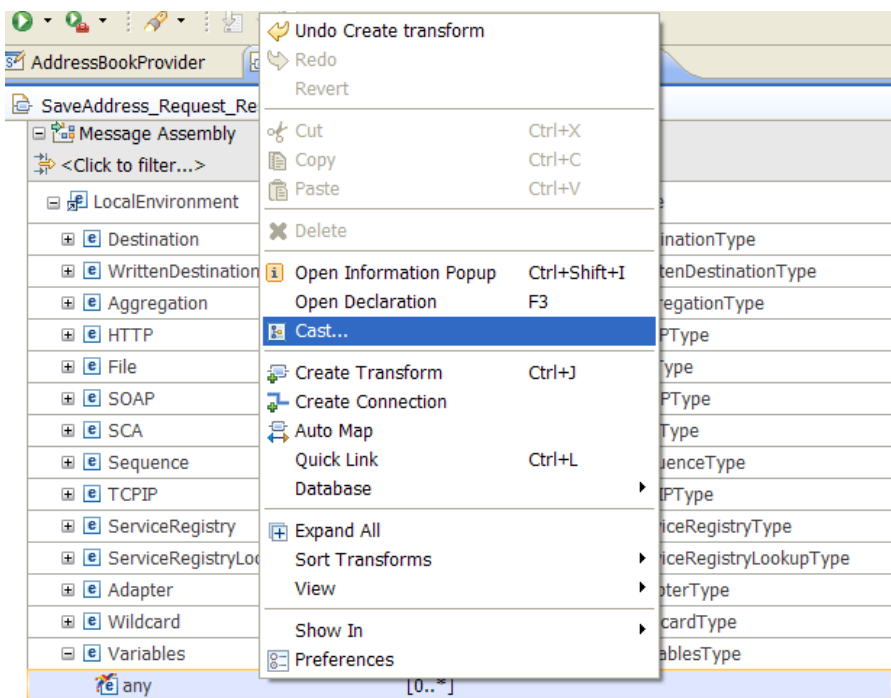
- The file **LEVariablesFolderStructure.xsd** opens in a new tab where you use the XML Schema editor to define your variables and their types.

In our example, we define the following schema:

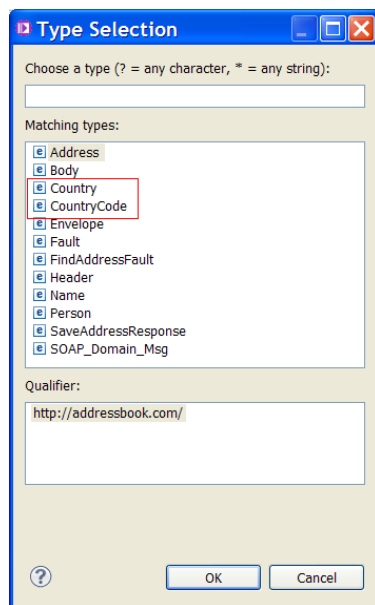
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Country" type="xsd:string"/>
  <xsd:element name="CountryCode" type="xsd:integer"/>
</xsd:schema>
```

Note: In our example, the nodes reading these elements require them to not be namespaced. For this reason, the schema is also defined without a namespace declaration.

3. Use the **Cast** function to define the local environment variables in the message map so they are visible under the **Variables** folder in the map. Complete the following steps to cast the **any** element to a variable and its type in the output local environment tree:
 - Right-click the **any** element, and then select **Cast**.



- In the Type Selection window, select **Country** and then click **OK**.



Results

You now have defined one local environment variable that can be used by other nodes in your message flow for routing or filtering.

You can see the element **Country** under the local environment **Variables** folder in the message map.

Variables	[0..1] _LocalEnvironmentVariablesType
any	[0..*]
Country	[0..*] string

What to do next

Configure the message map to include the SOAP message. For more information, see “Configuring the message map to include the SOAP message.”

Configuring the message map to include the SOAP message

In IBM Integration Bus, a SOAP message is described by a generic model that includes the SOAP *Envelope* and optionally *Attachments*. You define your SOAP message parts in a message map by using the **Cast** function.

About this task

A SOAP message consists of an *Envelope* and optionally *Attachments*. The envelope contains a SOAP header and a SOAP body. A SOAP body can include SOAP faults.

In IBM Integration Bus, when you use SOAP nodes, a SOAP message is described by a generic model. For more information, see SOAP tree overview.

In addition to the standard SOAP parts, the SOAP message generic model includes a *Context* part that includes contextual information about the current SOAP message being processed. This is the only part in a message map whose structure is included automatically. You must define the other SOAP message parts manually by using the **Cast** function.

The following table compares the SOAP message structure with the IBM Integration Bus SOAP message generic model:

Table 23. Comparison between the SOAP message structure and the IBM Integration Bus SOAP message representation

Standard SOAP message parts	Status	IBM Integration Bus SOAP message parts	IBM Integration Bus Status
		Context	Required
SOAP header (part of the SOAP envelope)	Optional	Header (part of the SOAP_Domain_Msg)	Optional
SOAP body (part of the SOAP envelope)	Required	Body (part of the SOAP_Domain_Msg)	Required
SOAP faults (part of the SOAP body)	Optional	Fault (part of the Body)	Optional
SOAP Attachments	Optional	Attachment (part of the SOAP_Domain_Msg)	Optional

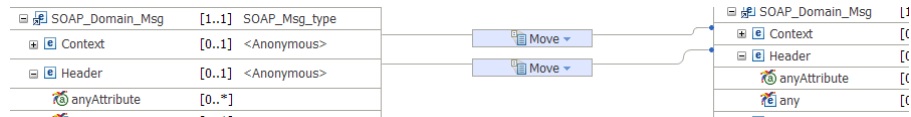
In this scenario, you will learn how to configure your message map to map the standard SOAP message parts which make up the **SOAP_Domain_Msg**.

Procedure

Complete the following steps to configure the **SOAP_Domain_Msg** when the Mapping node is connected directly from a SOAPInput node with no SOAPExtract node:

1. Define a **Move** transform between the input **Context** object and the output **Context** object.
2. Define a **Move** transform between the input **Header** object and the output **Header** object.

The following figure shows the message map after you define a **Move** transform to copy the **Header**.



The SOAP Header element contains application-specific information, including attributes that define how you should process the SOAP message.

3. Define the transformation for the **Body** object.

You define SOAP body parts by using the **Cast** function. You can cast attributes and other body parts. Then, define transforms between the input elements and the output elements in each body part that you have added.

Complete the following steps to define the SOAP body parts and their transformations:

- a. Cast the SOAP body *xsd:any* element into a specific type. For more information, see “Casting the SOAP body into a specific type” on page 266.
- b. Cast a SOAP body base type element to a derived type element. A derived type element is also known as an extension type element. For more information, see “Configuring derived types in the SOAP body” on page 268.

In a message map, you cast a base type to a derived type or extension type so that you can define transformations between subtypes of a data type. For example, addresses are represented differently for different countries. You might want to map addresses from different countries into a common complex structure for addresses.

- c. Create and configure the **If**, **Else if**, and **Else** transform to control the flow of the mapping between elements defined as a specific or a derived type in the input and output message assembly by setting conditions. For more information, see “Configuring an If, Else if, and Else transform in a message map” on page 272.
4. Define a **Move** transform between the input **Attachment** object and the output **Attachment** object.

Results

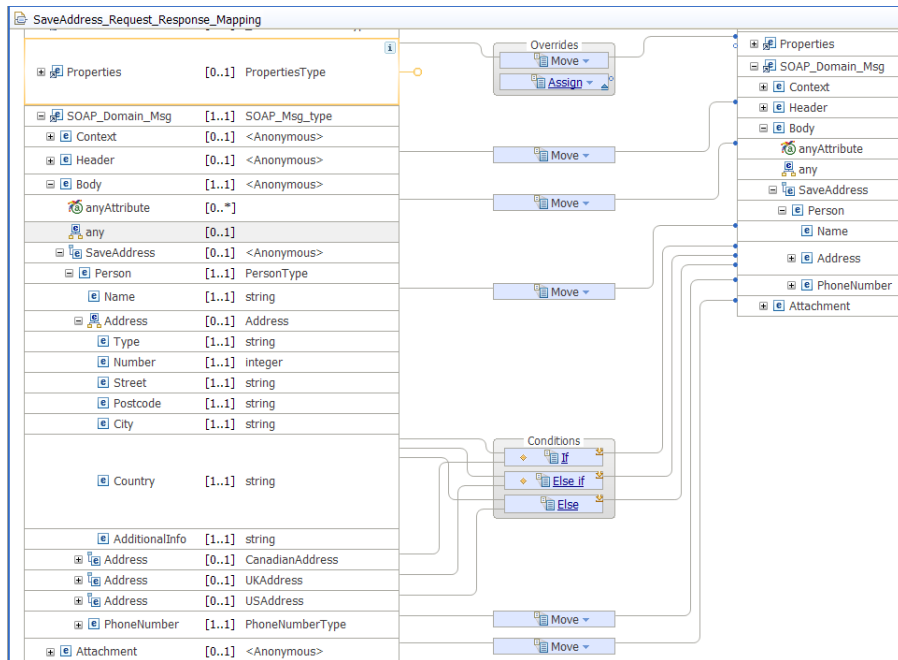
You have configured a message map that transforms a SOAP message.

When you use an **If**, **Else if**, and **Else** transform between your **SOAP_Domain_Msg** input object and **SOAP_Domain_Msg** output object, you must manually configure each element in the **SOAP_Domain_Msg**. You must map each element in the **SOAP_Domain_Msg** input object to the corresponding output object so that you do not lose the information of the element.

Note: Elements that are part of the input object and do not have a transform defined to an output object are deleted from the output structure and their value is lost.

You now have a message map that transforms address data, based on the country of the address. The message map contains a nested map that uses the **If**, **Else if**, and **Else** transform that you defined.

The following figure shows the message map after you complete the previous steps:



What to do next

You have successfully completed the scenario. Your map is now ready to use.

Casting the SOAP body into a specific type

You use the **Cast** function to redefine the Body of the input and output SOAP body that have a type *xsd:any* element in the message map. These elements are also known as wildcard elements.

Before you begin

Create a message map. For more information, see “Creating a message map to transform SOAP messages” on page 250.

About this task

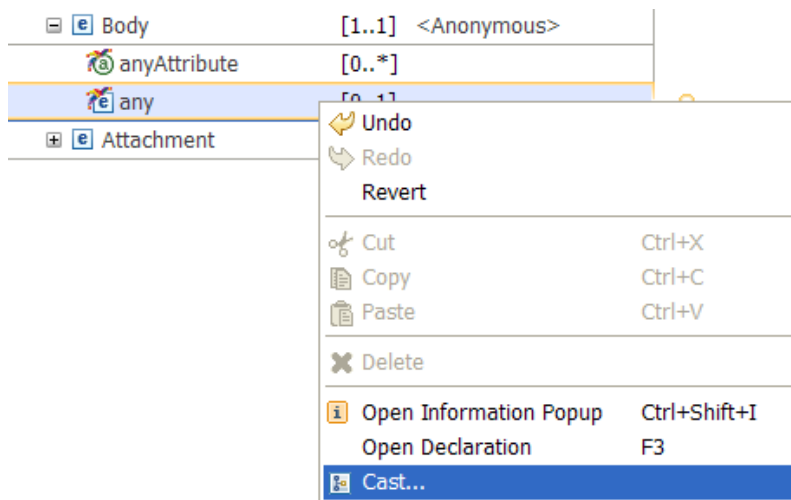
When you transform a SOAP message, you cast the Body wildcard on the input side into the type that is defined in the WSDL for the request of the SOAP operation. On the output side, you cast the Body wildcard to the type of the response message for the SOAP operation.

The scenario demonstrates how to cast the Body section. You can repeat the steps to cast SOAP Body attributes.

Procedure

To cast the SOAP body described as **any** in the message map, complete the following steps:

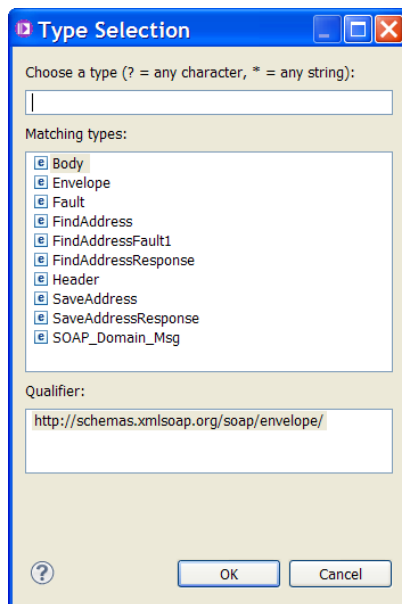
1. Right-click the element **any** located in the section of your **SOAP_Domain_Msg** where you want to specify a type, and then select **Cast**.
Right-click **Body**, and then select **Cast**.



2. In the Type Selection window, select a type.

The Type Selection window displays all the specific types that are available for selection. These types include the input and output elements defined in the WSDL file that describes your SOAP message.

Select **SaveAddress**, and then click **OK**.



Results

When you cast the element **any** of the **SOAP_Domain_Msg** Body, you add the complex element **SaveAddress** to the message map.

SaveAddress_Request_Response_Mapping		
SaveAddress_Request_Response_Mapping		
Message Assembly		SOAP_Domain_Msg
<Click to filter...>		
Properties	[0..1]	PropertiesType
SOAP_Domain_Msg	[1..1]	SOAP_Msg_type
Context	[0..1]	<Anonymous>
Header	[0..1]	<Anonymous>
Body	[1..1]	<Anonymous>
anyAttribute	[0..*]	
any	[0..1]	
SaveAddress	[0..1]	<Anonymous>
Person	[1..1]	PersonType
Name	[1..1]	string
Address	[1..1]	Address
PhoneNumber	[1..1]	PhoneNumberType
Attachment	[0..1]	<Anonymous>

What to do next

1. Repeat the previous steps to cast the output SOAP body as **SaveAddress** into your message map.
2. Configure derived types in the SOAP body. For more information, see “Configuring derived types in the SOAP body.”

Configuring derived types in the SOAP body

In a message map, you cast a base type to a derived type or extension type so that you can define transformations between subtypes of a data type.

Before you begin

Cast the SOAP body element **SaveAddress** in your message map. Complete the steps outlined in “Casting the SOAP body into a specific type” on page 266.

Your message map input Message Assembly should look like the one in the following figure:

SaveAddress_Request_Response_Mapping		
SaveAddress_Request_Response_Mapping		
Message Assembly		SOAP_Domain_Msg
<Click to filter...>		
Properties	[0..1]	PropertiesType
SOAP_Domain_Msg	[1..1]	SOAP_Msg_type
Context	[0..1]	<Anonymous>
Header	[0..1]	<Anonymous>
Body	[1..1]	<Anonymous>
anyAttribute	[0..*]	
any	[0..1]	
SaveAddress	[0..1]	<Anonymous>
Person	[1..1]	PersonType
Name	[1..1]	string
Address	[1..1]	Address
PhoneNumber	[1..1]	PhoneNumberType
Attachment	[0..1]	<Anonymous>

About this task

A *derived type* is a datatype that is related to another datatype known as the base type or supertype.

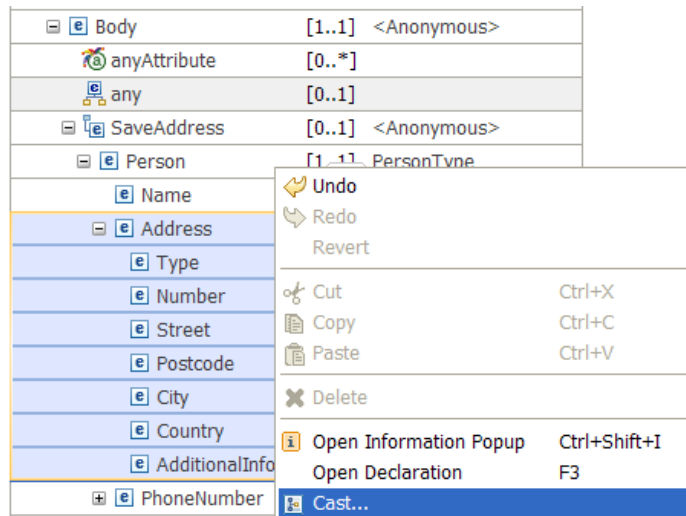
In the scenario, **Address** is the base type, and **USAddress**, **CanadianAddress**, and **UKAddress** are derived types of **Address**.

Requests to save an address can come from Canada, the US, or the UK. Addresses are represented differently for each country, for example, in Canada the address includes the province. The AddressBook service stores all addresses in a single location using a common complex structure for addresses.

Procedure

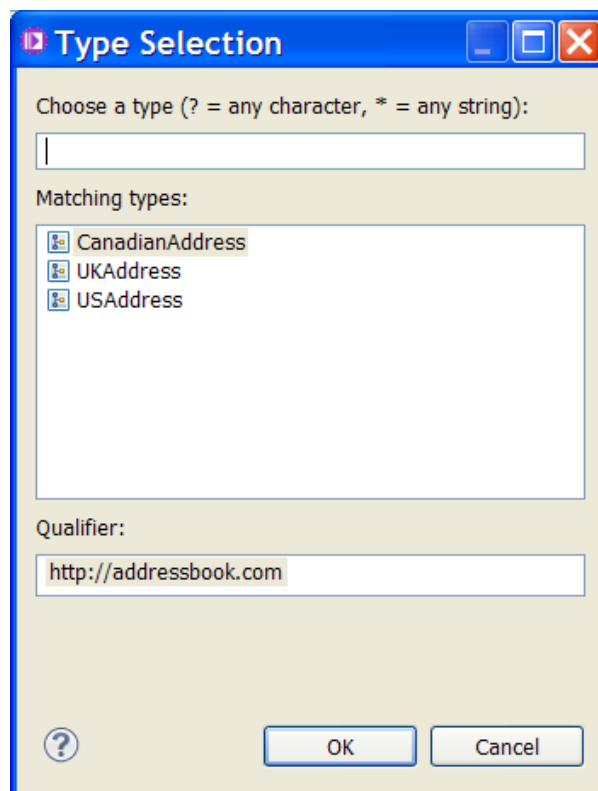
Complete the following steps to cast the **Address** base type to its derived types, so that addresses from different countries can be mapped into a common complex address type:

1. Select **Address**.
2. Right-click **Address**, and then select **Cast**.

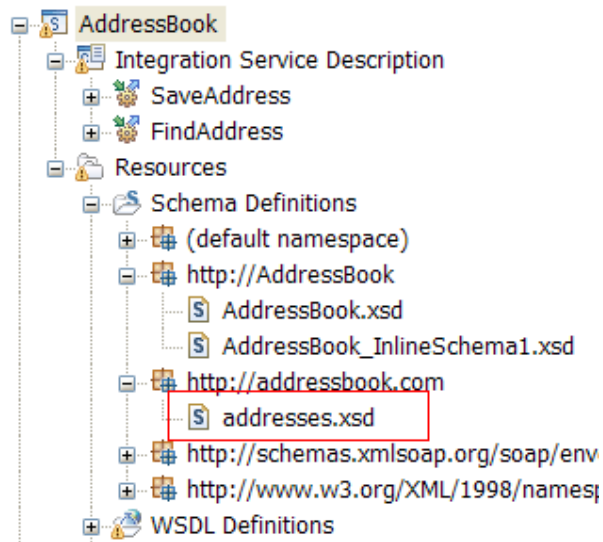


3. In the Type Selection window, choose a matching type, and then select **OK**.
The options available correspond to specific address types in the schema model that have been modeled using **Address** as the base type.

The following figure shows the Type Selection window that you get:



- a. Optional: Check the schema model in the Application Development view. Select the **AddressBook** service located under **Resources > Schema definitions > http://addressbook.com**.



Results

The message map contains two entries for Address. One corresponds to the base type **Address**. The other entry corresponds to an Address with the derived type **CanadianAddress**.

SaveAddress	[0..1]	<Anonymous>
Person	[1..1]	PersonType
Name	[1..1]	string
Address	[0..1]	Address
Type	[1..1]	string
Number	[1..1]	integer
Street	[1..1]	string
Postcode	[1..1]	string
City	[1..1]	string
Country	[1..1]	string
AdditionalInfo	[1..1]	string
Address	[0..1]	CanadianAddress
Type	[1..1]	string
Number	[1..1]	integer
Street	[1..1]	string
Postcode	[1..1]	string
City	[1..1]	string
Country	[1..1]	string
AdditionalInfo	[1..1]	string
province	[1..1]	string
postcode	[1..1]	string
PhoneNumber	[1..1]	PhoneNumberType

What to do next

1. Repeat the steps to add the following derived types: **UKAddress**, and **USAddress**. The following figure shows your message map input object after you add all the derived addresses.

SaveAddress_Request_Response_Mapping		
+	Properties	[0..1] PropertiesType
-	SOAP_Domain_Msg	[1..1] SOAP_Msg_type
+	Context	[0..1] <Anonymous>
-	Header	[0..1] <Anonymous>
	anyAttribute	[0..*]
	any	[0..*]
-	Body	[1..1] <Anonymous>
	anyAttribute	[0..*]
	any	[0..1]
-	SaveAddress	[0..1] <Anonymous>
-	Person	[1..1] PersonType
	Name	[1..1] string
	Address	[0..1] Address
	Address	[0..1] CanadianAddress
	Address	[0..1] UKAddress
	Address	[0..1] USAddress
	PhoneNumber	[1..1] PhoneNumberType
+	Attachment	[0..1] <Anonymous>

2. Define a conditional transform between elements of the SOAP body. For more information, see “Configuring an If, Else if, and Else transform in a message map.”

Configuring an If, Else if, and Else transform in a message map

You use the If, Else if, and Else transform to set conditions that control the flow of the data mapping between SOAP body elements defined as a specific or a derived type in the input and output message assembly.

