WebSphere Enterprise Service Bus

Development Guide

Version 7 Release 5.1

SC34-7206-03

Note

Before using this information and the product it supports, read the information in "Notices" on page 113.

This edition applies to version 7, release 5, modification 1 of WebSphere Enterprise Service Bus (product number 5724-I82) and to all subsequent releases and modifications until otherwise indicated in new editions.

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 2007, 2011.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

Chapter 1. Introduction 1

Chapter 2. Common Usage Patterns . . 3

Chapter 3. Service Component Architecture 11
Module 11
SCA implementation in Integration Designer . . . 12
Library 15

Chapter 4. Mediation Flow Component 17
Service Message Objects 19
SMO structure 19
Data representation of SMO 22

Chapter 5. Routing Messages Within a Mediation Flow 23
Example scenarios 24

Chapter 6. Data Transformation 29
Business Object Map and XSL Transformation mediation primitives 29
Comparison between the XSLT and BO Mapper mediation primitives 31
Example scenarios 31
Custom mediation 35
Message Element Setter and Data Handler mediation primitives 38
WebSphere Transformation Extender 42

Chapter 7. Invocation of Services . . . 43
Service Invoke and Callout nodes 43
Comparison of the Service Invoke mediation primitive and the Callout node 43
Retry 44
Service invocation styles 45
Synchronous SCA invocation 46
Asynchronous invocation 48
Invocation from within the mediation flow component 52
Dynamic invocation 53
Dynamic override of a fixed endpoint 54
Dynamic invocation with a target import . . . 55
Pure dynamic invocation 56
Dynamic endpoint selection 57

Chapter 8. Service gateway 63
Patterns Explorer 67
Dynamic service gateway pattern 69
Example scenario 70
Static service gateway pattern 70
Example scenario 71
Proxy gateway pattern. 71
Configuration store 72
Virtual services 73
Introduction to Business Space 74
Proxy gateway widget 75
Example scenario 77
Selecting service gateway patterns 78

Chapter 9. Aggregation. 79
Aggregation patterns 84
Aggregation of data from multiple sources . . . 84
Batch processing with message enrichment . . . 85
Batch processing requiring embedded aggregation 90
Nested aggregation 91
Error handling within an aggregation block . . . 93
Asynchronous parallel processing 94
Performance considerations when using aggregation 95

Chapter 10. Error handling in the mediation flow component 99
Error flow 102
Common patterns of usage for error handling . . 103

Chapter 11. Testing and Debugging 107
Unit testing 107
Cross-component trace 109
Testing Web services 110
Debugging mediation flows 112

Notices 113
Programming interface information 115
Trademarks 115

Sending your comments to IBM . . . 117

© Copyright IBM Corp. 2007, 2011

Figures

1. Shows the connection between the service requester, service provider and ESB 1
2. WebSphere ESB, installed on top of WebSphere Application Server 2
3. Simple service proxy pattern 4
4. Service selector pattern 5
5. Service translator pattern 6
6. Service gateway pattern 7
7. Message enrichment pattern 8
8. Batch aggregation pattern 9
9. Multiple source aggregation pattern 9
10. Structure of a module 12
11. Mediation module 13
12. Mediation flows 18
13. Overview of SMO structure 20
14. Mediation primitive 23
15. Branched mediation primitive 23
16. Message filter mediation primitive 24
17. Type Filter mediation primitive 25
18. Flow Order mediation primitive 26
19. Custom mediation primitive 26
20. Hover pane showing the source and target XML message type 30
21. Message Structure mapping using an XSLT mediation primitive within a mediation flow component 32
22. Message Structure mapping within Integration Designer 33
23. Service enrichment using a BO mapper within a mediation flow component 33
24. Format date function 34
25. The configuration of the format date transformation 34
26. Custom transformation in a BO map 34
27. Java code in the Properties view 34
28. Hover pane showing the message type and structure 36
29. Hoverpane showing the body structure of the SMO 37
30. Message Element Setter in a mediation flow component 38
31. Message Element Setter properties within Integration Designer 39
32. Data Handler Configuration 40
33. Data Handler within a mediation flow component 41
34. Data Handler in Integration Designer 41
35. WebSphere ESB shows how a component can be called synchronously or asynchronously . . 46
36. Synchronous SCA invocation 47
37. Asynchronous with deferred response SCA invocation 49
38. Car insurance comparison website using asynchronous with deferred response invocation 49
39. Asynchronous with callback SCA invocation 51
40. Dynamic override of a fixed endpoint using an Endpoint Lookup mediation primitive . . . 55
41. Dynamic invocation with a target import 56
42. Pure dynamic invocation 57
43. Endpoint Lookup within a mediation flow 59
44. Endpoint Lookup mediation primitive properties 59
45. Try alternate endpoint property of the Callout node 59
46. SLA Endpoint Lookup 61
47. General pattern for a service gateway module 64
48. Five structures that represent raw data 64
49. DataHandler mediation primitive used in a service gateway 66
50. Automatic parsing used in a service gateway 66
51. Pattern Explorer in IBM Integration Designer 67
52. Pattern Explorer configuration view 68
53. Pattern Explorer summary information 69
54. Common pattern that is used for a dynamic service gateway 70
55. Static service gateway pattern 71
56. Proxy gateway 72
57. High level interactions of the proxy gateway, the configuration store and the web based interface 73
58. Proxy gateway widget in Business Space 75
59. Select a proxy gateway widget 76
60. Editing a new proxy group 76
61. Editing a new proxy group 77
62. The Fan Out mediation primitive properties view 80
63. The Fan In mediation primitive properties view 81
64. Example of an aggregation block 82
65. Shared context example 82
66. Aggregation Fan Out branching 85
67. Shared context response messages 85
68. Batch systems business objects 86
69. The Customer business object 86
70. Batch request structure 86
71. Batch response structure 86
72. DispatchOrder array 87
73. Fan Out/Fan In flow detail 87
74. Fan Out/Fan In properties view 88
75. Message Element Setter properties 88
76. Transform inline map 89
77. Message Element Setter aggregateResult properties 89
78. SMO mapping 89
79. Flow input and output 90
80. HR and Payroll Record 90
81. The inner and outer flow 90
82. EmployeeRecord array 91
83. PayslipStatus object 91
84. Business objects for batch order example 92
85. Result of batched order example 92

© Copyright IBM Corp. 2007, 2011

86. Aggregation levels 92
87. Shared outlet responses 93
88. Aggregation block 93
89. Branched aggregation block 94
90. Service invoke mediation primitive property details 94
91. Setting the Asynchronous invocation qualifier 95
92. Setting the number of requests that should be processed 95
93. Deleting the message body 96
94. Aggregation context mapping 97
95. The failInfo element in the message context 99
96. Runtime failure and rollback transaction 100
97. Example of a modelled fault 101
98. Stop mediation primitive. 102
99. An example error flow to return an unmodeled fault 103
100. An example error flow to return a modelled fault 103
101. Synchronous invocation error handling 104
102. Asynchronous invocation error handling 104
103. Asynchronous invocation error handling without queue 2 105
104. Asynchronous invocation error handling 105
105. Synchronous and asynchronous invocation error handling 106
106. Asynchronous and synchronous invocation error handling 106
107. Integration Test Client properties 108
108. Input parameters 108

Chapter 1. Introduction

This development guide is for enterprise architects, integration architects and developers who are responsible for implementing integration and connectivity solutions. It begins with the high-level concepts and abilities that will help you to understand the capabilities and usage patterns of an Enterprise Service Bus (ESB). The development guide then describes how to implement the capabilities and patterns on WebSphere® Enterprise Service Bus.

Service-oriented architecture

Service-oriented architecture (SOA) is an architectural approach that supports the flexible integration of IT systems. It does this by providing a method to disconnect the service provider (service capability which delivers a defined action) from the service requester (client that sends the request). This enables the substitution of one service provider for another, without the requester being aware of the change and without the need to alter the architecture as shown in Figure 1.

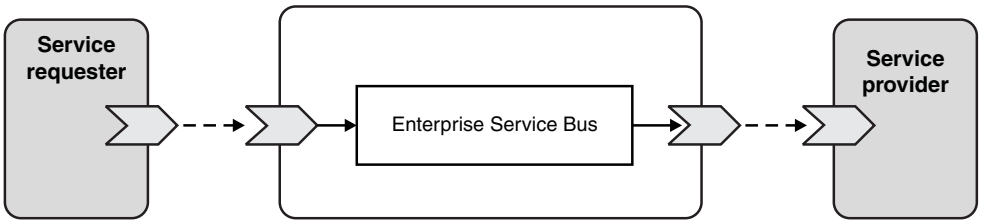


Figure 1. Shows the connection between the service requester, service provider and ESB

By adopting an SOA approach and implementing it using supporting technologies, companies can build flexible systems that implement changing business processes quickly, and make extensive use of reusable components.

Services are the building blocks of SOA, providing function as a base point for building distributed systems. Services can be invoked independently by either external or internal service consumers to process single functions, or can be linked together to form more complex functionality and so quickly devise new functionality.

The Enterprise Service Bus

An ESB is used to connect a service requester to a service provider so that messages can be routed between the two platforms. The ESB is a collection of software components that manage messaging from one part of the network to another. The ESB handles mismatches between the requesters and providers, including protocol, interface or quality of service mismatches.

The ESB processes messages exchanged between the service endpoints. A service endpoint is a location where the service is available on the network. In contrast with regular business application components, the ESB is concerned with the flow of the messages through the infrastructure and not just with the business content of the messages. Rather than performing business functions, the ESB performs

© Copyright IBM Corp. 2007, 2011

mediation capabilities including routing, transformation, and logging operations on the messages.

WebSphere Enterprise Service Bus

WebSphere ESB is designed to provide an ESB for IT environments built around open standards and SOA. It delivers easy to use functionality that is built on the messaging and web services technologies of WebSphere Application Server.

WebSphere Application Server is the foundation for WebSphere ESB, providing not only the required quality of service, the J2EE runtime environment, and the messaging infrastructure, but also providing broad support for open standards and web services. WebSphere ESB is built on the Network Deployment product and provides a wide range of capabilities for large enterprise networks, including clustering, failover, and scalability features.

IBM® Business Process Manager extends the capabilities provided in WebSphere ESB to allow business applications to be developed.

IBM Integration Designer, is the tool for use with WebSphere ESB. It is designed to build applications that will be deployed to either WebSphere ESB or IBM Business Process Manager .

WebSphere ESB supports connectivity between endpoints through a variety of protocols and application programming interfaces (APIs) such as:

- WebServices SOAP 1.1 and 1.2
- Java Message Service (JMS) 1.1
- WebSphere MQ
- Enterprise Java Bean (EJB) Stateless Session Bean
- HTTP
- Enterprise Information Systems (EIS) integration using WebSphere Adapters

Figure 2 shows how WebSphere ESB is built on top of the WebSphere Application Server. Because of this, WebSphere ESB can provide smooth interoperability with other products in the WebSphere portfolio, including WebSphere MQ and WebSphere Message Broker.

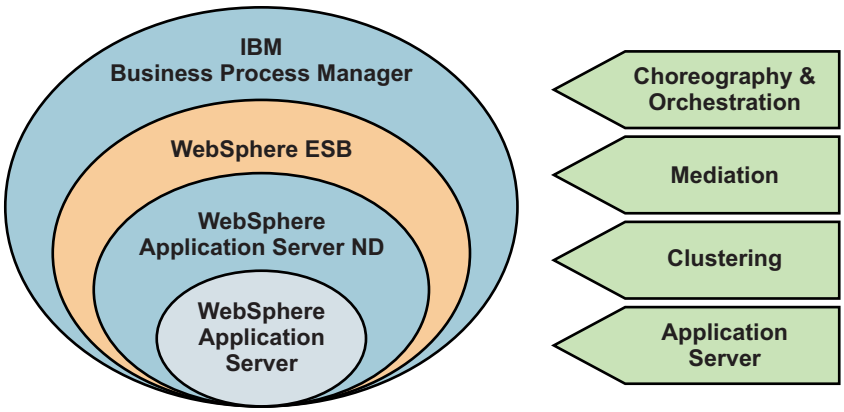


Figure 2. WebSphere ESB, installed on top of WebSphere Application Server

Chapter 2. Common Usage Patterns

There are seven common usage patterns that help to speed up the process of implementing connectivity and integration solutions. The patterns summarize some of the more common solutions in the connectivity space. These patterns are common usage scenarios that all Enterprise Service Bus (ESB) technologies should support.

"Simple service proxy pattern"

"Service selector pattern" on page 4

"Service translator pattern" on page 5

"Service gateway pattern" on page 6

"Message enrichment pattern" on page 7

"Batch aggregation pattern" on page 8

"Multiple source aggregation pattern" on page 9

Simple service proxy pattern

The simple service proxy pattern creates a one-to-one mapping between a service requester and service provider. By placing an ESB between the two services it hides the real location of the service provider. The simple service proxy pattern is also of particular value in providing a point for access control, request tracking or auditing.

You can use this pattern when:

- You want to provide access to a service through a control point without exposing the actual location (endpoint address) of the service to the clients.
- You must apply some form of control (access management, authorization, auditing or logging) every time a service is accessed.

Figure 3 on page 4 shows the simple service proxy pattern. The target service is hidden by deploying an ESB mediation which implements the same interface. This mediation acts as a virtual service (or proxy) which redirects all requests to the real service provider.

A service registry is a repository of data describing a set of services and their associated properties and behaviors. This registry can be queried by the ESB to determine the endpoint of the service provider.

© Copyright IBM Corp. 2007, 2011

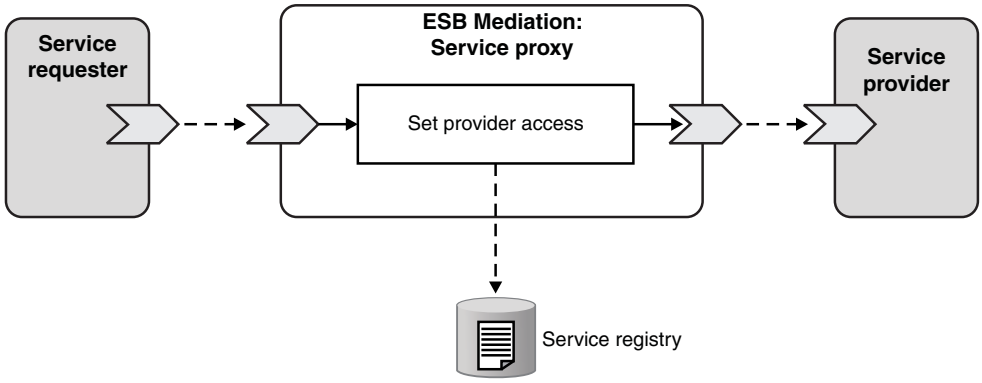


Figure 3. Simple service proxy pattern

For example, operations within a bank application often need to be audited as a legal requirement. The simple service proxy pattern can be used to implement this business requirement that ensures all requests are audited, and removes the need for the service provider to implement the capability.

Service selector pattern

The service selector pattern provides multiple implementations of the same service interface to be grouped behind a single endpoint address. Each implementation can have a different quality of service or behaviour, and each client request can be matched to a particular implementation determined by different criteria, such as the content of the message into the ESB.

You can use this pattern when:

- There are multiple implementations of a particular service all sharing the same interface.
- Different implementations provide different qualities of service.
- Requests from different clients are to be routed to a particular implementation in accordance with some set of predefined criteria.

Figure 4 on page 5 shows the service selector pattern, which implements the required logic to select a specific endpoint address based on the context or content of each client request.

A service registry is a repository of data describing a set of services and their associated properties and behaviors. This registry can be queried by the ESB to determine the endpoint of the service provider.

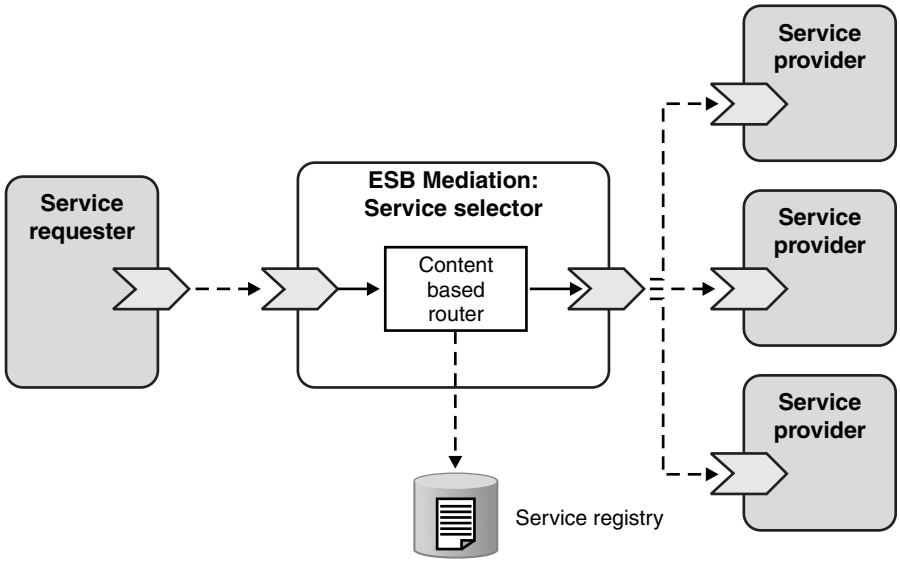


Figure 4. Service selector pattern

An example of when the service selector pattern would be used is if a fictitious company wants to differentiate itself from its competition by offering tiered levels of service. The company's goal is to offer delayed stock quotes to their standard customers and real-time quotes to their premium customers, who pay a fee.

The service requester application provides a query containing a stock symbol and customer ID to the content based router, which processes the query. The customer's subscription level is determined, and depending on the level of subscription, the query is routed to the appropriate service provider. The quote that is returned from the service provider is then returned to the client application. More on this scenario is found in the "Example scenarios" on page 24.

Service translator pattern

The service translator pattern provides access to a service implementation with a different interface or protocol than the interface or protocol used by the service requester. You can select operations to be restricted or restructured on some interfaces, and you can convert and format data for users of specific interfaces.

You can use this pattern when:

- You want to restrict access to certain functions on the user interface.
- The interface needs to be modified to conform to the requirements of certain departments.
- Requesting services expect to use different interfaces or protocols.

Figure 5 on page 6 shows the service translator pattern, which transforms the request and then routes it to the actual service provider. The mediation must also process the responses from the target service, transforming them back into the form expected by the client.

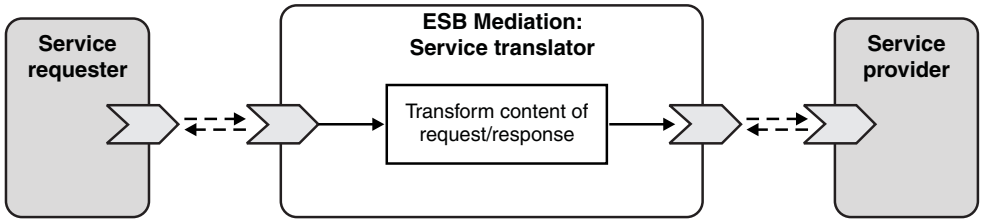


Figure 5. Service translator pattern

An example of the service translator pattern, is if a fictitious company, **Personal Holidays** book flights, and the flight availability must be checked against a fictitious airline company, **My Airways**, flight schedule. The Personal Holidays booking system represents information differently to the My Airways booking system. The service translator pattern within the ESB mediation is able to transform incoming messages from the service requester into the format that My Airways represent their information. More information about this scenario is found in the "Example scenarios" on page 31.

Service gateway pattern

The service gateway pattern introduces a point of control and can bring a range of services under the control of an ESB without the need to develop individual mediations for each service. The service gateway provides a single access point and acts as a proxy for multiple services. In addition, service gateways encapsulate transformations, routing, and common processing across all the services.

You can use this pattern when:

- You want to route messages to multiple endpoints using a variety of protocols.
- A set of services needs to be accessed in the same way without the need for multiple mediations.
- You want to provide generic processing (for example auditing, security enforcement point) across all the services.

Figure 6 on page 7 shows the service gateway pattern. Messages enter the gateway and the target service is identified. Common processing is completed (such as auditing) before the message is routed onwards to an end provider.

A service registry is a repository of data describing a set of services and their associated properties and behaviors. This registry can be queried by the ESB to determine the endpoint of the service provider.

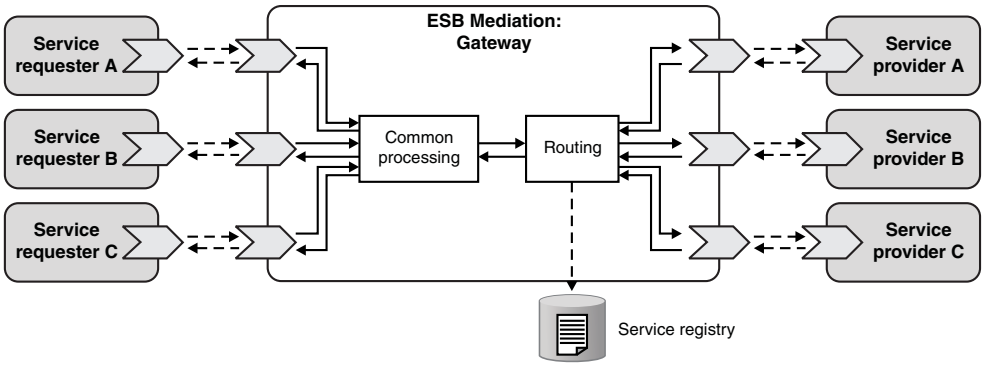


Figure 6. Service gateway pattern

For example a bank have a number of completely different services that need to be audited as a legal requirement. They want the same level of audit across all the services. The service gateway pattern is used to implement this business requirement.

Message enrichment pattern

The message enrichment pattern provides a method to modify or add to the existing content in a message before it is forwarded onto the service provider. An example of message enrichment is to retrieve data from a database and add the data to a given location in a message.

You can use this pattern when messages are sent, in order to obtain further information from a database.

Figure 7 on page 8 shows the message enrichment pattern. The message that arrives into the ESB does not contain all the data required by the service provider. Data is added by the ESB before the message is sent to the service provider. For example, the service requester only knows the customer ID and product ID. Message enrichment provides the full product and customer details for the order to be processed.

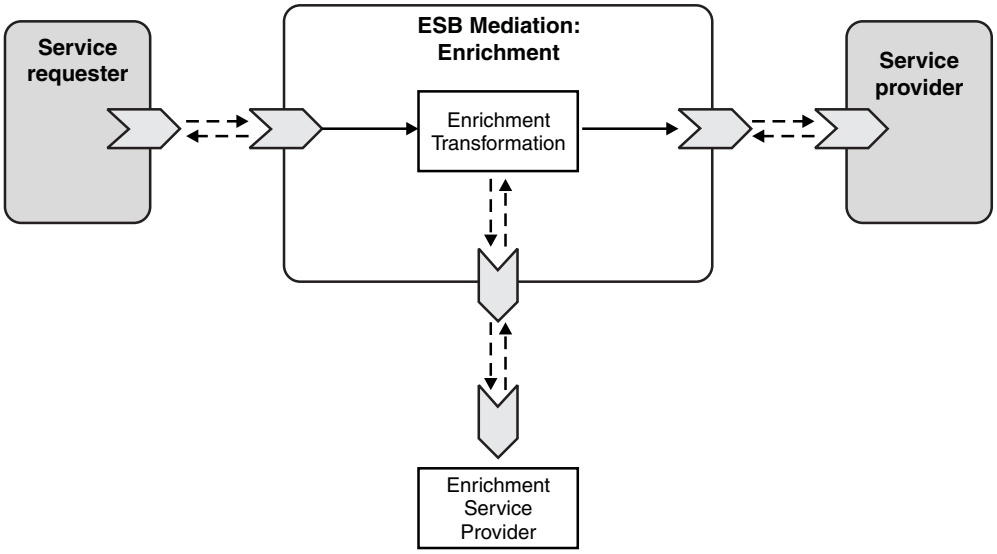


Figure 7. Message enrichment pattern

An example of when the service enrichment pattern can be used is if a particular field does not exist within a message structure. A Database Lookup mediation primitive can be used to establish the details of the field in any given instance. More information about this scenario is found in the "Service enrichment" on page 33.

Batch aggregation pattern

The batch aggregation pattern provides a method for an inbound request to be mapped into several individual outbound requests to the service provider. The responses from these requests can be collected (aggregated) into a single response for the original request.

You can use this pattern when there is a need for messages that include multiple records and each of which can be forwarded onto the service provider individually.

Figure 8 on page 9 shows the batch aggregation pattern, where the inbound message to the mediation is split into several outbound requests to the service provider, and then collected back into a single message to send back to the service requester. The service requester receives all the relevant information required in one batch message.

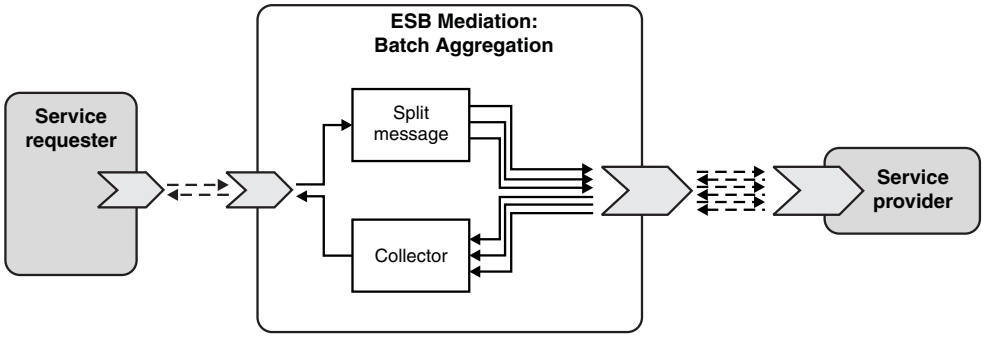


Figure 8. Batch aggregation pattern

Multiple source aggregation pattern

The multiple source aggregation pattern provides a method for an inbound request to be mapped into several individual outbound requests to multiple service providers. The responses from these requests can be collected (aggregated) into a single response for the original request.

You can use this pattern when a single message is sent into the ESB that requires two or more service providers to process the message, or parts of the message. An aggregated response is sent back to the service requestor.

Figure 9 shows the multiple source aggregation pattern, where the inbound message to the mediation is split into several outbound requests for several service providers. The results are then collected back into a single message to send back to the service requester. You can use this pattern if the inbound request needs information from many different service providers, for example a quote comparison web site.

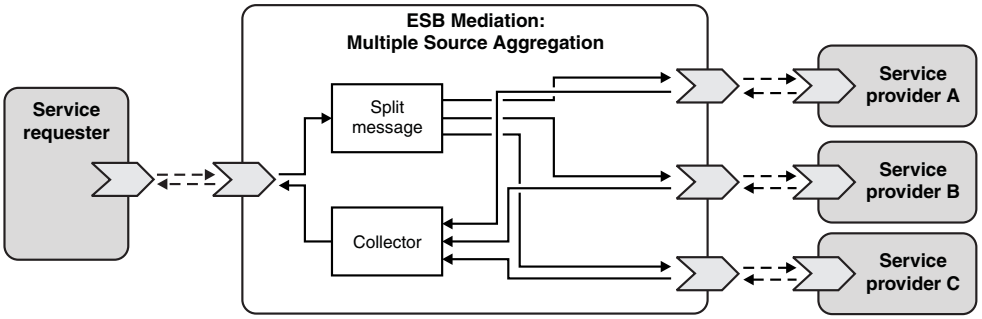


Figure 9. Multiple source aggregation pattern

Chapter 3. Service Component Architecture

Service Component Architecture (SCA) is a model to define and implement SOA based solutions. This section details the IBM implementation of SOA within WebSphere ESB, and its associated tooling, Integration Designer.

There are two defined sections within SCA:

1. Service assembly, which details the service definitions and relationships of the export, import, components and interfaces
2. Service implementation, which details the construction of the service definition and the transport of information through defined protocols (web services, JMS, MQ, adapters).

In an SOA framework, high-level business components are exposed as services. SCA provides a model for how the services in an SOA framework are structured.

Module

A module is a unit of deployment, where services are packaged together. Components within a module are assembled for performance, and can pass their data by reference. A module can be seen as a scoping mechanism; it sets an organizational boundary for services.

Modules can include:

- Service components, that hold the business function within a module. A service component configures a service implementation. The service component name inside a module is unique. A service component can have an arbitrary display name, that is more useful to a user. It is not necessary for each module to have a service component, but there is no limit to the number of service components per module.
- Imports, that are calls to services external to the module. It is not necessary for each module to have an import, but there is no limit to the number of imports per module. Imports are detailed later on in the chapter.
- Exports, that are used to expose components to callers that are external to the module. It is not necessary for each module to have an export, but there is no limit to the number of exports per module. Exports are detailed later on in the chapter.
- Interfaces, which can be associated with one or more components. The interfaces associated with a service component show the business operations associated with the component.
- Stand-alone references or reference applications that are not defined as SCA components (for example, JavaServer Pages). These applications can then interact with SCA components. There can be only one stand-alone reference per module.
- Other elements, for example WSDL files, Java classes, XSD files.

© IBM Corporation 2005, 2011 © IBM 2007, 2011

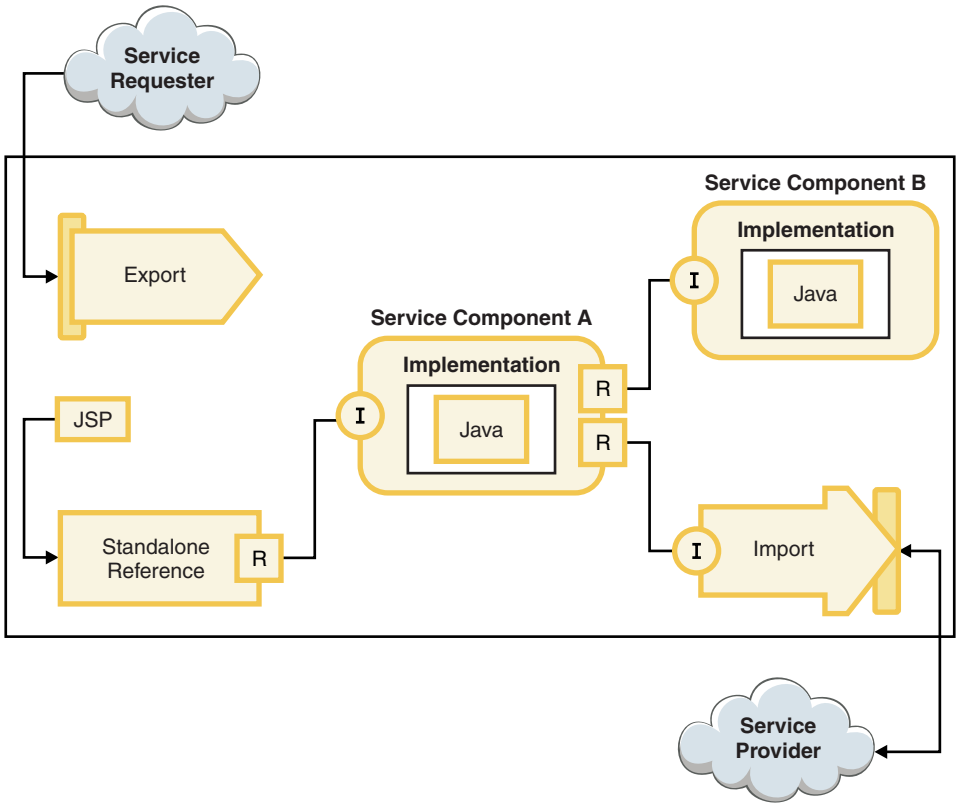


Figure 10. Structure of a module

Figure 10 shows an example module, which contains two service components, each containing an implementation. An export reveals Service Component A's capabilities to external service consumers. A stand-alone reference shows the component's capabilities to other non-SCA artifacts included within the deployment artifact such as JavaServer Pages (JSP). Service Component A may require the capabilities of two other services. One implemented by Service Component B, and one included using an SCA import. All exports, imports and components have interfaces that describe the inputs and outputs of these elements. References are used on service components and stand-alone references to declare the additional capabilities they require to function. Wiring is used between all the elements to represent their relationships.

SCA implementation in Integration Designer

You can build SCA modules within Integration Designer. Modules contain all the artifacts required to create a complete SCA application and they can be organized into projects. Additionally SCA libraries can be created in Integration Designer. You can build libraries to store artifacts, and you can share them between multiple SCA modules. Libraries are detailed later on in the chapter.

There are two types of modules that are supported by Integration Designer:

1. A business module that contains a choice of many component types, often used to support a business process.
2. A mediation module, which contains one or more mediation flow components. It is not necessary for a mediation module to contain any Java components, but there is no limit to the number of Java components that it can contain. Java

components augment the mediation flow component. The mediation module acts as a gateway to existing external services, which is common in ESB architectures.

A business module is deployed to an IBM BPM server. A mediation module is deployed to either an IBM BPM server or a WebSphere ESB server. Both types of modules support imports and exports.

Mediation Modules

Mediation modules are SCA modules that can change the format, content, or target of service requests. Mediation modules operate on messages that are in-flight between service requesters and service providers. Mediation modules can route messages to different service providers and can also transform the message content or structure. Mediation modules can provide functions such as message logging, and error processing that are tailored to your requirements. You can change certain properties of mediation modules, from the WebSphere ESB administrative console, without having to redeploy the module. Figure 11 shows a simplified example of a mediation module. The mediation module contains one mediation flow component.

