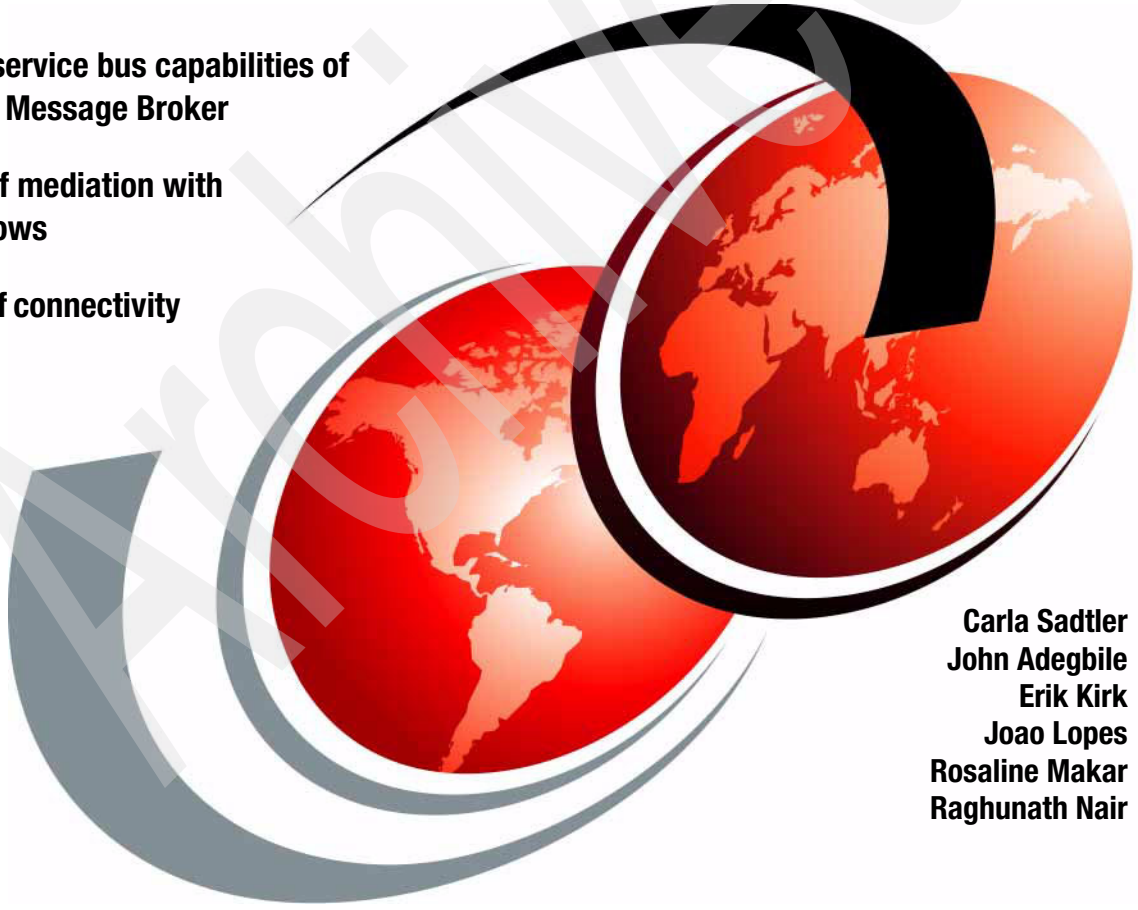
WebSphere. software

IBM

Using IBM WebSphere Message Broker as an ESB with WebSphere Process Server

- Enterprise service bus capabilities of WebSphere Message Broker
- Examples of mediation with message flows
- Examples of connectivity options



Carla Sadtler
John Adegbile
Erik Kirk
Joao Lopes
Rosaline Makar
Raghunath Nair

Archived

ibm.com/redbooks

Redbooks

Archived

International Technical Support Organization

Using IBM WebSphere Message Broker as an ESB with WebSphere Process Server

January 2008

Archived

SG24-7527-00

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

Archived

First Edition (January 2008)

This edition applies to IBM WebSphere Process Server Version 6.0.2 and WebSphere Message Broker Version 6.

© Copyright International Business Machines Corporation 2008. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices ix
Trademarks x

Preface xi
The team that wrote this book xi
Become a published author xiii
Comments welcome xiii

Part 1. WebSphere Message Broker as an ESB 1

Chapter 1. The IBM SOA Foundation and products in this book 3
1.1 IBM SOA Foundation 4
1.2 WebSphere Process Server 5
1.3 The enterprise service bus 6
1.3.1 The role of an enterprise service bus 6
1.3.2 WebSphere Message Broker V6 8
1.4 The focus of this book 10

Chapter 2. Business processes with WebSphere Process Server 11
2.1 Overview of WebSphere Process Server 12
2.2 An overview of SCA 14
2.3 Developing business processes 15
2.3.1 Typical development process 18

Chapter 3. Mediation with WebSphere Message Broker 21
3.1 WebSphere Message Broker 22
3.1.1 Mediation patterns with WebSphere Message Broker 22
3.2 Runtime architecture of WebSphere Message Broker 24
3.2.1 Broker 25
3.2.2 Execution groups 25
3.2.3 Configuration Manager 25
3.2.4 User Name Server 26
3.2.5 Broker domain 26
3.3 Development environment of WebSphere Message Broker 26
3.3.1 Message Brokers Toolkit 26
3.3.2 Node types 28
3.4 Mediation with message flows 30
3.4.1 Routing with message flows 30
3.4.2 Aggregation with message flows 34

© Copyright IBM Corp. 2008. All rights reserved.

3.4.3 Protocol transformation with message flows . . . 35
3.4.4 Message transformation with message flows . . . 39
3.4.5 Message enrichment with message flows . . . 47
3.5 Deploying message flow applications . . . 49

Chapter 4. Connectivity options for interoperability . . . 51
4.1 Messaging transport . . . 52
4.1.1 Message formats . . . 53
4.1.2 Mapping messages to a Service Component Architecture interface 55
4.1.3 MQ binding . . . 58
4.1.4 JMS bindings . . . 67
4.1.5 MQ JMS binding . . . 73
4.2 Web service bindings . . . 75
4.3 Connectivity option summary . . . 76

Part 2. Scenarios . . . 77

Chapter 5. Scenario: Message splitting and routing . . . 79
5.1 Scenario overview . . . 80
5.2 Environment . . . 81
5.3 WebSphere MQ configuration . . . 82
5.3.1 Local queues on the broker queue manager . . . 82
5.3.2 JMS entries . . . 82
5.4 Business integration module . . . 85
5.4.1 Business integration module components . . . 85
5.4.2 XML schema for message containing multiple orders . . . 86
5.4.3 Business object . . . 88
5.4.4 Interfaces . . . 89
5.4.5 Business process . . . 90
5.4.6 Import component and binding . . . 93
5.5 Message flow . . . 97
5.5.1 Scenario1MsgFlw message flow . . . 97
5.5.2 Scenario1MsgSetPrj message set project . . . 99
5.5.3 Message flow nodes . . . 102
5.6 DB2 database configuration . . . 111
5.7 Testing the flow . . . 112

Chapter 6. Scenario: WebSphere Message Broker acting as a transaction manager . . . 115
6.1 Scenario overview . . . 116
6.1.1 Process and message flow . . . 116
6.1.2 XA transactions with WebSphere Message Broker . . . 117

Archived

6.2 Environment 119
6.3 WebSphere Process Server configuration 119
6.3.1 Defining JMS entries 119
6.3.2 Configuring the listener port for the message-driven bean 123
6.4 WebSphere MQ configuration 124
6.4.1 Preparing the transaction manager 124
6.4.2 Preparing the resource managers 128
6.4.3 Making the libraries available 128
6.4.4 Configuring the JMS administered objects 129
6.5 Business integration module 135
6.5.1 Business integration module components 136
6.5.2 Business objects 137
6.5.3 Interfaces 139
6.5.4 Business process 140
6.5.5 Import component and binding 143
6.6 Message flows 144
6.6.1 PrjTxnFlowMSets message set 144
6.6.2 JMSREQRESP message flow 147
6.6.3 A checklist for deploying the message flow 159
6.7 Testing the flow 161

Chapter 7. Scenario: Data transformation and security 165
7.1 Scenario overview 166
7.1.1 Process and message flow 168
7.1.2 Service message formats 169
7.2 Environment 174
7.3 Preparing for security 175
7.3.1 Creating the keystore and truststore databases 175
7.3.2 Configuring WebSphere Message Broker security 176
7.3.3 Configuring WebSphere Process Server security 177
7.4 WebSphere MQ configuration 182
7.5 Business integration module 183
7.5.1 Components of the business integration module 183
7.5.2 Interfaces 188
7.5.3 Business process 189
7.5.4 Import components and binding 195
7.6 Message flows 196
7.6.1 Message set 196
7.6.2 The com.ibm.redbooks.sam730.BankUtils class reference 204
7.6.3 Enabling the HTTPInput node for SSL 205
7.6.4 VerifyCredit message flow 207
7.6.5 OpenAccount message flow 218

Archived

Chapter 8. Scenario: Aggregation 229
8.1 Scenario overview 230
8.2 Environment 231
8.3 Queue manager implementation of WebSphere MQ 232
8.4 Business integration module 232
8.4.1 Business integration module components 232
8.4.2 Business objects 233
8.4.3 Interfaces 236
8.4.4 Business process 236
8.4.5 Import component and binding 240
8.5 Message flows 240
8.5.1 Aggregation implementation overview 240
8.5.2 FindSuppliersMSetprj message set 243
8.5.3 FindSuppliersWSInput message flow 244
8.5.4 FindSuppliersAggFanOut message flow (aggregation fan-out) . . . 249
8.5.5 FindSuppliersAggFanIn message flow (aggregation fan-in) 256
8.5.6 FindSuppliersWSResponse message flow 262
8.5.7 Supplier1 message flow 266
8.5.8 Supplier2 message flow 272
8.6 Testing the flow 273

Chapter 9. Scenario: Asynchronous callback 275
9.1 Scenario overview 276
9.2 Environment 277
9.3 WebSphere Service Registry and Repository 278
9.3.1 OrderWebService Web service 278
9.3.2 Defining the registry 280
9.4 WebSphere MQ infrastructure 282
9.5 Business integration module 283
9.5.1 Business integration module components 283
9.5.2 Business objects 285
9.5.3 Interfaces 287
9.5.4 Business process 288
9.5.5 Import component and binding 294
9.5.6 Export component and binding 296
9.6 Message flow 297
9.6.1 OrderMessageFlow 297
9.6.2 ErrorHandleFlow 304

Archived

9.7 Testing the flow 309

Chapter 10. Scenario: Connectivity to third-party JMS providers 311
10.1 Scenario overview 312
10.2 Environment 313
10.3 WebSphere Process Server configuration 313
10.3.1 Service integration bus configuration 314
10.3.2 JMS entries 315
10.4 WebSphere Message Broker configuration 317
10.5 WebSphere MQ configuration 317
10.6 Progress SonicMQ configuration 318
10.7 SwiftMQ configuration 318
10.8 Business integration module 318
10.8.1 Business integration module components 319
10.8.2 Business objects 320
10.8.3 Interfaces 321
10.8.4 Business process 322
10.8.5 Import component and binding 325
10.9 Message flow 326
10.10 Testing the flow 331

Related publications 335
IBM Redbooks 335
Online resources 335
How to get Redbooks 337
Help from IBM 337

Index 339

Archived

Archived

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 give 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.

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.

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.

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.

© Copyright IBM Corp. 2008. All rights reserved.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX 5L™	DB2®	SupportPac™
AIX®	Everyplace®	System z™
CICS®	IBM®	Tivoli®
DataPower®	Redbooks®	WebSphere®
DB2 Universal Database™	Redbooks (logo) ®	z/OS®

The following terms are trademarks of other companies:

BAPI, SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

eXchange, EJB, Java, Java Naming and Directory Interface, JavaServer, JavaServer Pages, JDBC, JDK, JNI, JRE, JVM, J2EE, J2SE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Archived

Preface

IBM® WebSphere® Process Server is a business integration server that was built to support solutions that are based on the service-oriented architecture (SOA). It plays a key role in the architecture of the IBM SOA Foundation by providing functionality for process services. Another key component of the architecture is the enterprise service bus (ESB). IBM provides two key ESB products: IBM WebSphere Enterprise Service Bus and WebSphere Message Broker.

This IBM Redbooks® publication has been written for architects who are planning an SOA solution and application designers who are implementing an SOA solution with WebSphere Process Server and WebSphere Message Broker. In this book, we highlight the ESB capabilities of WebSphere Message Broker and explain how you can leverage them with WebSphere Process Server. In addition, we discuss interoperability and provide examples to illustrate the integration of the two products.

The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Raleigh Center.

Carla Sadtler is a Consulting IT Specialist at the ITSO, Raleigh Center. She writes extensively about the WebSphere and Patterns for e-business areas. Before joining the ITSO in 1985, Carla worked in the Raleigh branch office as a Program Support Representative. She holds a degree in mathematics from the University of North Carolina at Greensboro.

John Adegbile is an Advisory IT Specialist who is aligned with the Enterprise Application Integration capability in IBM Global Business Services, United Kingdom (UK). He has over 11 years of experience in application development and support in the Public Sector, Pharmaceutical, Retail, and Chemical industries. John's areas of expertise include WebSphere MQ and WebSphere Message Broker.

Erik Kirk is a Software Engineer in the AIM division of IBM. Erik began working for IBM in 2001 using a variety of technologies as he developed several tools to help facilitate the automation of internal business processes. He joined L2 support for WebSphere Message Broker in 2004. His current responsibilities

© Copyright IBM Corp. 2008. All rights reserved.

include resolving customer issues related to WebSphere Message Broker, developing and managing internal automation tools, and managing an internal knowledge base. Erik holds a Master of Computer Science degree from North Carolina State University.

Joao Lopes is an Application Architect working in the Service Innovation at IBM. He is part of the Architectural CoC for SOA in Brazil and has been working on some of the first implementations of SOA release products. As a Java™ developer, prior to J2EE™, he continues to participate in Open Sources development in ASF and also SourgeForge.net initiatives regarding OSOA projects such as Service Component Architecture (SCA) and Service Data Objects (SDO) for Java and C++. Joao holds a degree in physics.

Rosaline Makar is a software engineer in IBM Egypt, Cairo Technology Development Center (C-TDC). She earned a bachelor degree in Computer Engineering from Ain Shams University, Egypt, in 2005, and is currently preparing for a Master of Computer Science degree. Rosaline's areas of expertise include Web services, WebSphere Integration Developer, WebSphere Process Server, and WebSphere Enterprise Service Bus.

Raghunath Nair is a Principal Software Engineer in the India Software Labs in Bangalore. He has nine years of experience and has been working for IBM since early 2006 designing technical solutions involving WebSphere Process Server, WebSphere Message Broker, and WebSphere MQ. Raghu's previous experience was predominantly in the Enterprise Application Integration arena designing solutions using the Sun-SeeBeyond suite of products including ICAN, eGate, eInsight, and eXchange™. Raghu has a Bachelor of Technology degree in Electronics and Communications from the University of Calicut, Kerala, India.

Thanks to the following people for their contributions to this project:

Paul Lacy
Anthony O'Dowd
IBM Hursley

Stuart Jones
IBM US

Archive

Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Archived

Archived

Part 1

WebSphere Message Broker as an ESB

In this part, we highlight the enterprise service bus (ESB) capabilities of WebSphere Message Broker and explain how you can leverage them with WebSphere Process Server. This part includes the following chapters:

- Chapter 1, “The IBM SOA Foundation and products in this book” on page 3
- Chapter 2, “Business processes with WebSphere Process Server” on page 11
- Chapter 3, “Mediation with WebSphere Message Broker” on page 21
- Chapter 4, “Connectivity options for interoperability” on page 51

© Copyright IBM Corp. 2008. All rights reserved.

Archived

1

The IBM SOA Foundation and products in this book

The purpose of this book is to explain and illustrate the interoperability of WebSphere Process Server and WebSphere Message Broker. In this chapter, we set the stage by providing a brief background on the IBM SOA Foundation and the products discussed in this book. Specifically, we address the following topics:

- "IBM SOA Foundation" on page 4
- "WebSphere Process Server" on page 5
- "The enterprise service bus" on page 6
- "The focus of this book" on page 10

© Copyright IBM Corp. 2008. All rights reserved.

1.1 IBM SOA Foundation

The IBM SOA Foundation is an integrated, open set of software, best practices, and patterns that provides what you need to get you started with the service-oriented architecture (SOA). The SOA Foundation provides full support for the SOA life cycle through an integrated set of tools and runtime components that allow you to leverage skills and investments across the common runtime, tooling, and management infrastructure.

Because the components are modular, you can pick and choose the pieces that you need to deliver an immediate impact, while knowing that what you pick will work with pieces that you add later. In addition, since the SOA Foundation is scalable, you can start your business small and grow it as fast as it requires. The SOA Foundation provides extensive support for business and IT standards, facilitating greater interoperability and portability between applications. The foundation can also help you to leverage SOA to extend the value of the applications and business processes that are running your business today.

The IBM SOA reference architecture (Figure 1-1 on page 5) is a way of looking at the set of services that go into building an SOA. These capabilities can be implemented on a build-as-you-go basis, allowing capabilities and project-level solutions to be easily added as new requirements are addressed over time. The reference architecture shows the tight integration with other critical IT aspects such as security, IT monitoring, virtualization, and workload management.

In Figure 1-1, two areas are highlighted to draw attention to the areas of interest in this book. The first area is *process services*, which provide the control capabilities that are required to manage the flow and interactions of multiple services in ways that implement business processes. WebSphere Process Server provides the capabilities that are required for process services in an SOA solution.

The backbone of the reference architecture is the *enterprise service bus* (ESB) that facilitates communication between services. The ESB provides the inter-connectivity capabilities that are required to use services that are implemented across the entire architecture. Transport services, event services, and mediation services are all provided through the ESB. An ESB can be implemented with one product or multiple products that are combined to provide the function required. The ESB solution that is selected should be optimized to meet the unique business requirements of the enterprise. The solutions should also be capable of changing as your business requirements evolve.

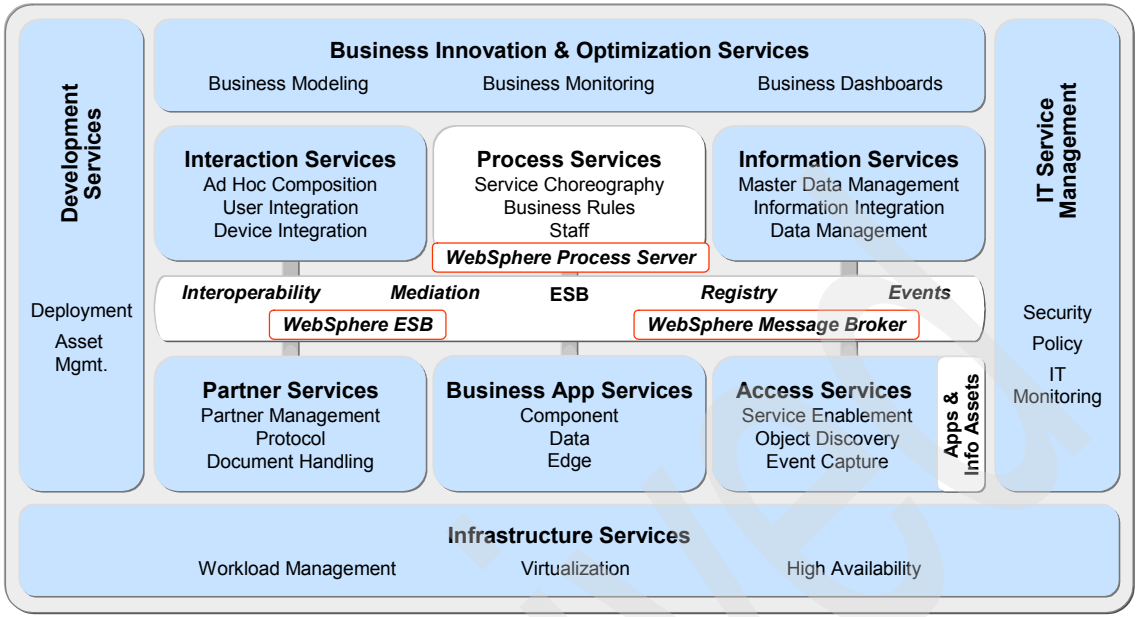


Figure 1-1 IBM SOA reference architecture

The focus of this book is on the implementation of process services with WebSphere Process Server and its use of WebSphere Message Broker as an ESB.

1.2 WebSphere Process Server

IBM WebSphere Process Server is a business integration server that was built to support solutions based on SOA. You can use it to build advanced business processes and traditional business integration such as enterprise application integration. WebSphere Process Server is based on WebSphere Application Server and contains the best of the product features that were previously found in WebSphere MQ Workflow, WebSphere InterChange Server and WebSphere Business Integration Server Foundation.

By building on top of WebSphere Application Server Network Deployment, WebSphere Process Server can take advantage of all the mature capabilities that it provides, such as clustering, high availability, embedded messaging, and transaction management.

While WebSphere Process Server includes WebSphere Enterprise Service Bus to provide ESB functionality, this does not preclude using WebSphere Message

Broker as an ESB, a combination of the two, or extending the ESB with additional products.

IBM WebSphere Integration Developer is the integration tool that you use to author SOA-based services and choreograph them into business processes that you can deploy on WebSphere Process Server.

1.3 The enterprise service bus

An ESB acts as a mediator between service consumers and service providers. It decouples the service consumer from the service provider, providing services to resolve differences in protocol and format. It shrinks the number of interfaces and improves the reusability of interface components to cut cycle time from design to deployment.

An ESB is an architectural concept that is implemented through the use of one or more products. IBM currently offers two ESB products. WebSphere ESB is built on proven messaging and Web services technologies. It provides standards-based Web services connectivity and XML data transformation. WebSphere ESB is shipped with WebSphere Process Server. WebSphere Message Broker provides universal connectivity, including Web services, and any-to-any data transformation. In addition, such products as DataPower® and WebSphere Transformation Extender can be used to extend the capabilities of the core ESB products.

ESB products: An ESB is a key part of a complete solution for connectivity. The functionality of an ESB can be served through one product or a combination of products. Selecting the right ESB product set with the right set of capabilities is critical for the success of a project. An ESB might need to be extended or complemented with additional features or products to deliver a complete solution to meet your connectivity requirements.

1.3.1 The role of an enterprise service bus

In a non-SOA solution, communication between service requesters and providers is accomplished via a series of direct connections, one connection per requester/provider pair. This requires that the requester be aware of the requirements of the provider and vice versa, for example, the transport protocol, message format, and location of the service. As enterprise solutions grow, the web of direct connections and the complexity of managing those connections also grows.

In an SOA solution, an ESB acts as an intermediary between requesters and providers. Connections are made to the ESB, rather than directly between the communicating pair. The ESB makes the transition between transport protocols and message formats used by the requester and provider. The location of each service is known to the ESB but not to the service. This decoupling of the consumer's view of a service from the implementation of a service provides the following advantages:

- Reduces the number, size, and complexity of interfaces
- Reduces the impact of changes made to the format and location of services, both in terms of impact to the applications and in terms of system management
- Enables integration between disparate resources
- Allows the substitution of one service provider for another without the service consumer being aware of the change or without the need to alter the architecture to support the substitution

The capabilities of an ESB vary depending on the products that are used to implement it. An ESB should provide the following basic functions:

- Routing messages between services

  The requester sends the request to the ESB. The ESB makes the call to the service provider. The ESB determines the destination for the service provider.

- Converting transport protocols between the requester and service

  Without an ESB infrastructure, service consumers connect directly to service providers using a transport protocol that is supported by the provider. With an ESB, there is no direct connection between the consumer and provider. The service consumer connects to the ESB by using its preferred transport protocol without any knowledge of how the connection to the provider is made. The ESB sends the request to the provider by using a protocol that is supported by the provider.

- Transforming message formats between requester and service

  Typically, interfaces and operations of disparate services are not identical. The ESB transforms the message from the source into a format that can be accepted by the target.

- Handling business events from disparate sources

  The same service provider that is responsible for a business function can be invoked from a variety of business contexts.

Archived

1.3.2 WebSphere Message Broker V6

WebSphere Message Broker is a powerful information broker that allows both business data and information, in the form of messages, to flow between disparate applications and across multiple hardware and software platforms. Rules can be applied to the data that is flowing through the message broker in order to route, store, retrieve, and transform the information.

WebSphere Message Broker offers the following features:

- Universal connectivity
  - Simplifies application connectivity to provide a flexible and dynamic infrastructure
- Routes and transforms messages *from* anywhere, *to* anywhere.
  - Supports a wide range of protocols, for example, MQ, Java Message Service (JMS) 1.1, HTTP(S), Web services, file, and user-defined protocols
  - Supports a broad range of data formats, for example, binary (C/COBOL), XML, industry (SWIFT, EDI, HIPAA, and so on), and user-defined formats
  - Supports interactions and operations that allow you to route, filter, transform, enrich, monitor, distribute, decompose, correlate, and more
- Simple programming
  - Message flows process and route messages. A message flow contains a series of connected nodes that have the required integration logic that is used to operate on messages as they flow through the broker.
  - Message trees describe the data in a format independent manner.
  - Transformation options include graphical mapping, Java, extended SQL (ESQL), Extensible Stylesheet Language (XSL), and WebSphere Transformation Extender.
- Operational management and performance
  - Extensive administration and systems management facilities for developed solutions
  - A wide range of operating system and hardware platforms supported
  - Performance of traditional transaction processing environments
- Support for adapters
  - A collection of software, application program interfaces (APIs), and tools in WebSphere Business Integration Adapters that enable applications to exchange business data through an integration broker

    WebSphere Business Integration Adapters rely on JMS messaging.

Archived

WebSphere Message Broker contains a choice of transports that enable secure business to be conducted at any time and any place, providing powerful integration using mobile, telemetry, and Internet technologies. WebSphere Message Broker is built upon WebSphere MQ and therefore supports the same transports. However, it also extends the capabilities of WebSphere MQ by adding support for other protocols, including real-time Internet, intranet, and multicast endpoints.

Drivers for using WebSphere Message Broker as an ESB

Consider the following reasons for using a WebSphere Message Broker as an ESB with WebSphere Process Server:

- You are currently using WebSphere Message Broker and want to leverage existing skills.
- You have extensive heterogeneous infrastructures, including both standard and non-standards-based applications, protocols, and data formats. For example, you are using industry formats such as SWIFT, EDI, or HL7.
- You are implementing a wide range of messaging and integration patterns, for example, complex event processing and correlation, message splitting, and aggregation.
- You need extensive pre-built mediation support.
- Reliability and extensive transactional support are key requirements.
- You want to achieve high-performance with horizontal and vertical scaling.
- You want integration with other IBM WebSphere and Tivoli® products, as well as third-party JMS providers.
- You want to natively transform non-XML formats without the use of adapters or user-written data transformations.
- You want advanced publish and subscribe capabilities, including the use of a wide range of transports; high speed, low-latency messaging; and both topic and content-based routing.
- You need complex event processing. Complex events are conditions that relate to multiple messages and are defined by applying a combination of logical, arithmetic, and temporal operators to messages or message attributes.

  The IntelligentFilter and SituationManager SupportPac™ provides the ability to handle complex events that span multiple messages. Its complex event processing engine enables fast and reliable development of reactive applications, without requiring programming skills.
- You want to integrate telemetry devices. WebSphere Message Broker can coordinate end-to-end telemetry integration. It can serve as a conduit for

Archived

passing data from remote telemetry devices into enterprise applications. Similarly, it can control data flow from command or control applications to remote control devices. For more information, see the technical overview of WebSphere Message Broker Version 6.0 in the WebSphere Message Broker information center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/ab20551_.htm

1.4 The focus of this book

Our intent for this book is to point out the key features that make WebSphere Message Broker a powerful choice in an ESB solution for WebSphere Process Server and to illustrate the interoperability between the two products. We use realistic examples and scenarios to illustrate the ESB capabilities of WebSphere Message Broker and to show how to interconnect the two.

For an excellent book about ESB design and using WebSphere ESB, WebSphere Message Broker and DataPower appliances as ESB solutions, refer to *Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB*, SG24-7369.

Archived

2

Business processes with WebSphere Process Server

IBM WebSphere Process Server hosts the applications that drive business processes. This book focuses on the interaction of a business process running in WebSphere Process Server and WebSphere Message Broker. In this chapter, we provide a quick introduction to WebSphere Process Server and the Service Component Architecture (SCA). It includes the following topics:

- "Overview of WebSphere Process Server" on page 12
- "An overview of SCA" on page 14
- "Developing business processes" on page 15

© Copyright IBM Corp. 2008. All rights reserved.

2.1 Overview of WebSphere Process Server

Figure 2-1 shows an overview of the components in WebSphere Process Server.

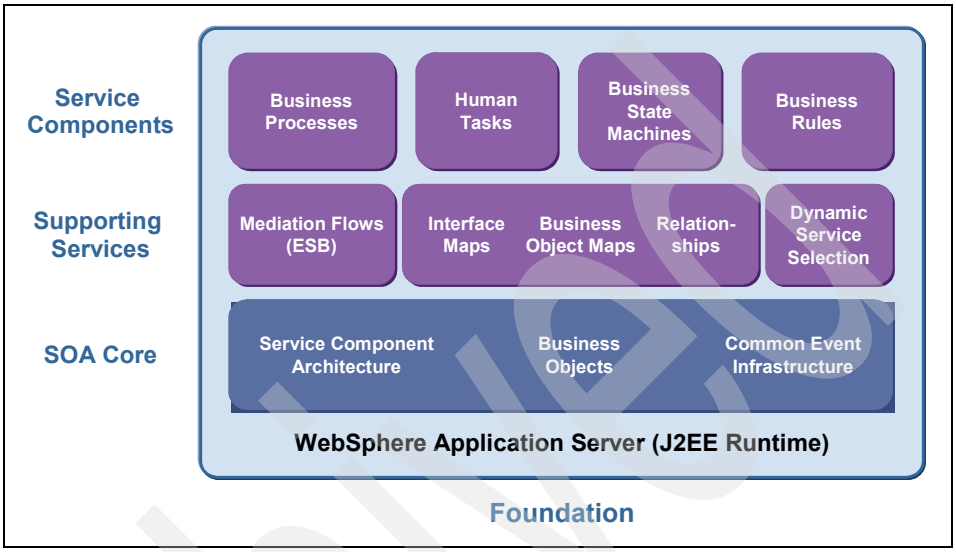


Figure 2-1 WebSphere Process Server architecture

WebSphere Process Server includes three layers:

- The *SOA core layer* consists of the following components:
  - The *SCA* presents all elements of business transactions, including access to Web services, Enterprise Information System (EIS) service assets, business rules, workflows, databases, and so on, as service components. The SCA separates business logic from implementation, so that you can focus on assembling an integrated application without knowing the implementation details. Service components can be assembled graphically in IBM WebSphere Integration Developer, and the implementation can be added later.
  - *Business objects* define the data flowing between SCA components. Business objects provide an abstraction for data access and are based on a data access technology called *Service Data Objects* (SDO). SDOs provide a universal means of describing disparate data. Business objects provide rich features to map, manage, and transform data to underlying IT and are described through a standards-based XML schema.
  - With the *Common Event Infrastructure* (CEI), service components can emit events that can be captured by business monitors such as

WebSphere Business Monitor for real-time monitoring of business processes.

- *Supporting services* are components that are needed in any integration solution, including data transformation and synchronization services. This layer contains the following components:
  - *WebSphere Enterprise Service Bus* provides the ESB functionality, with mediation flows that can operate on messages to perform XML-based data transformation, protocol transformation between various transports, custom Java operations, and routing.
  - Using *interface maps*, you can invoke components by translating calls to them. It is possible for interfaces of existing components to match semantically but not syntactically. This is especially true for components that already exist and services that need to be accessed.
  - With *business object maps*, you can translate one type of business object into another type of business object. You can use these maps in a variety of ways, for example, in an interface map to convert one type of parameter data into another.
  - You can use *relationships* to establish relationship instances between object representations of the same logical entity in disparate back-end systems. You may want to access the same logical entity within business integration scenarios. For example, the same customer's address might need to be updated in various back-end systems, such as an enterprise resource planning (ERP) system and a customer relationship management (CRM) system. These relationships can be established and managed automatically using the relationships service component. These relationships are typically accessed from a business object map when translating one business object format into another.
  - You can use *selectors* for dynamic selection and invocation of different services, which all share the same interface. WebSphere Process Server offers a Web-based interface to enable dynamic updates to the selection criteria and target services, which means that a module that has been deployed at a later time can still be called by this selector component, enabling dynamic changes to the integration solution.
- WebSphere Process Server provides the following *service components*:
  - A *business process* component implements a Business Process Execution Language (BPEL)-compliant process. You can develop and deploy business processes that support long- and short-running business processes and a compensation model within a scalable infrastructure. You can create BPEL models in WebSphere Integration Developer or import from a business model that you have created in WebSphere Business Modeler or any other modeling tool that supports the BPEL standard.

– *Human tasks* are stand-alone components that you can use to assign work to employees. Additionally, the human task manager supports the ad hoc creation and tracking of tasks. WebSphere Process Server also supports multi-level escalation for human tasks, including e-mail notification and priority aging.

– *Business state machines* provide another way of modeling a business process. Some processes are easily described as a sequential flow of activities, and they can be modeled as business processes. However some processes are driven by events rather than a sequence, and in this case, the business state machine is a better fit for modeling the process. One example is an ordering process where you can modify or cancel the order at any time during the order process until the order is fulfilled.

– *Business rules* are a means of implementing and enforcing business policy through externalization of business function. As a result, dynamic changes of a business process are enabled for a more responsive business environment.

2.2 An overview of SCA

Figure 2-2 provides a high level look at an SCA service component.

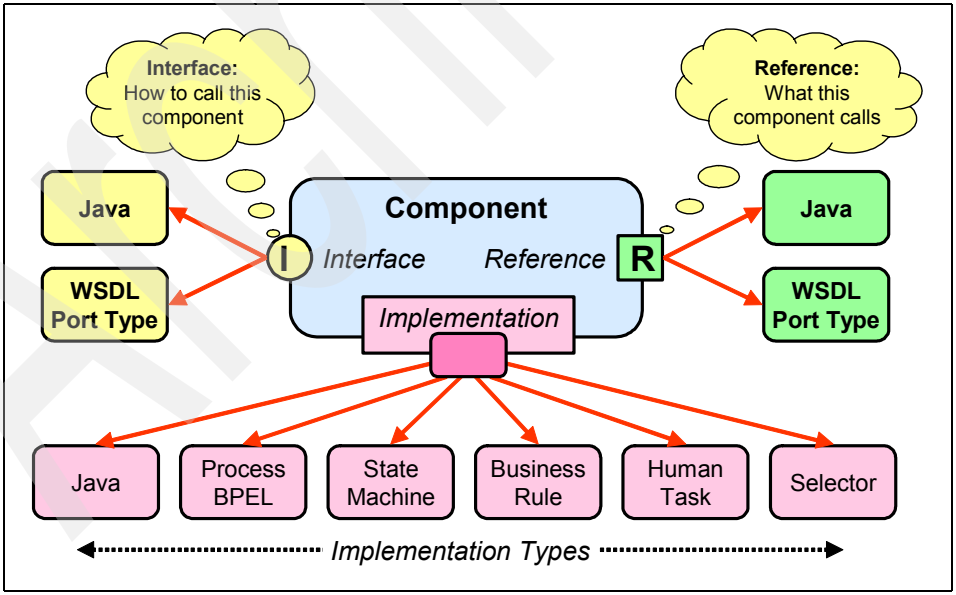


Figure 2-2 SCA service component

The following terms describe the SCA service component and the WebSphere Process Server implementation as shown in Figure 2-2 on page 14:

- SCA interface

  By definition, an *interface* is the place at which independent and often unrelated systems meet and communicate with each other. An interface can be defined by any programming or interface definition language.

  WebSphere Process Server currently supports a Java interface definition and an XML definition (Web Services Description Language (WSDL) port type). The arguments described in an XML schema are exposed to programmers as SDOs. The WebSphere Integration Developer tooling primarily generates interfaces using the WSDL representation.

- SCA implementation

  The SCA implementation specifies the implementation type of the component's interface. Developers can implement business services in their language of choice, for example, Java, BPEL, or state machine.

  Current implementation types include business process, human task, interface map, selector, business rules, business state machine, and Java.

- SCA references

  An SCA reference specifies other SCA services that a component uses. These can be soft links that do not have to specify which specific component will be used.

2.3 Developing business processes

Business processes are the implementation in WebSphere Process Server that drive the logic of a business solution. The focus of this book is not on how the business processes are created or the many functions that you can achieve with them, but on how WebSphere Message Broker can be used as a mediary in invoking the services called by a business process. To understand how this interaction happens, you must know how a business integration module is developed. You must also understand the business object representations, the business process implementation, and the bindings that define how a business process is accessed by clients and how a business process accesses its services.

WebSphere Integration Developer is the integration tool that you use to author SOA-based services and choreograph them into business processes that you can deploy on WebSphere Process Server. These artifacts are contained in a *business integration module*, or simply a *module*. WebSphere Process Server includes a fully integrated testing, debug, and deployment environment.

The primary perspective for working with business modules is the Business Integration perspective. When you create a module, an assembly diagram for the module is created. As you add components to your module, you populate the assembly diagram with these components. Figure 2-3 shows the Business Integration perspective with one module defined. The assembly diagram for the module is in the editor area on the upper right pane. Selecting a component displays information about it in the Properties view (the bottom right pane).

From the assembly diagram, you define the interfaces to each component, generate binding information for access to the module, and can generate the implementation for service components.

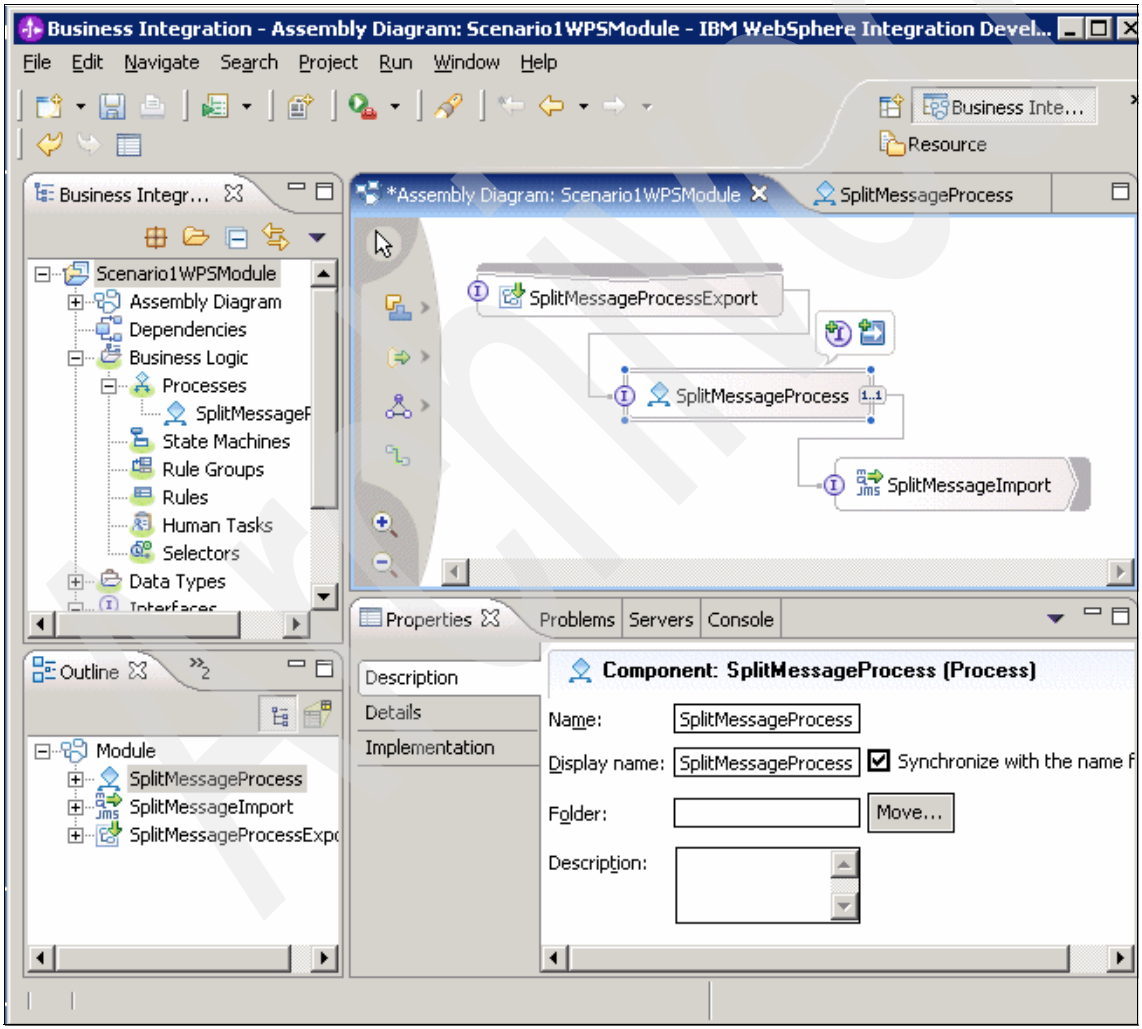


Figure 2-3 Assembly diagram of the Business Integration perspective

In Figure 2-3 on page 16, the assembly diagram contains three components:

- The *export component* represents the interface to the module for clients. An interface is defined and denoted on the export with the icon. An SCA binding has been generated for this component, denoted with the icon.
- The *import component* represents the interface to a service, in this case, an interface to WebSphere Message Broker. As you can see, it has an interface defined, and an MQ JMS binding generated for it, denoted with the icon.
- The *service component*, in the middle, has been implemented as a business process, denoted by the icon (Figure 2-4). When you double-click this component, the business process opens in the editor area. Using the menus and views in this perspective, you can add activities to the business process and customize their properties.

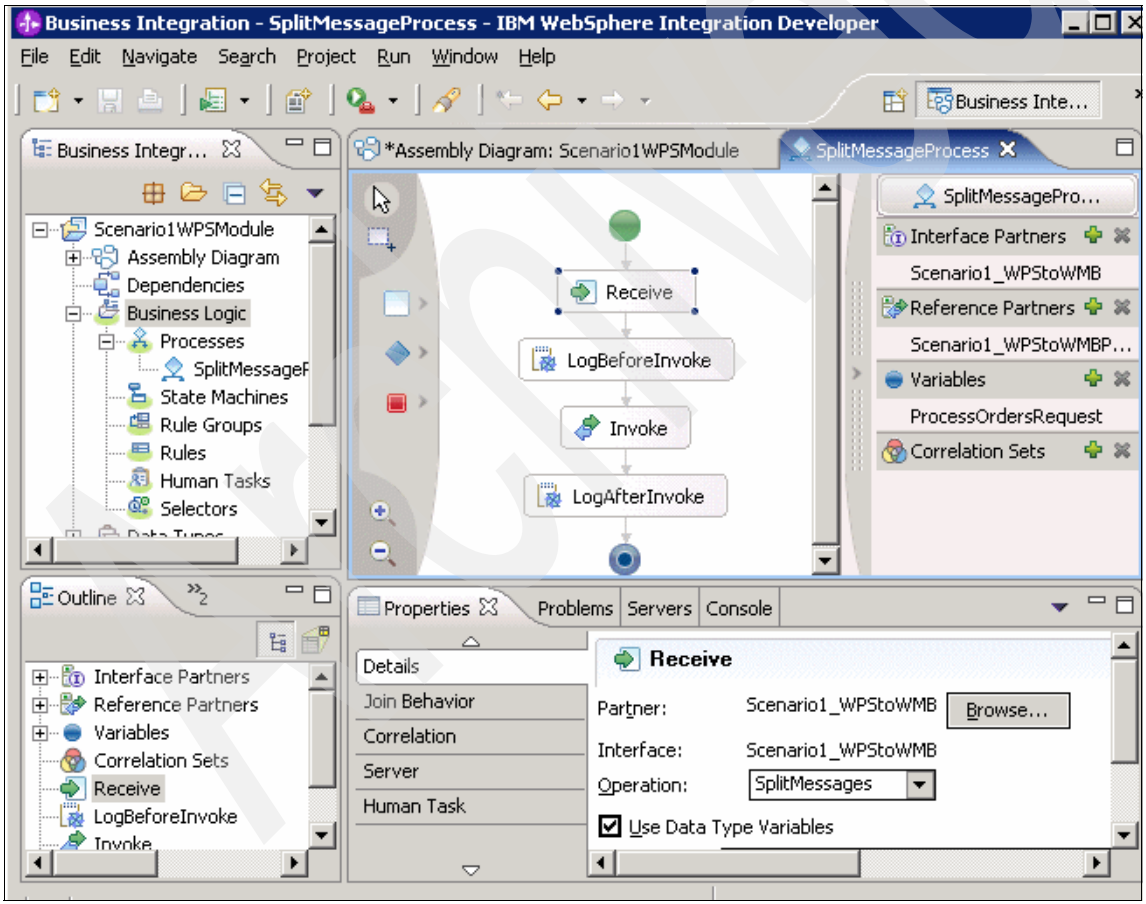


Figure 2-4 Business process of the Business Integration perspective

Obviously, this a simplification of the capabilities of WebSphere Integration Developer. The idea, however, is to provide a background for situations that you will encounter later in the scenarios.

2.3.1 Typical development process

In this section, we provide an overview of the development process for a business process. For examples of building business processes from beginning to end, see *Patterns: Building Serial and Parallel Processes for IBM WebSphere Process Server V6*, SG24-7205.

WebSphere Integration Developer provides an *assembly editor* where the developer can group service components into *modules* and specify which service *interfaces* are exposed by the module to outside consumers. Modules expose their functionality by way of *exports* and use the functionality of other modules using *imports*. Modules can be connected using *wires* to form complete integration solutions or composite applications.

We provide examples of some of these elements in the scenarios in Part 2, “Scenarios” on page 77.

The following steps illustrate a typical development flow used to build a business process:

1. Create a new *business integration module*, or simply a *module*, for the project.

   A *module* is a composite of service components, imports and exports. The implementations and interfaces referenced by module components can be contained in the module project or can be stored in a library.

   Tip: Create a library to contain artifacts, such as business objects and interfaces, and reference the library from the module. By using a library, you can share resources and code across modules.

   Each module has a module assembly. The module assembly contains a diagram, which is referred to as the *assembly diagram*, that shows the components of the module and how their interfaces and references are wired together.

   The module is the basic unit of deployment for WebSphere Process Server.

2. Create *business objects* that describe how the components communicate with each other. Business objects are containers for application data that represents business functions or elements, such as customer or order data.

3. Create the *interface* for each component. The interface determines the data that can be passed from one component to another. The interface consists of one or more operations, with each operation defining the input and output to the component. An operation can be a request-response operation or a one-way operation.
4. Create the *service component*. A service component can be implemented as one of the following types:
   - Java component
   - Business process (BPEL process)
   - State machines
   - Business rules
   - Selectors
   - Human task

   In this book, we are focused on the business process. When you create the business process component, an inbound interface (*interface partner*) is assigned to the component. You can create the interface when you create the business process or select an existing interface. The process editor is used to build the business process:

   a. Create *reference partners* for the business process. Reference partners are outbound interfaces of the business process.

   b. Add *activities* to the business process. Activities are the individual business tasks within the process that compose the larger business goal. Consider the following examples:

      - Add an Invoke activity for each call to a partner.
      - Add a Human task activity to send a process-related task to a human for completion.
      - Add a Choice activity to evaluate conditions and select a processing path.

5. Create *exports* and *imports* and bind them to a protocol. These components define a module's external interfaces or access points.

   Imports identify services outside of a module, so that they can be called from within the module. An import is created in the assembly diagram by adding the export or interface of the external module to the assembly diagram.

   Exports allow components to provide their services to external clients. An export component is created in the assembly diagram for a component, so that its business service can be used by other modules.

   Imports and exports require binding information that defines the means of transporting the data from the modules. You can change this binding later to use a different transport.

Archived

Connectivity with WebSphere Message Broker: The bindings define how the process will interact with WebSphere Message Broker. You can find more information about how WebSphere Process Server and WebSphere Message Broker interact in Chapter 4, “Connectivity options for interoperability” on page 51.

6. Test the business integration module in the integrated test environment.
7. Deploy the business module to WebSphere Process Server.

   When you deploy a module to the test environment or to WebSphere Process Server, WebSphere Integration Developer packages the module as a J2EE enterprise archive (EAR) file.

   For any given module project, up to three J2EE staging projects are generated with naming conventions that are based off of the module project's name:

   J2EE staging projects: J2EE staging projects (EJB projects and Web projects) are generated artifacts from the deployment process. When you select **Project** → **Clean**, the system discards the former J2EE staging project and regenerates it.

   – An Enterprise Application project
   – An Enterprise JavaBean (EJB™) project

     This project contains the generated EJBs that represent the runtime artifacts that make components into reality.

   – A Dynamic Web project

     This project contains artifacts that represent Web components, for example, servlets and JavaServer™ Pages™ (JSPs). A dynamic Web project is generated when needed.

   You are unable to see these projects in the Business Integration view. To view these projects, change perspectives, for example, the Web perspective.

Tip: Do not store any user logic in generated EJB or Web projects. Rather, create a separate Web project of your own and add it to the application as a dependent of the generated module.

Archived

3

Mediation with WebSphere Message Broker

In this chapter, we introduce the features of WebSphere Message Broker. We also introduce the runtime architecture and the message flows that perform the enterprise service bus (ESB) functionality.

Specifically, we cover the following topics in this chapter:

- "WebSphere Message Broker" on page 22
- "Runtime architecture of WebSphere Message Broker" on page 24
- "Development environment of WebSphere Message Broker" on page 26
- "Mediation with message flows" on page 30
- "Deploying message flow applications" on page 49

© Copyright IBM Corp. 2008. All rights reserved.

3.1 WebSphere Message Broker

Typically, WebSphere Message Broker is used to integrate disparate systems. It is designed to transform any format of data between any type of application using any communications protocol or distribution method. It is used where there is a need for high-performance and complex integration patterns.

In this section, we discuss the use of WebSphere Message Broker as the ESB for WebSphere Process Server.