Before you begin

Complete the following steps:

1. Cast the input and output message assembly body element **any** to **SaveAddress**. For more information, see “Casting the SOAP body into a specific type” on page 266.
2. Cast the **Address** base type defined in the input and output message assembly body to the **CanadianAddress** derived type, the **UKAddress** derived type, and the **USAddress** derived type. For more information, see “Configuring derived types in the SOAP body” on page 268.

About this task

You use an If, Else if, and Else transform to map multiple derived address types such as the **CanadianAddress** to the base address type **Address**.

In the scenario, each address contains a country-specific element:

- In a **CanadianAddress**, each address includes the element **Province**.
- In a **UKAddress**, each address includes the element **County**.
- In a **USAddress**, each address includes the element **State**.

The base address type **Address** includes an element named **AdditionalInfo**. You use this element to store additional information that does not have a corresponding element in the base address type.

Procedure

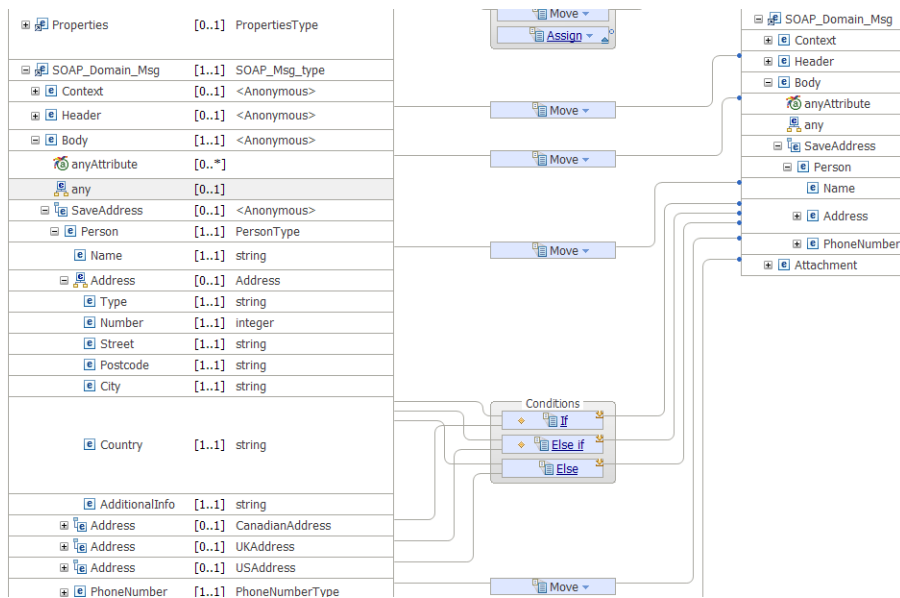
Complete the following steps to map a derived type to a base type by using an If, Else if, and Else transform in the scenario:

1. Create and configure the **If** condition of the If, Else if, and Else transform. For more information, see “Configuring the **If** condition in an If, Else if, and Else transform” on page 274.
2. Optional: Create and configure the **Else If** condition of the If, Else if, and Else transform. For more information, see “Configuring the **Else If** condition in an If, Else if, and Else transform” on page 277.
3. Create and configure the **Else** condition of the If, Else if, and Else transform. For more information, see “Configuring the **Else** condition in an If, Else if, and Else transform” on page 280.
4. Optional: Change the order in which the conditions you have defined are evaluated by the mapping engine. For more information, see “Changing the order of the conditions in an If, Else if, and Else transform” on page 282.
5. For each condition defined in the If, Else if, and Else transform, configure the nested map associated with the condition.
 - To configure a nested map manually, see “Configuring a nested map associated with an If, Else if, and Else transform condition manually” on page 283.
 - To configure a nested map automatically, see “Configuring a nested map associated with an If, Else if, and Else transform condition by using automap” on page 284.

Results

You now have a message map that transforms address data, based on the country of the address. The message map contains a nested map that uses the If, Else if, and Else transform that you defined.

The following figure shows the message map after you complete the previous steps:



What to do next

You have now completed all steps necessary to transform the sample SOAP message by using a message map that uses an If, Else if, and Else transform.

Configuring the If condition in an If, Else if, and Else transform:

You can use an If, Else if, and Else transform to control the flow of the data mapping between elements defined as a specific or a derived type in the input and output message assembly by setting conditions. To configure the If condition, you must connect an input element to an output element and select the core transform **If**.

Procedure

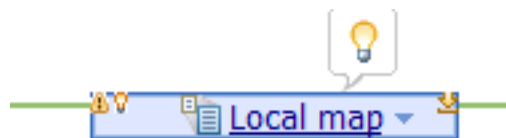
Complete the following steps to create and configure the **If** condition of an If, Else if, and Else transform:

1. Connect the element **Country** in the input message assembly object located under **SOAP_Domain_Msg > Body > SaveAddress > Person > Address** to the element **Address** in the output message assembly object located under **SOAP_Domain_Msg > Body > SaveAddress > Person**.

A **Local map** transform is automatically created.

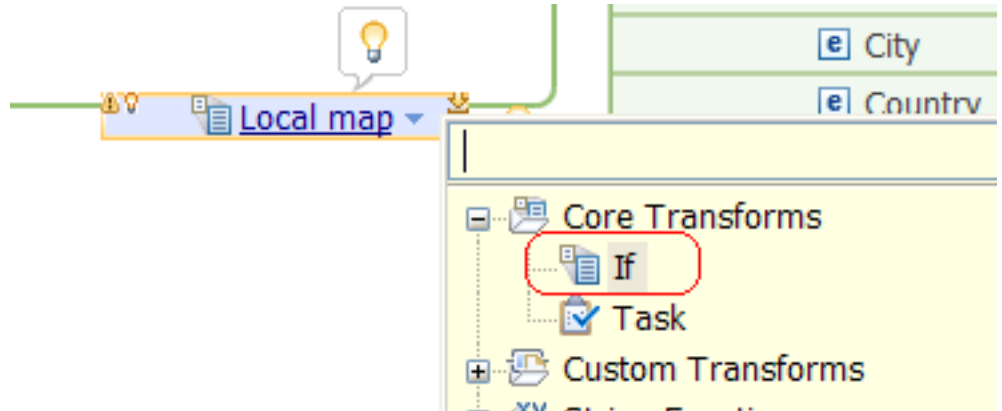
2. Connect the **Local map** condition to the output element **Address**.
3. Change the **Local map** transform to an **If** transform by selecting the arrow facing down that is located on the right hand side of the **Local map** transform.
 - a. Left-click the arrow located to the right of the **Local map** transform.

The following figure shows graphically how to select the **If** transform.



- b. Select the **If** transform located within **Core Transforms**.

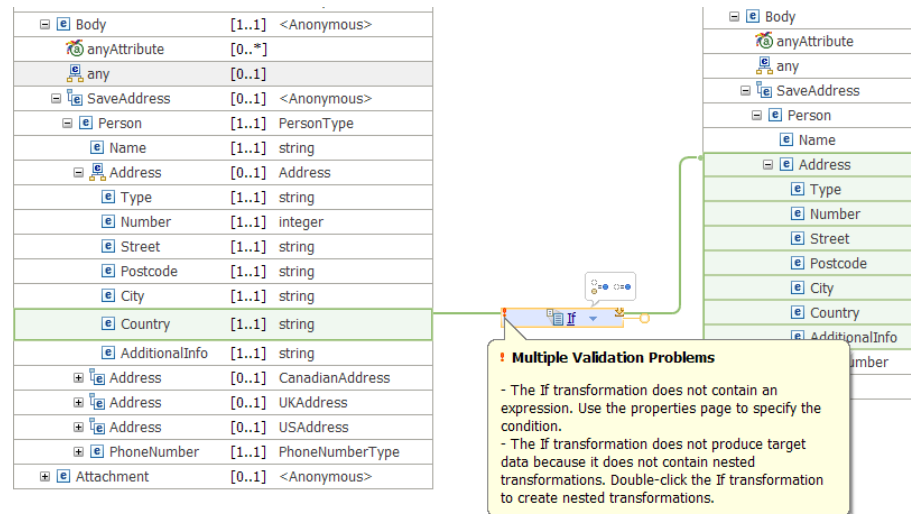
The following figure shows graphically the list of core transforms available:



You get an **If** condition with a red exclamation mark connected to two input elements and one output element.

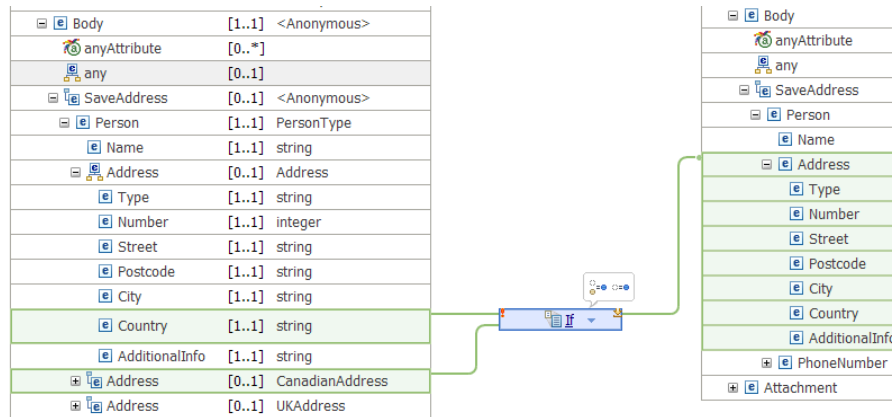
Note: You will resolve these errors by completing the scenario

The red exclamation mark on the left hand side of the **If** condition highlights multiple validation problems. One of the errors indicates that the **If** condition does not contain an expression. The second error informs you that you must define transformations for all the elements within the nested map associated with the **If** condition. This nested map is the map that you use to define how an address with a derived type **CanadianAddress** is mapped to the base address type **Address**.



4. Connect the address that has the **CanadianAddress** as its derived type to the **If** transform.

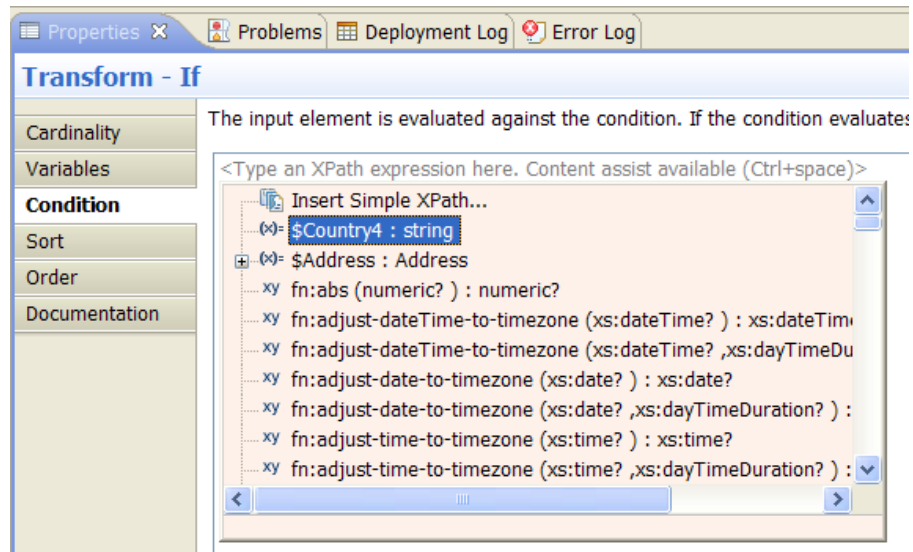
The following figure shows the message map after you create the connection:



5. Select the **If** condition, and then define the expression in the **Condition** tab under the **Transform - If** properties. Complete the following steps:

a. Press **Ctrl + spacebar** to obtain the list of elements.

The following figure shows the elements available for selection in the scenario:



Note: Although you can enter the XPATH expression directly, beware that depending on the steps you take to create your integration solution, the variable names that are generated are different from the element name in the schema file. The element name has an ID concatenated at the end of the name that is defined automatically by the tool.

b. Select an element and double-click on it.

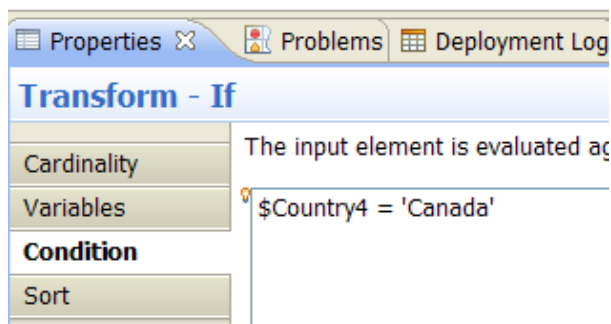
c. Define the XPATH expression related to that variable.

d. Repeat the following steps if your XPATH expression includes more than one input element.

The condition is an XPath 2.0 expression, that you can define directly, or you can create through the XPath expression builder by clicking **Edit**.

In the scenario, if you authored the message flow yourself, the expression will be similar but not exactly like `$Country4 = 'Canada'`.

The following figure shows the properties tab for the **If** transform:



Results

You now have defined and configured the **If** condition.

What to do next

Define the **Else If** condition of the **If** transform. For more information, see “Configuring the **Else If** condition in an **If**, **Else if**, and **Else** transform.”

Configuring the **Else If** condition in an **If**, **Else if**, and **Else** transform:

Create and configure an **Else If** condition after you define the **If** condition.

Before you begin

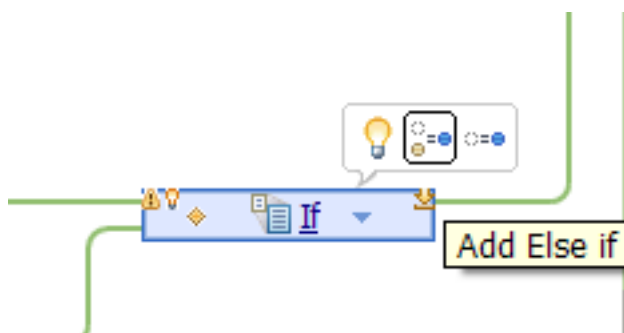
Define the **If** condition of the **If**, **Else if**, and **Else** transform. For more information, see “Configuring the **If** condition in an **If**, **Else if**, and **Else** transform” on page 274.

Procedure


Complete the following steps to create and configure the **Else If** condition of an **If**, **Else if**, and **Else** transform:

1. Select the diamond symbol located to the left of the **If** transform. The **Add Else If** option and the **Add Else** conditions appear to the right hand side of a light bulb in a pop up on top of the **If** transform.

This diamond symbol appears after you set the **If** condition.



2. Select **Add Else If** to add another address with a derived type of **UKAddress**.

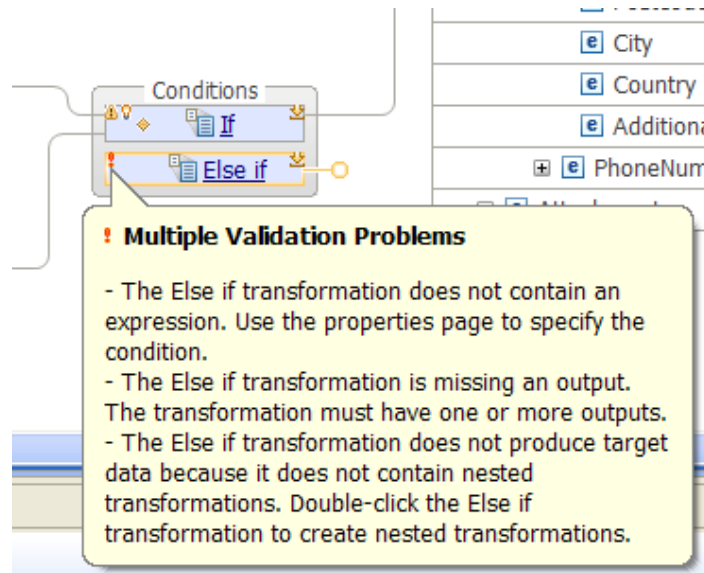
To add an address, select the **Add Else If** icon .

Note: If you have more derived types, repeat this step for each additional address that you have defined.

When you select the **Add Else If** condition, the mapping engine creates a **Conditions** box that includes the **If** condition and the **Else If** condition of the **If, Else if, and Else** transform that you are configuring.

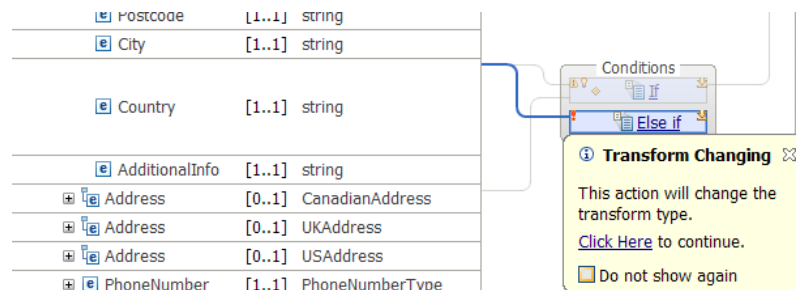
You get an **Else If** condition with a red exclamation mark.

The red exclamation mark on the left hand side of the **If** condition highlights multiple validation problems which you will resolve by completing the scenario.



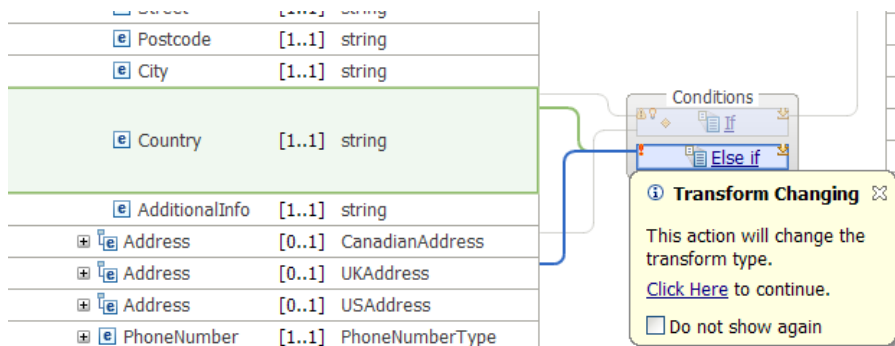
3. Connect the element **Country** in the input message assembly object located under **SOAP_Domain_Msg > Body > SaveAddress > Person > Address** to the **Else If** condition.

A connection is created between the element **Country** and the **Else If** condition. A window opens informing you that by creating this connection, the transform type changes. Click **Click here** to continue.



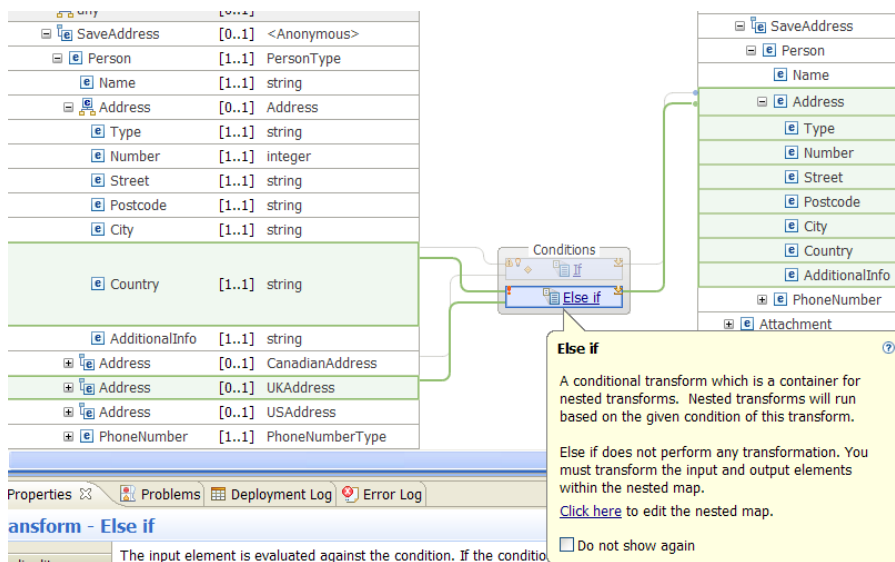
4. Connect the element **Address** with a derived type of **UKAddress** to the **Else If** condition.

A connection is created between the element **Country** and the **Else If** condition. A window opens informing you that by creating this connection, the transform type changes. Click **Click here** to continue.



5. Connect the **Else If** condition to the output element **Address**.

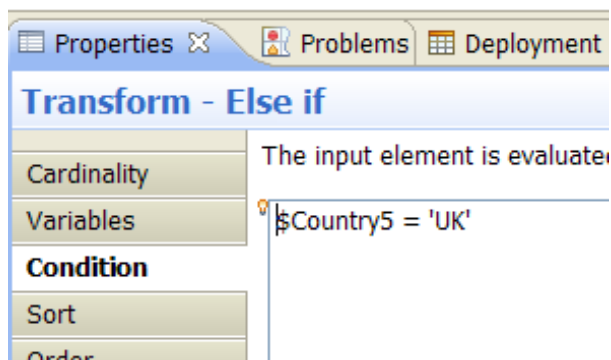
When you define this connection to the **Else If** condition, a pop up message displays to explain that you must transform the input and output elements within the nested map that is created where the input object is an address of type **UKAddress**, and the output object is an address of type **Address**. You can click **Click Here** to access the nested map, or you can click anywhere on the message map to continue configuring the **Else If** condition.



6. Select the **Else If** condition, and then define the following expression in the **If** transform properties: `$Country5 = 'UK'`.

The condition is an XPath 2.0 expression, that you can define directly, or you can create through the XPath expression builder by clicking **Edit**.