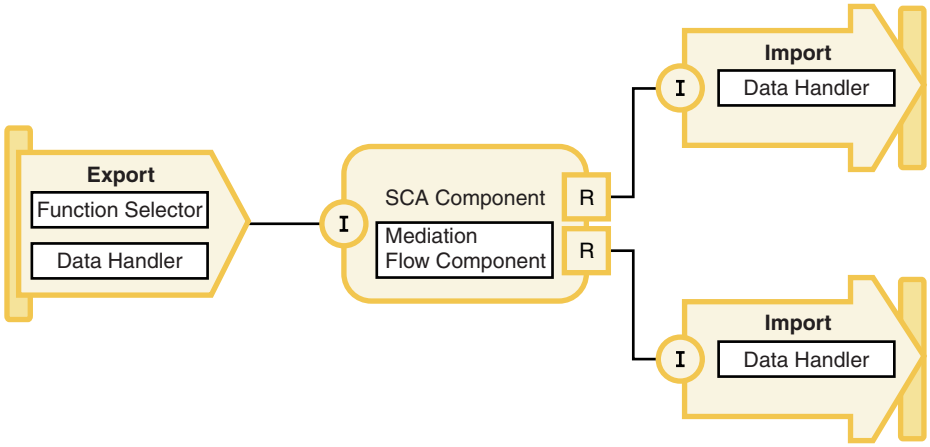


Figure 11. Mediation module

Module imports and exports

The mechanisms used for invocations from module to module and module to external services, are called imports and exports. Imports and exports are represented from the point of view of the module. The module is a self-contained bundle of components that perform a specific business function. A module invokes this capability using an import if it is required to use the business function from another service to call a function (external service or other module).

Exports provide the ability to make a service available over a number of different transport protocols. The export is associated to a particular component within the module.

Imports and exports also have the ability to interact with service providers across a variety of protocols and Application Programming Interfaces (APIs), such as:

- Web services

- HTTP
- JMS
- MQ
- EJB
- WebSphere Adapters
- SCA

Data handlers

Data handlers are reusable transformation logic in WebSphere ESB, which can be used by exports and imports. Data handlers transform native formats into business data and from business data into native formats required by external services.

In your SOA implementation, business data can flow between service providers and service requesters over a variety of protocols (HTTP, JMS, MQ, and so on), in a variety of data formats such as comma separated value, delimited, fixed width, COBOL and so on. Different protocols can have the same mechanisms for carrying the business data in their respective protocol envelope. For example, a HTTP message can encode its data using a comma delimited format, which can also be used in a JMS message. The data handler interface is protocol neutral, which enables data handlers to be usable across all of the protocols supported.

WebSphere ESB supplies a number of built in data handlers including:

- Atom feed format
- Delimited format
- Fixed width format
- JavaScript Object Notation (JSON) format
- SOAP data handler
- WTX Invoker Data Handler
- WTX MapSelection Data Handler
- XML Data Handler

Function selector

The function selector determines which operation defined on the associated interface is invoked called. When a message arrives at an export it will include raw transport header information and encoded data for the body of the message. The function selector can use header data, body data, or both, to determine which of the operations on the associated interface can be invoked. As this mechanism is likely to be dependent on the protocol headers, the implementation of the function selector will be protocol specific. For example, a JMS message header can specify the operation name and a function selector can use this JMS message header to determine the operation that is called.

SCA components

SCA components are building blocks for mediation modules. There are two types of SCA components, which are allowed within a mediation module:

1. Mediation flow component
2. Java component

Mediation flow component

Mediation modules contain a specific type of SCA component called a mediation flow component. A mediation flow component manipulates the message as it flows through the mediation module. A mediation flow component can be associated with several interfaces. A mediation flow is the process that occurs for a particular operation on the interface. There is at least one mediation flow for each operation on the interface.

An interface can have two types of operation:

1. One way (request only), with a single request flow that holds the logic for the request message.
2. Two way (request/response) with a request flow and a response flow. The request flow holds the logic for the request message. The response flow holds the logic for the response message.

A mediation flow can contain any number of mediation primitives. Mediation primitives are reusable blocks of function that are wired together to build a mediation flow. These mediation primitives can route and transform the message. WebSphere ESB supplies a set of mediation primitives that provide functionality for message processing. You can also contribute your own primitives. If the mediation module will only transform from one protocol to another, a mediation flow component is not required. For example, a service request might be received using SOAP/JMS but might need transforming to SOAP/HTTP before sending on. In this instance, a Web Service Export (SOAP/JMS) and Web Service Import (SOAP/HTTP) is required.

Java component

The Java component provides an alternative implementation option for creating SCA components within WebSphere ESB. You can use Java code, instead of the mediation flow component to manipulate the messages as they pass through the system. This can provide additional flexibility in the solutions you implement.

Library

A library is a project that is used to store resources and artifacts, so that the resources or artifacts can be shared by multiple modules defined in the library. The usage of libraries is good practice where interfaces are shared across multiple mediation modules.

Unlike modules or mediation modules, library projects can share resources. The resources or artifacts that are defined in the library are:

- Business objects
- Interfaces
- Mediation subflow
- Datamap
- Relationship
- Binding resource

There are two kinds of sharing:

1. Sharing of artifacts for development
2. Sharing of artifacts within the solution at run time

For a module or mediation module to use the resources from a library at run time, you must add a dependency, indicating that the module is dependent on the library. You can add a library to the module and select to deploy it with the module. You can also add library dependencies to a library; for example, if one library uses resources in another library, then you must add the library dependency.

Artifacts in a library are public, and they can be shared across modules. Artifacts in a module are private. WebSphere ESB integration developers should organize artifacts by taking into consideration logical function and visibility. For example, related data types that are used by various pieces of a system should be placed into a library. Other modules can then easily reuse this library. Only one copy of each element needs to be created and maintained using this method. Likewise, related business logic features of the application should be grouped into modules. Structuring an application in this way will yield a more flexible, easily maintained application. It will also be easier to add new features to the application.

Business objects and interfaces

Business objects are containers for application data, such as customer details or an invoice. Data is exchanged between components by way of business objects. When using imports and exports in assembly diagrams, it is good practice to put the business objects and interfaces that are used by the import and exports into a library so that they can be shared. Then, add a dependency on the library to all of the modules that use these common resources. Avoid copying the same business objects and interfaces into different modules to use them.

Mediation subflows

If you have mediation logic that you want to reuse across mediation flows you can define a mediation subflow within a library. Mediation flows cannot be placed in a library.

Library resources that are shared

If you choose to deploy a library with a module, which is the default setting, a copy of the library is made for each module when the module is deployed. After deployment, if shared resources change in the library, all modules using the resources have to be updated. For example, two modules share some resources in a library. The applications are deployed. One of the modules has to be updated, resulting in changes to some of the shared resources in the library. In this case, the second module also needs to be updated to reflect the changes in the shared resources.

Chapter 4. Mediation Flow Component

Mediation flows operate on messages that are in-flight between the service requester and service provider. Mediation primitives are the building blocks of mediation flows. You create mediation flows in mediation flow components. Mediation flow components can exist in either modules or mediation modules.

Mediation flows

A mediation exposes an interface to the service requester. The mediation might use another interface to call the service provider. The interface used by the service requester is referred to as the source interface, and the interface provided by the service provider is referred to as the target interface. Depending on the service requester and the service provider, the source and target interfaces might be different.

For each source operation, the mediation can contain a request flow. For each operation that has a response, the mediation has a response flow.

A request flow processes a request message from a service requester. The flow begins with a single input node for the source operation, followed by one or more mediation primitives wired together. Each target operation has a callout node, and the request flow can be wired to the callout node to call a particular operation. If a message is to be returned to the source directly after processing, the request flow can be wired to an input response node in the request flow. If fault messages are defined in the source operation, an input fault node is also created. Wiring the request flow to an input fault node allows a fault message to be returned directly to the service requester.

A response flow processes responses returned from the service provider. The flow begins with a callout response node for each target operation, followed by one or more mediation primitives wired together. The response flow contains a single input response node representing the source operation. Wiring the response flow to the input response node causes a response to be sent to the service requester. If fault messages are defined in the target operation, a callout fault node is also created in the response flow. The callout fault node allows fault messages that are returned by the target operation, to be processed. Errors that are returned by the operation but are not defined as fault messages, are propagated to the fail terminal of the callout response node.

A mediation flow component has an error flow for each source operation. The error flow acts as a catch-all for messages that are propagated from any unwired fail terminal on any mediation primitive or node, in either a request or a response flow.

Figure 12 on page 18 shows the three two flows within a mediation flow component.

© IBM Corporation 2005, 2011 © IBM 2007, 2011

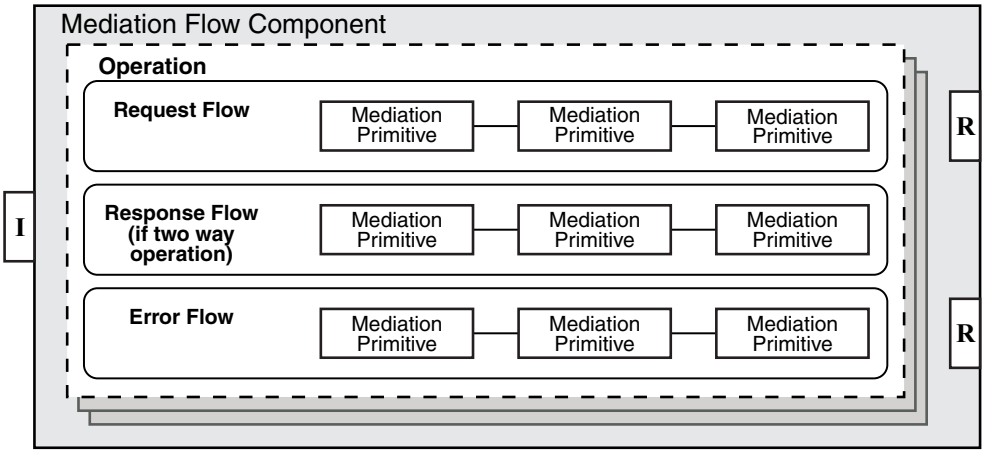


Figure 12. Mediation flows

Mediation primitives

Each mediation primitive allows you to manipulate messages. For example, you can route a message or change its content. Mediation primitives process messages as Service Message Objects (SMOs).

You can use mediation primitives to change the format, content, or target of service request messages.

All mediation primitives have an input terminal (called in) that can be wired to accept a message. Most mediation primitives have one fail terminal (called fail) and one or more output terminals. However, the Stop and Fail mediation primitives have no fail terminal and no output terminals.

If a failure occurs during the processing of the input message, the fail terminal propagates the original message, together with any failure information. If the fail terminal is not wired, the error flow runs. If the error flow has not been implemented, the running of the flow stops and a failure is reported to the system log.

If an output terminal of a mediation primitive is left unwired at run time, the unwired output terminal stops this path in the flow without generating a failure, and the message is consumed.

The behavior of a mediation primitive is controlled by properties. Each mediation primitive type has its own properties that configure the mediation primitives' capability.

Administrative control of mediations

Mediation primitives and nodes have properties that can be used to customize their behavior. Some of these properties can be made visible to the WebSphere ESB administrator by promoting them. Certain properties lend themselves to being administratively configured. Integration Designer lists the properties that you can choose to promote under the promoted properties of a mediation primitive.

Promoted properties are given an alias name, and you can set the alias name so that it is meaningful in the context of a particular mediation. The alias name is the

property name that is displayed on the WebSphere ESB administrative console; multiple promoted properties can be given the same alias name if they are of the same type. Therefore, what appears as a single property, in the administrative console, can set the same value in multiple mediation primitives. You can set the value of a promoted property, from Integration Designer, during installation of the module and once the application has been installed using the WebSphere ESB administrative console.

Service Message Objects

The Service Message Object (SMO) provides an abstraction layer for processing and manipulating messages exchanged between services.

Data passing through mediation flows is represented by the SMO. The SMO is considered to be an envelope containing the following groups of data:

- Context information (data other than the message payload and headers).
- Header information associated with the message. For example, Java Message Service (JMS) headers if a message has been conveyed using the JMS API, or MQ headers if the messages has come from WebSphere MQ.
- The body of the message: the message payload, is the application data exchanged between service endpoints.
- Message attachments

The SMO is a DataObject and its content can be accessed via the Service Data Object API (SDO API). SDO is a standard for representing structured business data. The body of the SMO contains a business object representing the application payload.

SMO structure

All SMOs have the same basic structure. The structure consists of a data object called a ServiceMessageObject, which contains other data objects representing the header, body, attachments, and context data.

The structure of a SMO is shown in Figure 13 on page 20.

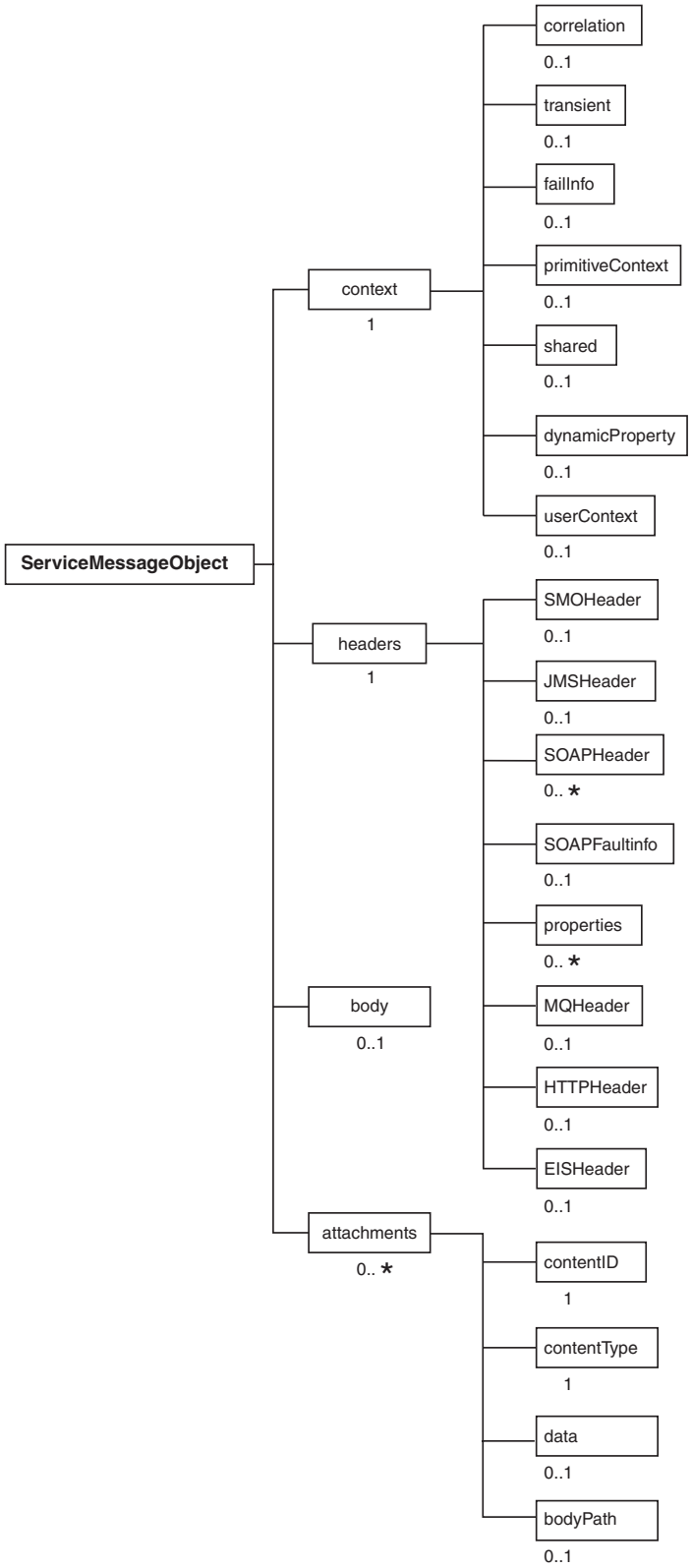


Figure 13. Overview of SMO structure. The context, headers, body and attachments of a ServiceMessageObject

The ServiceMessageObject Java interface provides access to the message headers, message body, message context and message attachments.

SMO headers

The SMO headers carry header information for different types of messages.

The SMO headers contain information that originates from a specific import or export binding (a binding specifies the message format and protocol details). Messages can come from a number of sources, so the SMO has to be able carry different kinds of message header. The kinds of message headers include:

- SOAP
- HTTP
- JMS
- WebSphere MQ
- SCA
- Web services
- WebSphere Adapters

SMO context

The SMO context lets mediation primitives pass data that is not part of the message payload, between themselves.

SMO context objects are either user-defined or system-defined.

System-defined context objects

The following context objects are system-defined:

- failInfo. The failInfo context object is used to hold exception information
- primitiveContext. The primitiveContext object contains information used by specific mediation primitives
- dynamicProperty. The dynamicProperty object is used to override promoted properties

User-defined context objects

User-defined context objects can be used to store properties that mediation primitives use later in the flow.

The following context objects are user-defined:

- correlation context is used for mediation primitives to pass values from the request flow to the response flow.
- transient context is used for passing values between mediation primitives in the current flow. Values in the transient context are not shared between the request and the response flows or between branches of the same flow.
- shared context is a storage area you can use if you want to aggregate data. The shared context is shared between all branches of a flow. It is not shared between the request and the response flows or between branches of the same flow.
- user context is slightly different to other user-defined context objects. It is used to pass data that is not part of the message payload, between multiple SCA components in the same module or across SCA module boundaries.

SMO body

The body of a SMO is a business object that is defined by reference to a Web Services Description Language (WSDL) message.

For each part defined by the WSDL message there is one element under the SMO body. The contents of the SMO elements are of the structure of the WSDL part definition. The element names depend upon the kind of WSDL message that the SMO is defined from.

SMO attachments

The SMO contains an `attachments` element for each attachment associated with a SOAP message.

The SMO `attachments` element let you send and receive SOAP messages that have attachments of various types.

You might want to send SOAP messages with attachments and let the attachments pass through the mediation flow unchanged, or you might want to create new attachments, perhaps from information in the message or from an external source.

Data representation of SMO

A SMO has an SDO and XML nature and behaves like a business object. This means that you can always obtain XML representations of the SMO. A SMO can also be treated as a business object.

You can use the SMO with business object utilities such as BOCopy and BOXMLSerializer. In addition if you need to examine the SMO, you can access and manipulate it using the BO API (SDO API). This provides extensive capabilities including the ability to get and set individual fields. The SMO also has its own Java interface which is a convenience method for accessing the top level structure of the SMO.

Many mediation primitives have properties that contain XPath expressions. You can use this XPath expression to specify which subset of the message for the mediation primitive, should be processed. Depending on the mediation primitive, you can specify: `/`, `/body`, `/headers`, `/context`, `/attachment`, or your own XPath expression. `/` refers to the complete SMO, `/body` refers to the body section of the SMO, `/headers` refers to the headers of the SMO. If you specify your own XPath expression, the part of the SMO you specify is processed.

Integration Designer shows the structure of a message and allows you to select locations within the message. In this way you can navigate the structure of a message and create XPath expressions.

Business objects provide a platform wide data representation. The structure of a business object is defined in an XML schema which is either imported or created in Integration Designer using the business object editor. Business objects are used to represent the data that is passed through an SCA module.

Chapter 5. Routing Messages Within a Mediation Flow

Within WebSphere ESB, there are several mediation primitives that can be used for routing messages in a mediation flow.

Most mediation primitives have one input terminal (in), one fail terminal (fail), and one or more output (out) terminals, as shown in Figure 14. The in terminal is wired to accept a message and the other terminals are wired to distribute messages. If a failure occurs, the fail terminal distributes the original message, together with any failure information.

Figure 14. Mediation primitive

Figure 14 shows that a mediation primitive with two out terminals and one fail terminal. A mediation primitive can be wired within mediation flows to specify the potential paths to which a message can be distributed. The actual path for a message is dependent on the mediation primitive. There are different paths a message can take within a mediation primitive:

- Messages might leave an out terminal one or more times.
- Messages might leave the fail terminal once.
- Messages might not leave any of the terminals.

Flow branching

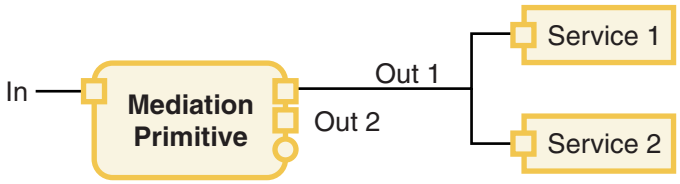


Figure 15. Branched mediation primitive

Figure 15 shows how a mediation primitive can have multiple wires from one output terminal branching to several services. You cannot specify the order in which messages will leave the terminal. If you need to send the message through a specific wire first, then you must use multiple output terminals each with a single wire. You can also select a mediation primitive that will allow you to control the order in which output terminals are invoked (for example see Figure 18 on page 26).

© Copyright IBM Corp. 2007, 2011

Example scenarios

This section describes example scenarios that a fictitious company want to implement. It then goes on to discuss the different mediation primitive options that can be used in the implementation.

The scenario addresses the business need of a financial services company that provides an interactive web-based stock market service to its customers.

The company wants to differentiate itself from its competition by offering tiered levels of service. The company's goal is to offer delayed stock quotes to their standard customers and real-time quotes to their gold customers, that is, customers who pay a fee.

The service requester provides a query containing a stock symbol and customer ID to the mediation primitive, which processes the query. The customer's subscription level is determined, and depending on the level of subscription, the query is routed to the appropriate service provider. The quote that is returned from the service provider is then returned to the client application. The following areas look at the ways in which the query can be processed by different mediation primitives.

Message filter mediation primitive

To facilitate the basic request, the company can use the Message Filter mediation primitive within their mediation flow component to selectively route messages that meet a certain criteria, through different paths of a flow, using XPath expressions.

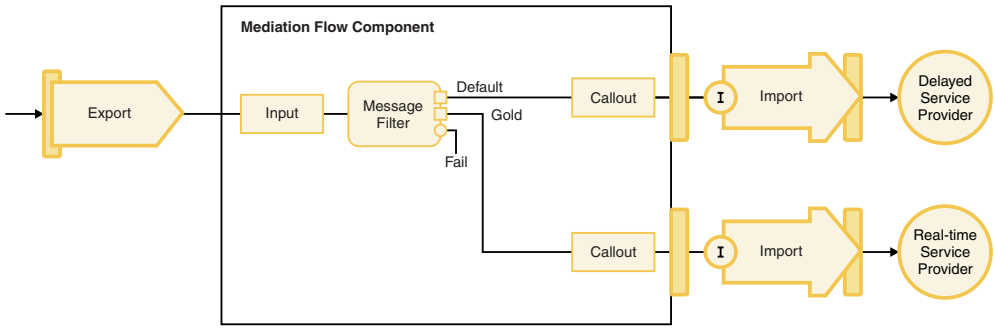


Figure 16. Message filter mediation primitive

Figure 16 shows how the gold output, is associated with the real-time service provider, to offer real-time stock quotes to all gold customers. The default output wire is associated with the delayed service provider, to give delayed stock quotes to all standard customers. The contents of the input message are compared with each XPath expression in turn, and depending on which condition is met, a message can be sent from the associated output terminal.

```
<body>
 <getStockQuote>
  <companyID>IBM</companyID>
  <customerType>GOLD</customerType>
  <person xsi:type="Customer">
   <customerID>013652</customerID>
   <customerName>Joe Bloggs</customerName>
```

```
    <customerAddress>123 One Street</customerAddress>
   </person>
  </getStockQuote>
</body>
```

The previous input message is received by the Message Filter mediation primitive. The following XPath expression is used to signal that the message is sent from the gold output wire:

```
/body/getStockQuote/customerType=='GOLD'
```

If this condition is not met, a message is sent from the default terminal.

Type Filter mediation primitive

The company want to extend the functionality of their mediation flow component, by giving employees the same real-time quotes that the gold customers receive.

To facilitate this request, the company use the Type Filter mediation primitive within their mediation flow component. The primitive uses XPath expressions that can route messages down different paths of a flow, based on their business object type. The Type Filter mediation primitive always sends a message from the first matching output terminal.

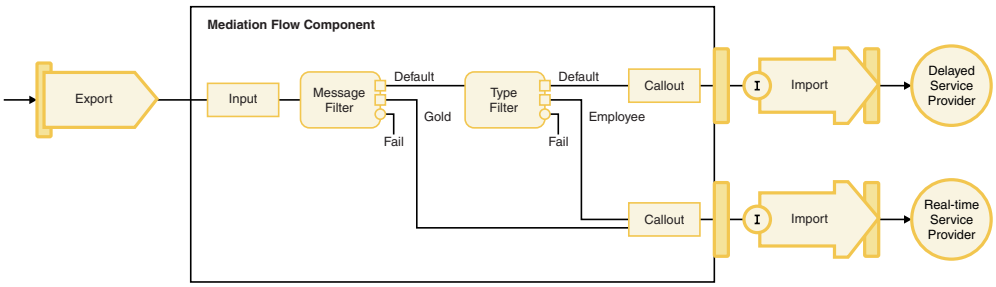


Figure 17. Type Filter mediation primitive

Figure 17 shows that the message that is sent from the Message Filter default output wire to the Type Filter mediation primitive. If the business object of type employee is found; the message is sent along the employee output wire to the real-time service provider. For any other business object the message is sent along the default output wire to the delayed service provider.

```
<body>
 <getStockQuote>
  <companyID>IBM</companyID>
  <customerType>STANDARD</customerType>
  <person xsi:type="Employee">
   <employeeID>013690</employeeID>
   <employeeName>Jack Bloggs</employeeName>
   <employeeAddress>123 Two Street</employeeAddress>
  </person>
 </getStockQuote>
</body>
```

The previous input message is received by the Type Filter mediation primitive. The following XPath expression and business object type are used to signal that the message is sent from the employee output wire:

```
XPath: /body/getStockQuote/person
Business object type: Employee
```

Flow Order mediation primitive

The company must log all requests that go to the real-time stock service provider.

To facilitate this request the company use the Flow Order mediation primitive. With this primitive, the company can order the output of the messages.

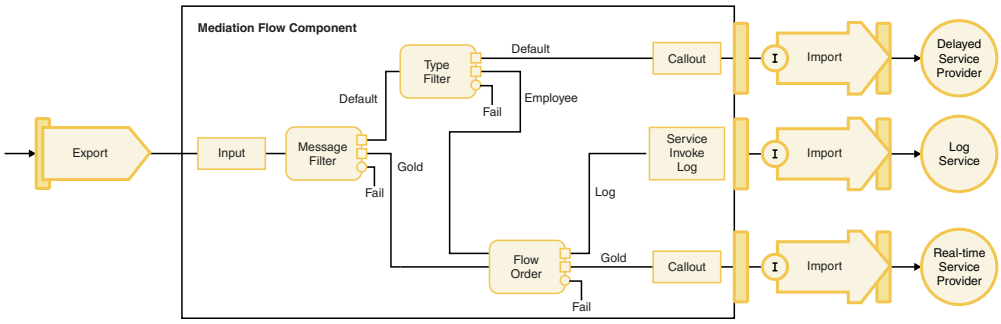


Figure 18. Flow Order mediation primitive

The log must be carried out before the messages are sent to the real-time service provider through the callout. The log service is a one-way operation. Figure 18 shows that the message that is output from both the Message Filter and the Type Filter mediation primitives is sent to the Flow Order mediation primitive. The output that is sent includes information of all gold member customers and employees that have requested real-time stock quotes. This mediation primitive will send a message to the Service Invoke Log, before sending a message to the callout.

Custom mediation primitive

The company have introduced a silver level of subscription. Silver members will receive real-time stock quotes between the hours of 9:00am and 12:00pm. Outside these hours, silver members receive delayed stock quotes.

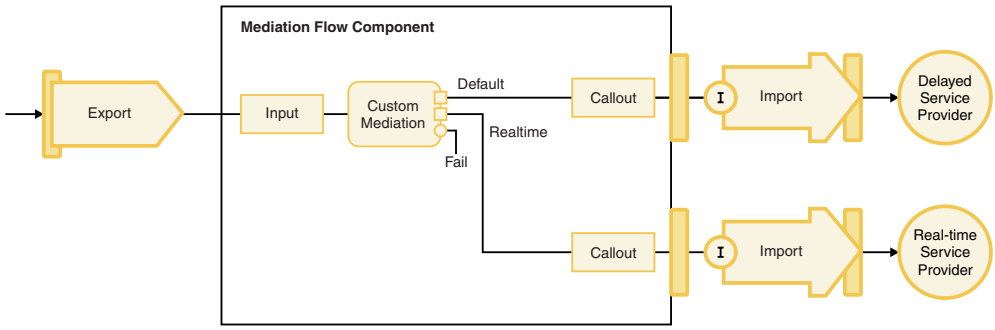


Figure 19. Custom mediation primitive

The enhanced logic to provide time dependent selection of the service is not possible with the Message Filter or Type Filter mediation primitives. In this case, a

Custom mediation primitive is used to provide more flexibility. Using a Custom mediation, the entire message can be accessed using a Java API. Logical expressions can be created using Java. The following Java code shows how the Custom mediation is created:

```
 String customerType = smo.getString("/body/getStockQuote/customerType");
if(customerType.equals("GOLD") || (customerType.equals("SILVER") &&
Calendar.getInstance().get(Calendar.HOUR_OF_DAY)<12 &&
Calendar.getInstance().get(Calendar.HOUR_OF_DAY)>8))
{
 // Fire realtime service output service
 realTimeTerminal.fire(smo);
}
else
{
 delayedTerminal.fire(smo);
}
```

Chapter 6. Data Transformation

Within a mediation module, the message is received from a service requester before it is sent to a service provider. The structure of the message from a service requester is not always the same as that of a service provider. One of the common uses of a mediation is to transform data from one format to another.

Business Object Map and XSL Transformation mediation primitives

There can be structural differences in the types of data within the source and target XML message body type.

An example is shown in Figure 20 on page 30. The only way that the two applications can communicate, is by using a transformation. Data transformation takes place within the mediation flow component. You can transform the entire message or a particular subsection of a message. When transforming the structure of a message, the two most useful mediation primitives are:

1. XSL Transformation (XSLT) mediation primitive
2. Business Object Mapper (BO Mapper) mediation primitive

© Copyright IBM Corp. 2007, 2011

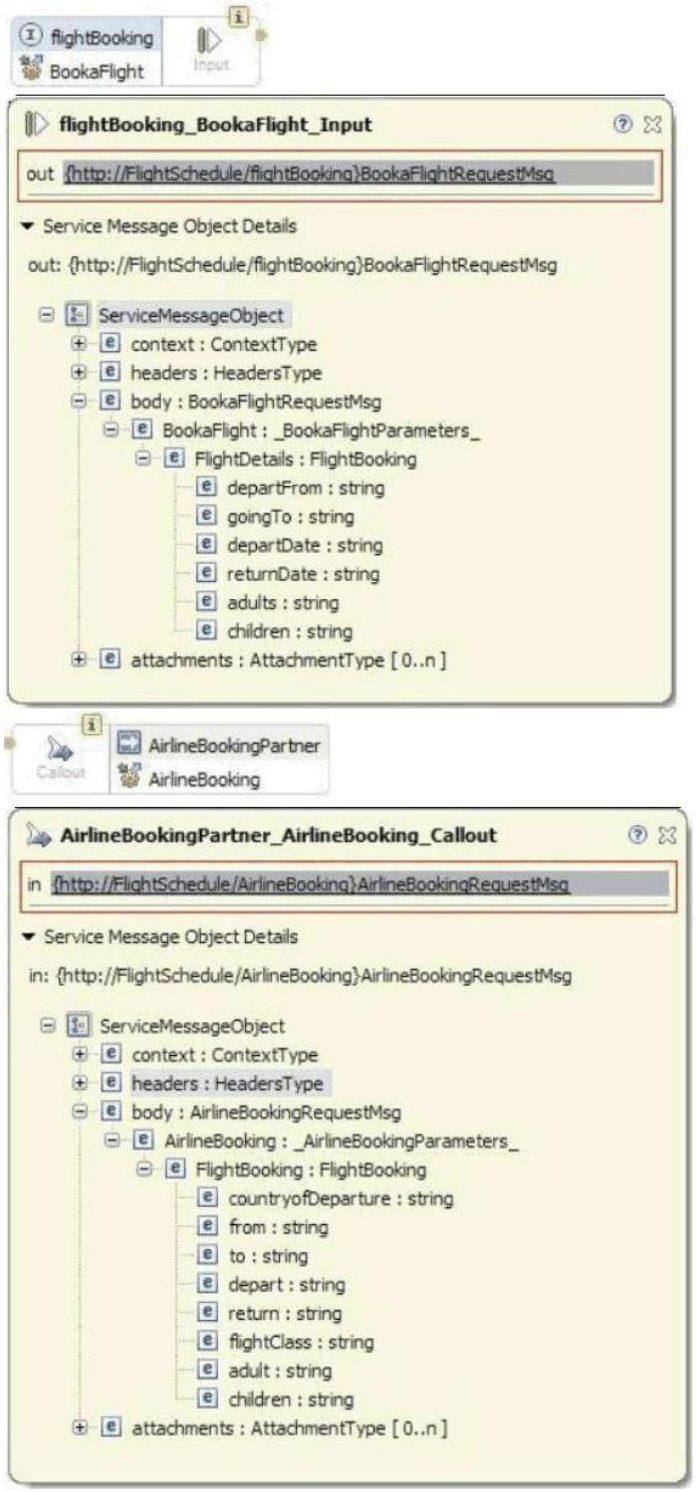


Figure 20. Hover pane showing the source and target XML message type

Both the XSLT and the BO Mapper mediation primitives transform the SMO. The difference is that the XSLT mediation primitive changes message body content according to an XSL style sheet. The BO Mapper mediation primitive changes the

message body content, according to a BO Map which is created in the Integration Designer. Both of these mediation primitives allow services that use different business objects to communicate and exchange data.