3.1.1 Mediation patterns with WebSphere Message Broker

Processing logic in WebSphere Message Broker is implemented via *message flows*. Through message flows, messages from business applications can be transformed and routed to other business applications. Message flows are created by connecting *nodes* together. A wide selection of built-in nodes are provided with WebSphere Message Broker. These nodes are used to perform tasks that are associated with message routing, transformation, and enrichment. The base capabilities of WebSphere Message Broker are enhanced by SupportPacs that provide a wide range of additional enhancements.

In the following sections, we summarize the mediation patterns that WebSphere Message Broker supports best and explain why a WebSphere Process Server-WebSphere Message Broker combination might be used.

Message routing

Packaged with WebSphere Message Broker is a variety of nodes through which connectivity is provided for both standards and non-standards-based applications and services. Routing can be point-to-point, or it can be based on matching the content of the message with a pattern specified in a node.

Aggregation is an advanced form of message routing. With aggregation, a request message is received, and *multiple* new different request messages are generated. Each new message is routed to its destination using a request-reply interaction. WebSphere Message Broker tracks the process, collecting each response and recomposing them into a single output message.

For information about the options for routing with WebSphere Message Broker, see 3.4.1, “Routing with message flows” on page 30, and 3.4.2, “Aggregation with message flows” on page 34.

Transport protocol conversion

WebSphere Message Broker provides universal connectivity between applications that use disparate transport protocols. WebSphere Message Broker enables connectivity between applications or business processes that use transport protocols such as SOAP, HTTP(S), Java Message Service (JMS), MQ, CICS®, TCP/IP, and FTP, MQe, MQ telemetry, and user-defined transports.

WebSphere Message Broker supports WebSphere Business Integration Adapters. For more information about available adapters, see the WebSphere Adapters page at:

http://www-306.ibm.com/software/integration/wbiadapters/

For information about the options for transport protocol conversion with WebSphere Message Broker, see 3.4.3, “Protocol transformation with message flows” on page 35.

Message transformation and enrichment

One of the key capabilities of WebSphere Message Broker is the transformation and enrichment of in-flight messages. This capability enables business integration without the need for any additional logic in the applications. For example, an application that generates messages in a custom format can be integrated with an application that only recognizes XML. This capability provides a powerful mechanism to unify organizations, because business information can now be distributed to applications that handle completely different message formats.

In WebSphere Message Broker, message transformation and enrichment are dependent upon a broker understanding the structure and content of the incoming message. Self-defining messages, such as XML messages, contain information about their own structure and format. However, before other messages, such as custom format messages, can be transformed or enhanced, a message definition of their structure must exist. The Message Brokers Toolkit contains facilities for defining messages to the message broker.

Using parsers and message sets, WebSphere Message Broker can validate and check that incoming messages comply with the format that is defined in the message set. A flow can be constructed to reject and handle non-compliant messages. Additionally, complex manipulation of message data can be performed using extended SQL (ESQL) or Java facilities that are provided in the Message Brokers Toolkit.

For information about the options for transport protocol conversion with WebSphere Message Broker, see 3.4.4, “Message transformation with message flows” on page 39, and 3.4.5, “Message enrichment with message flows” on page 47.

WebSphere Transformation Extender can be integrated into the WebSphere Message Broker ESB solution to extend the existing capabilities and simplify transformation development.

3.2 Runtime architecture of WebSphere Message Broker

WebSphere Message Broker consists of a development environment, in which message flows and message sets are designed and developed, and a runtime environment on which the message flows execute. Figure 3-1 shows an overview of the WebSphere Message Broker architecture.

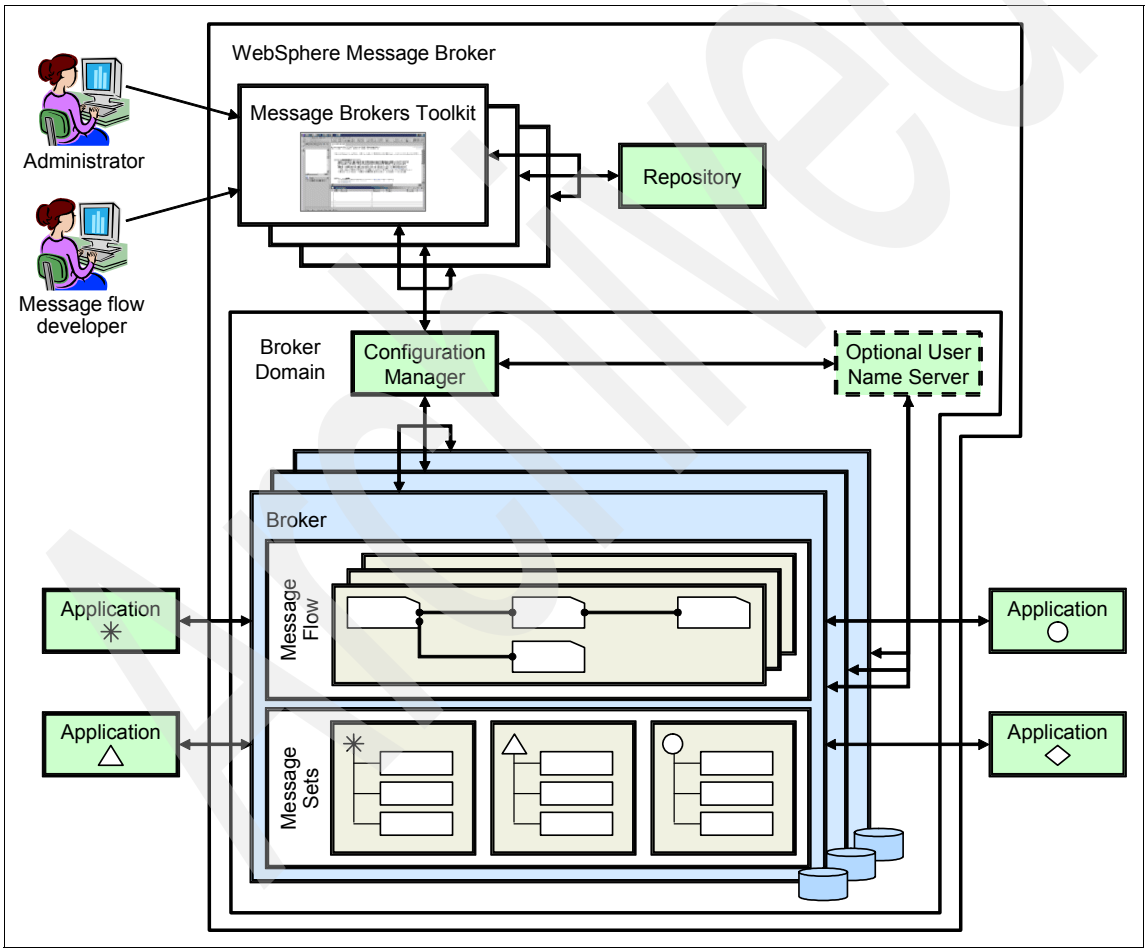


Figure 3-1 Architecture of WebSphere Message Broker

3.2.1 Broker

The *broker* is a set of application processes that host and run message flows. When a message arrives at the broker from a business application, the broker processes the message before passing it on to one or more other business applications. The broker routes, transforms, and manipulates messages according to the logic that is defined in their message flow applications. Each broker uses a database to store the message flows, configuration data, and the message sets deployed to it.

3.2.2 Execution groups

Execution groups are processes that host message flow threads. The execution groups facilitate the grouping of message flows within the broker with respect to functionality, load balancing, or other qualifications that are determined to be necessary. Each broker contains a main execution group. Additional execution groups can be created as long as they are given unique names within the broker.

Each execution group is a separate operating system process. Therefore, the contents of an execution group remain separate from the contents of other execution groups within the same broker. This separation can be useful for isolating pieces of information for security, because the message flows execute in separate address spaces or as unique processes.

Message flow applications are deployed to a specific execution group. To enhance performance, you can deploy additional message flow instances to an execution group. Each additional instance allocates an additional thread to host the message flow application. Therefore, more threads will be available in the execution group to process the incoming message at run time.

3.2.3 Configuration Manager

The *Configuration Manager* is the interface between the Message Brokers Toolkit and the brokers in the broker domain. The Configuration Manager stores configuration details for the broker domain in an internal repository, providing a central store for resources in the broker domain.

The Configuration Manager is responsible for deploying message flow applications to the brokers. The Configuration Manager also reports on the progress of the deployment and on the status of the broker. When the Message Brokers Toolkit connects to the Configuration Manager, the status of the brokers in the domain is derived from the configuration information stored in the internal repository of the Configuration Manager.

3.2.4 User Name Server

A *User Name Server* is an optional component that is required only when publish/subscribe message flow applications are running, and where extra security is required for applications to publish or subscribe to topics. The User Name Server provides authentication for topic-level security for users and groups that perform publish/subscribe operations.

3.2.5 Broker domain

Brokers are grouped together in broker domains. A *broker domain* contains one or more brokers and a single Configuration Manager. It can also contain a User Name Server. The components in a broker domain can exist on multiple machines and operating systems, and are connected together with WebSphere MQ channels. A broker belongs to only one broker domain.

3.3 Development environment of WebSphere Message Broker

In this section, we provide a brief introduction to the development environment of WebSphere Message Broker and the various mediation capabilities that it offers.

3.3.1 Message Brokers Toolkit

The Message Brokers Toolkit is an integrated development environment (IDE) and graphical user interface (GUI) based on the Eclipse platform. The Message Brokers Toolkit offers multiple perspectives for message flow development and runtime management.

Broker Application Development perspective

The Broker Application Development perspective is the default perspective that is displayed the first time that you start the Message Brokers Toolkit. Application developers work in this perspective to develop and modify message sets and message flows. Figure 3-2 on page 27 shows the Broker Application Development perspective with a message flow open in the Message Flow editor.

Archived

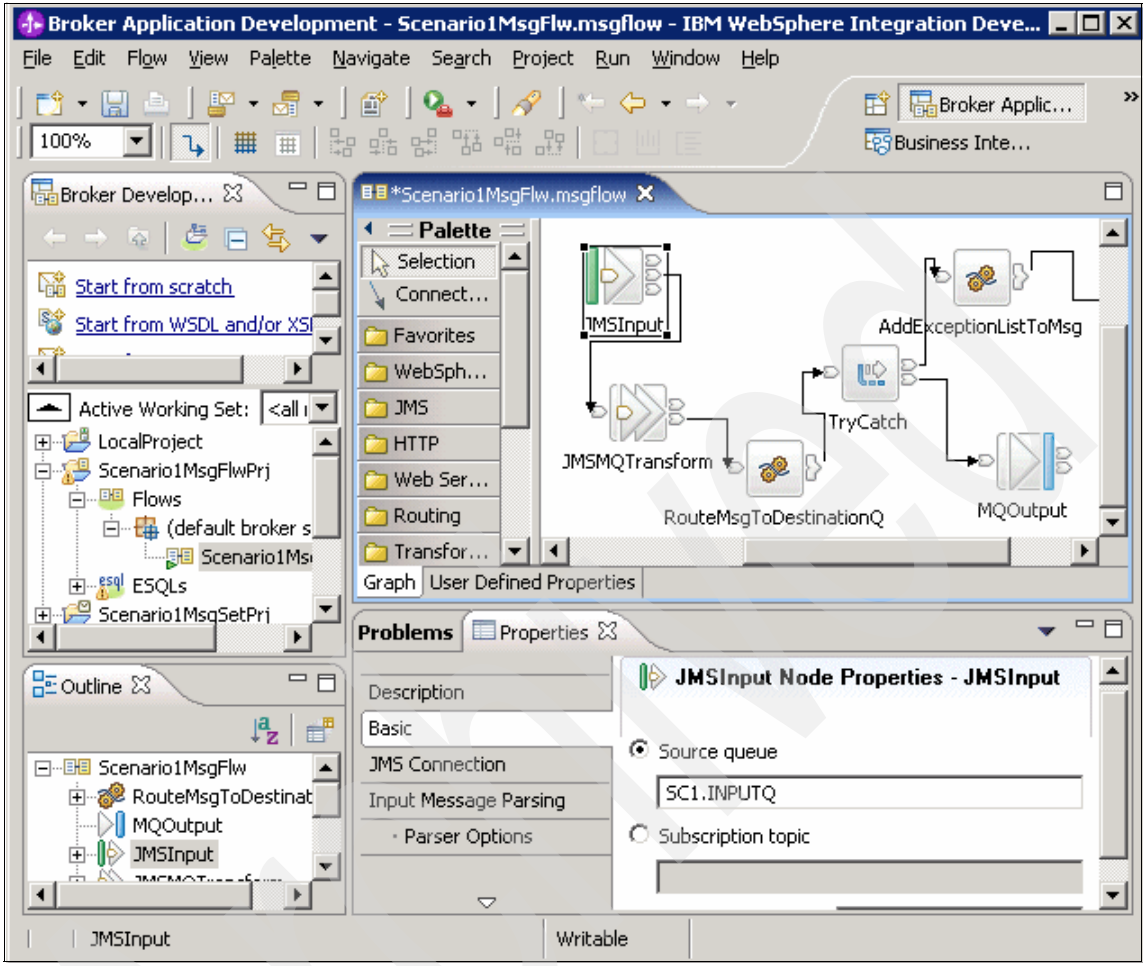


Figure 3-2 Application development perspective

Broker Administration perspective

The Broker Administration perspective is a broker administrative console that communicates with one or more Configuration Managers. Administrators work in this perspective to manage the resources (also referred to as *domain objects*) in the broker domain that are defined on one or more Configuration Managers.

Debug perspective

The Debug perspective is where application developers test and debug message flows.

Plug-in Development perspective

The Plug-in Development perspective is where application developers develop plug-ins for user-defined extensions.

Data perspective

The Data perspective is where application developers import relational database schemas for ESQL content assist and validation.

3.3.2 Node types

Message flows are created by adding nodes to the workspace, defining properties for these nodes, and wiring the nodes together in a logical flow. Table 3-1 lists the nodes that are provided by the Message Brokers Toolkit for message flow development.

You can find details about these nodes in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp

You can add nodes by using SupportPacs. For a list of SupportPacs, see the Business Integration - WebSphere MQ SupportPacs page at:

http://www-1.ibm.com/support/docview.wss?rs=849&uid=swg27007205

Table 3-1 Available node types in a Message Broker Toolkit

Category	Input/output type node	Flow Development nodes
WebSphere MQ	MQInput node MQOutput node	MQReply node MQGet node MQOptimizedFlow node
JMS	JMSInput node JMSOutput node	JMSMQTransform node MQJMSTransform node
HTTP	HTTPInput node HTTPRequest node HTTPReply node	
Routing		Filter node Label node RouteToLabel node Publication node AggregateControl node AggregateRequest node AggregateReply node

Archived

Category	Input/output type node	Flow Development nodes
Transformation		Mapping node XMLTransformation node Compute node JavaCompute node ResetContentDescriptor node
Construction		Trace node Input node Output node TryCatch node Throw node FlowOrder node Passthrough node
Database		Database node DataInsert node DataUpdate node DataDelete node Warehouse node Extract node
Validation		Check node Validate node
Timer		TimeoutControl node TimeoutNotification node
Additional protocols	SCADAInput node SCADAOutput node Real-timeInput node	Real-timeOptimizedFlow node
Available via SupportPac IA90		SOAPEnvelope node SOAPExtract node
Available via SupportPac IA9W		PasswordBasedEncryption node PasswordBasedDecryption node

Archive

3.4 Mediation with message flows

Message flows are used to perform mediation functions on messages that flow through WebSphere Message Broker. We address the following basic mediation functions in this section:

- Routing with message flows
- Aggregation with message flows
- Protocol transformation with message flows
- Message transformation with message flows
- Message enrichment with message flows

3.4.1 Routing with message flows

Routing a message involves sending an incoming message to a destination based on some criteria. The destination can be predefined (static) or based on information that is obtained at the time of the message flow (dynamically).

A common routing pattern includes a dynamic lookup of the destination based on the incoming message type and routing the message to that destination. This routing pattern typically consists of the steps shown in Figure 3-3.

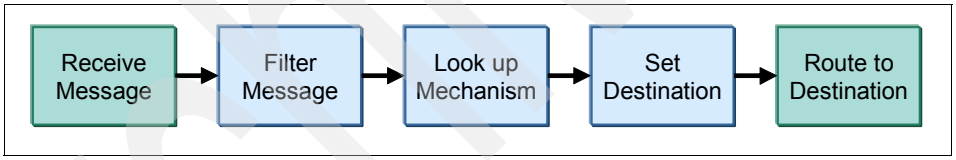


Figure 3-3 A typical routing mechanism

In this section, we discuss the following ways to implement routing in message flows and the nodes that are available to achieve the it:

- Routing using JavaCompute Out and Alternate terminals
- Routing using the RouteToLabel node
- Routing use a database

Aggregation is an advanced form of routing with a request/reply flavor. Refer to 3.4.2, “Aggregation with message flows” on page 34, which discusses this type of mediation. Also refer to Chapter 9, “Scenario: Asynchronous callback” on page 275. That chapter contains an example that uses the SRRetrieveITService node to perform a lookup in the WebSphere Service Registry and Repository for a service with specific characteristics and uses the information that it receives to provide a service URL dynamically.

Routing using JavaCompute Out and Alternate terminals

The JavaCompute node includes two-directional routing capability via two output terminals, labeled in the toolkit as the “Out” and “Alternate” terminals (Figure 3-4).

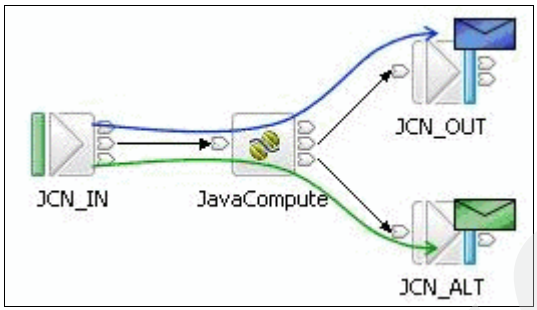


Figure 3-4 Using JavaCompute Out and Alternate terminals

The code extract in Example 3-1 gets these terminals, so that they can be propagated to within this method. The disadvantage of this technique is that it only routes to two terminals.

Example 3-1 Propagating to the Out terminal from the JavaCompute node

```
public void evaluate(MbMessageAssembly assembly) throws MbException
{
MbOutputTerminal out = getOutputTerminal("out");
MbOutputTerminal alt = getOutputTerminal("alternate");
out.propagate(assembly);
```

Archived

Routing using the RouteToLabel node

As shown in Figure 3-5, the JavaCompute node can be used with a RouteToLabel node and one or more Label nodes to dynamically determine the route that a message takes through the message flow. This flow illustrates an example where the JavaCompute node determines to which label node the RouteToLabel node propagates the message.

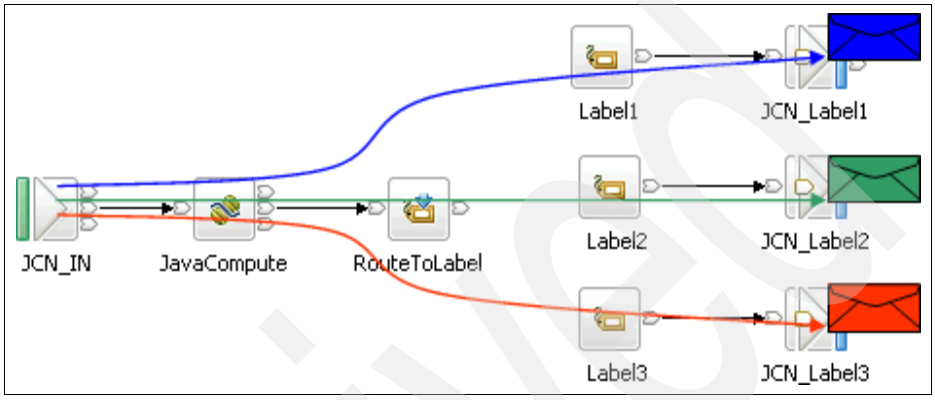


Figure 3-5 Usage of the RouteToLabel and Label nodes for routing

Routing use a database

Using a database table is the recommended way to store any form of routing or transformation data for the following reasons:

- It removes business data from the ESQL or Java code in the message flow.
- It allows for updates in the routing data without message flows having to be changed and redeployed.
- It makes ESQL or Java easier to read by removing the need to have large `If ... then ... else` type of code structures.
- It allows routing information to be stored in one place but used by many different message flows and possibly other applications.

The following approaches use a database as a means to find routing information:

- Directly accessing the database table

  The message flow in Figure 3-6 uses a Compute node to look up the destination and route the message to the target.

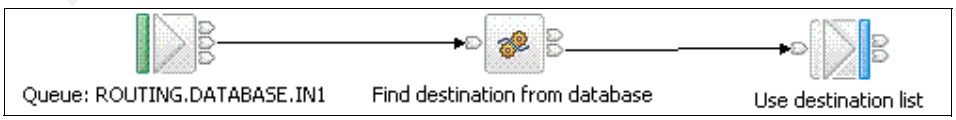


Figure 3-6 Database lookup of routing information

Archived

The disadvantage of this method is that performance overhead is incurred to access the database each time a message passes through a message flow.

- Using an in-memory cached version of the database table

  Starting with WebSphere Message Broker V6.0, database tables can be stored in-memory in a message flow using shared variables, removing the need for continued access to the database table. This method provides the advantages of using a database table without the performance overhead.

  The cached database table can be refreshed by restarting the message flow or by sending a refresh message to the message flow. This approach overcomes the disadvantage here that the cache will not pick up changes to database table as they happen. To pick up the changes, you must either restart the message flow or add a refresh mechanism to the message flow as shown in Figure 3-7.

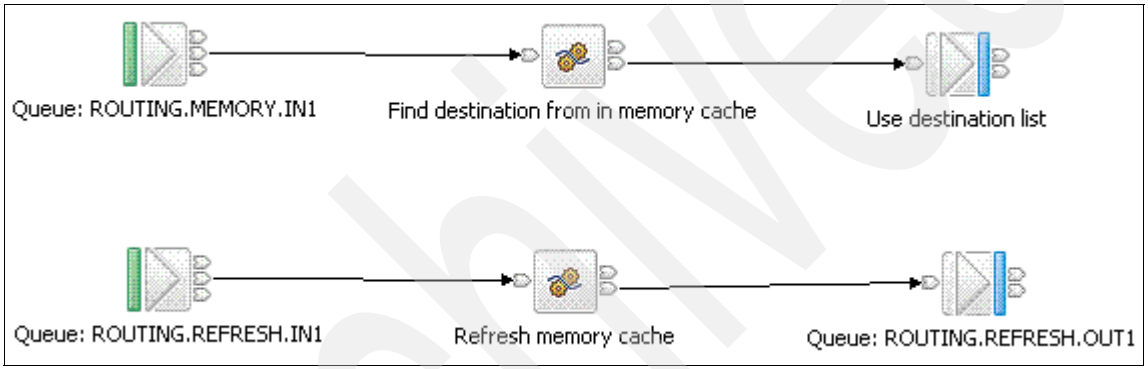


Figure 3-7 Routing with lookup using shared variables and refreshing of cache

Examples: The following examples illustrate routing using a database:

- The WebSphere Message Broker samples gallery includes a working example of the message flow shown in Figure 3-6 on page 32. In the Message Brokers Toolkit, select **Help** → **Samples Gallery** → **Application Samples** → **Message Brokers** → **Message Routing**.
- The WebSphere Message Broker samples gallery includes a working example of the routing flow in Figure 3-7 on page 33. In the toolkit, select **Help** → **Samples Gallery** → **Application Samples** → **Message Brokers** → **Message Routing**.
- Chapter 5, “Scenario: Message splitting and routing” on page 79, has an example of routing using a Compute node and database.

Archive

3.4.2 Aggregation with message flows

Aggregation is a mediation concept that involves the collection of the responses from two or more applications to fulfill a request from the requester. The responses are merged into a single message and sent to the requester or to another target application.

Aggregation in message flow applications can be described in three phases:

1. A message is received by the message broker. Based on this message, the broker requests information from one or more applications. The following examples are of initial message processing:
   - The initial message is a request that can be forwarded as a request to begin the aggregation.
   - The initial message requires parsing, so that parts of the message can be used to build different requests, where each request is sent to the corresponding application whose response will be aggregated with the others.
   - The initial message might need to be enriched in order to build the aggregation request.
   - The initial message might not have arrived over the MQ transport, so it needs to be transformed before the request is built.
2. After the message is processed and the requests are built, the requests are sent to the respective applications. The process of sending the requests is referred to as the *fan-out process*. The fan-out process can involve sending the request to any application that supports the MQ request-reply interaction model. Two built-in nodes are supplied for use in the fan-out flow: the AggregateControl node and the AggregateRequest node.

   If an application does not support the MQ request-reply interaction model, another message flow can be used to act as an interface to the application whose response is needed for the application. For example, if a Web service response is needed for the aggregation, an additional message flow can be introduced to act as an interface between the aggregation message flow application and the Web service by implementing the Web service with an MQ interface.

   For more information about how to expose a Web service as an MQ application service, search on “Broker implements non-web-service interface to new Web service” in the WebSphere Message Broker Information Center:

   `http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp`

3. The broker waits for the responses to the requests and compiles the responses into a single message. This process is referred to as the *fan-in process*. When the fan-in process completes, the message is delivered to its destination. The AggregateReply node is provided for this function.

Examples:

- The WebSphere Message Broker samples gallery includes a working example of the aggregation flows illustrated in this section. In the Message Brokers Toolkit, select **Help** → **Samples Gallery** → **Technology Samples** → **Message Brokers** → **Aggregation**.
- Refer to Chapter 8, “Scenario: Aggregation” on page 229, which contains an example of using aggregation.

You can design and configure a message flow that provides a similar function without using the aggregate nodes. You do this by issuing the subtask requests to another application (for example, using the HTTPRequest node) and recording the results of each request in the LocalEnvironment. After each subtask has completed, you merge the results from the LocalEnvironment in a Compute node and create the combined response message for propagating to the target application. However, all the subtasks are performed sequentially and do not provide the performance benefits of parallel operation that you can achieve by using the aggregation nodes.

3.4.3 Protocol transformation with message flows

An SOA solution uses an ESB to promote interoperability between disparate applications and systems. This interoperability often necessitates protocol and message transformation, so that applications can communicate (Figure 3-8 on page 36).

WebSphere Message Broker provides support for a variety of transport protocols for both inbound and outbound connectivity that extends the scope of a common ESB. Protocol switching with WebSphere Message Broker essentially means that a message flow receives a message over one transport type and sends it out over another. Protocol switching is automatic. However, the message format might also need to be modified for the target system. See 3.4.4, “Message transformation with message flows” on page 39.

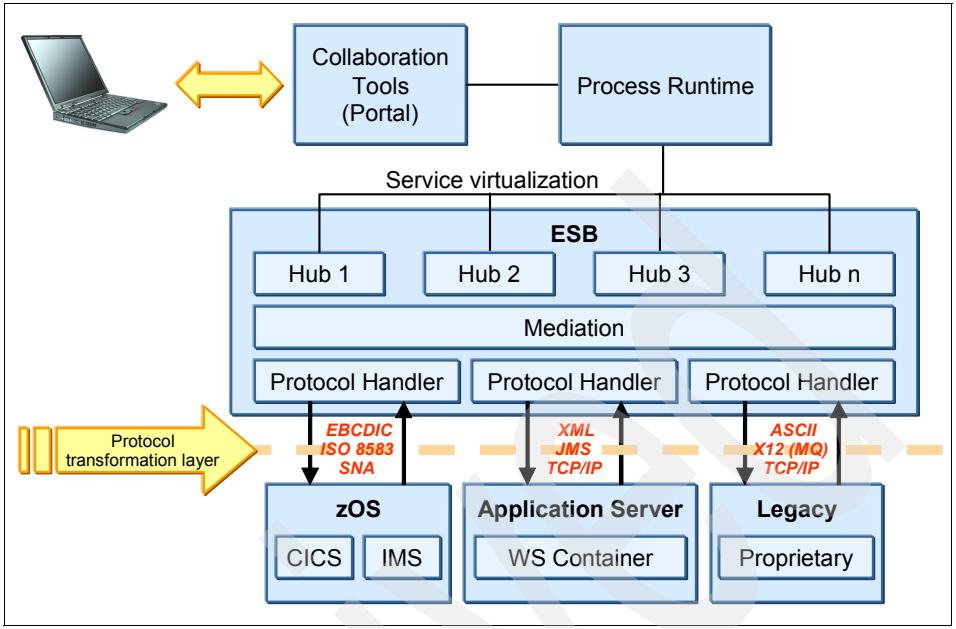


Figure 3-8 Sample protocol switching topology

WebSphere Message Broker supports the following transport types:

- WebSphere MQ
- HTTP/HTTPS
- Java Messaging Service
- Real-time and multicast
- File
- User-defined

Each of these types is summarized as follows:

- WebSphere MQ Transport

  This transport is used with the WebSphere MQ Server. This type supports WebSphere MQ applications to benefit from message routing and transformation. The message flow nodes for this transport include MQInput, MQOutput, MQReply, and Publication.

- WebSphere MQ Mobile Transport

  This transport is used exclusively with WebSphere MQ Everyplace® clients.

- WebSphere MQ Multicast Transport

  This transport is used with the publish/subscribe model.

- WebSphere MQ Real-time Transport (Multicast transport)

  This lightweight protocol is optimized for use with non-persistent messaging. It is used by JMS clients and provides high levels of message throughput. The supported nodes for this transport are Real-timeOptimizedFlow, Real-timeInput, and Publication.

- WebSphere MQ Telemetry Transport

  This transport is used by specialized applications on small footprint devices that require a low-bandwidth communication, typically for remote data acquisition and process control. A common example here is a RFID application running under WebSphere Premisses Server that need to be connected to an enterprise back-end system.

- WebSphere MQ Web services Transport

  This transport is used to transport XML messages under a message channel.

- HTTP/HTTPS Transport

  This transport uses the HTTP protocol running over TCP/IP (IP versions 4 or 6). When thinking about secure, reliable, and authorized access to a private network, HTTP/HTTPS is the more appropriate choice. This is the reason why SOAP over HTTP(S) is the most widely used transportation for Web services. The nodes provided to support this protocol are HTTPInput, HTTPReply, and HTTPRequest. All of these HTTP nodes support Secure Sockets Layer (SSL) channels.

- WebSphere Broker JMS Transport

  This transport is used to maintain JMS messages that conform to the JMS specification version 1.1. By providing native JMS interoperability, WebSphere Message Broker can act as a bridge between any combination of JMS providers, enabling seamless interaction with other vendors' message platforms. The nodes that support this protocol are JMSInput and JMSOutput. The MQJMSTransform and JMSMQTransform nodes can be used to transform the protocol between MQ and JMS.

The broker run time can even be extended to support flat file protocols, such as high availability VSAM or QSAM databases, or transaction containers, such as CICS External Call Interface (EXCI) calls. The extender packages also include SMTP handlers, FTP, SCADA-based telemetry protocols, and even plain TCP/IP sockets.

WebSphere Message Broker defines a set of extended nodes through SupportPacs that can be used to integrate applications of the enterprise information system. The CICSRequest, VSAMInput, VSAMRead, VSAMWrite, VSAMUpdate, and the VSAMDelete nodes are examples of how you can do that. An example is a back-office COBOL application running under CICS.

Figure 3-9 shows a CICSRequest node making a CICS EXCI request.

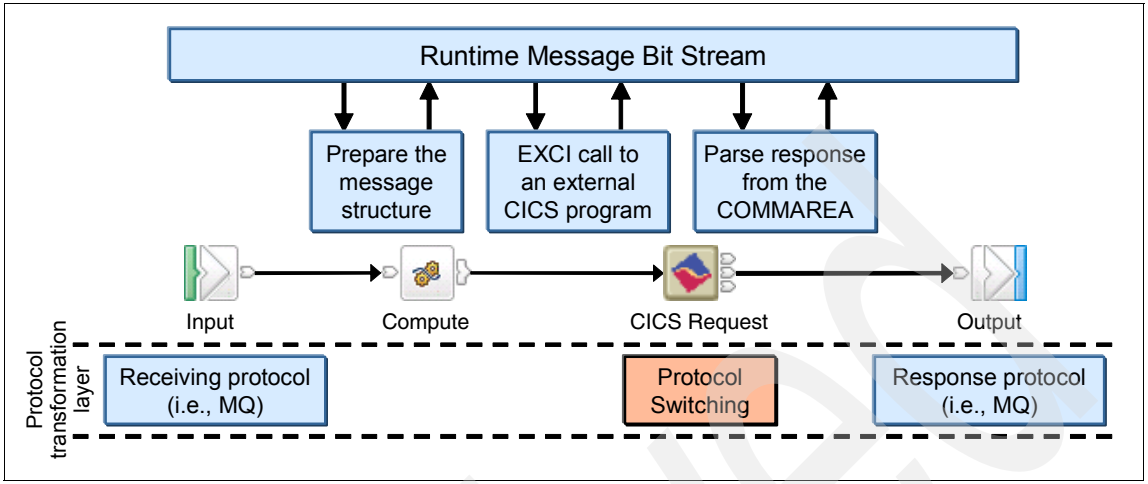


Figure 3-9 CICS protocol transformation example

The properties of the CICSRequest node specify the communication area (known as *COMMAREA*) and the CICS transaction program name that are going to be used in the CICS EXCI request. The response is returned on the same COMMAREA. After getting the response, control is passed to the node that is connected to one of the output terminals of the CICSRequest node.

SupportPacs: You can find the SupportPacs on the Web at:

`http://www-1.ibm.com/support/docview.wss?uid=swg27007197#5`

WebSphere Message Broker supports WebSphere Business Integration adapters to provide support for other transport protocols. WebSphere Message Broker also interacts with WebSphere Transformation Extender for Message Broker, which provides an additional set of connectors, and with WebSphere Message Broker File Extender, which brings support for file access to WebSphere Message Broker.

Archived

3.4.4 Message transformation with message flows

Messages have a defined structure that is known and agreed upon by the message sender and receiver. Applications typically use a combination of messages, including those that are defined by the following structures or standards:

- C and COBOL data structures
- Industry standards such as X12 or EDIFACT
- XML DTDs or schemas

Mediation flows must be able transform a message from one structure to another without drawbacks and with acceptable throughput. Messages in WebSphere Message Broker can be transformed using one of the following ways:

- ESQL
- Extensible Stylesheet Language Transformations (XSLT)
- Mapping node (graphical)
- JavaCompute node
- Runtime message set and definition

When a message arrives from a transport protocol wired to the message flow run time, the message bit stream is parsed using a physical format, such as XML. See Figure 3-10.

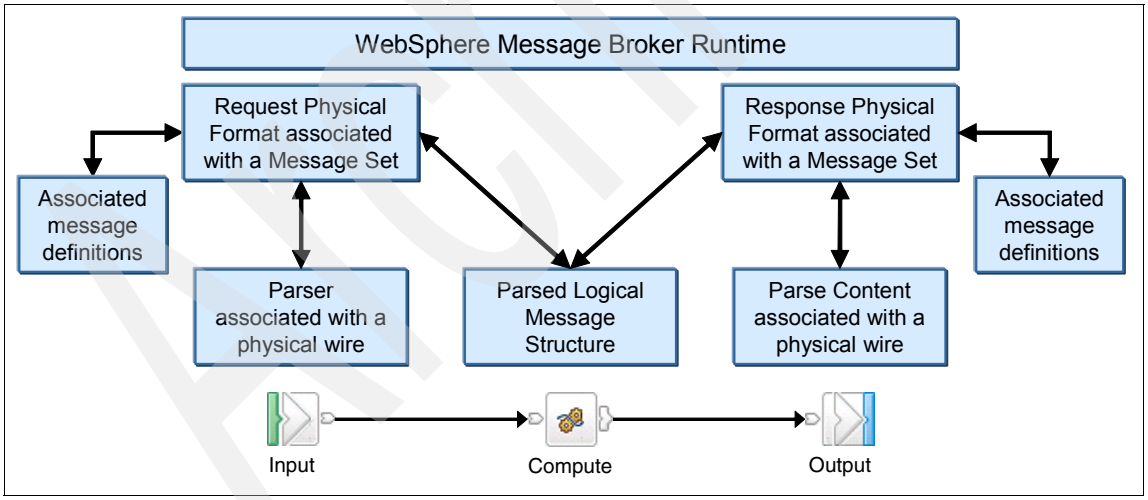


Figure 3-10 WebSphere Message Broker runtime architecture

When the message format is known, the broker parses an incoming message bit stream, using the message set and definition defined on the flow configuration, and converts it into a logical *message tree* for later manipulation. After the

message has been processed by the message flow, the broker converts the message tree back into a message bit stream. The transformation includes reformatting the message, concatenating the strings, or changing the element values.

The following physical formats are supported by the broker run time:

- XML

  This format is used as the default runtime configuration. The message structure is validated and transformed using the parser specification defined inside the message flow.

- TDS

  The tagged delimited string (TDS) physical format is designed to model messages that consist only of text strings. Examples of TDS messages are those that conform to the ACORD AL3, EDIFACT, HL7, SWIFT, and X12 standards. The TDS physical format allows a high degree of flexibility when defining message formats and is not restricted to modeling specific industry standards. Therefore, you can use the TDS format to model your own messages.

- CWF

  The Custom Wire Format (CWF) is a physical representation of a message that is composed of a number of fixed format data structures or elements, which are not separated by delimiters. The CWF physical format is typically used to describe messages that are mapped to a C structure, a COBOL copybook, or any other programming language data structure definition.

  Message set: A message set can have one or more physical formats on each XML, TDS, and CWF format.

WebSphere Message Broker typically supplies a range of parsers to parse and write message formats. Some message formats are self-defining and can be parsed without reference to a model. An example of a self-defining message format is XML. In XML, the message itself contains metadata as well as data values, enabling an XML parser to understand an XML message even if no model is available.

Most message formats, however, are not self-defining. That is, a binary message originating from a COBOL program and a SWIFT formatted text message do not contain sufficient metadata to enable a parser to understand the message. The parser must have access to a model that describes the message to parse it correctly.

Archive

Table 3-2 lists the supported parsers for WebSphere Message Broker.

Table 3-2 WebSphere Message Broker parsers

Parser name	Parser description
MRM	For modeling a wide range of messages including XML, fixed-format binary, and formatted text. This domain is usually used for enrichment of the message transformation, for example, the creation of COBOL cookbook using CWF.
XMLNSC XMLNS XML	For messages that conform to the W3C XML standard. This domain is usually used when a content validation is not required by the Message flow, that is the dynamic routing the message flow using WebSphere Service Registry and Repository.
IDOC	A parser that is implemented by using the SAP® BAPI® and the IDoc SAP format.
MIME	For handling multipart MIME messages such as SOAP with attachments or RosettaNet.
BLOB	A parser that is used with messages that do not need to be interpreted in a logical message tree. The run time internally interprets the message as a BLOB bit stream. It is commonly used with messages that do not have a well defined element tree structure, such as the ISO8583 standard.

Message models

Message modeling is a way of predefining the message formats that are used by applications. WebSphere Message Broker uses message models to automatically parse and write message formats.

A message model consists of the following components:

- Message set projects
- Message sets
- Message definition files
- Message categories

The majority of the model content is described by *message definition files*. These files use the XML schema to represent the messages. Each message definition file describes both the logical structure of the messages and the physical format or formats that describes the appearance of the message bit stream during transmission.

If you are using the MRM domain, you must provide physical format information. This information tells the MRM parser how to parse the message bit stream. If you are using one of the XML domains, physical format information is not

needed. However, if your messages are self-defining and do not require modeling, there are still advantages to modeling them:

- Enhanced parsing of XML messages

  Although XML is self-defining, without a model all data values are treated as strings. If a model is used, then the parser knows the data type of data values and can cast the data accordingly.

- Improved productivity when writing ESQL

  When you are creating ESQL programs for WebSphere Message Broker message flows, the ESQL editor can use message models to provide code completion assistance.

- Drag-and-drop message maps

  When you create message maps for WebSphere Message Broker message flows, the Mapping editor uses the message model to populate its source and target views. Without message models, you cannot use the Mapping editor.

- Runtime validation of messages

  Without a model, it is not possible for a parser to check that input and output messages have the correct structure and data values.

- Reuse of message models in whole or in part by creating new messages based on existing messages
- Automatic generation of documentation
- Provision of version control and access control for message models by storing them in a central repository

Tip: When creating output messages, the MRM parser can automatically generate the XML declaration and other XML constructs, based on options in the model, which simplifies the transformation logic. For more information about when to use MRM or XMLNS(C) domains, search on “XML messages in the MRM and XML domains” in the WebSphere Message Broker Information Center at:

`http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/ad10530_.htm`

Message transformation nodes

After a logical message structure is created, the implementing flow has full access to the parsed elements inside the message. In the following sections, we describe the nodes that can be used to transform a message within a mediation flow.

JavaCompute node

The JavaCompute node is a general-purpose programmable node. It is based on XML messages in the MRM and XML domains (J2SE™) 1.5 and supports XPath 1.1 to query on the message trees (Environment, LocalEnvironment, Properties, ExceptionList, and Message structures) in the message assembly. You can also use the Configuration Manager Proxy API to perform administrative tasks on the broker.

A JavaCompute node can be implemented as a filtering class (filtering the input content like a Compute node) or as a modifying class (changing only chunks of code). Depending on its content, a message can be propagated to one of the node's two output terminals (routing).

The Java code that is used by the node is stored in a Java project. Full debugging support is provided by using Eclipse Java debugger that is integrated with the Message Flow Visual Debugger.

Example 3-2 shows a message transformation using XPath in a JavaCompute node.

Example 3-2 Transforming a message using XPath in a JavaCompute node

```
...
// Evaluate XPath for the X12 legacy //
MbXPath xpSSN   = new MbXPath("/?$ssn[set-value($ssn)]", cliElement);
...
// define a value for the XPath variables //
xpSSN.assignVariable("ssn", ...);
...
// EVALUATE XPATH...
outMessage.evaluateXPath(xpSSN);
...
```

Examples: For an example of using a JavaCompute node for transforming a message, see Chapter 7, "Scenario: Data transformation and security" on page 165.

Using a JavaCompute node for message transformation includes the following benefits:

- J2SE 1.4.2-based runtime environment
- Familiar interface for Java developers
- A wide range of class libraries available

Archived

- The ability to operate on any MRM or XML message using MbXPath or MbElement APIs
- The ability to offload to IBM System z™ Application Assist Processor (zAAP) on z/OS®
- Transactional coordination with databases available when using MbSQLStatement (not when using Java Database Connectivity (JDBC™) or SQL for Java (SQLJ))

XMLTransformation node

The XMLTransformation node uses XSLT to transform an XML message to another format, which might or might not be XML, according to the rules provided by an Extensible Stylesheet Language (XSL) style sheet.

XSLT is a W3C-standard, XML-based language for transforming XML data. The transformation is usually from one XML document to another, but the output does not have to be in XML. XSLT relies on XPath to access and manipulate elements within a message tree. WebSphere Message Broker supports the use of style sheets that conform to XSLT 1.0 through the use of the XALAN-4J engine.

XSLT versus XSL: If an enterprise is using XML to describe business data, XSLT might be the language of choice for transforming between formats. Many XSL transformations might exist already, or developer skills might be best suited to creating transformations using this language.

WebSphere Message Broker V6 supports compiled style sheets, which improves the performance of the transformation.

The XMLTransformation node depends on the inbound message being XML. It does not have to be in one of the XML domains, but it must be a well-formed XML bit stream. The output is in the BLOB domain (Figure 3-11).

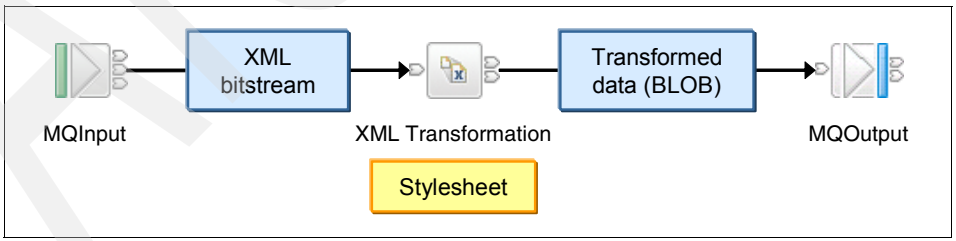


Figure 3-11 Message transformation using XSLT

Example: See Chapter 7, “Scenario: Data transformation and security” on page 165, for an example of using this node.

Use of the XMLTransformation node offers the following benefits:

- Reuse existing XSLT logic or create new XSLT logic for reuse elsewhere.
- Advanced XML tooling is available inside the Message Brokers Toolkit.
- You have the ability to offload to zAAP on z/OS.
- Incoming data must be in the XML format. A range of output formats is possible.
- Use the XSL debugger or node detailed trace option to debug transformations.

Mapping node

The Mapping node is used to construct one or more new messages and populate them with new information without writing code to it. It can modify elements from the entire logical message structure (that is, the Properties structure, LocalEnvironment structure, ExceptionList structure, and others).

The Mapping node uses at least two message set definitions to map and change message content. One defines the receiving message structure and the other defines the resulting message structure. The Mapping node provides a graphical way of mapping between fields in an input message or database, to one or more output messages or database tables.

Note: The source and target must be always defined as a database schema or MRM message set definition. The Mapping node cannot operate on other message domains.

Mapping nodes are not as rich in function as using Java or ESQL. However Mapping nodes offer the following benefits:

- It has an accessible graphical interface.
- It is suitable for someone who is not an experienced Java or ESQL developer.
- It can produce rapid results at development time.
- Input and output messages must be defined by a Message Set or database schema.

Archived

ResetContentDescriptor node

Use the ResetContentDescriptor node to request that a message be re-parsed by a different parser. If the new parser is MRM, you can also specify a different message template (message set, type, and format). This node does not re-parse the message, but the properties that you set for this node determine how the message is parsed when it is next re-parsed by the message flow.

Message format conversion: This node does not convert the message from one format to another. For example, if the incoming message has a message format of XML and the outgoing message format is CWF, the ResetContentDescriptor node does not do any reformatting. It invokes the parser to recreate the bit stream of the incoming XML message, which retains the XML tags in the message. When the message is re-parsed by a subsequent node, the XML tags are invalid, and the re-parse fails.

The ResetContentDescriptor node offers the following benefits:

- It can parse the message structure without changing its content.
- It can parse the messages to all the domains defined inside the run time.

Compute node

Use the Compute node to construct one or more new output messages. These output messages might be created by modifying the information that is provided in the input message. Alternatively, the output messages might be created by using only new information, which might (or might not) be taken from a database. Elements of the input message (for example, headers, header fields, and body data), its associated environment, and its exception list can be used to create the new output message.

The messages are created using ESQL. As with the JavaCompute node, the Compute node must also copy the input structure to the output message structure. The input message cannot be modified during the node execution. Therefore, all changes should be done on the output message structure (Example 3-3).