The following figure shows the properties tab for the **If** transform:



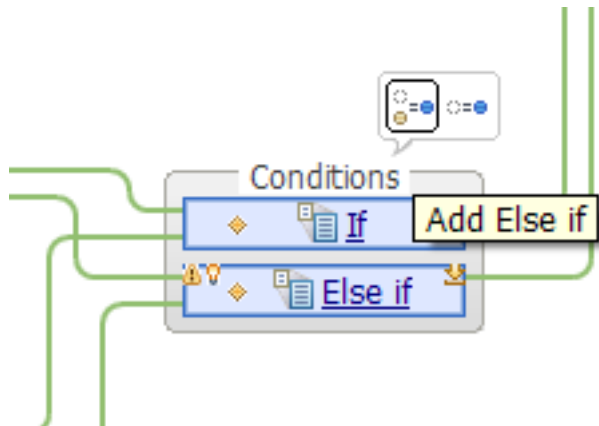
Results

After you define the **Else If** condition, the message map contains a **Conditions** container with two conditions.

What to do next

If there are other conditions, define **Else If** conditions for each one.

Note: To add more **Else If** conditions, select **Conditions**, and then **Add Else If**.



Define the **Else** condition of the If, Else if, and Else transform. For more information, see “Configuring the **Else** condition in an If, Else if, and Else transform.”

Configuring the Else condition in an If, Else if, and Else transform:

Create and configure an **Else** condition after you define the **If** condition and optionally more **Else If** conditions. The If, Else if, and Else always finishes with an **Else** condition. This is the condition that runs when none of the other conditions are true.

Before you begin

1. Define the **If** condition of the If, Else if, and Else transform. For more information, see “Configuring the **If** condition in an If, Else if, and Else transform” on page 274.
2. Define the **Else If** conditions of the If, Else if, and Else transform. For more information, see “Configuring the **Else If** condition in an If, Else if, and Else transform” on page 277.

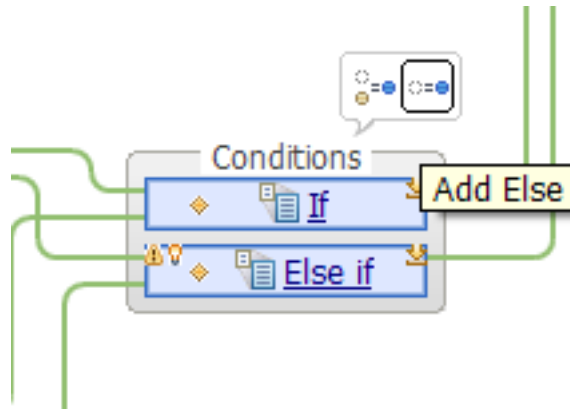
About this task

The **Else** condition is the path followed by addresses whose country is different from Canada or UK. In the scenario, it is the path that evaluates to true when a US address needs to be mapped.

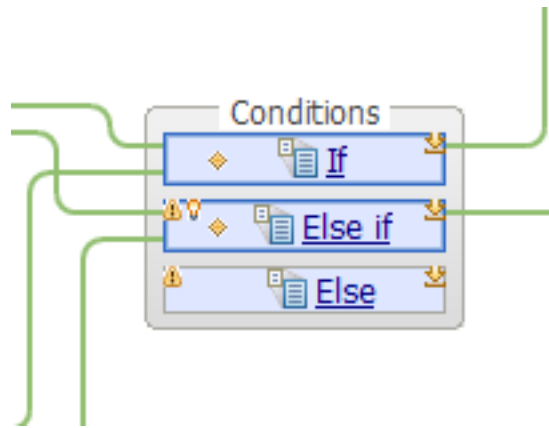
Procedure

Complete the following steps to create and configure the **Else** condition of an If, Else if, and Else transform:

1. Left-click **Conditions**, and then select **Add Else**.

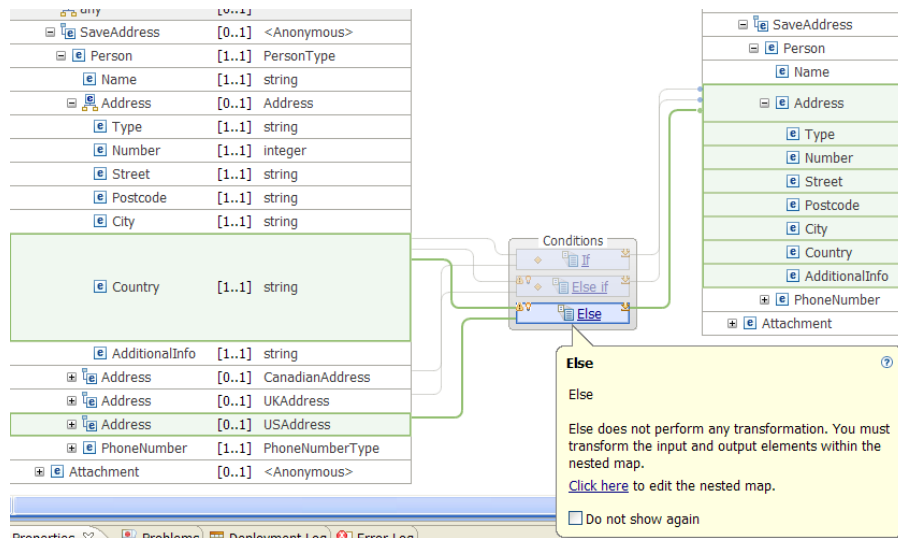


An **Else** condition is included within **Conditions**.



2. Connect the element **Country** in the input message assembly object located under **SOAP_Domain_Msg > Body > SaveAddress > Person > Address** to the **Else** condition.
3. Connect the element **Address** with a derived type of **USAddress** to the **Else** condition.

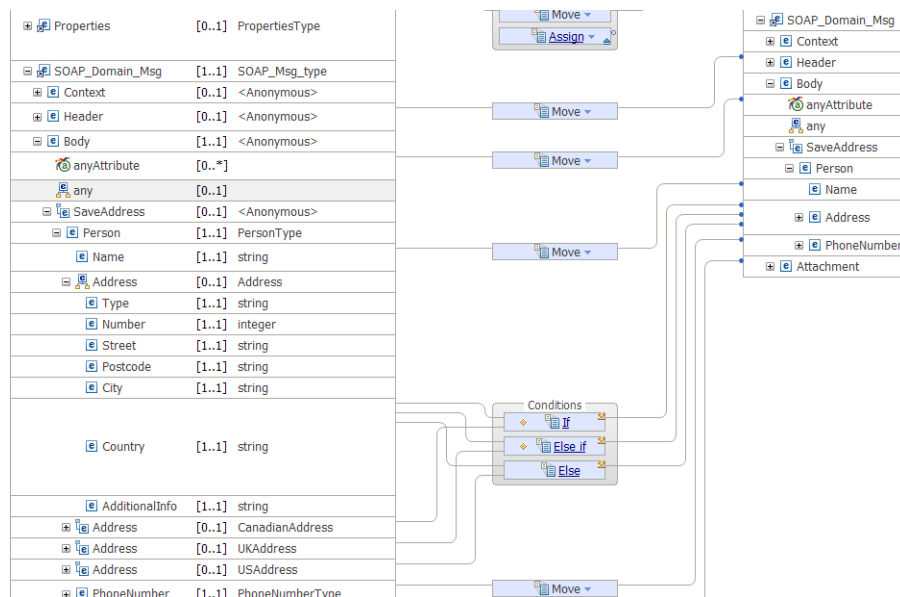
When you define the connection to the **Else** condition, a message displays to explain that you must transform the input and output elements within the nested map that is created where the input object is an address of type **USAddress**, and the output object is an address of type **Address**. You can click **Click Here** to access the nested map, or you can click anywhere on the message map to continue configuring the **Else** condition.



4. Connect the **Else** condition to the output element **Address**.

Results

A message map with three conditions is defined.



What to do next

Continue configuring the SOAP body. Return to “Configuring the message map to include the SOAP message” on page 264.

Changing the order of the conditions in an If, Else if, and Else transform:

You can change the order in which the mapping engine evaluates the conditions defined in an If, Else if, and Else transform.

Before you begin

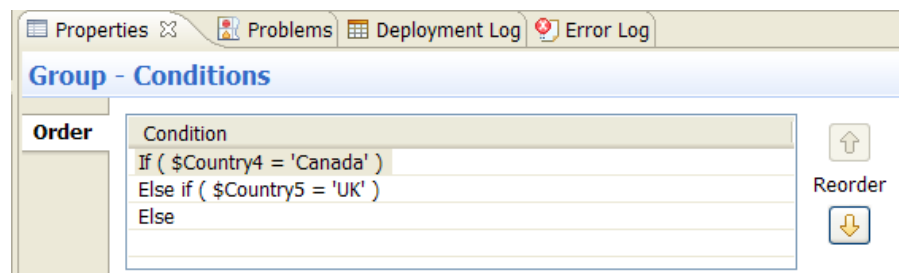
Define and configure the condition expressions for an If, Else if, and Else transform. For more information, see *“Configuring an If, Else if, and Else transform in a message map”* on page 272

Procedure

Complete the following steps:

1. Select **Conditions**. This is the container that includes the **If** condition, the **Else If** conditions, and the **Else** condition.
2. In the **Properties** tab, use the **Reorder** arrows to change the priority of a condition. The conditions and its expressions are updated automatically to reflect your changes.

The following figure shows the **Properties** tab:



Example

For example, to change the **If** condition so it becomes the **Else If** expression evaluated, select the row with the **If** condition, and then select **Reorder**.

Configuring a nested map associated with an If, Else if, and Else transform condition manually:

You can configure a nested map associated with an If, Else if, and Else transform manually by defining transforms between input and output elements.

Before you begin

Define and configure the conditional expressions for an If, Else if, and Else transform. For more information, see *“Configuring an If, Else if, and Else transform in a message map”* on page 272

About this task

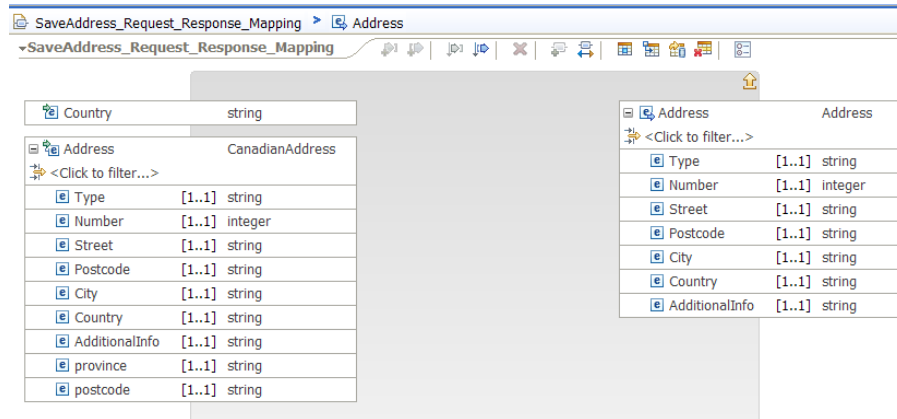
Each condition in an If, Else if, and Else transform has a nested map associated, which is used by the mapping engine to apply the transforms between the input object and the output object when the associated condition evaluates to true.

Procedure

For the **If** condition, complete the following steps to configure the nested map associated with it:

1. In the message map, double-click the **If** condition.

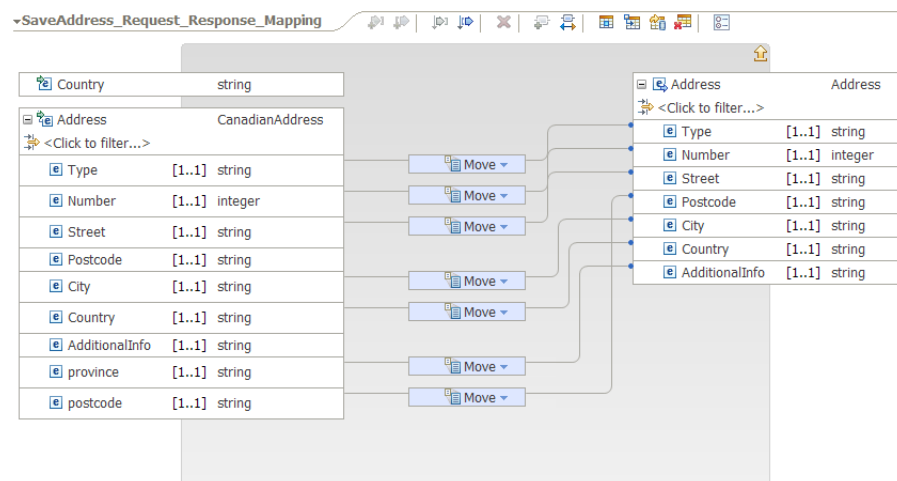
The nested map opens. The following figure shows the nested map:



2. Define transforms for each element in the input object that you want to maintain in the output object.

In the scenario, we have defined a **Move** transform between each input element and each output element. Note that the element **province** is mapped into the output element **AdditionalInfo**.

You get a nested map that transforms input elements into output elements.



What to do next

Repeat the steps to configure each nested map associated with an If, Else if, and Else transform condition.

Configuring a nested map associated with an If, Else if, and Else transform condition by using automap:

You can configure a nested map associated with an If, Else if, and Else transform automatically by using automap.

Before you begin

Define and configure the condition expressions for an If, Else if, and Else transform. For more information, see *“Configuring an If, Else if, and Else transform in a message map”* on page 272

About this task

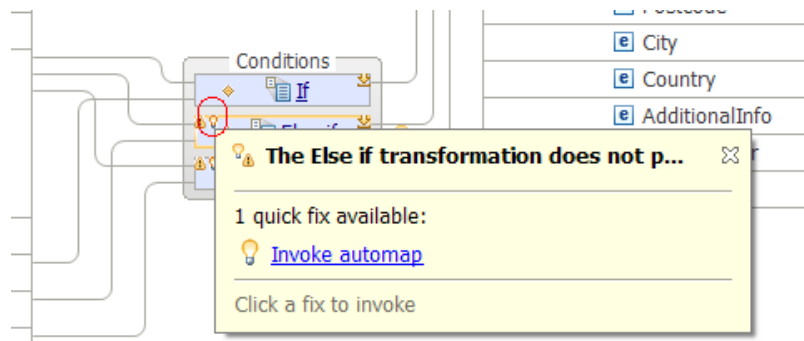
Each condition in an If, Else if, and Else transform has a nested map associated, which is used by the mapping engine to apply the transforms between the input object and the output object when the associated condition evaluates to true.

Procedure

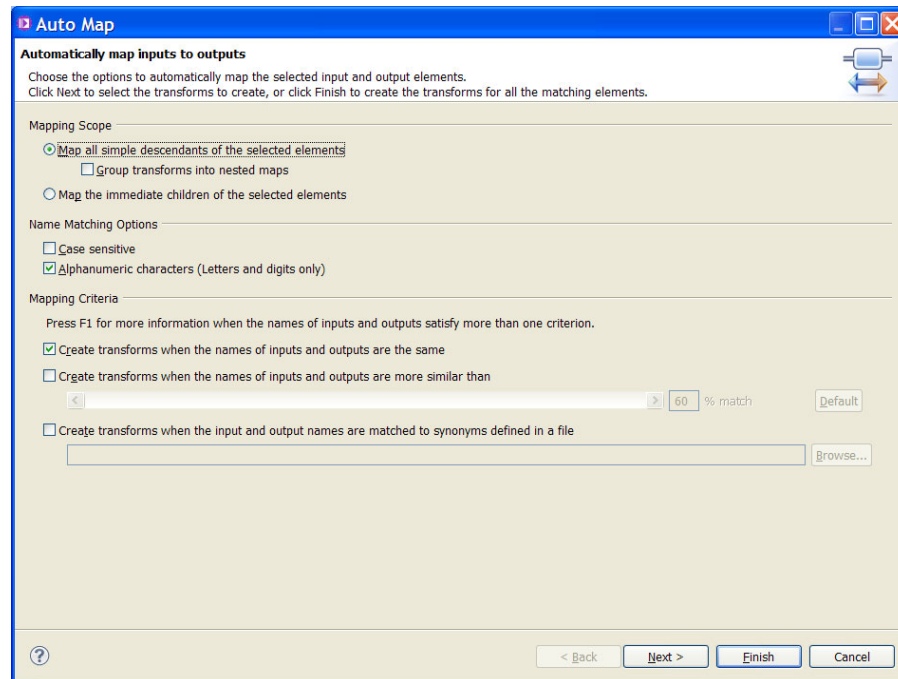
For the **Else If** condition, complete the following steps to configure the nested map associated with it:

1. In the message map, click the light bulb located on the top left corner of the **Else If** condition. A pop up opens. Select **Invoke automap**.

The following figure shows the pop up:

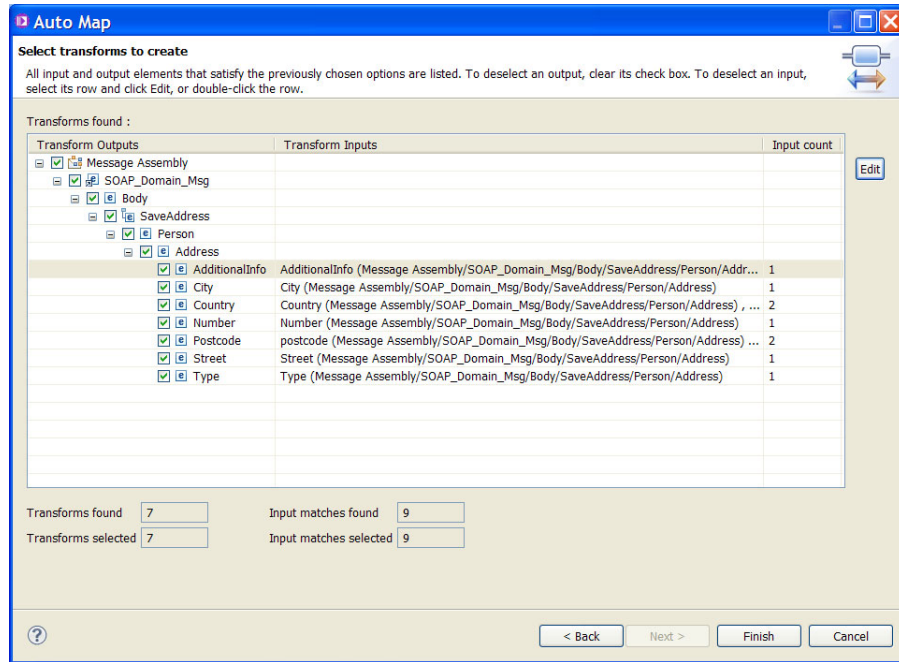


The Auto Map window opens.



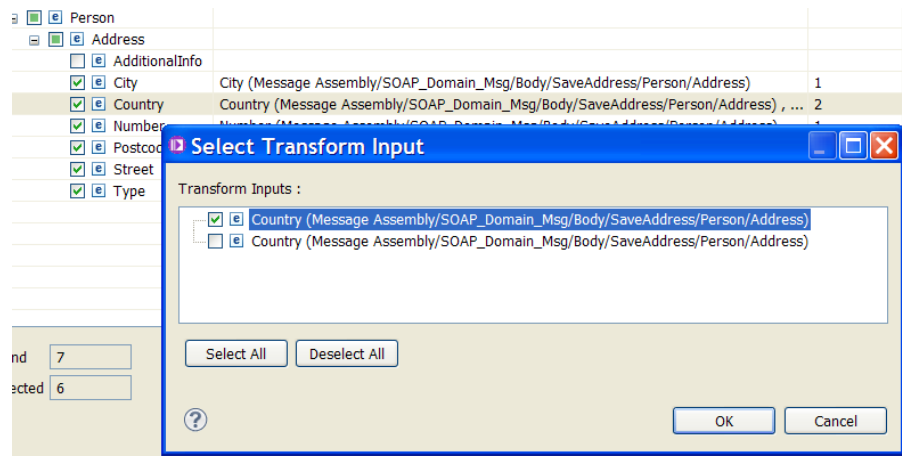
2. Click **Next**. The Select transforms to create window opens.

The Select transforms to create window displays the proposed transformation for each output element in the nested map. It also specifies the input count per output element so that you know how many input elements are available.

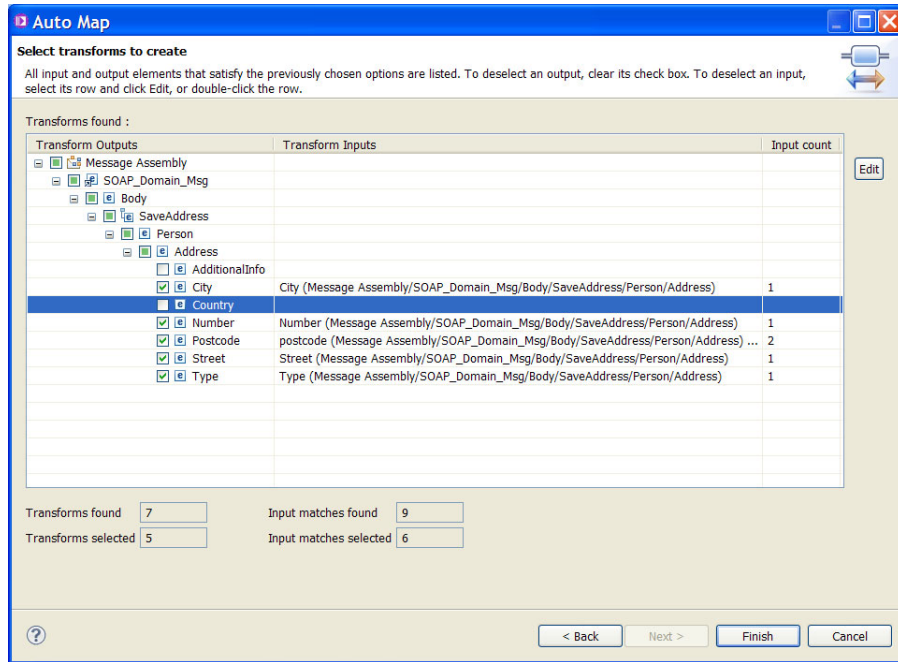


- Optional: If you want to define your own custom transformations for any of these elements, clear the relevant check boxes. For example, clear **AdditionalInfo**.