Within the Integration Designer tooling, the Mapping Editor makes populating a target XML document with data from a source XML document a visual task. The Mapping Editor is used by both the XSLT and BO Mapper mediation primitives. The source message structure for the interface and the target message structure for the interface are within the Mapping Editor. By drawing connections between the source and target fields, you create mappings between the source and target elements and attributes.

Within a transformation of the entire SMO, you can store data outside of the message body, for retrieval by downstream primitives. For example, the context of the SMO can be used to store or retrieve data from either the transient context if it is required in the same flow (request or response) or the correlation context if it is needed in both request and response flows.

Comparison between the XSLT and BO Mapper mediation primitives

There is some overlap between the capabilities of the BO Mapper and the XSLT mediation primitives. The following table summarizes the different capabilities of each.

Use XSLT when	Use BO Maps when
You have an existing stylesheet or want to use custom XSLT in the map.	You have an existing BO Map or submap.
You want to use XPath expressions inside the map.	You want to use the relationship service (particularly dynamic relationships).
You want to use Java snippets that use the DOM API to access or update data.	You want to use Java snippets that use the BO API (SDO API) to access or update data.
You want to use built-in XPath or EXSLT functions to access or modify data.	You want to order the completion sequence for the transforms.
You want to combine contents from more than one array (repeating element).	BO Maps can be faster when used with WebSphere Adapters, because the BO Maps work directly with the SDO API.

When performance is important and the advanced functions offered by XSLT are not required, the BOMapper mediation primitive should always be used.

Example scenarios

In this scenario, a fictitious company, **Personal Holidays** have a holiday booking system.

In order for customers to book flights with Personal Holidays, the flight availability must be checked against a fictitious airline company, **My Airways**, flight schedule. The interface for the Personal Holidays booking system is generic and not airline specific. The Personal Holidays booking system represents information differently to the My Airways booking system. The differences are shown in the following XML.

The Personal Holidays flight booking system message that represents a flight search:

```
<flightBooking>
 <departFrom>London</departFrom>
 <goingTo>Paris</goingTo>
 <departDate>29/10/2010</departDate>
 <returnDate>12/11/2010</returnDate>
 <adults>2</adults>
 <children>0</children>
</flightBooking>
```

The My Airways flight booking system message that represents a flight search:

```
<flightBooking>
 <countryOfDeparture>UK</countryOfDeparture>
 <from>London</from>
 <to>Paris</to>
 <depart>29/10/2010</depart>
 <return>12/11/2010</return>
 <flightClass>2</flightClass>
 <adult>2</adult>
 <children>0</children>
</flightBooking>
```

Mapping message structures

To implement the basic request, Personal Holidays are required to map their message structure with the My Airways message structure. To map these interfaces, they must use a data transformation. The transformation takes place on the body of the message within a simple mediation flow component. The integration developer uses an XSLT mediation primitive to transform the body of the message. This is shown in Figure 21.

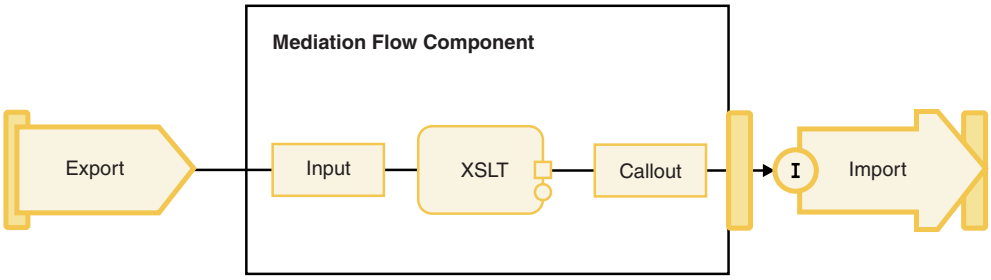


Figure 21. Message Structure mapping using an XSLT mediation primitive within a mediation flow component

When an XSLT mediation primitive is created, a mapping file associated with the primitive is created or selected that describes the transformation between the source and the target business objects. This is shown in Figure 22 on page 33. The image also shows the way the values in the source message are used to populate the target message.

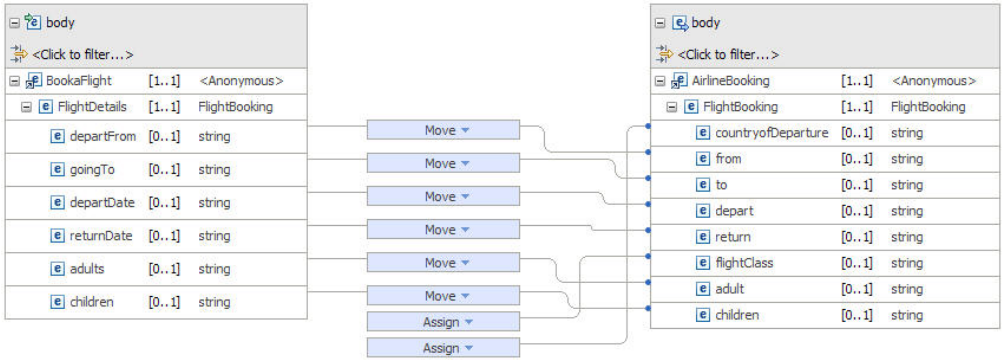


Figure 22. Message Structure mapping within Integration Designer

Service enrichment

Personal Holidays must determine the country of departure dynamically from within the mediation module.

The **countryOfDeparture** field does not exist within the Personal Holidays flight booking message structure. A Database Lookup mediation primitive is used to establish the country of departure in any given instance. The Database Lookup mediation primitive populates the country of departure into the transient context. In this example, the company use a BO Mapper mediation primitive to transform the transient context and body into the new required message structure. This is shown in Figure 23.

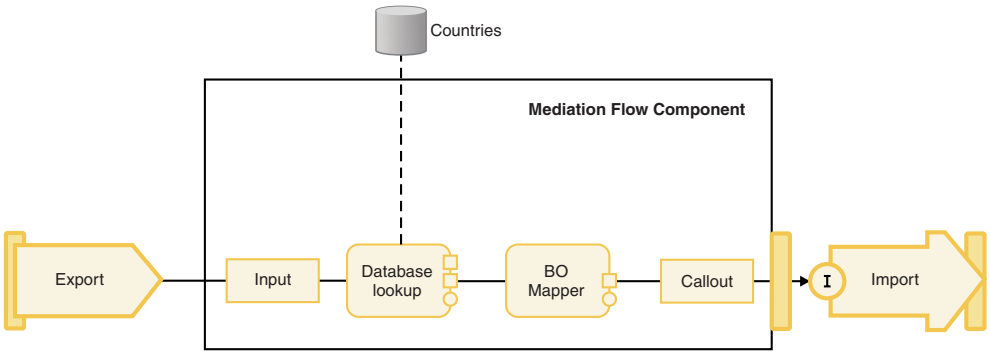


Figure 23. Service enrichment using a BO mapper within a mediation flow component

Data field level

In addition to changing the structure of a message, data transformation often requires changing the content of individual values. For example, it is common for date and time formats to differ between applications, or for string values to carry composed values that need to be extracted and separated. Within the mapping transformation primitives, you can operate on data values using a variety of built-in functions or calling out to custom code in a number of forms.

- The XML mapping editor provides built-in string, date/time, math and boolean functions, and allows you to call out to Java code that can use the DOM API in order to compose XPath expressions, or call out to other XSL templates. XML maps also provide the ability to look up values from external files or other sources, based upon a key taken from the message being transformed.

- The BO Map editor provides string extraction and link functions, and allows you to call Java that can use the BO (SDO) API. From a BO Map, you can also use the relationship transforms to manage sets of correlated data.

As an example, the Personal Holidays booking request includes a departure date of the form, yyyy-mm-dd (2010-11-10). My Airways expects the same date value to be represented in the form, dd-mm-yyyy (10-11-2010). In an XML map, the Format date function can apply a chosen pattern to a date field. This is shown in Figure 24 and Figure 25.

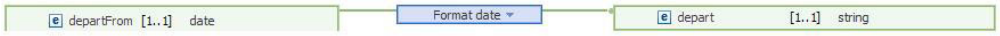


Figure 24. Format date function

In the Properties view, you should select the intended output format:

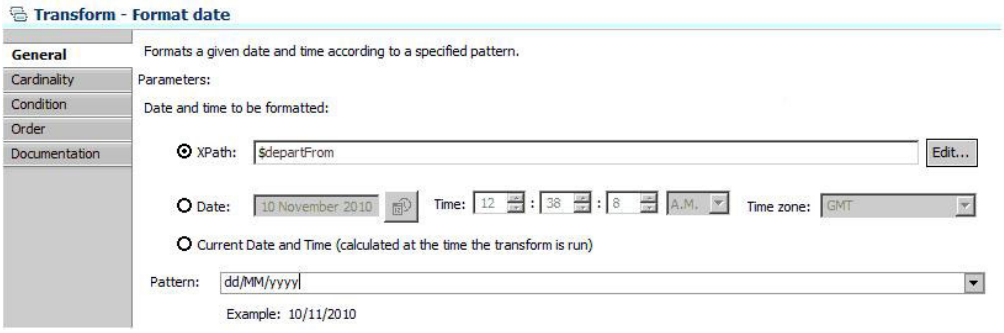


Figure 25. The configuration of the format date transformation

The value that is input to the function is the source field for the mapping. The transformed value is set into the output field. The transformation is performed in the XML domain, (inside the XSLT processor). The element value appears in XML. By contrast, operations that take place inside the BO Maps are performed in the SDO domain, that is using the Java business object representation. To perform an equivalent transformation in a BO Map, we would use a custom Java snippet, which would be passed to the data field as a **java.util.Date** object. This is shown in Figure 26 and Figure 27.

Figure 26. Custom transformation in a BO map

The Java code that implements the custom mapping is written in the Properties view:

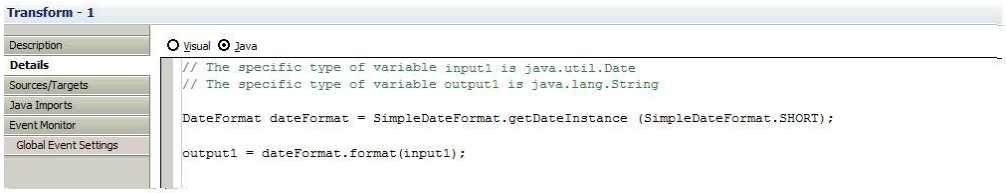


Figure 27. Java code in the Properties view

In Figure 27 on page 34 the **java.text.DateFormat** class is used to format the incoming **Date** object to produce a formatted string using the built in **SimpleDateFormat.SHORT** format. This corresponds to dd-mm-yyyy or yyyy-mm-dd. By using a Java snippet, you can access the source and target data fields as local variables. These are described in the generated comments produced by the editor. From a custom Java snippet, it is also possible to invoke code from other Java classes that are contained in the module or in an associated library project.

Custom mediation

There are cases where it is difficult to represent the necessary data transformation using a BO Mapper or XSLT mediation primitive. In these circumstances, you can use a Custom mediation.

This example describes an instance of when this could occur.

My Airways have created a new service that is used to calculate the number of regular meals and vegetarian meals required for a flight.

Customers are asked to select meals as a step in the flight booking process. This information will be requested for processing as a batch process to determine the number of regular and vegetarian meals required for a flight.

The following XML shows how the business object **FlightInfo** can contain a number of **SeatBooking** records. Each **SeatBooking** has a details field which is either a **SeatBookingDetails** object (normal customers) or a **FrequentFlyerDetails** object (frequent flyer customers). Each contains the customer's meal preference, but in different named fields, **mealType** and **mealPreference**. In this case, a Custom mediation is used to provide more flexibility to transform this message body.

```
...
<FlightInfo>
 <SeatBooking>
  <type>Economy</type>
  <details xsi:type="ns0:SeatBookingDetails">
   <mealType>regular</mealType>
   <otherDetails>...</otherDetails>
  </details>
 </SeatBooking>
 <SeatBooking>
  <type>Economy</type>
  <details xsi:type="ns0:FrequentFlyerDetails">
   <customerNumber>44906</customerNumber
   <seatPreference>Aisle</seatPreference>
   <mealPreference>regular</mealPreference>
   <milesTotal>490020</milesTotal>
   <dateJoined>2009-11-09-09+00:00</dateJoined>
   <lastUpdate>2010-02-11+00:00</lastUpdate>
  </details>
 </SeatBooking>
 <SeatBooking>
  <type>Business</type>
  <details xsi:type="ns0:SeatBookingDetails">
   <mealType>vegetarian</mealType>
   <otherDetails>...</otherDetails>
  </details>
```

```
 </SeatBooking>
 ...
</FlightInfo>
...
```

The Custom mediation code completes three steps:

1. It produces a new output SMO
2. It calls another Java class to calculate the total meals
3. It sets the total meals into the SMO body

Figure 28 shows the Custom mediation in Integration Designer. The Custom mediation needs to create an output SMO that has a body type **getMealTotalsResponseMsg** in the target namespace **http://Airline/GalleyFulfillment**.

The message type and structure can be inspected using the hover pane within Integration Designer. The hover pane appears when the mouse is moved over the primitive and the on the node is clicked once.

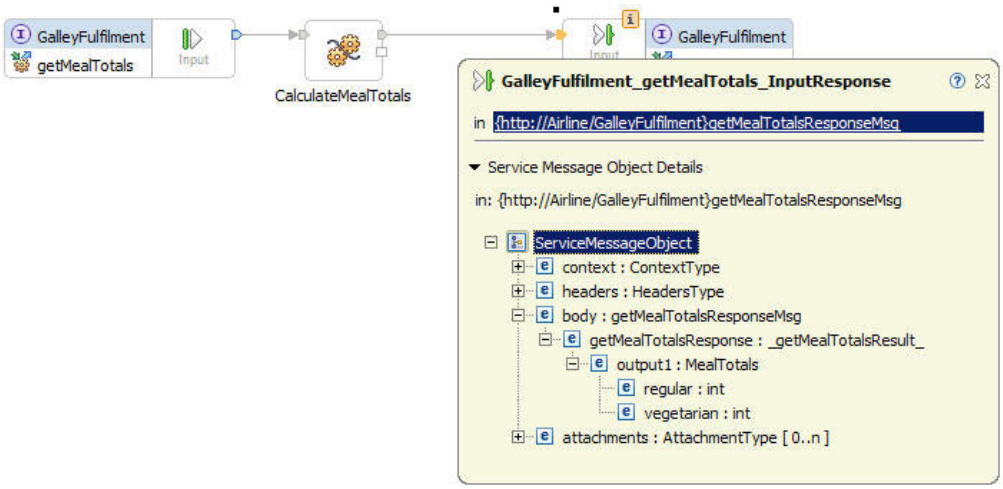


Figure 28. Hover pane showing the message type and structure

The following code shows how to create a new SMO body of a given message type. The BOFactory **createByMessage** is used:

```
BOFactory boFactory = (BOFactory) ServiceManager.INSTANCE.
 locateService ("com/ibm/websphere/bo/BOFactory");
DataObject newBody = boFactory.
 createByMessage ("http://Airline/GalleyFulfilment", "getMealTotalsResponseMsg");
```

The following Java code allows you to set a new body for the SMO.

```
smo.setBody (newBody);
out.fire (smo);
```

Though this code has created a SMO with the correctly typed body object, it has not been populated. The SMO body is a DataObject whose immediate child elements are defined in the set of parts defined in the WSDL message for the operation.

Figure 28 on page 36 shows the body structure of the SMO. The expanded view shows that the SMO body has a child element called **getMealTotalsResponse**. This has a child element **output1** of type **MealTotals**.

To complete the code for the Custom mediation, we call the utility method **getMealTotals** (which is provided by an external Java class, enabling re-use of the utility method **getMealTotals**, perhaps in a Library) and set the processed DataObject into the **output1** child property of the **getMealTotalsResponse** DataObject. The DataObject doesn't exist in the body, it is created before setting the final result into it as shown in the following code:

```
BOFactory boFactory = (BOFactory) ServiceManager.
 INSTANCE.locateService ("com/ibm/websphere/bo/BOFactory");
DataObject newBody = boFactory.
 createByMessage ("http://Airline/GalleyFulfilment", "getMealTotalsResponseMsg");

DataObject mealTotals = MealCalculator.getMealTotals (smo.getDataObject
 ("/body/getMealTotals/input1"));
DataObject totalsResponse = newBody.createDataObject ("getMealTotalsResponse");

totalsResponse.setDataObject ("output1", mealTotals);

smo.setBody(newBody);
out.fire (smo);
```

Figure 29 shows how the correct path inside the input of the SMO body can be found using the hover pane in Integration Designer.

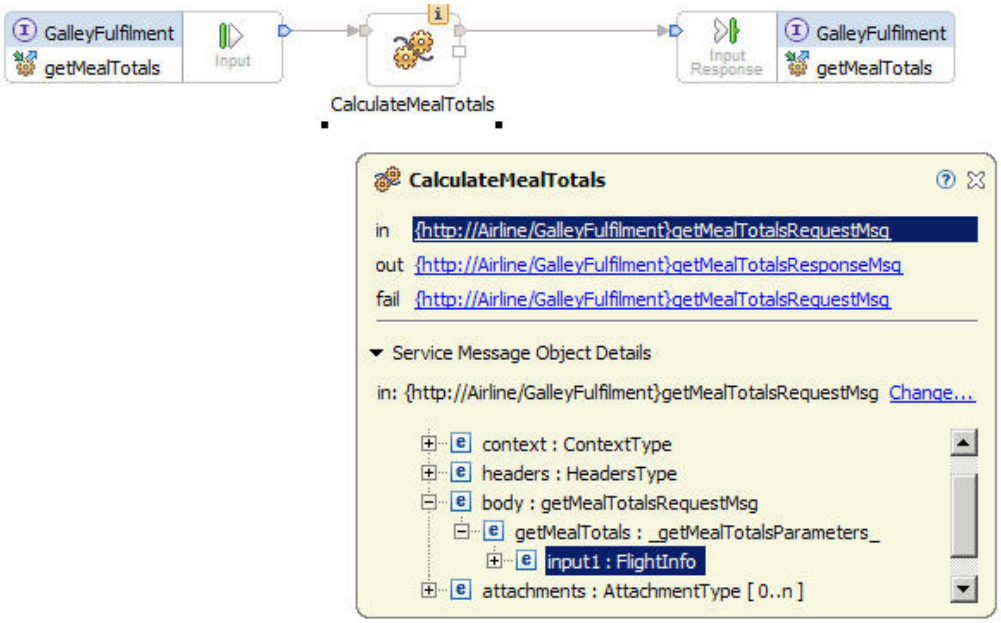


Figure 29. Hoverpane showing the body structure of the SMO

Note: As an alternative approach, Custom Java snippets can be used within XSLT and BO maps to complete functionality.

Message Element Setter and Data Handler mediation primitives

In addition to the BO Mapper, XSLT and Custom mediation primitives, there are other data transformation mediation primitives available within WebSphere ESB. These transformation primitives do not change the entire message structure of the SMO, they transform the content of a particular subsection of the message.

When transforming a subsection of a message, two mediation primitives that can be used are:

1. Message Element Setter mediation primitive
2. Data Handler mediation primitive

Both of these mediation primitives can change a subsection within the context, the headers, the body, or the attachments of the SMO. The Message Element Setter mediation primitive sets, deletes, or copies content into specified fields, or appends to an array of fields within a message. The Data Handler mediation primitive transforms native data to a business object, or vice-versa, so that it has a structure that can be accessed within the mediation flow component.

Message Element Setter mediation primitive

The Message Element Setter mediation primitive provides a simple mechanism for setting message content. It does not change the type of the message. You can set, copy, append or delete message elements using statements containing XPath expressions. You can specify multiple statements in one Message Element Setter mediation primitive by setting multiple targets, with the changes carried out in the sequence specified.

You can set target message elements to a constant value, or to a value copied from a location in the input SMO. If the target message element you specify does not exist then it will be created in the SMO. You can also delete message elements if they are optional or repeating elements.

Example scenario

The fictional company Personal Holidays are making an internal service available to external clients using a Web Service interface and a mediation flow, as shown in Figure 30.

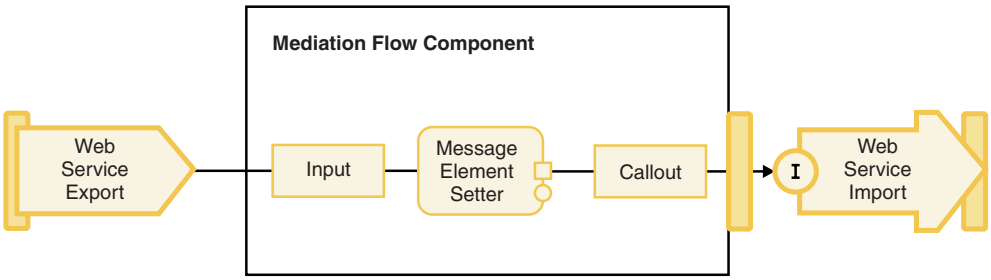


Figure 30. Message Element Setter in a mediation flow component

The external clients send in SOAP messages, through the web service export, that contain a SOAP header called **PersonalHolidaysClient**. The value of this header must be stored in the SMO correlation context so it can be used in the response flow but should not be passed to the internal service. Additionally the internal

service must have a SOAP Header named **PersonalHolidaysInternalHeader**, which must be set to a value of **EXTERNAL_CLIENT_SERVICE_CALL** for all incoming messages from the external clients.

We can use a Message Element Setter primitive to achieve this by:

1. Copying the value of the **PersonalHolidaysClient** SOAP header to the appropriate position in the correlation context.
2. Override the **PersonalHolidaysClient** to be **PersonalHolidaysInternalHeader** by:
   a. Setting the name of the **PersonalHolidaysClient** SOAP header to **PersonalHolidaysInternalHeader**.
   b. Setting the value of the **PersonalHolidaysInternalHeader** SOAP header to **EXTERNAL_CLIENT_SERVICE_CALL**.

Figure 31 shows the properties view within Integration Designer, where the actions of the Message Element Setter primitive are set. Each statement has the additional information needed to perform the action. All of the statements have a target XPath expression identifying the target of the action.

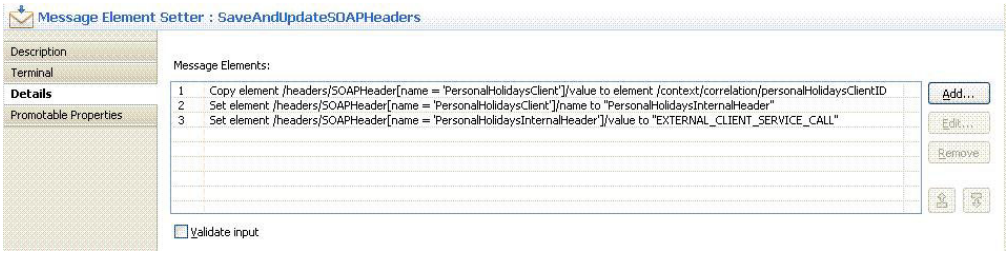


Figure 31. Message Element Setter properties within Integration Designer

Data Handler mediation primitive

The Data Handler mediation primitive transforms a targeted section of a message. The Data Handler primitive is used to convert an element of a message from a physical format to a logical structure, or from a logical structure to a physical format. The source and target of the transformation are indicated by XPath expressions. The Data Handler mediation primitive can be used when you are trying to integrate services, and parts of the message have not been parsed. By transforming the native data in a message into a business object, that data can be used by other mediation primitives. The part of the message that is not referenced by the XPath expression is left unmodified. Figure 32 on page 40 shows the data formats that can be transformed by a Data Handler mediation primitive. Additionally you can define and use your own data format transformation.

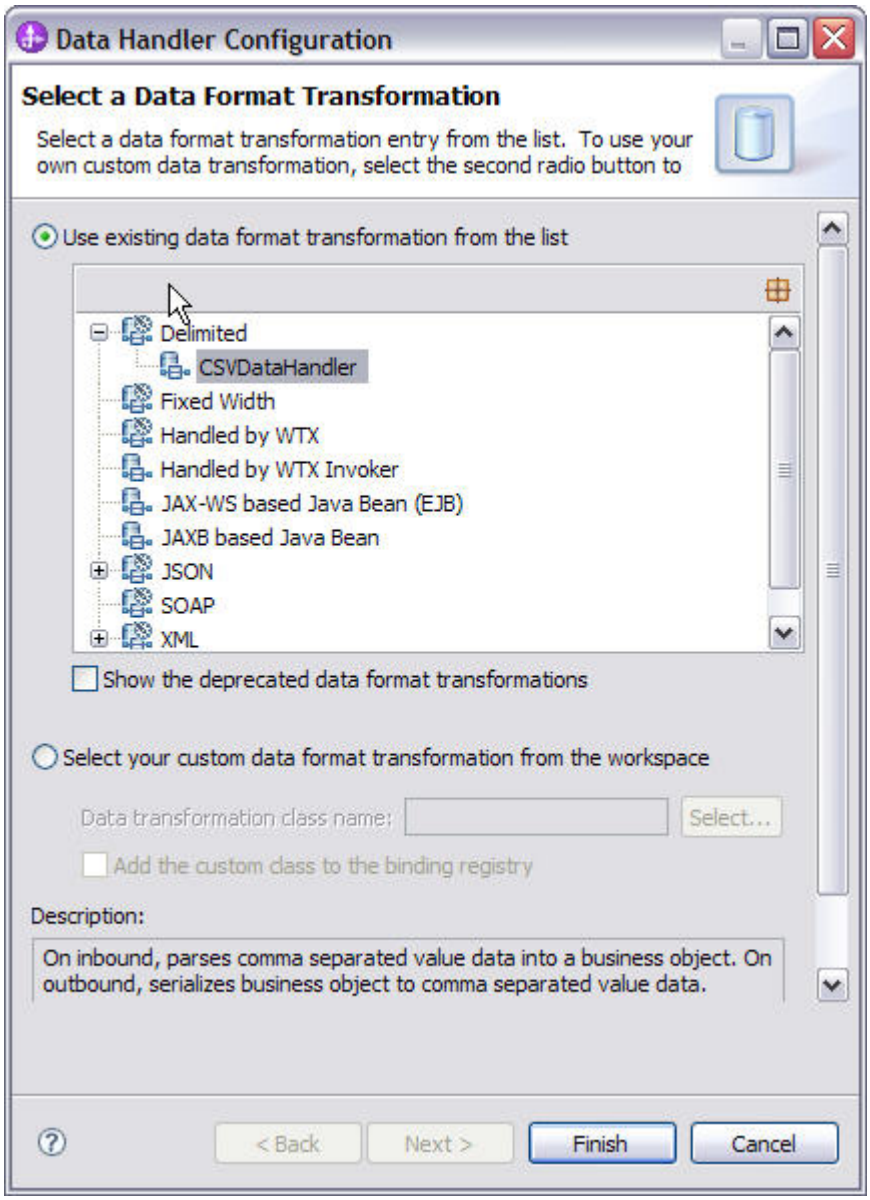


Figure 32. Data Handler Configuration

Example scenario

The fictional company My Airways outsource the supply of their airline meals to a fictional company called Flight Meals. My Airways specify that the ingredients of meals must be supplied to customers, should they request them. Flight Meals provide menuCourseIngredients information in a CSV format, as shown in the following XML:

```
<menuCourseDetails>
 <menuCourseID>DessertABC1234</menuCourseID>
 <menuCourseDescription>Sticky Date Pudding served with Caramel Sauce</menuCourseDescription>
 <menuCourseIngredients>Dates,Butter,Sugar,Bicarbonate of Soda,Eggs,Flour,
   Cream,Vanilla essence</menuCourseIngredients>
</menuCourseDetails>
```

My Airways process the menuCourseIngredients information into a business object format as shown in the following XML:

```
<menuCourseDetails>
 <menuCourseID>DessertABC1234</menuCourseID>
 <menuCourseDescription>Sticky Date Pudding served with Caramel Sauce</menuCourseDescription>
 <menuCourseIngredients>Dates,Butter,Sugar,Bicarbonate of Soda,Eggs,Flour,
   Cream,Vanilla essence</menuCourseIngredients>
 <ingredientsList>
  <ingredients>Dates</ingredients>
  <ingredients>Butter</ingredients>
  <ingredients>Sugar</ingredients>
  <ingredients>Bicarbonate of Soda</ingredients>
  <ingredients>Eggs</ingredients>
  <ingredients>Flour</ingredients>
  <ingredients>Cream</ingredients>
  <ingredients>Vanilla essence</ingredients>
 </ingredientsList>
</menuCourseDetails>
```

A Data Handler mediation primitive is used to transform data from a CSV format into the required business object. Figure 33 shows how a Service Invoke mediation primitive is used to call the Flight Meals Service Provider, to retrieve the menuCourseIngredients, containing the CSV formatted ingredients information from the Flight Meals Service Provider. The CSV formatted message is then passed to a Data Handler mediation primitive, which is able to transform the CSV data into a business object that can be used by the My Airways Service Provider.

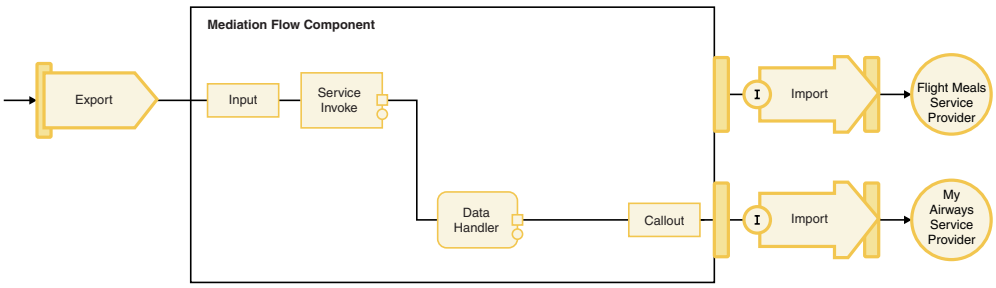


Figure 33. Data Handler within a mediation flow component

Within Integration Designer, the source and target XPath expressions are specified for the Data Handler mediation primitive, and the **CSVDataHandler**, data format transformation is chosen. This is shown in Figure 34.

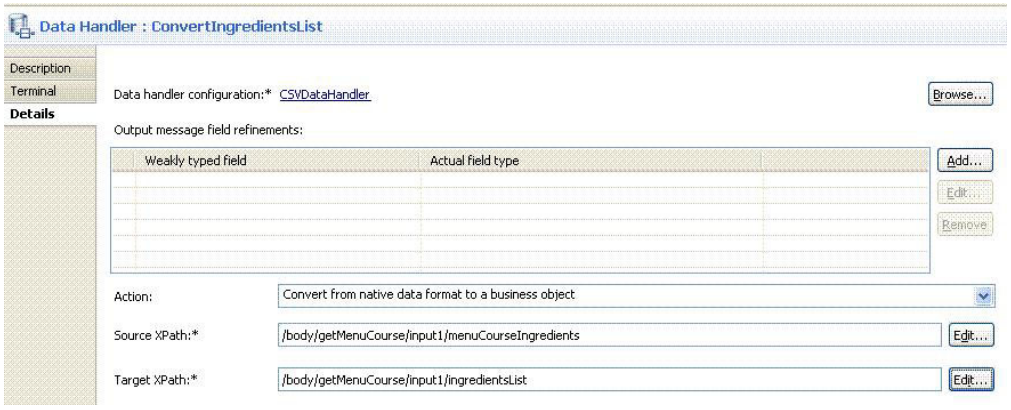


Figure 34. Data Handler in Integration Designer

WebSphere Transformation Extender

WebSphere Transformation Extender (WebSphere TX) is a separately licensed product and is not included as part of WebSphere ESB. Using WebSphere TX, you can create maps that can transform data between a wide variety of formats.

WebSphere TX for Integration Servers adds and enables the following components for use in your message flows:

- A mediation primitive for running WebSphere TX maps, called the WTX Map primitive
- A data format transformation for running WebSphere TX maps called the WTX Data Format Transformation

For more information on using WebSphere TX, see http://publib.boulder.ibm.com/infocenter/wtxdoc/v8r3m0/index.jsp.

There are some circumstances in which it is not possible to transform the message to your required structure using the BO Mapper, XSLT and Data Handler mediation primitives. In these cases WebSphere TX maps can be used to transform the data. WebSphere TX maps can be called using the WTX Map mediation primitive or by using the WTX Data Format Transformation in a Data Handler mediation primitive or in an Import or Export.

You can use the WebSphere TX maps and type trees in the following ways:

- Pulling data from sources such as files and queues within a map
- Calling external programs and services from a map
- Splitting and routing batches of messages

Use WebSphere TX type trees if you need message modeling features that are not available with native data format transformations, such as:

- Modelling complex binary or text message formats that Business Objects are unable to model
- Complex validation without having to write code, using component rules on type trees
- Use WebSphere Transformation Extender Industry Packs to integrate a range of industry standard data formats with your enterprise infrastructure through predefined type tree templates and maps that the packs provide.

Chapter 7. Invocation of Services

Services can be invoked with WebSphere ESB, so that you can create solutions that interact with other services. You can use the Callout node or the Service Invoke mediation primitive to call a service. Service invocations can be made by both request and response flows, using synchronous and asynchronous programming styles. This sections details the different ways that services invocations can be made.

Service Invoke and Callout nodes

You can call a service using either a Service Invoke mediation primitive or a Callout node.

Service Invoke mediation primitive

A Service Invoke mediation primitive is used to call a service from inside a mediation flow, rather than waiting until the end of the mediation flow and using the Callout node. The input message is used to call the service, and if the call is successful, the response is used to create the output message. If the call is unsuccessful, you can retry the same service or call another service.

You can have multiple Service Invoke mediation primitives inside one mediation flow. Therefore, you can have a series of service calls. You can use the Service Invoke mediation primitive in a request or response mediation flow.