Example 3-3 Simple Compute node code

```
CREATE COMPUTE MODULE Test_Flow_1_Compute
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      -- CALL CopyMessageHeaders();
      -- CALL CopyEntireMessage();
      RETURN TRUE;
   END;
```

```
   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;

   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

The Compute node offers the following benefits:

- It is easy to manipulate the element tree structure.
- It rapidly changes SQL code to the Output message structure.
- It easily transforms messages from one format to another.
- It has five output terminals to route messages to different endpoints.

> **Example:** Most of these transformation nodes are demonstrated in Chapter 7, “Scenario: Data transformation and security” on page 165.

3.4.5 Message enrichment with message flows

Message enrichment involves taking an in-flight message and changing it in some way before sending it to its destination. The following main sources of data are used to enrich a message:

- Databases
- MQ queues
- Broker properties and environment values
- Flat files

Database content to enrich messages

You can use the following built-in nodes to retrieve data from a supported database and enrich an in-flight message:

- Compute node

  A Compute node can be configured for database access. The ESQL code in the node is used to retrieve data from the database and add this data to a message.

Archived

- JavaCompute node

  A JavaCompute node can be configured for database access. The Java code in the node is used to retrieve data from the database and add this data to a message.

- Mapping node

  A Mapping node can be configured for database access. A message mapping file is used to map values that are taken from a database to the message.

You can find detailed information about databases that WebSphere Message Broker supports in the “Supported databases” topic in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp?topic=/com.ibm.etools.mft.doc/ah10030_.htm

Messages on queues to enrich messages

You can use the MQGet node to enhance a message with all or part of the content of a message held on an MQ queue. The node can be included anywhere in a message flow:

1. The node receives an input message on its input terminal from the preceding node in the message flow.
2. The node issues an MQGET call to retrieve a message from a WebSphere MQ queue and builds a result message tree.
3. The node uses the input tree and the result tree to create an output tree that is then propagated to its Output, Warning, or Failure terminal, depending on the configuration of the node and the result of the MQGET operation.

Broker properties and environment values

Using the Compute or JavaCompute node, you can access the broker properties at run time and use the values to enrich a message. The broker properties have the following categories:

- Those relating to a specific node
- Those relating to nodes in general
- Those relating to a message flow
- Those relating to the execution group

You can find detailed information about broker properties that are accessible through ESQL or Java in the “Broker properties” topic in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp?topic=/com.ibm.etools.mft.doc/ak01012_.htm

Archived

SupportPac (IA06 - Get all environment variables) provides the GetEnvPlugIn node, which can be used to enrich a message with data relating to the values of the environment variables of the broker machine.

You can find further information about SupportPacs on the WebSphere Message Broker Product Support Web site at:

http://www-1.ibm.com/support/docview.wss?rs=849&uid=swg27007205

Flat files

WebSphere Message Broker has no built-in function to get messages from flat files. However, using the WebSphere Adapter for flat files in conjunction with WebSphere Message Broker, data from a flat file can be read in and the content can be used by a broker flow to enrich a message.

3.5 Deploying message flow applications

Message flow applications contain message flows and the message sets that are comprised of the message definitions used to model the messages within the message flows. Message flow applications can be deployed to the execution groups of the brokers by first adding the components of the message flow application to a *broker archive file* (bar file), and then deploying the bar file to the broker's execution group. Both of these actions can be done by using commands or with a graphical interface in the Message Brokers Toolkit.

You can create a bar file by using a graphical interface in the Message Broker Toolkit, from which you can select the components to include. If properties, such as queue names or other configurable properties, have been promoted to the bar level, then the broker archive editor offers options to configure these properties as corresponding flows are added to the bar file.

For more information about promoted properties, see the WebSphere Message Broker online help system at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/ac00640_.htm

The introduction of allowing configurable properties to be edited outside of the message flow and at the bar level during deployment time was an effort to help automate deployments to brokers. To further facilitate the automation of deployments, the **mqsicreatebar** command was added, allowing the use of scripts to create bar files.

Another extension of the automation effort came with the introduction of the **mqsiapplybaroverride** command. This command is used to set the configurable properties of the bar file, setting properties that differ between test and production environments. For more information about this command, see the "**mqsiapplybaroverride** command" topic in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/an19545_.htm

A bar file can be deployed to the broker's execution group using the Broker Administration perspective of the Message Brokers Toolkit. From the perspective, the administrator connects to the broker domain and performs a drag and drop of the bar file onto the desired broker's execution group. Bar files can also be deployed by right-clicking the bar file and selecting **Deploy File**.

The bar file can also be deployed to the broker's execution group by using a command line interface. The corresponding command is **mqsideploy**. For more information, search on the "**mqsideploy** command" topic in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/an09020_.htm

Archive

4

Connectivity options for interoperability

WebSphere Process Server provides both messaging and Web services transport bindings that are suitable for connectivity with WebSphere Message Broker. In this chapter, we help you to understand the options for connecting WebSphere Process Server to WebSphere Message Broker and explain how to implement the connections.

© Copyright IBM Corp. 2008. All rights reserved.

4.1 Messaging transport

Messaging is a loosely-coupled style of interaction among applications that involves an intermediary called a *messaging provider*. In this overview, we discuss the messaging concepts that you should know.

Messaging between applications can be of a synchronous or asynchronous interaction. With *asynchronous messaging*, applications send messages to the messaging provider and continue processing, without waiting for a response. The messaging provider delivers the message to the target application. The messaging provider offers the application quality of service levels that specify how the message delivery is managed. For example, the quality of service can determine if there is a guarantee that messages are delivered, that there are no duplicates, that a message sequence is maintained, and so forth.

With asynchronous messaging, sending and receiving applications do not need to know of each other's existence or the nature of the messages that each application understands. Each application is concerned with defining the format of the messages that it will use to communicate and with establishing access to the services that are offered by the messaging provider. The messaging provider provides services to dynamically route and optionally transform the message, so that it can be understood by the receiving application.

Although messaging is usually asynchronous, it is also possible to have *synchronous messaging*, which means that both applications are available simultaneously and communicate directly in a tightly coupled manner. The sending application waits for a response before any further processing. Synchronous messaging can be used when the performance overhead that is added by introducing an intermediary is not justified or when guaranteed delivery is not as high a priority. Response time is often an important aspect of synchronous messaging. Synchronous messaging means that updates and changes to an application must be made known to its messaging partner.

Many applications need to interact with one another in support of a business. These applications change as the business itself changes. Therefore, loosely-coupled applications that use a messaging provider can be key to enabling the exchange of data between disparate applications that are distributed across the enterprise. Not surprisingly, asynchronous messaging plays a significant role in service-oriented architecture (SOA).

WebSphere Process Server provides three bindings for messaging transport: MQ, MQ JMS, and JMS.

4.1.1 Message formats

In this section, we introduce rgw MQ and Java Message Service (JMS) message formats.

MQ message format

An MQ message consists of two parts, the headers used by WebSphere MQ and the application data as shown in Figure 4-1. The application data is private to applications.

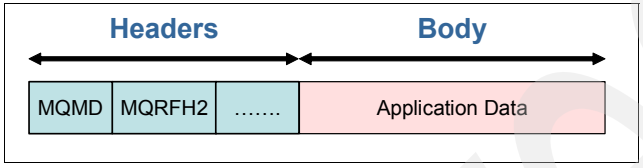


Figure 4-1 MQ message format

MQ messages must contain a message descriptor (MQMD) header. The MQMD contains the control information that accompanies the application data when a message travels between the sending and receiving applications. The MQMD has several fields, including the following fields:

- MsgId

  The MsgId is the identifier of the message that is sent to the queue. The application can supply a message identifier, or it can prompt the queue manager to generate a unique one.

- CorrelId

  The correlation identifier is normally used to provide an application with some means of matching a response with the request message. Usually the correlId of the reply is equal to the MsgId of the request message.

- ReplyToQ

  In this field, you enter the name of the queue that is set by the sending application. The response is put into this queue.

MQ messages can contain other types of headers, including MQRFH2, which carries JMS-specific data that is associated with the message content. The MQRFH2 Version 2 is an extensible header and can also carry additional information that is not directly associated with JMS.

Archived

JMS message format

JMS messages contain a header, the properties, and body as shown in Figure 4-2. Only the header is required. The properties and the body are optional.

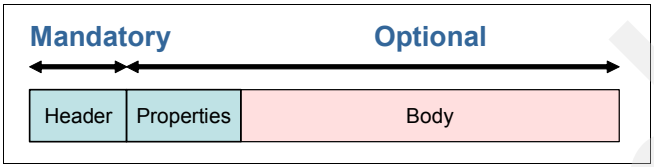


Figure 4-2 JMS message format

A JMS message header contains values that both clients and providers use to identify and route messages. The header has several fields, including the following fields:

- JMSMessageID

  The JMSMessageID is the identifier of the message that is sent to the queue or topic.

- JMSCorrelationID

  The correlation identifier is normally used to provide an application with some means of matching a response with the request message. Usually the correlation ID of the reply is equal to the message ID of the request message.

- JMSDestination

  This field represents the queue or the topic to which the message is sent.

- JMSReplyTo

  This field contains the name of the queue where the response should be put. It is set by the sending application.

JMS properties are used to support application-defined property values. Properties provide an efficient mechanism to filter application-defined messages by using message selectors.

JMS defines several message body types that cover the majority of messaging styles that are currently in use. See Table 4-1 on page 55.

Archived

Table 4-1 JMS message body types

JMS message type	Body contents type
TextMessage	java.lang.String
StreamMessage	A stream of Java primitive values. It is filled and read sequentially.
MapMessage	A set of name-value pairs, where names are strings and values are Java primitive types. The entries can be accessed sequentially or randomly by name.
BytesMessage	A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.
ObjectMessage	Serializable Java object.

4.1.2 Mapping messages to a Service Component Architecture interface

In order for a Service Component Architecture (SCA) application to communicate to an existing messaging client, certain SCA artifacts must be created. In this section, we provide guidance and steps on how to create a business object and an interface to represent a messaging client application in an SCA manner.

We also discuss imports, exports, and bindings. SCA applications use imports with messaging bindings to send messages to messaging clients. These applications use exports with messaging bindings to receive messages from messaging clients.

We discuss the following SCA artifacts and their relationship to a JMS client:

- Business objects
- Interfaces
- Imports, exports and their bindings

Business objects

Service Data Objects (SDOs) define the data flowing between SCA components. WebSphere Process Server uses *business objects*, which are enhanced SDOs to pass data between SCA components.

In order for an SCA application to communicate with an existing messaging client, it must have a business object that represents the message that is used by the messaging client.

Archive

If the message that is used by the client is in XML format, the XML schema that defines it is used as the business object. If this schema exists, it can be imported into the business integration module. The imported XML schema appears under Data Types in the module. If this schema does not already exist, a business object can be created with attributes that mirror the XML format that is used by the client.

If the client message is not in XML format and no XML schema defines it, a business object must be created with attributes that reflect logically the message format used by the JMS or MQ client. For example, if the message format is a Java bean, the business object has attributes that represent the attributes of the Java bean.

Data bindings handle the transformation of data passed in native format from a messaging system or enterprise information system (EIS) to an SDO in an SCA application.

Interfaces

An *interface* represents the input and output of a messaging client application. That is, the interface represents the interaction with the client.

An interface has *operations* that describe the messaging client application operations. An operation can be a *one-way operation*, sending a message and not expecting a response, or a *request-response operation*, sending a message and expecting a returning message.

Imports, exports and bindings

A business object represents the message, and an interface represents the interaction of the input and output of the messaging client.

Messaging client: A messaging client is considered to be any application that interacts with WebSphere Process Server using a messaging binding.

But how is the data transferred between the client and these SCA artifacts? Business objects and interfaces are found in a module, which is similar to a project container. Imports and exports define a module's external interfaces or access points. The messaging client interacts with the business objects and interfaces of an SCA application through the import and export components.

Import components identify services outside of a module that can be called from within the module. When a message is put on a destination queue by the SCA application, the associated messaging client processes the event.

Archive

With *export components*, SCA components can listen to requests from external clients. When a message is put on a destination, the associated SCA export processes the event invoking the targeted SCA component.

Example: The scenario in Chapter 9, “Scenario: Asynchronous callback” on page 275, is an example of the collaboration between imports and exports. In this scenario, an import is used to send a message to a service. An export is used to receive the reply.

Binding information is required for import and export components to define the transport to be used for sending messages. The available messaging bindings include JMS, MQ, and MQ JMS bindings.

Table 4-2 shows the relationship between bindings, messages in the context of imports and exports, and types of operations in interfaces.

Table 4-2 Relationship of bindings, import and export messages, and types of operations in an interface

JMS, MQ or MQ JMS bindings	One-way operation in an interface	Request-response operation in an interface
Import binding	Client receives a message	Client receives and sends a message
Export binding	Client sends a message	Client sends and receives a message
Note: In the scenarios in this book, the client is WebSphere Message Broker.		

Data binding

To convert data that is external to WebSphere Process Server to and from data objects, some transport bindings require a data binding to be specified. The *data binding* translates between the data provided by the transport binding as part of a message and a data object. Figure 4-3 on page 58 shows how data bindings work to translate data between SCA import and export components and WebSphere MQ.

Archived

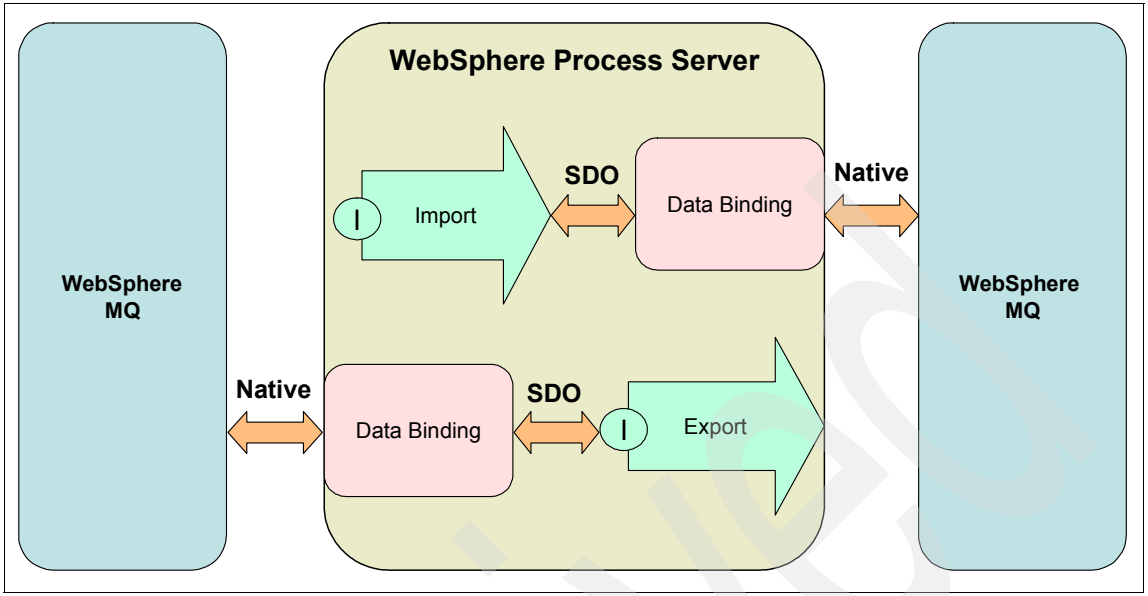


Figure 4-3 Data bindings

Typically, some pre-supplied data bindings are available. However, if these do not meet your needs, you can write a custom data binding that conforms to the `commonj.connector.runtime.DataBinding` interface. In fact, some data bindings conform to a more specific interface that extends this one and that provides methods to get and set transport-specific data.

The DataBinding interface specifies two methods, `getDataObject()` and `setDataObject()`. The first method is used by the WebSphere Process Server to get the data object that should be used within the module and is typically used when the data binding is specified on an export. The second method is used to pass the data object into the data binding to be translated into external data and is typically used on an import. These roles are swapped when a reply message is handled in a request-response operation.

4.1.3 MQ binding

Example: An example of an MQ binding is provided in Chapter 9, “Scenario: Asynchronous callback” on page 275.

IBM WebSphere MQ is a popular middleware set of products that provide a well-known set of messaging communications between applications, which can

themselves be on many dissimilar systems. An MQ binding provides a way to communicate directly with WebSphere MQ using both exports (incoming) and imports (outgoing). An MQ binding communicates with WebSphere MQ using native MQ messages.

An MQ binding is new for WebSphere Process Server 6.0.2, and is designed to provide a direct replacement for the WebSphere MQ Link functionality that is still available as part of the WebSphere Application Server platform. It is more straightforward to configure and is generally recommended over MQ Link. An MQ binding also provides a convenient way to communicate with WebSphere Message Broker via WebSphere MQ.

In the binding settings, receive and send destination names are specified. These correspond to destinations on a WebSphere MQ queue manager. Such details as the queue manager name and the host name for the queue manager must also be specified.

In this section, we show how a WebSphere MQ message maps to SCA artifacts. That is, we show how a message maps to a business object and how input and output from a WebSphere MQ client maps to an interface's operations.

Restrictions: An MQ binding has the following restrictions:

- The publish/subscribe method of distributing messages is not currently supported by the MQ binding, although WebSphere MQ itself supports publish-subscribe.
- MQ client connections are the only type of connections that are supported.
- Channel compression, which compresses the data that flows on a WebSphere MQ channel to improve performance, is not supported.
- WebSphere MQ headers have specific data types and parsers for them determine the data type of each field in the header. MQ bindings currently provide support for the MQRFH and MQRFH2 header.

MQ import and data bindings

Four *MQ native body data bindings*, which serialize between the SDO and the MQ message body, are supplied with WebSphere Process Server. In addition, you can use a user-supplied data binding.

Figure 4-4 shows the different data bindings that are provided by WebSphere Integration Developer for an MQ binding. We explain them in the list that follows.

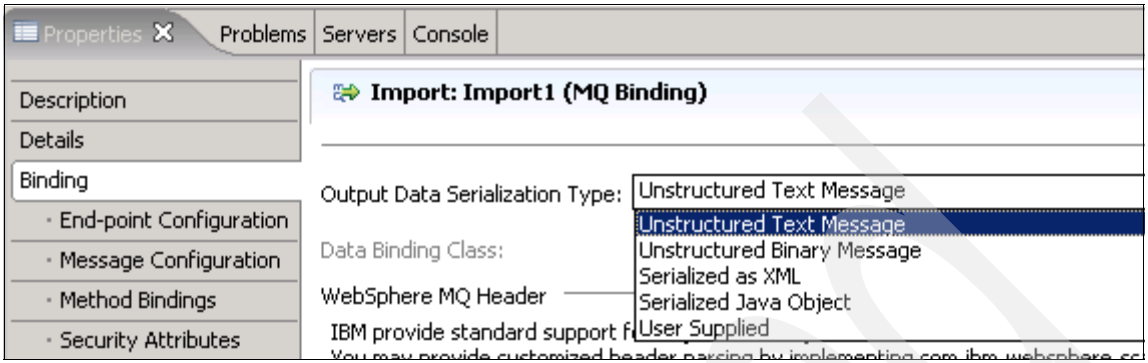


Figure 4-4 Data binding options for MQ binding

- *Unstructured Text Message* converts an SDO to and from a string. The SDO is expected to be a JMSTextBody type of business object.
- *Unstructured Binary Message* converts an SDO to and from a string. The SDO is expected to be a JMSBytesBody type of business object.
- *Serialized as XML* serializes SDO to XML or deserializes XML to SDO. The SDO can be of any type.
- *Serialized Java Object* serializes and deserializes an SDO to and from a Java object. The SDO can be of any type.

MQ bindings versus MQ JMS or JMS bindings: Do not be confused by the use of the JMSTextBody and JMSBytesBody business objects with the "Unstructured Text Message" and "Unstructured Binary Message" bindings. The MQ native bindings have no relationship with the MQ JMS or JMS bindings. These business objects are simply reused for this purpose because they have the same structure, which is a single string in the case of JMSTextBody or a single hexBinary in the case of JMSBytesBody.

JMSTextBody and JMSBytesBody have the same message contents as listed in Table 4-1 on page 55.

The alternative is to provide your own custom data binding, in which case you can select **User Supplied**. The implementation must conform to the MQ data binding interface. That is, you must implement the interface `com.ibm.websphere.sca.mq.data.MQBodyDataBinding`, which extends the interface `commonj.connector.runtime.DataBinding`.

WebSphere Process Server provides complete support for the MQMD, MQRFH, and MQRFH2 headers, providing interoperability with JMS and WebSphere Message Broker. Partial support is provided for messages with a format that begins with MQH, for example messages with a format for the form MQHxxxxx. These headers are exposed as binary data in SCA.

Figure 4-5 illustrates the different MQ native data bindings, the MQ message structure (generated from the those bindings), and the recommended message domain (parsers) to be used by WebSphere Message Broker.

Serialized as XML data binding: The Serialized as XML data binding type generates an XML message. This data binding is used in Chapter 9, “Scenario: Asynchronous callback” on page 275.

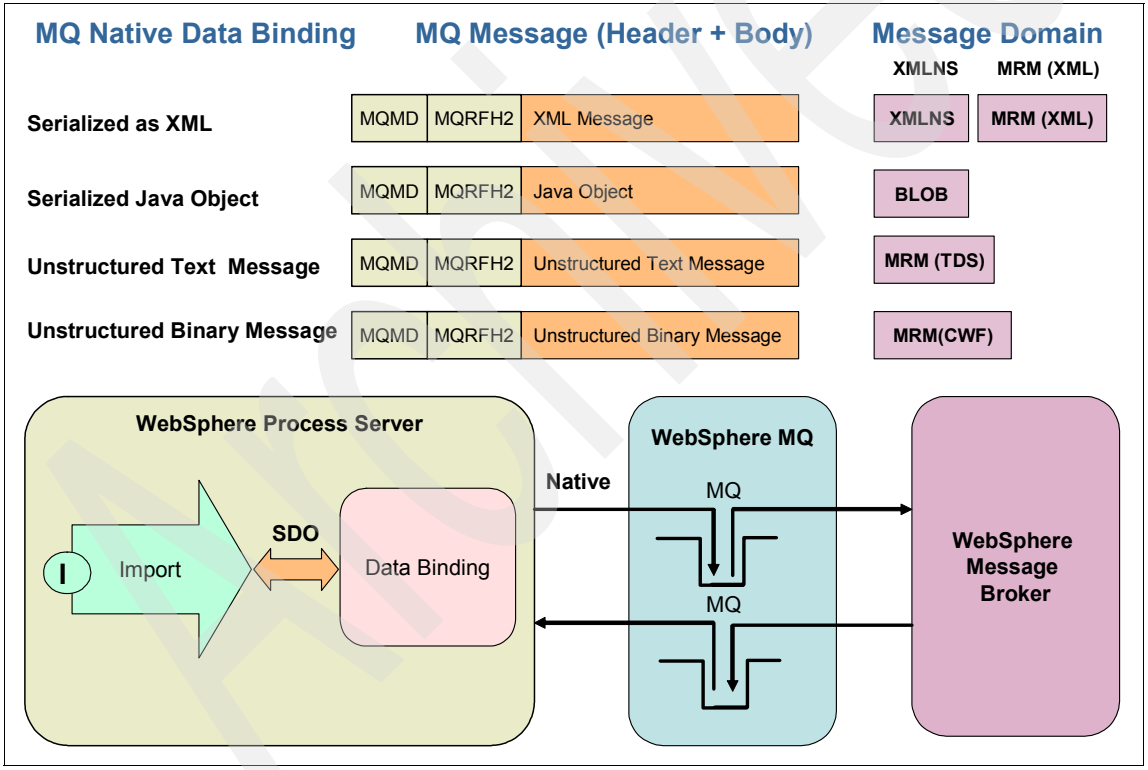


Figure 4-5 Relationship of MQ data binding to the message domain

Example

Using the Unstructured Text Message data binding stores the SDO as a string in the MQ message body, which can be parsed by the MRM parser in WebSphere Message Broker using TDS as the physical format. To accomplish this scenario:

1. To use Unstructured Text Message as the data binding, the SDO must be of JMSTextBody type. Include this type in an SCA module using WebSphere Integration Developer:

   a. Right-click the business integration module and select **Open Dependencies**.

   b. In the window that opens (Figure 4-6), under Predefined Resources, select **Schema for simple JMS Data Bindings**.

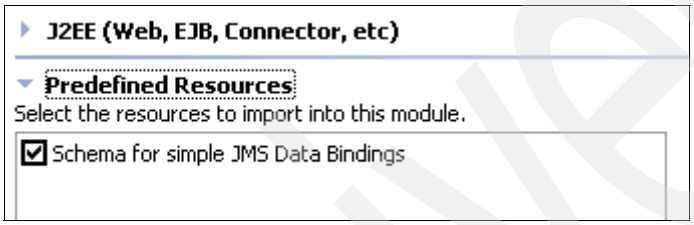


Figure 4-6 Adding a schema for JMS data bindings to the workspace

   c. Save your work and close the dependencies editor.

   d. In the navigation area, expand **Data Types**. The new business objects, including JMSTextBody, are displayed (Figure 4-7).

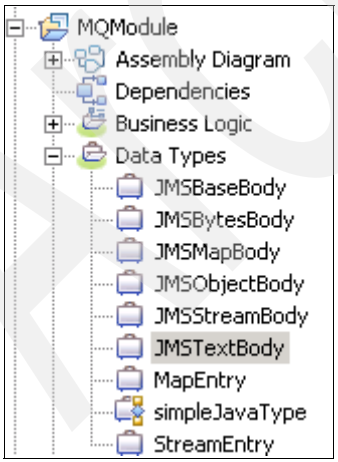


Figure 4-7 JMS data types

Archived

If you open JMSTextBody, you will see that it has only one attribute, which is `value` of type `String` (Figure 4-8).

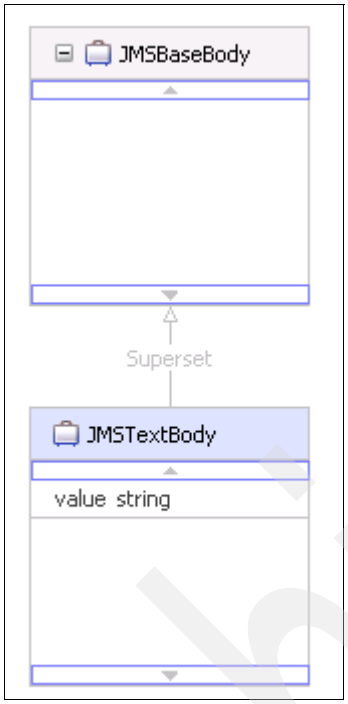


Figure 4-8 JMSTextBody data type

2. Create an interface with an operation that takes JMSTextBody as its input data type as shown in Figure 4-9.

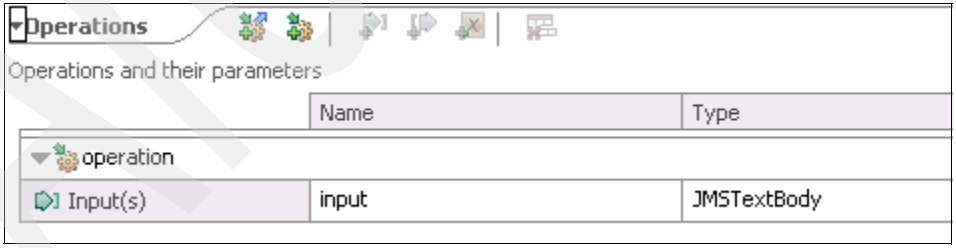


Figure 4-9 Interface operation using JMSTextBody as the input type

Archived

3. Using the Message Brokers Toolkit, create a message set and create a complex type that has TDS as its physical format. For example, the complex type can be `Name` that has `First` and `Last` as its elements. In addition, create a message of this complex type. Figure 4-10 shows an example of a complex message type called `NameOB`.

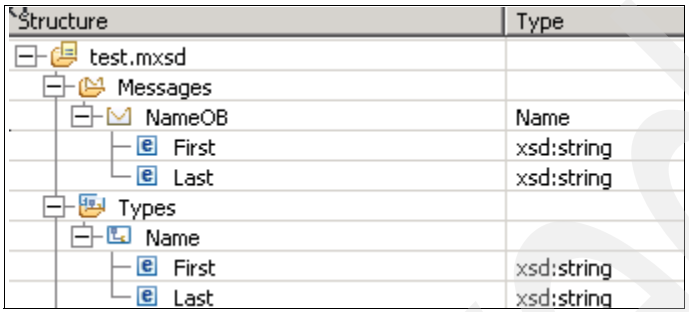


Figure 4-10 Message definition file

4. In the message flow, click **Input Message Parsing**, and define the properties of the MQInput node as shown in Figure 4-11.

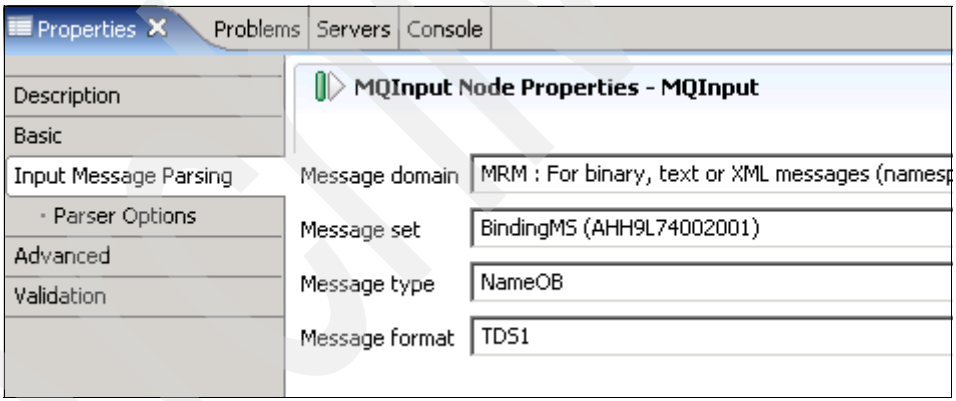


Figure 4-11 MQInput node parsing options

5. Send a message from WebSphere Process Server to WebSphere Message Broker. For example, the message can be `Mary;Mendez`.

Archived

Message IDs and correlation IDs

If the MQ import interface has a request-response operation, then the request message will have an ID. Figure 4-12 shows the Request Message ID Options field.

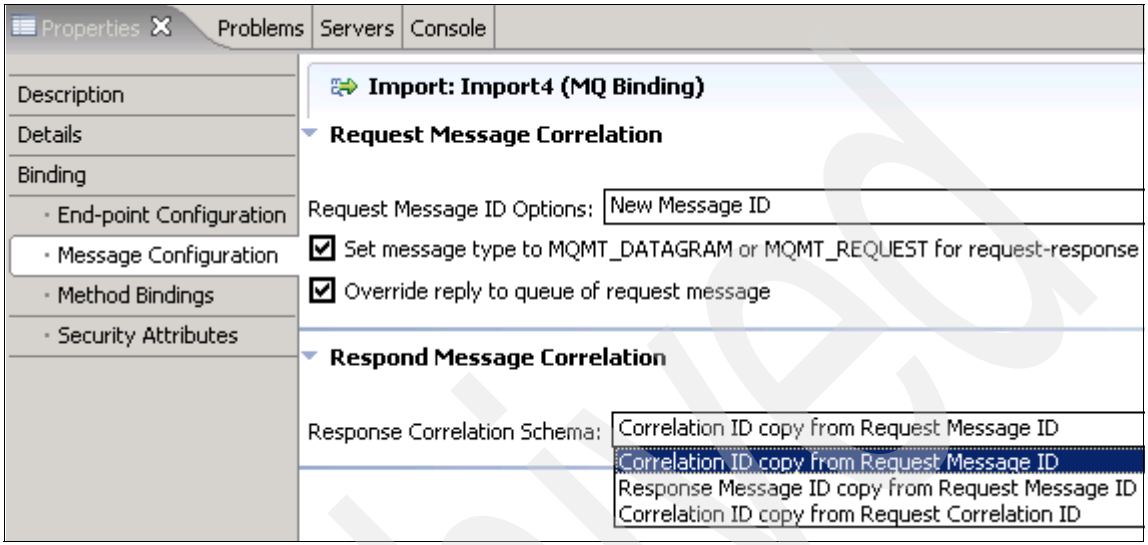


Figure 4-12 Request and response message correlation settings

The Request Message ID Options field has the following options:

- *New Message ID* lets the queue manager select a unique message ID for the request (default).
- *Copy from SCA message* sets the message ID to be the same as the ID carried in the headers of the SCA message. If no ID is found, one is created.

As for the response message, you must set the Response Correlation Schema field in order to correlate the response message to the request message.

Important: The Response Correlation Schema field indicates how the MQ import expects the receiving MQ application (message flow deployed on WebSphere Message Broker) to set the correlation ID. If the MQ application does not do this, the MQ import cannot correlate requests and replies.

Archived

Figure 4-12 on page 65 shows different correlation schemes, which are explained as follows:

- *Correlation ID copy from Request Message ID* (default) means that the correlation ID of the response message will be set to the message ID of the request message.
- *Response Message ID copy from Request Message ID* means that the response message will have a message ID field set to the message ID of the request message.
- *Correlation ID copy from Request Correlation ID* means that the response message will have a correlation ID set to the correlation ID of the request message.

MQ export and data bindings

Data bindings for MQ on an export component are the same MQ native data bindings as with MQ on an import component.

MQ function selector

Some mechanism must be available that can determine which SCA operation an incoming message corresponds with. An MQ message contains only data and carries no indication about the target operation for which it is intended. Therefore, we must map an MQ message to an operation of the targeted service interface. The MQ export listens to a particular destination. When a message arrives on that destination, the function selector determines the target operation.

Different types of function selectors are associated with an MQ export that can be set in the MQ export binding. See Figure 4-13.

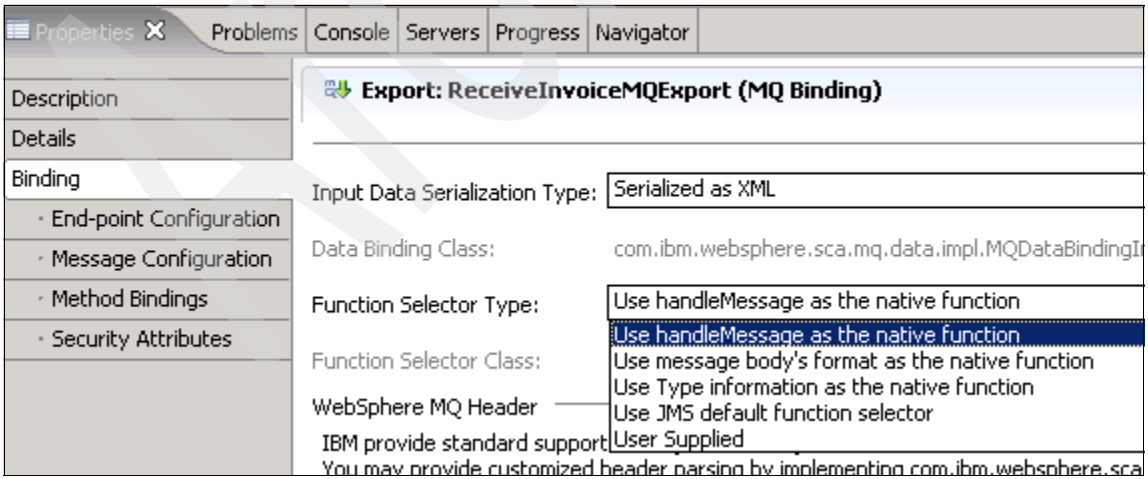


Figure 4-13 MQ binding Function Selector Type

Function Selector Type has the following options:

- *Use handleMessage as the native function* (default) returns handleMessage as a value. The value is mapped using the export's method bindings to the name of an operation on the interface. The native method is assumed to be handleMessage from the SCA application's perspective. The SCA application does not need to understand the internal representation of the MQ message, but only how it drives the current operation on the interface. This option is the recommended selection for most applications. It might be necessary to edit one of the method bindings to bind an operation to the handleMessage native function.
- *Use message body's format as the native function* finds the Format field of the last header and returns that field as a String.
- *Use Type information as the native function* creates a method in your export binding by retrieving a URL that contains the Msd, Set, Type and Format properties that are found in the MQRFH2 header. The name of the method is the interface operation name.
- *Use JMS default function selector* means that the JMS function selector is used to create a method in your export binding (see "JMS function selector" on page 71). The name of the method is the interface operation name. This function selector reads the native operation from the TargetFunctionName property of the folder of an MQRFH2 header.
- *User supplied* means that you create your own function selector by implementing the `com.ibm.websphere.sca.mq.selector.MQFunctionSelector` class.

4.1.4 JMS bindings

Examples: JMS bindings are used in the scenario in Chapter 10, "Scenario: Connectivity to third-party JMS providers" on page 311.

The JMS bindings allow inbound JMS communication (via exports) and outbound communication (via imports). The JMS bindings in WebSphere Process Server support "plain" JMS messages, not SOAP. WebSphere Process Server is based on WebSphere Application Server and gives you the following options for JMS connectivity with WebSphere Message Broker:

- The default messaging provider

  This option is included in WebSphere Application Server. The transport mechanism for this provider is the service integration bus within WebSphere Application Server.

- The WebSphere MQ messaging provider

  You can use an installed WebSphere MQ as the messaging provider.

Point-to-point (via *queues*) and publish/subscribe (messages sent or received on a *topic*) communication patterns are supported by both providers. Standard JMS terminology collectively refers to queues and topics as *destinations*. Destinations are defined to the Java Naming and Directory Interface™ (JNDI) namespace, and the bindings designate the JNDI name for the destination.

Point-to-point request-response operations in a business integration module are supported by specifying the following information:

- For JMS bindings on an export (incoming operations), a receive queue for incoming messages (request) and a send queue for outgoing messages (reply)
- For JMS bindings on an import (outgoing operations), a send queue for outgoing messages (request) and a receive queue for incoming messages (reply)

One-way operations: One-way operations require only one destination to be specified. Publish/subscribe can only be used with one-way operations.

JMS import data bindings

Eight *JMS body data bindings* , which serialize between SDO and the JMS message body, are supplied with WebSphere Process Server. Figure 4-14 shows the different data bindings that are provided by WebSphere Integration Developer.

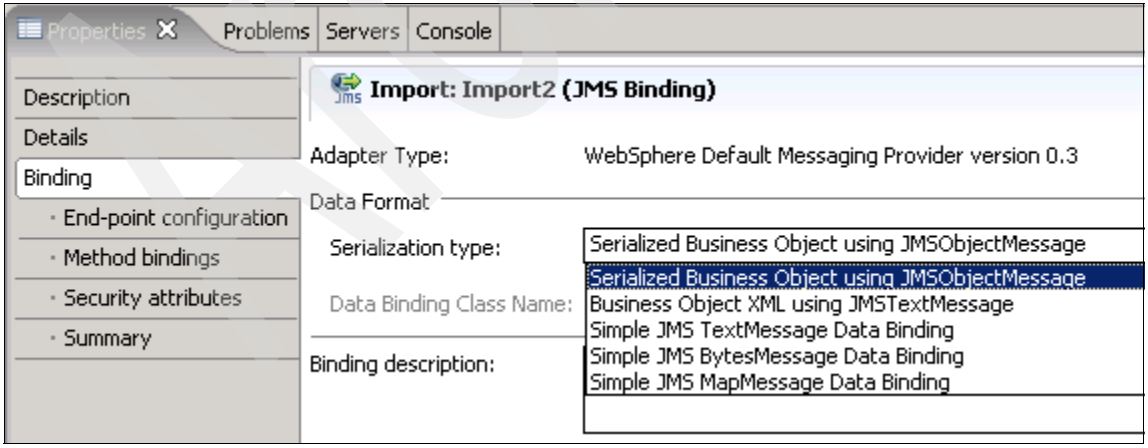


Figure 4-14 JMS data bindings

WebSphere Integration Developer offers the following data bindings:

- *Serialized Business Object using JMSObjectMessage* serializes (or deserializes) the business object whose underlying implementation is an SDO, using the standard Java serialization mechanism, and places it in a JMS object message. It is typically only useful when communicating with another JMS import or export that uses the same data binding.
- *Business Object XML Using JMS TextMessage* serializes the business object to and from XML and uses a JMS text message to communicate with the JMS client. Your wrapped data objects must be a complex type. The objects cannot be a simple type.
- *Simple JMS TextMessage Data Binding* sends the JMSTextBody business object in a TextMessage message body to the client. The message body type is a String.
- *Simple JMS BytesMessage Data Binding* sends the JMSBytesBody business object in a BytesMessage message body to the client. The message body type is a byte array.
- *Simple JMS MapMessage Data Binding* sends the JMSMapBody business object in a MapMessage message body to the client. The message body type is a set of name/value pairs that can be addressed by name. The data type of the value is a Java primitive.
- *Simple JMS StreamMessage Data Binding* sends the JMSStreamBody business object in a StreamMessage message body to the client. The message body type is a stream of Java primitives.
- *Simple JMS ObjectMessage Data Binding* sends the JMSObjectBody business object in an ObjectMessage message body to the client. The message body type is a serialized Java object.
- *Simple JMS Message Data Binding* does not have a message body. It can be used to notify the JMS client of an event or to send simple data in the message headers or properties.

Important: Several default serialization types are provided. For serialization types beginning with *Simple JMS*, you must have previously created the business objects for simple JMS binding as described in “Example” on page 62.

If the JMS client is expecting a different serialization style, select **User Defined** and specify the appropriate data binding implementation.

Figure 4-15 illustrates the different JMS data bindings, the JMS message structure (generated from the those bindings), and the recommended message domain (parsers) to be used by WebSphere Message Broker.

Business Object XML using JMS TextMessage: The Business Object XML using JMS TextMessage data binding type is used in Chapter 10, “Scenario: Connectivity to third-party JMS providers” on page 311.

For Business Object XML using JMS TextMessage data binding, you can use the XML message domain if the business objects are not in the namespace, which is not recommended.

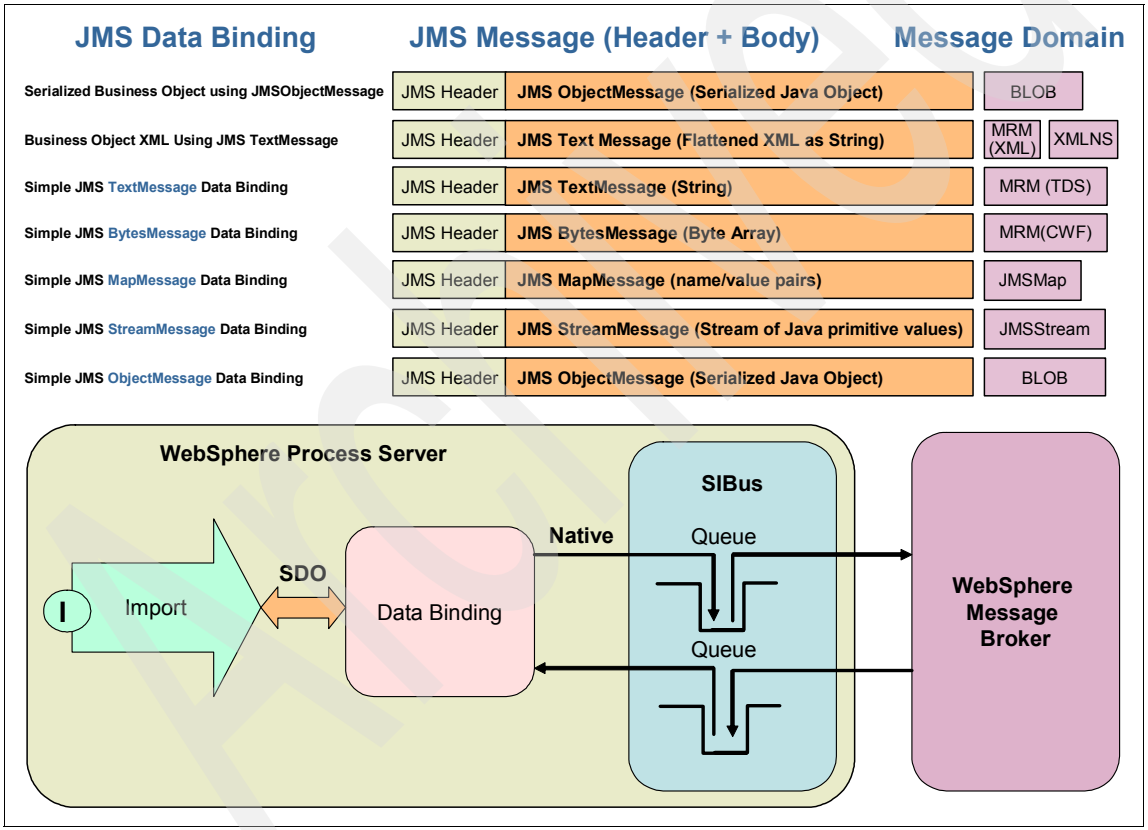


Figure 4-15 JMS data binding to message domain relation

JMSMessageID and JMSCorrelationID: For request-response operations, the JMSCorrelationID of the response message must contain the JMSMessageID of the request message. This JMSMessageID is used to retrieve callback information and to return the response to the calling service. When an SCA JMS import is used to invoke a JMS client (message flow deployed on WebSphere Message Broker), the JMS client must take the JMSMessageID from the request message and set it into the JMSCorrelationID of the response message.

JMS export data bindings

JMS bindings on export components have the same data binding options as the JMS import components.

JMS function selector

Some mechanism must be available that can determine which of the possible JMS export components the message should be delivered to and which SCA operation the message corresponds with. A JMS message contains only data, and the message might contain an indication about the target operation, but there is no standard way to retrieve this information. Therefore, we must map a certain message to a certain operation of the targeted service interface. The JMS export listens to a particular destination. When a message arrives at that destination, the function selector determines the target operation.

In the export binding, the method binding has a Native method attribute (Figure 4-16), which contains an identifier that represents a JMS message.

Figure 4-16 Export JMS binding settings

By default, the value of the Native method field is set to the interface operation name when the export is created. The export provides a default function selector that returns the value of JMS String Property TargetFunctionName from the header. TargetFunctionName must be set by the sending party.

Consider a case where an import sends messages to a client, and an export receives the reply from the client. The default behavior is for the import binding to

set the JMS header property TargetFunctionName to the name of the interface operation name as shown in Figure 4-17. The default function selector used by the export extracts the TargetFunctionName property from the JMS header to correctly identify the incoming message.

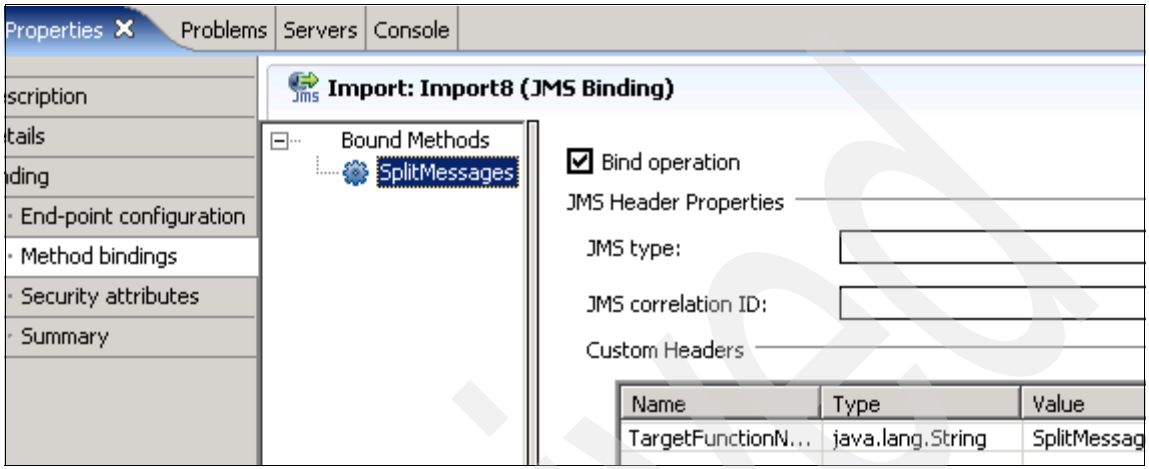


Figure 4-17 Import settings

Figure 4-18 summarizes this scenario.

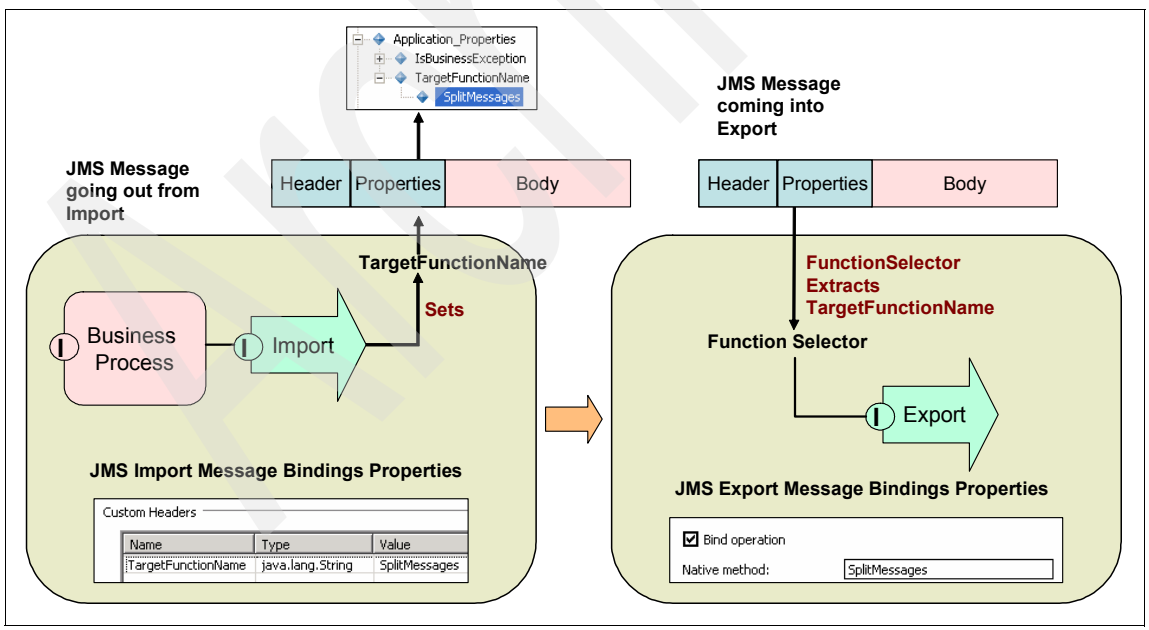


Figure 4-18 Function selector

4.1.5 MQ JMS binding

Examples: MQ JMS bindings are used in the following scenarios:

- Chapter 5, “Scenario: Message splitting and routing” on page 79
- Chapter 6, “Scenario: WebSphere Message Broker acting as a transaction manager” on page 115

MQ JMS is a set of Java classes that enables JMS applications using WebSphere MQ as the messaging provider.

The JMS API is an open standard API for sending and receiving messages. It allows components based on J2EE to create, send, receive, and read messages. Using this standard makes your MQ applications more portable and allows your MQ applications to use different JMS implementations for different circumstances.

Distribution of messages: In MQ JMS, both the point-to-point and publish/subscribe distribution of messages are supported unlike MQ, which is limited to point-to-point.

If you are using MQ JMS binding, you can choose one of the following options to define the destination:

- Specify the JNDI name for a pre-configured messaging provider resource, for example, the JNDI name for the queue or topic and the JNDI name for the connection factory.
- Specify properties that are required to configure a new messaging provider resource, meaning, the WebSphere queue manager and WebSphere queue name.

If the import or export component has an interface with a one-way operation, then both queues (point-to-point) and topics (publish/subscribe) are supported by the MQ JMS binding. For request-response operations, only point-to-point is supported.

In WebSphere Message Broker, you can use either MQ or JMS input and output nodes as the corresponding message point. No special configuration is necessary on the MQ JMS binding that makes it necessary for you to use one or the other.

Archived

Message correlation

If the MQ JMS import interface has a request-response operation, then you must set the Response Correlation Schema field in order to correlate the response message to the request message. Figure 4-19 shows the different correlation schemes:

- *Request Message ID to Correlation ID* adds a request ID to the request message. It is expected that the reply copies the request ID to the correlation ID field of the response message, so that the requestor can correlate the reply message with the request message.
- *Request Correlation ID to Correlation ID* adds the correlation ID to the request message. It is expected that the reply copies the request correlation ID to the correlation ID field of the response message so that the requestor can correlate the reply message with the request message.

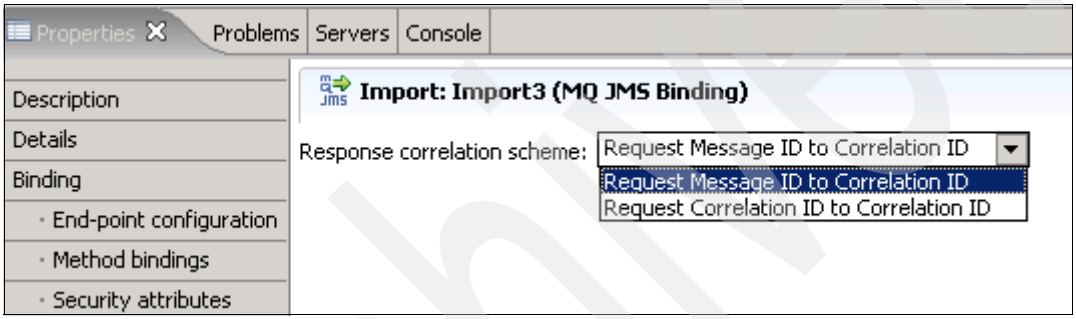


Figure 4-19 Response correlation schema

JMS binding versus MQ JMS binding: The JMS binding is used if you are using the default messaging provider, and the MQ JMS binding is used if you are using the WebSphere MQ provider.

MQ JMS data bindings

Export and import components with an MQ JMS binding have the same data binding options that are provided by the JMS binding. See “JMS import data bindings” on page 68.

Function selector

The function selector mechanism for the MQ JMS binding is the same as for the JMS binding.

4.2 Web service bindings

Examples: The following scenarios contain examples of Web service bindings:

- SOAP/HTTPS is used in Chapter 7, “Scenario: Data transformation and security” on page 165.
- SOAP/HTTP is used in Chapter 8, “Scenario: Aggregation” on page 229.

Web service bindings allow the call-out of SOAP-based Web services (via imports), as well as exposing SOAP-based Web services interfaces (via exports). Both SOAP/HTTP and SOAP/JMS are supported.

Both the HTTP and JMS transport types are presented as one binding type in WebSphere Integration Developer. However, when you generate a Web service binding, you are asked which transport type you want to use. SOAP/HTTP is more common than SOAP/JMS.

Because SOAP is well-defined and maps cleanly into the SCA and SDO model, no requirement (or facility) is necessary to provide any form of data binding, either for imports or exports. The Web service bindings support the well-known encoding styles: document/literal, doc-lit-wrapped, RPC-literal, and RPC-encoded.

Binding compliance: The bindings are compliant with the WS-I Basic Profile 1.1. For more information, see the Web Services Interoperability Organization (WS-I) Web site at:

http://www.ws-i.org/Profiles/BasicProfile-1.1.html

SOAP message format

A SOAP message is an envelope that contains zero or more headers and exactly one body as shown in Figure 4-20.

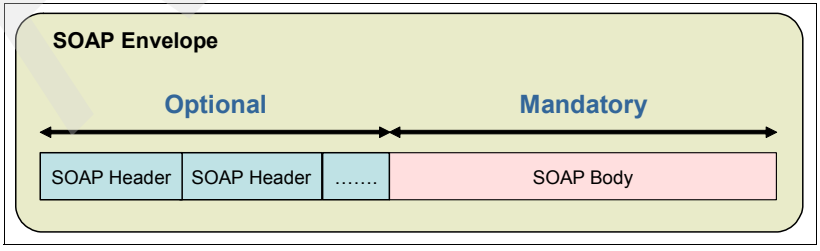


Figure 4-20 SOAP message format

The message has the following components:

SOAP Envelope The top element of the XML document that provides a container for SOAP headers and body

SOAP Header Contains control information, such as quality of service attributes

SOAP Body Contains the message identification and its parameters

4.3 Connectivity option summary

The binding type that you choose for WebSphere Process Server depends on the input type of the WebSphere Message Broker flow. See Table 4-3.

Table 4-3 Connectivity options

Import binding	WebSphere Message Broker flow has...
MQ binding	MQInput node
JMS binding	JMSInput node
MQ JMS binding	MQInput node or JMSInput node
SOAP/HTTP Web service binding	HTTPInput node
SOAP/JMS Web service binding	JMSInput node

Archived

Part 2

Scenarios

In this part, we provide several scenarios to illustrate the integration of WebSphere Process Server and WebSphere Message Broker. This part includes the following chapters:

- Chapter 5, “Scenario: Message splitting and routing” on page 79
- Chapter 6, “Scenario: WebSphere Message Broker acting as a transaction manager” on page 115
- Chapter 7, “Scenario: Data transformation and security” on page 165
- Chapter 8, “Scenario: Aggregation” on page 229
- Chapter 9, “Scenario: Asynchronous callback” on page 275
- Chapter 10, “Scenario: Connectivity to third-party JMS providers” on page 311

© Copyright IBM Corp. 2008. All rights reserved.

Archived

5

Scenario: Message splitting and routing

In the scenario presented in this chapter, we demonstrate the following concepts:

- Message manipulation

  The message takes a single message that consists of multiple logical parts and splits it into multiple messages that each consist of one logical part.

- Content-based routing using a database

  Messages are routed to a destination that is determined by querying routing rules for the message type. The routing rules are stored in an external relational database.

- Message transformation

  The message is delivered to WebSphere Message Broker as a Java Message Service (JMS) message and then sent to the back-end services as an MQ message.

- Connectivity using an MQ JMS binding

  WebSphere Process Server uses an import component with an MQ JMS binding to send a message to WebSphere Message Broker. A JMSInput node is used in Message Broker to receive the message from WebSphere Process Server.

© Copyright IBM Corp. 2008. All rights reserved.

- One-way interaction pattern

  A one-way request is sent from WebSphere Process Server to WebSphere Message Broker.

5.1 Scenario overview

The business process in this scenario is responsible for processing a list of orders that are contained in a single JMS message. The list is sent to WebSphere Message Broker where a message flow extracts each order and routes it to the correct supplier based on a supplier ID in the order.

Figure 5-1 shows a high-level overview of this scenario.

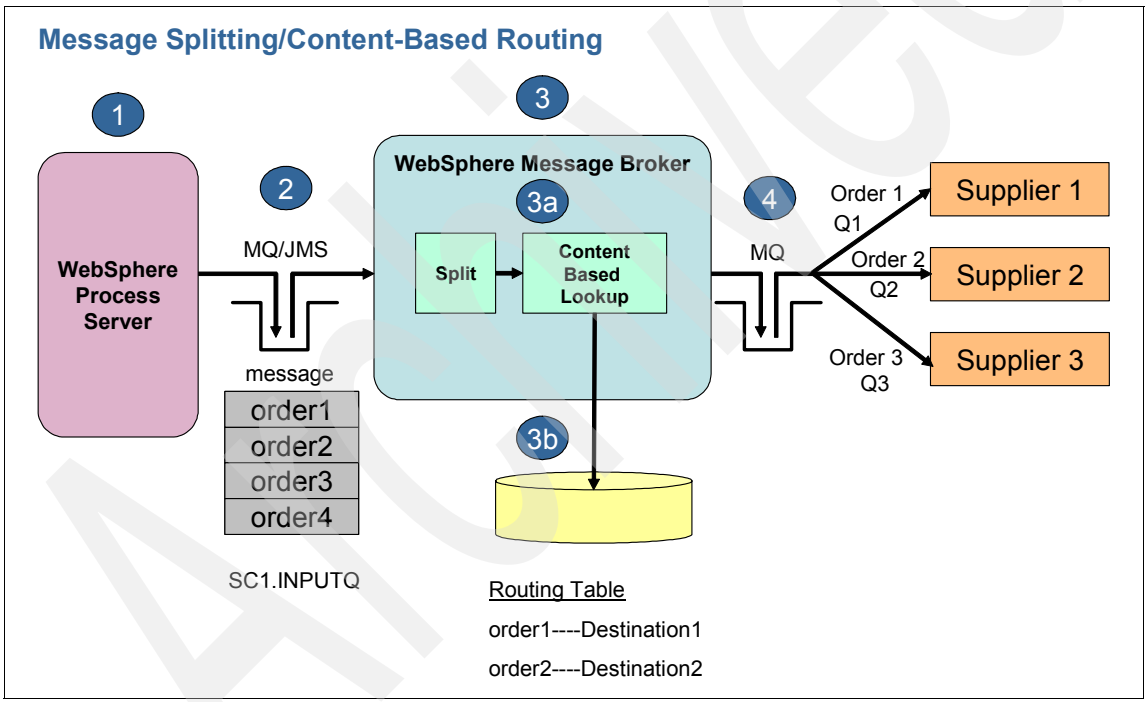


Figure 5-1 Scenario overview

Archive

We describe the highlights of this scenario, numbered in Figure 5-1 on page 80, as follows:

1. WebSphere Process Server hosts a business process that sends a multi-part message to WebSphere Message Broker. The message contains a list of orders.
2. The message is sent as a JMS message over an import component with an MQ JMS binding. It is placed on a WebSphere MQ queue for delivery. This is a one-way request. Therefore, no response is expected from the broker.
3. The delivery of the message triggers a message flow in the WebSphere Message Broker to begin processing:
   a. The message flow converts the JMS message to an MQ message with a format that matches a message model. The message flow then splits the message into separate instances while adhering to the message model. Each instance represents a single order that must be delivered to the correct supplier based on the supplier ID of the order.
   b. Each instance is parsed to obtain the supplier ID. The destination queue is set based on the destination queue that is retrieved from an externalized relational database that corresponds to the supplier ID in the order.
4. The message is then routed to the destination queue that corresponds to the supplier ID in the order message. The message is delivered as an MQ message to a queue that is hosted by WebSphere MQ.

5.2 Environment

The scenario was developed in an environment with the following components:

- WebSphere Integration Developer 6.0.2.2
- WebSphere Process Server 6.0.2.0 (Build number o0648.28)
- WebSphere Application Server 6.0.2.17 (Build number cf170648.10)
- WebSphere Message Broker run time 6.0.0.5
- WebSphere Message Brokers Toolkit 6.0.2 (Interim Fix 009)
- WebSphere MQ 6.0.0.0
- DB2® Enterprise Server 8.1.7.445

Archive

5.3 WebSphere MQ configuration

Some preparation is required for the message flow. The preparation includes creating the WebSphere MQ queues and the JNDI entries that are required for the JMSInput node.

5.3.1 Local queues on the broker queue manager

The message flow requires the following local queues on the broker queue manager:

- SC1.INPUTQ
- SC1.BACKOUTQ
- Q1
- Q2
- Q3
- TRYCATCHQ
- HOLDINGQ_MSGS_WITH_DB_ACCESS_ISSUES
- HOLDINGQ_MSGS_WITH_INVALID_DESTINATIONS

The queues were created using the MQSC commands in Example 5-1.

Example 5-1 MQSC commands to create the queues