- For elements with an **Input count** greater than one, double-click the element, and then select the option that you want to apply for the transformation of that element.

For example, the element **Country** has two possible input elements that you can use as the output value. Choose one.



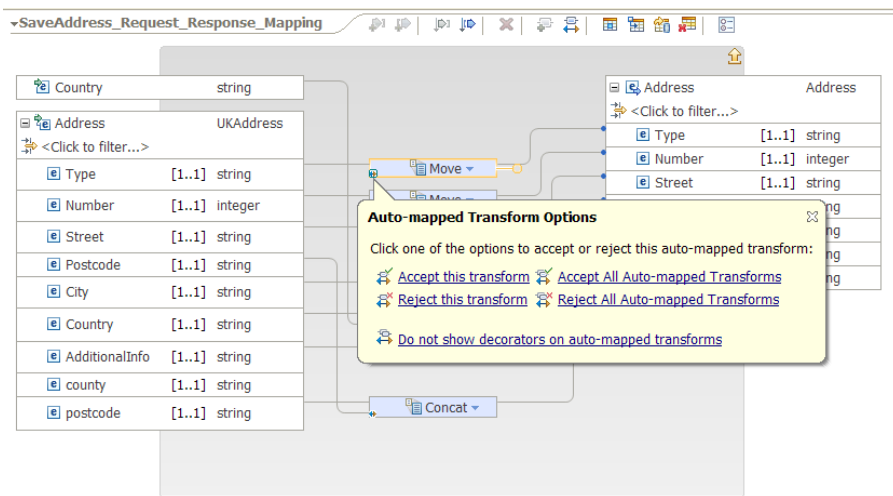
The following figure shows what the **Auto Map** looks like after you have reviewed and configured the proposed transformations:



5. Click **Finish**. Click the icon located in the left side of any of the transforms, and then select **Accept All Auto-mapped Transforms**.

Transforms are defined between the input and output elements based on the options that you selected.

The following figure shows the proposed auto-mapped transform options:



When you select **Accept All Auto-mapped Transforms**, you confirm that you the proposed transforms are correct.

What to do next

Repeat the steps to configure each nested map associated with an If, Else if, and Else transform condition by using automap.

Chapter 28. Scenario: Using a message map to enrich a message with data from a database

You can use a message map to enhance an existing message with data from one or more database tables.

This scenario was developed using a set of sample IBM Integration Bus resources. If you want to try the scenario, you can either use your own integration solutions, or set up a copy of the sample resources as described in the scenario.

Introduction to the "Using a message map to enrich a message with data from a database" scenario

This scenario shows how you can enhance a message in IBM Integration Bus by using a message map. The data is available in a database system. The data is stored across multiple database tables. All the tables are located within the same database schema.

About this task

WebSphere Message Broker Version 8.0 introduces graphical data maps. These message maps replace the previous message map format, and are created as .map files. Message maps offer the ability to transform a message without the need to write code, providing a visual image of the transformation, and simplifying its implementation and ongoing maintenance.

WebSphere Message Broker Version 8.0 also introduces the **Select** transform that allows you to enrich a message by accessing data located in an external database system. This feature simplifies the programming model. It eliminates the requirement to use a Database node, a JavaCompute node, a .NETCompute node, or a Compute node to access data located in a database. You can design simpler message flows by using a single Mapping node to complete graphically a message transformation that requires data from an external database system.

Note: You can continue to use maps that were created in versions earlier than WebSphere Message Broker Version 8.0. However, if you need to modify any of your legacy maps, or if you want to use the **Select** transform, you must convert these legacy message maps into .map message maps. For more information about converting maps, see *Converting a message map from a .msgmap file to a .map file*.

Read the following topics to understand the scenario and the concepts the scenario is intended to demonstrate:

- "Context" on page 290
- "Technical solution" on page 291

What to do next

Implement the solution. For more information, see "Implementing the solution" on page 292.

Context

This scenario shows how you can enhance a message in IBM Integration Bus by using a message map. In this scenario, the data is available in an external database system. The data is stored across multiple database tables, all of which are located within the same database schema.

Your company has implemented an AddressBook service that is used by different departments in different countries in your organization. This service allows your employees to obtain a client's mail address or to save a new client's mail address.



The company uses IBM Integration Bus to develop and manage a number of integration solutions that transform and communicate data between source and target systems. In order to make the service reusable by multiple applications, you design an application responsible for the transformation of the different address formats between the requesting application and the AddressBook service. The AddressBook service is a SOAP-based service that stores a new address or returns an address to the user. You use a message map to define how to transform the SOAP message based on the operation that your user requests.

The company uses DB2 Version 9.7 as the external database system that hosts client's details and addresses.

The scenario uses the following database tables:

- *Person*: This table contains an entry per client. The client ID element is used to link information for this client across all tables in the database. The database automatically assigns the ID value when a new record is created. This table contains all clients from all countries.
- *Address*: This table contains an entry per client with the address details.
- *Phone*: This table contains an entry per client with the phone details.

In IBM Integration Bus, you have the following choices to implement a message flow that connects to a database, and retrieves information to enrich the message:

- You can use a Mapping node to graphically connect to a database and retrieve data to use in the node and later on in the message flow.
- You can use a Database node in a message flow to connect to a database and retrieve data that you can use later on in the message flow.
- You can program a Compute node, JavaCompute node, or .NETCompute node to connect to a database and retrieve data to use in the message flow.

This scenario demonstrates how to use a Mapping node to connect to a database, retrieve data from multiple tables, and graphically populate elements in a SOAP message with this information in the IBM Integration Studio.

Technical solution

You can use a message map to enhance an existing message with data from one or more database tables. Data from the database can then be used to enrich, route, and transform messages within IBM Integration Bus.

In IBM Integration Bus, to connect to a database, you must configure the development environment and the IBM Integration Bus runtime environment:

1. To have visibility of the database resources during the development phase, you must connect the IBM Integration Studio to the development database.
2. To enable the deployed map to execute in the run time, you must create a JDBC provider configurable service that defines the connection to the runtime database. This database is normally a different database server from the one you use for development, and the artifacts could be in a different database schema.

To configure the IBM Integration Studio to connect to a database, you must create a database definition file in a data design project, and configure a database connection.

- Data design project: A specialized type of project where you store your database resources.
- Database definition file: A configuration file where you specify the database physical details such as database type and version, and a connection.
- Database connection: Configuration that details the database resources, that is, the schema, the tables, the store procedures, the indexes, and other resources, that you need access to from within your IBM Integration Bus project resources.

To access information stored in a database from resources in a IBM Integration Bus project, you must include a reference to the data design project in your application, service, or Message Broker project.

In IBM Integration Bus, you can use a message map to access information in a database, and then use this information to perform transformations on the message or to enrich a message.

During the design phase, you must complete the following steps in the IBM Integration Studio to access graphically database information in a message map:

1. Add a reference to each database table from where you must retrieve data.
2. Use a **Select** transform to define how to use the database information in the message map. The **Select** transform has embedded a nested map. You must define the transforms in this nested map.
3. Use a **Failure** transform to handle database failures. The **Failure** transform has embedded a nested map. You can define the transforms in this nested map if you wish to provide specialized handling of any database exceptions that are hit running the generated SQL statements when the map executes. If you take the default of not adding a **Failure** transform, IBM Integration Bus will handle the error, reporting it to the system log, and then rolling back the current message transaction.

To configure the IBM Integration Bus run time to connect to a database, you must establish a connection with the database to fulfill the operations that are performed by the Mapping node. You must define a JDBC provider configurable service.

Use this scenario to learn how to use a Mapping node to connect to a database, retrieve data from multiple tables, populate elements in a SOAP message with this information, and handle a database SQL exception. Use this scenario to also learn how to configure the JDBC provider configurable service.

Implementing the solution

During the development phase, you can use a Mapping node to connect to a database, retrieve data from multiple tables, and then populate elements of a SOAP message. During the design of the map, you can connect the IBM Integration Studio to a database to discover the tables. To enable the run time to establish a connection with the database to fulfill the operations that are performed by the Mapping node, you must create a JDBCProvider configurable service.

Before you begin

1. Create the initial configuration. For more information, see “Creating the scenario graphical data map configuration.”
2. Create the database resources. For more information, see “Creating the scenario database configuration” on page 294.

Procedure

Complete the following steps to implement a Mapping node that connects to a database and enriches a message with the database information:

1. Configure the database physical model in IBM Integration Bus by running discovery. For more information, see “Configuring a database in the IBM Integration Studio” on page 295.
2. Configure your integration solution to include the database connection details. For more information, see “Configuring an integration solution to access database resources” on page 303.
3. Add the relevant database tables to your message map. For more information, see “Adding database tables your message map” on page 305.
4. Configure the **Select** transform in your message map to retrieve the database information. For more information, see “Configuring the Select transform in a message map” on page 309.
5. Optional: Handle database failures. For more information, see “Handling database failures in a Select transform” on page 312.
6. Configure the run time to enable it to establish a connection with the database to fulfill the operations that are performed by the Mapping node. For more information, see “Configuring a database to be available at run time” on page 316.

Creating the scenario graphical data map configuration

This scenario was developed by using a sample initial configuration. You can either follow the instructions to add your own database to a message map, or set up the sample final configuration to try out the scenario in the same way as it was originally developed.

Before you begin

- Download a copy of the `FindAddressInitialConfiguration.zip` file.
- Download a copy of the `FindAddressFinalConfiguration.zip` file to set up the final scenario configuration and see the result of following the steps that are documented in the scenario.
- Make sure you have access to an IBM Integration Bus runtime environment and an IBM Integration Studio installation with the default configuration deployed. For more information on installing IBM Integration Bus components, see [Installing in the IBM Integration Bus information center](#).

Procedure

Complete the following steps to set up the sample initial configuration that was used to develop the scenario:

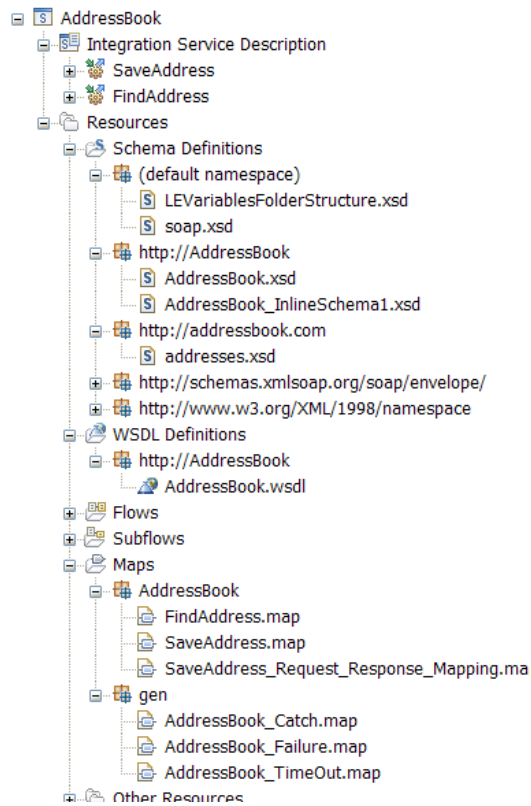
1. Install IBM Integration Studio. For more information, see [Installing in the IBM Integration Bus information center](#).
2. Import the `AddressBookInitialConfiguration.zip` file:
 - a. Click **File** > **Import**. The Import wizard opens.
 - b. Expand **Other**, click **Project Interchange**, then click **Next**.
 - c. Specify the location of the `AddressBookInitialConfiguration.zip` file.
 - d. Specify the location of the open workspace.
 - e. Select the projects that you want to import into your workspace. For this scenario, select all projects. Then, click **Finish**.

Results

You imported the scenario source files.

In the **Application Development** view, you should see the following resources:

- A `SaveAddress` operation
- A `FindAddress` operation
- Multiple xsd structures such as `addresses.xsd`
- The `AddressBook` wsdl file
- The `FindAddress.map` file
- The `SaveAddress.map` file
- The `SaveAddress_Request_Response_Mapping.map` file
- The `Catch.map` file
- The `Failure.map` file
- The `TimeOut.map` file



In your database, you see the ADDRESS, PERSON, and PHONE tables:

IBM-8B27326AD33 - DB2 - CLIENTS - Tables		
Name	Schema	Table space
ADDRESS	ADDRESSBOOK	USERSPACE1
PERSON	ADDRESSBOOK	USERSPACE1
PHONE	ADDRESSBOOK	USERSPACE1

What to do next

Follow the steps for “Configuring a database in the IBM Integration Studio” on page 295.

Creating the scenario database configuration

The message flow that is used in this scenario requires an external database, which is used in one of the message maps that enriches the message data as it runs the transformation. DB2 must be set up in advance.

Before you begin

- Download a copy of the createdbtablesclients.zip file.

Procedure

Complete the following steps to set up the sample DB2 database configuration that was used to develop the scenario:

1. Create a database that is named **CLIENTS**.
 - a. Open a DB2 command prompt and create the database. Click **Start > All Programs > IBM DB2 > DatabaseInstance > Command Line Tools**, and select **Command Window**. *DatabaseInstance* is your DB2 instance name. The default name is **DB2COPY1 (default)**.
A DB2 - CLP window opens.
 - b. Create the **CLIENTS** database. Run the following command: `DB2 CREATE DB CLIENTS`
You receive the following message: `DB20000I The CREATE DATABASE command completed successfully.`
 - c. Test the database connection. Run the following command: `DB2 CONNECT TO CLIENTS`

Note:

If you receive the error `ErrorCode = -4499, SQLState = 08001`, check that the port used is correct, and try again.

2. Create the tables using the SQL **createdbtablesclients.sql** script that is provided in the scenario.
 - a. Unzip the file **createdbtablesclients.zip**.
 - b. From the DB2 command prompt, run the following command: `db2 -vf Sqlscriptdirectory\createdbtablesclients.sql`, where *Sqlscriptdirectory* is the directory where you unzip **createdbtablesclients.zip**.

Results

You have a database that is named **CLIENTS**, and the following database tables that are created under the **ADDRESSBOOK** schema:

- PERSON
- ADDRESS
- PHONE

What to do next

Follow the steps for “Configuring a database in the IBM Integration Studio.”

Configuring a database in the IBM Integration Studio

To make database information accessible from a Mapping node during the development phase, you must define a database definition file with extension `.dbm`. This file is contained in a data design project. You must define one database definition file per database.

Before you begin

To start the scenario, create the initial configuration. For more information, see “Creating the scenario graphical data map configuration” on page 292.

About this task

You can use IBM Integration Bus to access a database and manipulate your business data.

During the development phase, you must configure the database before you can access the data from your a message flow in the IBM Integration Studio. IBM Integration Bus supports the databases that are listed in IBM Integration Bus Requirements.

IBM Integration Bus can access databases that are set up on the local computer or on a remote server, subject to restrictions. For more information, see IBM Integration Bus Requirements.

This scenario demonstrates how to configure a local database in the IBM Integration Studio.

Procedure

To configure the **CLIENTS** database, complete the following steps:

1. Create a data design project as a container for a database definition file. See “Creating a data design project.”
2. Define your database and how it interacts with your integration node. See “Creating the database definition file” on page 298.

Creating a data design project

Create a data design project to contain the database definition file that describes your database.

About this task

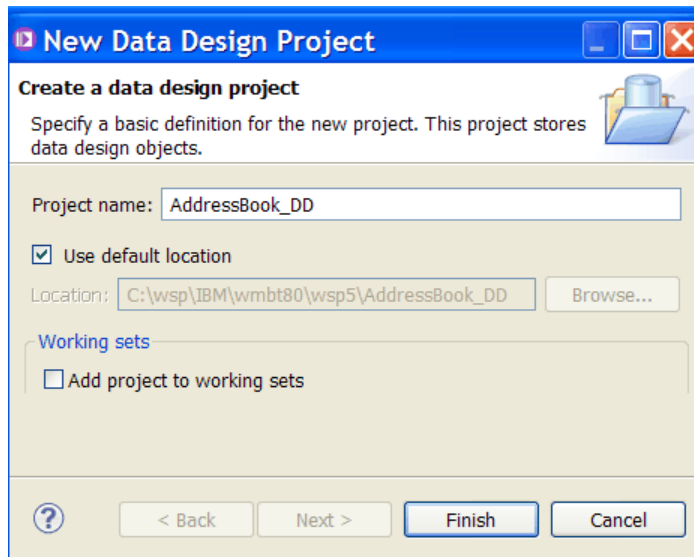
A data design project is a specialized type of project where you store database definition files that hold information about database resources.

In this scenario, you create the **AddressBook_DD** data design project.

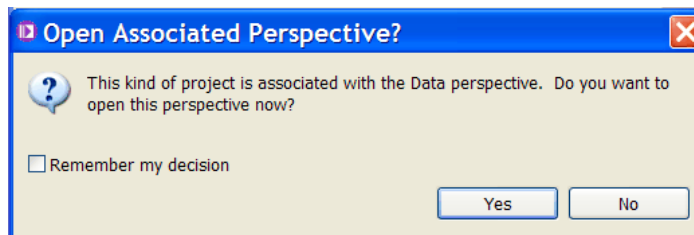
Procedure

Create the **AddressBook_DD** data design project in the Application Development view:

1. Click **File > New > Other > Data Design Project**. The New Data Design Project wizard opens. If this is the first database design project you create within a new workspace you might see the Confirm Enablement window first.
2. Enter **AddressBook_DD** as your *Project name*, and then click **Finish**.



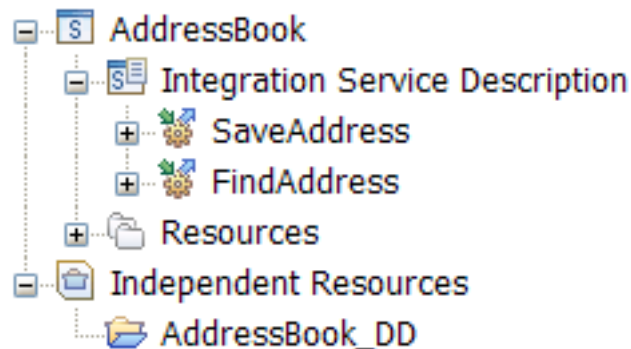
The Open associated perspective? window is displayed.



3. Click No.

Results

Your data design project is created, and is displayed in the Application Development view, under **Independent resources**.



What to do next

After you create a data design project, you must add a database definition (.dbm) file for your database. For more information, see “Creating the database definition file” on page 298.

Creating the database definition file

To create database mappings by using the Mapping node, you must have a database definition file (.dbm) that is contained in a data design project.

Before you begin

Create the **AddressBook_DD** data design project. For more information, see “Creating a data design project” on page 296.

About this task

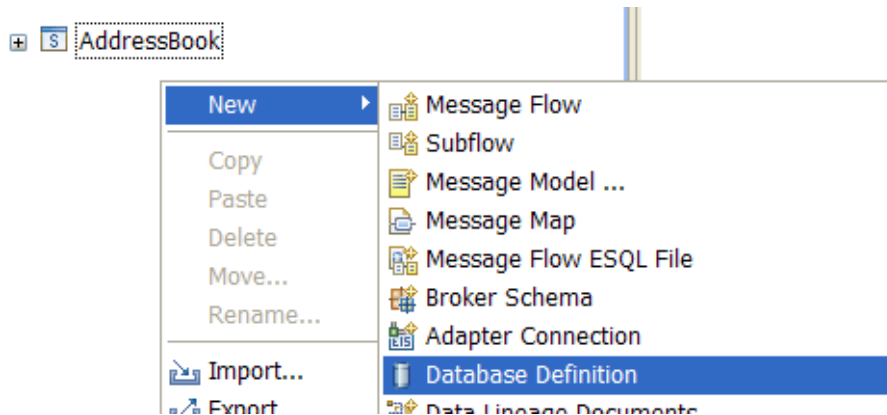
A database definition file holds the physical data model that details all the database resources, such as the schema, the tables, and other resources, that you need access to.

Note: Database definition files in the IBM Integration Studio are not automatically updated. If you modify your database, you must recreate the database definition file describing the connection to the database.

Procedure

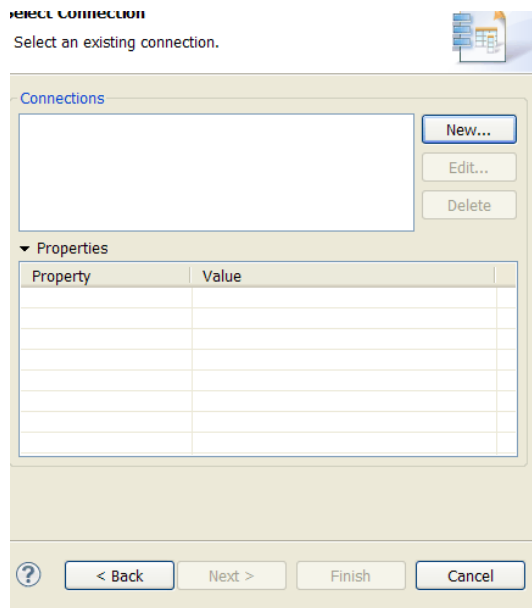
Complete the following steps to create the **CLIENTS.dbm** database definition file by using the New Database Definition File wizard:

1. In the Application Development view, right-click and select **New > Database Definition**.



The New Database Definition File wizard is displayed.

2. From the *Data design project* drop-down list, select **AddressBook_DD**. From the **Database** drop-down list, select **DB2 for Linux, Unix, and Windows**. From the *Version* drop-down list, select **V9.7**. Then, click **Next**.