Callout node

The Callout receives the message and calls the requested service and operation. There is a Callout node for each connected target operation in the mediation flow.

If the call is successful, the Callout Response node in the response flow receives the response message.

If the call is unsuccessful, the Callout can be set to retry service invocations depending on the type of fault received.

Comparison of the Service Invoke mediation primitive and the Callout node

You can call a service using either a Service Invoke mediation primitive or a Callout node. You must consider the similarities and differences between using either the Service Invoke mediation primitive and the Callout node.

Similarities between the Service Invoke mediation primitive and the Callout node:

- The in terminal of the Service Invoke mediation primitive corresponds to the in terminal of the Callout node.
- The out terminal of the Service Invoke mediation primitive corresponds to the out terminal of the Callout Response node.
- The fail terminal of the Service Invoke mediation primitive corresponds to the fail terminal of the Callout Response node (for an unmodeled fault).

© Copyright IBM Corp. 2007, 2011

- A modelled fault output terminal of the Service Invoke mediation primitive, corresponds to a modelled fault terminal of the Callout Fault node.

Differences between the Service Invoke mediation primitive and the Callout node:

- The Service Invoke mediation primitive does not switch from request flow to response flow.
- The Service Invoke mediation primitive does not modify either the transient context or the correlation context.
- A Service Invoke mediation primitive has a separate timeout terminal. A Callout node only has a fail terminal.

Although, there is a functional similarity between the Service Invoke mediation primitive and the Callout node, the Service Invoke mediation primitive does not replace the Callout node. Consider the following when deciding whether to use the Service Invoke mediation primitive or the Callout node:

Table 1. Considerations for using the Service Invoke mediation primitive or the Callout node

Use the Service Invoke if	Use the Callout if
You need to call a service and return the response within a mediation flow.	You need to mediate a message (without calling an intermediate service) and then call a service provider. The Callout provides the simplest model for this configuration.
You need to interact with multiple services, and produce an output that combines service responses.	
You need to call an intermediate service. For example, you might use an intermediate service to adjust a message, or validate a message externally. The mediation flow contains a Service Invoke mediation primitive, and a Callout node that is connected to the service provider.	

Retry

Callout nodes and Service Invoke mediation primitives provide retry capabilities for failed service invocations. A number of properties need to be set in order to enable and configure the retry.

- **Retry On**, determines whether, and how, fault responses cause a retry. The following values are valid:
  - Never - operation call is never retried, no matter the type of fault
  - Any fault - operation call is retried on both modelled and unmodeled faults
  - Modelled fault - operation call is retried only when a modelled fault is received
  - Unmodeled fault - operation call is retried only when an unmodeled fault is received

  Modelled faults are those that are explicitly listed in a WSDL file, any other fault is an unmodeled fault. This property is only applicable to request-response operations.
- **Retry Count**, determines how many times a service call should be retried before an output terminal is fired. The output terminal that is fired can be of the following types: modelled fault, timeout or fail. If `Try Alternate Endpoints` is false or if there are no alternate endpoints specified in the SMO headers, all attempts will retry the original service.

- **Retry Delay** is the number of seconds to wait between retry attempts. If left equal to 0, there will be no delay between retry attempts meaning they will run as fast as the server can process them.
- **Try Alternate Endpoints**. If the SMO Header contains alternate endpoints, retry calls will be made to them in the order they appear. `Use Dynamic Endpoint` must also be set for this function to be available.
- **Async Timeout** specifies the time to wait for responses when performing an asynchronous with deferred response invocation. If this timeout occurs and the `Retry Count` has not been reached, it is treated as an unmodeled fault and retry will be performed. The retry delay will take place following the Async Timeout. If the `Retry Count` has been reached, then the Timeout terminal of the Service Invoke primitive, or the fail terminal of the Callout Response node is fired.

Service invocation styles

Using WebSphere ESB, service invocations can be made by both request and response flows.

With SCA, you can call service components using synchronous and asynchronous programming styles.

A component can be called synchronously or asynchronously, independent of whether the implementation is synchronous or asynchronous. This is shown in Figure 35 on page 46. The component interfaces are defined in the synchronous form and asynchronous support is generated for them. You can specify the preferred interaction style of the component as synchronous or asynchronous. The asynchronous style is used if interface contains at least one operation that can take a significant amount of time to complete. As a consequence, the service requester must avoid keeping a transaction active while waiting for the operation to complete. The interaction style applies to all the operations in the interface.

The underlying SCA programming model provides two different invocation styles that can be used for request-response operations:

1. Synchronous SCA invocation
2. Asynchronous SCA invocation
   - Asynchronous with deferred response SCA invocation
   - Asynchronous with callback SCA invocation

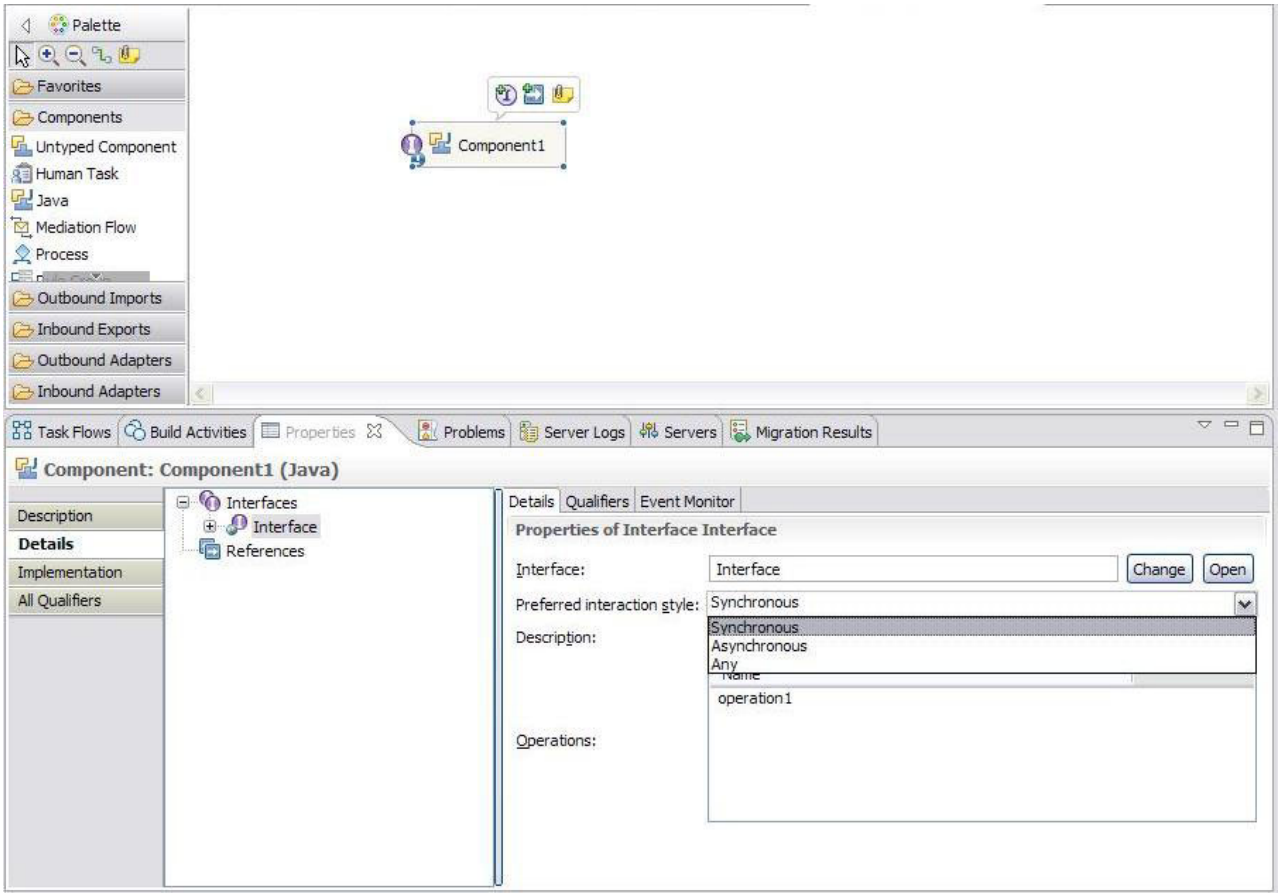


Figure 35. WebSphere ESB shows how a component can be called synchronously or asynchronously

Synchronous SCA invocation

When a service component is called synchronously, or performs a synchronous invocation, both the service requester and the service provider run in the same thread (a thread of execution can be defined as the smallest unit of processing that can be scheduled by an operating system). The calling component within WebSphere ESB is blocked until a response is received from the provider.

Once the service provider has been called, all processing is suspended until the thread returns a response. This is shown in Figure 36 on page 47.

Synchronous SCA invocation is useful if the service requester depends upon receiving a response from the service before continuing its processing.

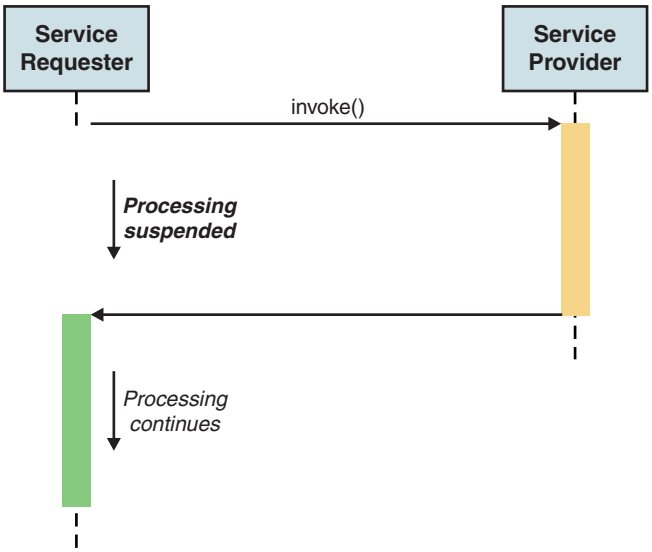


Figure 36. Synchronous SCA invocation

Example scenario

Jane wants to transfer money through online banking, to Stephen's bank account. The bank must ensure that Jane's account is debited, and Stephen's account is credited. Both of the actions must be completed, and if one of the actions fail then neither of them must occur. If Jane's bank account is debited, and Stephen's bank account fails to be credited, a rollback action must occur to credit Jane's bank account, so that money is not lost at any point of the transaction. A confirmation message will only be sent to Jane if the transfer is successful.

Synchronous SCA invocation is an applicable style in this situation, because a message is sent to the service in one thread of execution, and no processing is done by the service requester until the message returns for processing to continue in the same thread.

You can complete this scenario using the user interface. If you are using the SCA programming model directly, then you can use the following Java code, which shows how this scenario is made possible in the implementation:

```
public void transferFunds(String from, String to, float amount) {

  BOFactory bofactory = (BOFactory)ServiceManager.INSTANCE.
    locateService("com.ibm.websphere.bo.BOFactory");

  DataObject creditDetails = bofactory.
    create("http://ServiceInvocationStyles", "Credit");
  creditDetails.setString("creditor", to);
  creditDetails.setFloat("amount", amount);

  Boolean creditSuccess = (Boolean)locateService_AccountInterfacePartner().
    invoke("creditAccount", creditDetails);

  //If the above synchronous bank credit fails then the debit will not occur
  if (creditSuccess.booleanValue()) {

   DataObject debitDetails = bofactory.
     create("http://ServiceInvocationStyles", "Debit");
```

```
    debitDetails.setString("debitor", from);
    debitDetails.setFloat("amount", amount);

    Boolean debitSuccess = (Boolean)locateService_AccountInterfacePartner().
      invoke("debitAccount", debitDetails);

    //Only when the complete transaction is complete do we send a confirmation
    if (debitSuccess.booleanValue()) {
     locateService_CustomerInterfacePartner().
       invoke("sendConfirmation", "Your transfer was successful");
    }
   }

  }
```

Asynchronous invocation

When a service component is called asynchronously, the service requester and the service provider run in different threads. For an asynchronous invocation, the calling component within WebSphere ESB is not blocked while the request is processed by the provider. The SCA runtime returns control to the requester immediately, allowing it to continue processing.

With asynchronous invocation in SCA, there are three types of asynchronous interaction styles available: one way, deferred response, and request with callback. With all three types of asynchronous invocation, the service requester receives control back immediately from the SCA runtime.

There are two ways that the service requester can capture the response at a later time.

- **Asynchronous with deferred response**, where the service requester calls an operation on the service provider, and then continues processing. At a later time, the service requester calls the service provider for a response.
- **Asynchronous with callback**, where the service requester calls an operation on the service provider and specifies a callback handler. When the response is available, the SCA runtime returns the response to the requester by calling this callback handler.

Asynchronous with deferred response SCA invocation

Asynchronous with deferred response SCA invocation is useful if the service requester has further processing to perform that does not depend on information to be returned in the response.

When using asynchronous with deferred response invocation, resources within the SCA runtime thread are held until a response is available from the service provider. If the service provider takes a hours or days to process the request, the invocation style asynchronous with callback should be used. Asynchronous with deferred response SCA invocation is shown in Figure 37 on page 49.

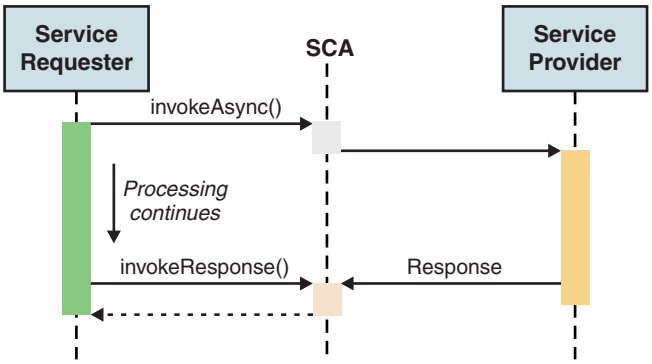


Figure 37. Asynchronous with deferred response SCA invocation

Example scenario

A car insurance comparison website would prefer to use the asynchronous with deferred response style of invocation. The service requester requests a search on car insurance from two different car insurance providers. The responses are not dependent upon each other. When the car insurance providers return a response, the service requester can choose the cheapest quote. Figure 38 shows how the service requester can call both service provider A and B.

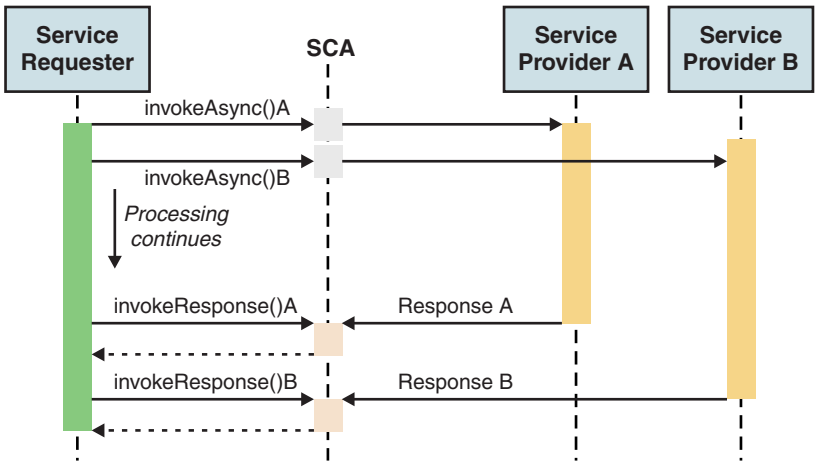


Figure 38. Car insurance comparison website using asynchronous with deferred response invocation

The service requester creates a request to service provider A, and calls the SCA runtime. The SCA runtime returns control to the requester immediately, allowing it to continue processing. The service requester is then able to create a second request for service provider B.

The service requester makes further calls to the SCA runtime to collect the responses. These are processed, and the final response is then returned to the user.

You can complete this scenario using the user interface. If you are using the SCA programming model directly, then you can use the following Java code, which shows how this scenario is made possible in the implementation:

```
public float getBestQuote() {

BOFactory bofactory = (BOFactory)ServiceManager.INSTANCE.
   locateService("com.ibm.websphere.bo.BOFactory");
DataObject transferDetails = bofactory.
  create("http://ServiceInvocationStyles", "CarDetails");
transferDetails.setString("make", "Ford");
transferDetails.setString("model", "Focus");

//Send the first request
Ticket quote1Ticket = locateService_InsuranceQuotePartner().
  invokeAsync("getQuote", transferDetails);

//Send the second request
Ticket quote2Ticket = locateService_AnotherInsuranceQuotePartner().
  invokeAsync("getQuote", transferDetails);

long waitFor = 10000;

//Get the response from the first service
Float quote1 = (Float)locateService_InsuranceQuotePartner().
  invokeResponse(quote1Ticket, waitFor);

//Get the response from the second service
Float quote2 = (Float)locateService_AnotherInsuranceQuotePartner().
 invokeResponse(quote2Ticket, waitFor);

if (quote1 < quote2) {
 return quote1;
} else {
 return quote2;
}
}
```

Asynchronous with callback SCA invocation

In this invocation style, the calling component (service requester) specifies a callback handler that will be used to receive and process the response. After the service requester has sent the request, the thread that made the request completes processing. SCA runtime resources associated with that thread are then released. When the response from the service provider is available, a new SCA runtime thread is used to call the callback handler to perform the processing required by the service requester.

This style of invocation is used for long running processes or services which could take hours or days to return a response. Asynchronous with callback SCA invocation is shown in Figure 39 on page 51.

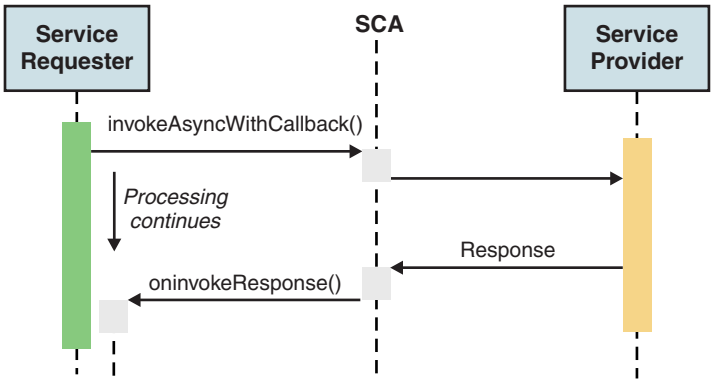


Figure 39. Asynchronous with callback SCA invocation

Example Scenario

A company would prefer to use this style of invocation if they perform audits. The service requester wants to update employees' payment details, and only needs to be notified when the update has been made.

The service requester sends a series of calls to update employees' payment details, to the service provider. The service provider invokes an audit service for each request. The audit can take from a few hours to a whole day to make a record of each employees payment details. When the audit service is complete, the response is returned to the service requester using a callback handler, and an e-mail is sent to the user to notify them that the update has been successful.

You can complete this scenario using the user interface. If you are using the SCA programming model directly, then you can use the following Java code, which shows how this scenario is made possible in the implementation:

```
  public void updateEmployees(List<Employee> employees) {
  BOFactory bofactory = (BOFactory)ServiceManager.INSTANCE.
    locateService("com.ibm.websphere.bo.BOFactory");


   Iterator <Employee> iterator = employees.iterator();
   while(iterator.hasNext()){
   Employee employee = (Employee) iterator.next();

   DataObject employeeDetails = bofactory.
     create("http://ServiceInvocationStyles", "Employee");
   employeeDetails.setString("firstName", employee.firstName);
   employeeDetails.setString("lastName", employee.lastName);
   employeeDetails.setInt("lastName", employee.age);

   locateService_EmployeeInterfacePartner().
     invokeAsyncWithCallback("updateEmployee", employeeDetails);
  }
 }
 /**
  * Method generated to support the async implementation using callback
  * for the operation "EmployeeInterface#updateEmployeeDetails(DataObject employee)"
  * of wsdl interface "EmployeeInterface"
  */
 public void onUpdateEmployeeDetailsResponse(Ticket __ticket,
   DataObject returnValue, Exception exception) {

  BOFactory bofactory = (BOFactory)ServiceManager.INSTANCE.
    locateService("com.ibm.websphere.bo.BOFactory");
  DataObject email = bofactory.create("http://ServiceInvocationStyles", "Email");
```

```
 email.setString("to", returnValue.getString("email"));
 email.setString("subject", "Employee record updated");
 email.setString("content",
   "Your employee details have been successfully updated to "+returnValue);

 locateService_EmployeeInterfacePartner().invoke("sendEmail", email);
}

public class Employee {
 String firstName;
 String lastName;
 int age;
}
```

Invocation from within the mediation flow component

The configuration of the Service Invoke mediation primitive or the Callout node can determine the invocation style that is used.

Within Integration Designer, the options for the invocation style configuration are:

- Sync
- Async
- Default

Sync invocation style

The synchronous invocation style can be used in all cases, regardless of whether the operation is one-way or request-response, and regardless of the invocation style used to invoke the mediation flow component.

Async invocation style

When the invocation style option is set to async, the invocation style used by the mediation flow component for service invocations is determined by the following options:

Table 2. Invocation style used by the mediation flow component for service invocations

How the mediation flow component is called	OneWay/ Request-Response	Require mediation flow to wait for service response when the flow component is invoked asynchronously with callback	Invocation style
Synchronous	Both	Any	Asynchronous
Asynchronous	Both	Any	Asynchronous
Asynchronous with callback	OneWay	Any	Asynchronous
	Request-Response	True	Asynchronous
		False	Asynchronous with callback

The Service Invoke mediation primitive and the Callout node both have a property that, when specified, prevents the mediation flow component from using the asynchronous with callback style for the service invocation. The Integration Designer tool identifies this property with a check box labeled **Require mediation flow to wait for service response when the flow component is invoked asynchronously with callback**.

Default invocation style

The default option is available, to support legacy applications.

When the invocation style is default, the invocation style used by the mediation flow component for service invocations is determined by three factors:

- The preferred interaction style of the target import or component.
- The invocation style used to call the mediation flow component.
- The use of the property: **Require mediation flow to wait for service response when the flow component is called asynchronously with callback.**

For more information about the Default invocation style, see: http://www.ibm.com/developerworks/library/ws-websphereesb2/.

Dynamic invocation

When a mediation module is developed and deployed, the flow of messages through the module is determined by the wiring of components. The configuration of the components is used to determine which external services are to be called.

Applications can override or change the configuration and reroute messages at run time. You might do this dynamically by overriding a value specified for an endpoint address. Alternatively, you can select a new target import. In each case, the message flow changes according to the information in the message.

For example, you can use Integration Designer to create a binding on an import. The import contains endpoint information, specifying the location of a remote service. This fixed endpoint information can be overridden dynamically by information carried in the message.

If a message flow is changed dynamically within a mediation module, the new route applies only to that message and any response coming back. When the message has been sent and any response returned, the dynamic routing changes are discarded and the original fixed values defined in the module are used as the default values again. Any response to a dynamic invocation returns using the same route as the dynamic invocation.

There are three main ways that dynamic invocation takes place:

- Dynamic override of a fixed endpoint, in which a service is called using any supported import binding. The service is available at a different endpoint from the endpoints originally specified when the mediation module was created and deployed.
- Dynamic invocation with a target import, in which a service is called using any supported import binding. The binding must already be available within the mediation module. It is selected at run time according to information contained in the message.
- Pure dynamic invocation, in which a service is called without needing an import in the mediation module. No additional information is required apart from the details in the message.

The Endpoint Reference (EPR) structure within the request message is used to provide the information required for a dynamic invocation. Details about the EPR can be obtained by retrieving information about the endpoint from a suitable storage location, such as WSRR or a database. This information is used to set EPR details for the message using the SCA Endpoint Reference API. The SMO structure:

```
/headers/SMOHeader/Target/address
/headers/SMOHeader/Target/@bindingType
/headers/SMOHeader/Target/@import
/headers/SMOHeader/AlternateTarget/address
/headers/SMOHeader/AlternateTarget/@bindingType
/headers/SMOHeader/AlternateTarget/@import
```

During a dynamic invocation, the invocation target type is identified by examining the endpoint URI.

- If the URI begins with `sca`, this indicates an SCA component.
- A URI prefix of `http` or `jms` is assumed by default to indicate a web service endpoint.
- A URI prefix of `http` does not by default indicate an HTTP service, and a URI prefix of `jms` does not by default indicate a JMS service. However, if the reference is wired to an import, or a target import is specified with a JMS or an HTTP binding, the URI is assumed to be JMS or HTTP, and not web services.
  - The `bindingType` field in the Endpoint Reference can be used to change this. For example, using the SCA Endpoint Reference API to set the `bindingType` field to a value of `EndpointReference.BINDING_TYPE_HTTP` or a mediation flow to set the `Target@bindingType` attribute to HTTP, results in a URI prefix of `http` in the EPR being interpreted as indicating an HTTP binding for that message.

Dynamic override of a fixed endpoint

Dynamic invocation by overriding a fixed endpoint can take place, when a service is called using any supported import binding.

This capability is used when the service is available from a different endpoint than originally specified when the mediation module was created and deployed.

Dynamic override of a fixed endpoint can be considered if a company is carrying out maintenance on the fixed service. In this instance, the company does not need to change their entire mediation flow to accommodate for a back up service. The company must modify the target address in the SMO header, which is done by registering the Back-up Service in WebSphere Service Registry and Repository (WSRR) and using an Endpoint Lookup mediation primitive to retrieve the Back-up Service target address details.

Figure 40 on page 55 shows how a message received by this module would usually be sent to the fixed endpoint for the Internal Service. The SMO can be modified to override the fixed endpoint and route the message to a Back-up Service.

A new target address is set for the message by changing SMO values, using the Endpoint Lookup mediation primitive, and registering the Back-up Service in WSRR. The message can then be dynamically routed to the Back-up Service.

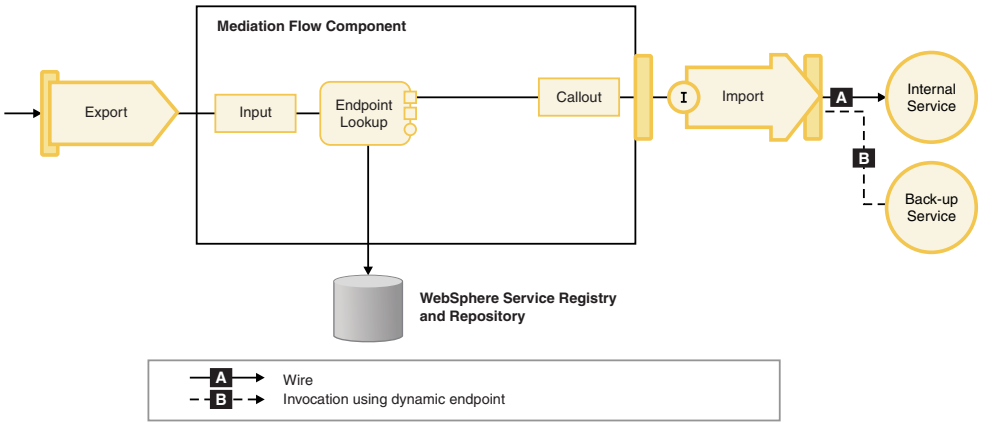


Figure 40. Dynamic override of a fixed endpoint using an Endpoint Lookup mediation primitive

Dynamic invocation with a target import

Dynamic invocation of endpoints can be enhanced using the `Target@import` attribute within the SMO header.

Dynamic invocation with a target import allows the configuration and protocol used to call a service to be dynamically selected at run time by identifying a supported import binding. The import and its binding must already be available within the module and is selected at run time according to information contained in the message.

You can use Integration Designer to create a mediation module that selects the target service dynamically at run time. The target services might use different protocols, different formats or different quality-of-service values. Each combination must be known at the time the mediation module is developed. This means that for each combination of protocol, format, and quality-of-service value, Integration Designer includes an import in the mediation module with an appropriate configuration.

For example a financial services company that provides an interactive web-based stock market service to its customers, has introduced tiered levels of service for Gold and Standard customers:

- Gold customers receive a reliable real-time stock quote, which is mapped within the mediation flow component using a JMS binding.
- Standard customers receive a delayed stock quote, which is mapped with a WebServices binding.

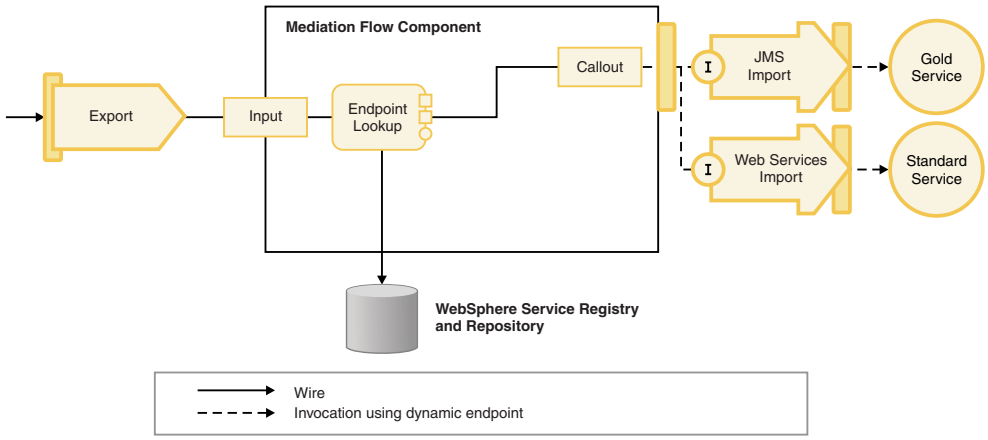


Figure 41. Dynamic invocation with a target import

As shown in Figure 41, two imports are needed in the mediation module for the two different bindings. During development of the mediation module, an import might be wired to another component or might be left unwired. If the module is unwired, a connection must be created at run time by specifying a target import and creating the connection dynamically.

In addition to dynamically selecting an import, a new target address can also be set using the incoming message content and routing information. The mediation flow component sets the new address by changing the SMO values using the Endpoint Lookup mediation primitive, and registering the Gold and Standard Services in WSRR. The choice between the target service destinations is made dynamically at run time using available metadata provided by the message or by WSRR. The name of the target import is placed in the SMO Header by the Endpoint Lookup mediation primitive. If an import is targeted that does not support the target endpoint, a ServiceRuntimeException will occur.

Pure dynamic invocation

Pure dynamic invocation calls a service without needing an import in the mediation module.

No additional information is required apart from the details in the message. An example is where the intended service is identified by a name, which is looked up at runtime. Pure dynamic invocation assumes a specific set of configuration parameters.

Web services and SCA URLs are supported, but other types are not because of their need for additional configuration that is provided by an import binding.

The default for pure dynamic invocation using a Web Service binding is SOAP 1.1, using HTTP or JMS as specified in the target URL. Pure dynamic invocation is also supported for SOAP 1.2 when using HTTP by specifying an appropriate value for the bindingType field of the EPR. Pure dynamic invocation using SOAP 1.2 and JMS is not supported.

The default for pure dynamic invocation using a Web Service binding is SOAP 1.1, using HTTP or JMS as specified by the target URL. Pure dynamic invocation is also supported for SOAP 1.2 with HTTP.

- For pure dynamic invocation using SOAP 1.1 and HTTP or JMS, the binding type should either be unset or it should be set to one of the following values:
  - Using the SCA Endpoint Reference API:
    - EndpointReference.BINDING_TYPE_WEB_SERVICE
    - EndpointReference.BINDING_TYPE_WEB_SERVICE_SOAP_1_1
  - Using the SMO structure
    - Target@bindingType=WS
    - Target@bindingType=WS1.1
  -
- For pure dynamic invocation using SOAP 1.2 and HTTP, the EndpointReference binding type should be set to:
  - EndpointReference.BINDING_TYPE_WEB_SERVICE_SOAP_1_2 using the SCA Endpoint Reference API
  - Target@bindingType=WS1.2 using the SMO structure

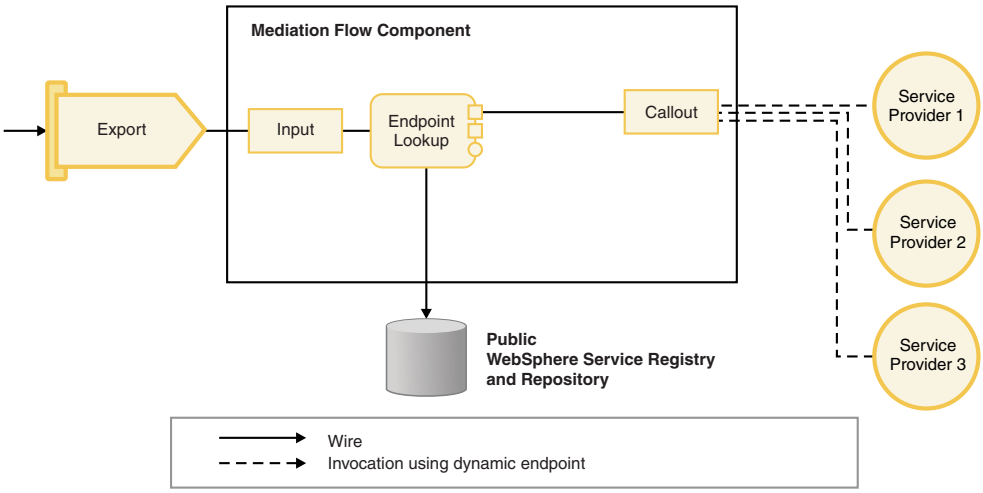


Figure 42. Pure dynamic invocation

An example of when pure dynamic invocation could be used is with a search engine. Figure 42 shows how a call can be made to the mediation flow component. An Endpoint Lookup mediation primitive is used to retrieve all endpoints from a public WSRR. A public WSRR allows companies to register themselves with service providers on an ongoing basis. This indicates that the endpoint is constantly changing, and so a pure dynamic environment is required. The advantage of pure dynamic invocation, is that the mediation module does not need to be changed in order to respond to new service providers.