```
runmqsc WBRK6_DEFAULT_QUEUE_MANAGER
define qlocal (SC1.INPUTQ) replace
define qlocal (SC1.BACKOUTQ) replace
define qlocal (Q1) replace
define qlocal (Q2) replace
define qlocal (Q3) replace
define qlocal (TRYCATCHQ) replace
define qlocal (HOLDINGQ_MSGS_WITH_DB_ACCESS_ISSUES) replace
define qlocal (HOLDINGQ_MSGS_WITH_INVALID_DESTINATIONS) replace
end
```

5.3.2 JMS entries

The message flow uses a JMSInput node and requires a JNDI queue connection factory and queue entries for that point to the physical queue manager and MQ queues.

MQ queue name: The MQ queue name must be exactly the same as the JNDI name defined in the JNDI bindings file.

Archived

We used the following steps to create these JNDI definitions:

1. Edit the *MQ_Install_Dir*\Java\bin\JMSAdmin.config file and ensure that the following two lines are uncommented in the file:

   ```
   INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
   PROVIDER_URL=file:/C:/JNDI-Directory
   ```

   Comment out the other `INITIAL_CONTEXT_FACTORY` and `PROVIDER_URL` lines.

2. The folder specified as PROVIDER_URL must exist on the system. Create the folder or change PROVIDER_URL to use an existing folder.

   ```
   PROVIDER_URL=file:/C:/Scenario1JNDI-Directory
   ```

3. Put the definitions shown in Example 5-2 in the C:\Scenario1JNDI-Directory\createJNDI.defs file.

   Example 5-2 The createJNDI.defs script

   ```
   # Define a QueueConnectionFactory
   # Only parameters being overridden from their default values are
   specified.
   # This sets up a MQ client binding.
   DEF QCF(SC1_QCF) +
   TRANSPORT(CLIENT) +
   QMANAGER(WBRK6_DEFAULT_QUEUE_MANAGER) +
   HOSTNAME(localhost) +
   PORT(2414)

   # Define a  Queue Destination
   DEF Q(SC1.INPUTQ) +
   QUEUE(SC1.INPUTQ) +
   QMANAGER(WBRK6_DEFAULT_QUEUE_MANAGER)

   # Backout Queue
   DEF Q(SC1.BACKOUTQ) +
   QUEUE(SC1.BACKOUTQ) +
   QMANAGER(WBRK6_DEFAULT_QUEUE_MANAGER)

   END
   ```

4. Execute the following commands:

   ```
   cd MQ_Install_Dir\Java\bin\
   JMSAdmin < C:\Scenario1JNDI-Directory\createJNDI.defs
   ```

5. The following commands confirm that the command ran correctly:

   ```
   JMSAdmin
   dis ctx
   ```

Archived

These two commands should give you results similar to those that are shown in Figure 5-2.

```
5724-H72, 5655-L82, 5724-L26 (c) Copyright IBM Corp. 2002,2005.
All Rights Reserved.
Starting WebSphere MQ classes for Java(tm) Message Service
Administration

InitCtx> dis ctx

  Contents of InitCtx

      .bindings                 java.io.File
      createJNDI.defs           java.io.File
   a  SC1.BACKOUTQ              com.ibm.mq.jms.MQQueue
   a  SC1.INPUTQ                com.ibm.mq.jms.MQQueue
   a  SC1_QCF
com.ibm.mq.jms.MQQueueConnectionFactory

  5 Object(s)
    0 Context(s)
    5 Binding(s), 3 Administered

InitCtx>
```

Figure 5-2 Successful creation of JNDI definitions

6. The WebSphere MQ client classes must be made available to the broker.

   Copy the Java.jar files and any native libraries for the JMS provider client from *MQ_Install_Dir*\Java\lib into the broker shared-classes directory. For example, on a Microsoft® Windows® system, the directory typically has the following format:

   ```
   C:\Documents and Settings\All Users\Application
   Data\IBM\MQSI\shared-classes
   ```

   Copying the files into the shared-classes directory ensures that the Java class path for the JMS nodes is set correctly.

For additional information about connecting JMS transport nodes, refer to the article “Connecting the JMS Transport Nodes for WebSphere Message Broker v6 to Popular JMS Providers” on the Web at:

http://www.ibm.com/developerworks/websphere/library/techarticles/0610_bicheno/0610_bicheno.html

Archived

5.4 Business integration module

In this section, we demonstrate the implementation of a business integration module deployed on WebSphere Process Server. The business requirement is for a large order that contains several orders to be split into single instances. Each instance of the order is to be routed to a destination queue based on the supplier ID.

The business integration module takes a message, which consists of multiple parts (orders), and breaks it into multiple messages with one part (order) each. The module determines the appropriate destination for each message by matching the content in the message to an entry in a database and sends each message to the destination.

5.4.1 Business integration module components

This business scenario is implemented in a business integration module called *Scenario1WPSModule*. Figure 5-3 shows the assembly diagram.

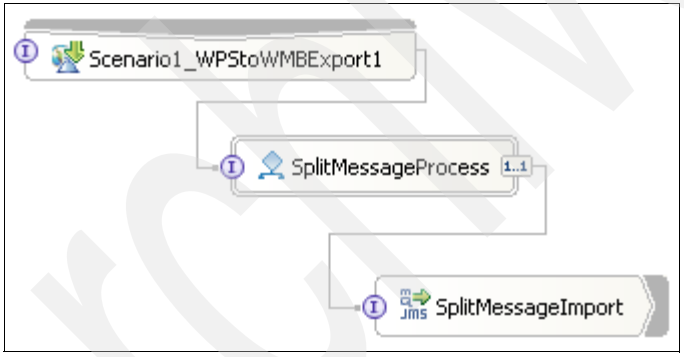


Figure 5-3 Assembly diagram

The assembly diagram contains the following components:

- *Scenario1_WPStoWMBExport1* is the export component that the client uses to access the business process.
- *SplitMessageProcess* contains the business process implementation.
- *SplitMessageImport* is the import component that sends request messages to a service via WebSphere Message Broker.

All three components use a single interface, the Scenario1_WPStoWMB interface. This interface is defined in 5.4.4, “Interfaces” on page 89.

5.4.2 XML schema for message containing multiple orders

The XML schema that is used to define the message format for the messages containing multiple orders is contained in a file called *ProcessOrdersRequest.xsd*. Example 5-3 shows this schema.

Example 5-3 ProcessOrdersRequest schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://sam730.us.ibm.com"
targetNamespace="http://sam730.us.ibm.com"
elementFormDefault="unqualified" version="1.0">
 <xs:element name="ProcessOrdersRequest">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="BatchNumber">
     <xs:simpleType>
      <xs:restriction base="xs:string">
       <xs:pattern value="SCN1_[0-9]{6}"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>
    <xs:element name="Orders">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Order" maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="OrderID" type="xs:integer"/>
          <xs:element name="Supplier">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="SupplierID" type="xs:integer"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="Items">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="Item" maxOccurs="unbounded">
              <xs:complexType>
               <xs:sequence>
                <xs:element name="ItemCode" type="xs:integer"/>
                <xs:element name="ItemDescription" type="xs:string"/>
```

Archived

```
                    <xs:element name="Qty" type="xs:integer"/>
                    <xs:element name="Price" type="xs:decimal"/>
                   </xs:sequence>
                  </xs:complexType>
                 </xs:element>
                </xs:sequence>
               </xs:complexType>
              </xs:element>
             </xs:sequence>
            </xs:complexType>
           </xs:element>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:schema>
```

Input message example

Example 5-4 shows a large order message. The large order consists of multiple Order elements. It uses the ProcessOrdersRequest schema.

Example 5-4 Large message format

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:ProcessOrdersRequest xsi:schemaLocation="http://sam730.us.ibm.com
C:\Documents%20and%20Settings\Administrator\My%20Documents\Redbook%20working%20files\
Scenario1_v2.xsd" xmlns:tns="http://sam730.us.ibm.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <BatchNumber>SCN1_000000</BatchNumber>
   <Orders>
      <Order>
         <OrderID>0</OrderID>
         <Supplier>
            <SupplierID>1</SupplierID>
         </Supplier>
         <Items>
            <Item>
               <ItemCode>0</ItemCode>
               <ItemDescription>String</ItemDescription>
               <Qty>0</Qty>
               <Price>0.0</Price>
            </Item>
```

Archived

```
            </Items>
         </Order>
         <Order>
            <OrderID>0</OrderID>
            <Supplier>
               <SupplierID>2</SupplierID>
            </Supplier>
            <Items>
               <Item>
                  <ItemCode>0</ItemCode>
                  <ItemDescription>String</ItemDescription>
                  <Qty>0</Qty>
                  <Price>0.0</Price>
               </Item>
            </Items>
         </Order>
      </Orders>
</tns:ProcessOrdersRequest>
```

5.4.3 Business object

The module uses a business object named *ProcessOrdersRequest*. The attributes of the object are inherited from ProcessOrdersRequest.xsd. See Figure 5-4.

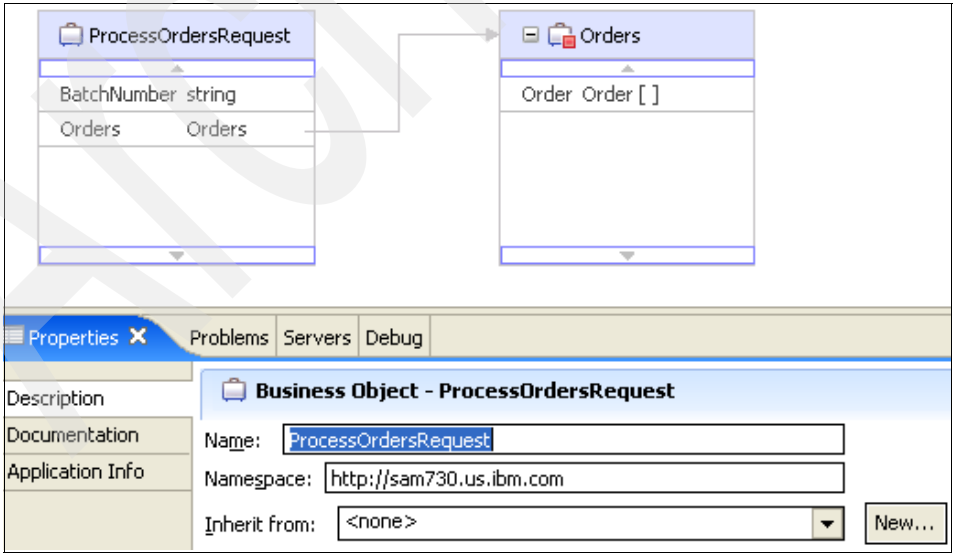


Figure 5-4 ProcessOrdersRequest business object

Archived

The business object contains two attributes:

- *BatchNumber* is a string that represents an alpha numeric sequence. The batch number is what correlates each instance within the message.
- Several instances of *Order* might exist within the Orders structure. Each instance of Order contains elements that you might intuitively associate with an order such as price, quantity, item code, and so on.

The namespace `http://sam730.us.ibm.com` is used.

5.4.4 Interfaces

The module uses one interface called *Scenario1_WPStoWMB* (Figure 5-5). All three components use the same interface.

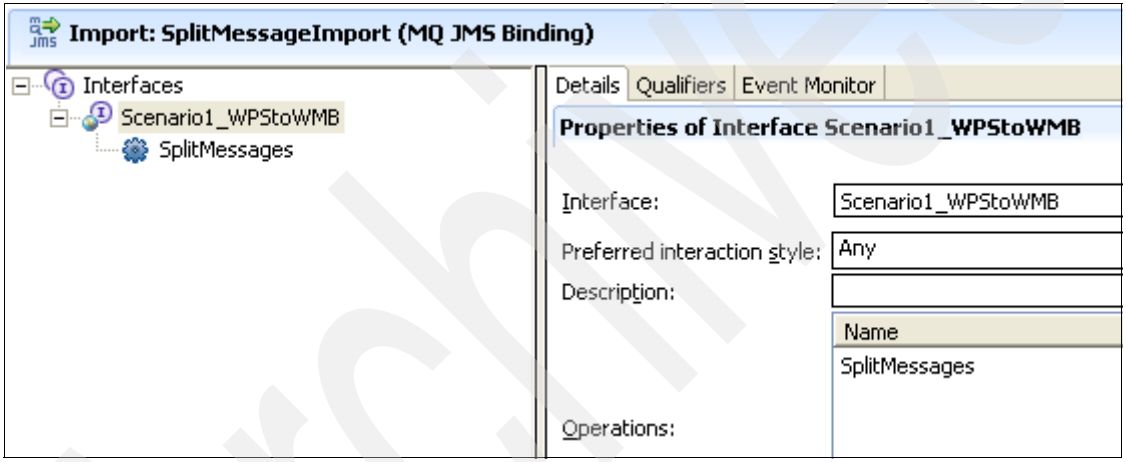


Figure 5-5 Scenario1_WPStoWMB interface

The interface has a one-way operation (Figure 5-6) called *SplitMessages* that uses the ProcessOrdersRequest business object.

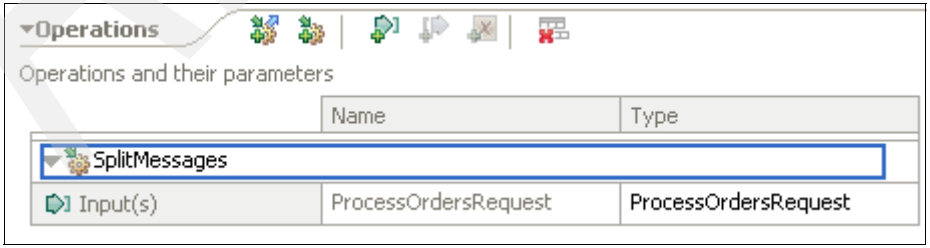


Figure 5-6 SplitMessages interface

5.4.5 Business process

The business process called *SplitMessageProcess* has the interface Scenario1_WPStoWMB (Figure 5-5 on page 89). In the following sections, we describe the business process shown in Figure 5-7.

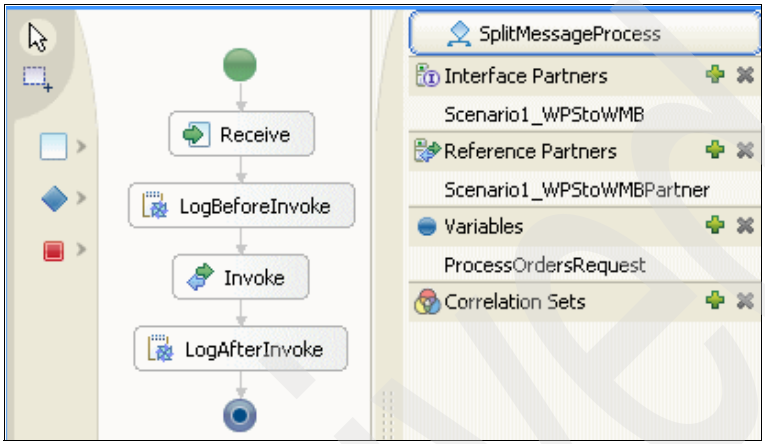


Figure 5-7 Business process overview

The business process uses the default properties.

Interface and partner references

The business process uses one interface partner and one reference partner. You can see these listed to the right of the business process in Figure 5-7.

An *interface partner* is the process interface and exposes operations that can be called by external partners. The Scenario1_WPStoWMB interface is defined as an interface partner (Figure 5-5 on page 89) for the interaction between the business process and its clients.

Reference partners specify the interface that is used in the invocation of another service. In this scenario, Scenario1_WPStoWMBPartner is defined as a reference partner (Figure 5-8 on page 91) and specifies that the Scenario1_WPStoWMB interface is to be used to call the service. In the assembly diagram, this reference partner is wired to the SplitMessageImport component.

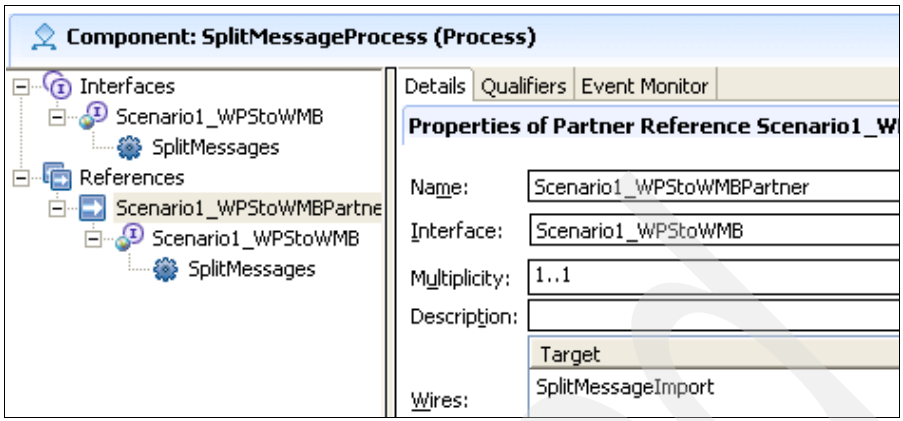


Figure 5-8 Business process reference

Note that both the interface partner and the reference partner use the same interface, because just one operation is being invoked.

Receive activity

The Receive activity receives the input into the business process from a client (Figure 5-9). It is associated with the Scenario1_WPStoWMB interface.

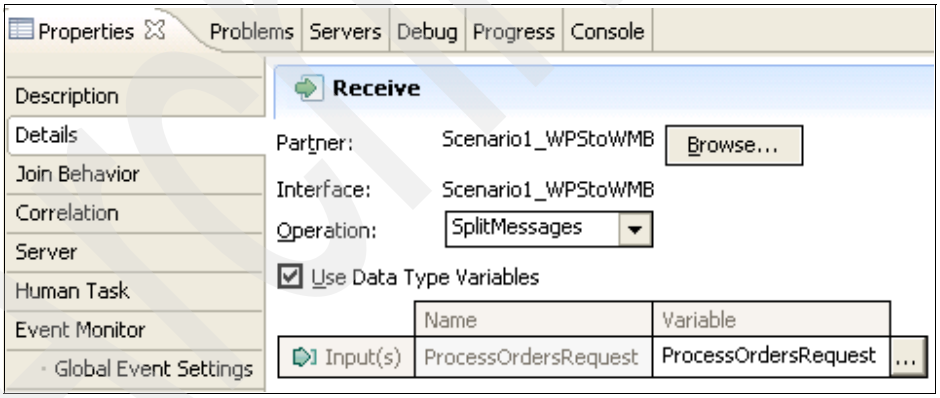


Figure 5-9 Receive activity

Invoke activity

The Invoke activity sends the message to a queue (Figure 5-10 on page 92). It is also associated with the Scenario1_WPStoWMB interface. The Invoke activity is associated with Scenario1_WPStoWMBPartner. This reference partner is wired to the SplitMessageImport component.

Archive

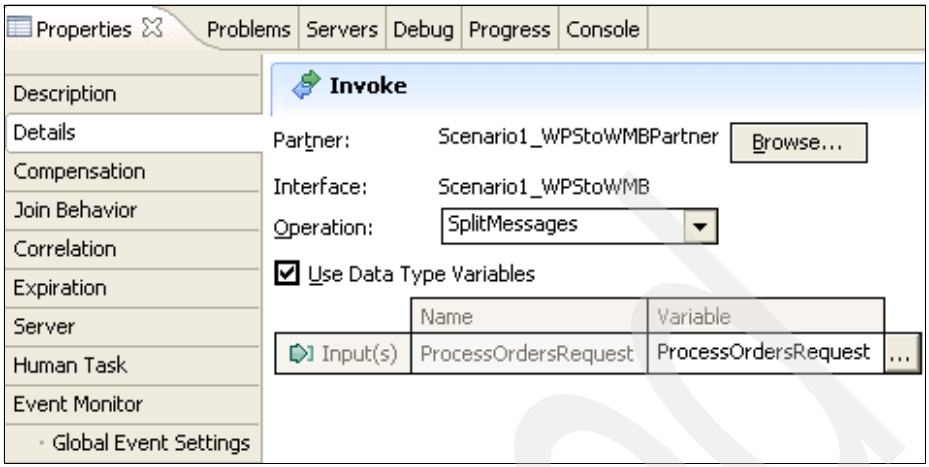


Figure 5-10 Invoke activity

Java snippets

We included two basic Java snippets in the business process. These snippets simply write a timestamp entry to the SystemOut log indicating when the process was invoked and when it completed.

Figure 5-11 shows the LogBeforeInvoke snippet.

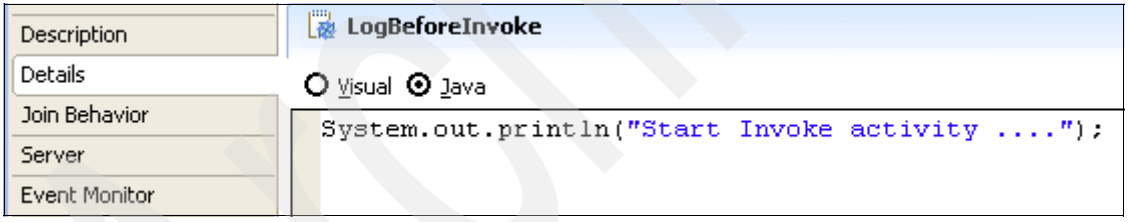


Figure 5-11 LogBeforeInvoke

The LogAfterInvoke snippet, shown in Figure 5-12, is similar with the exception of the message.

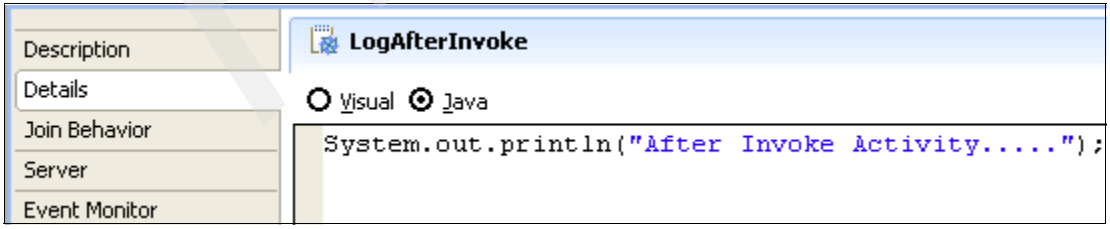


Figure 5-12 LogAfterInvoke

5.4.6 Import component and binding

The SplitMessageImport component is used to invoke the message flow on WebSphere Message Broker. An MQ JMS binding is generated for it. The binding uses the WebSphere MQ messaging provider defined to WebSphere Process Server.

Alternative: The only other choice for accessing the WebSphere MQ Messaging Provider is an MQ binding.

The business process uses the SplitMessageImport component and associated MQ JMS binding to put a message on the queue SC1.INPUTQ to be retrieved by WebSphere Message Broker. Figure 5-13 shows the settings for the bindings.

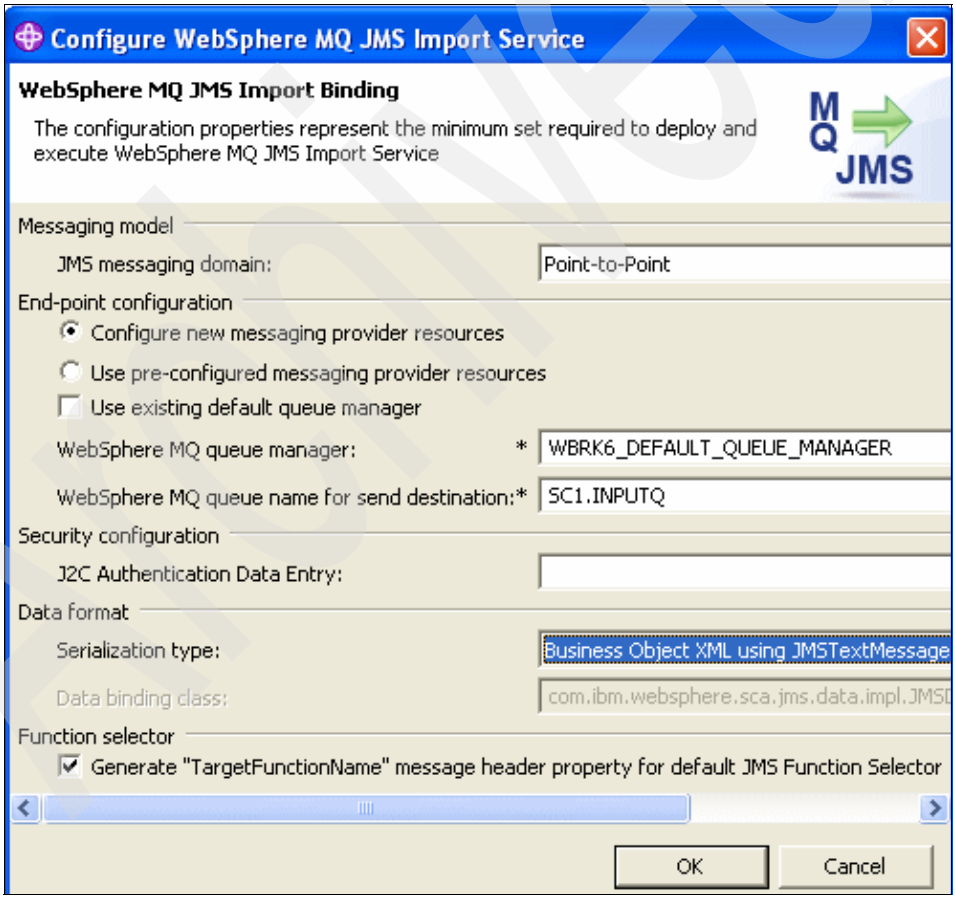


Figure 5-13 MQ JMS Import Binding properties

We explain each of the binding settings as follows:

- For JMS messaging domain, select **Point-to-Point**.

  The only other choice here is to select publish/subscribe.

- For WebSphere MQ queue manager, type `WBRK6_DEFAULT_QUEUE_MANAGER`.

  This is the queue manager for WebSphere Message Broker. This queue manager manages the queue where the message will be sent.

- For WebSphere MQ queue name for the send destination, type `SC1.INPUTQ`.

  This is the queue where the message will be sent. The MQ queue name must be exactly the same as the JNDI name that is defined in the JNDI bindings file.

- For Serialization type, select **Business Object XML using JMSTextMessage**.

  This serialization type is selected because the process is sending a complex message. A complex message contains compound elements within it. The example message in Example 5-4 on page 87 is a complex message because it has compound elements within it. The other factor in deciding to select that serialization type was that the input node in the WebSphere Message Broker flow uses the MRM domain (with XML format), which is the recommended serialization type for that domain.

After the binding is generated, the end-point binding settings are configured for the import component. Figure 5-14 shows the connection factory properties.

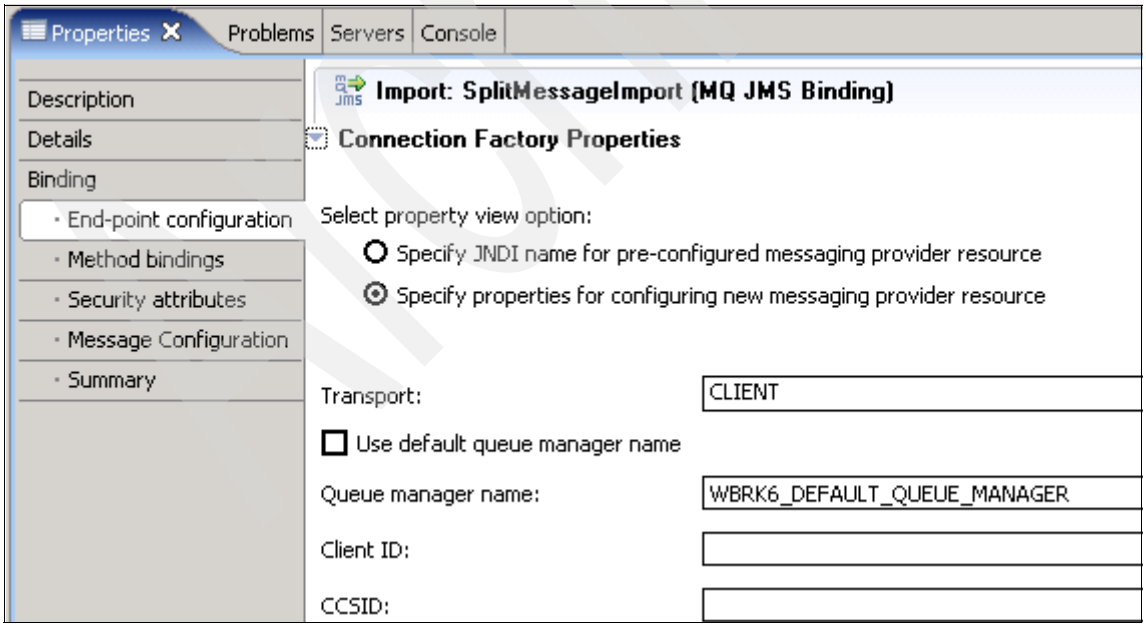


Figure 5-14 Import binding connection factory properties

The connection properties are defined as follows:

- For Transport, select **CLIENT**.

  Options include CLIENT or BINDINGS. In this scenario, client transport is used. With this option, client configuration properties become active and require a host name, channel, and port to be specified.

  Using a client transport gives you more control. It also means that you must specify the values for the client configuration including host name, channel and port, and optionally the client channel definition table. The WebSphere MQ client connection is used to connect to the queue manager.

  BINDINGS is the default setting and is simpler to set up. You do not need to specify host name, channel, or port because WebSphere MQ JMS classes use the Java Native Interface (JNI™) to call directly into the existing queue manager API rather than communicating through a network. Bindings is a shared memory protocol and can offer better performance. Bindings can only be used when the queue manager is on the same node as the JMS client. To use the bindings connection, WebSphere MQ JMS classes must be installed on the WebSphere MQ server.

- For Queue manager name, type `WBRK6_DEFAULT_QUEUE_MANAGER`.

  The name of the broker queue manager is filled in based on the value that you specify when you create the binding (Figure 5-13 on page 93).

Figure 5-15 shows the updated Client Configuration Properties section.

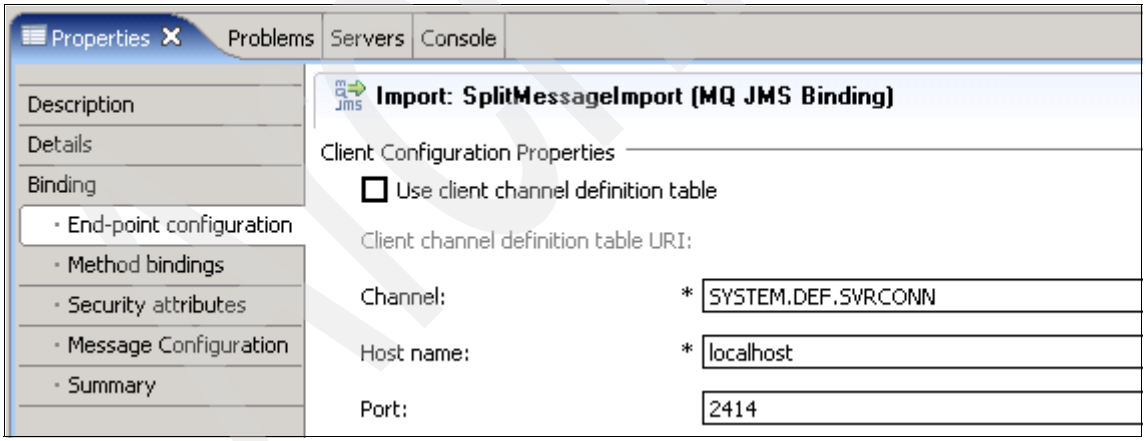


Figure 5-15 Import binding client configuration properties

We selected the following properties for the client configuration:

- For Channel, type `SYSTEM.DEF.SRVCONN`.

  The name of the channel is used to connect to the broker queue manager. It is only necessary for a client connection.

- For Host name, type `localhost`.

  In this field, you specify the name of the destination machine. Alternatively, you can supply an IP address.

- For Port, type `2414`.

  In the Port field, you type the TCP/IP port that is used for connection to the broker queue manager. It is only necessary for a client connection.

Figure 5-16 shows the Send Destination Properties.

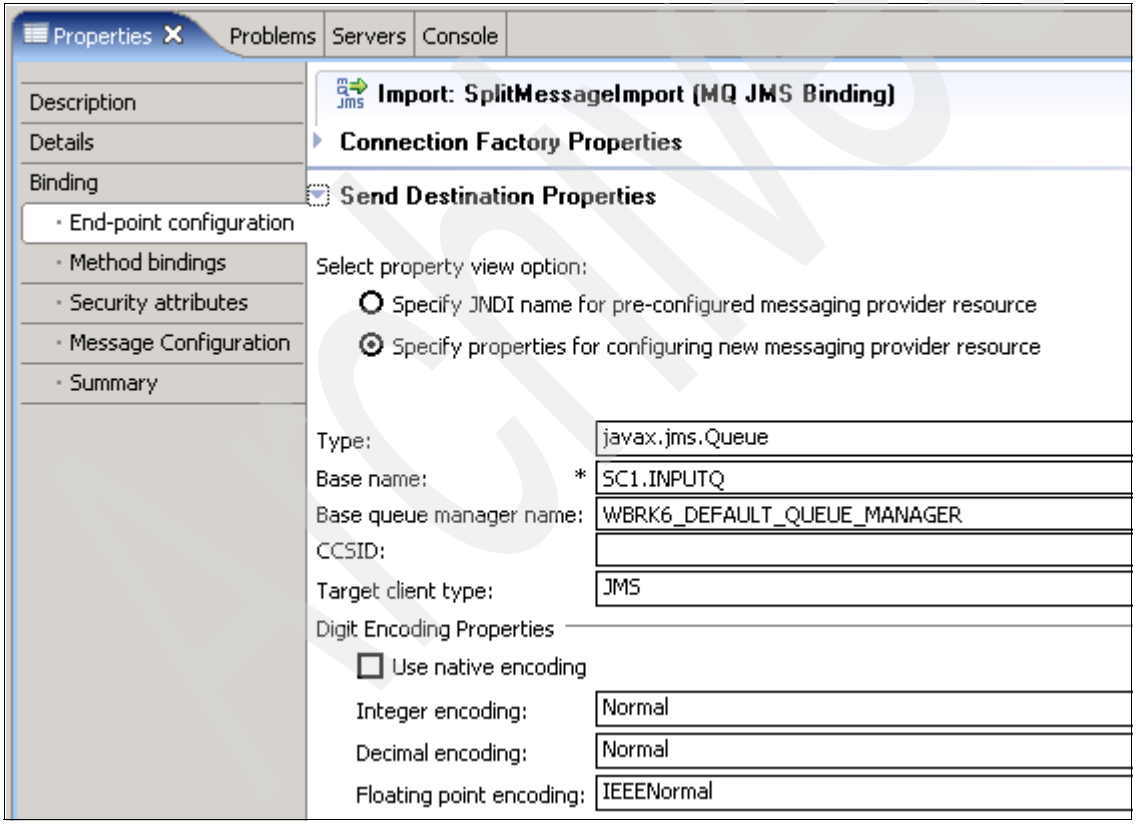


Figure 5-16 Import properties - End-point-configuration

We defined these properties as follows:

- For Type, type `javax.jms.Queue`.

  In this field, you specify the destination type, which is a queue.

- For Base name, type `SC1.INPUTQ`.

  Base name is the name of the queue to which messages are sent on the queue manager that is specified for the base queue manager name.

  Topic destination type: If the destination type is *topic*, the base name field should contain the name of the WebSphere MQ topic to which messages are sent.

- For Target client type, select **JMS**.

  In this field, you specify whether the client is MQ or JMS.

5.5 Message flow

In this section, we demonstrate the implementation of the message flow that is deployed on WebSphere Message Broker. This scenario uses one message flow called Scenario1MsgFlw.

5.5.1 Scenario1MsgFlw message flow

Figure 5-17 shows the Scenario1MsgFlw message flow.

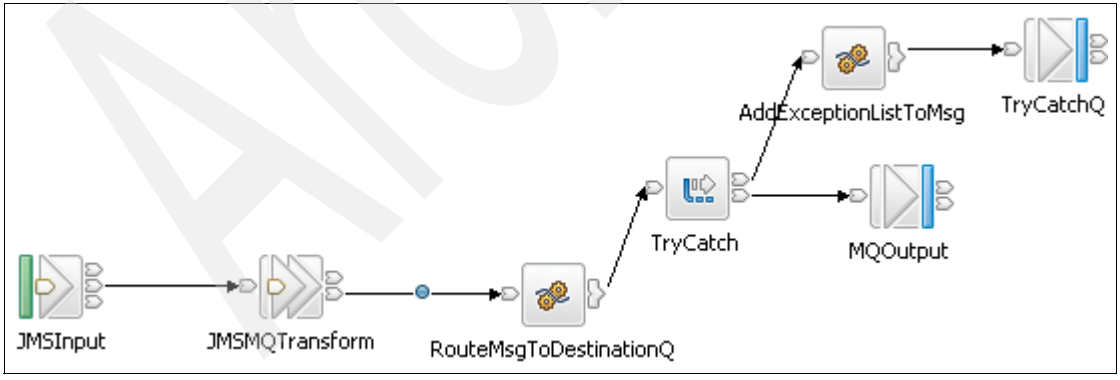


Figure 5-17 Scenario1MsgFlw

Archived

The flow follows this sequence:

1. The JMS message arrives on a queue in the JMS format as a list of orders. The JMSInput node receives the message and forwards it to the JMSMQTransform node.
2. The JMSMQTransform node converts the message from the JMS format to the MQ format. The message is then propagated to the RouteMsgToDestinationQ Compute node.
3. The RouteMsgToDestinationQ Compute node splits the message into single instances, where each message represents a single order. It then determines the destination queue for each order by using the order's supplier ID tag to search an external relational database for the corresponding destination queue.

   The destination queue is added to the LocalEnvironment folder in the message tree. The destination queue is either the queue that is associated with a supplier, if a match was found during the database search, or a holding queue that is to be investigated by the support team.

   Each message is then propagated to the next node in the flow.
4. The TryCatch node routes the message down its Try terminal to the MQOutput node.
5. The MQOutput node attempts to deliver the message to the destination queue. If the delivery fails for any reason, an exception list is added to the message tree and the message is then rolled back to the TryCatch node. The message is then sent down the Catch terminal of the TryCatch node.
6. The Compute node attached to the Catch terminal appends the exception list to the body of the message and propagates this message to an MQOutput node (TryCatchQ). The MQOutput node delivers the message to the MQ destination queue, TryCatchQ.

Archive

5.5.2 Scenario1MsgSetPrj message set project

The validation of incoming messages is performed by using a message set. The message set was created from the XML schema in Example 5-3 on page 86. The message set is assigned to the input node (JMSInput), so that all incoming messages are effectively validated against the XML schema.

SupportPac IA9A alternative: Alternatively, SupportPac IA9A, an XML validator plug-in node, can be used to validate XML messages against the XML schema. This SupportPac supplies a WBI Message Broker plug-in node that validates an XML message against an XML schema. If the message conforms to the schema, the message is passed to the Out terminal unchanged. However, if errors are encountered, the errors are added to the Environment tree and are passed to the Invalid terminal, along with the unchanged message.

The XML validator node checks the bit stream of a message, as opposed to the logical structure, and therefore supports any message format, such as XML, XMLNS, and BLOB. Configurable parameters are supplied to allow the schema defined in the XML document to be overridden, and can be set either via the node properties or at run time, by the use of Environment or LocalEnvironment variables.

The node is implemented as a Java plug-in and supports all Message Broker platforms except z/OS.

You can find further information about SupportPacs on the WebSphere MQ Family SupportPacs page at:

http://www-1.ibm.com/support/docview.wss?rs=849&uid=swg27007205

Archive

The Scenario1MsgSetPrj message set contains the message definitions that model the messages that are involved in the message flow. See Figure 5-18.

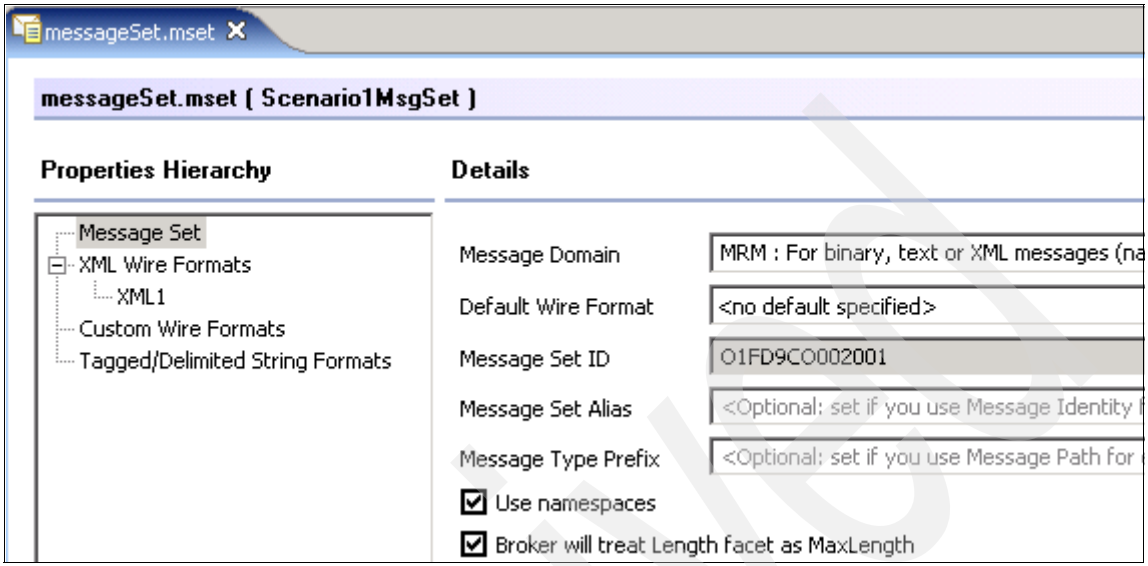


Figure 5-18 messageSet.mset

The message set is identified by the message set ID. Any message to be modeled by a message definition within this message set must have the Message Set ID property set in the Properties folder of the message tree. MRM is the message domain for this message set. The message set supports XML.

Archived

Message definitions

The Scenario1MsgDef.mxsd message definition file (Figure 5-19) shows the structure of how the ProcessOrdersRequest message will be modeled.

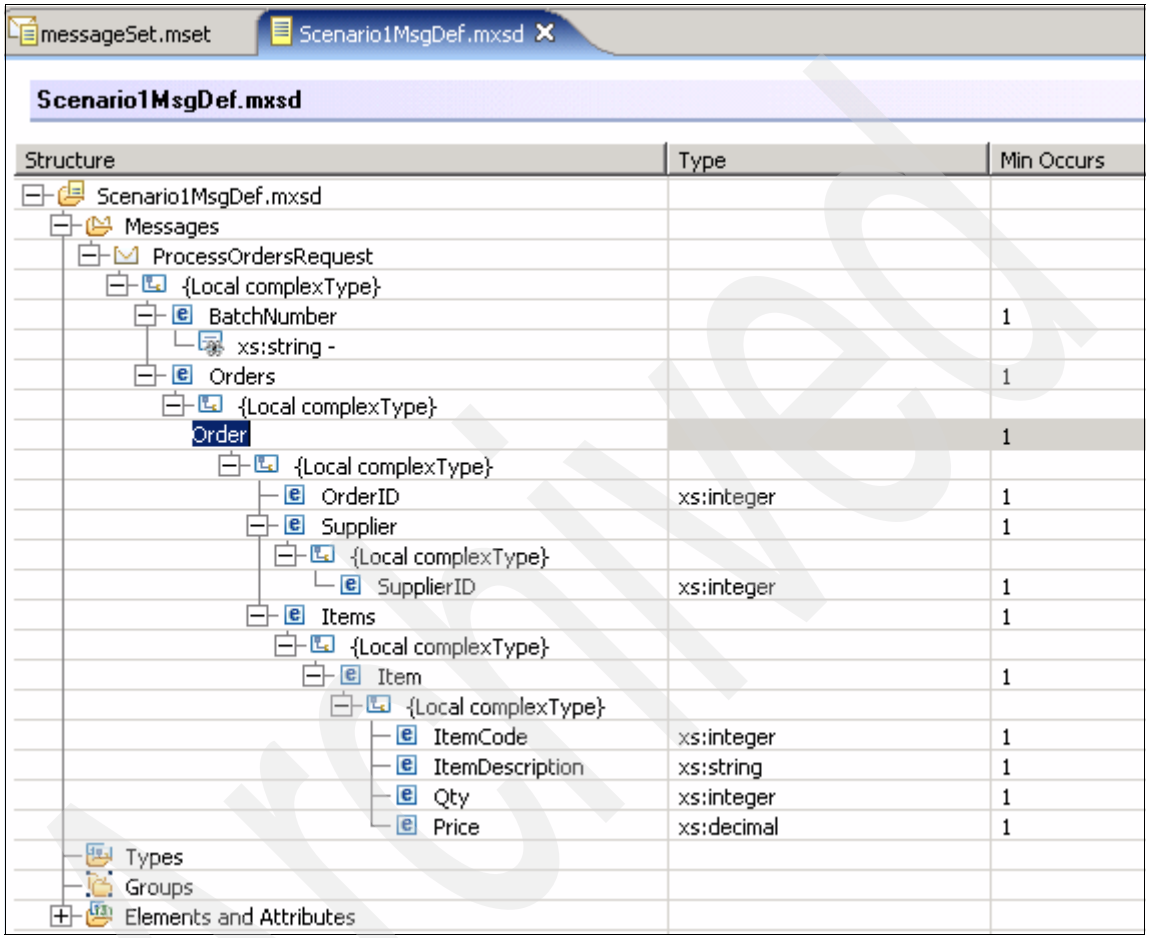


Figure 5-19 The Scenario1MsgDef.mxsd

Archived

5.5.3 Message flow nodes

In this section, we describe each node in the message flow (see Figure 5-17 on page 97).

JMSInput node

The JMSInput node begins the processing of the message by getting the message from the input queue, parsing the message, and propagating the message to the JMSMQTransform node.

The Basic properties (Figure 5-20) define the source input queue for this message flow. The MQ queue name must be exactly the same as the JNDI name defined in the JNDI bindings file.

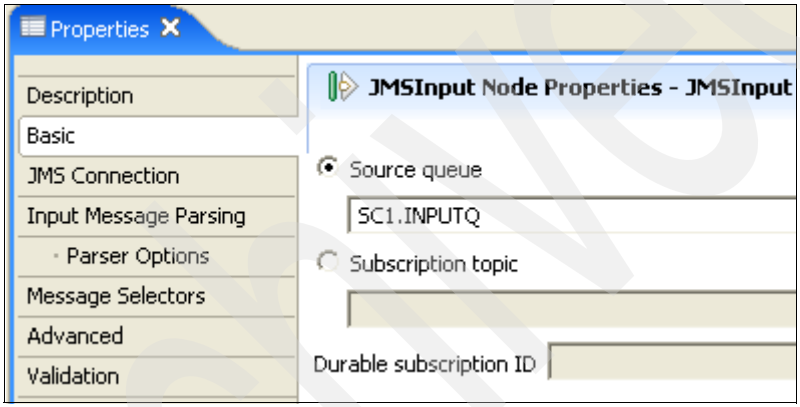


Figure 5-20 JMSInput Node - Basic

Archived

The JMS Connection properties (Figure 5-21) include the initial context factory, the location of the JNDI bindings file, the name of the connection factory, and the backout queue for this JMS connection. The JNDI definitions must match those created in 5.3.2, “JMS entries” on page 82.

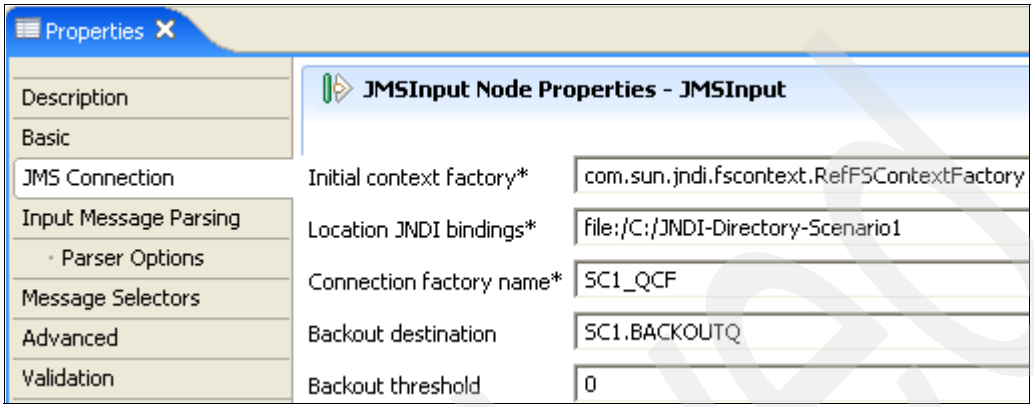


Figure 5-21 JMSInput Node - JMS Connection

The Input Message Parsing properties define how the message is modeled. The properties shown in Figure 5-22 indicate that the message type used is ProcessOrdersRequest of the Scenario1MsgSet(O1FD9CO002001) message set. Messages are expected to be in an XML wire format that conforms to the list of properties that are contained in the XML1 wire format label in the message set.

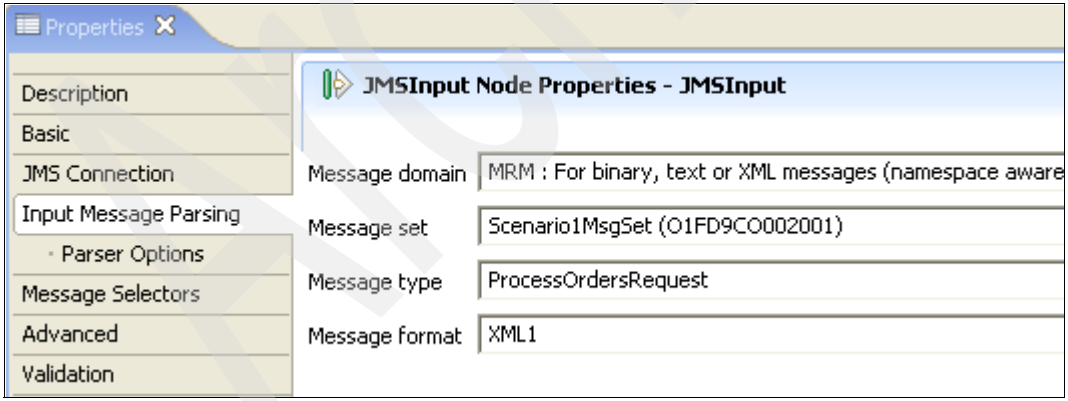


Figure 5-22 JMSInput Node - Input Message Parsing

The Input Message Parsing properties consist of the following fields:

- *Message domain* specifies the parser being used.
- *Message set* is a unique ID that is generated for each message set created (Figure 5-18 on page 100).
- *Message type* can be a list of messages that are defined in the message set. Because only one message was defined in our message set, we have just one message type that we can select from the drop-down list.
- *Message format* is the name of the container for the physical representation of a message. The default name for this container is XML1. We have accepted the default name and default values for the physical properties of the message.

The Parser Options properties (Figure 5-23) specify that the entire message is parsed upon entry to the JMSInput node. Should parsing fail, the message is backed out to the backout queue.

Tip: For a production message flow, we recommend that you configure a series of nodes that are attached to the catch and failure terminals, so that parsing or message validation failures are handled more comprehensively. You can find further information about error handling routines in *WebSphere Message Broker V6, Best Practices Guide: Bullet Proofing Message Flows*, REDP-4043.

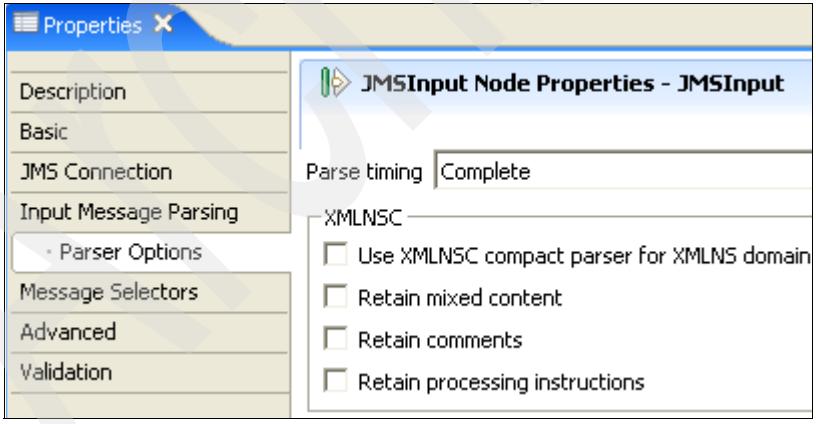


Figure 5-23 JMSInput Node - Parser Options

JMSMQTransform node

The JMSMQTransform node converts messages from the JMS format to the MQ format. The node is not configurable.

RouteMsgToDestinationQ Compute node

The RouteMsgToDestinationQ Compute node parses the incoming message and creates new messages for each order that is found. The node sets the destination queue for the new message based on a database lookup.

The Basic properties (Figure 5-24) define the location of the extended SQL (ESQL) module that is used to implement the logic of this node and the data source name of the database that is used to host the routing table that is accessed by this node.

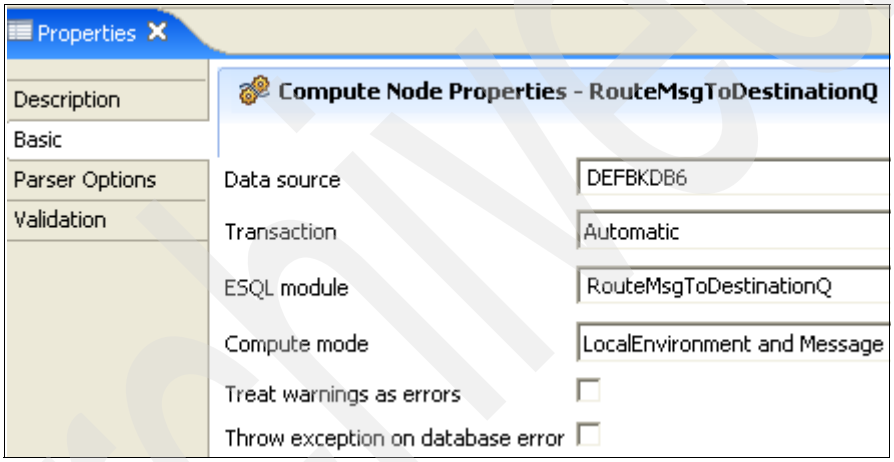


Figure 5-24 RouteMsgToDestinationQ - Basic properties

Tip: The elements selected in the Compute mode field (LocalEnvironment and Message) are the parts of the message that will be altered in the Compute node. All other elements are copied unaltered to the Output terminal.

Select only those parts of the message tree that you need to alter in the Compute node. By selecting on those parts, you can improve performance since only those elements that are selected will be held in memory during that portion of processing.

Archive

Example 5-5 shows the ESQL code that implements the node. It performs the following actions:

1. The message is split into individual orders.
2. Each order is parsed to determine the supplier ID that is associated with the order. The destination queue to which this order is delivered must match the supplier's queue.
3. To determine the destination queue for the order, a database table is searched for the destination queue that is associated with the supplier ID.
4. If the destination queue is returned from the database successfully, the destination queue of the message is set to that queue.
5. If database errors are reported by the select statement, the message is routed to a holding queue for messages for which no destination queue could be determined due to issues in communicating with the database.
6. If no database error occurred, but no destination queue is returned, the message is routed to a different holding queue.
7. For each message contained in the multipart message, a single message is sent to the out terminal of the compute node. Each of these single messages will have a destination queue set in LocalEnvironment.

Example 5-5 ESQL code for RouteMsgToDestinationQ