3. Create a database connection. In the New Database Definition File - Select connection, select **New**.



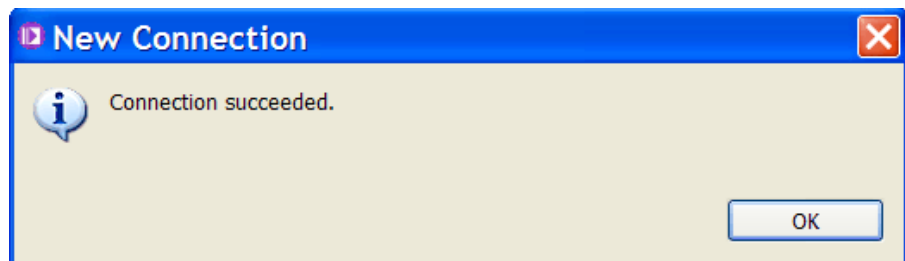
A New Connection window opens.

4. In the New Connection - Connection Parameters, edit the following properties to configure the **CLIENTS** database connection:
 - a. Enter **CLIENTS** as the *Database* value.
 - b. Enter **db2admin** as the *User name*, and enter your database administrator password in *Password*.

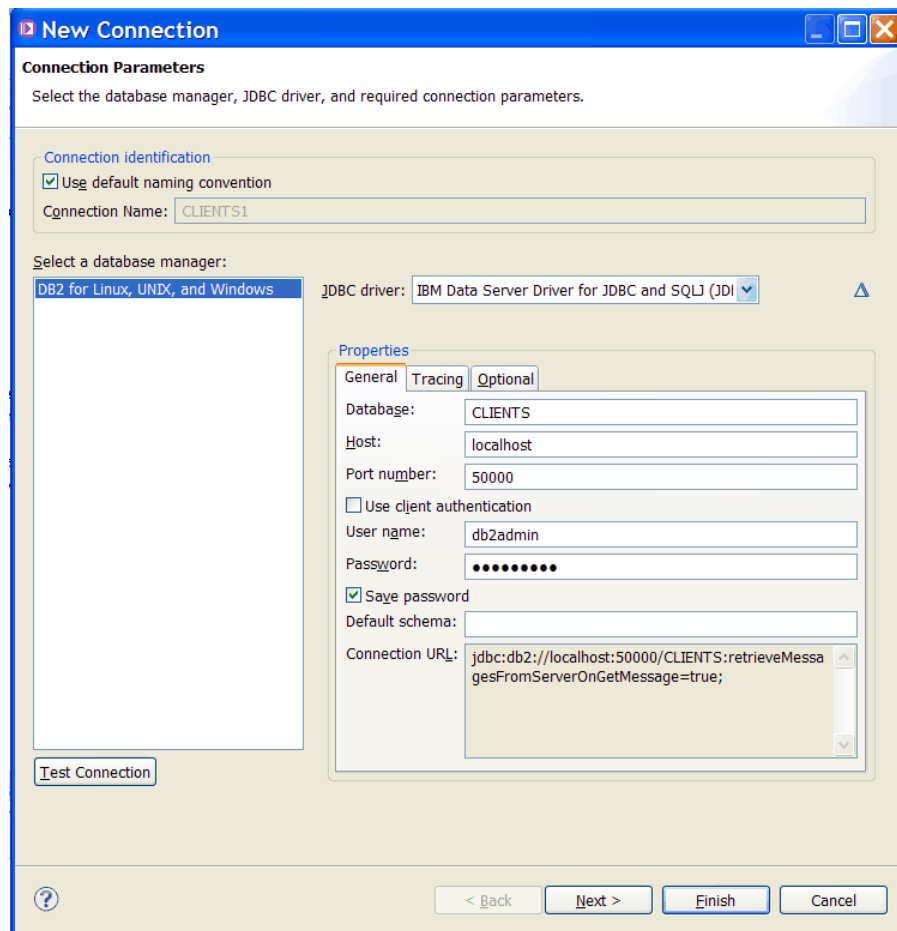
Note: You can use a different user name to connect to the database. This user must have database administration permissions.

- c. Click **Test Connection** to verify the settings that you selected for your database.

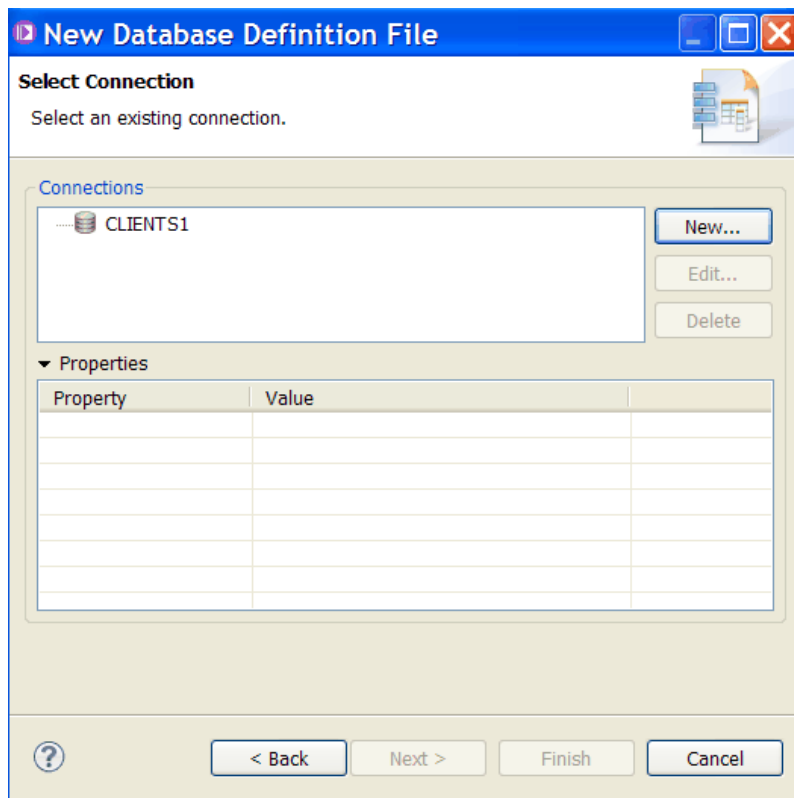
If the connection is successful, the following window opens:



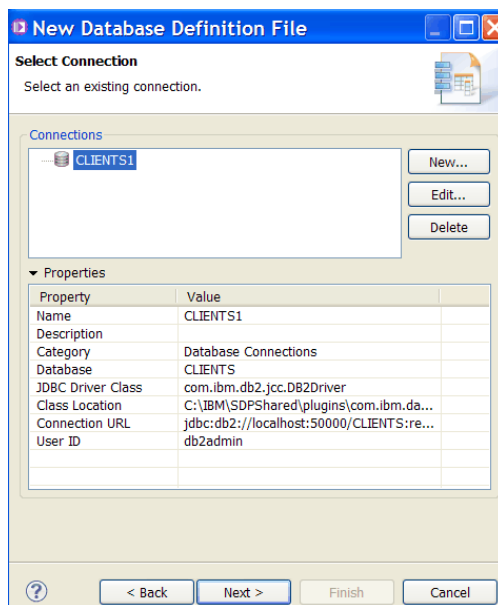
- d. Select *Save Password*.



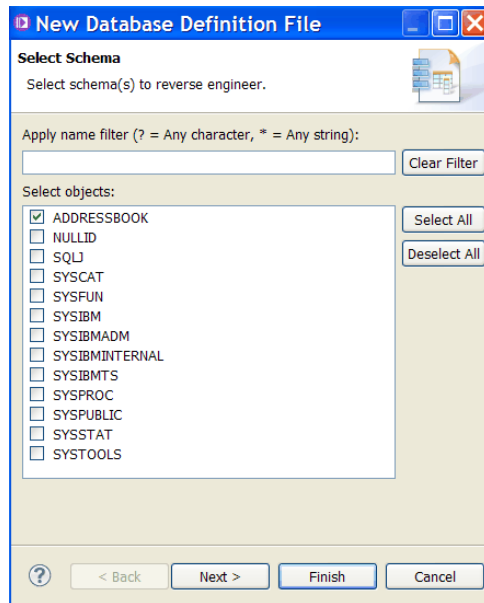
5. In the New Connection - Connection Parameters, click **Finish**.
The **CLIENTS1** connection is created.



6. Select the **CLIENTS1** connection, and then select **Next**.



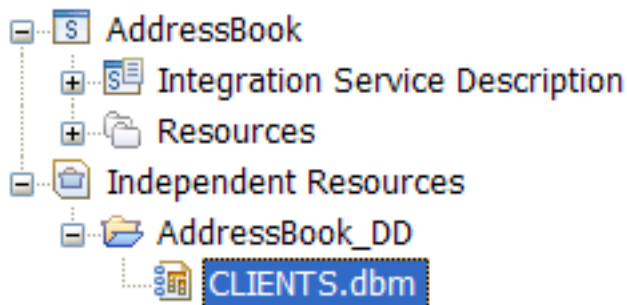
7. Select the schema **ADDRESSBOOK**, and then click **Next**.



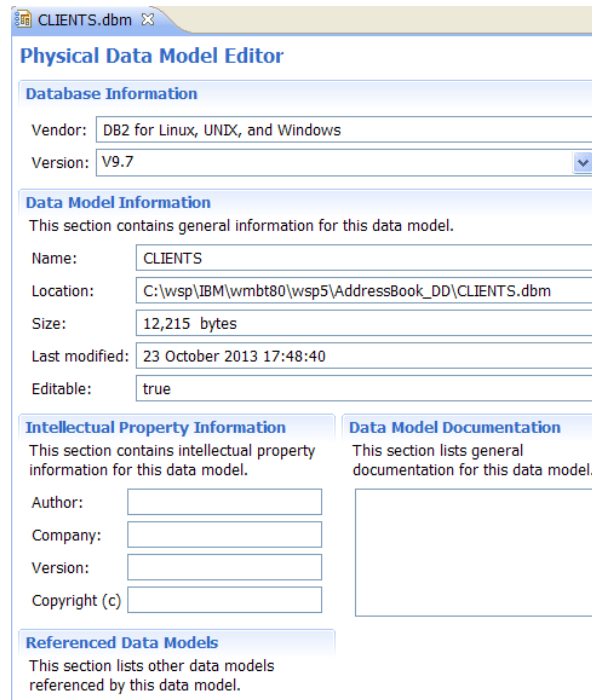
8. In the New Database Definition File - Database Elements window, select **Tables**, then click **Finish**.

Results

The CLIENTS.dbm database definition file is available in the Application Development view under **Independent resources**. The database definition file is created inside the **AddressBook_DD** data project.



The following figure shows the **CLIENTS** database definition file opened in the Physical Data Model Editor.



What to do next

After you create the database definition file **CLIENTS.dbm**, you must configure your integration solution to access specific database resources. For more information, see “Configuring an integration solution to access database resources.”

Configuring an integration solution to access database resources

To access database tables, you must associate the data design project where your database definition file is located with your integration solution.

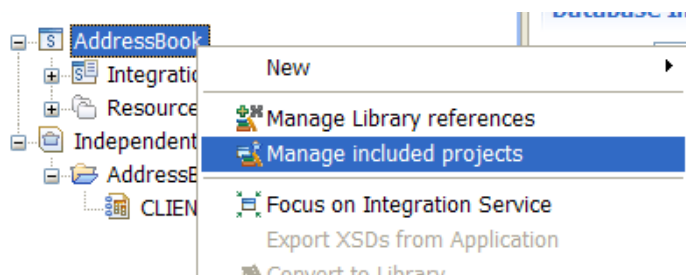
Before you begin

Create the **CLIENTS.dbm** database definition file. For more information, see “Creating the database definition file” on page 298.

Procedure

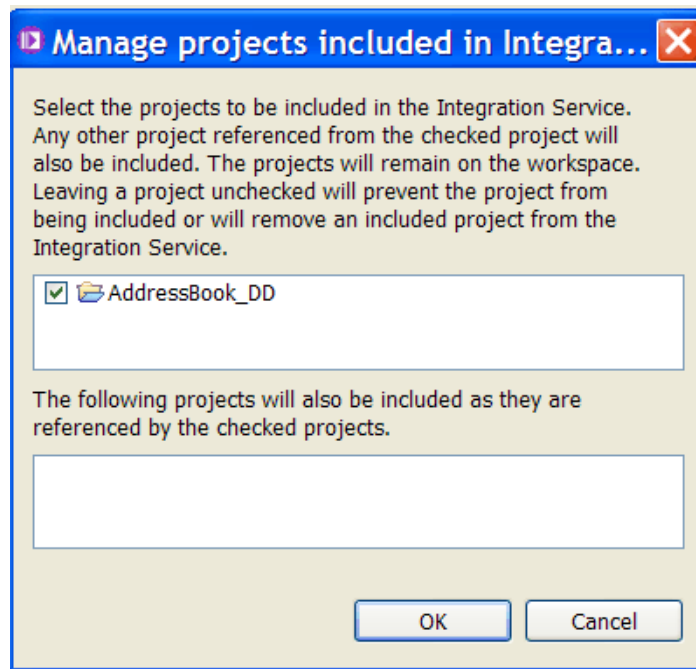
To associate the **CLIENTS.dbm** database definition file to the **AddressBook** integration service, complete the following steps:

1. In the Application Development view, right-click **AddressBook**, and then select **Manage included projects**.



The Manage projects included in Integration Service opens.

2. Select the **AddressBook_DD** data design project.

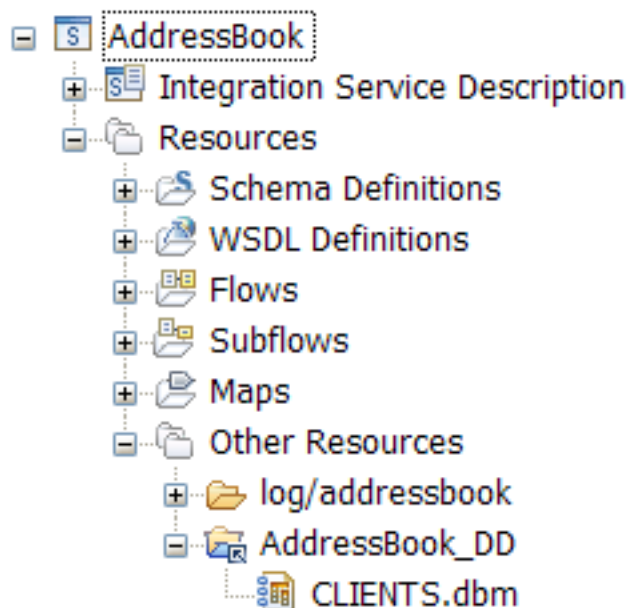


3. Select **OK**.

Results

The data design project **AddressBook_DD** is included as part of the integration service **AddressBook**.

The following figure shows how the data design project **AddressBook_DD** is located under **Other Resources** within the integration service **AddressBook**.



What to do next

After you configure the **AddressBook** integration service to include the data design project **AddressBook_DD**, you must add the database tables to your message map. For more information, see “Adding database tables your message map.”

Adding database tables your message map

To retrieve data from the database, you must define which database tables the message map uses.

Before you begin

Configure the **AddressBook** service to include the data design project **AddressBook_DD** that contains the **CLIENTS.dbm** database definition file. For more information, see “Configuring an integration solution to access database resources” on page 303.

About this task

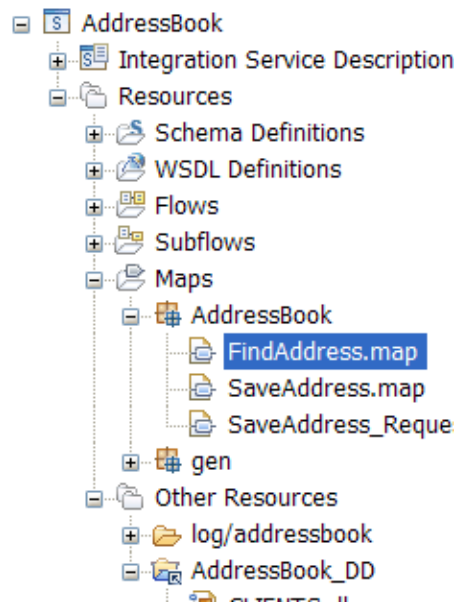
Note: When you add database tables to your message map, you should add all the tables under the same database schema together, that is, one resultset per set of tables. You reduce the number of connections that IBM Integration Bus requires to retrieve database information from those tables.

Procedure

Complete the following steps to add **PERSON**, **ADDRESS**, and **PHONE** database tables under the **ADDRESSBOOK** schema:

1. Open the message map **FindAddress** by completing the following steps:
 - a. In the Application Development view, navigate to **AddressBook > Resources > Maps > AddressBook**.
 - b. Double-click on **FindAddress.map**.

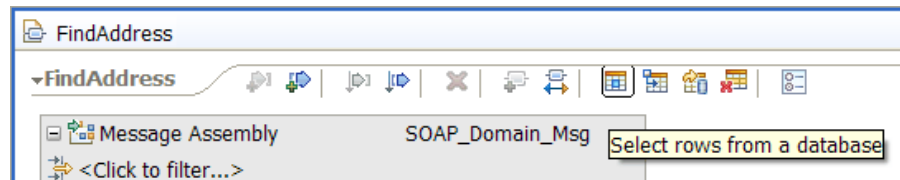
The following figure shows the navigation tree where you can find the message map **FindAddress.map**:



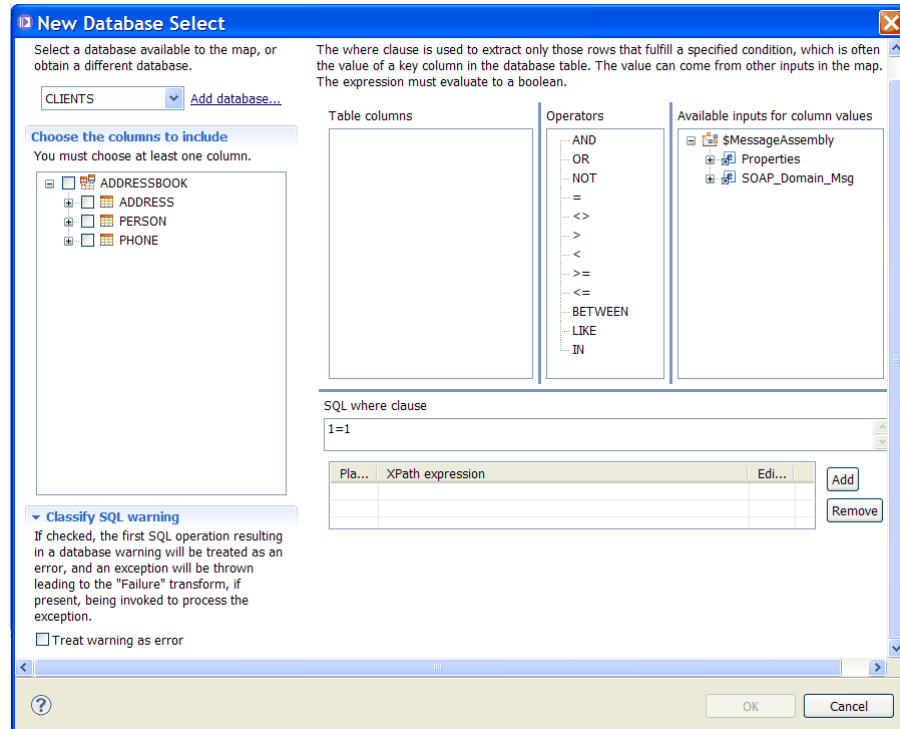
The message map **FindAddress.map** opens in a new tab.

2. Click the **Select rows from a database** icon.

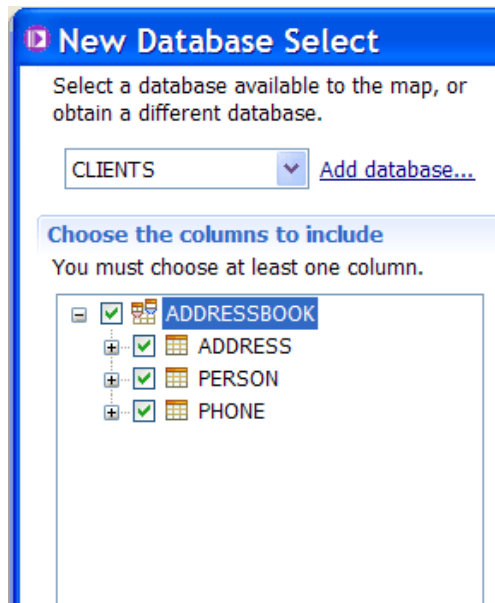
The following figure shows the icon that you must choose to select the option **Select rows from a database**:



The New Database Select wizard opens.



3. Select the schema **ADDRESSBOOK**, and the database tables **PERSON**, **ADDRESS**, and **PHONE**.



4. Define the **SQL where clause** expression that you use to extract a single address record from the database.

To define the expression, you can drop or double-click a column, an operation, or an input into the **SQL where clause** pane, use copy and paste, or use content assist (CTRL+Space).

You can use the following **SQL where clause** expression:

```
ADDRESSBOOK.PERSON.LASTNAME IN ? AND ADDRESSBOOK.PERSON.COUNTRY = ?1
```

where ? represents the XPath expression:

```
$MessageAssembly/SOAP_Domain_Msg/Body/{http://AddressBook}:FindAddress/FindAddress/{http://addressbook.com}:Name
```

and ?1 represents the XPath expression:

```
$MessageAssembly/SOAP_Domain_Msg/Body/{http://AddressBook}:FindAddress/FindAddress/{http://addressbook.com}:Country
```

The following figure shows the expression defined within IBM Integration Bus:

Define a where clause

The where clause is used to extract only those rows that fulfill a specified condition, which is often the value of a key column in the database table. The value can come from other inputs in the map. The expression must evaluate to a boolean.

Table columns

Operators

Available inputs for column values

SQL where clause

```
ADDRESSBOOK.PERSON.LASTNAME IN ? AND ADDRESSBOOK.PERSON.COUNTRY = ?1
```

Pla...	XPath expression
?	\$MessageAssembly/SOAP_Domain_Msg/Body/{http://AddressBook}:FindAddress/
?1	\$MessageAssembly/SOAP_Domain_Msg/Body/{http://AddressBook}:FindAddress/

5. Select OK.

Results

Under **Message Assembly**, the **Select from CLIENTS** section is added. This section contains one resultset. The resultset has three tables. To see which table an element belongs to, select the element in the resultset, and then view the Properties tab.