Dynamic endpoint selection

In a mediation flow, you can select a service address based on a condition or property in your flow, and call an endpoint using the selected service. This section details some of the different mediation primitives that retrieve an endpoint address from a registry, to route the message to the retrieved endpoint.

Endpoint Lookup mediation primitive

Use the Endpoint Lookup mediation primitive to dynamically route messages to appropriate service endpoints. The Endpoint Lookup searches for service information in a WSRR registry.

You can use the Endpoint Lookup mediation primitive to retrieve service endpoint information from a WSRR registry that can be local or remote. The service endpoint information can relate directly to web services, or to SCA module exports, or to manually defined MQ, JMS and HTTP services.

In order to use the Endpoint Lookup mediation primitive you need to add service endpoint information to your WSRR registry.

- If you want to extract service endpoint information about web services, your WSRR registry must contain either the appropriate WSDL documents, or SCA modules that contain exports using web service bindings.
- If you want to extract service endpoint information about exports that use the other SCA bindings, your WSRR registry must contain the appropriate SCA modules.
- If you want to extract service endpoint information about services that are accessed using MQ, JMS or HTTP but are not defined in a SCA module, your WSRR registry must contain an appropriate "Manual JMS/HTTP/MQ endpoint with associated Interface" Business Object with the endpoint information and associated interface correctly set.

The Endpoint Lookup mediation primitive lets you retrieve service endpoint information that relates to the following:

- Web services using SOAP/HTTP.
- Web services using SOAP/JMS.
- SCA module exports with Web Service bindings, using SOAP/HTTP.
- SCA module exports with Web Service bindings, using SOAP/JMS.
- SCA module exports with the default SCA binding.
- SCA module exports with the MQ binding.
- SCA module exports with the MQ JMS binding.
- SCA module exports with the JMS binding.
- SCA module exports with the Generic JMS binding.
- SCA module exports with the HTTP binding.
- Manually defined MQ endpoints with associated interfaces.
- Manually defined JMS endpoints with associated interfaces.
- Manually defined HTTP endpoints with associated interfaces.

When the Endpoint Lookup mediation primitive receives a message it sends a search query to the registry. The search query is constructed using the Endpoint Lookup properties that you specify and the query might return nothing, or it might return one or more service endpoints.

The Endpoint Lookup primitive will update the target structure in the SMO header with the first service endpoint. The Endpoint Lookup primitive will then update the alternative target list with the remaining service endpoints.

Example scenario

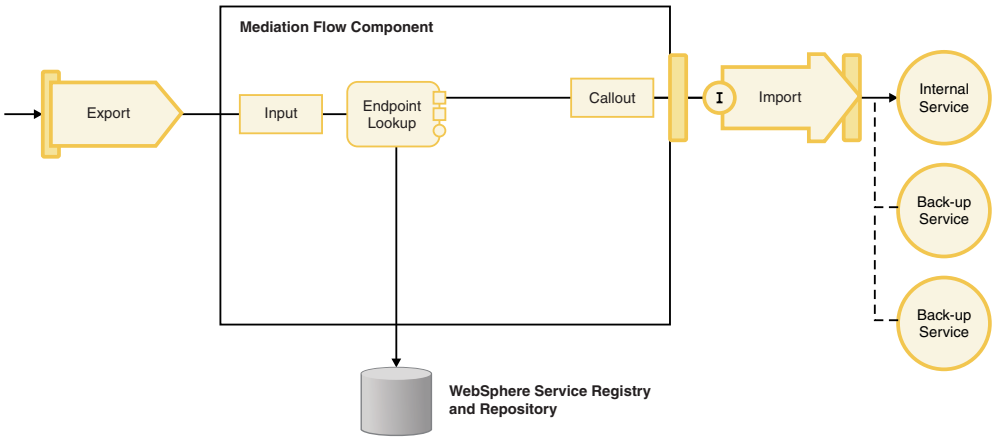


Figure 43. Endpoint Lookup within a mediation flow

The fictional company Personal Holidays is making an internal service available to external clients using a mediation flow. If the server that runs the internal service fails, one or more back-up servers which are also hosting the internal service are called instead, as shown in Figure 43.

The scenario is implemented by registering the service endpoints of the main server and the back-up servers in WSRR. Personal Holidays then use the Endpoint Lookup Primitive to retrieve all of the endpoints from WSRR. Figure 44 shows the Endpoint Lookup mediation primitive's properties.

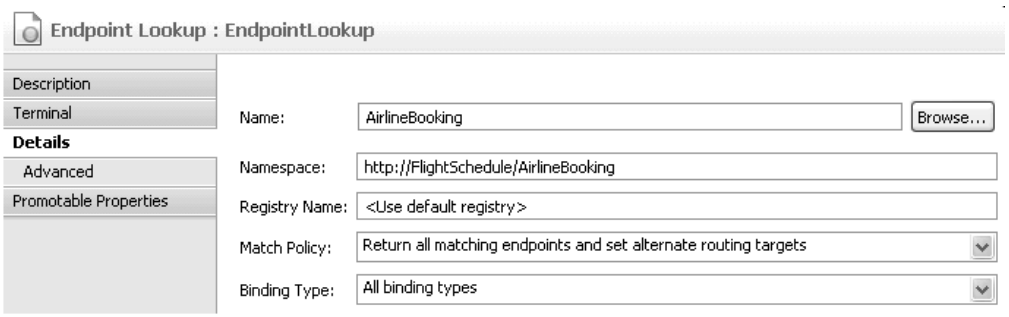


Figure 44. Endpoint Lookup mediation primitive properties

Personal Holidays also set the **Try alternate endpoints** property on the Callout node to try the main server endpoint first. If that fails, the Callout node will retry the call using the service endpoint information for the back-up servers. The Callout node properties are shown in Figure 45 .

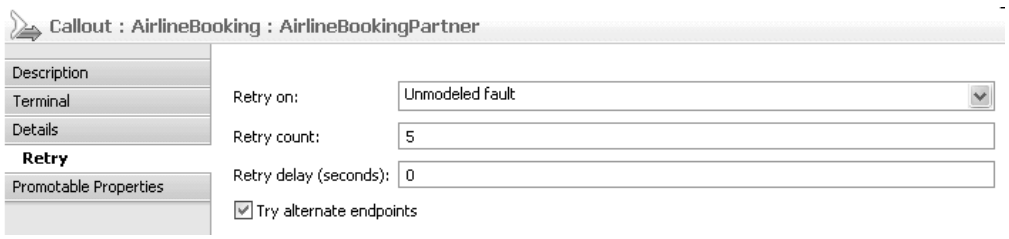


Figure 45. Try alternate endpoint property of the Callout node

By storing the service endpoints in WSRR, Personal Holidays can update their back-up services at any time, without needing to modify the mediation flow component.

UDDI Endpoint Lookup mediation primitive

Use the UDDI Endpoint Lookup mediation primitive to retrieve service endpoint information from a UDDI version 3 registry. The UDDI registry only holds Web Services information.

To use the UDDI Endpoint Lookup mediation primitive you need to add service endpoint information to your registry.

The UDDI Endpoint Lookup mediation primitive lets you retrieve service endpoint information that relates to the following:

- Web Services using SOAP/HTTP.
- Web Services using SOAP/JMS.

When the UDDI Endpoint Lookup mediation primitive receives a message it sends a search query to the registry. The search query is constructed using the UDDI properties that you specify and the query might return nothing, or it might return one or more service endpoints.

The UDDI Endpoint Lookup primitive will update the target structure in the SMO Header with the first service endpoint. The UDDI Endpoint Lookup mediation primitive will then update the alternative target list with the remaining service endpoints.

The UDDI Endpoint Lookup "Example scenario" on page 58, can be implemented with a UDDI version 3 registry, and a UDDI Endpoint Lookup mediation primitive in place of WSRR.

Service Level Agreement Endpoint Lookup mediation primitive

Use the Service Level Agreement (SLA) Endpoint Lookup mediation primitive to look up endpoints referenced from an active service level agreement (SLA) between a service consumer and a service provider.

If you use WebSphere ESB to integrate and model your business in WSRR, using its Governance Enablement Profile (GEP), you can select dynamic endpoints based on a number of factors modelled in that profile. For example:

- Whether the consumer of the endpoint has a valid SLA for the endpoint.
- Whether the particular SLA is active.
- Whether the endpoint is online.
- Whether the endpoint has a certain desired classification; for example, whether it is Production or Development.

Example scenario

The fictional company Personal Holidays is making an internal service available to external clients using a mediation flow. They want to use the WSRR GEP to ensure that only consumers with a valid SLA can access the internal service.

The scenario is implemented in a similar way to the "Example scenario" on page 58, using an SLA Endpoint Lookup mediation primitive in place of the Endpoint Lookup mediation primitive. The SLA Endpoint Lookup primitive is configured to retrieve a consumer ID from the SMO and to relay that information to WSRR.

WSRR returns endpoint information only for services that have an SLA and the SLA consumer matches the provided consumer ID, as shown in Figure 46.

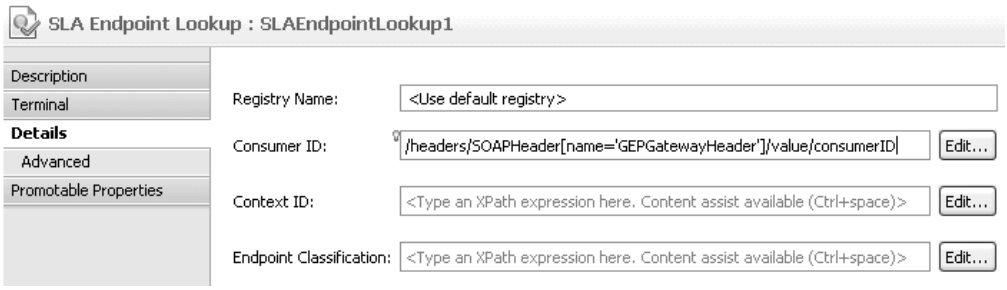


Figure 46. SLA Endpoint Lookup

Custom Endpoint Lookup mediation primitive

There are situations in which the supplied Endpoint Lookup mediation primitives are not sufficiently flexible. In these circumstances, use a Custom mediation primitive. If you need to access data in a WSRR registry, use the WebSphere ESB WSRR proxy.

Example scenario

The fictional company My Airways has registered some WSDL-defined service providers in a WSRR registry, and want to retrieve the service endpoint URL that is based on a user property, which is determined from the SMO at runtime. My Airways use a Custom mediation primitive to retrieve the service endpoint URL and set it in the SMO, so the service providers are called dynamically.

The Custom mediation primitive code completes three steps:

- It sends an XPath query to the WSRR registry to look up the provider service WSDL information.
- It processes the results returned from WSRR, retrieving the service endpoint URL.
- It sets the service endpoint URL into the SMOHeader Target address field. The URL is used when an import with a Web Services binding is called, and the Callout node has been configured to use a dynamic endpoint.

The following is the Custom mediation primitive code:

```
String endpointAddress = null;

// Get the factory and then get the default instance of a ServiceRegistryProxy.
Can also use getServiceRegistryProxy("RegistryName")
com.ibm.wsspi.sibx.mediation.wsrr.client.ServiceRegistryProxy srProxy =
com.ibm.wsspi.sibx.mediation.wsrr.client.ServiceRegistryProxyFactory.getInstance().
getDefaultServiceRegistryProxy();

try
{
   // Set up the WSRR XPath query - this query looks for WSDL Ports that
    have the PortType name 'MyAirwaysAccount'and the required customerType user property
   // and the PortType namespace 'http://www.MyAirways.com/AccountManagement/'
   String customerType = smo.getString("/body/bookFlight/customerType");
    String query = "/WSRR/WSDLService/ports[binding(.)/portType(.)
   [@name = 'MyAirwaysAccount' "+
       "and @namespace='http://www.MyAirways.com/AccountManagement/'
       and @customerType ='" + customerType + "']]";
```

```
   // Run the query & get the results
  com.ibm.wsspi.sibx.mediation.wsrr.client.data.
  ServiceRegistryDataGraphList list = srProxy.query(query, -1, false);

   // Iterate through the results
   Iterator<DataGraphType>  resultsIterator = list.getDataGraphs().iterator();
   while(resultsIterator.hasNext() && endpointAddress == null)
   {
     DataGraphType dgt = resultsIterator.next();
     BaseObject[] baseObjects = dgt.getWSRR().getArtefacts();


    com.ibm.ws.management.discovery.EndpointAddress endpointAddress = null;

     // Find the SOAP Address
    for(int i=0; i<baseObjects.length; i++)
    {
         BaseObject bo = baseObjects[i];
         if(bo instanceof com.ibm.wsspi.sibx.mediation.wsrr.client.jaxrpc.WSDLService)
         {
           com.ibm.wsspi.sibx.mediation.wsrr.client.jaxrpc.WSDLPort[] ports = ((com.ibm.
          wsspi.sibx.mediation.wsrr.client.jaxrpc.WSDLService)bo).getPorts();
            // Look for the SOAP Address in the first port

            endpointAddress = ports[0].getSOAPAddress().getLocation();
            break;
         }
      }
   }
}
catch(ServiceRegistryProxyException e)
{
   e.printStackTrace();
}

System.out.println(endpointAddress);

// If we have successfully retrieved an endpoint from WSRR, set it into the SMO
if(endpointAddress != null)
{
   smo.setString("headers/SMOHeader/Target/address", endpointAddress);
}
```

Chapter 8. Service gateway

A service gateway is a single access point and acts as a proxy for multiple services. A service gateway enables transformations, routing, and common processing across all the services.

A service gateway module is a single mediation that handles the requests for multiple service consumers and providers. There are four steps that are typical of any service gateway:

1. **Common processing** - once the message is received by the gateway, common processing occurs for all messages, such as adding protocol level headers or logging the message.
2. **Service identification** - the message that is processed by the gateway needs to be identified as a certain service type. For example, the message is queried to determine if it is for service provider A, B, or C.
3. **Endpoint routing** - when it is known that a message will be delivered to a particular service provider, it is mapped to a network-addressable endpoint so that the message can be forwarded to the service provider.
4. **Service-specific processing** - any processing that is required for the particular target service is performed.

The order of the four steps can be changed according to different scenarios.

There are three service gateway patterns provided out of the box that are supported by WebSphere ESB:

1. "Dynamic service gateway pattern" on page 69
2. "Static service gateway pattern" on page 70
3. "Proxy gateway pattern" on page 71

Each of these patterns follow a general format within the tooling environment in IBM Integration Designer, as shown in Figure 47 on page 64. The service gateway is represented as an SCA module. An export exposes the endpoint and has the service gateway interface attached. The service gateway interface instructs the runtime that it is in the service gateway mode.

© Copyright IBM Corp. 2007, 2011

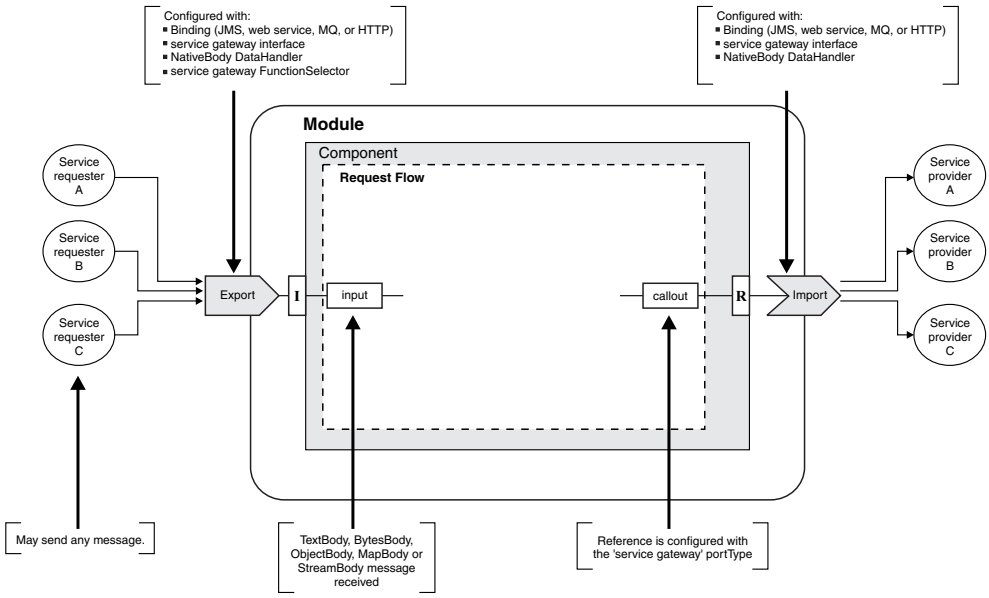


Figure 47. General pattern for a service gateway module

The service gateway interface has two operations: requestOnly and requestResponse. Both operations use messages that have an anyType structure. An anyType message structure allows it to be one of five core structures that represent raw data that is sent to the mediation flow component, as shown in Figure 48. This is not to accommodate any business object. Implement one or both of the interface operations.

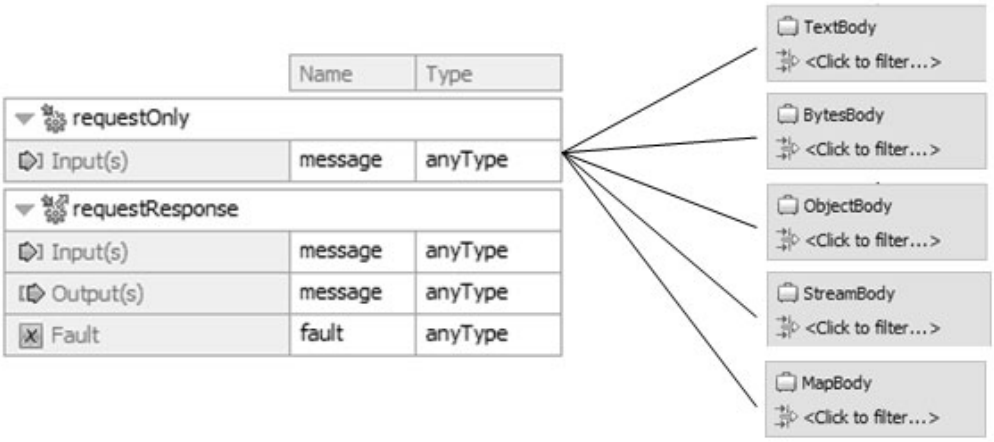


Figure 48. Five structures that represent raw data

The binding type must be taken into consideration, to determine which business object is created. The following table shows the business objects that are created for the different binding types.

Binding type	Business object created
WebSevice Binding	TextBody
HTTP Binding	

Binding type	Business object created
HTTP Content-Type=text/*	TextBody
HTTP Content-Type=application/soap+xml	TextBody
HTTP Other	BytesBody
JMS Binding	
JMS TextMessage	TextBody
JMS BytesMessage	BytesBody
JMS StreamMessage	StreamBody
JMS MapMessage	MapBody
JMS ObjectMessage	ObjectBody
MQ Binding	
MQ Format = MQSTR	TextBody
Other	BytesBody

The content of the message is in a serialized form (such as XML) when it reaches the mediation flow component and is held within a TextBody structure. Header information is parsed as a business object and can be found in the header section of the Service Message Object (SMO). To change the body of the message, it can be parsed within the mediation flow component and changed using transformation mediation primitives and then serialized before it is passed to the service gateway import. To parse the message within the mediation flow component, the schema information associated with the message type should be available to the module.

There are two ways that WebSphere ESB can parse a message within the service gateway pattern:

1. "Manual parsing of the gateway message"
2. "Automatic parsing and any message type support" on page 66

Manual parsing of the gateway message

The DataHandler mediation primitive can be used to convert a message from a serialized form (such as XML) to a business object structure. This mediation primitive differs from other transformation primitives such as XSLT or the BO Mapper, which convert one business object into another business object.

The DataHandler mediation primitive is used extensively within the static service gateway pattern, where you must parse the serialized message body into the concrete business object structure. Each DataHandler primitive can only handle a single message type, therefore branching and multiple DataHandler primitives are required, as shown in Figure 49 on page 66.

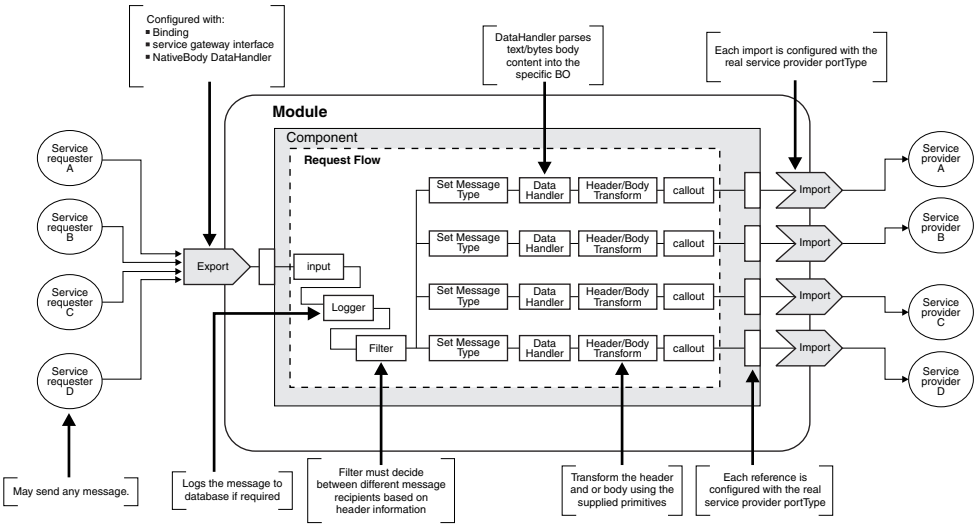


Figure 49. DataHandler mediation primitive used in a service gateway

Automatic parsing and any message type support

In IBM Integration Designer, as a default all terminals are defined with concrete message types. The terminals can be defined to have any message structure, which is set on individual terminals, allowing any business object to move through the mediation flow. WebSphere ESB parses the serialized message body into a concrete business object at the edge of the mediation flow component (input and callout). To enable this, the input nodes must be configured with the feature titled **Automatically convert the service gateway message**. This allows the parsing and serialization to occur automatically when the inbound message is XML encoded within a TextBody structure. The overall service gateway design is shown in Figure 50.

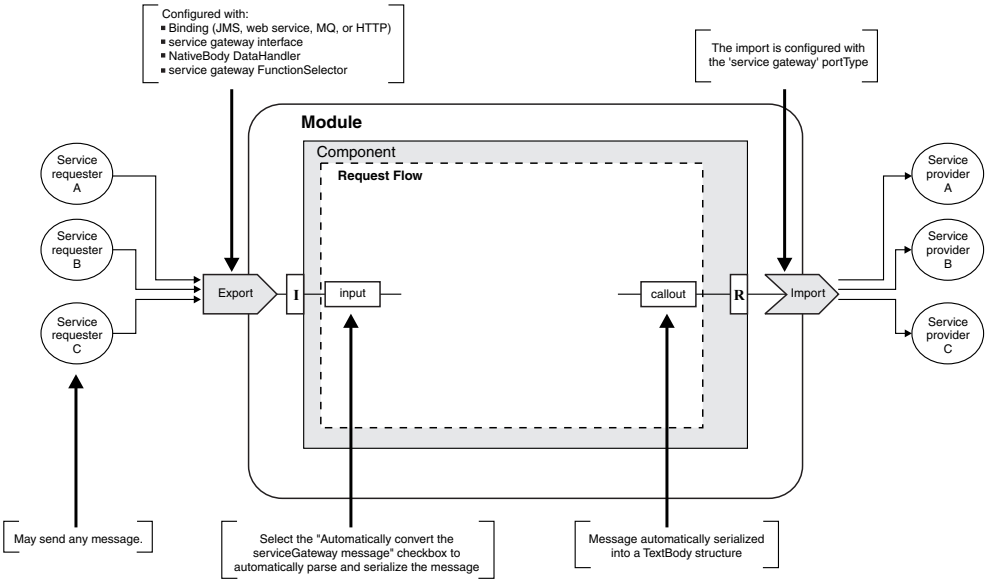


Figure 50. Automatic parsing used in a service gateway

Patterns Explorer

IBM Integration Designer provides patterns that accelerate the development of a service gateway. Each pattern includes a wizard that generates a service gateway implementation, which is based on a set of parameters you provide. You can make updates to the generated implementation to complete your service gateway.

Patterns can be accessed through IBM Integration Designer Patterns Explorer, where you can create an integration solution for common scenarios. The following service gateway patterns are available:

- Proxy gateway pattern
- Static gateway pattern
- Dynamic service gateway

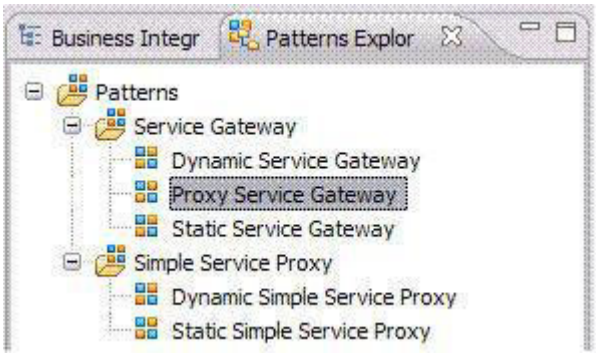


Figure 51. Pattern Explorer in IBM Integration Designer

Once a pattern has been selected, a pattern specification view opens which provides documentation about the pattern and its use. Click the 'Create New Instance' button, which opens a pattern configuration view. Specify the parameters used to generate the pattern, as shown in Figure 52 on page 68.

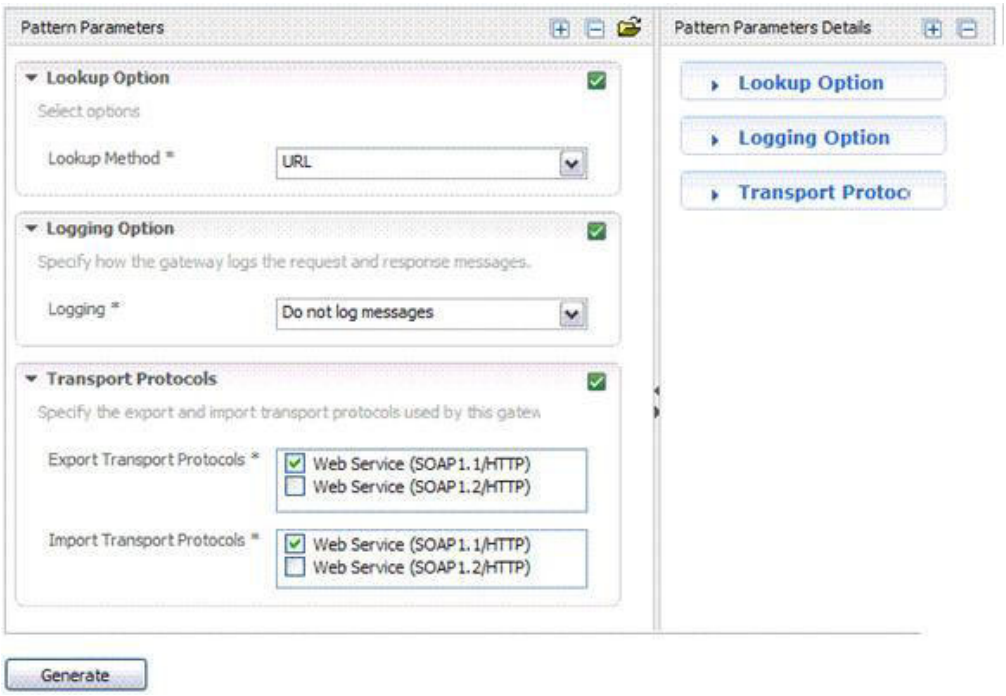


Figure 52. Pattern Explorer configuration view

Once the parameters have been entered, click the Generate button which creates the SCA module artifacts and provides a summary of information about the generated module as shown in Figure 53 on page 69.

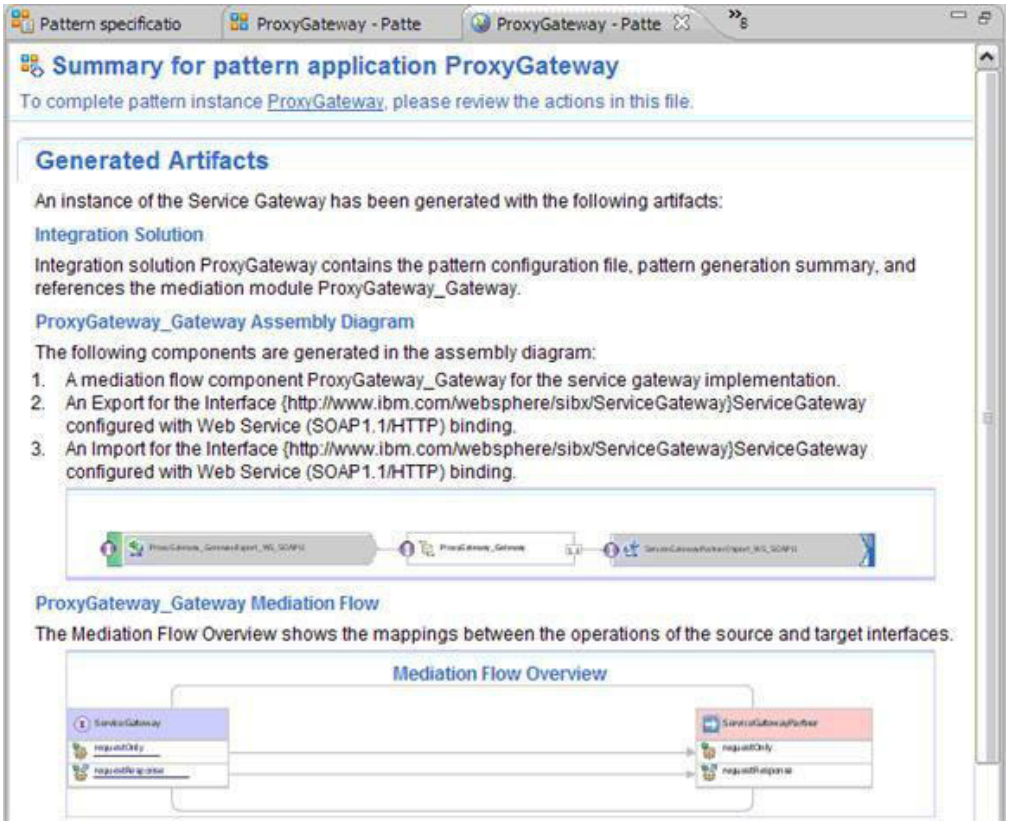


Figure 53. Pattern Explorer summary information

Dynamic service gateway pattern

With the dynamic service gateway, you can enter new services into a gateway without any modification to the deployed module.

Schema information for services that are handled by the gateway are optional, if access to the body of the message is not required. This provides the flexibility to build a customizable service gateway, adding and removing services as required based on an external registry or database for endpoint resolution. Figure 54 on page 70 shows the common pattern used for a dynamic service gateway. The dynamic service gateway can be used with either web services, JMS, HTTP or MQ bindings.

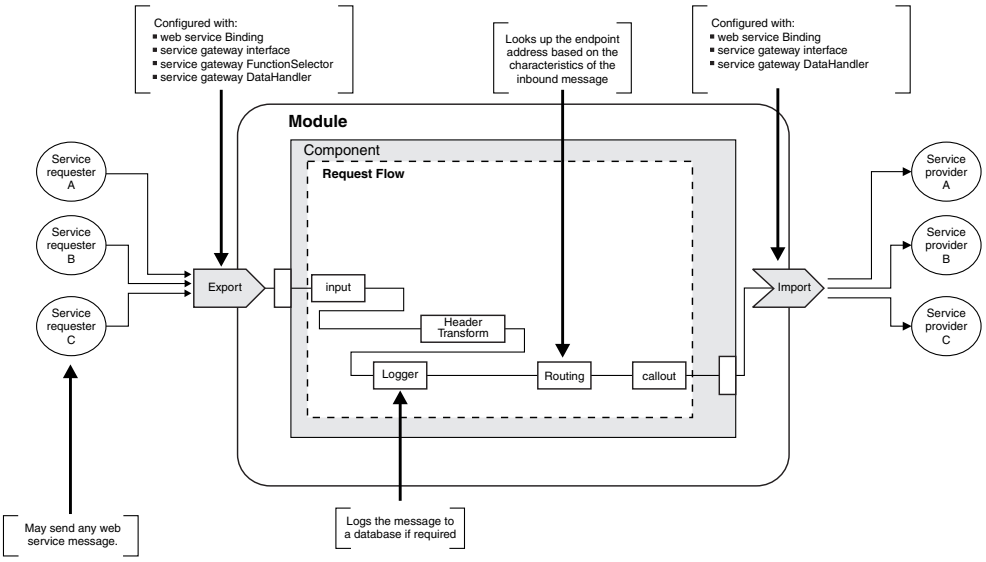


Figure 54. Common pattern that is used for a dynamic service gateway

To create a web service client, you retrieve the WSDL file for the service provider, and pass it into the web service tooling to generate a client stub. For a dynamic service gateway the creation of this WSDL file is a manual process, where you must download the target service provider WSDL, replace any policy information with that attached to the service gateway export and then override the endpoint to that of the service gateway.

Example scenario

A fictitious airline company has four different services that are based on different protocols:

- Flight booking - MQ binding
- Seat reservation - JMS binding
- Flight schedule - HTTP binding
- Food selection- web services binding

The airline company have a requirement that all messages must be audited in a single unit before the message reaches the service provider. In the future the airline company want to be able to associate new services without redeploying the gateway. The dynamic service gateway is used because it has a multi protocol requirement.