```
CREATE COMPUTE MODULE RouteMsgToDestinationQ
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      /* In this module, we loop through all the instances in the incoming message.
         We query the database for an onward routing queue for each supplier id.
         If we have a match, we route that instance of the message to the relevant
         queue. If we do not have a match or if we encounter problems, we craft an
         appropriate message and write that message to a queue.
      */
      DECLARE forRef REFERENCE TO InputRoot.MRM;--used to loop through input msg
      DECLARE ileRef REFERENCE TO InputLocalEnvironment;
      DECLARE oleRef REFERENCE TO OutputLocalEnvironment;
      DECLARE suppID CHAR; -- used to store the supplier id

      MOVE forRef FIRSTCHILD NAME 'Orders'; -- Move our loop reference to the start
                                 -- of the folder holding all instances
                                 -- of Order elements.

      -- Perform the following actions for each instance of
      -- 'Order' within the incoming message
      FOR forRef AS forRef.Order[] DO (1)
         CALL CopyMessageHeaders();
```

```
SET suppID = forRef.Supplier.SupplierID;(2)

SET ileRef.Variables.destQ[] = (SELECT R.DESTINATION_QUEUE (3)
                        FROM Database.BKR_FLOW_DATA.ROUTING AS R
                        WHERE R.SUPPLIER_ID = suppID);

CASE -- Let's see if we have any errors/missing values after our attempt
   -- to retrieve routing details from the database
   WHEN SQLCODE <> 0 THEN (5)
      -- There is a database access problem.
      --Let's report it on the relevant queue
      SET oleRef.Destination.MQDestinationList.DestinationData[1].queueName
            = 'HOLDINGQ_MSGS_WITH_DB_ACCESS_ISSUES';
      CREATE LASTCHILD OF OutputRoot DOMAIN('XML');
      SET OutputRoot.XML.Message.Summary =
            'Database problem encountered when attempting to '||
            'retrieve routing data for an instance of a '||
            'ProcessOrdersRequest message';
      SET OutputRoot.XML.Message.Details.Detail[1] =
            'SQLSTATE is: ' || CAST(SQLSTATE AS CHAR);
      SET OutputRoot.XML.Message.Details.Detail[2] =
            'SQLCODE is: ' || CAST(SQLCODE AS CHAR);
      SET OutputRoot.XML.Message.Details.Detail[3] =
            'SQLERRORTEXT is: ' || CAST(SQLERRORTEXT AS CHAR);
      SET OutputRoot.XML.Message.Details.Detail[4] =
            'SQLNATIVEERROR is: '  || CAST(SQLNATIVEERROR AS CHAR);

   WHEN NOT EXISTS(ileRef.Variables.destQ[]) THEN (6)
      -- No routing entry found for supplier. Send a message
      -- to the 'invalid destination' queue
      SET oleRef.Destination.MQDestinationList.DestinationData[1].queueName =
            'HOLDINGQ_MSGS_WITH_INVALID_DESTINATIONS';
      CREATE LASTCHILD OF OutputRoot DOMAIN('XML');
      SET OutputRoot.XML.Message.Summary =
      'No routing entry exists for the following instance of a
ProcessOrdersRequest message';

   ELSE (4)
      -- All ok. Let's setup the destination queue using the routing
      -- details retrieved from the database
      SET oleRef.Destination.MQDestinationList.DestinationData[1].queueName =
         TRIM(ileRef.Variables.destQ[1].DESTINATION_QUEUE);
END CASE;
```

```
        --Build the rest of the message to propagate
        SET OutputRoot.MRM.BatchNumber = InputBody.BatchNumber;
        SET OutputRoot.MRM.Orders.Order = forRef;
        PROPAGATE;

      END FOR;

      RETURN FALSE;
   END;

   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;
END MODULE;
```

TryCatch node

The TryCatch node routes the message to the MQOutput node via its Try terminal. Should an exception be generated in the MQOutput node when it attempts to put the message into the destination queue, the message is rolled back to the TryCatch node. The TryCatch node then routes the message down its Catch terminal to the AddExceptionListToMsg Compute node.

AddExceptionListToMsg Compute node

When an exception is generated in a flow, an exception list is generated and added to the message tree. The AddExceptionListToMsg Compute node adds that exception list to the body of the message before the message is written to the TryCatch queue.

Archived

The Basic properties (Figure 5-25) define the location of the ESQL module that is used to implement the logic of this node.

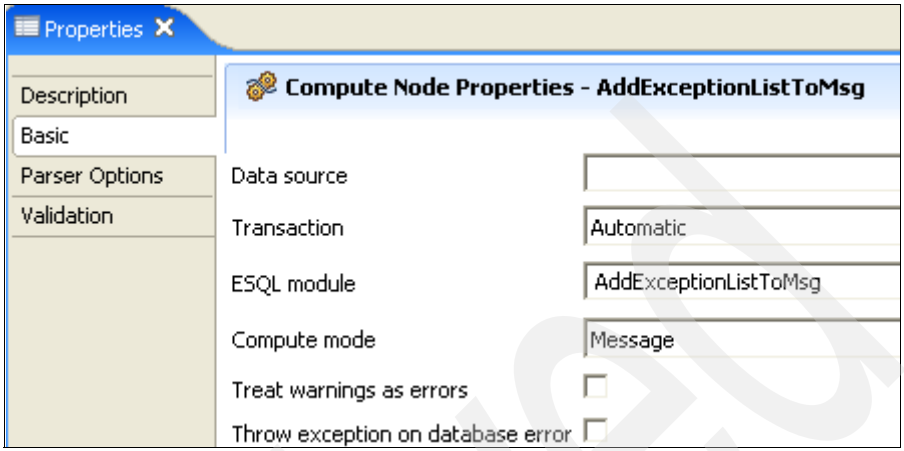


Figure 5-25 AddExceptionListToMsg node - Basic properties

Example 5-6 shows the ESQL that is used to implement this node.

Example 5-6 ESQL code for AddExceptionListToMsg

```
CREATE COMPUTE MODULE AddExceptionListToMsg
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      /* In this module, we simply create a message containing the message
         we tried to write to the destination queue and the exception generated
         when we attempted to write that message.
      */
      DECLARE qName CHAR;

      CALL CopyMessageHeaders();
      SET qName =

InputLocalEnvironment.Destination.MQDestinationList.DestinationData[1].queueName;
      CREATE LASTCHILD OF OutputRoot DOMAIN('XML');
      SET OutputRoot.XML.Message.Summary =
         'There was a problem writing a message to the queue: '
         || qName || ' The exception generated was:';
      SET OutputRoot.XML.Message.ExceptionList =
         InputExceptionList.*[1].RecoverableException[<];
      SET OutputRoot.XML.Message.OriginalMessage = InputRoot;
      RETURN TRUE;
   END;
```

Archived

```
   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;
END MODULE;
```

MQOutput node

The MQOutput node delivers the message to the destination queue defined in the LocalEnvironment element of the message tree:

`LocalEnvironment.Destination.MQDestinationList.DestinationData[1].queueName`

The destination was previously defined in the LocalEnvironment element by the RouteMsgToDestinationQ Compute node.

In the Advanced properties of the MQOutput node, you set Destination mode to `Destination list`, so that the MQOutput node knows to search the LocalEnvironment structure of the message for the destination queue (Figure 5-26).

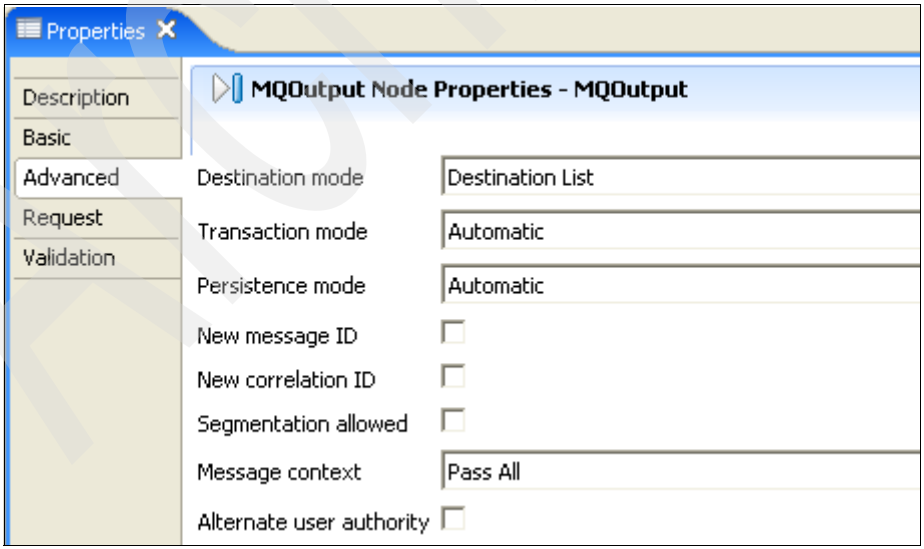


Figure 5-26 MQOutput node - Advanced properties

TryCatchQ MQOutput node

TryCatchQ receives the message if the delivery of the message to the destination queue fails. The Basic properties of TryCatchQ (Figure 5-27) define the queue to which this message will be written.

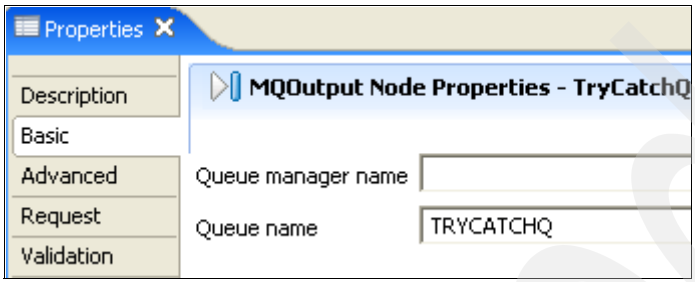


Figure 5-27 TryCatchQ node - Basic properties

5.6 DB2 database configuration

This scenario uses DB2 Enterprise Server as the RDBMS. A list of other suitable databases can be found in the “Supported databases” topic in the WebSphere Message Broker Information Center at:

`http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/topic/com.ibm.etools.mft.doc/ah10030_.htm`

In this scenario, we use the broker database and add a table that will be used for routing messages to the relevant destination queues. This table has a different schema to separate it from the broker database tables.

Recommendation: In production systems, we recommend that you have a separate database for application data. The application database should be on the same host as the broker for good performance.

In this scenario, we used the broker database to host application data because the data volume is trivial.

The DB2 control center was used to perform the following tasks, which are necessary to create and initialize the database:

1. A new schema called BKR_FLOW_DATA was created to hold message flow related data.
2. A new table called ROUTING was created to store the routing data. The schema for the table was specified to be the new BKR_FLOW_DATA schema.

Archived

3. The columns shown in Table 5-1 were added to the table.

Table 5-1 Routing database columns

Column name	Data type
SUPPLIER_ID	Integer
DESTINATION_QUEUE	Character (length=50)

The SUPPLIER_ID column was defined as the primary key.

4. The routing entries shown in Figure 5-28 were added to the table. To add these entries, you must open the table for editing and then add rows.

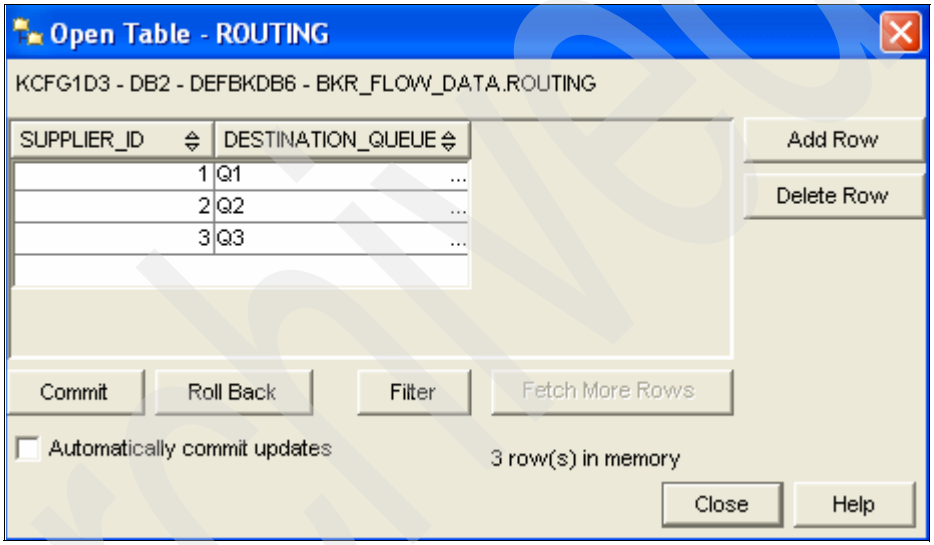


Figure 5-28 Routing entries in the database table

5.7 Testing the flow

In this section, we show the integration testing between the integration module and the message flow in WebSphere Integration Developer. The flow was tested using the Integration Test Client.

To test the flow:

1. Select **Senario1WPSModule**, right-click, and select **Test** → **Test Module**.
2. Enter values for the ProcessOrdersRequest business object attributes in the Initial request parameters area.

3. Click **Continue**. See Figure 5-29.

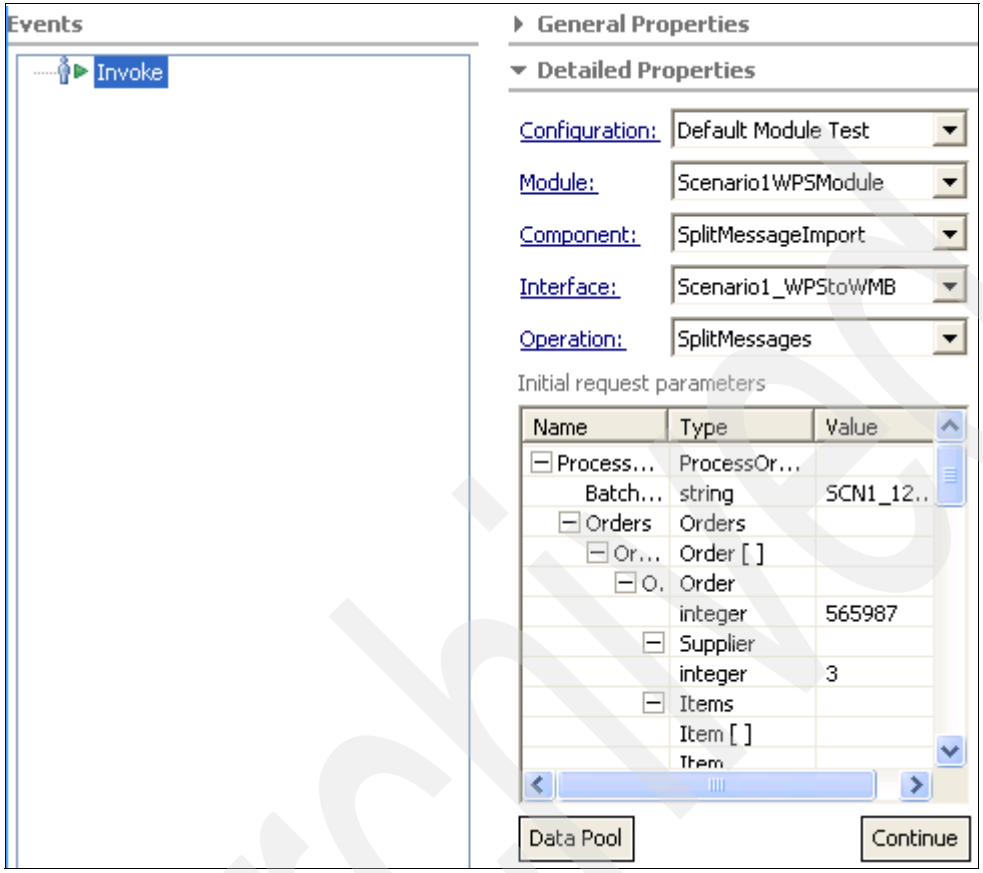


Figure 5-29 Invoking the test by clicking Continue

The messages are written to the queues. These messages will be single instances of Order. The messages can be viewed using the WebSphere MQ Explorer or RFHUtil.

An entry is also in the SystemOut log for the WebSphere Process Server application server. Example 5-7 shows a sample of the output that is generated.

Example 5-7 Example output in SystemOut.log

```
[9/20/07 9:34:20:281 EDT] 00000062 SystemOut     O This is before
Invoke activity ....
[9/20/07 9:34:20:844 EDT] 00000062 SystemOut     O After the
Invoke.....
```

Archived

6

Scenario: WebSphere Message Broker acting as a transaction manager

In the scenario presented in this chapter, we demonstrate the following concepts:

- XA transaction management in a mediation

  The message flow performs a globally coordinated XA transaction taking a single message and routing it to two WebSphere MQ JMS providers, that is queue managers.

- Connectivity using an MQ JMS binding

  WebSphere Process Server uses an import component with an MQ JMS binding to send a Java Message Service (JMS) message to WebSphere Message Broker. WebSphere Message Broker uses a JMSInput node to receive the message from a queue.

- Request-reply interaction pattern

  A request is sent from WebSphere Process Server to WebSphere Message Broker. Depending on the outcome of the transaction, a status message is sent back to WebSphere Process Server.

© Copyright IBM Corp. 2008. All rights reserved.

WebSphere Message Broker interacts with WebSphere MQ JMS providers using JMSOutput nodes.

6.1 Scenario overview

In this scenario, we illustrate a business process that sends a request for a transfer of funds from Bank1 to Bank2 in a globally coordinated transaction. A message flow deployed on WebSphere Message Broker picks up the message, and depending on outcome of the transaction, a status is sent back to the business process in a synchronously.

WebSphere Message Broker can integrate with most third-party vendors and, therefore, can work as an integration hub for various JMS providers. In addition, WebSphere Message Broker can act as a transaction manager for these multiple messaging products.

6.1.1 Process and message flow

Figure 6-1 shows the basic flow of this scenario.

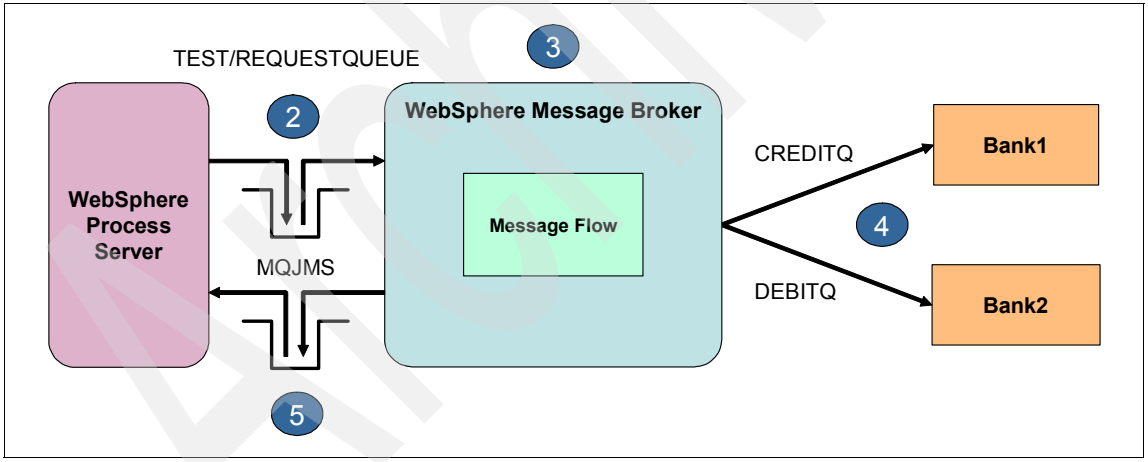


Figure 6-1 Transfer of funds scenario

We describe the highlights of this scenario, numbered in Figure 6-1, as follows:

1. A short-running business process in WebSphere Process Server sends a request to transfer funds from one bank account to another.
2. The message is sent as a JMS message to a queue on the queue manager of WebSphere Message Broker.

3. A JMSInput node on the message flow deployed on WebSphere Message Broker retrieves the JMS message from the WebSphere queue and manages the transaction with two WebSphere MQ queue managers for a funds transfer. The same can be extended to two third-party JMS providers as well.
4. A JMSOutput node in the message flow sends the JMS message to Bank1. Another JMSOutput node sends the same message to Bank2 for debiting. These messages are part of an XA transaction, where either both get committed or both get rolled back simultaneously.
5. Depending on the outcome of the transaction, a message with a status of *transaction* is sent back to the business process synchronously.

6.1.2 XA transactions with WebSphere Message Broker

A *transaction* (also called a *local* or *simple transaction*) is a series of actions that are performed as a single logical unit of work in which either all of the actions are performed or none of them are performed. For a transaction to commit successfully, all of the individual resources must commit successfully. If any of them are unsuccessful, the transaction must roll back in all of the resources. A distributed transaction occurs across multiple independent transactional resources.

The XA specification documents the Open Group X/Open Distributed Transaction Processing (DTP) model, which defines how an application program uses a transaction manager to coordinate a distributed transaction across multiple resource managers. Any resource manager that adheres to the XA specification can participate in a transaction coordinated by an XA-compliant transaction manager, thereby enabling different vendors' transactional products to work together. All XA-compliant transactions are distributed transactions. XA supports both single-phase and two-phase commit.

You can learn about the XA specification in the book *Distributed Transaction Processing: The XA Specification* by The Open Group on the Web at:

`http://www.opengroup.org/public/pubs/catalog/c193.htm`

Useful terms:

Transaction manager
The person who coordinates the transaction that controls a global unit of work. Must be able to communicate with all participants within that unit of work.

Resource manager
The participants within a global unit of work. Owns and manages resources that can be accessed and updated by applications. Such examples are a WebSphere MQ queue manager whose resources are its queues or a DB2 database whose resources are its tables.

Unit of work When an application updates the resources of one or more resource managers, it is often vital that all updates complete successfully as a group or none of them complete. Updates that complete this way are said to occur within a unit of work or transaction.

Syncpoint coordinator
The computer subsystem that manages units of work. The point in time when all updates within a unit of work are either committed or backed out is called a *sync point*.

The JMS transport nodes in WebSphere Message Broker offer different levels of transactional support, which can be configured as a property of the node instance depending on your particular requirements:

- Non-transacted

  Messages passing through the flow are not part of any transaction. This setting best suits non-persistent messages where high throughput is a requirement.

- Local transactions

  The individual nodes can be enabled for local transactions, which means that each node uses a transacted JMS session to their JMS provider, so that the messages are sent and received as part of individual transactions.

- Global transactions (XA transactions)

  JMS destinations that supply messages to a JMSInput node, or receive messages from a JMSOutput node, can be coordinated as part of a message flow global transaction. Therefore, the whole operation that is performed by the flow can be contained in a transaction, so that all or none of the processing takes place, depending on the outcome of the transaction. Global transactions are available to any JMS provider that conforms to the JMS 1.1

specification and supports the JMS XAResource API through the JMS session. An example of such a provider is WebSphere MQ.

Global transactions use the WebSphere MQ queue manager as the transaction coordinator. The coordinator handles events, such as transaction start and end, and the need for other resources to be included in the unit-of-work, including resources such as updates to WebSphere MQ queues, DB2 databases, and any XA-compliant JMS provider destination. Therefore you can combine one or more JMS providers in a single XA transaction. For example, within the same XA transaction, you can include:

- Receiving a message from WebSphere MQ.
- Logging the message to a database.
- Sending the message to a JMS provider.

6.2 Environment

We developed the scenario in an environment with the following components:

- WebSphere Integration Developer 6.0.2.2
- WebSphere Process Server 6.0.2.0
- WebSphere Application Server 6.0.2.17
- WebSphere Message Broker run time 6.0.0.5
- WebSphere Message Brokers Toolkit 6.0.2 (Interim Fix 009)
- WebSphere MQ 6.0.2.2

6.3 WebSphere Process Server configuration

We define the following items to WebSphere Process Server for this scenario. We performed the configuration using the WebSphere Process Server administrative console.

6.3.1 Defining JMS entries

This scenario requires that JMS entries be defined in WebSphere Process Server. We used the WebSphere administrative console to perform the following tasks:

1. Define a JMS queue connection factory for WebSphere MQ by specifying the properties shown in Figure 6-2 on page 120. The connection factory specifies the information that is required to connect to WebSphere MQ, including the queue manager name, host, port, and channel. The JNDI name is `jms/ESBQCF`.

Archived

Queue connection factories can be viewed and defined from the administrative console by selecting **Resources** → **JMS Providers** → **WebSphere MQ** → **WebSphere MQ queue connection factories**.

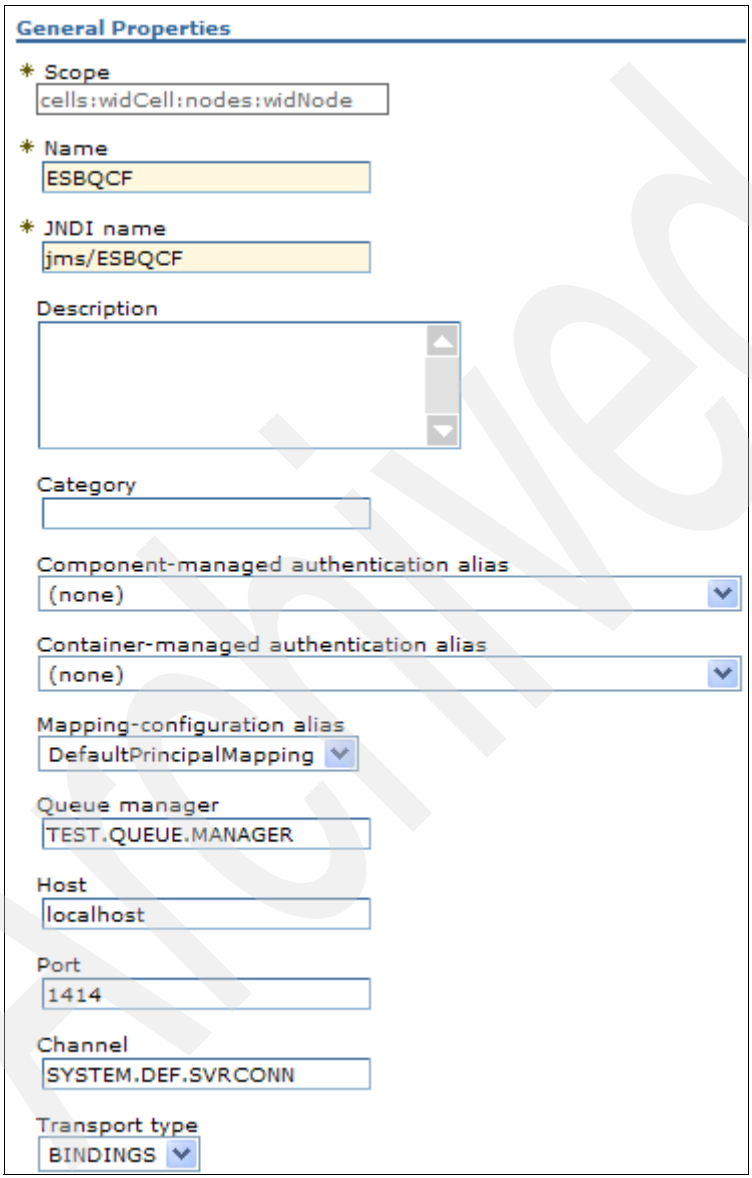


Figure 6-2 jms/ESBQCF queue connection factory

Archived

2. Define a JMS queue for sending the requests by using the properties shown in Figure 6-3. The JMS queue specifies the information that is required to define the TEST/RESQUESTQUEUE WebSphere MQ queue, including the queue manager name and queue name (Figure 6-3). The JNDI name is `jms/RequestQ`.

   JMS queues can be viewed and defined from the administrative console by selecting **Resources** → **JMS Providers** → **WebSphere MQ** → **WebSphere MQ queue destinations**.

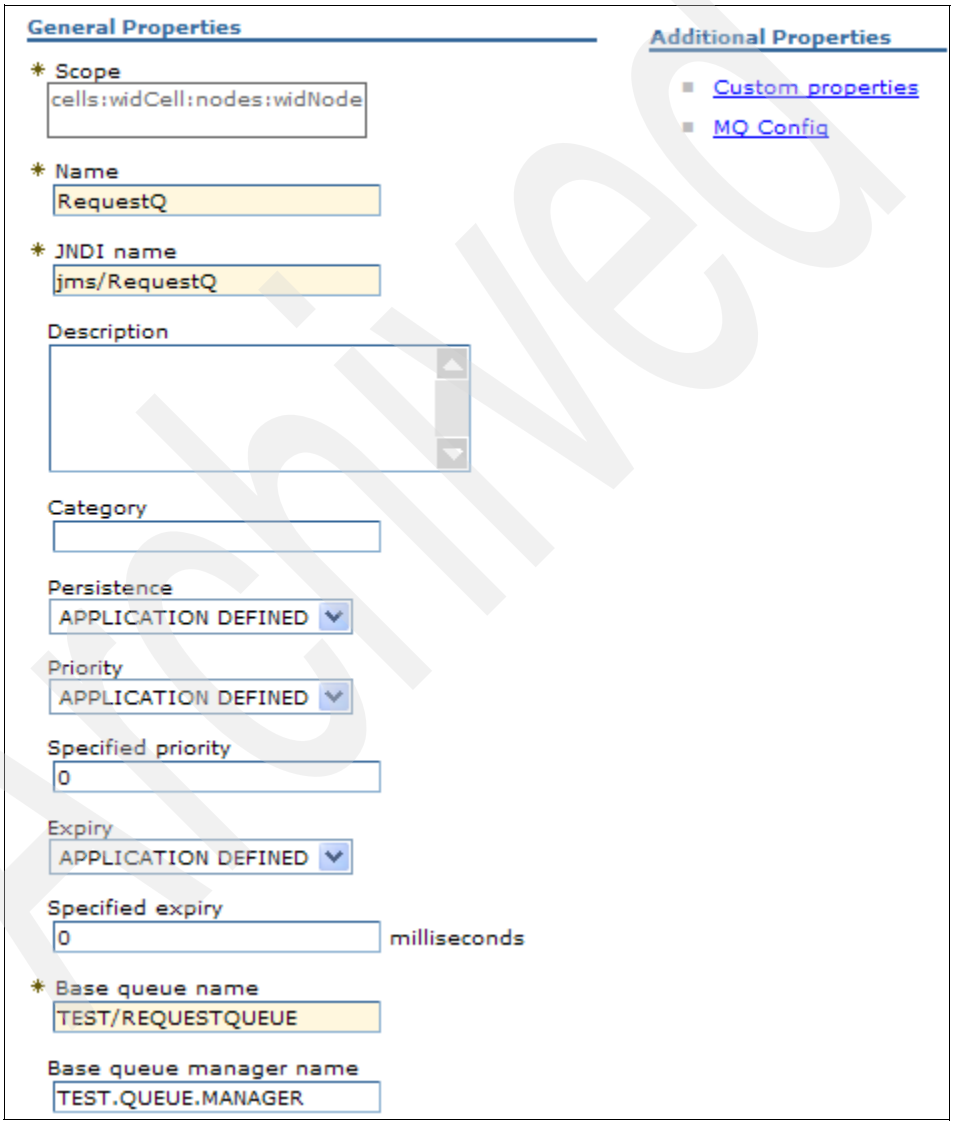


Figure 6-3 jms/RequestQ JMS queue definition

Archived

3. Define a JMS queue for receiving the status of transaction as shown in Figure 6-4.

   The JMS queue specifies the queue manager name and queue name for TEST/RESPONSEQUEUE. The JNDI name is `jms/ResponseQ`.

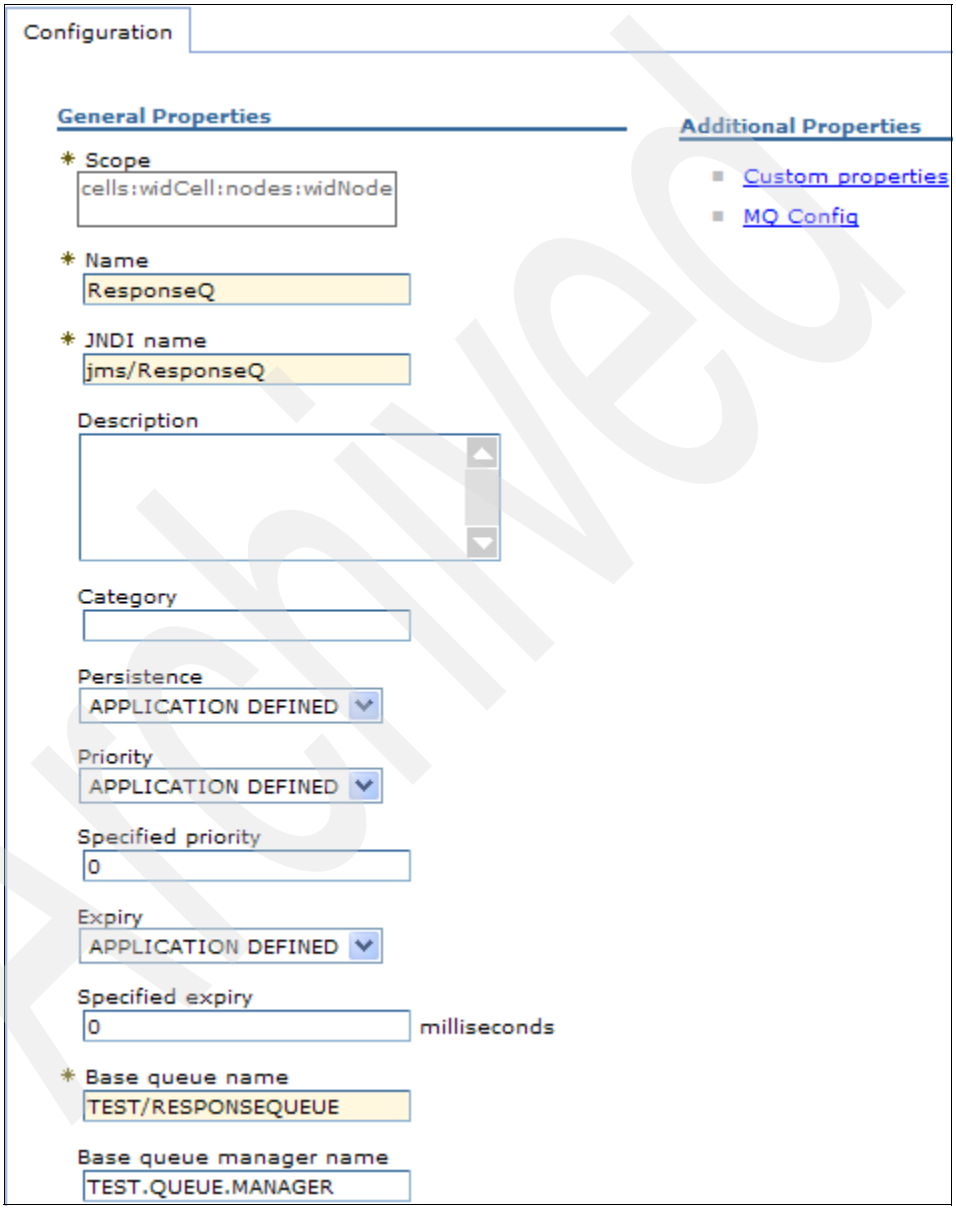


Figure 6-4 jms/ResponseQ JMS queue definition

Archived

6.3.2 Configuring the listener port for the message-driven bean

Since this is a request-reply interaction, a message-driven bean (MDB) is generated internally to listen for a response. A listener port is required for the MDB to listen for messages on a particular queue. Each port specifies the JMS connection factory and JMS queue that an MDB, deployed against that port, will listen to. A message listener port is configured at the application server level.

In this scenario, we configured the message listener port using the properties shown in Figure 6-5. The listener port was configured to listen to `jms/ResponseQ` for response messages from WebSphere Message Broker (Figure 6-5).

Listener ports can be viewed and configured from the administrative console by selecting **Servers** → **Application Servers** → ***server_name*** → **Communications** → **Messaging** → **Message Listeners** → **ListenerPort**.

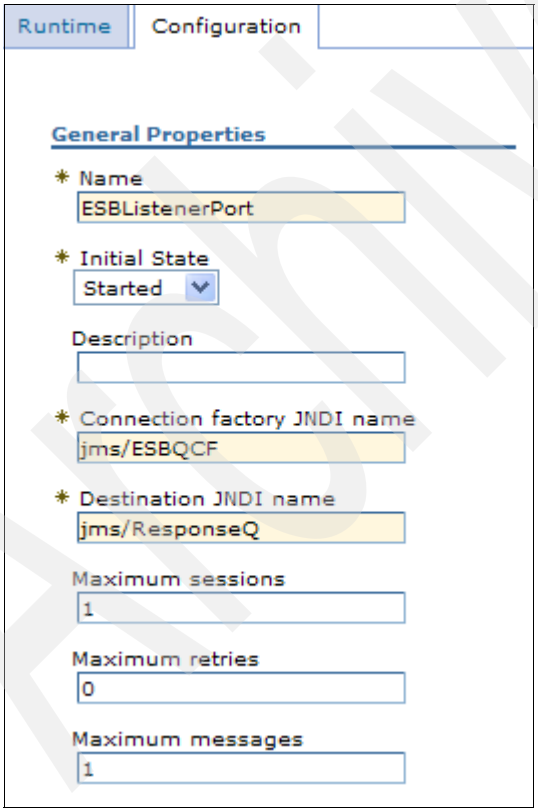


Figure 6-5 JNDI configuration for the listener port

Archived

6.4 WebSphere MQ configuration

The WebSphere MQ JMS providers must be prepared for the business scenario and for global transactions. We performed the following actions, which are explained in the sections that follow:

- Preparing the transaction manager
- Preparing the resource managers
- Making the libraries available
- Configuring the JMS administered objects

6.4.1 Preparing the transaction manager

In this scenario, the queue manager for WebSphere Message Broker, `TEST.QUEUE.MANAGER` manages the global transaction.

We performed the following actions to prepare the queue manager:

1. Create the following queues on TEST.QUEUE.MANAGER:
   - TEST/REQUESTQUEUE

     This queue holds the request from the business process. The JMSInput node in the message flow picks up messages that arrive at this particular queue and starts flow processing.
   - TEST/RESPONSEQUEUE

     The message flow responds back to the business process with the status of the transaction in this queue. The listener port listens on this queue for the response message.
   - DLQ
   - BOQUEUE
2. Define the XA resource managers to the queue manager.

   On distributed systems, the external syncpoint coordinator for the broker is WebSphere MQ.

   Before you deploy a message flow in which the Transaction Coordination property is set to `Global`, modify the queue manager qm.ini file to include definitions for each JMS provider resource manager that participates in globally-coordinated transactions. The resource managers to be involved in global units of work are coordinated by the queue manager. One XAResourceManager stanza is required for each instance of a resource manager type. Since all the resource managers in this scenario are WebSphere MQ, one stanza is required.

Archived

z/OS: On z/OS, the external syncpoint manager is Resource Recovery Services (RRS). Syncpoint control for the JMS provider is managed with RRS syncpoint coordination of the queue manager of the broker. You do not need to modify the qm.ini file.

If the queue manager resides on a Linux® and UNIX® system, add a stanza to the queue manager qm.ini file for each JMS provider, for example:

```
XAResourceManager:
Name=Jms_Provider_Name
SwitchFile=/install_dir/bin/ JMSSwitch.so
XAOpenString=Initial Context,location JNDI,Optional_parms
ThreadOfControl=THREAD
```

On Windows systems, this information is stored in the Windows registry and can be added using the WebSphere MQ Explorer. To define an XA resource manager using the WebSphere MQ Explorer, select **Properties** → **XA resource managers** from the context menu for the queue manager. The Edit XA Resource window (Figure 6-6) that opens.

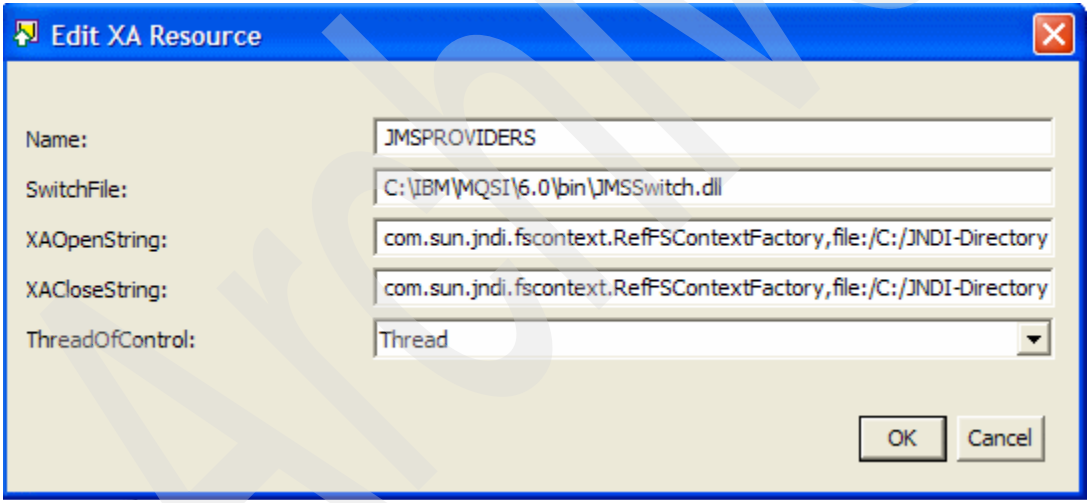


Figure 6-6 XA Resource stanza for the JMS providers

Archived

Complete the fields as follows:

a. In the Name field, type `JMSPROVIDERS`.

   This is the name of the resource manager instance. The name must be unique within one queue manager stanza.

b. For SwitchFile, type *`WebSphere Message Broker Install Directory`*`\MQSI\6.0\bin\JMSSwitch.dll`

   On distributed platforms, an additional component, called the *switch file*, is needed for global transactions. The switch file contains the XA switch structure of the resource manager. The switch file is a shared library (a dynamic link library (DLL) on Windows). The SwitchFile setting defines the path to the JMSSwitch library that is supplied in the bin directory of the broker.

   When the broker's WebSphere MQ queue manager starts, it loads this switch file. The switch file forwards XA/Open transaction calls from the syncpoint coordinator to the JMS provider. (The broker's queue manager acts as the syncpoint coordinator in our scenario.) This ensures that the JMS resources that participate in the transaction can be coordinated in synchronization with other resource managers that are involved in the same transaction.

c. For XAOpenString, type `com.sun.jndi.fscontext.RefFSContextFactory,file:/C:/JNDI-Directory`.

   You enter a string of data that WebSphere MQ passes in its calls to the resource manager's xa_open function. WebSphere MQ and the queue manager call the xa_open function when the queue manager starts and when you make the first call to MQBEGIN in your WebSphere MQ application process. In this example, a file-based initial context factory has been used for WebSphere MQ JMS provider.

   The XAOpenString parameters are specified as:

   `Initial_Context,Location_JNDI,Optional_Parms`

   The values for XAOpenString are as follows:

   - *Initial Context* is the value that is set in the JMSInput node basic property Initial Context Factory.
   - *Location JNDI* is the value that is set in the JMSInput node basic property Location of JNDI. This value should include the leading keyword, such as file:/, iiop:/, or ldap:/.

   Important: The values for the Initial Context factory and Location of JNDI bindings in the stanza must match those specified in the JMSInput or JMSOutput nodes in the message flows.

- *Optional_Parms* as the name suggests refers to optional parameters. The optional parameters are comma-separated and positional. Therefore, any parameters that are missing must be represented by a comma. They could be of the following types:
  - *LDAP Principal* matches the value that is set for the broker by using the **mqsicreatebroker** or **mqsichangebroker** commands.
  - *LDAP Credentials* matches the value that is set for the broker by using the **mqsicreatebroker** or **mqsichangebroker** commands.
  - *Recovery Connection Factory Name* is the JNDI administered connection factory that is defined in the bindings file.

    The recovery connection factory name must match a queue connection factory name in the JNDI administered objects.

  If a value is not specified, you must add a generic queue connection factory called `recoverXAQCF` to the bindings file. You can see this in "Creating the queue connection factories" on page 131. In this example, we have omitted this value.

  Whether the name is specified or a default is used, the recovery connection factory should be defined as an XA queue connection factory for the JMS provider that is associated with the initial context factory.

d. For XACloseString, type `com.sun.jndi.fscontext.RefFSContextFactory, file:/C:/JNDI-Directory`

   You enter the string to be passed to the resource manager's xa_close entry point. The entry should match the values provided for the XAOpenString.

e. For ThreadOfControl, type `Thread`.

   The queue manager uses this parameter for serialization when calling the resource manager from a multi-threaded process. A value of `Thread` indicates that the resource manager is "thread aware." XA function calls can be made to the external resource manager from multiple threads at the same time.

Archive

6.4.2 Preparing the resource managers

Two WebSphere MQ queue managers are used to receive the JMS messages and participate in the global transaction.

JMSPROV1

Queue manager JMSPROV1 uses port 1515. The CREDITQ queue was created on JMSPROV1.

JMSPROV1

Queue manager JMSPROV2 uses port 1616. The DEBITQ queue was created on JMSPROV2.

6.4.3 Making the libraries available

To make the libraries available, we performed the following steps:

1. Update the Java CLASSPATH environment variable for the broker's queue manager to include a reference to xarecovery.jar, for example:

   broker_install_dir/classes/xarecovery.jar

2. Update the Java CLASSPATH environment variable for the broker's queue manager to include a reference to xahandler.jar and com.ibm.mqetclient.jar.

3. Update the Java PATH environment variable for the broker's queue manager to point to the bin directory where the switch file is located, for example:

   broker_install_dir/bin

4. The JMS provider might supply additional Java archive (JAR) files that are required for transactional support. See the documentation that is supplied with the JMS provider for more information. You must add any additional JAR files to the broker's shared-classes directory.

   For example, on platforms other than z/OS, the WebSphere MQ JMS provider supplies an extra JAR file, com.ibm.mqetclient.jar. Ensure that this JAR file is in the shared-classes directory. On Windows, this directory is C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes.

On z/OS, there is no shared-classes library. We must configure the broker CLASSPATH and LIBPATH variables in the BIPBROF member of the broker partitioned data set. First, we add each JMS provider JAR file, for example, the WebSphere MQ JMS JAR files, in the CLASSPATH. Next we update the LIBPATH with any native libraries, for example, the directory location for the WebSphere MQ JMS native libraries. Finally, we submit the BIPGEN JCL job to update the broker ENVFILE with these new values.

Archived

6.4.4 Configuring the JMS administered objects

We created a bindings file to hold the JMS administered objects. There are two options to create these objects:

- Use the new WebSphere MQ Explorer option in WebSphere MQ V6.0.2.2 to configure the JMS administered objects using a GUI. We used this method in this scenario to create the bindings.
- Use the JMSAdmin.bat utility provided with WebSphere MQ installation.

The bindings file is configured so that a JMS client can connect to and access administered objects in the WebSphere MQ JNDI namespace.

We performed the tasks in the following sections in WebSphere MQ Explorer on the broker's WebSphere MQ queue manager, which acts as the syncpoint transaction coordinator.

Archived

Adding the initial context and connecting to the JNDI namespace

To create and configure JMS objects in WebSphere MQ Explorer, we added an initial context to define the root of the JNDI namespace in which the JMS objects are stored in the naming and directory service. For more information, see “Adding an initial context” in WebSphere MQ Help.

1. From WebSphere MQ Explorer, select **JMS Administered Objects** → **Add Initial Context**.
2. In the Connection details window (Figure 6-7), select a file system to hold the administered objects.
3. Click **Finish** to add the context and connect to the JNDI namespace.

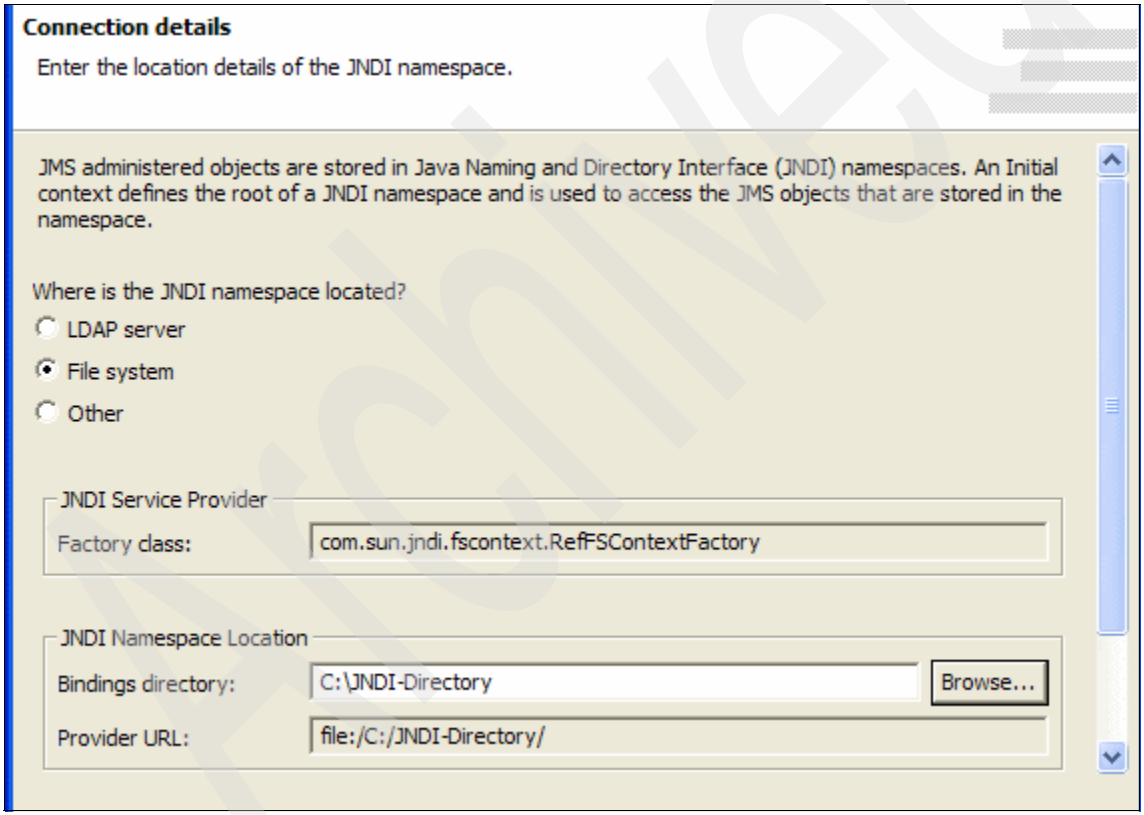


Figure 6-7 Adding the initial context

Creating the queue connection factories

We created the following connection factories for this scenario. With the exception of the parameters specified, accept the default options.

From WebSphere MQ Explorer, follow these steps:

1. Select **JMS Administered Objects** → ***bindings directory*** → **Connection Factories** → **New** → **Connection Factory**.
2. Type the name `jms_JMSPROV1`, and select **WebSphere MQ** as the messaging provider. Click **Next**.
3. In the next window (Figure 6-8), select **Queue Connection Factory** as the type and make sure that **Support XA transactions** is selected. Click **Next**.

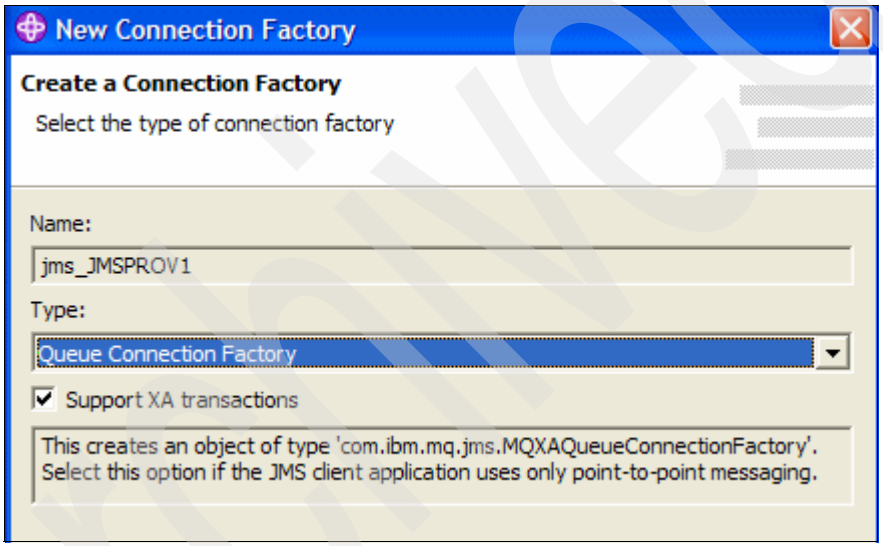


Figure 6-8 Creating an XA queue connection factory for queue manager JMSPROV1

Archived

4. In the next window (Figure 6-9), for Transport, select **Bindings** and click **Next**.

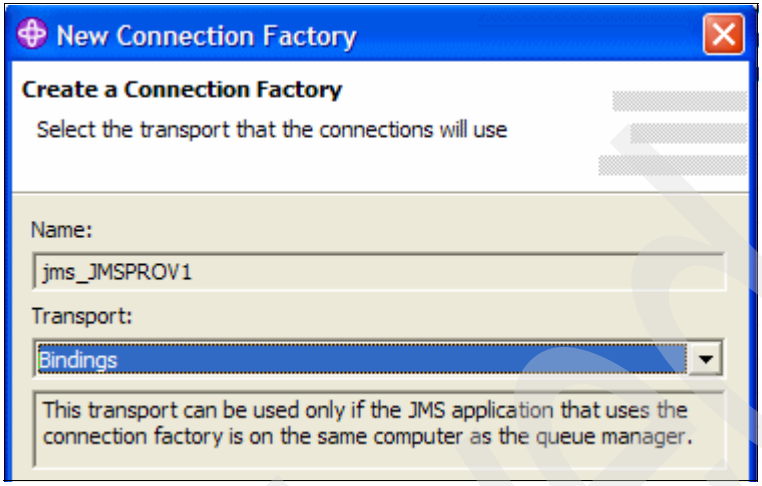


Figure 6-9 Select Transport option - Bindings

Important: The JMSInput node is always on the same machine as the broker queue manager and always uses the Bindings transport. However, the queue managers for the JMSOutput nodes (JMSPROV1 and JMSPROV2) might be on a system that is remote to the message flow. For the connection factories for these providers, select **MQ Client**. This means that the connection uses TCP/IP.

If you select MQ Client as the transport and you selected the Support XA transactions check box on the previous page of the wizard (Figure 6-8 on page 131), you must install the WebSphere MQ Extended Transactional Client on the WebSphere Message broker system. Refer to the following resources for instructions about installing this component:

- "Installing the WebSphere MQ client" topic in the WebSphere MQ Information Center

  `http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp?topic=/com.ibm.mq.csqzaf.doc/amq2512.htm`

- WebSphere MQ Extended Transactional Client Web page

  `http://www.ibm.com/software/integration/wmq/transclient.html`

- "Units of work and transactions" in *WebSphere MQ V6 Fundamentals*, SG24-7128

5. On the next page, accept the defaults and click **Next**.

6. In the Properties window (Figure 6-10), click **Connection** on the left, and for Base queue manager, type `JMSPROV1`. Then click **OK**.

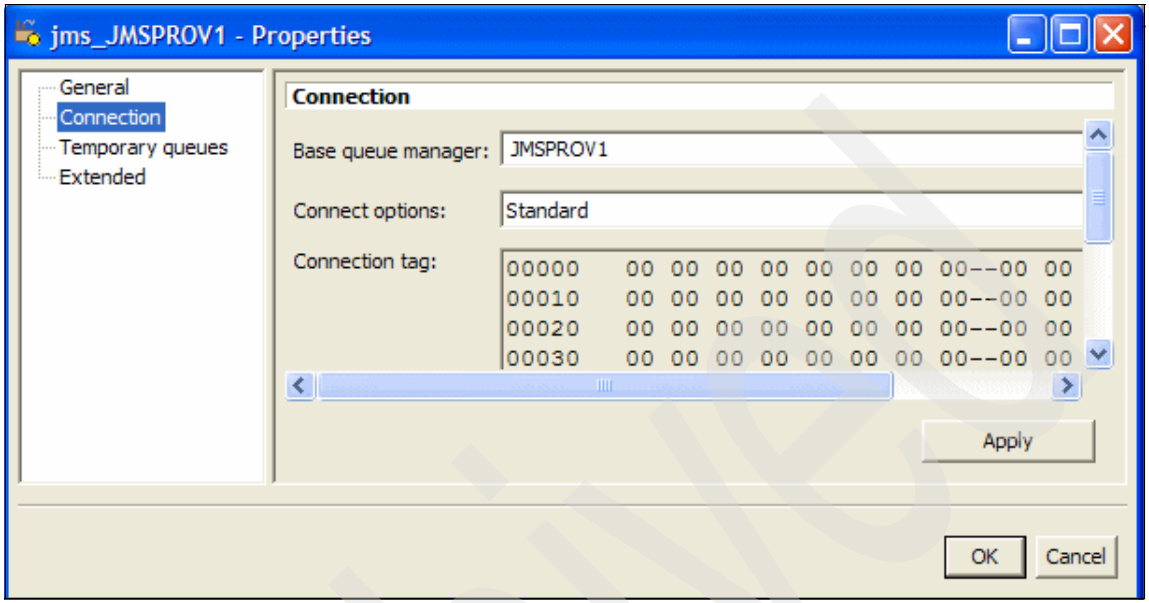


Figure 6-10 Specifying the base queue manager

7. On the next window (not shown), click **Finish**.

> **z/OS**: The only JMS provider that is supported on z/OS is WebSphere MQ JMS. The only transport option that is supported for WebSphere MQ JMS on z/OS is the Bind option.

8. Repeat the process as explained in step 1 on page 131 through step 7 to add the following connection factories.
   a. Name: `jms_PROV2`:
      - Transport: **Bindings**
      - Base queue manager: `JMSPROV2`
   b. Name: `jms_ESBQCF`:
      - Transport: **Bindings**
      - Base queue manager: `TEST.QUEUE.MANAGER`
   c. Name: `recoverXAQCF`:
      - Transport: **Bindings**
      - Base queue manager: `JMSPROV1`

Figure 6-11 shows a summary of the JMS connection factory definitions.

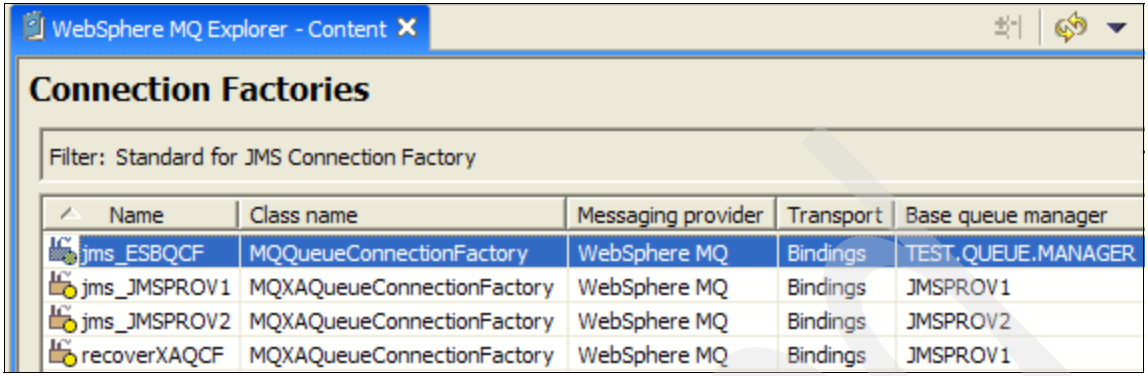


Figure 6-11 Summary of connection factories created

Note that the connection factories for the queue managers that are participating in the global transaction have a class name of MQXAQueueConnectionFactory. This was a result of selecting the Support XA transactions option.

Creating the JMS destinations

In a similar fashion, we configured the destinations. To open the wizard to create each destination, select **JMS Administered Objects** → ***bindings directory*** → **Destinations** → **New** → **Destination**.

We defined each of the following destinations. With the exception of the values that are specified, we accepted the defaults.

1. Name: `jms_ReqQ:`
   - Queue manager: `TEST.QUEUE.MANAGER`
   - Queue: `TEST/REQUESTQUEUE`
2. Name: `jms_RespQ:`
   - Queue manager: `TEST.QUEUE.MANAGER`
   - Queue: `TEST/RESPONSEQUEUE`
3. Name: `jms_boq`
   - Queue manager: `TEST.QUEUE.MANAGER`
   - Queue: `BOQUEUE`
4. Name: `jms_creditq`
   - Queue manager: `JMSPROV1`
   - Queue: `CREDITQ`

Archive

5. Name: `jms_debitq`
   - Queue manager: `JMSPROV2`
   - Queue: `DEBITQ`

Figure 6-12 shows a summary of the JNDI queue destinations.

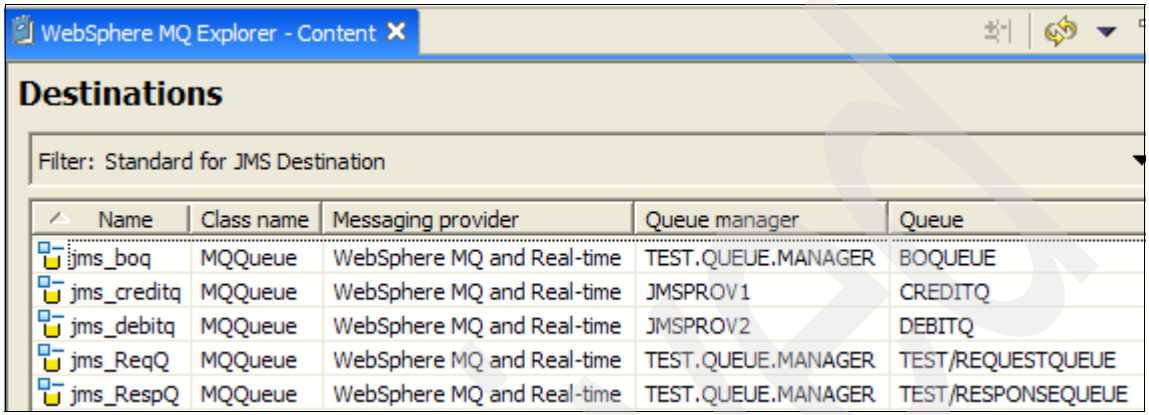


Figure 6-12 Summary of queues defined in the JNDI

6.5 Business integration module

In this section, we demonstrate the implementation of a business integration module that is deployed on WebSphere Process Server. This module receives a message from the export component, triggering a short-running business process. The process sends a JMS message to a queue that is configured on the queue manager of WebSphere Message Broker. After sending the message to a flow deployed in message broker, it listens on a response queue, again configured on the queue manager of WebSphere MQ.

6.5.1 Business integration module components

The business process is implemented in a business integration module called TransferFundsModule. Figure 6-13 shows the assembly diagram for this module.

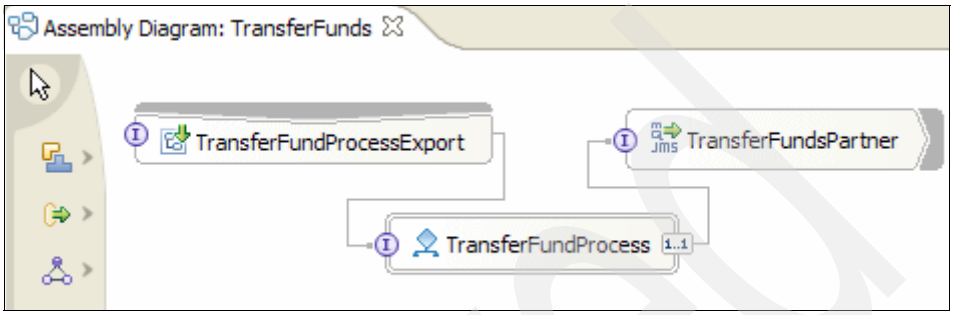


Figure 6-13 TransferFunds module assembly diagram

This diagram consists of the following components:

- *TransferFundProcessExport* is the export component that the client uses to access the business process.
- *TransferFundProcess* contains the business process implementation.
- *TransferFundsPartner* is the import component that sends messages to a service via WebSphere Message Broker.

All three components use a single interface, TransferFundInterface.

Archive

6.5.2 Business objects

The module uses two business objects, one for the request object and one for the response object:

- Figure 6-14 shows the TransferReqBO business object. This business object has the `TransferId`, `SourceBank`, `TargetBank`, and `Amount` string attributes, and a namespace of `http://TransferFunds`.

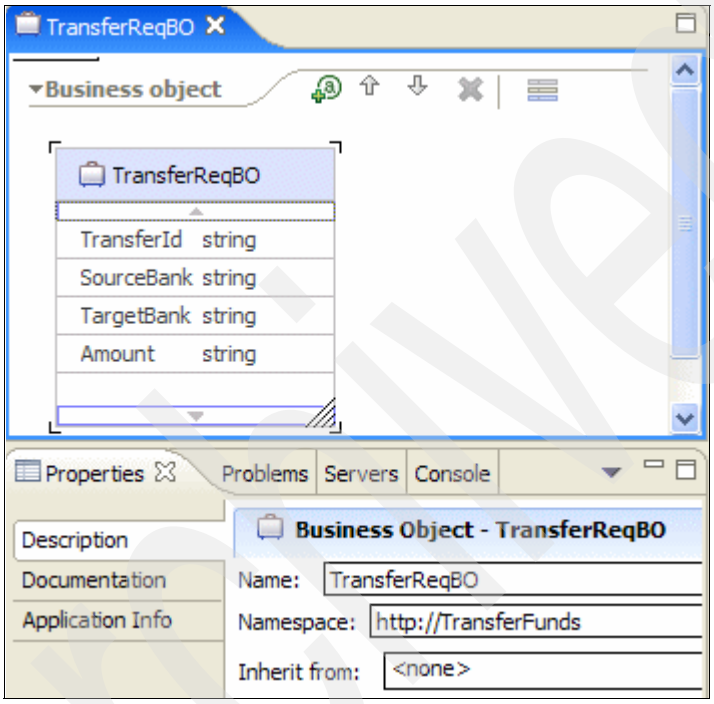


Figure 6-14 TransferReqBO

Archived

- Figure 6-15 shows the TransferStatusBO business object. This business object has the `TransferId` and `Status` string attributes and a namespace of `http://TransferFunds`. The `status` attribute indicates the status of the transaction as returned by the message flow in WebSphere Message Broker.

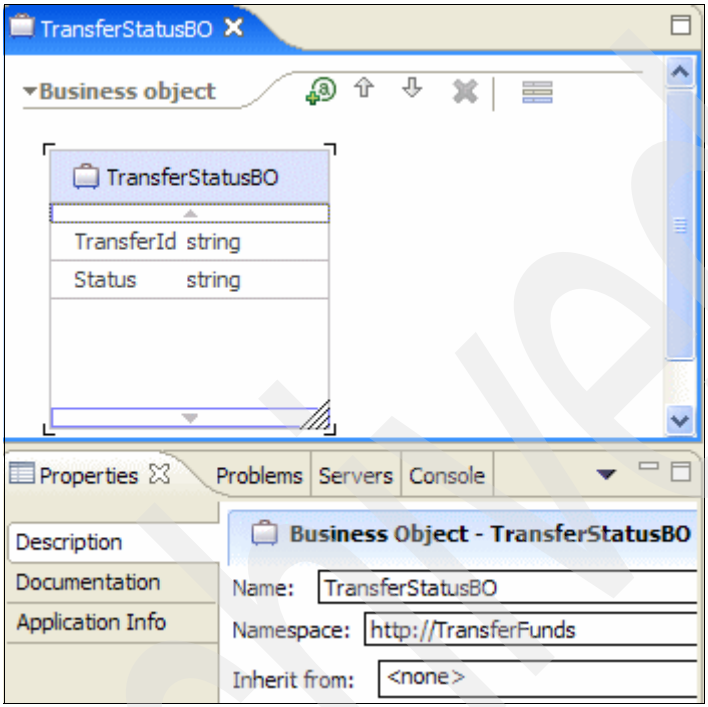


Figure 6-15 TransferStatusBO Business Object

Archived

6.5.3 Interfaces

The module uses one interface, TransferFundInterface, with a request-response operation as shown in Figure 6-16.

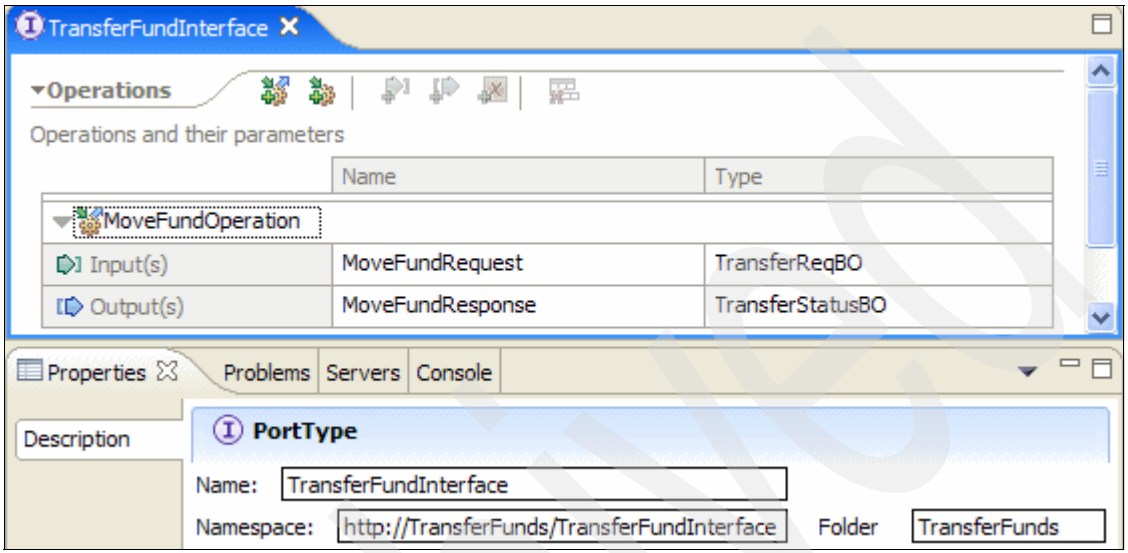


Figure 6-16 TransferFundInterface

The Input type is TransferReqBO (Figure 6-14 on page 137), and the output type is TransferStatusBO (Figure 6-15 on page 138).

Archived

6.5.4 Business process

Figure 6-17 shows the TransferFundProcess business process.

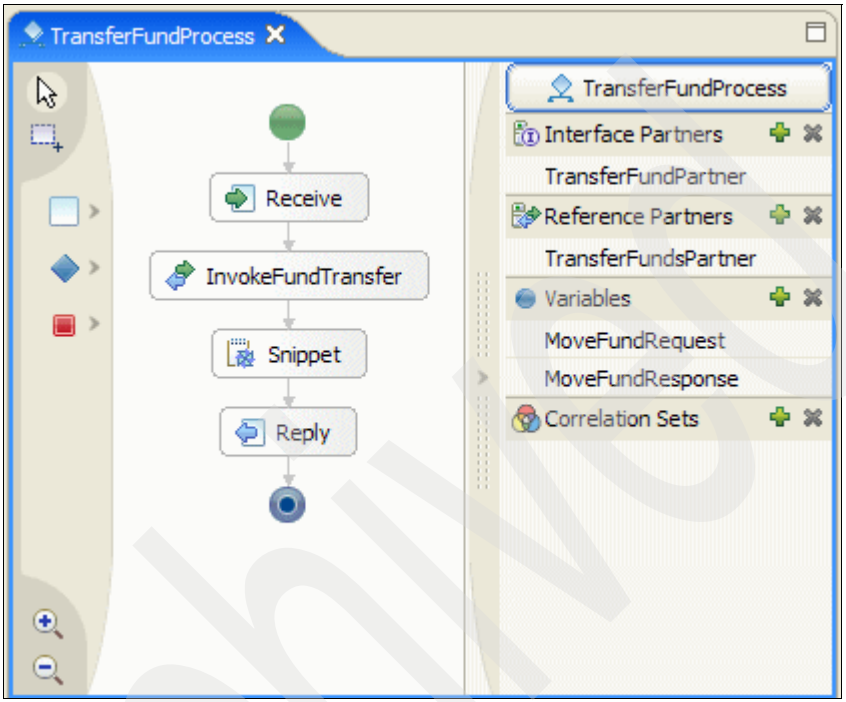


Figure 6-17 The TransferFund business process

The properties for the business process are the default properties.

Interface and partner references

The business process uses one interface partner and one reference partner. You can see these listed to the right of the business process in Figure 6-17.

An *interface partner* is the process interface and exposes operations that can be called by external partners. The interface partner for this process, TransferFundPartner, uses the TransferFundInterface interface.

A *reference partner* specifies the interface that is used in the invocation of another service. TransferFundsPartner is defined as a reference partner and specifies that the TransferFundInterface interface is to be used to call the service. In the assembly diagram, this reference partner is wired to the TransferFundsPartner import component.

As shown in Figure 6-18, both the interface partner and reference partner use the same interface, because just one operation is being invoked.

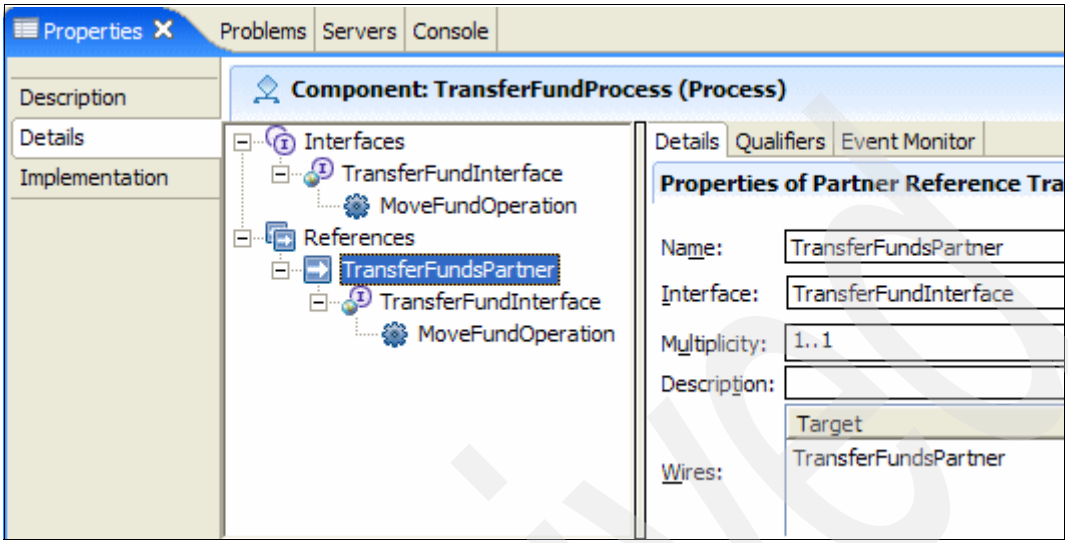


Figure 6-18 Business process references and interfaces

Receive activity

The Receive activity (Figure 6-19) receives the request from the client. It is associated with the TransferFundInterface interface.

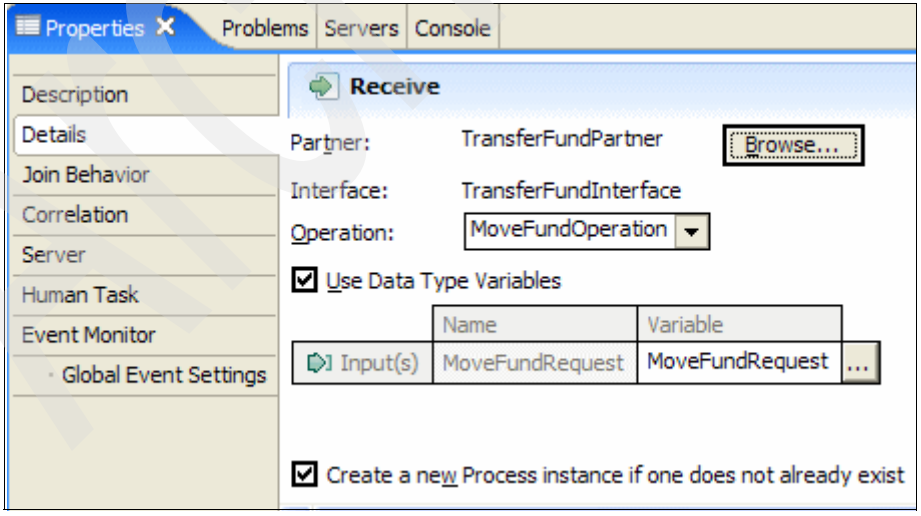


Figure 6-19 Receive activity

InvokeFundTransfer Invoke activity

The InvokeFundTransfer Invoke activity (Figure 6-20) is responsible for sending the request JMS message to the queue on WebSphere MQ. The Invoke activity is associated with the TransferFundsPartner.

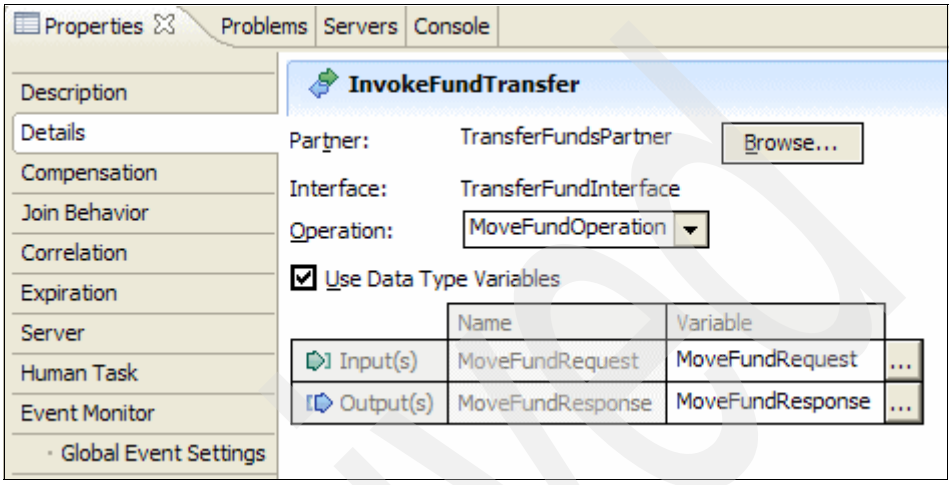


Figure 6-20 InvokeFundTransfer Activity

Java snippet (optional)

A Java snippet (optional) component displays the results of the invoke operation.

Reply

The Reply activity (Figure 6-21) sends back the response to the client. It is associated with the TransferFundInterface interface.

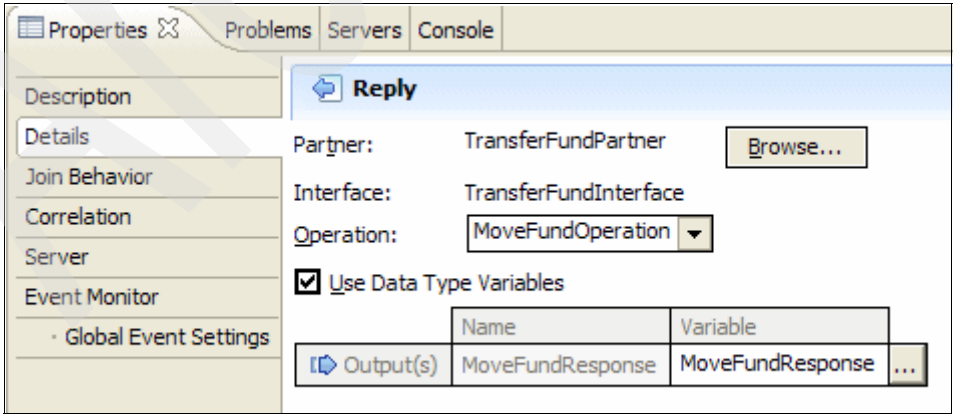


Figure 6-21 Reply activity

6.5.5 Import component and binding

The business process uses an import component to send a JMS message to WebSphere Message Broker. This import component has the TransferFundInterface interface (Figure 6-16 on page 139). The import has a MQ JMS binding with the settings shown in Figure 6-22.

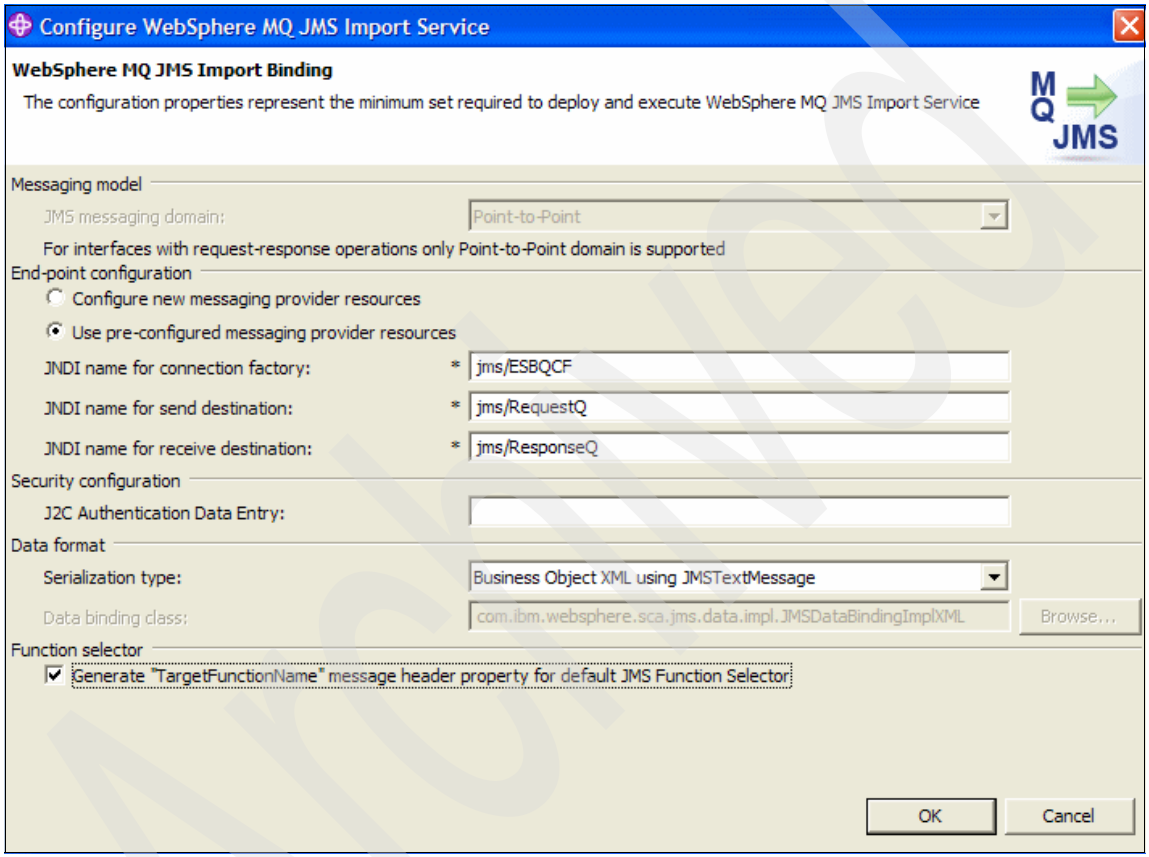


Figure 6-22 WebSphere MQ JMS binding on the import

We highlight the following settings:

- For JMS Messaging Domain, select **Point-to-Point**.

  The only supported choice for request-response operations is Point-to-Point.

- For JNDI name for connection factory, type `jms/ESBQCF`.

  This field contains the JNDI name of the queue connection factory that is configured to WebSphere Process Server (Figure 6-2 on page 120).

- For JNDI name for send destination, type `jms/RequestQ`.

  This name is the JNDI name of the destination queue configured to WebSphere Process Server (Figure 6-3 on page 121).

- For JNDI name for receive destination, type `jms/ResponseQ`.

  This name is the JNDI name of the receive destination that is configured to WebSphere Process Server (Figure 6-4 on page 122) for receiving the response from WebSphere Message Broker flows.

- For Serialization type, select **Business Object XML using JMSTextMessage**.

  This serialization type supports JMSTextMessage. It serializes the DataObject into an XML document that conforms to the schema of the DataObject and sets it to the text field of the JMSMessage, or parses an XML document and sets it into the DataObject. This class extends the general purpose data binding class com.ibm.ws.sca.databinding.impl. DataBindingImplXML, which provides the function to convert an XML string to a DataObject and vice versa.

  For more information about various types of serialization options, see “JMS import data bindings” on page 68.

6.6 Message flows

In this section, we demonstrate the implementation of the message flow that is deployed on WebSphere Message Broker. This scenario uses one message flow called JMSREQRESP and two message sets.

Note: During the creation of this scenario, we used information from the following Education Assistant module for debugging problems:

http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wmb_v6/wmb/6.0/ApplicationDevelopment/V6_Enhancements/JMS/player.html

6.6.1 PrjTxnFlowMSets message set

You must have message definitions for any messages that you want to include in a message mapping. Two message definitions are used by the message flow, one for each business object, TransferReqBO and TransferStatusBO, that is defined for the TransferFund business integration module. The XML Schema

Definition (XSD) files for these business objects have been imported into the message set as message definition files (.mxsd files).

Figure 6-23 shows the outline of the message definition for TransferReqBO.

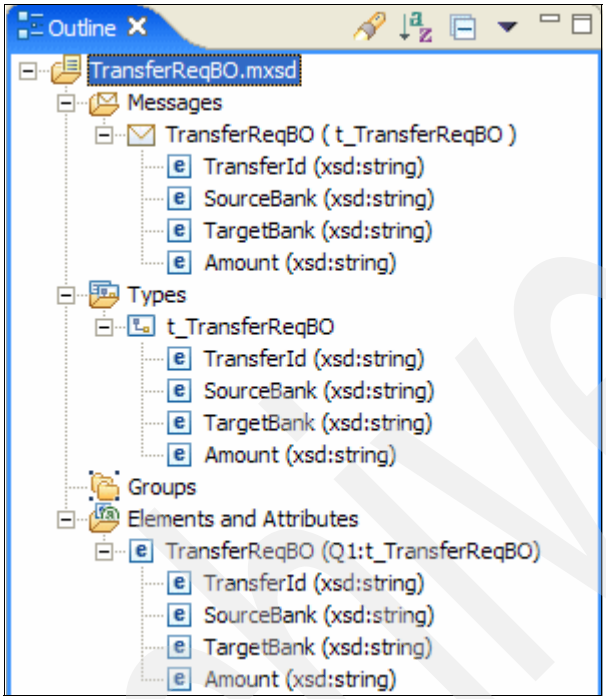


Figure 6-23 TransferReqBO message definition

Archived

Figure 6-24 shows the outline of the message definition for TransferStatusBO.

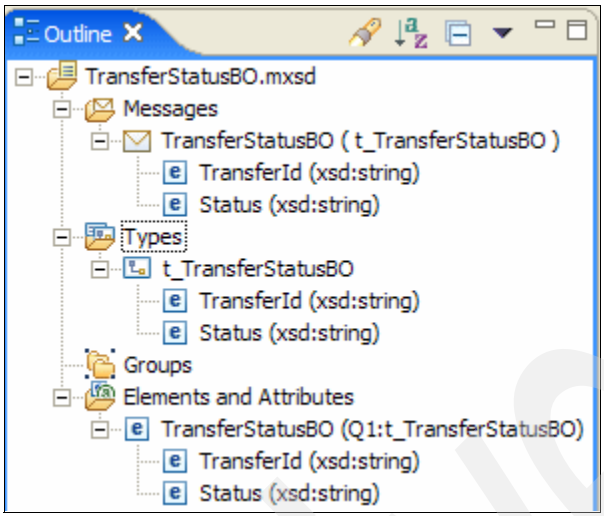


Figure 6-24 TransferStatusBO message definition

Figure 6-25 shows the message set. The message domain is XMLNS. Notice that the Use namespaces option is selected, which indicates that the message definitions that you create within the message set are XML namespace aware.

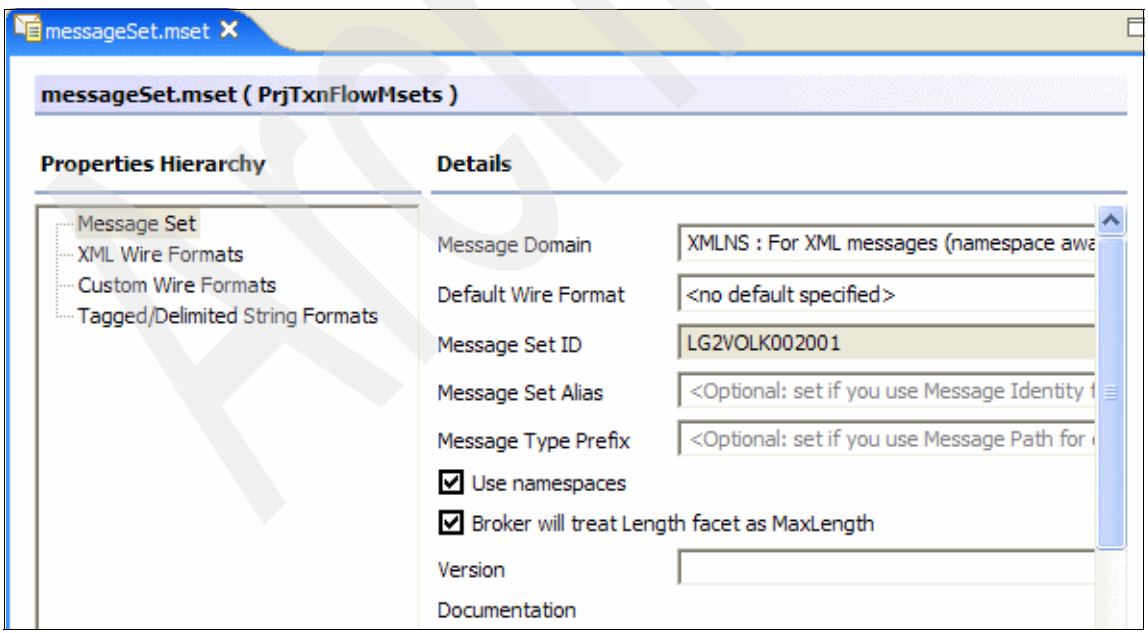


Figure 6-25 Message set definition

Archived

6.6.2 JMSREQRESP message flow

Figure 6-26 shows the JMSREQRESP message flow.

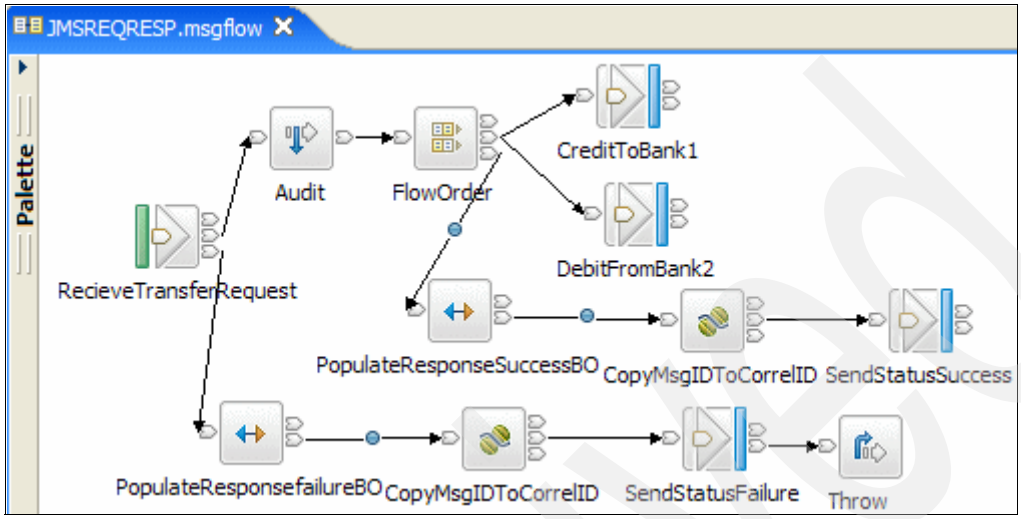


Figure 6-26 JMSREQRESP message flow

RecieveTransferRequest JMSInput node

The RecieveTransferRequest JMSInput node is responsible for pulling messages from WebSphere MQ queue manager on the WebSphere Process Server. Figure 6-27 shows the basic properties for the node. The source queue for messages is jms_ReqQ. The JMS destination for this queue was defined in "Creating the JMS destinations" on page 134. It corresponds to TEST/REQUESTQUEUE on TEST.QUEUE.MANAGER.

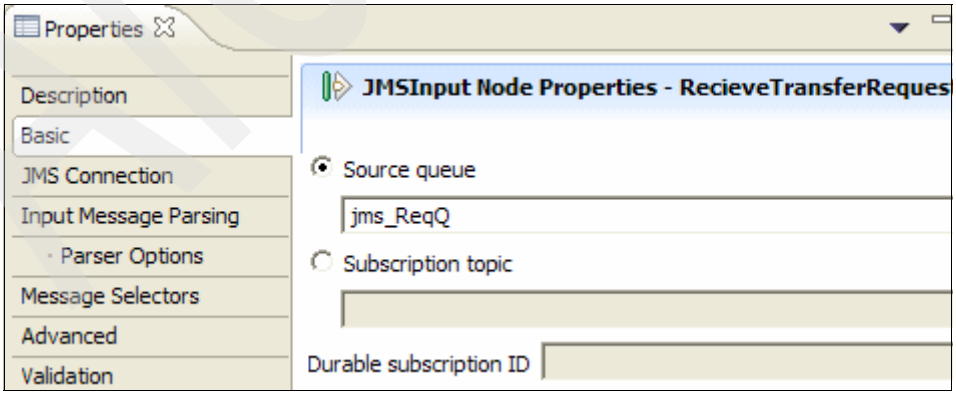


Figure 6-27 ReceiveTransferRequest - Basic tab

Figure 6-28 shows the JMS Connection properties.

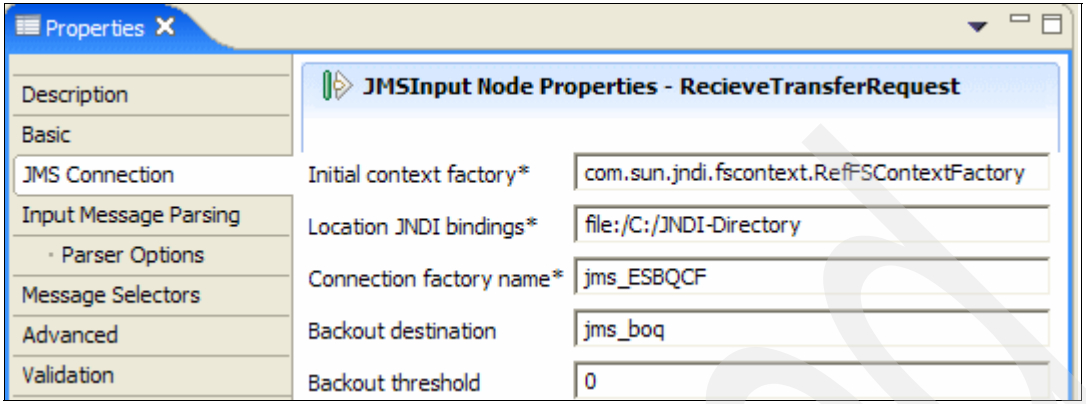


Figure 6-28 ReceiveTransferRequest - JMS Connection

In this window, we highlight the following fields:

- For Initial context factory, type `com.sun.jndi.fscontext.RefFSContextFactory`.

  This field is used to obtain and look up the JNDI administered objects for the JMS provider. The value `com.sun.jndi.fscontext.RefFSContextFactory` defines the file-based initial context factory for the WebSphere MQ JMS provider. This value matches the value that is used when you defined the objects to WebSphere MQ (Figure 6-7 on page 130).

- For Location JNDI bindings, type `file:/C:/JNDI-Directory`.

  This field specifies the file system path for the bindings file. This value matches the value that is used when you defined the objects to WebSphere MQ (Figure 6-7 on page 130).

- For Connection factory name, type `jms_ESBQCF`.

  The connection factory was defined in “Creating the queue connection factories” on page 131.

- For Backout destination, type `jms_boq`.

  The JMS destination for this queue was defined in “Creating the JMS destinations” on page 134.

Archived

The input message parsing settings (Figure 6-29) define the XMLNS message domain.

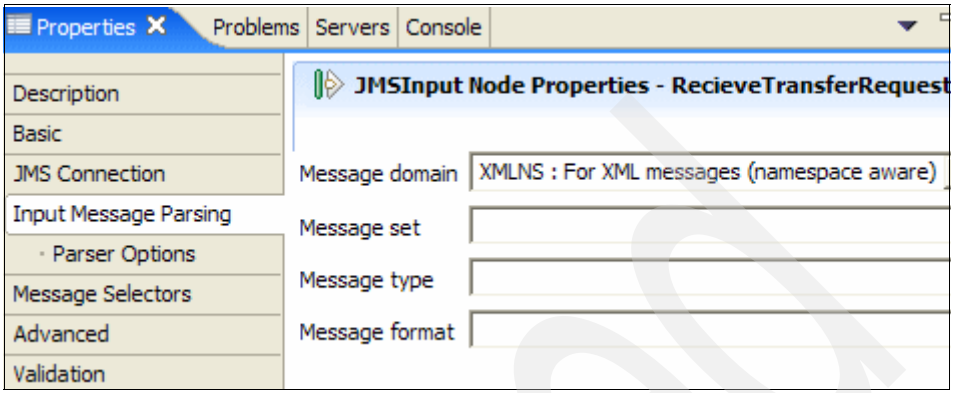


Figure 6-29 ReceiveTransferRequest - Input Message Parsing

The transaction mode is set to local (Figure 6-30). The message is received under the local syncpoint control of the JMSInput node. The JMSInput node coordinates the commit or roll back of JMS messages that are received by the node, along with any other resources such as WebSphere MQ that perform work within the message flow. If you select this value, the node uses a transacted JMS session.

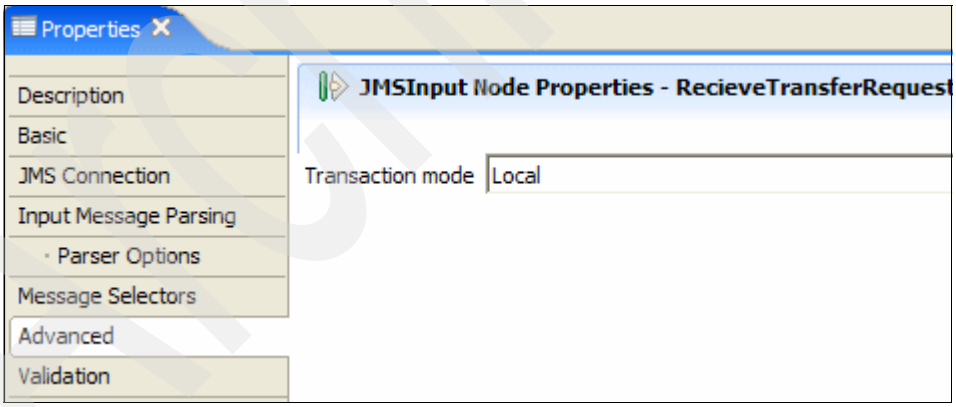


Figure 6-30 ReceiveTransferRequest - Advanced

FlowOrder node

The FlowOrder node is used to control the order in which the message is processed by a message flow:

1. The input message is propagated to the first output terminal, which is connected to the `CreditToBank1` and `DebitFromBank2` nodes. The order in which the message is sent to these two nodes is random. However, all must complete successfully before the message is propagated to the nodes on the second terminal.
2. If the processing is successful, the input message is propagated to the second output terminal, and the sequence of nodes, which starts with PopulateResponseSuccessBO, processes the message.

DebitFromBank2 JMSOutput node

The DebitFromBank2 JMSOutput node is responsible for sending the message to the JMS provider JMSPROV2. Figure 6-31 shows the basic properties. The destination queue is jms_debitq. The JMS definition for this queue was defined in “Creating the JMS destinations” on page 134. It corresponds to DEBITQ on JMSPROV2.

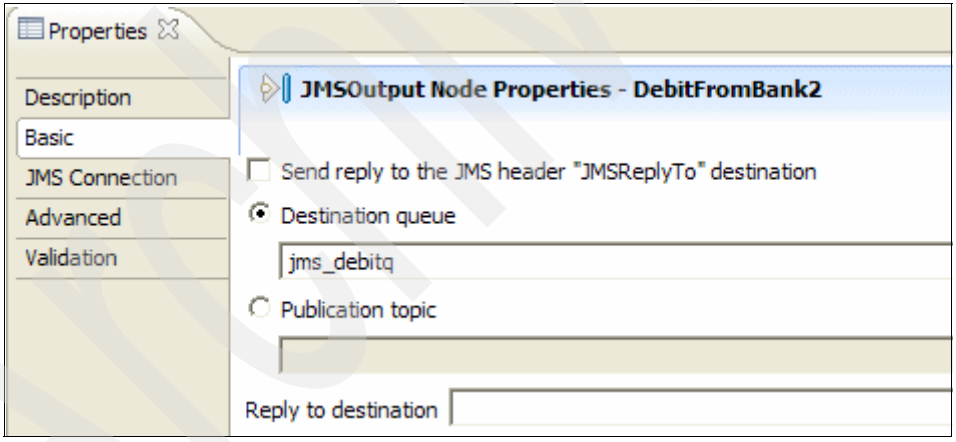


Figure 6-31 DebitFromBank2 - Basic

Figure 6-32 shows the JMS connection properties. The connection factory name is jms_JMSPROV2, which corresponds to the JMSPROV2 base queue manager. The connection factory was defined in “Creating the queue connection factories” on page 131.

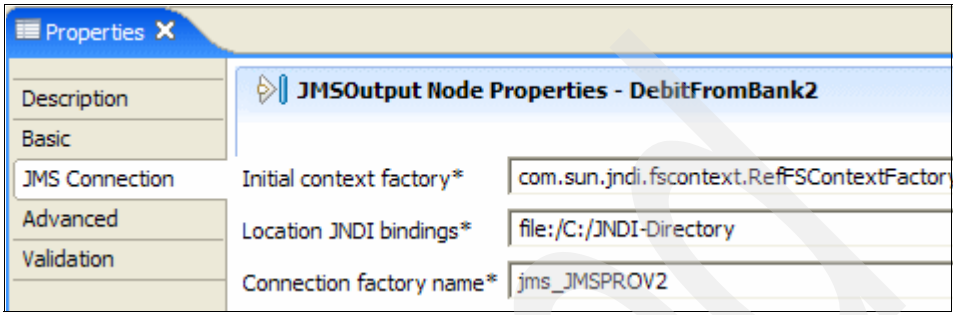


Figure 6-32 DebitFromBank2 - JMS Connection

Since the JMSOutput node is part of a global transaction, the transaction mode property should be specified as `Global` as shown in Figure 6-33. This means that the message is sent under external syncpoint coordination, within a WebSphere MQ unit of work. Any messages that are sent subsequently by an output node in the same instance of the message flow are put under sync point, unless the output node overrides this setting explicitly.

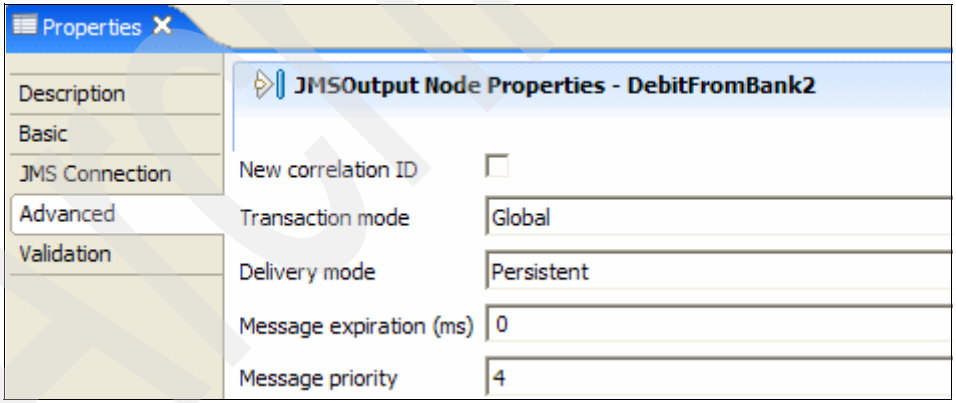


Figure 6-33 DebitFromBank2 - Advanced

CreditToBank1 JMSOutput node

The CreditToBank1 JMSOutput node is responsible for sending the message to the JMS provider JMSPROV1. Figure 6-34 shows the basic properties. The destination queue is jms_creditq. The JMS definition for this queue was defined in “Creating the JMS destinations” on page 134. It corresponds to CREDITQ on JMSPROV1.

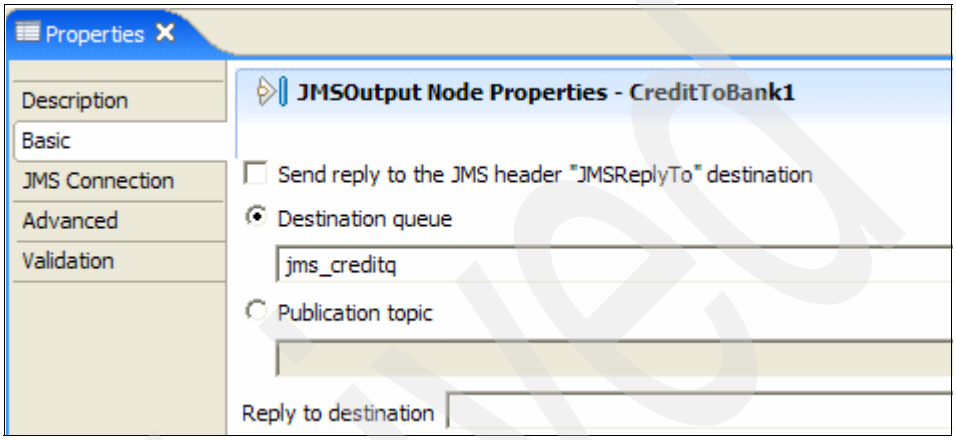


Figure 6-34 CreditToBank1 - Basic

Figure 6-35 shows the JMS connection properties. The connection factory name is jms_JMSPROV1, which corresponds to the JMSPROV1 base queue manager. The connection factory was defined in “Creating the queue connection factories” on page 131.

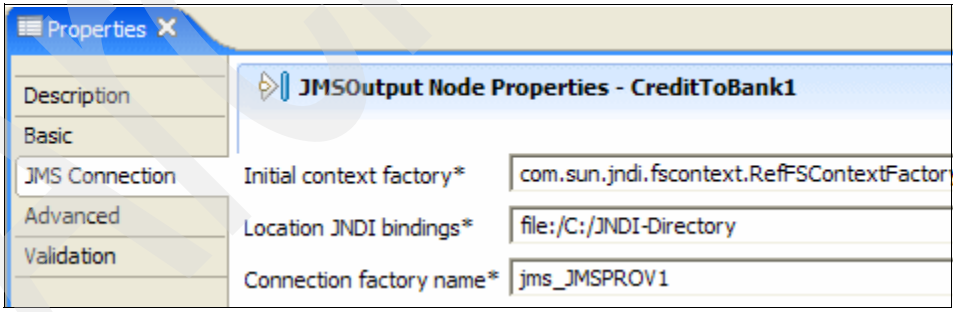


Figure 6-35 CreditToBank1 - JMS Connection

Archived

The JMSOutput node CreditToBank1 is also part of the global transaction. Therefore, the Transaction mode property is set to `Global` (Figure 6-36).

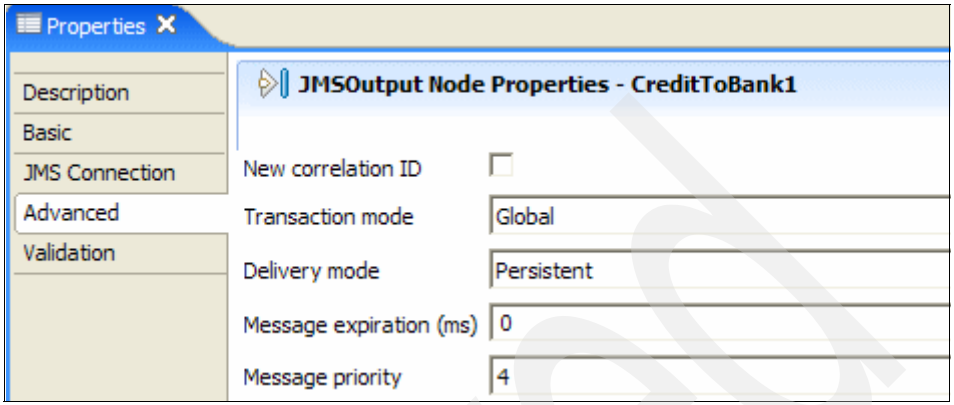


Figure 6-36 CreditToBank2 - Advanced

PopulateResponseSuccessBO Mapping node

The Mapping node maps the status of the transaction to the response business object TransferStatusBO and copies the transaction ID from the input message to the output message. A message map is created for transforming from the source message format to the target message format.

Figure 6-37 shows the basic properties of the Mapping node.

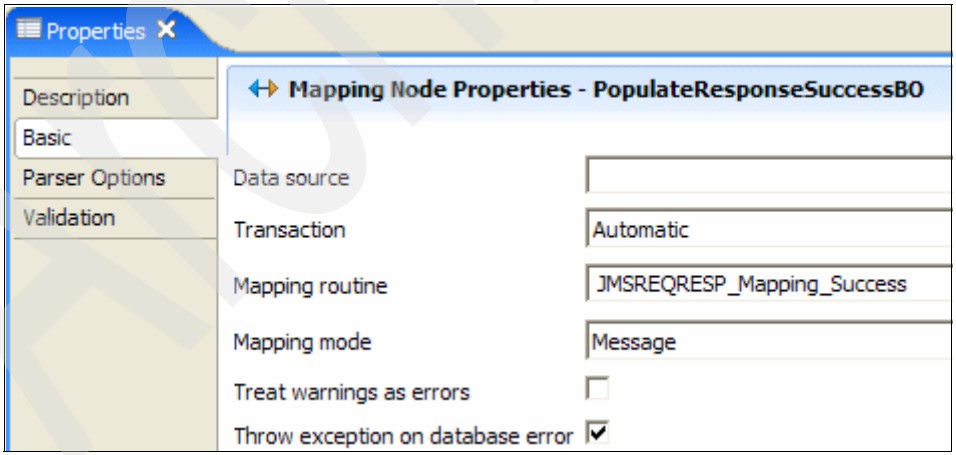


Figure 6-37 PopulateResponseSuccessBO

The Mapping routine field in the basic properties points to the message map, which is shown in Figure 6-38.

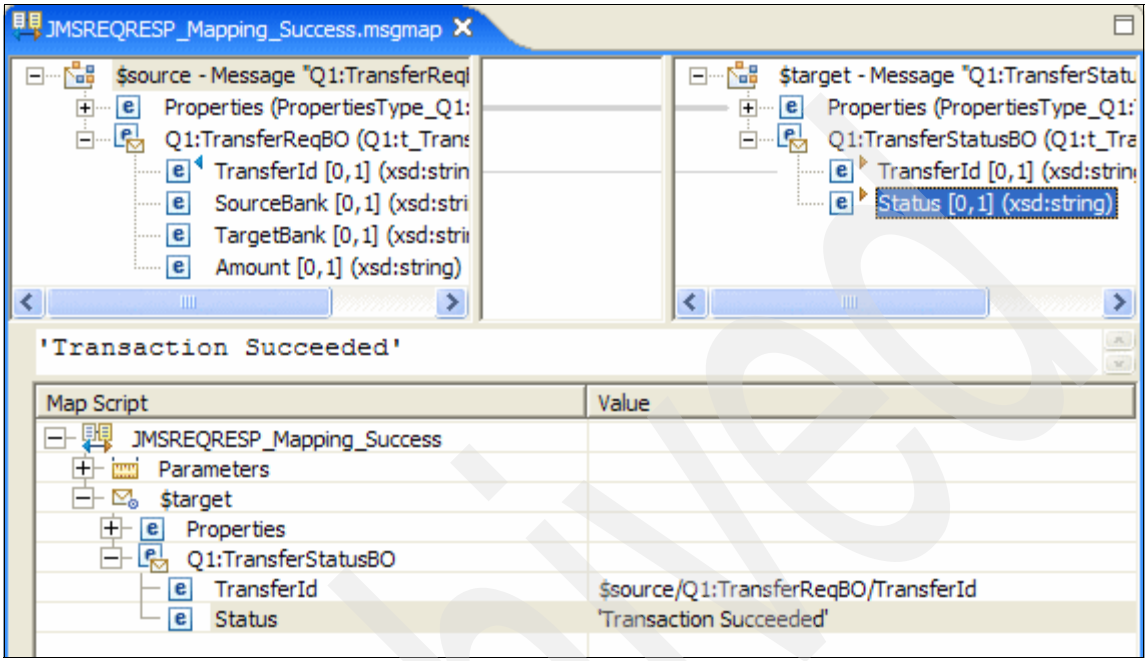


Figure 6-38 Mapping_Success message map

CopyMsgIDToCorrelID JavaCompute node

The message that is sent back to the business process must have a correlation ID that allows the process to correlate the response with the request message. This nodes copies the Message ID of the original message to the correlation ID that is specified in the JMS header properties (Figure 6-39).

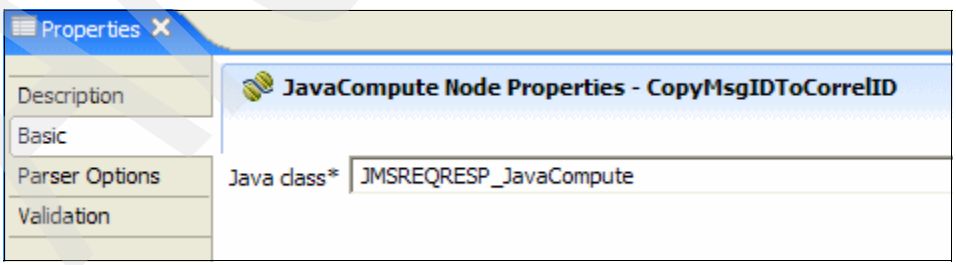


Figure 6-39 CopyMsgIDToCorrelID

Archive

Example 6-1 shows the Java code.

Example 6-1 Java code in JavaCompute node