Beware that tables are included in alphabetical order.

The following figure shows the resultset that you get when you include the tables PERSON, ADDRESS, and PHONE.

FindAddress

Message Assembly SOAP_Domain_Msg

Properties [0..1] PropertiesType

SOAP_Domain_Msg [1..1] SOAP_Msg_type

Select from CLIENTS

ResultSet [0..*]

ID	[1..1]	BIGINT
COUNTRY	[1..1]	CHAR
TYPE	[1..1]	CHAR
NUMBER	[1..1]	INTEGER
LINEADDRESS2	[1..1]	CHAR
LINEADDRESS1	[1..1]	CHAR
POSTCODE	[1..1]	CHAR
CITY	[1..1]	CHAR
ADDITIONALINFO	[1..1]	CHAR

ID	[1..1]	BIGINT
COUNTRY	[1..1]	CHAR
NAME	[1..1]	CHAR
LASTNAME	[1..1]	CHAR

ID	[1..1]	BIGINT
COUNTRY	[1..1]	CHAR
AREA	[1..1]	INTEGER
PREFIX	[1..1]	INTEGER
LOCAL	[1..1]	INTEGER

What to do next

You must configure the **Select** transform in your message map. For more information, see “Configuring the **Select** transform in a message map.”

Configuring the **Select** transform in a message map

You use the **Select** transform to retrieve database information and to perform transformations between the input elements and the output elements of the Message Assembly.

Before you begin

Add the database tables to the message map **FindAddress.map**. For more information, see “Adding database tables your message map” on page 305.

About this task

You can use the **Select** transform in a message map to enrich a message with database information.

The **Select** transform retrieves records from a database based on the **SQL where clause** that you define when you add tables to a message map.

A **Select** transform has a nested map. This nested map is where you transform the input and output elements of the Message Assembly.

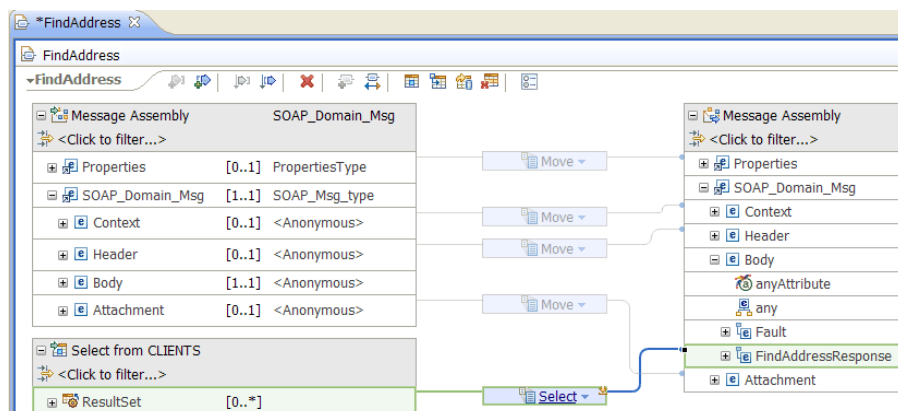
You can set the cardinality of the **Input array indexes** in the **Select** transform properties view to work with a particular row or set of rows, or you can leave this field blank to choose all rows.

This section explains how to configure the **Select** transform when the data available in three database tables is retrieved in one result set.

Procedure

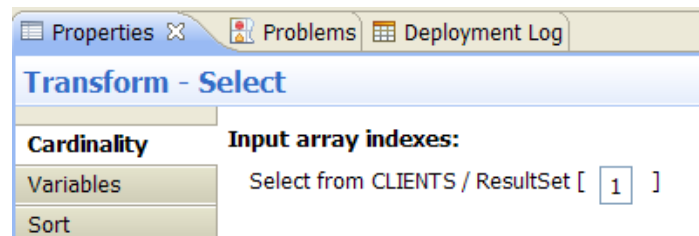
Complete the following steps to enrich a message with the address of a client from the **CLIENTS** database:

1. Open the message map **FindAddress.map** in the Graphical Data Mapping editor.
2. Connect the database section **ResultSet** to the message assembly body section **FindAddressResponse** with a **Select** transform.

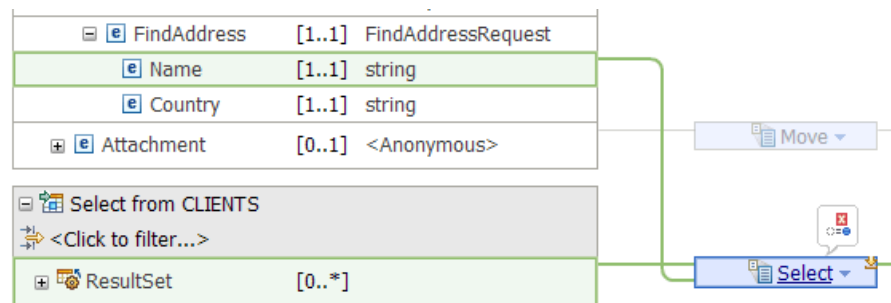


- Set the cardinality of the **Input array indexes** to 1 in the **Select** transform properties view to indicate that you only wish to work with the first row of the result set returned by the database.

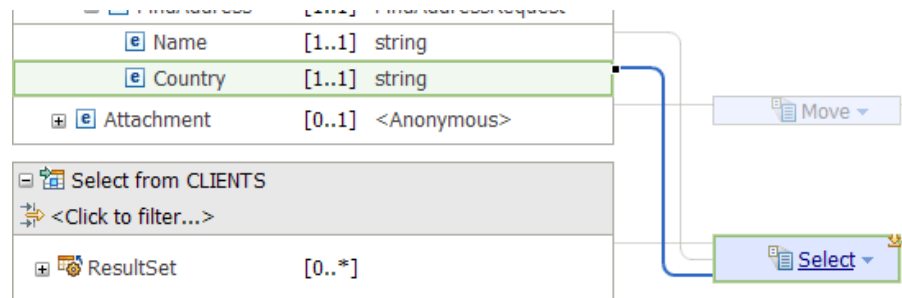
The following figure shows the Properties tab of the **Select** transform:



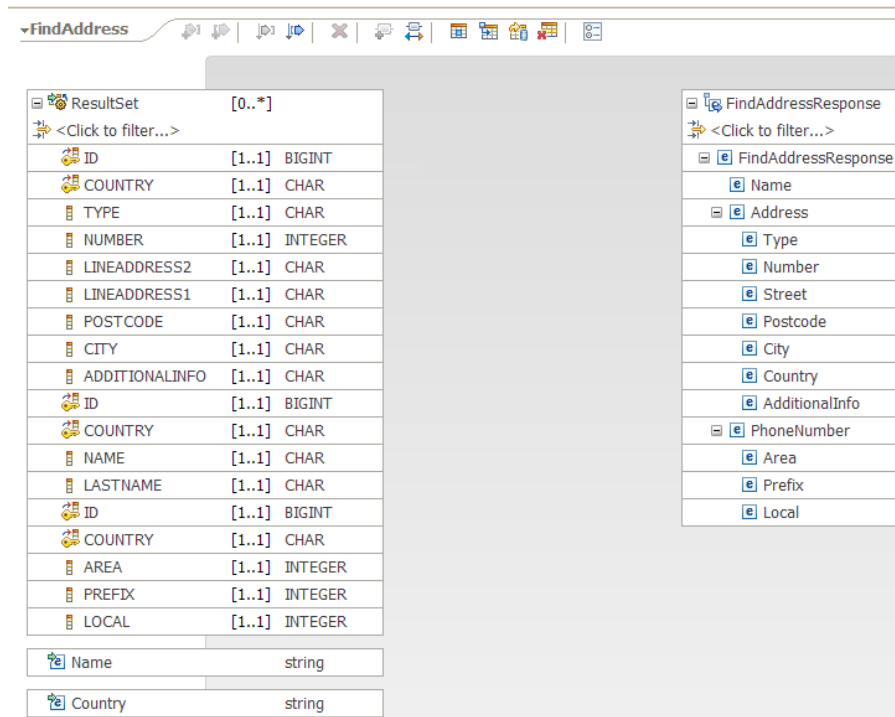
- Connect the message assembly element **Name** to the **Select** transform. The following figure shows the element **Name** connected to the **Select** transform:



- Connect the message assembly element **Country** to the **Select** transform. The following figure shows the element **Country** connected to the **Select** transform:



- Click **Select**. The nested map associated to the **Select** transform opens. The following figure shows the nested map with the input and output objects.

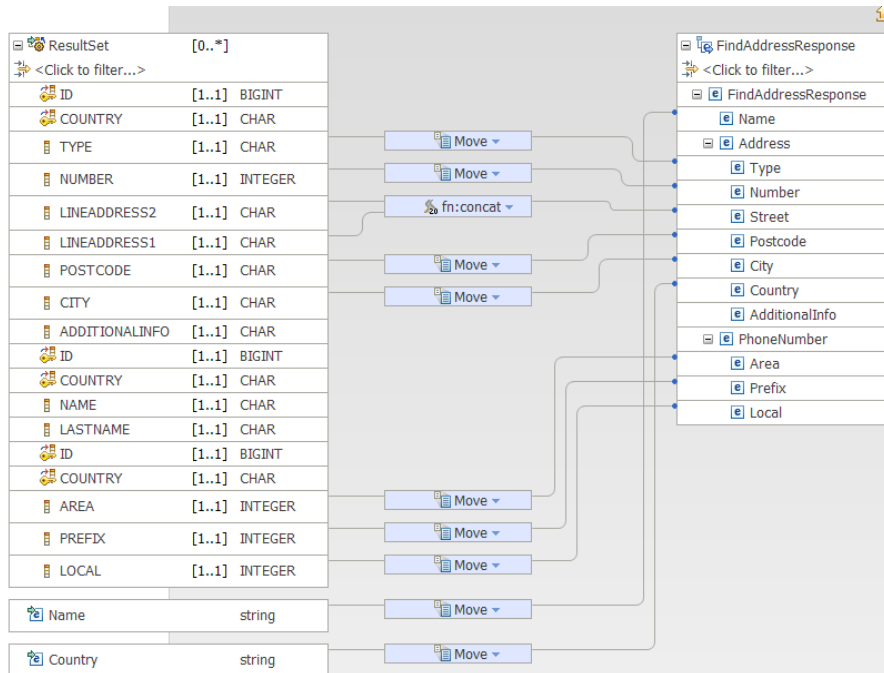


7. Define the transforms between the database elements and the message assembly output elements. This can be automatically completed by using the Auto Map capability. To manually define each transform complete the following steps:
 - a. Connect the input element **Name** to the output element **Name** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - b. Connect the input element **Country** to the output element **Country** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - c. Connect the element **TYPE** to the element **Type** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - d. Connect the element **NUMBER** to the element **Number** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - e. Connect the elements **LINEADDRESS2** and **LINEADDRESS1** to the element **Street** in the message assembly body section **FindAddressResponse** with a **fn:concat** transform.
 - f. Connect the element **POSTCODE** to the element **Postcode** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - g. Connect the element **CITY** to the element **City** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - h. Connect the element **ADDITIONALINFO** to the element **AdditionalInfo** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - i. Connect the element **AREA** to the element **Area** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - j. Connect the element **PREFIX** to the element **Prefix** in the message assembly body section **FindAddressResponse** with a **Move** transform.
 - k. Connect the element **LOCAL** to the element **Local** in the message assembly body section **FindAddressResponse** with a **Move** transform.

Results

You have successfully configured the nested map of the **Select** transform.

The following figure shows the nested map:



What to do next

Handle failure of the **Select** transform in a message map. For more information, see “Handling database failures in a Select transform.”

Handling database failures in a Select transform

You can configure a **Failure** transform for each **Select** transform that you define in a message map to handle explicitly SQL database exceptions. By default, the Mapping node throws database exceptions that can be handled by other nodes in the message flow.

Before you begin

Configure the **Select** transform in a message map. For more information, see “Configuring the Select transform in a message map” on page 309.

About this task

By default, the Mapping node throws database exceptions that the SOAPInput node catches and automatically uses to build a SOAP fault to return to the client.

In the scenario, you use an optional **Failure** transform to process the first SQL exception that might be thrown from the **Select** transform database transaction. You build a SOAPFault to include the database exception detail and the Name and Country elements used for the search of an address which failed.

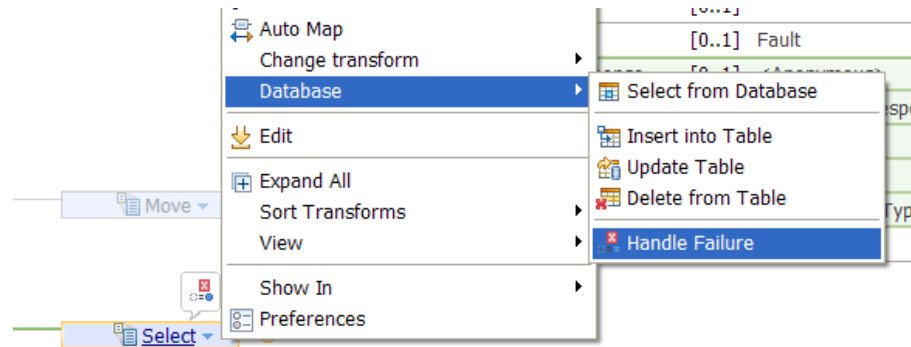
A **Failure** transform has a nested map. This nested map is where you transform the input and output elements of the Message Assembly to define how to handle failure.

Procedure

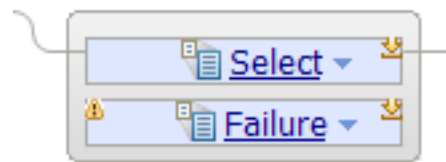
To configure the **Failure** transform in the scenario, complete the following steps:

1. Right-click **Select**, and then select **Database > Handle Failure**.

The following figure shows the graphical path choices to add a **Failure** transform to a **Select** transform:

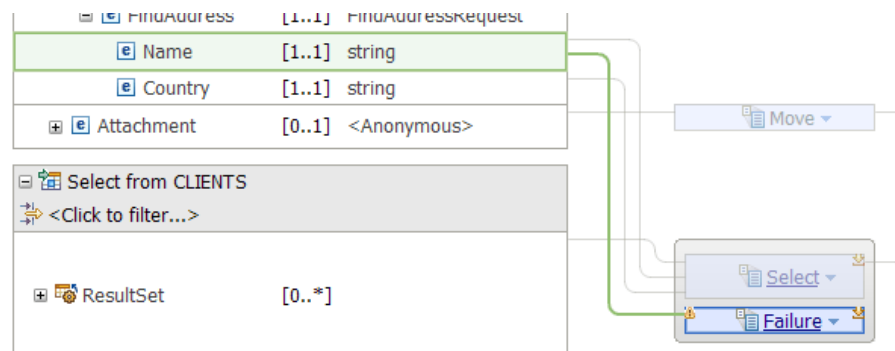


A **Failure** transform is added to the **Select** transform.



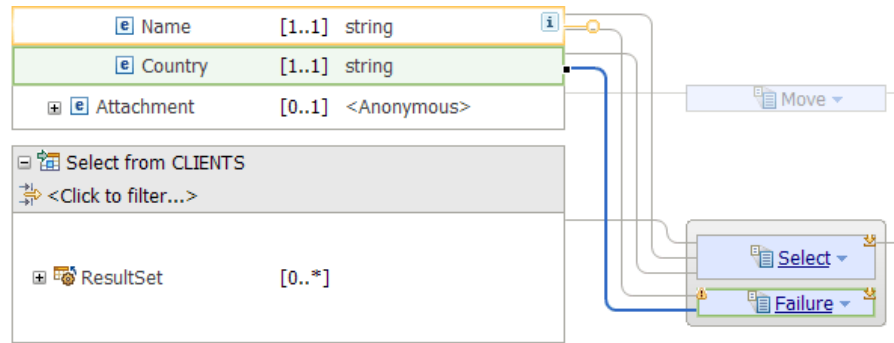
2. Connect the message assembly element **Name** to the **Failure** transform.

The following figure shows the element **Name** connected to the **Failure** transform:

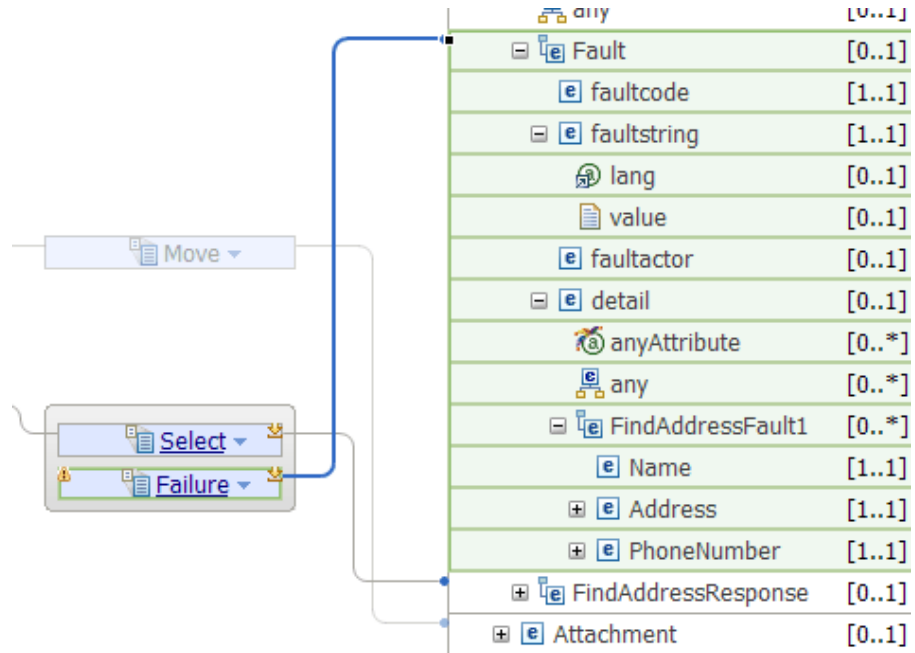


3. Connect the message assembly element **Country** to the **Failure** transform.

The following figure shows the element **Country** connected to the **Failure** transform:

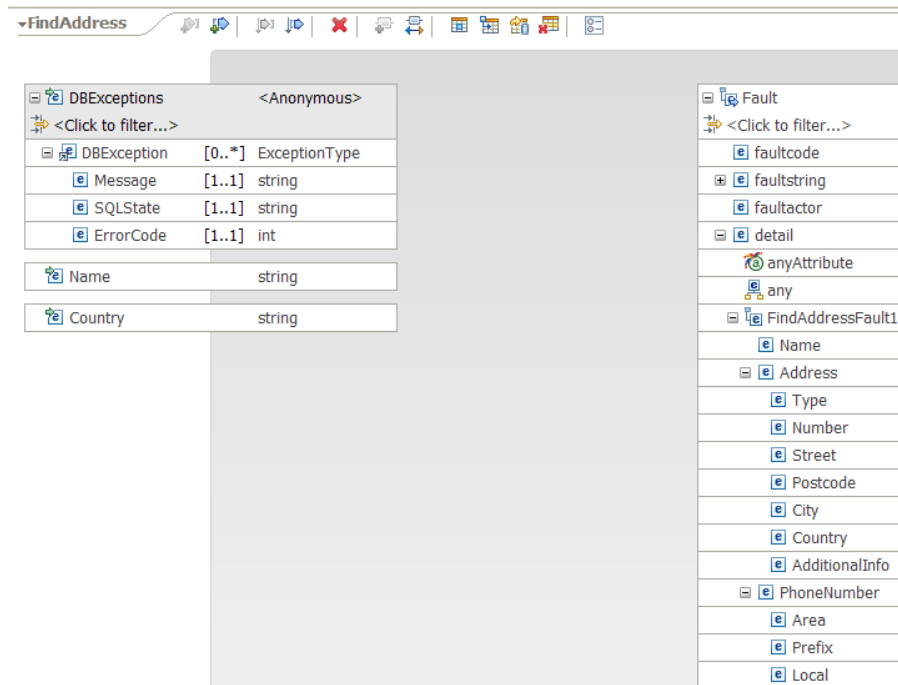


4. Connect the **Failure** transform to the output element **Fault**.

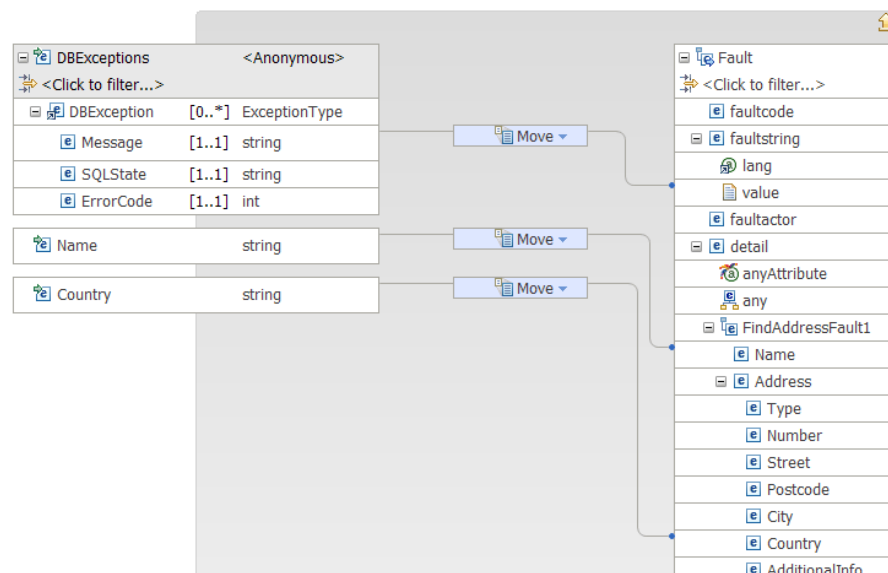


5. Select **Failure**.

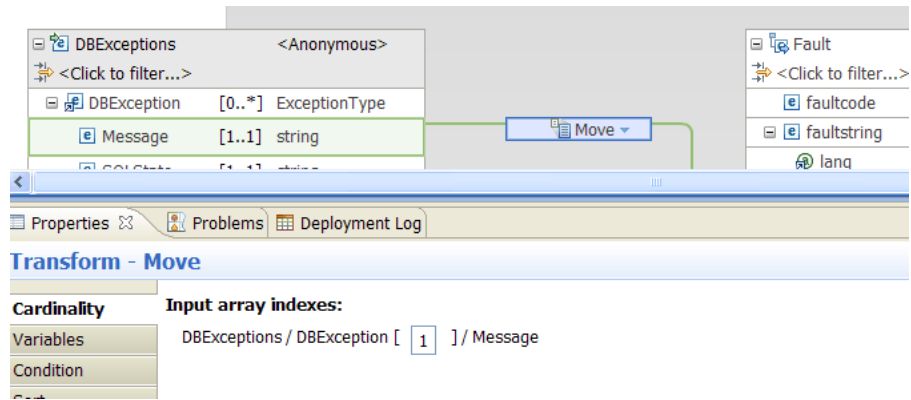
The **Failure** transform nested map opens.