Static service gateway pattern

The static service gateway provides a single access point for many service requesters and service providers.

The available service types are static and cannot be changed after the module has been deployed to the runtime. It is possible to override the endpoint that the service exposes by using a service registry or database. The benefit of using this gateway pattern, is that the message body can be changed easily within the module. The static service gateway pattern can be used with either HTTP, JMS, MQ or web services. Figure 55 on page 71 shows the common pattern used for a

static service gateway.

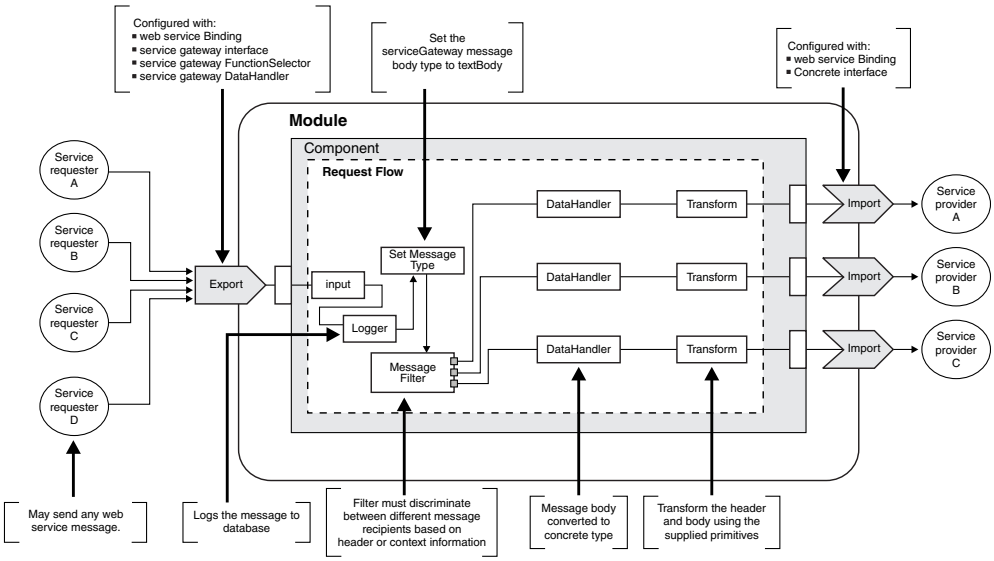


Figure 55. Static service gateway pattern

To create a web service client, you retrieve the WSDL file for the service provider, and pass it into the web service tooling to generate a client stub. For a static service gateway the creation of this WSDL file is a manual process, where you must download the target service provider WSDL, replace any policy information with that attached to the service gateway export and then override the endpoint to that of the service gateway.

Example scenario

A fictitious car insurance comparison web site provides quotes to customers from 3 different car insurance providers. Each car insurance provider service has a different security configuration.

The comparison web site must audit the body of the message before it is forwarded on to the relevant service providers. The static service gateway is used because, only a section of the message is audited. Since each service provider has a different security configuration, a different SCA import is needed for each service provider, which is provided using the static service gateway pattern.

Proxy gateway pattern

The proxy gateway provides a generic capability across web service only service providers. This pattern has no dependencies on other products, such as a registry.

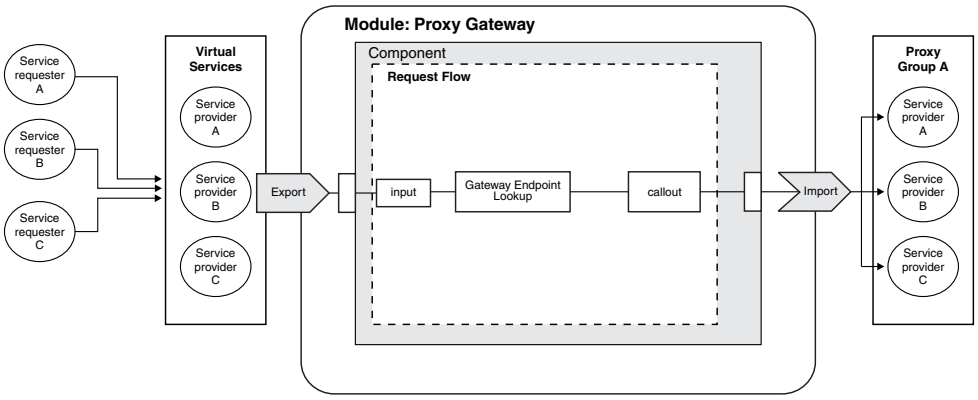


Figure 56. Proxy gateway

Figure 56 shows the relationship between a proxy gateway, proxy group, service provider and virtual service.

- Virtual service - the exposed endpoint of the proxy gateway for a service provider within the associated proxy group.
- Proxy gateway - the entity that provides the gateway function for one or more proxy groups.
- Proxy group - a collection of service providers that are grouped together into a logical set.

Configuration store

WebSphere ESB includes a built-in configuration store which holds data about the relationships between the proxy gateway, proxy groups and associated service providers.

To change the data within the built-in configuration store, a web based interface is provided. Using the interface you can create, delete and amend virtual services associated with a proxy group.

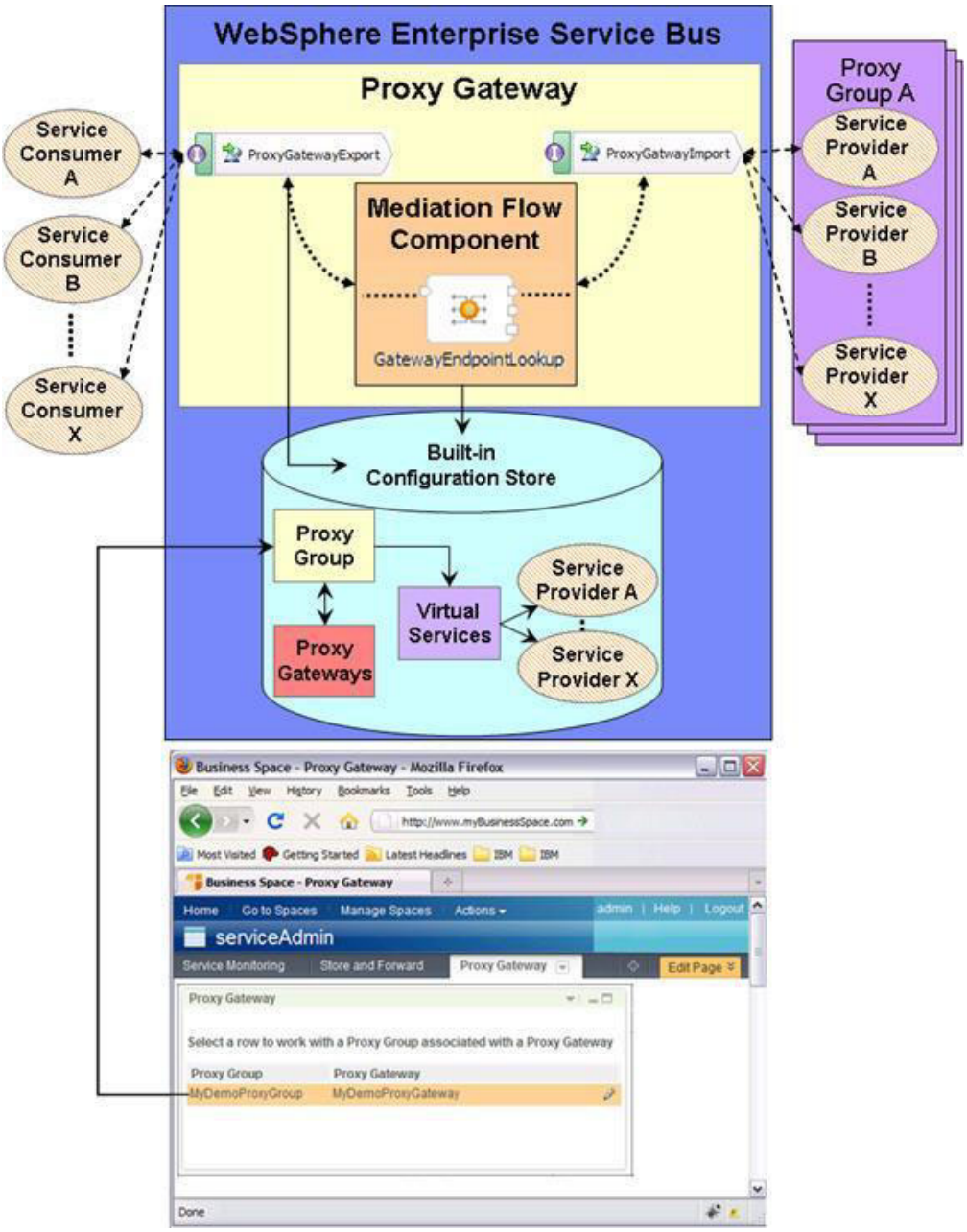


Figure 57. High level interactions of the proxy gateway, the configuration store and the web based interface

Figure 57 shows the high level interactions of the proxy gateway, the configuration store and the web based interface. The "Built-in Configuration Store" maintains the associations between proxy groups and proxy gateways. The proxy gateways contain definitions of virtual services and their defined endpoints. The browser window shows the Business Space widget that is used to administer the proxy groups and their virtual service definitions.

Virtual services

Network addressable endpoints for virtual services

The proxy gateway has an individual network addressable endpoint for each virtual service it is exposing, for example:

```
http://localhost:9080/proxyGateway/sca/ProxyGatewayExport/VirtualServiceA
http://localhost:9080/proxyGateway/sca/ProxyGatewayExport/VirtualServiceB
...
```

The proxy gateway web service endpoint accepts messages based on a URL pattern instead of the typical single URL pattern of a normal web service export.

Normal web service export URL:

```
http://localhost:9080/proxyGateway/sca/ProxyGatewayExport
```

Proxy gateway web service export URL pattern:

```
http://localhost:9080/proxyGateway/sca/ProxyGatewayExport/*
```

Virtual service WSDL generation

To create a web service client, you would normally retrieve the WSDL file for the service provider, and pass it into the web service tooling to generate a client stub. It might be difficult to retrieve the WSDL from a service gateway because the WSDL that the client uses is a combination of the WSDL for the gateway and the WSDL for the target service provider. For a dynamic or static service gateway the creation of this WSDL file is a manual process: you must download the target service provider WSDL, replace any policy information with the WSDL attached to the service gateway export and then override the endpoint to that of the service gateway. This lets the service consumer know that a service gateway is being used, and because a manual process was used to edit it, there is a chance that mistakes have been made when it was edited.

Automated WSDL resolution is supplied for the proxy service gateway, based on the following pattern:

```
<NORMAL-EXPORT-URL>/<VIRTUAL-SERVICE-NAME>?wsdl
```

For example:

```
http://localhost:9080/proxyGateway/sca/ProxyGatewayExport/VirtualServiceA?wsdl
```

Virtual service endpoint resolution

Since the gateway is a proxy for multiple service providers, it is necessary to be able to determine the type of message the gateway has received, and resolve this to a network addressable endpoint. The proxy gateway handles this automatically using the dedicated URL and the information in the built-in configuration store. The virtual service name is extracted ffrom the URL pattern and passed to the built-in configuration store to resolve it to a network addressable endpoint.

Introduction to Business Space

Business Space provides a browser-based graphical user interface that lets users interact with content from products in the IBM Business Process Manager portfolio. It provides a customizable and collaborative environment for monitoring, reviewing, and administering common business processes, such as human task flows, mediations and modeling.

Business Space includes the concept of spaces, templates, pages and widgets. A space is a collection of pages based on a template, and a page is a canvas that contains a number of widgets. Widgets are the pluggable user interface

components that you use to define the functionality of your business spaces. WebSphere ESB and IBM Business Process Manager ship a number of built-in templates and widgets.

In the context of WebSphere ESB, Business Space provides administrative capabilities. You can control and manage the applications installed on the system using these capabilities.

Proxy gateway widget

The proxy gateway widget displays a list of proxy groups that exist in the built-in configuration store.

Each proxy group in the list includes information about the installed proxy gateways that are associated with the group. When a proxy gateway is installed on a server, the proxy groups it references are automatically added to the configuration store if they don't already exist. If you uninstall a proxy gateway from the server, the proxy groups it is associated with are not deleted. This prevents the unintentional loss of configuration data for the proxy group when no proxy gateway applications are installed that reference it. You must delete the proxy group if it is no longer needed.

Figure 58 shows the proxy gateway widget in Business Space. It shows what you see when:

- a proxy group is shared by multiple proxy gateways
- a proxy gateway has multiple proxy groups
- a proxy group exists without an associated proxy gateway

Figure 58. Proxy gateway widget in Business Space

Figure 58 shows:

- There are two proxy gateways: PGW1_Gateway and PGW2_Gateway.
- There are four proxy groups: PGW1ProxyGroup, PGW2ProxyGroup, PGW3ProxyGroup, PGWSharedGroup.
- The first row has a proxy group (PGW1ProxyGroup) that is associated with a single proxy gateway (PGW1_Gateway).
- The second row has a proxy group (PGWSharedGroup) shared by multiple proxy gateways (PGW1_Gateway, PGW2_Gateway).
- Rows two and three illustrate a proxy gateway (PGW2_Gateway) that is associated with multiple proxy groups (PGWSharedGroup, PGW2ProxyGroup).

- The fourth row has a proxy group (PGW3ProxyGroup) without an associated proxy gateway.

Using the proxy gateway widget

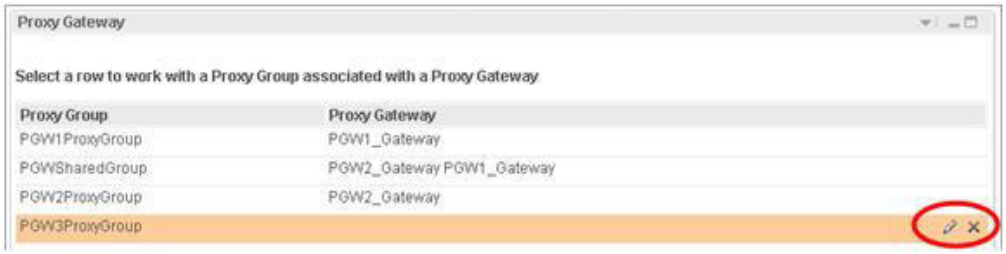


Figure 59. Select a proxy gateway widget

Figure 59 shows how to select a proxy gateway to edit. To do this you must:

1. Select the relevant proxy group row in the table, which displays a pencil icon.
2. Click the (pencil icon) which changes the widget to a new view where you can configure the virtual services for the proxy group.
3. If the selected row is for a proxy group without an associated proxy gateway, a (X icon) is visible where you can delete the proxy group.

When you create a new proxy group in the configuration store, there are no virtual services defined. When you edit it for the first time, the panel shown in Figure 60 is displayed, where you must add a virtual service by filling out the WSDL Location field.

Figure 60. Editing a new proxy group

To add a virtual service, you must enter the URL to the WSDL that is usually available by entering the service endpoint, followed by ?wsdl. For example:

```
http://localhost:9080/BankServiceProviderWeb/sca/accountExport1?wsdl
```

(If the service provider does not support the ?wsdl convention, the WSDL can be loaded from another location such as a registry.)

You must then click Add Service, which opens the panel for defining a virtual service, as shown in Figure 61 on page 77.

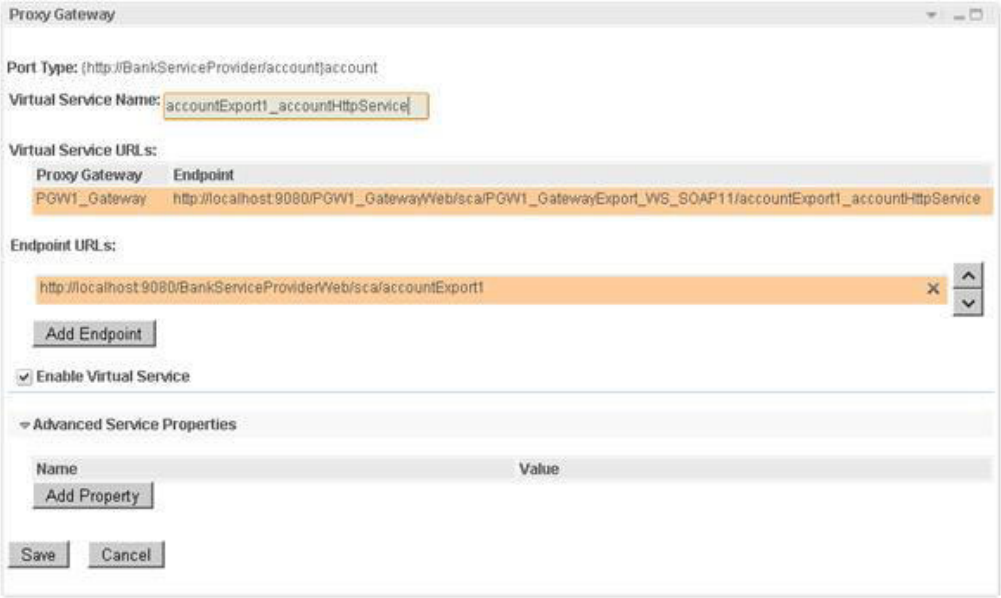


Figure 61. Editing a new proxy group

The information in Figure 61 can be separated into six sections:

1. Port Type - The WSDL port type for the virtual service.
2. Virtual Service Name - The key that is given to the URL of the export for the associated proxy gateways to create a dedicated URL for the virtual service. You can change the generated value to something more relevant. For example, the default value shown in the image is accountExport1_accountHttpService, which you can change to Account or AccountService.
3. Virtual Service URLs - Network addressable endpoints that are exposed for the virtual service via the associated proxy gateways.
4. Endpoint URLs - An ordered list of network addressable endpoints that are used to forward messages for this virtual service. These are available within the Target and AlternateTarget elements of the SMO after a Gateway Endpoint Lookup mediation primitive.
5. Enable Virtual Service - must be selected if the virtual service will be available to a Gateway Endpoint Lookup mediation primitive when it queries the built-in configuration store.
6. Advanced Service Properties - A series of user defined key value pairs associated with the virtual service. These are available within the EndpointLookupContext of the SMO, after a Gateway Endpoint Lookup mediation primitive.

When you click Save, the widget returns to the original list of virtual services associated with the proxy group.

Example scenario

A fictitious bank have web service enabled all service providers. They must put in place security and auditing across all service providers. To complete this, the bank create a proxy gateway. In the future the bank want to add more service providers without redeploying the application.

Selecting service gateway patterns

This section looks at some generic scenarios where you can use each of the service gateway patterns.

- **Web service based service gateway** - If a web service only service gateway is required, then use the proxy gateway pattern. A proxy gateway provides built-in routing capabilities and client side WSDL resolution. The other patterns, such as the dynamic service gateway, will need extensive user code to provide similar functionality.
- **Messaging or HTTP based service gateway with XML data encoding** - If the encoding style of the messaging and HTTP service gateways is XML serialization of business objects, then use the dynamic service gateway. WebSphere ESB provides an option to automatically parse incoming messages to the gateway. Therefore, access to the body of the message can be achieved using the dynamic service gateway pattern.
- **Messaging or HTTP based service gateway with non-XML data encoding** - If access to the body of the message is not required, use the dynamic service gateway. If access to the body of the message is required, use the static service gateway.
- **Protocol switching service gateway with XML data encoding** - When a multi-protocol service gateway is required and message encoding on all protocols is XML serialization of business objects, use the dynamic service gateway. WebSphere ESB provides an option to automatically parse incoming messages to the gateway. Therefore if access to the body of the message is required, use the static service gateway.
- **Protocol switching service gateway with non-XML data encoding** - If access to the body of the message is not required, use the dynamic service gateway. If access to the body of the message is required, use the static service gateway.

Chapter 9. Aggregation

Aggregation is a powerful and important enterprise service bus (ESB) pattern. An inbound request might map into several individual outbound requests. The responses from these requests can be used to enrich a final request to a service or can be aggregated into a single response for the original request.

Common scenarios are:

- A car comparison insurance quote system where a single request is submitted to an ESB that causes multiple insurance service providers to be queried before the best quote is returned.
- A holiday booking request arrives at the ESB and needs to be forwarded to separate hotel, flight and car hire booking systems, before a combined response can be returned.
- A store ordering system, where a salesperson can submit a request for a number of orders and have them dispatched. The original message coming into the ESB identifies the customer for each order via a customer ID. So, the order request message needs to be enriched with each customer's shipping details so that the dispatch step has all the information it needs to succeed.

When creating aggregation solutions using WebSphere ESB there are several core parts of the programming model that you must understand before going into the details of the patterns:

- The Fan Out mediation primitive
- The Fan Out context
- The Fan In mediation primitive
- The Service Invoke mediation primitive
- Aggregation block
- Shared context
- Special considerations

Fan Out mediation primitive

The Fan Out mediation primitive is normally used at the start of an aggregation. It provides the ability to either branch (Once mode) the mediation flow, or fire its output terminals multiple times (Iterate mode), based on the size of a repeating structure within the Service Message Object (SMO). For example, if the inbound message contains a structure that needs to be forwarded onto two services, you can branch the flow at the Fan Out mediation primitive. To call a service multiple times, each with individual requests from a repeating structure, you can use the Iterate mode of the Fan Out mediation primitive. You can configure the two modes from the Fan Out mediation primitive properties view, as shown in Figure 62 on page 80.

© Copyright IBM Corp. 2007, 2011

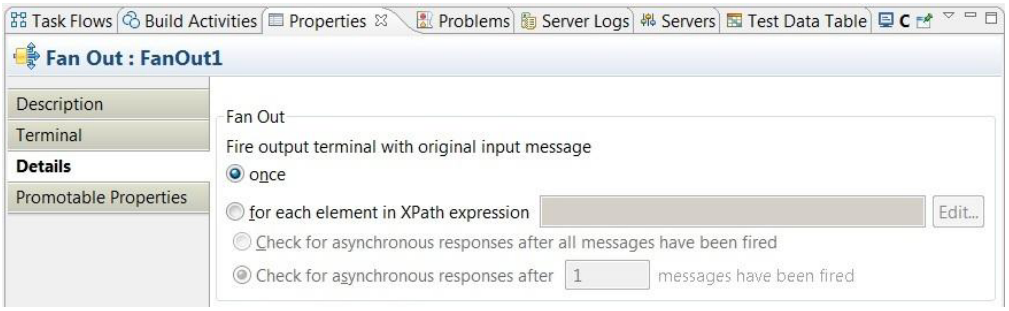


Figure 62. The Fan Out mediation primitive properties view

The mediation primitive is configured for Once mode by default, but this can be changed by selecting the **for each element in XPath expression** button and specifying the location of the repeating structure within the SMO.

Fan Out context

The Fan Out context is used to store the current item of a repeating element when Fan Out is used in Iterate mode. The Fan Out context contains an integer field containing the index of the occurrence of the repeating element along with an element containing the occurrence itself. For instance, if there was a repeating element of an element called *order*, the Fan Out context would look like this:

```
<FanOutContext>
 <iteration>0</iteration>
  <occurrence>
   <order>
    <id>1234567890</id>
   </order>
  </occurrence>
</FanOutContext>
```

Fan In mediation primitive

The Fan In mediation primitive is a partner to the Fan Out mediation primitive. The Fan Out mediation primitive is used to decide how many messages to generate, whereas the Fan In mediation primitive is used to decide when the aggregation can be considered complete. The Fan In mediation primitive is always partnered with a Fan Out mediation primitive in the same flow (for example, a Fan Out/Fan In mediation primitive pairing can not expand across a request/response flow boundary), and the Fan In mediation primitive acts as a decision point for when to leave the aggregation block, explained later in this section. The Fan In mediation primitive receives a number of messages until a decision point is reached, at which point the last message that was received is propagated to the output terminal. The Fan In mediation primitive can only be used in combination with the Fan Out mediation primitive.

Three types of decision points for the Fan In mediation primitive are supported:

- **Simple count**: the output terminal is fired when a set number of messages is received at the input terminal. However, more messages might be sent out by the Fan Out mediation primitive, because the Fan In mediation primitive count decision point can be reached more than once, resulting in multiple firings of the output terminal.

- **XPath decision**: the output terminal is fired if an XPath expression evaluation of the incoming message evaluates to *true*. It is possible for the output terminal to fire multiple times.
- **Iterate**: the Fan In mediation primitive waits to receive all messages produced by the corresponding Fan Out mediation primitive. All of the occurrences in the repeating element have been processed.

Figure 63 shows these three options, which are available in the Fan In mediation primitive properties view.

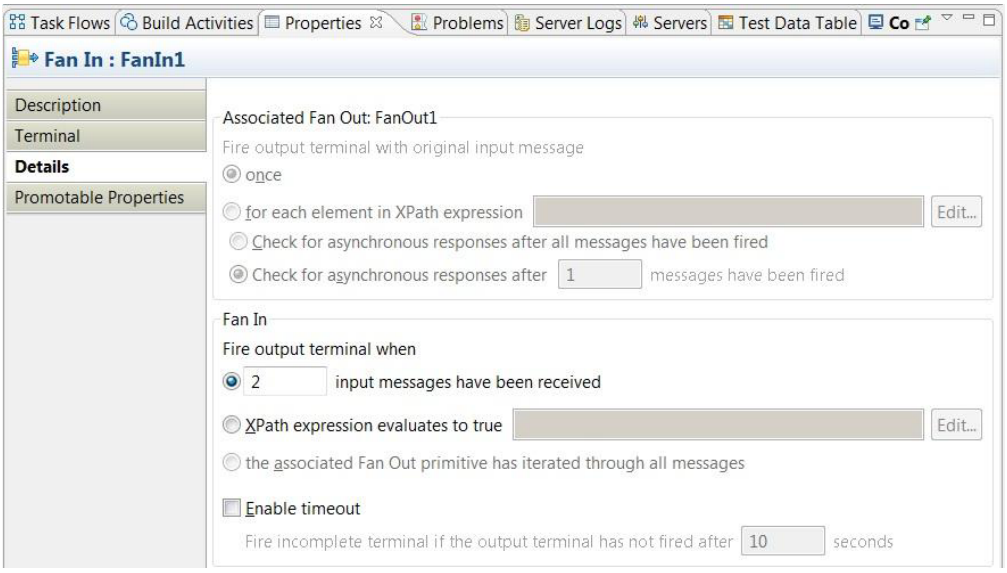


Figure 63. The Fan In mediation primitive properties view

The Fan In mediation primitive has two input terminals (**in** and **stop**), two output terminals (**out** and **incomplete**) and a fail terminal (**fail**). The input terminals are wired to accept a message and the other terminals are wired to propagate a message.

The **in** terminal accepts input messages until a decision point is reached. When a decision point is reached the **out** terminal is fired. Although the **out** terminal is fired when a decision point is reached, the **in** terminal can still accept input messages, under some circumstances. For example, if the count value is reached, the **out** terminal is fired. However, if the Fan Out mediation primitive is still sending messages, the Fan In mediation primitive can still receive messages.

You can stop a Fan In operation by wiring the preceding mediation primitive to the Fan In **stop** terminal. The **stop** terminal causes the **incomplete** output terminal to be fired, and this stops the associated Fan Out mediation primitive from sending any more messages. The **incomplete** output terminal is also fired if a timeout occurs. Table 3 on page 83 describes terminal behaviour further.

Service Invoke mediation primitive

The Service Invoke mediation primitive is used between the Fan Out and Fan In mediation primitives to call the services required. The Service Invoke mediation primitive is discussed in the Chapter 7, “Invocation of Services,” on page 43 chapter.

Aggregation block

When you create an aggregation solution, the Fan Out mediation primitive declares the start of the aggregation and a Fan In mediation primitive symbolizes the end of the aggregation. The logic between these two mediation primitives can be considered a repeating block if the Fan Out is configured in Iterate mode, or branching if configured in Once mode, as shown in Figure 64.

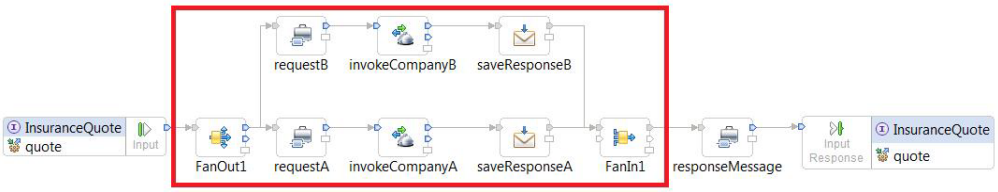


Figure 64. Example of an aggregation block

Shared context

WebSphere ESB has three user-defined data store contexts within the SMO:

- Data stored in the Transient context is available within either a request or response flow.
- Data stored in the Correlation context is available in both the request and response flows.
- Data stored in the Shared context is available in all branches and executions of an aggregation block.

To highlight the difference Figure 65 shows an example.

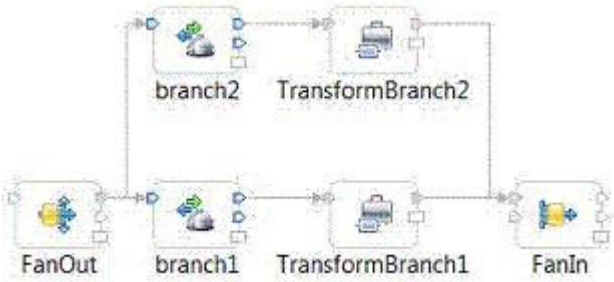


Figure 65. Shared context example

In the case of this example, the SMOs passing through branch 1 and 2, and therefore the Transient and Correlation contexts, are different copies. This means that any changes made to the SMO in one branch are not made to the other branch. The Shared context area of the SMO has been designed for sharing aggregation data between the different branches of the flow inside an aggregation block. Access to the Shared context is only available inside the flow which contains the aggregation block that uses it.

Aggregation mediation programming model special considerations

When using aggregation capabilities consider these points:

- The Shared context is part of the SMO, and is designed for use inside aggregation blocks because the data put into it is available in all the branches of a flow. The design of the Shared context structure is essential to a successful aggregation design. The goal is to store the response from each service invocation into a structure that can then be converted into a single response message after the aggregation block.
- The Shared context is shared across SMOs and should be considered read-only outside the aggregation block.
- At development time it is difficult to determine which SMO body type will be emitted from the Fan In mediation primitive output terminal. Mediation primitives after the Fan In mediation primitive should be used to generate a new message body using the information in the Shared context.
- If a message arrives at a Fan Out mediation primitive, one or more output terminals of the Fan In mediation primitive will fire (assuming no ServiceRuntimeExceptions occur within the aggregation block, for example, as a result of running a Fail mediation primitive), even if a message is not received by the Fan In mediation primitive. Table 3 highlights this behaviour.

  Note: In some circumstances it is possible that none of the messages sent by the Fan Out mediation primitive arrive at the corresponding Fan In mediation primitive. Using Figure 65 on page 82 to illustrate, consider this example: if the second output terminal on primitives branch 1 and branch 2 are fired, this does not cause the messages to arrive at the Fan In mediation primitive, because there is no wired path. This is represented in Table 3 by the message result "No".

Table 3. Which output terminal is fired from the Fan In mediation primitive

Fan In mediation primitive mode	Message arrives at Fan In mediation primitive	Output terminal fired	SMO of Fan In mediation primitive
Iterate	No	Complete	Message body empty; SMO headers and context are the same as the message received at the Fan Out
Iterate	Yes (**in** terminal)	Complete	Last message received at the Fan In mediation primitive input terminal
Iterate	Yes (**stop** terminal)	Incomplete	Last message received at the Fan In mediation primitive input terminal
Simple count/XPath	No	Incomplete	Message body empty; SMO headers and context are the same as the message received at the Fan Out
Simple count/XPath	Yes (**in** terminal)	Incomplete/Complete *	Last message received at the Fan In mediation primitive input terminal
Simple count/XPath	Yes (**stop** terminal)	Incomplete	Last message received at the Fan In mediation primitive input terminal
Iterate/Simple count/XPath	Yes, timeout value expired and decision type not complete	Incomplete	Last message received at the Fan In mediation primitive input terminal
Iterate/Simple count/XPath	Yes, timeout value expired and decision type complete	Complete	Last message received at the Fan In mediation primitive input terminal

* Depends on if the condition (Simple count or XPath) evaluated to **true** for the last message that was received

Aggregation patterns

This section explains aggregation patterns and how they are constructed.

The patterns are based on the aggregation-related **Common Usage Patterns**, as discussed in chapter 2:

- **Aggregation of data from multiple sources**: This pattern is useful if the results from multiple service invocations need to be combined into a single message.
- **Batch processing with message enrichment**: This pattern is useful if the message to be processed contains a repeating element, and each element needs to be enhanced separately.
- **Batch processing requiring embedded aggregation**: This pattern is useful if the scenario is the basic batch processing pattern described in the second pattern, but the enhancement requires information from multiple sources.
- **Nested aggregation:** This pattern is useful if the scenario is the basic batch processing pattern described in the second pattern, but the repeating elements themselves contain repeating elements.

Aggregation of data from multiple sources

This scenario is based on an insurance quote comparison website that involves a web interface that allows users to specify details of their quote.

A business processing engine controls the process and submits a single request to an aggregation mediation, which submits this request to multiple insurance firms. The responses are then aggregated into a single response for the business process to manipulate before displaying the request to the user.

Design the aggregation

The input into the flow is the InsuranceQuote interface, which specifies a Quote business object with the relevant data. The first step in planning the aggregation architecture is to determine if all data required to call the subsequent services are available in the inbound message, or if message enrichment is required. In this scenario, the Quote object contains all the information for both services. If this is not the case, then use a combination of this pattern and the batch processing with message enrichment pattern.