```
// Add user code below------------------------------------------
outAssembly.getMessage().getRootElement().getFirstChild().getNextSiblin
g().getFirstChild().getFirstElementByPath("Header_Values").getFirstElem
entByPath("JMSCorrelationID").setValue(outAssembly.getMessage().getRoot
Element().getFirstChild().getNextSibling().getFirstChild().getFirstElem
entByPath("Header_Values").getFirstElementByPath("JMSMessageID").getVal
ue());
// End of user code ---------------------------------------------
```

SendStatusSuccess JMSOutput node

The SendStatusSuccess JMSOutput node sends the message indicating success to the Response queue where the business process will retrieve it. The destination queue, jms_RespQ, was defined to the JNDI name space in “Creating the JMS destinations” on page 134. It corresponds to the TEST/RESPONSEQUEUE on TEST.QUEUE.MANAGER. See Figure 6-40.

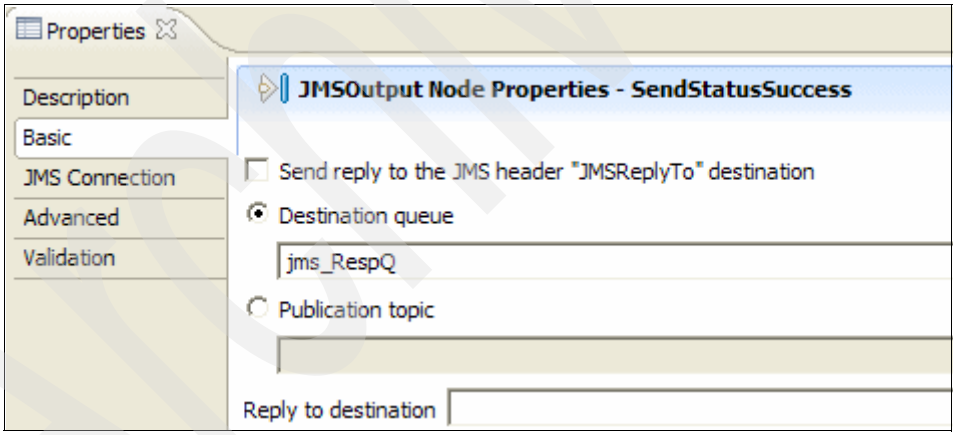


Figure 6-40 SendStatusSuccess - Basic

Figure 6-41 shows the JMS connection properties. The connection factory name is jms_ESBQCF, which corresponds to the TEST.QUEUE.MANAGER base queue manager. The connection factory was defined in “Creating the queue connection factories” on page 131.

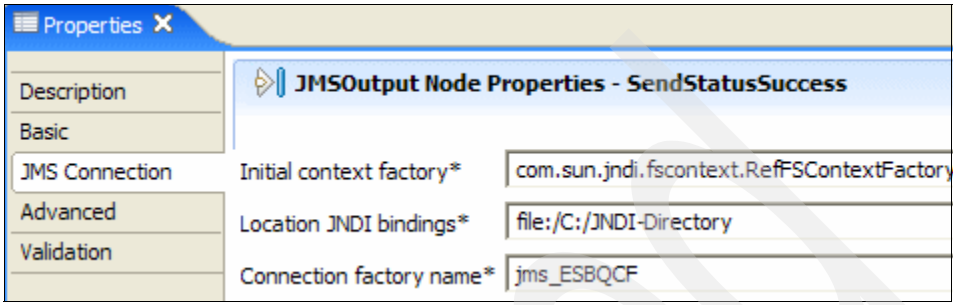


Figure 6-41 SendStatusSuccess - JMS Connection

PopulateResponseFailureBO Mapping node

The PopulateResponseFailureBO Mapping node (Figure 6-42) is connected to the Catch terminal of the ReceiveTransferRequest JMSInput node. It maps the status of the transaction to the output business object and copies the transaction ID from the input message to the output message.

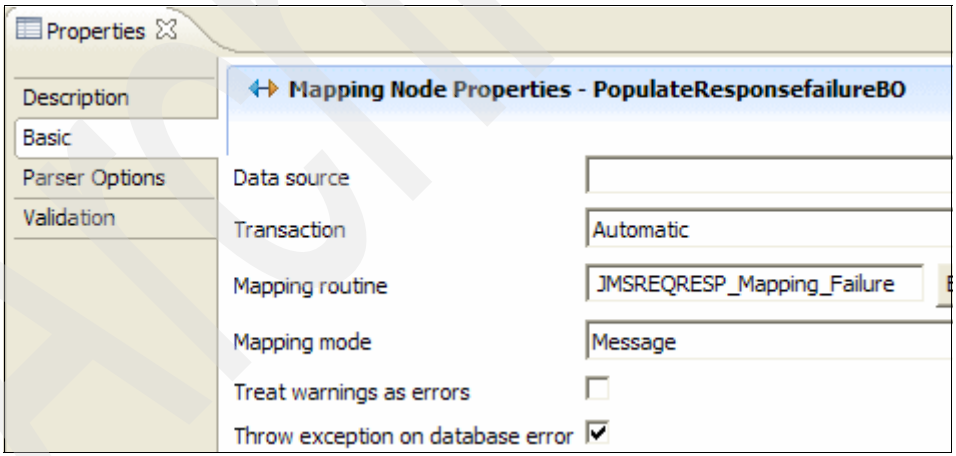


Figure 6-42 PopulateResponseFailureBO - Mapping Node

The basic properties point to the map shown in Figure 6-43.

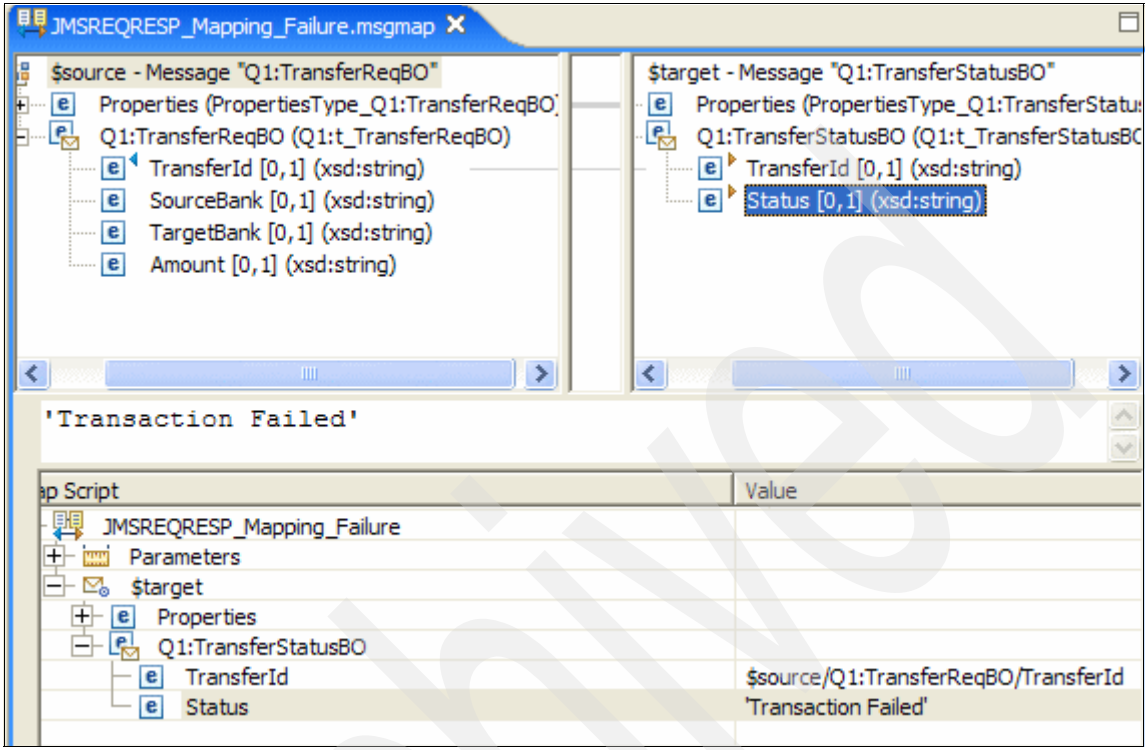


Figure 6-43 Mapping_Failure message map

SendStatusFailure JMSOutput node

The SendStatusFailure JMSOutput node is used to send a message indicating failure to the business process. The message is sent to TEST/TESTRESPONSEQ. This node has the same properties as the JMSOutput node used for the success message. See “SendStatusSuccess JMSOutput node” on page 155.

Archived

Throw node

A Throw node is used to throw an exception within a message flow. The Throw node shown in Figure 6-44 is the last of the series of nodes that is connected to the Catch terminal of the JMSInput node.

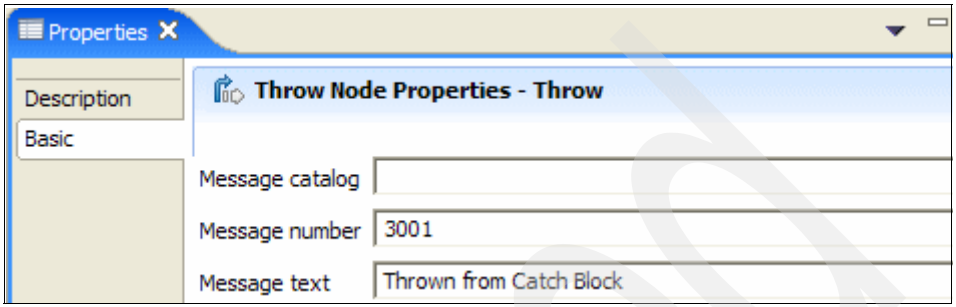


Figure 6-44 Throw node

When an exception occurs in the message flow anywhere along the nodes connected to output terminal of the JMSInput node (ReceiveTransferRequest), the exception is caught and processed by the Catch terminal of the JMSInput node.

In this scenario, a message is sent to the two JMS providers, the CreditToBank1 and DebitFromBank2 JMSOutput nodes, downstream under XA transactional control. Suppose that CreditToBank1 can send the message successfully, but DebitFromBank2 fails. An exception is thrown by DebitFromBank2 indicating that it cannot put the message. This exception is caught by the Catch terminal of the JMSInput node, and the catch path is invoked. To prevent the first, successful CreditToBank1 node from sending the message, a Throw node is included in the catch path. The Throw node sends the error back to the JMSInput node, which causes a rollback to be issued for the successful message. Hence the transactional integrity is maintained.

Archived

6.6.3 A checklist for deploying the message flow

To deploy the message flow, perform the following actions on the WebSphere Message Broker system:

1. Using the Message Brokers Toolkit, select the **Coordinated Transaction** message flow property in the bar file properties as shown in Figure 6-45.

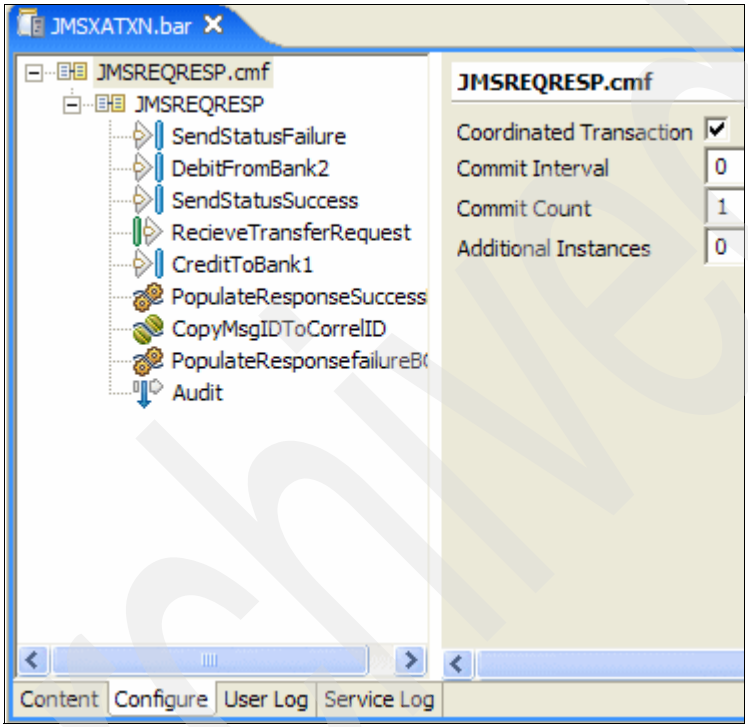


Figure 6-45 Coordinated Transaction option

2. Ensure that the xahandler.jar and com.ibm.mqetclient.jar are in the Java classpath on the broker system.
3. Ensure that any JAR files that are required by the JMS provider for transactional support are placed in the broker shared-classes directory.
4. Update the Java PATH environment variable for the broker's queue manager to point to the bin directory, where the switch file is located, for example:

   ```
   install_dir/bin
   ```

5. Make sure that the connection factories for the queue managers who are participating in the global transaction have a class name of MQXAQueueConnectionFactory (Figure 6-11 on page 134). If not, recreate

the connection factory and select the **Support XA transactions** option (Figure 6-8 on page 131).

6. The service user ID used by the broker must match the queue manager. The easiest way to do this is to change the service ID of WebSphere MQ. This is because MUSR_MQADMIN has a hidden password, which makes it challenging to change it.

   To change the user name that is associated with WebSphere MQ Services from MUSR_MQADMIN to something else, use the following command:

   ```
   AMQMSRVN -user <domain\>NEW_NAME -password <password>
   ```

   Here *NEW_NAME* is the new user name you have chosen and can be qualified by a domain name, if required.

   WebSphere MQ allocates the correct security rights and group membership to the new user account.

   If for any reason you must reset the user account back to the default MUSR_MQADMIN account, use the following command:

   ```
   AMQMJPSE -r
   ```

7. Use the XAOpen string using the default recoverXAQCF. For example, use the following string:

   ```
   com.sun.jndi.fscontext.RefFSContextFactory,file:/C:/temp/jndi/ESB
   ```

   Do not use this string:

   ```
   com.sun.jndi.fscontext.RefFSContextFactory,file:/C:/temp//jndi/ESB,,
   ,ESB.XAQCF.
   ```

   This string is configured on the broker's queue manager. See 6.4.1, "Preparing the transaction manager" on page 124.

8. In the message flow, ensure that the transaction mode for each node that is required to be part of the XA transaction is set to `Global`.

9. If you experience problems, set the environment variable XAJMS_TRACEFILE=C:\XAJMS so that trace files are put to a directory to aid in diagnostics.

Archive

6.7 Testing the flow

In this section, we show the integration testing between the integration module and the message flow in WebSphere Integration Developer. We tested the flow by using the Integration Test Client. The test client is invoked from the context menu of the business integration module, TransferFundModule, in the Business Integration view.

1. Select **TransferFundModule**, right-click, and select **Test** → **Test Module**.
2. From the Emulators section of the Configuration tab, remove TransferFundPartner.
3. In the Initial request parameters area, enter values for the TransferReqBO business object attributes (Figure 6-46).

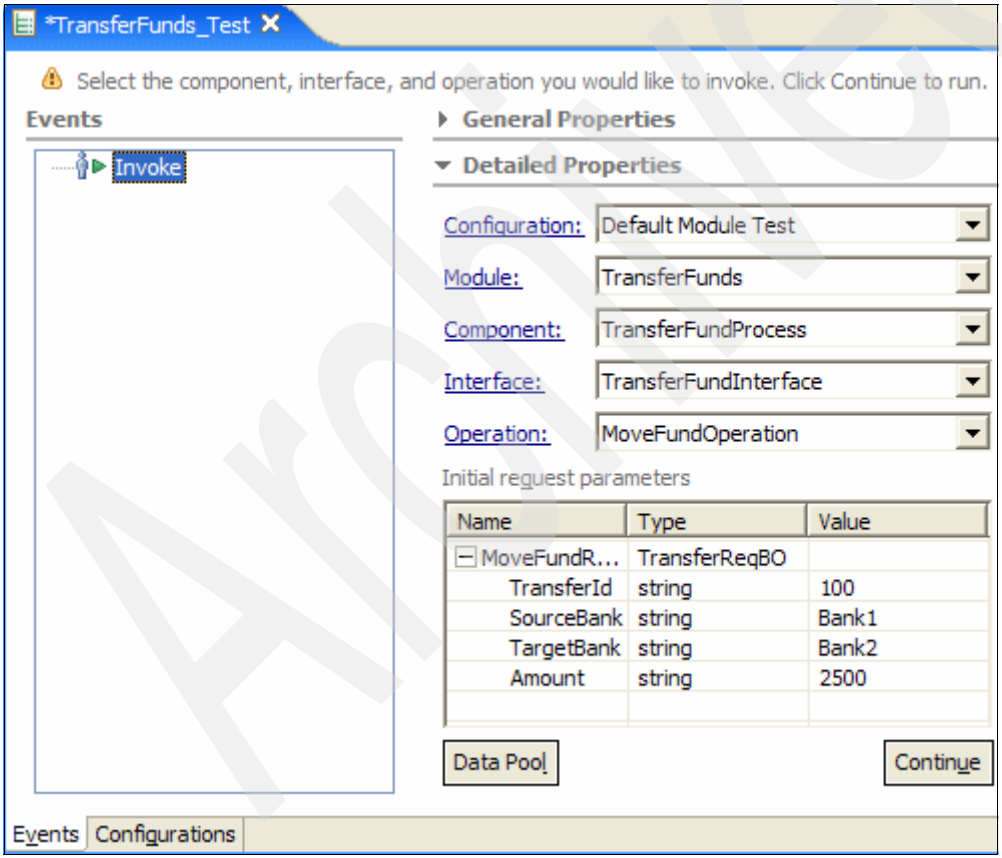


Figure 6-46 TransferFund Test module

Archived

4. Since a request-response operation is used, a response is returned indicating the status of the transaction. The JMS messages are delivered to the queue managers, and a TransferStatusBO business object is returned as shown in Figure 6-47. You can look in the SystemOut log for the WebSphere Process Server application server to view messages or errors that are produced by the business process application.

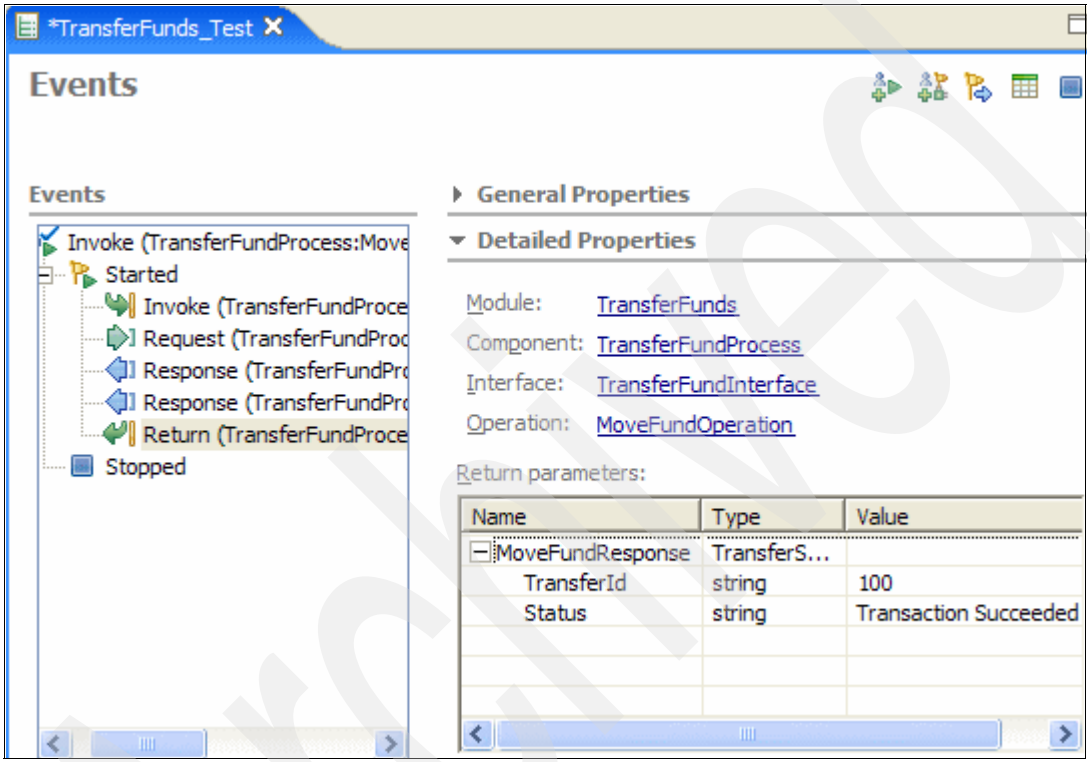


Figure 6-47 Test module - successful transaction

5. In case of success, verify that the message has successfully reached the output queues on the JMSPROV1 and JMSPROV2 queue managers.

Archived

6. You can use the same test to verify the transactionality of the message flow. To test this, make any one of the queues, either CREDITQ on JMSPROV1 or DEBITQ on JMSPROV2, put-Inhibited, *or* shut down one of the queue managers. The test should have results as indicated in Figure 6-48.

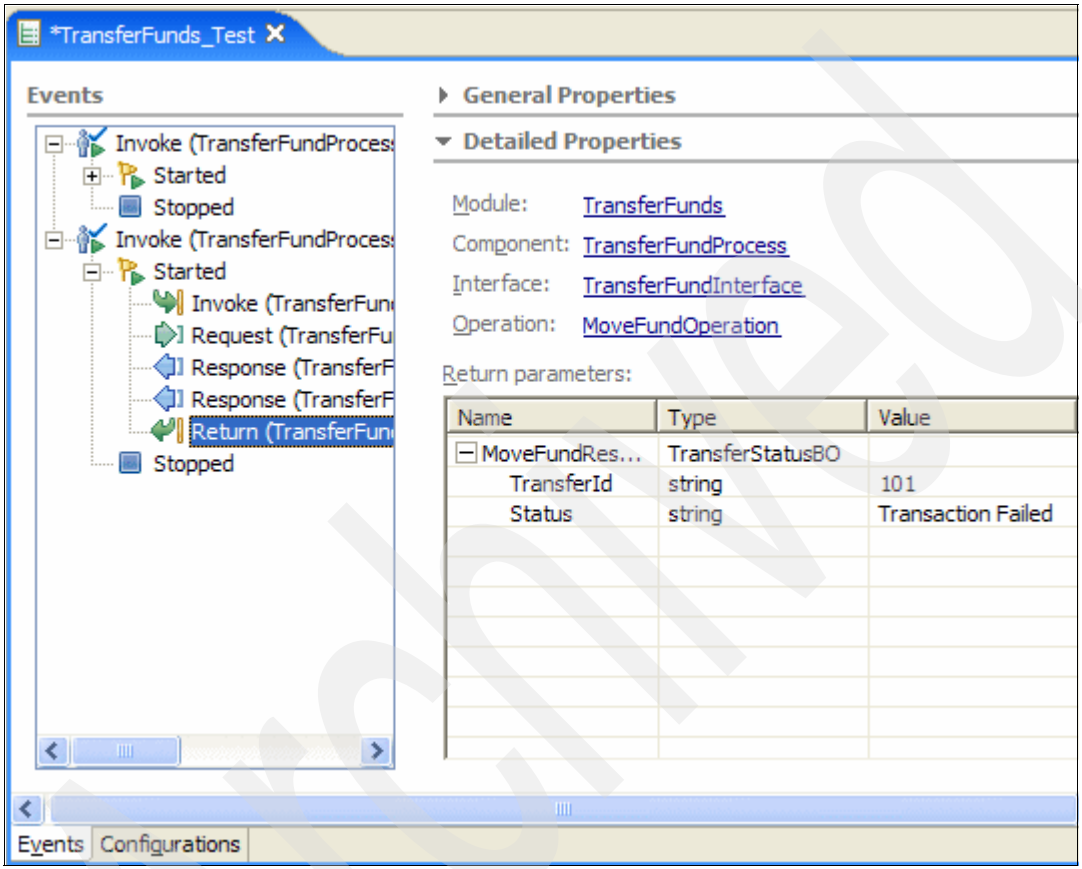


Figure 6-48 Test Module - transaction failure

Since a backout queue has been configured on TEST/REQUESTQUEUE, the failing request message is backed out to BOQUEUE.

Archived

Archived

7

Scenario: Data transformation and security

In the scenario presented in this chapter, we demonstrate the following concepts:

- Message transformation

  Messages are transformed from SOAP to MQ messages. Different wire formats are used depending on the destination.

- Protocol transformation

  Messages enter the message flow using SOAP/HTTP and are sent to the back-end services using WebSphere MQ.

- Content-based routing using a JavaCompute node
- WebSphere Message Broker techniques, including:
  - Message ordering and serialization
  - Changing the message element structure of using XPath
  - Parsing SOAP messages using the IA9O support pack and the native Extensible Stylesheet Language (XSL)/XML transformation node
- Securing communication using HTTPS and SHA/DES encryption using the Message Broker IA9W SupportPac

© Copyright IBM Corp. 2008. All rights reserved.

- Connectivity using a SOAP/HTTP Web service binding to WebSphere Message Broker

  An HTTPInput node is used in Message Broker to receive the message.
- One-way and request-reply interaction

7.1 Scenario overview

In this scenario, we define a loan request to a fictitious banking transaction. The loan account transaction must execute a number of services before the client can have access to the new account privileges. Figure 7-1 shows the flow of the requests.

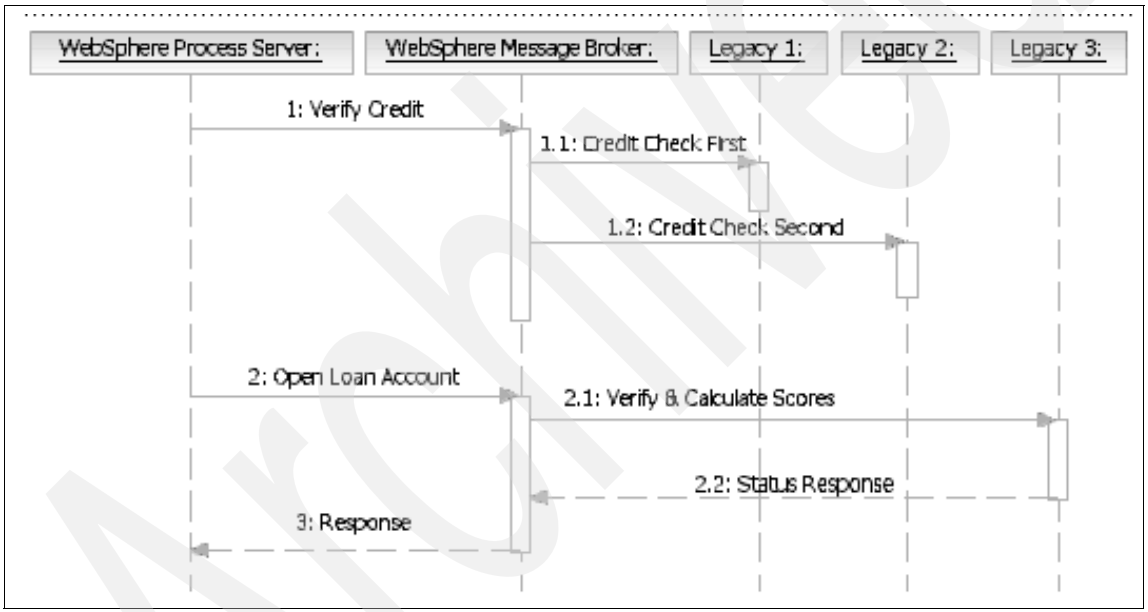


Figure 7-1 Business Process general flow

Legacy 3: Legacy 3 is a generic designation for the application that has the credit scores. The actual application (either Legacy 1 or Legacy 2) that supplies this information is determined dynamically by the message flow.

Archived

The following services are used to process the loan request:

- The *Verify Credit service* is a fire/forget service that is triggered by the Process Server and is consumed by the enterprise service bus (ESB; WebSphere Message Broker). This service is responsible for starting the validation batch process for the new loan accounts. The loan scores are stored in the application of the enterprise information system.
- The *Check Credit services* are fire/forget services that are dispatched by the ESB (WebSphere Message Broker) for two C applications of the enterprise information system, Legacy 1 and Legacy 2. These applications run as batch programs on a business partner. These services are wired using WebSphere MQ and have two different string protocols, X12 and a proprietary message format respectly.
- The *Open Loan Account service* uses a request-reply interaction to consolidate the scores that are defined by the credit verification batch services.

In the test environment for the scenario, all systems were on the same network. However, in reality, all three tiers of the solution could reside on separate networks, making secure communication between them critical.

In this scenario, we illustrate how to secure communication between WebSphere Process Server and WebSphere Message Broker and between WebSphere Message Broker and the enterprise information systems.

Archived

7.1.1 Process and message flow

Figure 7-2 provides a high-level look at this scenario.

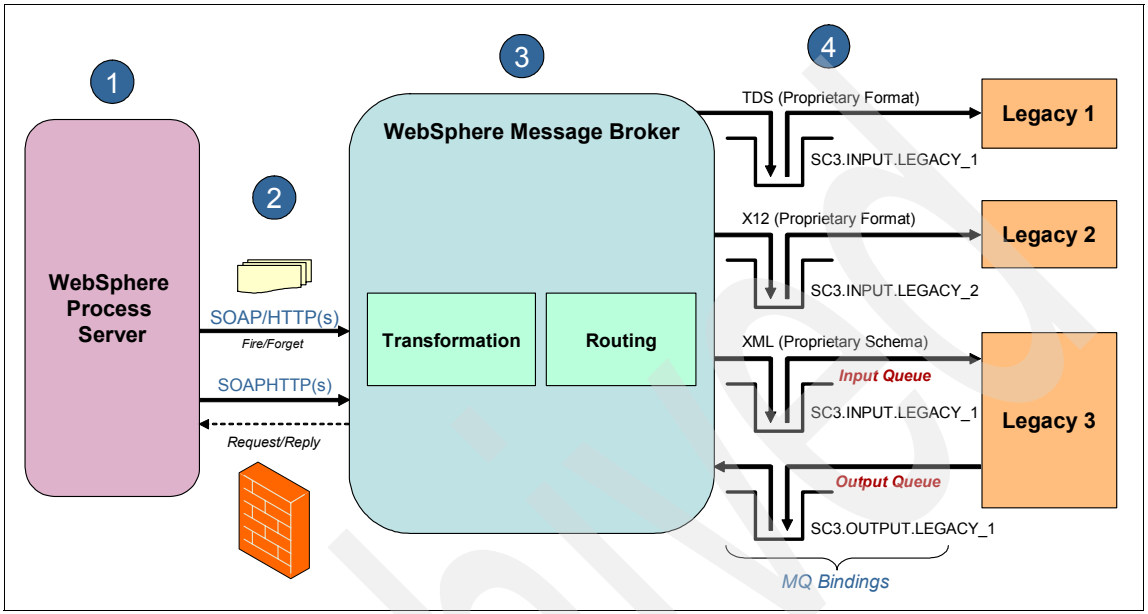


Figure 7-2 Scenario overview

We describe the highlights of this scenario, numbered in Figure 7-2, as follows:

1. WebSphere Process Server hosts a business process that sends two separate messages to WebSphere Message Broker. The first message is sent by using a one-way operation. No response is expected. The second message is sent by using a request-response operation.
2. Both messages are sent to WebSphere Message Broker by using SOAP/HTTPS transport.
   a. The first call creates a Secure Sockets Layer (SSL) handshake and sends a fire-forget SOAP message to the broker's VerifyCredit message flow.
   b. The second service call creates an SSL handshake and sends a SOAP message to the brokers OpenAccount message flow.
3. WebSphere Message Broker receives each request and starts the appropriate message flow.
   a. The VerifyCredit message flow sends the credit request to two services. It transforms the message by using the appropriate tagged delimited string (TDS) format for each service. The first message is parsed to the proprietary format expected by the Legacy 1 application. The second

message is parsed to an X12 string format for Legacy 2. Each new message is then routed to its destination.

b. The OpenAccount message flow transforms the message using XSL translation and determines the destination for the message based on the loan amount requested.

4. The message structures are parsed using the following wire protocols:

   a. The VerifyCredit message flow delivers the new messages to the Legacy 1 and Legacy 2 services by using WebSphere MQ.

   b. The OpenAccount message flow is encrypted by using a 64-bit algorithm (DES) and delivered to either the Legacy1 or Legacy 2 application by using WebSphere MQ. The application of the enterprise information system responds with an encrypted XML message that contains a simple `true|false` flag that indicates whether the client has the appropriate score necessary for a new loan account.

The enterprise information systems are emulated in this scenario using a message flow.

7.1.2 Service message formats

In this scenario, we illustrate how to use the following resources for transformation:

- A Web Services Description Language (WSDL) file that contains Web Services Interoperability Organization (WS-I) definitions, messages, and bindings
- Two schemas (XSD files) to format the internal parts of the message (application of the enterprise information system)
- An XSL file to automatically transform a XML message to another

The message from WebSphere Process Server is a SOAP message, implementing the SOAP 1.1 specification. Messages that are sent to the back-end services are plain XML messages.

Archived

The messages are defined in the bank_definitions.xsd file for the bank.wsdl port or types used by the enterprise information systems in the VerifyCredit message flow. The legacy.xsd file is used for the enterprise information systems in the OpenAccount message flow as illustrated in Figure 7-3.

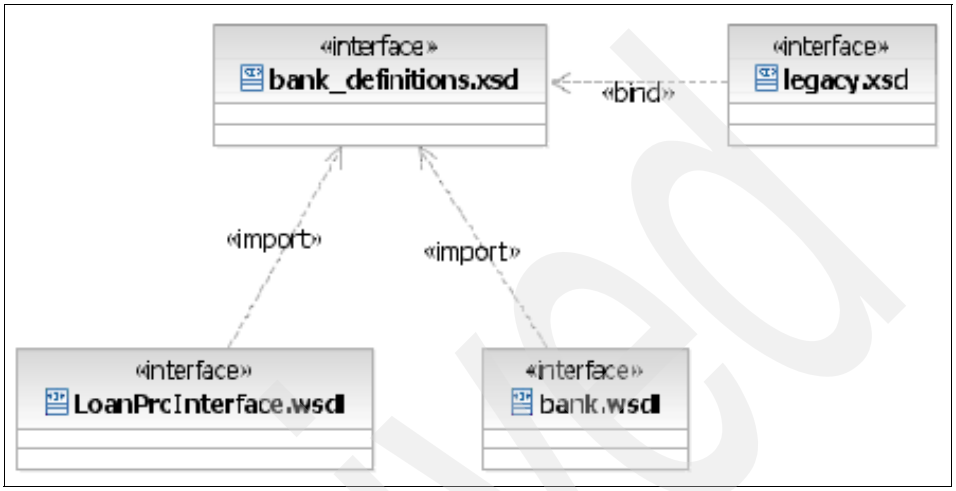


Figure 7-3 Service interface organization

Bank_definitions schema definition

The bank_definitions.xsd schema file defines the elements in use by the messages that are defined in the bank.wsdl WSDL file. WebSphere Process Service uses the bank_definitions schema implementing four elements with four different complex types:

- VerifyCreditType complex type

  The element structure that is necessary to send the client data to the validation systems running on the back office (Example 7-1)

Example 7-1 VerifyCredity complex type structure