6. Define the transforms between the input elements and the message assembly output elements inside the nested map. Complete the following steps:
 - a. Connect the input element **Name** to the output element **Name** in the message assembly body section **Fault** with a **Move** transform.
 - b. Connect the input element **Country** to the output element **Country** in the message assembly body section **Fault** with a **Move** transform.
 - c. Connect the database exception element **Message** to the output element **value** in the message assembly body section **Fault** with a **Move** transform.



7. Set the cardinality index for the database element **Message** to 1 in the Properties tab of the **Move** transform.
 The following figure shows the Properties tab of the **Move** transform for the element **CITY**:



Results

You have successfully completed the development steps of the scenario.

Configure the JDBC connection at run time. For more information, see “Configuring a database to be available at run time.”

Configuring a database to be available at run time

To make database information accessible at run time, you must establish a connection with the database to fulfill the operations that are performed by the Mapping node. You must define a JDBC database connection.

About this task

The IBM Integration Studio connects to the database CLIENTS and runs discovery so the Graphical Data Mapping editor can use the database definition to visualize the database tables PERSON, ADDRESS, and PHONE.

At run time, the Mapping node uses a JDBC provider configurable service to obtain the configuration parameters that will enable it to make the connection to the runtime database that the message map will execute against.

Procedure

To configure the JDBC connection between the IBM Integration Bus run time and the runtime database, you must complete the following steps:

1. Create a JDBC provider configurable service. For more information, see “Configuring a JDBC provider configurable service.”

The JDBC provider configurable service provides the IBM Integration Bus run time with the information that it needs to complete the connection to the runtime database.

2. Set up security. For more information, see “Securing the JDBC provider configurable service” on page 318.

Some databases require access to be associated with a known user ID and password, for other, this association is optional. A DB2 database requires a data source login name and password on all connections.

Configuring a JDBC provider configurable service

You can configure a JDBC provider configurable service by running the `mqsicreateconfigurable-service` command.

Before you begin

The integration node named Server1 is created and started.

About this task

The IBM Integration Studio connects to the database CLIENTS and runs discovery so the Graphical Data Mapping editor can use the database definition to visualize the database tables PERSON, ADDRESS, and PHONE.

At run time, the Mapping node uses a JDBC provider configurable service named CLIENTS to obtain the configuration parameters that will enable it to make the connection to the runtime database PCLIENTS that the message map will execute against.

You configure the runtime database resources by defining the JDBC provider configurable service properties. You must set the following properties:

- **databaseName** property: You must set its value to be the runtime database name PCLIENTS.
- **databaseSchemaNames** property: You must set its value to use at run time the database schema PADDRESSBOOK.

Note: The table names must be the same in the database development environment and in the run time database environment.

The following table lists the database resource names in the development environment, and in the runtime environment:

Table 24. Scenario database resource names

	Development database resource names	Runtime database resource names
Database name	CLIENTS	PCLIENTS
Schema name	ADDRESSBOOK	PADDRESSBOOK
Tables names	PERSON, ADDRESS, PHONE	PERSON, ADDRESS, PHONE

Procedure

To configure the JDBC provider configurable service **CLIENTS**, run the **mqsicreateconfigurableservice** command:

```
mqsicreateconfigurableservice Server1 -c JDBCProviders -o CLIENTS -n databaseName,databaseSchemaNames -v PCLIENTS,PADDRESSBOOK
```

where

- Server1 is the name of the runtime integration node.
- **-o** defines the name of the JDBC configurable service.

Set the value to the name of the development database, that is, CLIENTS.

CLIENTS is the development database name that you used to configure the data definition file in the IBM Integration Studio.

- **-n** defines the list of properties that you must set to configure the JDBC connection.

These properties are required by the Mapping node to access the database information at run time.

You must define the **databaseName** property, and the **databaseSchemaNames** property.

- **-v** defines the values you set for each property defined in **-n**.
 - Set the **databaseName** property to be the name of your runtime database, that is, PCLIENTS.
 - Set the **databaseSchemaNames** property to be the name of your runtime schema, that is, PADDRESSBOOK.

Results

A JDBC provider configurable service is available at run time.

What to do next

Secure the JDBC connection. For more information, see “Securing the JDBC provider configurable service.”

Securing the JDBC provider configurable service

You secure the JDBC connection to a DB2 database by running the **mqsisetdbparms** command, and the **mqsichangeproperties** command.

About this task

A DB2 database requires a data source login name and password on all connections.

Procedure

You must secure the JDBC connection to the DB2 database by completing the following steps:

1. Identify the user ID and password that you want to associate with the JDBC connection.
In the scenario, **db2admin** is the user ID used. Request the user ID and password of your installation from your system administrator.
2. Run the **mqsisetdbparms** command to associate the user ID and password with the security identity **scenario** that is associated with the database.

```
mqsisetdbparms Server1 -n jdbc::SecurityIdentity -u userID -p password
```

where:

- **-n** is the security identity that is used to authenticate the JDBC connection. Set the value to `jdbc::scenario`.

Note: In the scenario, you create a security identity whose value is **scenario**. However, you can use any name for the security identity. The security identity name that you define in this step must be used to configure the **securityIdentity** property of the JDBC configurable service in the following step.

- *userID* is your user ID.
- *password* is the password of the user ID.

Run the following command:

```
mqsisetdbparms Server1 -n jdbc::scenario -u db2admin -p password
```

Note: The security identity that you define in this step is also used to configure the **securityIdentity** property of the JDBC configurable service.

3. Update the **securityIdentity** property of the **CLIENTS** JDBC configurable service to associate the JDBC connection with the database security identity. Run the **mqsichangeproperties** command.

```
mqsichangeproperties Server1 -c JDBCProviders -o CLIENTS -n  
securityIdentity -v scenario
```

where:

- **Server1** is the name of the runtime integration node.
- **-o** defines the name of the JDBC configurable service. Set the value to **CLIENTS**.
- **-c** defines the type of the configurable service. Set the value to **JDBCProviders**.
- **-n** defines **securityIdentity** as the name of the property that you must set.
- **-v** defines the value of the **securityIdentity** property. Set the value to **scenario**.

Results

You have secured the JDBC connection.

You have completed the scenario.

Chapter 29. Using or converting legacy resources into message maps

You can compile and deploy legacy message maps and legacy message flow nodes in WebSphere Message Broker Version 8 and later versions. However, if you need to modify any of these resources, you must convert them into message maps that the new Mapping node consumes.

About this task

A legacy message map is a message map created as a .msgmap file in earlier versions of WebSphere Message Broker, for example in WebSphere Message Broker Version 7.

Note: From WebSphere Message Broker Version 8 onwards, legacy message maps are accessible in read-only mode and cannot be modified by using the IBM Integration Studio.

The functions provided by the following legacy message flow nodes in WebSphere Message Broker Version 7 are replaced with the new Mapping node in WebSphere Message Broker Version 8 and later versions:

- DataDelete node
- DataInsert node
- DataUpdate node
- Extract node
- Mapping node
- Warehouse node

Procedure

Follow these guidelines when you use legacy message maps and legacy message flow nodes in your integration solutions:

- You can import message flows developed in earlier versions of WebSphere Message Broker that use legacy message maps.
- You can import a message flow that contains legacy message flow nodes. You can continue to view and deploy these nodes, but you cannot modify them.
- You can compile and deploy message flows that use legacy message maps and legacy message flow nodes.

If you use a legacy message map as part of your integration solution, the BAR file that you use to deploy the solution must have the option **Compile and in-line resources** set. This setting might be incompatible with some other functions, such as deployable subflows that require deploy as source mode. For more information, see Adding files to a BAR file.

- You must convert a legacy message map to a message map before you can modify any of its mapping operations by using the Graphical Data Mapping editor. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

You can convert your legacy message map created as a .msgmap file to a message map created as a .map file.

- You must convert a legacy message flow node before you can modify any of its transformation logic.

Note: The conversion of a legacy message flow node into a message map and a Mapping node is a manual process. For more information, see Chapter 8, “Creating a message map,” on page 45 and “Replacing a WebSphere Message Broker Version 7 Mapping node” on page 339

- You must change the transformation logic in your integration solution when a legacy message map is invoked from an ESQL CALL statement. For more information, see “Converting a legacy message map that is called from an ESQL statement in a Compute node” on page 337.

A message map cannot be called from a ESQL CALL statement.

Changes in behavior in message maps converted from legacy message maps

From WebSphere Message Broker Version 8 onwards, you transform data graphically by using a message map. These maps are managed through the Graphical Data Mapping editor. You define your transformation logic by using XPath 2.0 expressions. You can also call Java methods, ESQL procedures, or complex XPath expressions by using specialized transforms such as the **Custom Java**, **Custom XPath**, or **Custom ESQL** transforms.

Behavior changes during the development phase

Nulls behavior

When you convert a legacy message map that includes handling for nilled elements, check how a message map handles NULL values. For more information, see Chapter 5, “Handling nulls in message maps,” on page 35.

- In ESQL, a special NULL value is defined, and is distinct from empty. When you assign NULL to a named element, or set the element from the returned NULL value of a called ESQL function, you delete the element from the tree.

In a message map, the ESQL NULL produces an empty element, or an empty element with the `xsi:nil` attribute set when the element is defined as nillable in the model. Consequently, in some cases the output of the map might include unexpected empty elements that can cause processing problems, including XML schema validation violations. Such problems typically occur when an ESQL user-defined function that returns ESQL NULL in some conditions is called. To avoid these problems, add a condition to the Custom Transform to prevent it from being invoked if it would return NULL.

Assigning literal values to output elements

Use the Assign transform to set literal values in output elements. The Assign transform uses a string representation, which is assigned to the relevant output element and so must be formatted according to its type. The property value does not need to be in quotation marks, as any quotation marks would be passed as part of the string value. To provide an explicitly typed value, use the `xs:<type>` Cast transform with no input wiring.

Literals in conditional expressions

You could build expressions in the legacy message mapping editor that implied a type cast and used the underlying string value representation.

The Graphical Data Mapping editor uses XPath expression syntax and enforces strict typing. For example, testing a Boolean-typed element for the string literal value **true** would cause a type exception.

You can use the `xs:<type>` functions in your expressions to avoid these incorrect typing issues.

Complex type text values in condition expression

A legacy message map does not require the user to be explicit when accessing mixed content text values from a complex type element in a condition expression.

The Graphical Data Mapping editor is based on standard XPath syntax, and requires the explicit use of `/text()` to signify that the mixed content text value is to be used. As a result, a converted map with a conditional expression that referenced mixed content text values might fail until the path expression is extended to add the missing `/text()`.

Literals in submap calls

The legacy message map editor did not correctly validate the typing of submap inputs. Users could edit the normal element path value of a submap input, and instead provide an untyped literal value.

The Graphical Data Mapping editor requires that all submap inputs are wired to an appropriately typed input element.

"For each index" counter variables

Some transformations require the use of the "For each index" counter value. The WebSphere Message Broker Version 6.1 and WebSphere Message Broker Version 7 message map provided the `msgmap:occurrence` function to obtain the current loop count. The Graphical Data Mapping editor provides a **For loop** counter variable which can be used to provide equivalent function. The name of this variable is fixed format `$<For each primary input element name>-index` can be obtained using content assist `ctrl-space` in the relevant "ForEach transform" Filter properties expression panel, or in the content assist in any nested transforms.

Behavior changes during the deployment phase

From WebSphere Message Broker Version 8 onwards, message maps can be deployed only as source. You must provide any of the following resources before deploying your solution:

- For text and binary messages, you must provide the DFDL schema file or the message set that defines your input and output messages.
- For XML messages, you must provide the XML schema files that define your input and output messages.

If your message is modeled in a message set, the message map requires the message set schema (`.xsdzip` file) to be deployed to run your message map. If your existing message set is used for text and binary formats only, you can deploy your message map with only a `.dictionary` file in the integration node archive (BAR). In this case, you must modify the message set to additionally set the XMLNSC domain support option, so it is added to a BAR with both a `.dictionary` file and `.xsdzip` file. If this option is not set, a warning is displayed in the Problems view, along with a quick fix action.

Behavior changes at run time

- From WebSphere Message Broker Version 8 onwards, the Graphical Data Mapping editor has a dedicated Java based run time. As a result, map execution benefits from full support for XPath 2.0 and Java JIT optimization, which offers increased reliability.
- A message map can be invoked only from a Mapping node.
- - A submap can be invoked only from a top level message map, that is, a map consumed by a Mapping node.
- The message map runs in the Java virtual machine of the integration node. You must ensure that the integration node is configured correctly. For more information, see [Setting the JVM heap size](#).

Planning the conversion of a legacy message map

Before you convert a legacy message map, review this section to help you plan the migration.

About this task

You can compile and deploy legacy message maps and legacy message flow nodes in WebSphere Message Broker Version 8 and later versions. However, if you need to modify any of these resources, you must convert them into message maps that the new Mapping node consumes.

Procedure

Complete the following tasks to plan the conversion of a legacy message map to a message map:

1. Verify that all the projects that include resources used by the legacy message map are available in your workspace, and the project dependencies are defined.
The conversion process needs access to all projects that include resources used by the legacy message map to be able to convert automatically your map transformations.
2. Identify the legacy message map resources used in your transformations.
 - a. Check the input and output structures to the legacy message map.
 - Are you doing transformations that include the local environment tree?
 - Are you doing transformations that include data structures with *xsd:any* elements?

For more information, see [“Converting a legacy message map that includes transformations of the local environment tree or *xsd:any* elements”](#) on page 332.

- b. Check your transformation for any of the following type of transformations:
 - Transformations that include calls to user-defined ESQL procedures. For more information, see [“Converting a legacy message map that includes user-defined ESQL procedures”](#) on page 333.
 - Transformations that include calls to ESQL mapping functions. For more information, see [“Converting a legacy message map that includes ESQL mapping functions”](#) on page 334.
 - Transformations that include calls to message map functions. For more information, see [“Converting a legacy message map that includes calls to message map functions”](#) on page 336.

- Transformations that include calls to relational database operations. For more information, see “Converting a legacy message map that includes relational database operations” on page 337.
- Transformations that include calls to a message map. For more information, see “Converting a legacy message map that is called from an ESQL statement in a Compute node” on page 337.

What to do next

1. Define schema models for any *xsd:any* element in your input or output structures.
2. Run the conversion process. For more information, see “Converting a message map from a .msgmap file to a .map file.”
3. Review the newly created message map and complete the post-conversion tasks. For more information, see “Managing conversion warnings on converted legacy message maps” on page 327 and “Managing conversion errors on converted legacy message maps” on page 330.
4. Update the message flow that includes the legacy message map to include the new Mapping node and reference the newly created message map. For more information, see “Replacing a WebSphere Message Broker Version 7 Mapping node” on page 339.
5. Deploy and test your message flow. For more information, see Deploying solutions.

Converting a message map from a .msgmap file to a .map file

You must convert a message map from a previous version of IBM Integration Bus to a graphical data map before you can modify it by using the Graphical Data Mapping editor.

Before you begin

Before you convert a legacy message map, complete the following steps:

1. Import your resources from WebSphere Message Broker Version 6.1 or WebSphere Message Broker Version 7. For more information, see Importing resources from previous versions. Alternatively, you can migrate the WebSphere Message Broker Version 6.1 or WebSphere Message Broker Version 7 Toolkit development resources. For more information, see Migrating development resources to IBM Integration Studio Version 10.0.
2. Verify that all the projects that include resources used by the legacy message map are available in your workspace, and the project dependencies are defined.
3. Review “Changes in behavior in message maps converted from legacy message maps” on page 322 and “Considerations for mapping messages modeled in message sets” on page 13.

About this task

In WebSphere Message Broker Version 8 and later, if you want to modify mapping operations that are defined in a legacy message map, you must first convert your map to a graphical data map (.map file). A graphical data map is known as a message map in IBM Integration Bus.

Note: The conversion process is not reversible. However, you can run the conversion of a legacy message map multiple times. You must rename the converted legacy message map by removing *_backup* from the converted map name.

To use a converted legacy message map in your message flows, you must replace legacy Mapping nodes with new Mapping nodes. For more information, see “Replacing a WebSphere Message Broker Version 7 Mapping node” on page 339.

Procedure

To convert a legacy message map to a message map by using the IBM Integration Studio, complete the following steps:

1. Start the conversion process: In the Application Development view, right-click the message map that you want to convert, and click **Convert Message Map from .msgmap to .map**.

To convert multiple legacy message maps, right-click a folder, project, application, or library that contains one or more maps, and click **Convert Message Map from .msgmap to .map**.

- Your converted message map is created, and is displayed in the Application Development view.
- Your legacy message map is renamed *MessageMapName.msgmap_backup*, and is displayed in the Application Development view.

2. Open the converted map in the Graphical Data Mapping editor: In the Application Development view, double-click your new message map.


The message map opens in the Graphical Data Mapping editor.

3. Review the transformation logic that was created by the conversion process to ensure that it produces the correct output for your application.

- a. Review and replace each **Task** transform. For more information, see “Managing conversion errors on converted legacy message maps” on page 330.

If your legacy message map contains complex mapping structures that the conversion process was not able to re-create, your message map includes **Task** transforms to assist you in manually re-creating those structures.

Task transforms are listed in the Problems view.

- b. Review all the conversion annotation icons () that are associated to transforms in your map. For more information, see “Managing conversion warnings on converted legacy message maps” on page 327.

Conversion annotation icons are displayed on the lower left of the transform in the Graphical Data Mapping editor.

Note: You can accept or reject all the transforms in a converted map in one single click.

Results

Your message map is converted to a message map that can be modified by using the Graphical Data Mapping editor.


What to do next

Modify each message flow that references the legacy message map so that new Mapping nodes reference your new message map. For more information, see

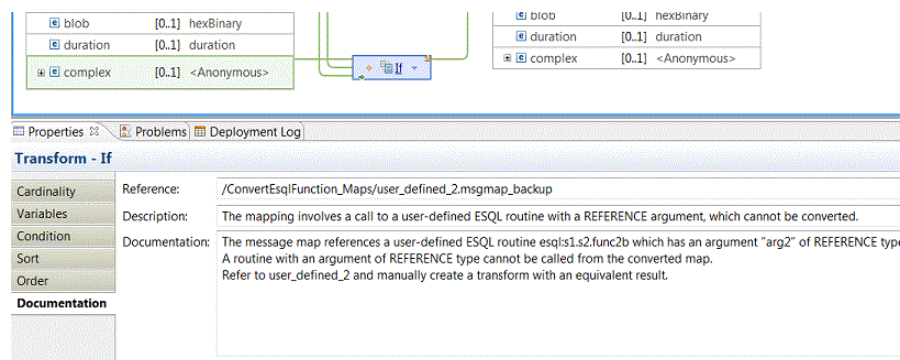
Managing conversion warnings on converted legacy message maps

You can accept or reject conversion actions suggested by the map conversion process when you transform a legacy message map into a message map.

About this task

If you convert a legacy message map (.msgmap file) to a message map (.map file), and your message map contains complex mapping structures that the conversion process is able to recreate, your new map might include conversion warnings on one or more of the transforms that are created by the conversion process. These warnings are displayed in the Graphical Data Mapping editor as conversion annotation icons ().

Note: The conversion process creates one warning per transform that needs to be reviewed. You must check all the warnings present on one or more of the transforms in your converted map. Why? The output of the converted transform might be different to the behavior in your original message map.



If a transform contains a conversion warning, it will have a small green arrow in the bottom-left of the transform.

Procedure

In the Graphical Data Mapping editor, open your converted map, and complete the following actions to resolve the conversion warning:

1. Select a transform with a warning.
2. Review the transform properties in the Properties view to ensure that the transformation logic is correct.
3. Accept or reject the proposed transform. Select one of the following actions:
 - a. Select **Accept Converted Transform** to remove the conversion warning and accept the transformation logic implemented by the conversion process.
 - b. Select **Reject Converted Transform** to remove the warning. The conversion process replaces the converted transform with a **Task** transform. Then, manually re-create the transformation logic.
 - 1) Select the **Task** transform.
 - 2) In the **Task** transform Properties view, check out the information provided in the **Documentation** tab.

3) Edit the **Condition** property of the transform and manually define the transformation logic.

In the **Documentation** tab of the transform, the map conversion process includes the mapping structure from your legacy message map.

- c. Select **Accept Converted Transforms** to remove all the conversion warnings in your map and accept all the transformation logic implemented by the conversion process.