The mediation flow includes a Fan Out mediation primitive (operating in Once mode) receiving the Quote message as input and branching the logic to each insurance back-end service (InsuranceCompanyA and InsuranceCompanyB). Before each Service Invoke mediation primitive, an XSLT mediation primitive converts the body of the message to the correct format, and immediately after the Service Invoke mediation primitive, a Message Element Setter mediation primitive stores the response within the Shared context for later retrieval when building the aggregated response. These two branches are then collected by a Fan In mediation primitive (operating in Count mode). When the second message arrives at the Fan In mediation primitive, its output terminal fires with this message, and then a final XSLT mediation primitive transforms the values within the Shared context into the aggregated response message. This is illustrated in Figure 66 on page 85.

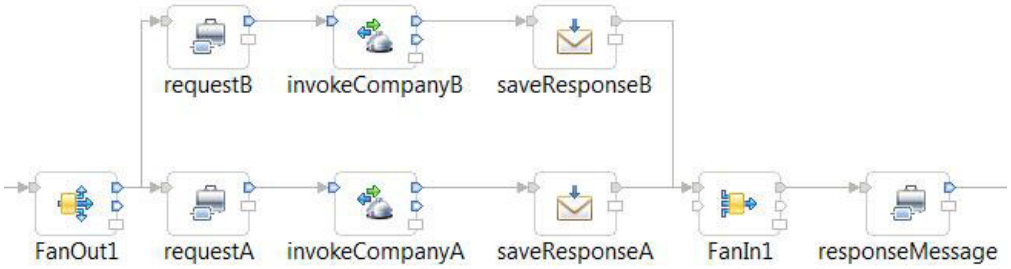


Figure 66. Aggregation Fan Out branching

In this case, the Shared context contains the parts of the response messages that need to be stored to generate the aggregated response, as show in Figure 67.

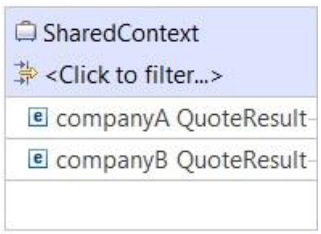


Figure 67. Shared context response messages

This is a simple but powerful usage pattern for aggregation, especially when it is combined into a more complex scenario.

Batch processing with message enrichment

This scenario is based on a store ordering system, where a salesperson can submit a request for a number of orders and have them dispatched. The original message that comes into the mediation flow identifies the customer for each order through a customer ID. The order request message needs to be enriched with each customer's shipping details so that the dispatch step has all the information it needs to succeed. This enrichment is done within the request flow by calling out to a customer information system, implemented using a Service Invoke mediation primitive.

Design the aggregation

The ordering and dispatch systems in this scenario are batch systems, where multiple records arrive within the same request message. However, the customer information system is implemented to receive a single record. After the ESB receives the batch message, each of the occurrences of the repeating element in the request message is processed individually, and then a batch message appropriate to the dispatch system is created. Therefore, aggregation is needed.

To explore how this basic scenario is built, you must first examine the business objects used.

The inbound message into the ordering system from the sales person includes a batched array of SalesOrder objects tagged by a batchID, as shown in Figure 68 on page 86

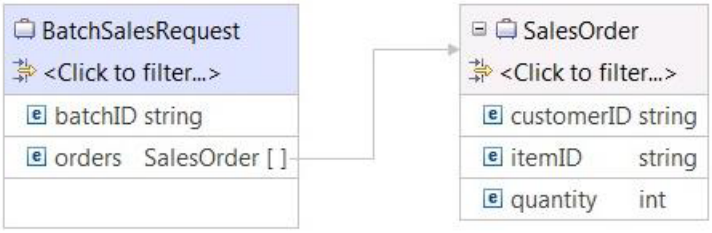


Figure 68. Batch systems business objects

The customerID within each SalesOrder is submitted to the customer service, which provides the **Customer** business object, as in Figure 69.

Figure 69. The Customer business object

An enriched request, including the full customer details, is submitted to the BatchDispatch service with the request structure as shown in Figure 70.

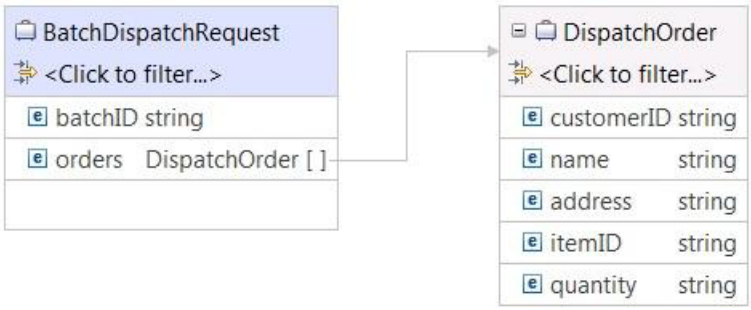


Figure 70. Batch request structure

Figure 71 then shows the request provides the following response structure that will be transformed into the sales response.

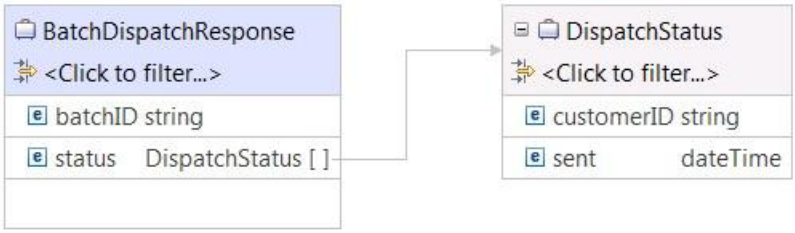


Figure 71. Batch response structure

Shared context

The structure of the Shared context is important in an aggregation design. The goal is to build an array of DispatchOrder objects so that a message can be created to send to the dispatch system. Each DispatchOrder is created from data in the original SalesOrder and is enhanced with further data from the customer information system. Next, an aggregated picture of all of these individual orders is built by moving the orders into an array of DispatchOrder objects as shown in Figure 72.

Figure 72. DispatchOrder array

The current field is used in each iteration of the Fan Out mediation primitive to create the DispatchOrder for the customer in question. This is then appended to the aggregationStore, which can be seen as a running total of the aggregation so far.

The flow detail

Now take a look at the flow in detail, starting with the aggregation mediation primitives, Fan Out and Fan In. The Fan Out mediation primitive is set to Iterate mode for the repeating elements called *SalesOrders*, in the request message. The Fan In mediation primitive setting **Fire output terminal when** is set to the option **the associated Fan Out primitive has iterated through all messages** , which ensures the mediation primitive waits until all messages from the Fan Out mediation primitive have been processed. Figure 73 shows the flow and Figure 74 on page 88 shows the Fan Out and Fan In mediation primitive properties.

Figure 73. Fan Out/Fan In flow detail

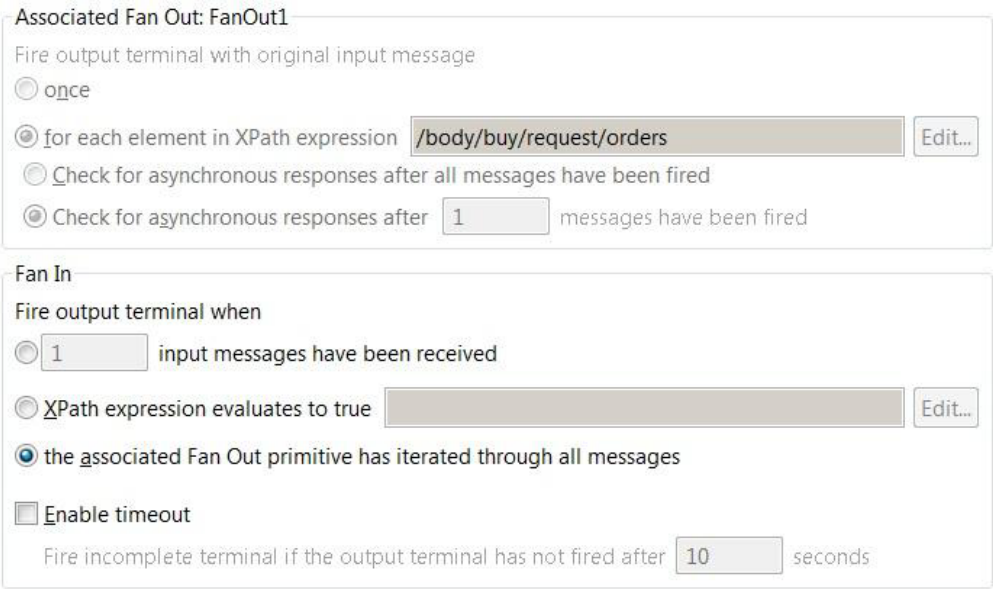


Figure 74. Fan Out/Fan In properties view

A Message Element Setter mediation primitive called *setBatchID* is located after the Fan Out mediation primitive. This Message Element Setter mediation primitive is used to copy the batchID from the request message into the Shared context, as depicted in Figure 75.

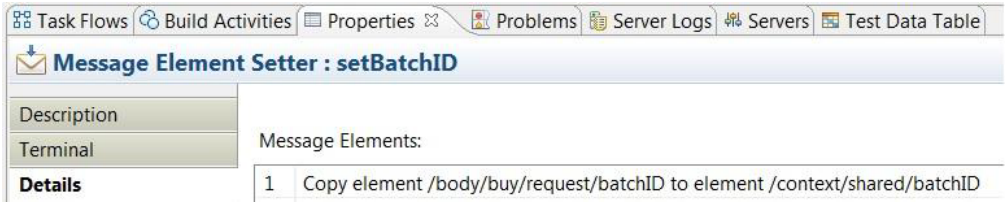


Figure 75. Message Element Setter properties

Notice that in this example, the current iteration's output is built from the customer details returned from the service invocation. Additional information (itemID and the quantity) has been copied from the current element iteration's occurrence in the Fan Out context, as shown in Figure 76 on page 89.

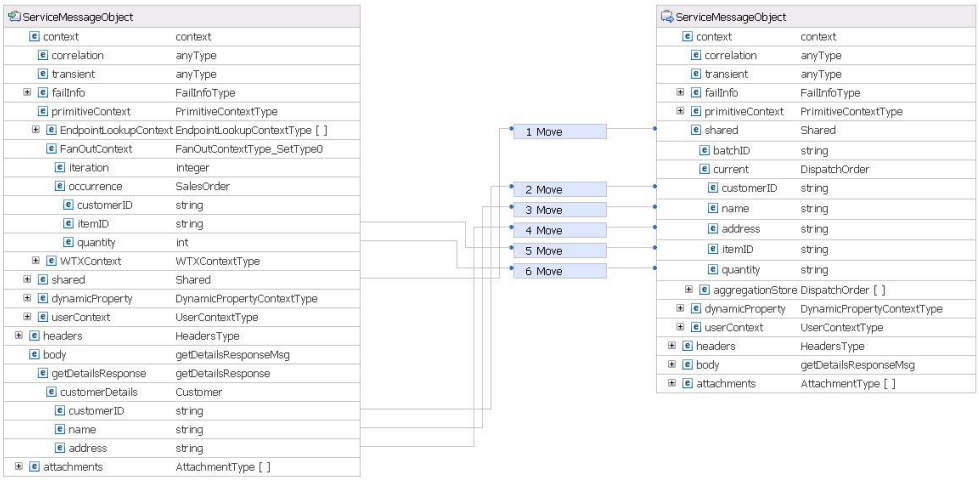


Figure 76. Transform inline map

The complete results from the current aggregation iteration in the current object of Shared context now show. The next mediation primitive in the flow is a Message Element Setter mediation primitive called aggregateResult, which appends the current object into the aggregationStore in Shared context, as depicted in Figure 77.

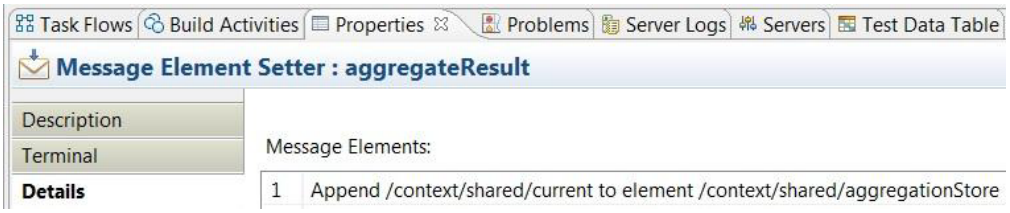


Figure 77. Message Element Setter aggregateResult properties

The aggregation flow used for each element is now complete. The elements aggregationStore and batchID must be converted into the batchDispatchRequest needed for the Callout node, as shown in Figure 78.

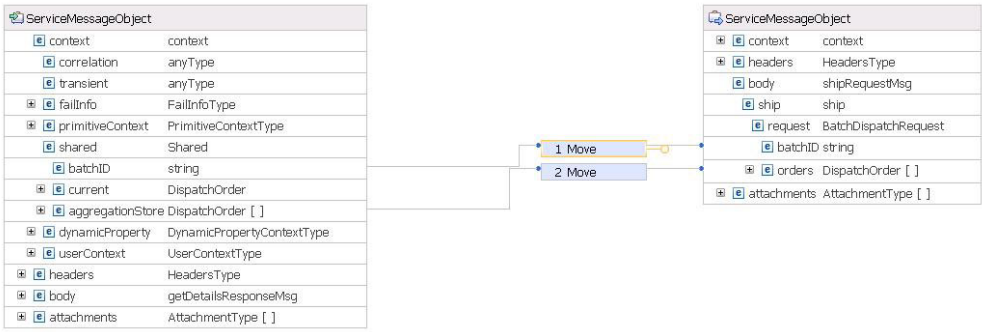


Figure 78. SMO mapping

The key to a successful aggregation flow is to build up the required information in each iteration in a current object, and then to attach current objects into the running total aggregationStore in the Shared context.

Batch processing requiring embedded aggregation

This scenario combines the two previously described examples, where an aggregation of data from multiple sources is embedded within a batch processing aggregation.

The scenario is a personnel-based application that creates payslip information for specific employees. The back-end service is represented by a Callout node and requires full employee records, rather than the employee IDs, which arrive at the mediation flow. These employee records are created by the mediation from two sources, an HR system and a Payroll system. The former stores information concerning the employee's name, address, and vacation entitlement, while the latter contains information regarding salary and department.

Design the aggregation

The input to the flow is an array of employee identifiers, while the output is an array of status objects, indicating whether the payslip processing succeeded or not, as shown in Figure 79

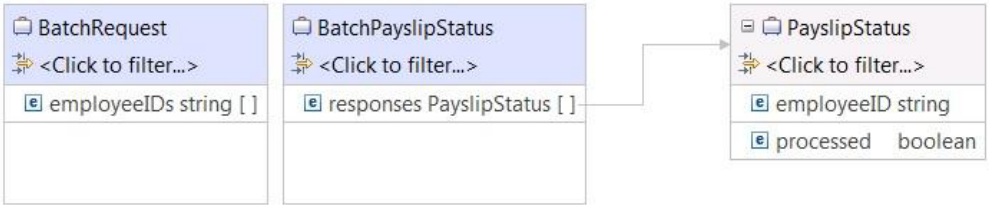


Figure 79. Flow input and output

The HR and Payroll systems that are called from within the mediation flow can only handle a single record, as shown in Figure 80.

Figure 80. HR and Payroll Record

The outer aggregation iterates through all the employeeIDs, while the inner aggregation builds a response aggregated from the HR and Payroll services. This is depicted in Figure 81.

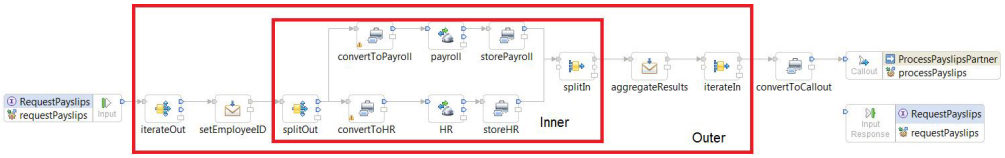


Figure 81. The inner and outer flow

In this case, after the two services have been called successfully, the HRRecord and PayrollRecord must be combined into a single EmployeeRecord. The EmployeeRecord must then be appended to an array of records to comply with the interface for the Payslip service, which is called via the Callout node, as in Figure 82. Consequently, the same model for Shared context as previously described is used.

Figure 82. EmployeeRecord array

After the data has been aggregated and batched, the Payslip service is called, which takes an array of EmployeeRecord objects and returns an array of PayslipStatus objects, as shown in Figure 83.

Figure 83. PayslipStatus object

Nested aggregation

This scenario is based on a supermarket company, where each outlet sends multiple restocking orders to a central company-wide system that batches the requests into a single message. This single message is submitted to the mediation flow where, for each order and for each branch, a separate invocation to the order system should occur.

Design the aggregation

The ordering system only accepts a single record, whereas the input into the mediation module is a nested batch message. This introduces the need for aggregation, and in this case nested aggregation, because the batch request message is repeated on an outlet basis, and then repeated on an order basis. Orders are sent individually; the response from each order is stored into a single response message, which can be aggregated later. This single response message shows the status of all the elements in the batch request. To explore how this basic scenario is built, first examine the business objects used, as shown in Figure 84 on page 92.

Figure 84. Business objects for batch order example

The GroupsOrders request is a batched array of Outlets tagged by an ID, and Outlets contains a batched array of Orders. Figure 85 shows the result.

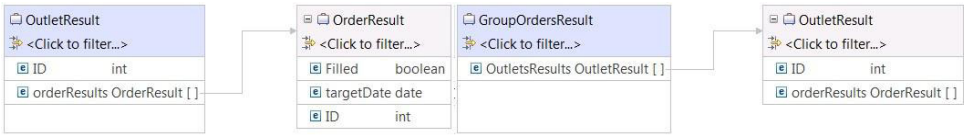


Figure 85. Result of batched order example

Shared context

The structure of the Shared context is essential to a successful aggregation design, and it is even more critical within a nested aggregation. The goal is to store the response from each Order into a structure that can then be converted into a single response message. The initial request is at a company-wide scope, the first aggregation is at the Outlet scope, and the second aggregation is at the Order scope, as depicted in Figure 86.

Figure 86. Aggregation levels

Understanding the aggregation structure helps you define the correct Shared context. The logic pattern within the mediation flow is:

- Aggregate on an Outlet basis.
  - Aggregate on an Order basis.
    - Call the Order service.
    - Store the current Order service within the current Outlet response.
  - Repeat for each Order associated with the Outlet.
  - Store the current Outlet in the company-wide aggregation store.
- Repeat for each Outlet.
- Convert all results into a response message.

With this pattern you can store the overall company-wide response, the current Outlet response, and the current Order response in the Shared context, as shown in Figure 87 on page 93.

Figure 87. Shared outlet responses

The flow detail

Now look at the flow in detail. It is separated into three sections:

- Company-wide scope
- Outlet or Store scope
- Order scope

This is shown in Figure 88.

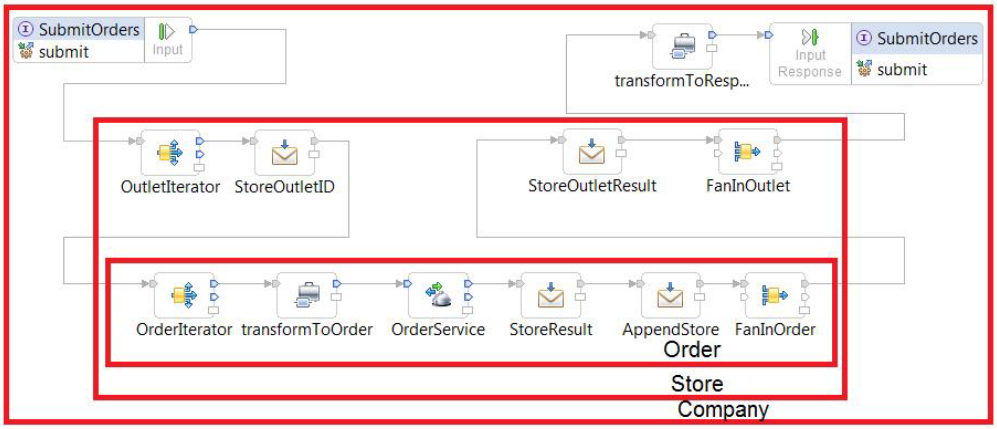


Figure 88. Aggregation block

Error handling within an aggregation block

To handle an error condition within an aggregation block a Stop or Fail mediation primitive can be used. Alternatively, you can wire to the Stop input terminal of the Fan In mediation primitive.

- The Stop mediation primitive stops the current branch of execution and the aggregation will continue to either another branch or iteration.
- The Fail mediation primitive terminates the entire mediation flow and return a ServiceRuntimeException to the caller of the mediation. In general, the Fail mediation primitive should not be used within an aggregation, unless you want to terminate the processing of the entire flow immediately.
- The Fan In mediation primitive provides a Stop input terminal that can be wired to signal that the aggregation should be terminated. When a message arrives at the Stop terminal the incomplete output terminal is fired. Additional mediation primitives can then be wired to handle this situation and if required this could include a Fail mediation primitive.

The Fan In mediation primitive includes a property that specifies the timeout value. The timeout period starts when the associated Fan Out mediation primitive fires an output terminal for the first time. If a message arrives at the Fan In mediation primitive **in** terminal after this timeout period, it is considered late, and the incomplete terminal is fired. The default value for the timeout property is `-1`, which means there is no timeout, and no messages are considered late.

Asynchronous parallel processing

To reduce the overall processing time of an aggregation you can call multiple services concurrently, instead of processing each service in series.

A branched aggregation block such as shown in Figure 89 will not, by default, be processed in parallel.

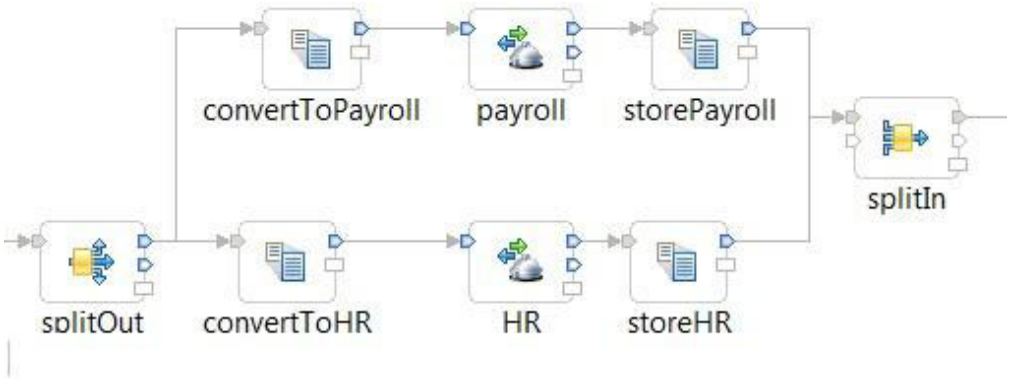


Figure 89. Branched aggregation block

To enable multiple service invocations to occur before the processing of response messages you must complete the following steps:

1. Specify the Invocation style property on each Service Invoke mediation primitive to be **Async**, as shown in Figure 90.

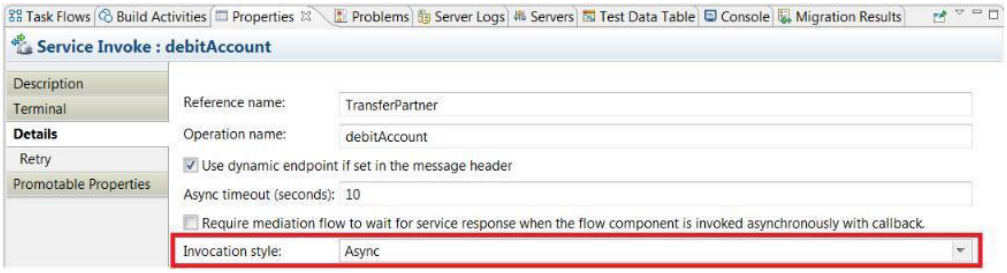


Figure 90. Service invoke mediation primitive property details

2. If the mediation flow component is configured to run under a Global Transaction, ensure that the component Reference that is associated with the Service Invoke mediation primitive has a qualifier for Asynchronous invocation that is set to **Call**, as shown in Figure 91 on page 95.

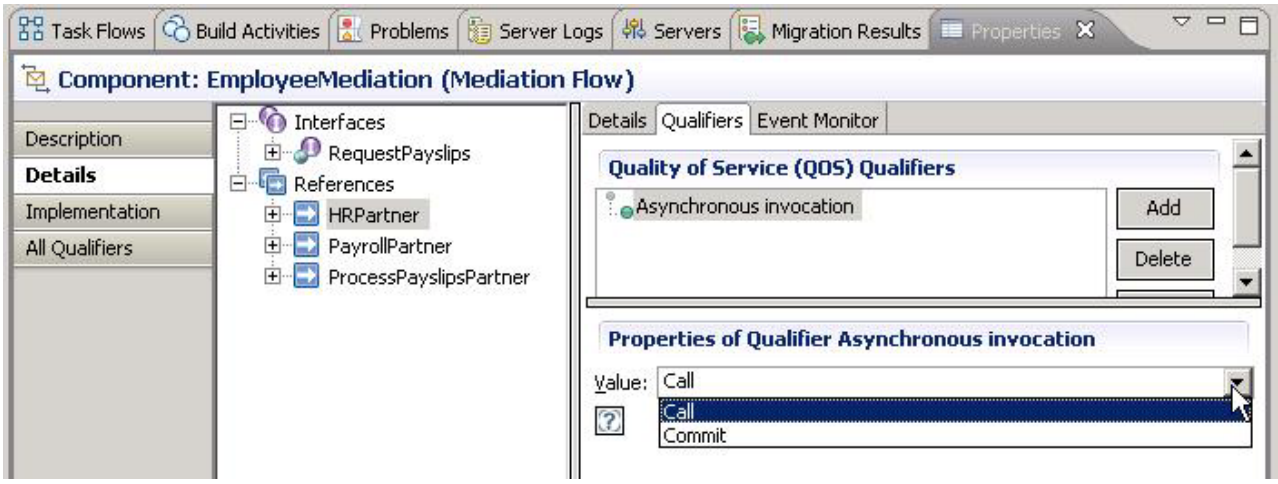


Figure 91. Setting the Asynchronous invocation qualifier

If a global Transaction is configured, and the Asynchronous invocation setting is **Commit**, then request messages are never sent, because the service request cannot be sent until the transaction commits.

3. An additional step of configuration is required when the aggregation block processes in Iterate mode. Using the Fan Out mediation primitive you can specify the number of concurrent unprocessed requests that can be made before the response messages are processed. By specifying an integer value in the **Check for asynchronous responses after** `X` **messages have been fired** field, as shown in Figure 92, the runtime completes the processing of `X` requests and then processes the response messages before further iterations are processed. This way you can restrict the number of parallel requests being processed by back-end services, such as the Payroll and HR services. To process all entries within the array before the handling of any response messages, select **Check for asynchronous responses after all messages have been fired**.

Figure 92. Setting the number of requests that should be processed

By using the Invocation style and Asynchronous invocation settings both Payroll and HR services are called prior to either of the response messages being processed.

Performance considerations when using aggregation

When using aggregation, there are some performance considerations to be aware of.

Aggregation patterns often require multiple requests to be built from a single input request and multiple responses to be merged into a single response. In the examples shown in previous Aggregation-related sections this is done using BOMapper mediation primitives. If there are no complex operations required in

these transformations, and the data is simply being moved from one part of the message to another, then the performance will be significantly improved by using BOMapper mediation primitives instead of XSL Transformation (XSLT) mediation primitives. This is even more relevant for larger messages. If you have complex operations, only use XSLT mediation primitives. The Data Transformation section discusses these mediation primitives in more detail.

Batch processing considerations

In the batch processing scenario it is important to consider the impact that processing large batched requests may have on other applications hosted on the same server. Processing requests with a large number of repeating elements; for example, a large number of sales orders, can use a significant amount of system memory, which will remain in use until all processing is complete. Depending on the response times of the services being called within the aggregation block processing may slow down system resources for long periods. If the same server used for processing batch requests is also being used to host applications that need to respond quickly to requests, you can see a degradation in service while the batched request is being processed. You can avoid this issue by processing large batched requests on a separate server or by processing the requests during periods of low system usage.

For the reasons explained before, it is important to minimize the amount of memory used when processing large batched request messages. When using the batched processing pattern, the body of the original request message is rarely required inside the aggregation block because the Fan Out mediation primitive stores the current instance of a repeating element in the Fan Out context. In this example it is the sales order that is being processed. At the end of the aggregation block the response data is held in the Shared context and it is used to build a new response message body. If the request is large, then significant savings in memory usage and reductions in processing time can be achieved by deleting the message body immediately after it has been processed in the Fan Out mediation primitive, by using the Message Element Setter mediation primitive, as shown in Figure 93.

Figure 93. Deleting the message body

Other memory considerations

After an aggregation block is completed the responses that have been stored in the Shared context are combined into a new response message body. This is done immediately after the Fan In mediation primitive, as shown in the BOMapper in Figure 66 on page 85 and in Figure 73 on page 87.

At this point the contents of the Shared context are not typically required and performance is improved if Shared context is not propagated through the remainder of the flow. If you do not require any of the fields in the context then leave the context unmapped. Otherwise, expand the Context element and map only the parts that are required in the rest of the mediation flow, as shown in Figure 94.

Figure 94. Aggregation context mapping

Chapter 10. Error handling in the mediation flow component

This topic provides information on the various ways to deal with errors in the mediation flow, including ways to use the Stop and Fail mediation primitives, where to look for fail information in the message, and how to handle modelled and unmodeled faults.

Fail terminal

Mediation primitives that process messages have a fail terminal. When there is a failure in the mediation primitive, the fail terminal is called and exception information is sent, along with the input message. The exception information is stored in the failInfo element in the message context. A message showing the failInfo is shown in Figure 95. You must wire the fail terminal of a mediation primitive to another mediation primitive in order to access the failInfo.

Figure 95. The failInfo element in the message context

Runtime failure in the mediation primitive

When a failure occurs in a mediation primitive, the unwired fail terminal sends a runtime exception, and the flow is considered to have failed. If the mediation flow component is running under a global transaction (default), primitives that use resources that participate in the global transaction will rollback.

© Copyright IBM Corp. 2007, 2011

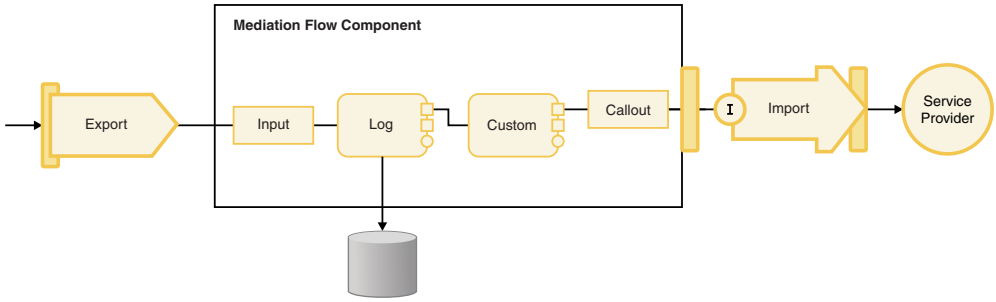


Figure 96. Runtime failure and rollback transaction

In Figure 96, assume that a global transaction is not configured, and a failure occurs in the Custom mediation primitive. As the fail terminal has not been wired, the flow fails. The log function would still complete as the mediation flow component is not under a global transaction, and therefore still logs to the database.

When the flow is reconfigured, a global transaction is configured, and a failure occurs again in the Custom mediation primitive, the flow fails. If the Log mediation primitive has its transaction mode property set to *same*, the log entry will be rolled back as a part of the global transaction, and no entry will occur in the database. If the transaction mode property was set to *new*, then the log to the database commits outside the global transaction so an entry will still occur in the database.

Modelled faults

WSDL operations have three types of messages:

1. input
2. output
3. fault

Modelled faults are business error conditions, that are defined as a fault message in a WSDL operation. In the mediation flow component, modelled faults are handled through Input Fault and Callout Fault nodes, which have a terminal for each unique type of fault message. The Input and Callout Fault nodes are created in the mediation flow editor when there is a modelled fault defined in the source or target operation.

- An **Input Fault node** is created when a source operation has a modelled fault message defined. The Input Fault node has an input terminal for each fault message type that is defined in the source operation. Any message that is sent to an Input Fault node will result in a modelled fault error message being returned from the source operation. Wiring to this node means that the fault message is returned to the service requester. The Input Fault node is created in the request and response flows.
- A **Callout Fault node** is created in the response flow for each target operation that has a modelled fault defined. The Callout Fault node has an out terminal for each fault message type that is defined in the target operation. When a modelled fault occurs, the Callout Fault node sends a message to the mediation primitive or node to which it is wired.

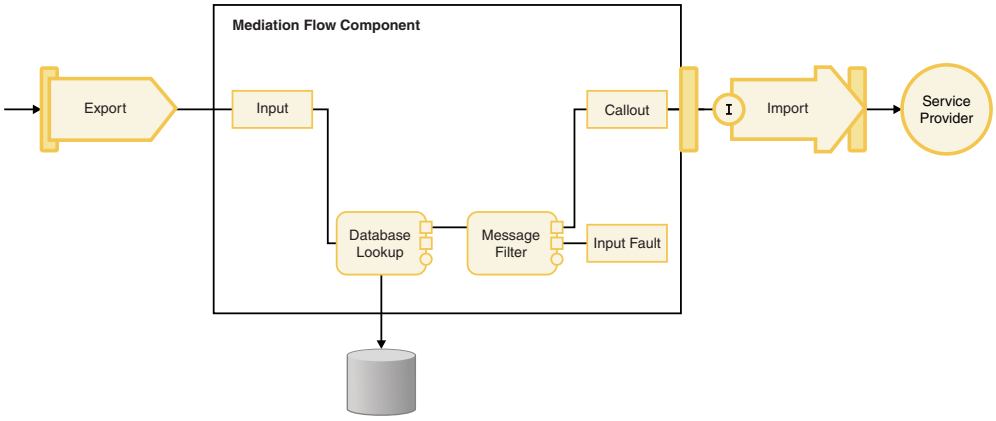


Figure 97. Example of a modelled fault

As shown in Figure 97, an example of when modelled faults can be used within the mediation flow component, is if a fictitious company offers a tiered level of service. The company offers priority booking information about upcoming theatre shows to Gold customers, who pay a premium. Standard customers must manually check the website for any information. An input message can be sent to a Database Lookup mediation primitive that retrieves the membershipType and sets an isValid property in the message. The message then goes onto a Message Filter mediation primitive, to route the message based on the value of isValid. If the membershipType is valid (Gold), the message is sent to the target operation. If the membershipType is not valid, the message is returned to the service requester, through the Input Fault node.