- d. Select **Reject Converted Transforms** to remove all the conversion warnings. The conversion process replaces each transform with a **Task** transform. Then, manually re-create the transformation logic.

What to do next

Check the following conversion warning descriptions and the actions you can take to resolve them:

Conversion warning description	Action to resolve it
The message map references the function {0}. The numbers of inputs provided by the message map and expected by the function definition are {1} and {2}, respectively. The extra input is:{3} Refer to {4} to resolve the discrepancy.	Review the input connections to the transform and remove the one that is not needed. For more information, see “Converting a legacy message map that includes ESQL mapping functions” on page 334.
The message map references the function {0}. The numbers of inputs provided by the message map and expected by the function definition are {1} and {2}, respectively. Refer to {3} and provide values for all inputs.	Review your input connections and add the missing ones. The transform is missing at least one input connection For more information, see “Converting a legacy message map that includes ESQL mapping functions” on page 334.
The signature for Java method {0} in class {1} is not found in the workspace within the referenced projects and the Java build path. Refer to {2} for the original mapping and make sure the Java method is accessible from the map.	Define a project reference from the project containing the message map to the Java project that contains the Java class and method.
Review the definition being used as the cast for {0}, since it is defined in the following schemas. {1}	Check that the schema used to cast an xsd:any element or attribute is the correct one.
The connection to input {0} was converted from expression {1}. There is more than one element that can be identified as {1}. Review the input connection, and move the connection to another {0} if appropriate.	The transform was converted from a mapping with an ambiguous input.
The transformation code was converted from {0}. It is advised to review to ensure equivalent transformation outcome. Refer to {1} for the original function usage.	Review the transformation logic.

Conversion warning description	Action to resolve it
<p>The function {0} was converted from {1}. It is advised to review to ensure equivalent transformation outcome. Refer to {2} for the original function usage.</p>	<p>Review the transformation logic.</p>
<p>The transform was created by converting a non-trivial mapping in an older version of message map. User review is advised to ensure equivalent transformation outcome. Refer to {0} for the original mapping.</p>	<p>Review the transformation logic.</p>
<p>The transform was converted from a call to a user-defined ESQL function. All inputs of this transform are optional, if the user-defined ESQL function returns NULL when none of the inputs are present, an unexpected empty output element could be produced. To avoid an empty output element, consider setting a condition on the transform as shown below so that the transform will only be performed when at least one input exists: t{0} Refer to {1} for the original mapping.</p>	<p>A condition might be needed on the transform.</p>
<p>The transform was converted from an XPath expression involving an XML type cast. Before conversion, XPath functions were implemented in the runtime using equivalent ESQL functions. Some of these ESQL functions have more lax value typing than defined in the XPath specification of the function. The Graphical Data Mapper provides conforming XPath functions. As a result, the transform may fail at runtime with invalid value for type casting issues. In particular, the ESQL equivalent function may have provide a default value when the parameter value was empty. To resolve these type of issues, add conditions to detect and prevent the values that are invalid for the XPath function parameters being passed to the function. For example, the following condition will make the transform executed only when all inputs exist: \t{0} Refer to {1} for the original mapping.</p>	<p>A condition might be needed on the transform.</p> <p>For more information, see “Converting a legacy message map that includes ESQL mapping functions” on page 334.</p>
<p>The XPath expression was converted from an expression involving a call to the msgmap:exact-type function. It is advised to review to ensure equivalent transformation outcome. Refer to {0} for the original function usage.</p>	<p>Review the XPath expression.</p> <p>For more information, see “Converting a legacy message map that includes calls to message map functions” on page 336.</p>

Conversion warning description	Action to resolve it
The transform involves the following input element which involves a text content: Determine if the element or its text content is the intended mapping input or output. If necessary, modify the corresponding connection. Refer to {0} for the original mapping.	Review the XPath expression. For more information, see “Changes in behavior in message maps converted from legacy message maps” on page 322.
The transform involves the following output element which involves a text content: Determine if the element or its text content is the intended mapping input or output. If necessary, modify the corresponding connection. Refer to {0} for the original mapping.	Review the XPath expression. For more information, see “Changes in behavior in message maps converted from legacy message maps” on page 322.
The XPath function call was converted from {0}. Ensure the argument number, order and type matches the supported function specification. Refer to {1} for the original function usage.	Review the XPath expression against the XPath specification.

Managing conversion errors on converted legacy message maps

The conversion process creates a **Task** transform with an error when it cannot automatically convert a legacy message map transformation. You can use any of the transforms in the Graphical Data Mapping editor to reconstruct an equivalent transformation. Typically, you might use a **Custom XPath** transform, a **Custom Java** transform, or a **Custom ESQL** transform to re-create the transformation logic.

About this task

These errors are displayed in the Graphical Data Mapping editor as **Task** transforms marked with an error.

You can see the error description associated with the conversion in the Problems view or by hovering over the red exclamation symbol associated to a **Task** transform.

Procedure

In the Graphical Data Mapping editor, open your converted map, and complete the following actions to resolve the error:

1. Select a **Task** transform.
The transform properties are displayed in the Properties view.
2. In the Properties view, click the **Documentation** tab to review details about the mapping structure that was not re-created by the conversion process.

In the **Documentation** tab of the transform, the map conversion process includes the mapping structure from your legacy message map. For example, you can find the following description for conversion of unsupported legacy transforms:

The expression {0} used for {1} cannot be automatically converted into a supported transform. \n Refer to {2} and manually create a transform with an equivalent expression.

3. Change the **Task** transform to a **Custom XPath** transform, a **Custom Java** transform, or a **Custom ESQL** transform.

Click the arrow in the transform box, and select a transform from the list of available ones. For more information, see Chapter 9, “Editing message maps,” on page 61.

4. Configure the transform properties to manually re-create the transformation logic.

What to do next

Check the following conversion error descriptions and the actions you can take to resolve them:

Conversion error descriptions	Action to resolve it
The statement {0} cannot be automatically converted into a supported transform. Refer to {1} and manually create mappings to perform the corresponding transformation.	<p>Check the Documentation tab of the Task transform to get details of the original expression.</p> <p>Use a Custom XPath transform, a Custom Java transform, or a Custom ESQL transform to re-create the transformation logic.</p>
The expression {0} cannot be automatically converted into a supported transform. Refer to {1} and manually create a transform with an equivalent expression.	<p>Check the Documentation tab of the Task transform to get details of the original expression.</p> <p>Use a Custom XPath transform, a Custom Java transform, or a Custom ESQL transform to re-create the transformation logic.</p>
The XPath function call {0} for {1} is not supported. Refer to {2} and manually create a transform with an equivalent expression.	<p>Check the Documentation tab of the Task transform to get details of the original expression.</p> <p>Use a Custom XPath transform, a Custom Java transform, or a Custom ESQL transform to re-create the transformation logic.</p>
The message map contains an erroneous expression. An attempt is made to convert the expression. Refer to {0} and review the converted expression.	<p>Check the Documentation tab of the Task transform to get details of the original expression.</p> <p>Use a Custom XPath transform, a Custom Java transform, or a Custom ESQL transform to re-create the transformation logic.</p>

Conversion error descriptions	Action to resolve it
The message map references a user-defined ESQL routine {0} which has an argument {1} of REFERENCE type. A routine with an argument of REFERENCE type cannot be called from the converted map. Refer to {2} and manually create a transform with an equivalent result.	<p>Check the Documentation tab of the Task transform to get details of the original expression.</p> <p>Use a Custom XPath transform, a Custom Java transform, or a Custom ESQL transform to re-create the transformation logic.</p> <p>For more information, see “Converting a legacy message map that includes user-defined ESQL procedures” on page 333.</p>

Converting a legacy message map that includes transformations of the local environment tree or *xsd:any* elements

Before you convert a legacy message map that includes transformations of the local environment tree or *xsd:any* elements, you must provide the XML schema of the input and output data structure in a library. The library with the schemas must be visible by the project hosting the imported legacy message map.

About this task

In a message map, the Variables folder in the local environment tree is represented by *xsd:any*. In IBM Integration Bus, you must qualify the Variables folder to provide the elements for your map. For more information, see “Mapping data in the local environment tree” on page 111.

When you convert a legacy message map, you can encounter any of the following conversion behaviors:

- You have an *xsd:any* element and the schema model associated to the message set that you use to qualify it in your map. The conversion process casts the *xsd:any* element to the schema model automatically.
- You have an *xsd:any* element and no schema model describing its structure. The conversion process fails the first time. You must define the model and run again the conversion process.
- You have a legacy message map where you edit the path expression in your map to define the element that must be read to qualify the *xsd:any* element. The conversion process fails the first time. You must define the schema model and run again the conversion process.

Procedure

In the Application Development view, complete the following steps to convert a legacy message map that includes transformations of the local environment tree and of *xsd:any* elements:

1. If you use the local environment tree in your legacy message map transformations, create an XML schema model in a library project. The model must define the Variables folder data structure.
2. If your input message or your output message include *xsd:any* elements that you use as part of your transformations, define the XML schema model for each one.

3. Start the conversion process: Right-click the message map that you want to convert, and click **Convert Message Map from .msgmap to .map**.
4. Open the converted map in the Graphical Data Mapping editor: Double-click your new message map.
5. Review the conversion errors and build the list of unresolved elements. For more information, see “Managing conversion errors on converted legacy message maps” on page 330.
6. Create XML schema models for each unresolved data structure in a library that is referenced by the project hosting your converted map.
For more information, see “Mapping data in the local environment tree” on page 111.
7. Delete the converted legacy message map.
8. Rename the converted legacy message map by removing *_backup*.
9. Rerun the conversion process: Right-click the message map that you want to convert, and click **Convert Message Map from .msgmap to .map**.
The conversion process casts automatically your *xsd:any* elements. Any related errors to unresolved elements disappear.

What to do next

Continue converting your legacy message map. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

Converting a legacy message map that includes user-defined ESQL procedures

When you convert a legacy message map that includes ESQL procedures, the conversion process converts each ESQL procedure to an equivalent **Custom ESQL** transform that invokes the ESQL. A **Task** transform is added to your converted map when an ESQL procedure does not fulfill the requirements to be called from a map on a Mapping node.

Before you begin

Read the section **Requirements for ESQL modules called from a graphical data map** in the following topic: “Custom ESQL” on page 208

Procedure

Check the conversion process behavior when you convert a legacy message map to a message map that includes ESQL procedures:

1. By default, the conversion process converts an ESQL procedure to a **Custom ESQL** transform. For more information, see “Custom ESQL” on page 208.

Each converted ESQL procedure is deployed as source. If you are not using IBM Integration Bus application and library projects to store your ESQL procedures, the ESQL procedures must be uniquely named because they are deployed independently to the same integration server.

2. The conversion process converts an ESQL procedure that uses the ESQL REFERENCE data type to a **Task** transform.

You must replace the **Task** transform with a **Custom XPath** transform, a **Custom Java** transform, or a **Custom ESQL** transform that provides equivalent function.

3. The conversion process converts an ESQL procedure that has an INOUT argument to a **Custom ESQL** transform where the INOUT argument is converted as an IN argument.

You can replace the **Custom ESQL** transform with a **Custom XPath** transform, or a **Custom Java** transform when the default conversion transform is not valid.

What to do next

Continue converting your legacy message map. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

Converting a legacy message map that includes ESQL mapping functions

When you convert a legacy message map that includes ESQL mapping functions, the conversion process converts some ESQL functions to equivalent XPath 2.0 functions (`fn:functionName`), or to cast type functions (`xs:type`). A **Task** transform is added to your converted map when there is no automatic conversion for an ESQL function.

Procedure

Check the conversion process behavior when you convert a legacy message map to a message map that includes ESQL mapping functions:

1. When a legacy message map includes calls to predefined ESQL mapping functions, each ESQL function call is converted to an XPath expression, cast type function, or to a **Custom XPath** transform in the converted map. For each expression, `xs:type` function, or **Custom XPath** transform in the converted map, complete the following steps:

- a. Check that the expression, `xs:type` function, or **Custom XPath** transform re-creates the required behavior.

If your ESQL mapping function has optional input parameters, you must implement conditions to handle this situation. By default, the conversion process assumes that all input parameters are mandatory.

- b. For each expression, `xs:type` function, or transform, check that the correct number of inputs is connected.

In earlier releases of WebSphere Message Broker Version 8, the number of inputs wired to a transform and required to implement a transformation in a legacy message map was not enforced. When the Graphical Data Mapping editor converts a transform that includes an ESQL mapping function, it creates an XPath function that conforms to the XPath 2.0 specification, and wires the input elements to the transform as defined in the legacy message map. As a result, a converted map might have more inputs than the XPath expression requires, or less inputs than the ones required to perform the calculation. Consequently, the converted map will fail to run when you deploy it.

2. If there is no XPath equivalent of an ESQL mapping function, the function is replaced with a **Task** transform in your converted map. You must replace each of these **Task** transforms with a **Custom XPath** transform, a **Custom Java** transform, or a **Custom ESQL** transform that re-creates the required behavior.

- a. Check the **Documentation** properties of the transform in the converted map for more information on how the ESQL function was implemented in your legacy message map.

The following ESQL mapping functions that you can use in a legacy message map have no XPath equivalent in message maps:

- Certain mathematical functions:
 - ACOS
 - ASIN
 - ATAN
 - ATAN2
 - BITAND
 - BITNOT
 - BITOR
 - BITXOR
 - COS
 - COSH
 - COT
 - DEGREES
 - EXP
 - LN
 - LOG
 - LOG10
 - MOD
 - POWER
 - RADIANS
 - RAND
 - SIGN
 - SIN
 - SINH
 - SQRT
 - TAN
 - TANH
- Decimal function:
 - TRUNCATE
- Certain String functions:
 - LTRIM
 - RTRIM
 - TRIM-LEADING
 - TRIM-TRAILING
 - REPLICATE
 - SPACE
 - TRIM-BOTH(Singleton FROM Source)
The simple form TRIM-BOTH (Source) is converted.
- Certain field functions:
 - ABITSTREAM

- BITSTREAM
- SAMEFIELD
- Certain date time functions:
 - TIMESTAMP
 - CURRENT-GMTDATE
 - CURRENT-GMTTIME
 - CURRENT-GMTTIMESTAMP
- All INTERVAL- functions
- The ESQL LIKE function
- The ESQL FOLLOWING form of the ESQL POSITION function
- All SQL functions
- The UIDASCHAR and UIDASBLOB functions

What to do next

Continue converting your legacy message map. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

Converting a legacy message map that includes calls to message map functions

The conversion process converts a number of supported calls to predefined legacy message map functions (`msgmap:functionName`) to an XPath expression, or a **Custom XPath** transform.

About this task

If there is no XPath equivalent of a message map function, the conversion process creates a **Task** transform instead.

The following legacy message map functions have no XPath equivalent:

- `msgmap:element-from-bitstream`
- `msgmap:cdata-element`
- `msgmap:db-path`

Procedure

Complete the following steps to convert a transformation that includes a call to a message map function that is not converted automatically:

1. Select a **Task** transform.
2. In the **Task** transform Properties view, check out the information provided in the **Documentation** tab.
3. Replace the **Task** transforms with any of the available transforms. Consider using a **Custom XPath** transform, a **Custom Java** transform, or a **Custom ESQL** transform.
4. Manually re-create the equivalent transformation logic of the message map function.

What to do next

Continue converting your legacy message map. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

Converting a legacy message map that includes relational database operations

You must convert manually a transformation that includes relational database operations.

About this task

A legacy Mapping node connects to a database via ODBC. A database configuration file describes the ODBC configuration to your database. You configure the database information in your legacy message map by setting the **Data Source** property in the map.

From WebSphere Message Broker Version 8 onwards, the Mapping node connects to databases via JDBC type 4 connections. You configure the database connection details in a JDBCProvider configurable service.

Database transforms in a message map use support for JDBC connections that are built into IBM Integration Bus run time.

For information on constructing database transforms and configuring your JDBC database connection, see Chapter 19, “Mapping database content,” on page 159.

Procedure

To re-create the transformation logic of a transformation that includes a relational database operation, you can use any of the following transforms when replacing the **Task** transform in the new message map:

1. “Select” on page 227
2. “Update” on page 228
3. “Delete” on page 216
4. “Database Routine” on page 216

What to do next

Continue converting your legacy message map. For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.

Converting a legacy message map that is called from an ESQL statement in a Compute node

To convert a legacy message map that is called from an ESQL CALL statement in a Compute node, you must implement your ESQL logic in a Mapping node, and call the converted legacy message map through a **Submap** transform.

Before you begin

Before you convert a legacy message map, you must complete one of the following tasks:

- Import your resources from WebSphere Message Broker Version 6.1 or WebSphere Message Broker Version 7. For more information, see Importing resources from previous versions.
- Migrate the WebSphere Message Broker Version 6.1 or WebSphere Message Broker Version 7 Toolkit development resources. For more information, see Migrating development resources to IBM Integration Studio Version 10.0.

Review “Changes in behavior in message maps converted from legacy message maps” on page 322 and “Considerations for mapping messages modeled in message sets” on page 13.

About this task

In WebSphere Message Broker Version 8 and later versions, a message map, that is a graphical data map, cannot be called from ESQL statements in a Compute node. For compatibility with earlier releases of the product, you can still deploy and run a legacy message map in a BAR file. You must include the whole message flow to the BAR file and set the compile and in-line option.

If you need to modify the logic in your legacy message map, you must convert the map to a message map and modify your message flow logic.

Procedure

To convert a legacy message map that is called by an ESQL CALL statement in a Compute node, complete the instructions outlined in any of the following steps:

- Replace the Compute node and the legacy message map with a new message map.
 1. Convert the legacy message map (.msgmap) to a message map (.map). For more information, see “Converting a message map from a .msgmap file to a .map file” on page 325.
 2. Replace each Compute node that includes a call to the legacy message map with a Mapping node.
 3. Create a new message map for each Mapping node.

In this message map, complete the following steps:

 - a. Create a message model that defines the overall input and output data structure to the Mapping node.
 - b. Define the transforms that implement equivalent logic to the ESQL routine.
 - c. Replace the CALL statement with a **Submap** transform. For more information, see “Submap” on page 227.
 - d. Configure the converted legacy message map as the mapping routine of the **Submap** transform.
- Replace the called message map with an ESQL function that provides equivalent logic. It might be possible to use support pac "IA9Y: Map to ESQL Plug-In" to help with the conversion of the message map to ESQL <http://www.ibm.com/support/docview.wss?uid=swg24017156>.

What to do next

Deploy and test your message flow. For more information, see Deploying solutions.

Replacing a WebSphere Message Broker Version 7 Mapping node

A WebSphere Message Broker Version 7 Mapping node cannot reference a graphical data map (.map file). If you want to reference a graphical data map, you must replace the WebSphere Message Broker Version 7 Mapping node.

Before you begin

Before you replace a Mapping node, you must complete the following task:

- Import your resources from WebSphere Message Broker Version 7. For more information, see *Importing resources from previous versions*.

You might also need to complete the following task:

- Convert a message map to a graphical data map. For more information, see *“Converting a message map from a .msgmap file to a .map file”* on page 325.

About this task

If you import your integration solutions from WebSphere Message Broker Version 7, you can still compile and deploy message flows that use WebSphere Message Broker Version 7 Mapping nodes. However, in WebSphere Message Broker Version 8 and later, Mapping nodes that were created in WebSphere Message Broker Version 7 are accessible in read-only mode and cannot be modified.

If you want to reference a graphical data map (.map file) in a Mapping node, you must use the Mapping node from WebSphere Message Broker Version 8 or later.

To replace a WebSphere Message Broker Version 7 Mapping node by using the IBM Integration Studio, complete the following steps:

Procedure

1. In the Application Development view, double-click a message flow that contains one or more WebSphere Message Broker Version 7 Mapping nodes.
The message flow opens in the Message Flow editor.
2. In the Message Flow editor, identify a WebSphere Message Broker Version 7 Mapping node that you want to replace.
3. In the Palette, expand the **Transformation** section, and then drag a new Mapping node from the Palette to the canvas of the Message Flow Editor.
A new Mapping node is added to your message flow, and is assigned a default name.
If you rename the node, the name that you choose must be unique in the message flow.
If you do not change the default name at this time, you can change it later. For more information, see *Renaming a message flow node*.
4. Select your new Mapping node.
The node properties are displayed in the Properties view.
5. In the Properties view, a default value is entered in the *Mapping routine* property, and must be replaced by choosing one of the following actions:
 - To reference an existing graphical data map, click **Browse...** to locate it, or specify your .map file in the format `{BrokerSchemaName}:MapName. {default}` indicates that no Broker schema is used by the graphical data map. For more information, see *“Referencing an existing message map from a Mapping node”* on page 175.

- To create a new graphical data map, double-click the Mapping node, or right-click the Mapping node and click **Open Map**. For more information, see “Creating a message map from a Mapping node” on page 49.
6. Move the existing connections from your WebSphere Message Broker Version 7 Mapping node to your new Mapping node. For more information, see Connecting message flow nodes.
 7. Select your WebSphere Message Broker Version 7 Mapping node, and press the delete key (**del**) to remove it from your message flow. For more information, see Removing a message flow node.
 8. Repeat steps 3 on page 339 through 7 for each WebSphere Message Broker Version 7 Mapping node that you want to replace in your message flow.

Results

You have removed your WebSphere Message Broker Version 7 Mapping nodes and replaced them with new Mapping nodes.

What to do next

Deploy and test your message flow. For more information, see Deploying solutions.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Important: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at Copyright and trademark information (www.ibm.com/legal/copytrade.shtml).



Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

User Technologies Department (MP095)
IBM United Kingdom Laboratories
Hursley Park
WINCHESTER,
Hampshire
SO21 2JN
United Kingdom

- By fax:
 - From outside the U.K., after your international access code use 44-1962-816151
 - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
 - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
 - IBMLink: HURSLEY(IDRCF)
 - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.



Printed in USA