Unmodeled faults

Errors that are returned, and are not defined as WSDL faults are called unmodeled faults. There is no Input or Callout Fault node created for these types of faults in the mediation flow editor. In this case, the input message type is sent to the Callout Response node's fail terminal. The failure information is captured in the failInfo element of the message context.

To handle an unmodeled fault, you can wire the fail terminal of a Callout Response node to a mediation primitive. For example, you could wire the fail terminal on a Callout Response node to a Message Logger mediation primitive to log all unmodeled faults. You can set a property on the Callout Response node to determine whether the entire request message or just the message header information should be passed out of the fail terminal (there can be a performance overhead if you choose to pass the entire request message).

If the fail terminal on the Callout Response node is not wired and an unmodeled fault is received, a mediation runtime exception will occur.

Stop mediation primitive

The Stop mediation primitive has one input terminal and no out terminals. When the fail or out terminal on a mediation primitive is wired to the Stop mediation primitive, that particular branch comes to an end.

The mediation flow editor generates warnings if the out terminal of a mediation primitive is not wired. The Stop mediation primitive is used to stop these warnings. In the runtime, out terminals that are not wired are automatically propagated to stop.

In Figure 98, the fail terminal of the Message Filter mediation primitive is wired to a Message Logger, so that the failed messages can be stored. To indicate that no more actions are needed, the Message Logger is wired to a Stop mediation primitive.

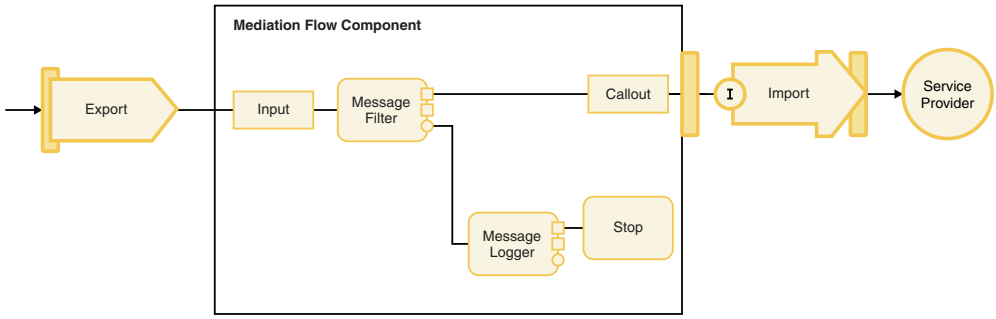


Figure 98. Stop mediation primitive

Fail mediation primitive

The Fail mediation primitive can be used if you want to stop the flow and send a runtime exception. You can wire the out or fail terminal on a mediation primitive, to the Fail mediation primitive. If the mediation flow component is running under a global transaction, mediation primitives participating in the global transaction will rollback.

Faults in the Service Invoke mediation primitive

There are two types of fault that can occur at the Service Invoke mediation primitive:

- Modelled faults, which are sent through the out terminal.
- Unmodeled faults, which are sent through the fail terminal.

Error flow

A mediation flow component has an error flow for each source operation. The error flow acts as a catch-all for messages that are propagated from any unwired fail terminal on any primitive or node, in either a request or a response flow.

By default, an error flow consists of:

- An Error Input node, which has a catchAll terminal, with the type set to anyType. The Error Input node propagates the Service Message Object (SMO) from the unwired terminal, that contains the error information.
- An Input Response node for (request and response) operations. You can use this node to return a message from the source operation.
- An Input Fault node, created when a source operation has a WSDL fault message defined. The Input Fault node has an input terminal for each fault

message type that is defined in the source operation. Any message that is sent to an Input Fault node results in a WSDL fault error message being returned from the source operation.

You can wire mediation primitives to the Error Input node to capture error information. For example, a Message Logger mediation primitive is wired to log the SMO. You can also put your error handling logic in a subflow, so that it can be reused in multiple error flows.

You can use an error flow to audit any unhandled errors that may occur in the operation request or response flows. For example, as shown in Figure 99, you can use a Message Logger mediation primitive to capture the SMO, and then wire the Log mediation primitive to a Fail primitive.

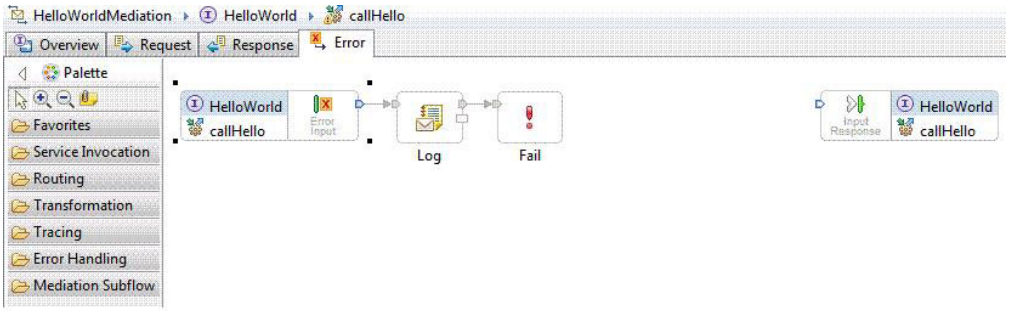


Figure 99. An example error flow to return an unmodeled fault

An error flow can use the information available in the failInfo element of the SMO and return a modelled fault message. To do this, within the error flow, you can map the failureString from the failInfo to the modelled fault message and wire it to the Input Fault node. This is shown in Figure 100.

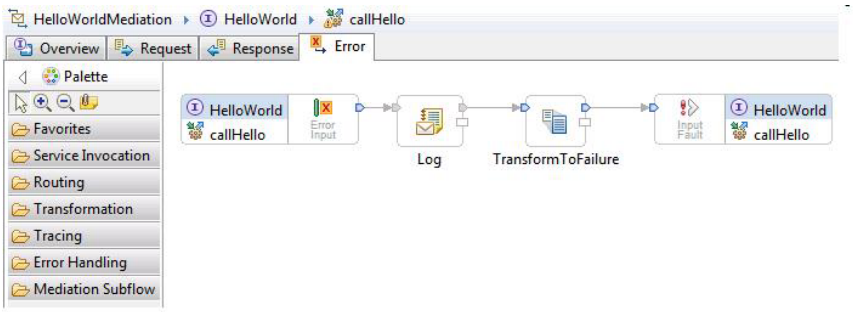


Figure 100. An example error flow to return a modelled fault

Common patterns of usage for error handling

When a service component is called synchronously or asynchronously unexpected runtime errors can occur. In SCA, error handling is put in place to make sure that a message is not lost. This topic provides information on how messages are handled by the SCA runtime when an error occurs. You will find information for both synchronous and asynchronous invocation styles.

Synchronous invocation

A transaction is a unit of activity, within which multiple updates to resources can be made dependent such that all or none of the updates are made permanent. Transaction qualifiers can be used to control transactional behaviour of service invocations.

Note: In this section, we have assumed that all components are set to run in a global transaction.

When a service component is called synchronously, and performs a synchronous invocation, both the service requester and the service provider run in the same thread of execution. The calling component within WebSphere ESB is blocked until a response is received from the provider.

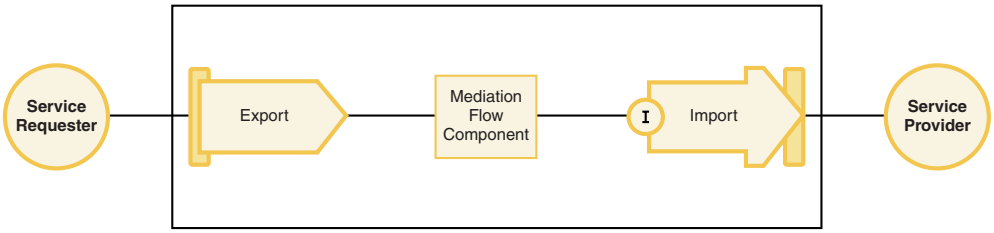


Figure 101. Synchronous invocation error handling

As shown in Figure 101 there are no queues in a synchronous flow. If an unhandled error occurs at any point in the flow, any active transactions will rollback and an unmodeled fault is returned to the service requester.

Asynchronous invocation

When a service component is called asynchronously, the service requester and the service provider run in different threads of execution. The SCA runtime returns control to the requester immediately.

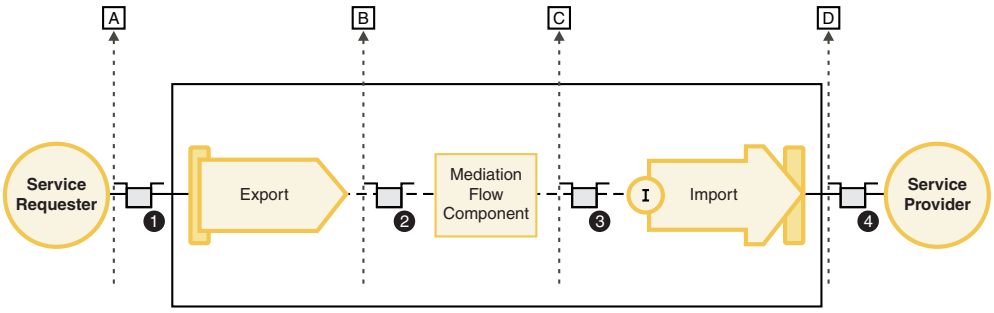


Figure 102. Asynchronous invocation error handling

Figure 102 shows how errors can be handled within an asynchronous flow. The flow has been split into transaction boundaries (A, B, C, D), each new boundary is marked by a new queue (1, 2, 3, 4). If an error occurs within a transaction boundary, the message is rolled back to the queue at the start of the transaction. For example, if an error occurs within the mediation flow component, the message is rolled back to queue number 2, because it is within the transaction boundary B.

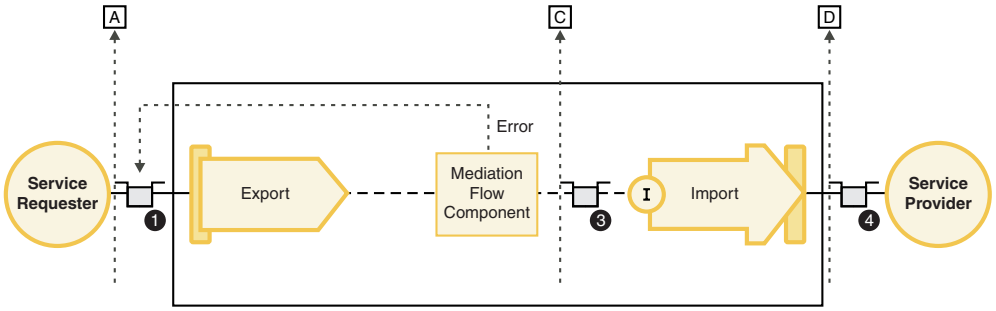


Figure 103. Asynchronous invocation error handling without queue 2

By default, to improve performance, when the first asynchronous invocation is made, queue 2 is omitted from the flow. As shown in Figure 103, one thread of execution will start from queue 1. If an exception occurs at the mediation flow component, the message will be sent back to queue 1, because it is within transaction boundary A. The message is redelivered to the export and the previously omitted queue 2 is reintroduced as shown in Figure 104. Any subsequent retries, will include queue 2. This performance optimization occurs automatically. Specifically JMS, MQ and asynchronous SCA bindings all have this optimized behaviour.

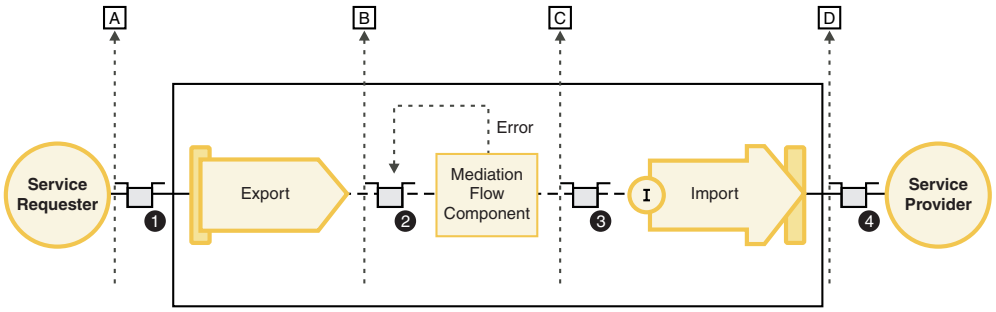


Figure 104. Asynchronous invocation error handling

As a default, when a message is rolled back to an internal module destination, SCA retry logic indicates that the message will be redelivered up to five times. If the message fails again, it will be automatically forwarded to the defined exception destination. This can be viewed via the Service Integration Bus Explorer in the Administrative Console. If the Failed Event Manager is enabled, messages on the exception destination are removed and stored in a database as failed events. Each module has its own internal exception queue. Binding retries can be configured at the binding, or on the messaging provider. Binding retries occur at least once, so that the Failed Event Manager can be aware of any fault.

The Failed Event Manager is a web-based client for working with and resubmitting the failed invocations. It is an integration application and is available in the Administrative console. It displays the number of failed events and provides a number of search capabilities. Messages that exceed their retry count are automatically retrieved by the Failed Event Manager. Asynchronous export and import bindings can be configured to send failed messages to the Failed Event Manager before they exceed their retry count.

Mixed invocation styles

When a service requester calls a service provider, the invocation can be made either synchronously or asynchronously. There are situations where the service requester makes a synchronous call, but the service provider must be called asynchronously by the mediation flow component. In this occasion, queues 1 and 2, and boundaries A and B will not exist, and the flow will look similar to Figure 105. In this situation, if an error occurs before boundary C, it will be sent back to the service requester.

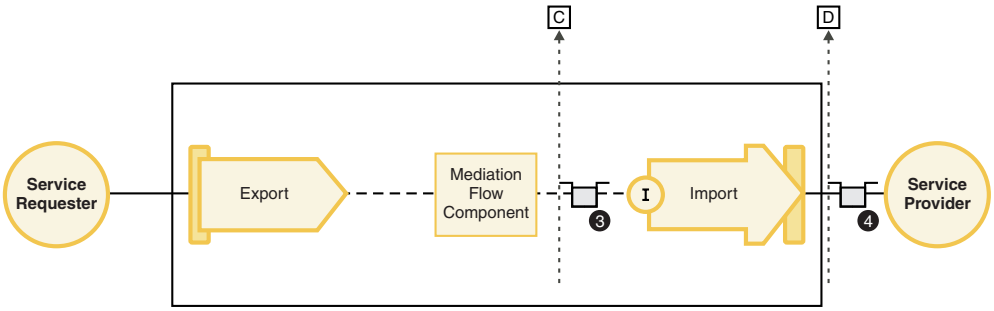


Figure 105. Synchronous and asynchronous invocation error handling

If the service requester performs an asynchronous invocation, and the service provider is a synchronous service, the mediation flow component will call the import synchronously. In this situation, queues 3 and 4, and boundaries C and D will not exist within Figure 106. In this situation, if an error occurs after boundary B, it will be sent back to the service provider.

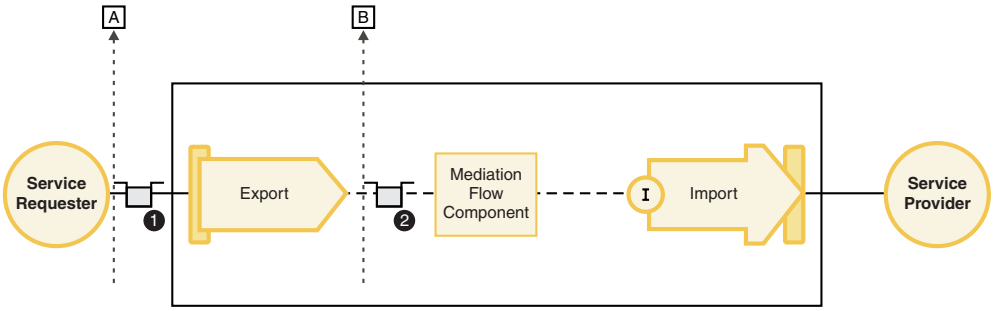


Figure 106. Asynchronous and synchronous invocation error handling

Chapter 11. Testing and Debugging

This topic provides information on tools in Integration Designer. The tools can be used to test modules and components, and to debug problems.

Unit testing

Modules are typically developed independently of other modules. Within a module, a component does not need or have visibility to another component's implementation. In Integration Designer, the integration test client is the designated tool for testing modules and components in isolation. The test client features a sophisticated user interface that enables you to easily manage and precisely control your tests.

Testing components

As you develop a component, you typically want to unit test, fix any problems, and unit test the changes again. In the integration test client, you can choose the operation that you want to test, enter values, and invoke the operation to see if the results are as you expected. You can also choose to emulate components and imports, which means that you do not actually need to run them during your testing. A manual emulation event is automatically generated in the following situations:

- You have chosen to test an entire module and the integration test client has encountered a component that has not been implemented.
- You have chosen to test a set of components or a single component in a module, which means that any other unselected components in the module are automatically emulated regardless of whether they are implemented or not.

A manual emulate event is an interactive event that requires you to enter input and output parameter values, or select an exception to throw.

Example

The component HelloWorldMediation calls an external Web service, which takes a string input and returns the concatenated string. To test the operation callHello independently of the Web service, you can use an emulate event.

1. From the module assembly diagram, call the **Test Component in Isolation**.
2. In the integration test client, select an interface and an operation that you want to test.

© Copyright IBM Corp. 2007, 2011

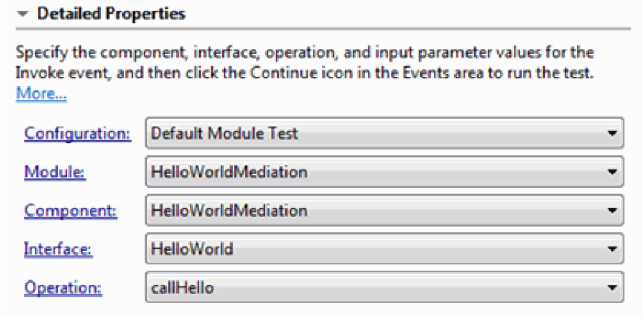


Figure 107. Integration Test Client properties

3. Fill in the input parameter values for the selected operation.

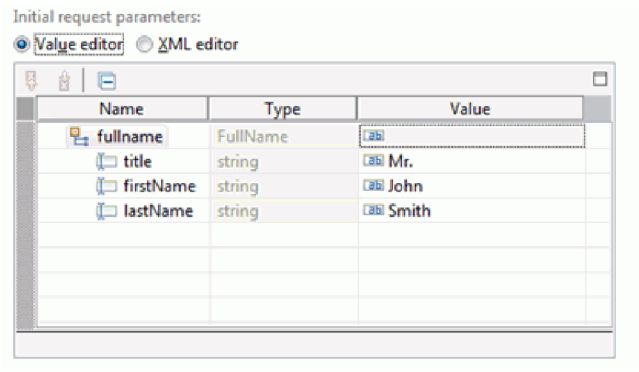
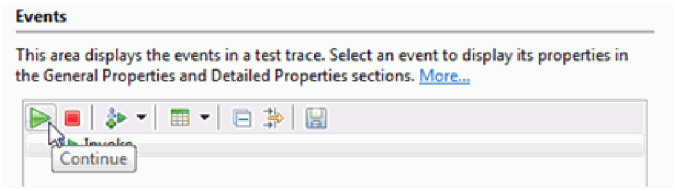
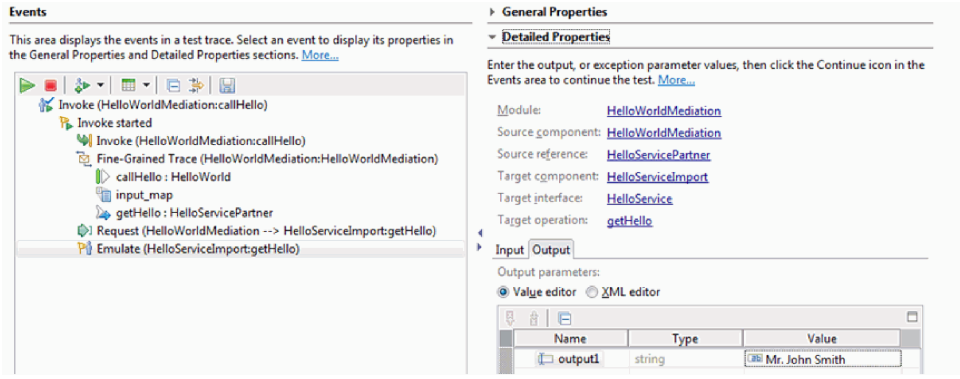


Figure 108. Input parameters

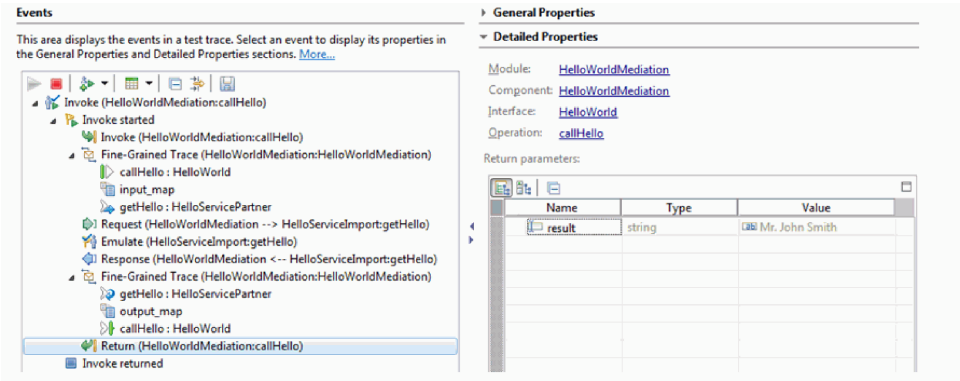
4. Click the **Continue** button to begin invoking the selected operation with the specified parameter values.



5. If the selected component is referenced to another component or Import, then the integration test client will pause at the point where the reference is used. It also allows you to emulate the results returned by the reference, so that you can test your component even if the dependent component is not implemented.



6. Once the values for emulation are specified, click the Continue button to continue.
7. If everything goes well, then the expected result is returned.



Cross-component trace

SOA applications tend to be large and complex, and are typically composed of other applications. The most time spent in troubleshooting an SOA application is often in narrowing the problem to a particular area. This is where cross-component trace is one of your most useful tools.

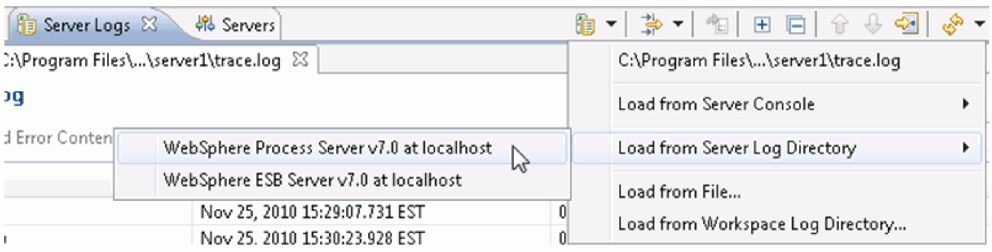
The Server Logs view displays standard server console and log records. However, if you enable cross-component tracing, the Server Logs view also displays invocation records that contain the invocation data that was passed between the components in your application. The invocation records are displayed in hierarchical format in the Server Logs view, which enables you to more easily understand the relationships that exist between the records. When you enable cross-component tracing, the Server Logs view becomes an even more powerful tool for problem determination. Typically, the process of using the Server Logs view for problem determination begins when you encounter problems when testing your application using the integration test client or another tool.

To turn on cross-component trace:

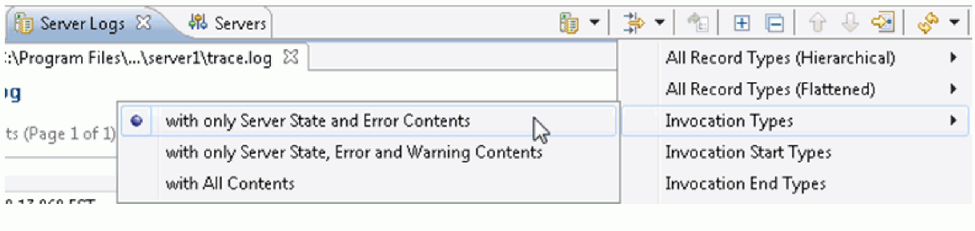
1. Bring up administrative console on a running server by right-clicking the server and from the menu select Run administrative console.
2. Log on to the console, expand Troubleshooting and click Cross-Component Trace. Select your server. For the Configuration values, select Enable with data snapshot as this collects the most data, which is the recommended setting for tracing.
3. Save the configuration.
4. Log out of the administrative console and restart your server so the changes take effect.

To produce the cross-component trace, re-run your test.

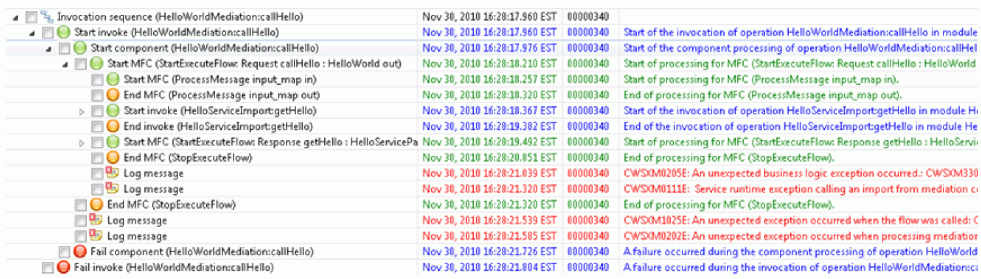
To view the cross-component trace, open the Server Logs view selecting Window > Show View > Server Logs from menu bar. In the Server Logs view, you could display the traces either from a live server console or from a trace file. To display the trace, select “Load from Server Log Directory” and “trace.log” from the log directory.



You can filter the log message to locate the invocation event by selecting Invocation Types > with only Server State and Error Contents



By expanding the invocation event, you can analyze the path or "call stack" of the test message and quickly identify the failing application.



The Server Logs view provides powerful filtering and searching capabilities to help you locate useful information from a large number of log records. Detailed information about working with server console and log records in the Server Logs view is found in the Integration Designer topic F"Using the Server Logs view for problem determination".

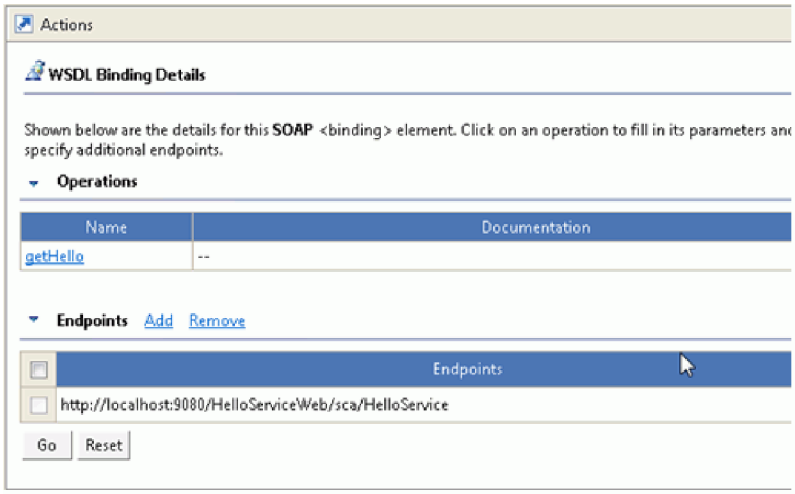
Note that turning on the cross component trace has a performance impact on the server. When the tracing is completed, you can turn off the cross component trace via the administrative console.

Testing Web services

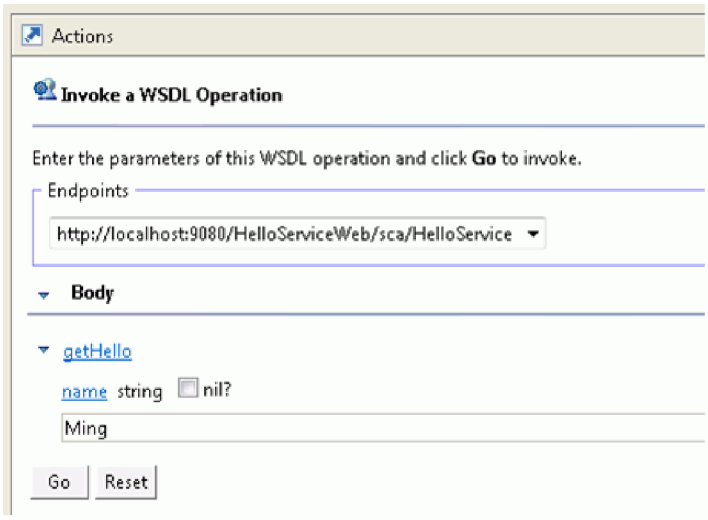
You can use the Web Services Explorer to test Web services.

To test a Web service using Web Services Explorer:

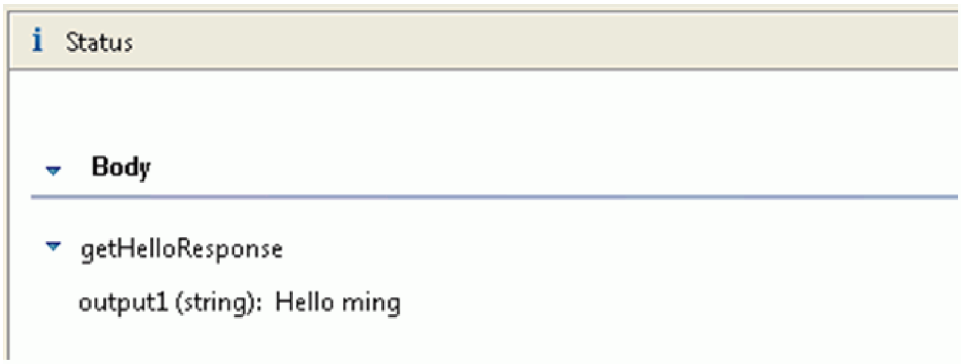
1. Select the interface that the Import of the SOA application is bound to. Select **Web Service > Test with Web Service Explorer** from the context menu.
2. Enter the end points in the WSDL binding. The end point information can be retrieved from the properties of the Import.



3. After the end point is defined, you can click the operation and fill in a test message.



4. Clicking the "Go" button will instruct the Web Services Explorer to send out the SOAP message. The response message will be displayed in the status pane.



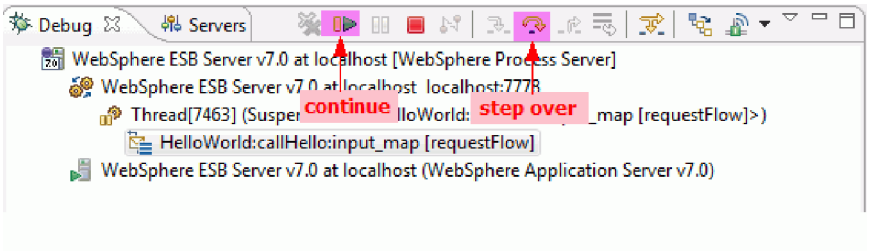
The Web Services Explorer also supports HTTPS. The SSL certificates are managed by the JRE.

Debugging mediation flows

The integration debugger allows you to debug your mediation flow component using a graphical user interface. You can control the flow execution by using breakpoints to step over the running of mediation primitives. When the server is running in debug mode, and a breakpoint is encountered, the running stops. When a component thread pauses at a breakpoint, you can control program flow or alter the state of the running component. You can use the variables view to inspect the data or message.

To debug a mediation flow, perform the following steps:

1. Start the server in debug mode by selecting **Debug** from the server's context menu.
2. Add the projects that you want to debug to the server.
3. Open the mediation flow that you want to debug.
4. Select a mediation primitive that you want the flow execution to pause at, right-click and select **Debug -> Add Breakpoint**.
5. If you want to debug within a subflow, you must open the subflow and set a breakpoint on a mediation primitive in the subflow.
6. Open the assembly diagram and launch the integration test client to test the mediation flow component.
7. Enter the test data in value editor of the integration test client.
8. Click on the Continue button to start executing the test.
9. The flow execution will pause at the first breakpoint. You can inspect the message coming in to the mediation primitive in the Variables view.
10. When you have inspected the message, you can then step over the primitive or continue the flow execution until it hits the next breakpoint.



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.

© Copyright IBM Corp. 2007, 2011

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.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of WebSphere ESB.

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, ibm.com®, are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" www.ibm.com/legal/copytrade.shtml. Other product and service names might be trademarks of IBM or other companies.

This product includes software developed by the Eclipse Project (http://www.eclipse.org/).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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.

© Copyright IBM Corp. 2007, 2011

SC34-7206-03