```
<complexType name="VerifyCreditType">
   <sequence>
      <element name="client-data">
         <complexType>
            <sequence maxOccurs="1">
               <element name="FrName" type="string" minOccurs="1" />
               <element name="MidName" type="string" minOccurs="1" />
               <element name="Surname" type="string" minOccurs="1" />
               <element name="DtBirth" type="date" minOccurs="1" />
               <element name="LastName" type="string" minOccurs="1" />
               <element name="SSN" type="string" minOccurs="1" />
```

Archived

```
            <element name="Address" type="string minOccurs="1" />
         </sequence>
      </complexType>
   </element>
</sequence>
</complexType>
```

- OpenAccountType complex type

  The element structure that is used to the select the desired loan amount, duration, and tax rate (Example 7-2)

Example 7-2 OpenAccount complex type structure

```
<complexType name="OpenAccountType">
   <sequence>
      <element name="SSN">
         <simpleType>
            <restriction base="string">
               <pattern value="[A-Z][0-9]"></pattern>
            </restriction>
         </simpleType>
      </element>
      <element name="AskedAmount" type="float" />
      <element name="Duration" type="integer" />
      <element name="MonthlyTax" type="float" />
   </sequence>
</complexType>
```

- submitFaultType complex type

  Defines the fault elements that are used when a business exception occurs, during the processing of the message flows (Example 7-3)

Example 7-3 submitFault complex type structure

```
<complexType name="submitFaultType">
   <sequence>
      <element name="SSN" type="string" />
      <element name="RC" type="integer" />
      <element name="Cause" type="string" />
      <element name="Description" type="string" />
   </sequence>
</complexType>
```

Archived

- ackOAType complex type

  Defines only one Boolean simple element that is used to return a statement to the requestor (WebSphere Process Server) that specifies whether the open account request was approved (Example 7-4)

Example 7-4 ackOA complex type

```
<complexType name="ackOAType">
   <sequence>
      <element name="ackOA" type="boolean" />
   </sequence>
</complexType>
```

These elements represent the entire service virtualization between WebSphere Process Service and WebSphere Message Broker. As part of the service decomposition, the message flow uses the legacy.xsd schema to transform the incoming messages to their expected legacy formats.

Legacy.xsd schema

The legacy.xsd schema has defined only two complex types: *legacy_bank_req* for the input request and *legacy_bank_res* for the output response (Example 7-5).

Example 7-5 Legacy request and response complex types

```
<complexType name="legacy_bankReqType">
   <sequence>
      <element name="batch_data">
      <complexType>
        <attribute name="ssn" type="string" use="required" />
        <attribute name="amount" type="float" use="required" />
      </complexType>
      </element>
   </sequence>
</complexType>
<complexType name="legacy_bankResType">
   <sequence>
      <element name="approved" type="boolean" />
   </sequence>
</complexType>
```

Archived

Bank Service interface definition

The open loan account process has two service interfaces, one for the business process and one for the WebSphere Process Server or WebSphere Message Broker interoperability channel.

The external process interface is called *LoanPrcInterface*, which has one request-response operation called *loanOperation* (Example 7-6). Messages that are exchanged between clients and the business process use this interface.

Example 7-6 LoanPrcInterface port

```
<wsdl:portType name="LoanPrcInterface">
   <wsdl:operation name="loanOperation">
      <wsdl:input message="tns:loanOperationRequestMsg"
                  name="loanOperationRequest"/>
      <wsdl:output message="tns:loanOperationResponseMsg"
                  name="loanOperationResponse"/>
   </wsdl:operation>
</wsdl:portType>
```

Messages: The messages of the external process interface contain the elements that are defined inside the bank_definitions.xsd schema. The Input receives the VerifyCredit, and the output receives the ackOA element parts.

The interface for messages exchanged between the business process and the message flow in WebSphere Message Broker is called *BankPortType*. In the sample application in Example 7-7, this interface is stored in bank.wsdl. The interface has two operations. The first operation is called *BankVerifyCredit* and is used to deliver the client data to the broker's VerifyCredit message flow. The second operation is called *BankOpenAccount* and is used to send the loan account details and get a response (either true/false or success/failure flag).

Operation similarities: These two operations are in the same port and are both bound to an HTTPS call.

Example 7-7 Internal bank operations using WebSphere Message Broker

```
<wsdl:portType name="BankPortType">
   <wsdl:operation name="BankVerifyCredit">
      <wsdl:documentation>
         Verifies a client credit.
      </wsdl:documentation>
      <wsdl:input message="tns:VerifyCreditMsg"/>
   </wsdl:operation>
```

```
   <wsdl:operation name="BankOpenAccount">
      <wsdl:documentation>
         Open a loan account.
      </wsdl:documentation>
      <wsdl:input message="tns:OpenAccountMsg"/>
      <wsdl:output message="tns:ackOAMsg"/>
      <wsdl:fault message="tns:BankOpenAccount_OpenFaultMsg"
                  name="OpenFault"/>
   </wsdl:operation>
</wsdl:portType>
```

Fault message for BankOpenAccount: The fault message defined on the "BankOpenAccount" operation is not implemented in the message flow and is stated here for the purpose of explanation only.

7.2 Environment

This scenario was developed in an environment with the following software:

- WebSphere Message Broker run time 6.0.0.5
- WebSphere Message Brokers Toolkit 6.0.2.9
- WebSphere Integration Developer 6.0.2.2
- WebSphere MQ 6.0.2.2
- IBM DB2 Universal Database™ 8.2
- IBM Java Runtime 1.5
- IBM IA9O (SOAP nodes) and IA9W (Security nodes) WebSphere Message Broker support packs
- OpenSSL 0.9.8e
- Microsoft® Windows XP SP2

Note: You can download WebSphere Message Broker support packs from the WebSphere MQ - SupportPacs by Product page at:

http://www-1.ibm.com/support/docview.wss?uid=swg27007197

Archived

7.3 Preparing for security

All service operations between WebSphere Process Server and WebSphere Message Broker are bound using HTTP under an SSL handshake protocol and session.

7.3.1 Creating the keystore and truststore databases

To create a secure connection between the WebSphere Message Broker listener and the WebSphere Process Server binding, a private key database (keystore) and a public key database (truststore) must be created.

Self-signed certificates: In this scenario, we only use self-signed certificates that are extracted from the keystore and are imported to the truststore.

Several ways are available to construct a keystore and a truststore, that is ikeyman. We use the most common ones of the **keytool** and **openssl** commands. The **openssl** command is part of the OpenSSL distribution, and **keytool** is part of the Java runtime environment (JRE™) installation. The OpenSSL run time comes as a source-code package, and as in UNIX-based platforms, Windows machines must also compile and create the binaries.

For more information about OpenSSL, refer to the Web site at:

http://www.openssl.org

Recommendation: Several pre-compiled keystore and truststore versions on the Internet can be used to run this case. However, for a real implementation, we recommend that you create the files on the desired platform, either Windows 2003 Server or AIX® 5L™.

We use the following steps to create a public and private key database (using the **keytool** and **openssl** commands) and import them into the HTTPSListener of the broker and the bindings of WebSphere Process Server:

1. Define a private key and certificate for your own certificate authority (CA):

```
# openssl req -new -newkey rsa:1024 -nodes -out ca.csr -keyout
ca.key -config openssl.cnf
```

2. Define a self-signed certificate (ca.cer) that is valid for one year:

```
# openssl x509 -trustout -signkey ca.key -days 365 -req -in ca.csr
-out ca.cer
```

3. Import the self-signed certificate into the Java Development Kit (JDK™) certificate authority (CA):

```
# keytool -import -keystore $JAVA_HOME/lib/security/cacerts -file
ca.cer -alias sam730rr_ca
```

> **Changing the password:** The default password for the cacerts file is "changeit." If you want to change the password, use the following command:
>
> ```
> $ keytool -storepasswd -new <<new password>> -storepass changeit
> -keystore $JAVA_HOME/lib/security/cacerts
> ```
>
> If this command does not work on Windows, instead of defining the *JAVA_HOME* environment variable, type the full JDK path name:
>
> ```
> C:/jdk1.5.0_12/jre/lib/security/cacerts
> ```

4. Create a keystore called *sam730rr.keystore*:

```
# keytool -genkey -alias sam730rr -keyalg RSA -keysize 1024
-keystore sam730rr.keystore -storetype JKS
```

5. Import your self-signed certificate to your keystore:

```
# keytool -import -alias sam730rr_ca -keystore sam730rr.keystore
-trustcacerts -file ca.cer
```

6. Create a truststore called *sam730rr.truststore* using the self-signed certificate:

```
# keytool -import -alias sam730rr_ca -keystore sam730rr.truststore
-file ca.cer
```

7.3.2 Configuring WebSphere Message Broker security

After creating a self-signed certificate, a keystore should be defined for the HTTPListener object on the broker. We used the following steps to link the keystore to a broker called *SAM730BRK* inside WebSphere Message Broker:

1. Go to the directory where the keystore or truststore was created, and type **mqsiprofile** to configure the necessary environment variables that are used by the broker. *$MQSI_HOME* must configured on the *$PATH* variable.
2. Turn on SSL support:

```
# mqsichangeproperties SAM730BRK -b httplistener -o HTTPListener -n
enableSSLConnector -v true
```

3. Define the keystore to be used with this new SSL connection. In this case, we use the keystore that was defined previously:

```
# mqsichangeproperties SAM730BRK -b httplistener -o HTTPSConnector
-n keystoreFile -v sam730rr.keystore
```

4. Specify this keystore password:

```
# mqsichangeproperties SAM730BRK -b httplistener -o HTTPSConnector
-n keystorePass -v sam730rr
```

5. Specify a port that should be bound to the SSL connector. In this case, we are listening to HTTPS requests on port 10443:

```
# mqsichangeproperties SAM730BRK -b httplistener -o HTTPSConnector
-n port -v 10443
```

6. Restart the broker:

```
# mqsistop SAM730BRK
# mqsistart SAM730BRK
```

The broker is now listening for HTTPS requests on port 10443 using the CA self-signed certificate that we created previously.

7.3.3 Configuring WebSphere Process Server security

The Process Server security configuration must be executed on the WebSphere Application Server itself and on the Java snippet just after the service input on the business process. To start configuring the SSL on the application server:

1. Go to the administrative console:

```
http://localhost:9060/ibm/console
```

> **Port 9060:** By default, the WebSphere Application Server administrative console is bound to the port 9060.

2. After login, select **Security** → **SSL**.
3. Click **New JSSE repertoire**.

Archived

4. Complete the following fields to define the JSSE repertoire (Figure 7-4):

   a. Alias: `sam730JSSE`
   b. Key file name: `${SAM730_SECURITY}/sam730rr.keystore`
   c. Key file password: Your password
   d. Key file format: **JKS**
   e. Trust file name: `${SAM730_SECURITY}/sam730rr.truststore`
   f. Trust file password: Your password
   g. Trust file format: **JKS**

   This example uses a WebSphere environment variable called SAM730_SECURITY to identify the location of the stores.

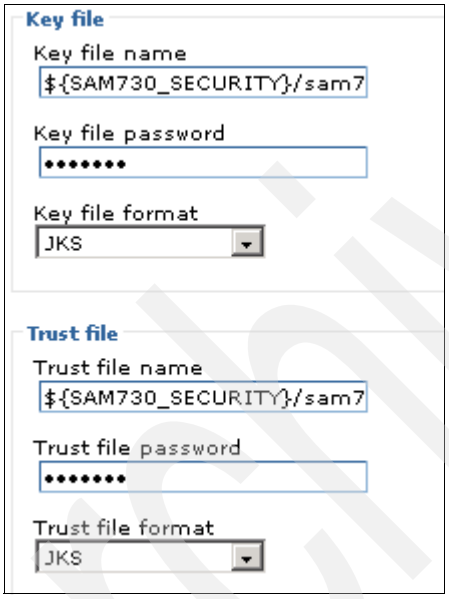


Figure 7-4 Key and trust store definition

5. After you create the JSSE repertoire, select **Security** → **Global Security**.

Archived

6. Under General Properties, for Active authentication mechanism, select **Lightweight Third Party Authentication (LTPA)**. See Figure 7-5.

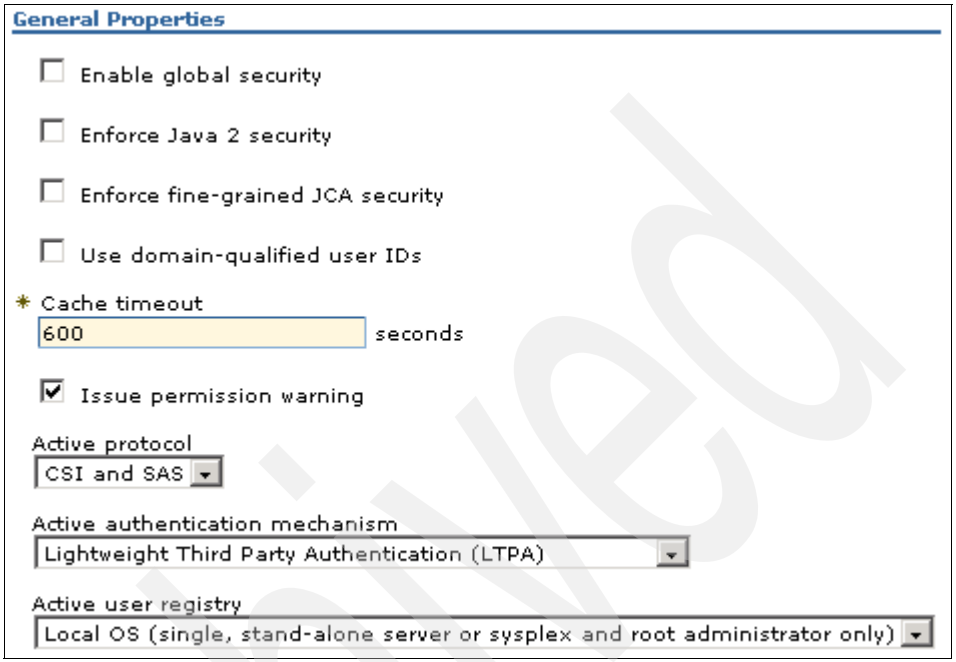


Figure 7-5 Configuring global security

7. Expand the **Authentication protocol** section on the right and click **CSIv2 inbound authentication**.

Archived

8. Under General Properties, select **Supported** for both Basic authentication and Client certificate authentication (Figure 7-6).

Figure 7-6 Inbound/Outbound authentication

9. Repeat step 7 on page 179 and step 8 for **CSIv2 outbound authentication**, ensuring that **Supported** is selected for both Basic authentication and Client certificate authentication.
10. Back in the Global Security Authentication protocol section, click **CSIv2 inbound transport**.
11. Under General Properties, select **SSL-supported** (Figure 7-7) and for SSL settings, select the **<node>/sam730JSSE** repertoire.

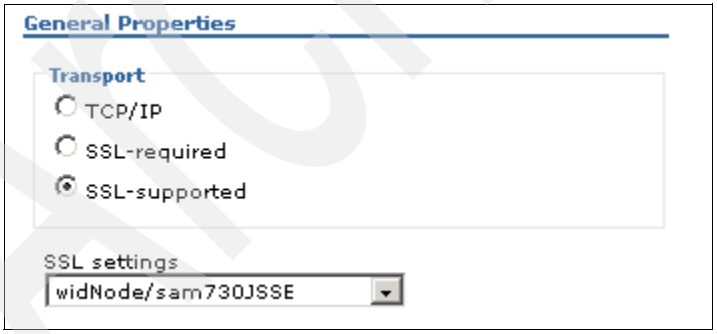


Figure 7-7 Transport configuration

12. Repeat steps 10 and 11 for the **CSIv2 outbound transport**.
13. Select **Server** → **Application servers**. Click the server in the list of servers (**server1** in this example) to open the configuration page.

Archived

14. In the Container Settings section, expand **Web Container Settings** and select **Web container transport chains**.

15. In the configuration list that opens (Figure 7-8), click **WCInboundAdminSecure**.

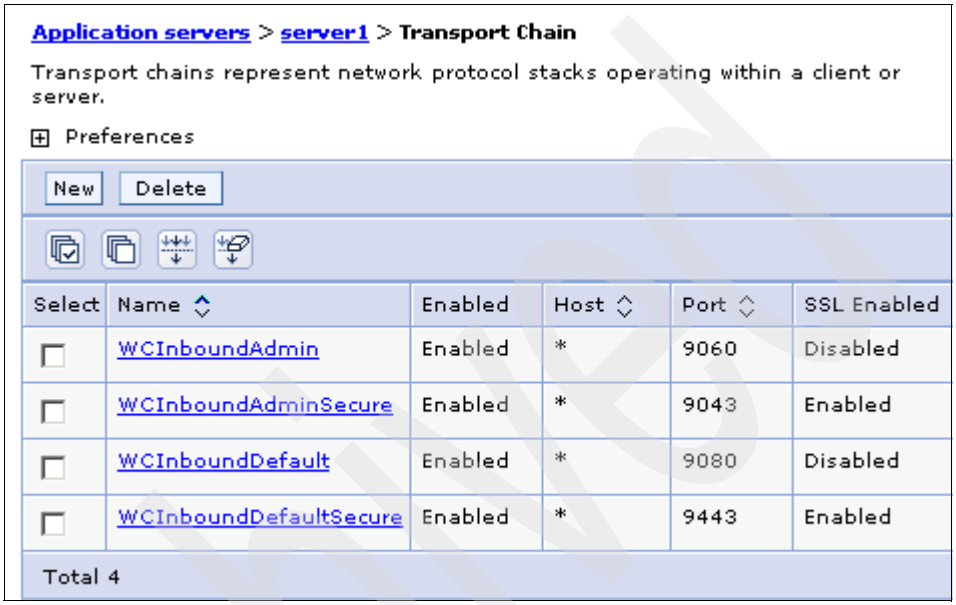


Figure 7-8 Transport Chain list configuration

16. In the WCInboundAdminSecure view, click **SSL Inbound Channel (SSL_1)**.

17. From the SSL inbound configuration, change SSL repertoire to **<*node*>/sam730JSSE** (Figure 7-9).

Figure 7-9 Adding the JSSE repertoire to the Web container inbound configuration

18. Save all changes to the master repository.

After we finish these steps, we restart the WebSphere Process Server application server. The JSSE repertoire is now configured to use the keystore and truststore on secure server/client connections.

The second part of the security configuration is the client properties that must be defined to the run time. The WS-I client configuration is implemented using J2EE and can be changed by using the Java virtual machine (JVM™) environment entries by using the -D flag or directly editing the System.setProperty method. We must set the properties shown in Table 7-1 before the HTTPS request.

Table 7-1 SSL property elements

Property name	Property value
javax.net.ssl.trustStore	The complete file path of the truststore
javax.net.ssl.trustStorePassword	The password of the truststore
javax.net.ssl.keyStore	The complete file path of the keystore
javax.net.ssl.keyStorePassword	The password of the keystore

We must set these flags before the business flow can use the interfaces on the import components. You will see this done in the business process with a Java Snippet activity later in “ConfigSSLEnvironment Java Snippet activity” on page 191.

7.4 WebSphere MQ configuration

To run this scenario, a queue manager for the configuration manager and runtime broker must be defined inside the WebSphere MQ. The following queues must also be created in order to run this scenario (see Example 7-8 on page 183):

- `DEFINE QLOCAL('SC3Q.INPUT')`

  Queue used by the VerifyCredit flow.
- `DEFINE QLOCAL('SC3Q.INPUT.LEGACY_1')`

  Queue used for sending messages to Legacy 1.
- `DEFINE QLOCAL('SC3Q.INPUT.LEGACY_2')`

  Queue used for sending messages to Legacy 2.
- `DEFINE QLOCAL('SC3Q.OUTPUT.ERR')`

  Error queue used to store exceptions that occurred during the flow execution.

- `DEFINE QLOCAL('SC3Q.OUTPUT.LEGACY_1')`

  Queue used for output from Legacy 1.

- `DEFINE QLOCAL('SC3Q.OUTPUT.LEGACY_2')`

  Queue used for output from Legacy 2.

Example 7-8 Creating a local queue using the runmqsc command

```
# runmqsc
...
: DEFINE QLOCAL('SC3Q.OUTPUT.LEGACY_2') +
        MAXDEPTH(1000) MAXMSGL(1048576) +
        NOTRIGGER REPLACE
...
```

7.5 Business integration module

In this section, we demonstrate the implementation of a business integration module deployed on WebSphere Process Server. The business integration module receives a message from the client over the export component. The process sends two messages to WebSphere Message Broker using a secure channel (HTTPS). After the reply arrives from the broker, the business process replies to its client.

7.5.1 Components of the business integration module

The Open Loan Account business process is implemented in a business integration module called *LoansModule*. Figure 7-10 shows the assembly diagram for this module.

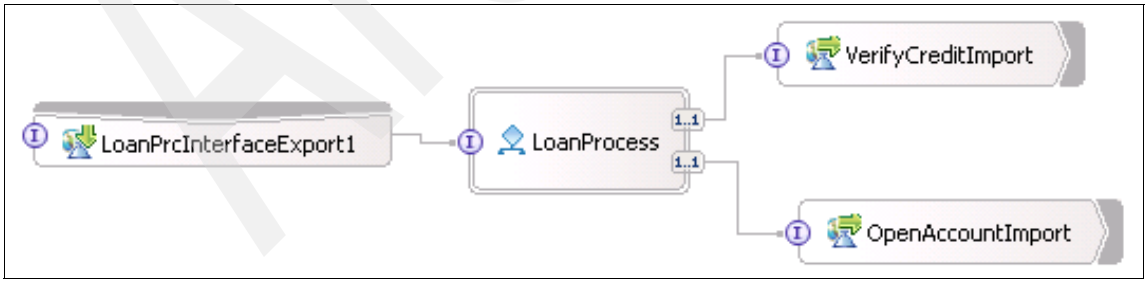


Figure 7-10 LoansModule assembly

Archived

The assembly includes the following components:

- *LoanPrcInterfaceExport1* is the export component that the client uses to access the business process.
- *LoanProcess* contains the business process implementation.
- *VerifyCreditImport* is used to send a message to WebSphere Message Broker where it is processed by the VerifyCredit message flow.
- *OpenAccountImport* is used to send a message to WebSphere Message Broker where it is processed by the OpenAccount message flow.

The LoanPrcInterface interface is used to access the process. The BankPortType interface is used to access the services.

Business objects

The business integration module uses four business objects, which we describe in the following sections.

VerifyCreditType

The VerifyCreditType business object (Figure 7-11) contains the data that is submitted by the client that is requesting the loan.

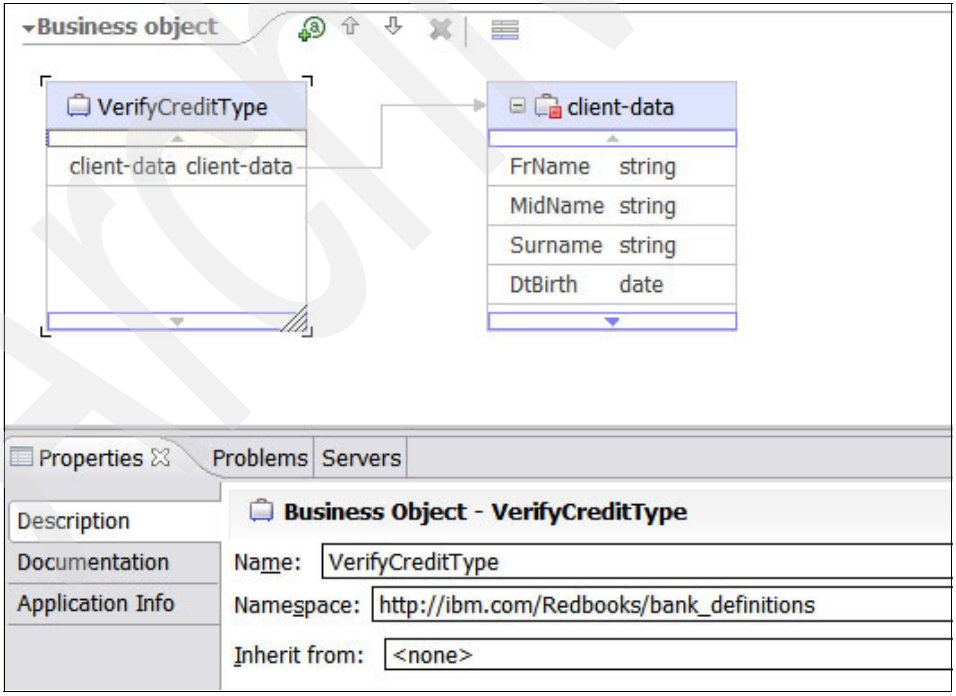


Figure 7-11 VerifyCreditType business object

This business object has one attribute called `client-data` of type `client data` with four attributes:

- FrName: The client's first name
- MidName: Middle name
- Surname: Last name
- DrBirth: Birth date

All elements are defined within the namespace `http://ibm.com/Redbooks/bank_definitions`.

Namespace: The namespace of each business object should match with the targetNamespace in the WSDL defined in "Bank Service interface definition" on page 173.

ackOAType

The ackOAType business object (Figure 7-12) is used to respond to the client, indicating the status of the loan request. A value of *true* means that the loan is approved. A value of *false* means that the loan was not approved.

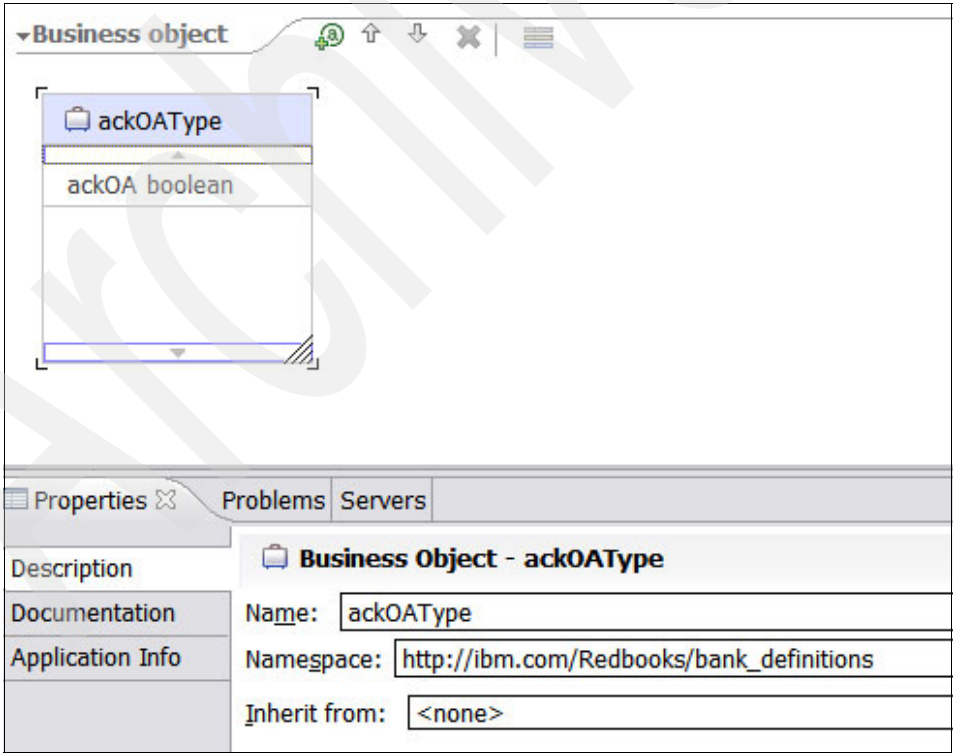


Figure 7-12 The ackOA business object

Archived

This business object has the attribute ackOA, which is a Boolean element. The business object is defined in the namespace `http://ibm.com/Redbooks/bank_definitions`.

OpenAccountType

The OpenAccountType business object (Figure 7-13) is used to send relevant data to the enterprise information systems to process the loan.

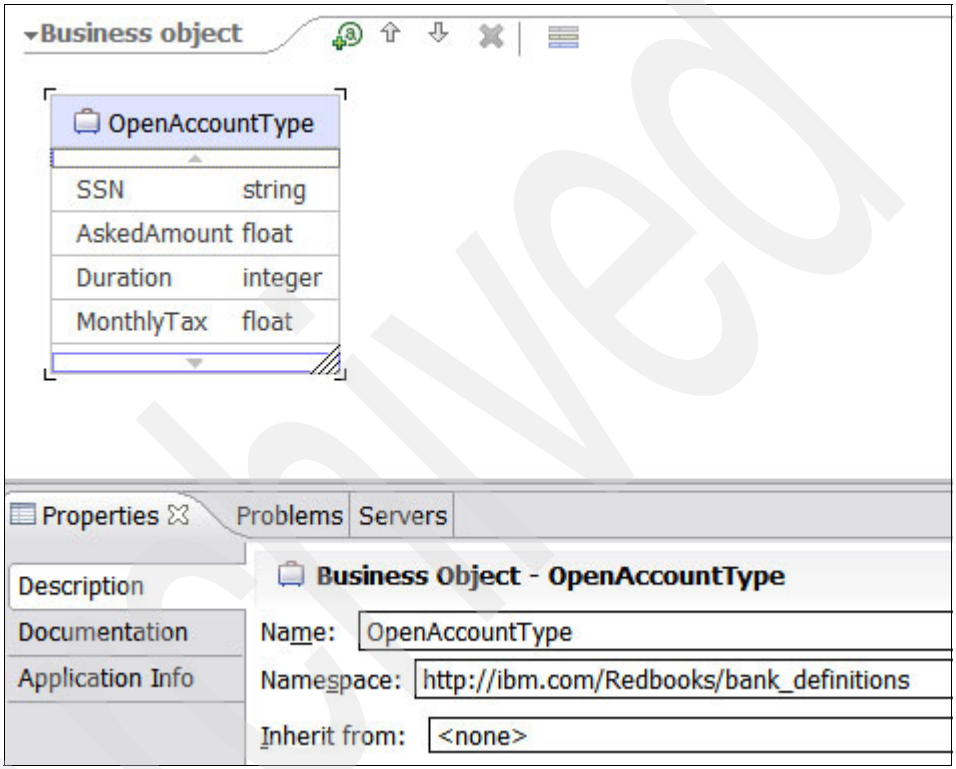


Figure 7-13 OpenAccount business object

The OpenAccount business object has the following attributes:

- SSN: The client's Social Security Number (SSN)
- AskedAmount: The desired loan amount (AskedAmount)
- Duration: The loan payment time
- MonthlyTax: The tax rate applied monthly

All elements are defined in the namespace `http://ibm.com/Redbooks/bank_definitions`.

Archived

submitFaultType

The submitFaultType business object (Figure 7-14) is used to transmit error messages between the message flow and the business process.

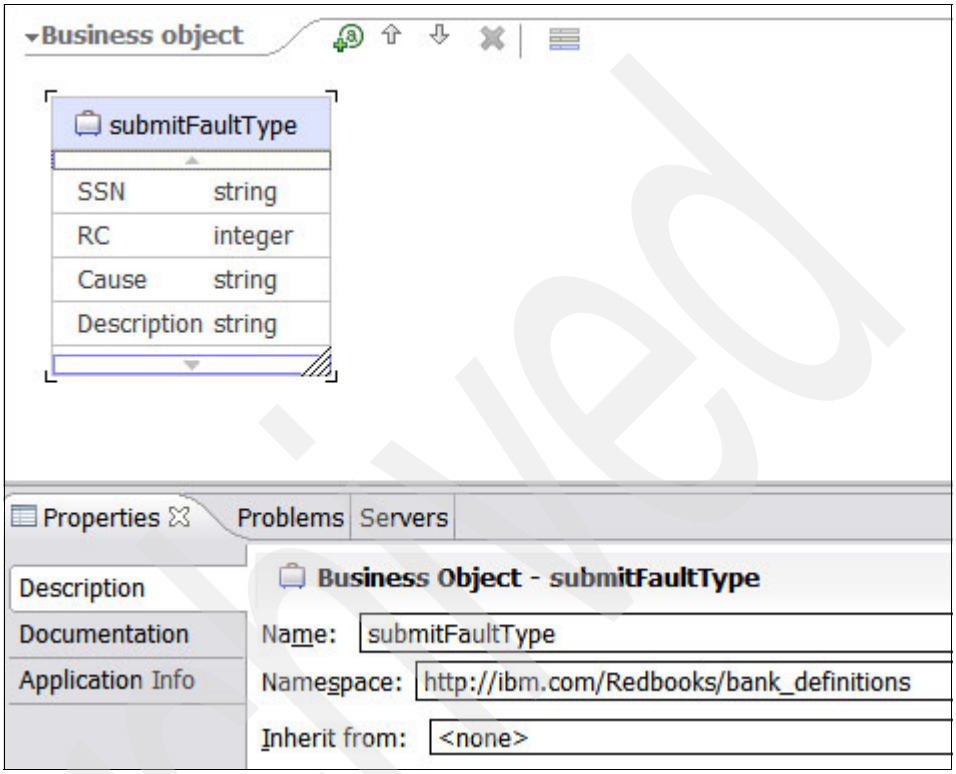


Figure 7-14 submitFaultType business object

The submitFaultType business object has the following attributes:

- SSN: The Social Security Number
- RC: The return code
- Cause: The exception cause
- Description: An error description

All elements are defined within the namespace `http://ibm.com/Redbooks/bank_definitions`.

Archived

7.5.2 Interfaces

The business integration module uses two interfaces:

- LoanPrcInterface to invoke the business process
- BankPortType to consume services from the message flows

LoanPrcInterface

The LoanPrcInterface interface, shown in Figure 7-15, is used to provide external access to the business process. It defines a request-reply interaction with the service requestor by using the VerifyCreditType business object as the message input, and the ackOAType business object as the message output.

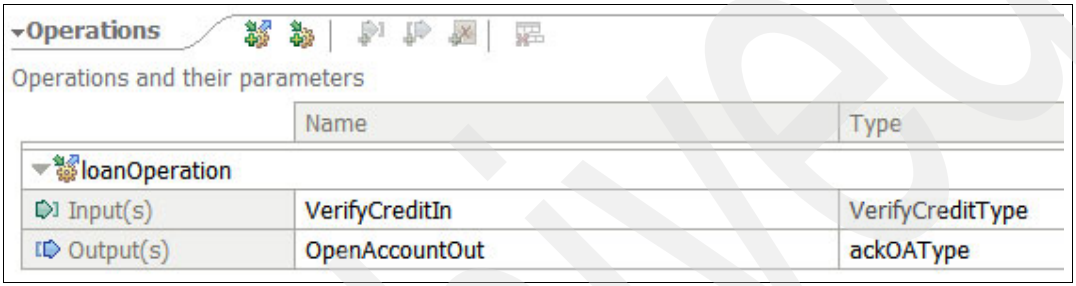


Figure 7-15 LoanPrcInterface interface configuration

BankPortType

The BankPortType interface has two operations as shown in Figure 7-16. One accesses the VerifyCredit service, and the other accesses the OpenAccount service. Both services are message flows in WebSphere Message Broker.

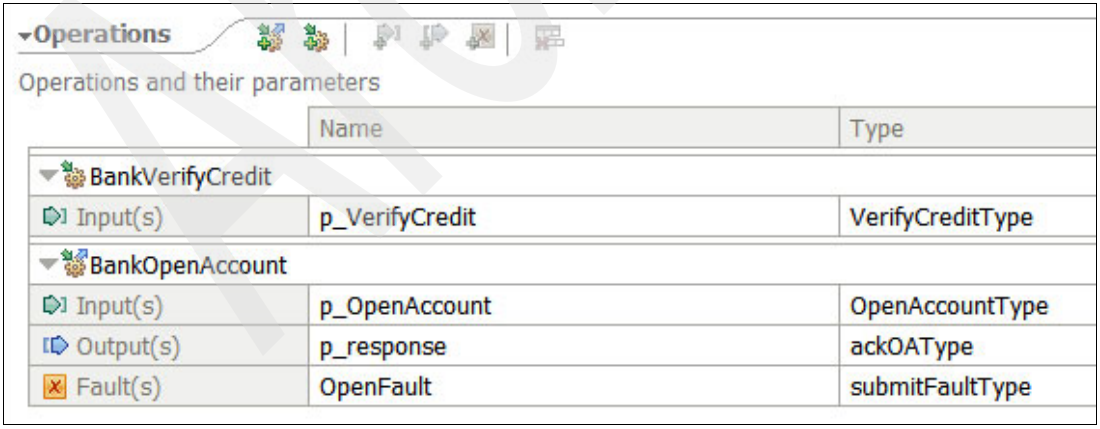


Figure 7-16 BankPortType interface configuration

The BankVerifyCredit operation is a one-way operation. The input to the operation is the VerifyCreditType business object (Figure 7-11 on page 184).

The BankOpenAccount operation is a request-response operation. It has three input/output types. The OpenAccountType business object is the input message, the ackOAType business object is the output message, and the submitFaultType business object is the fault message.

WSDL files: Interfaces are actual WSDL files. The WSDL file name usually reflects the interface name. For example, the file name for LoanPrcInterface is LoanPrcInterface.wsdl. However, in our example, the file name for the BankPortType interface is actually *bank.wsdl*. We highlight this fact, because later you will see this file name used.

7.5.3 Business process

Figure 7-17 shows the LoanProcess business process.

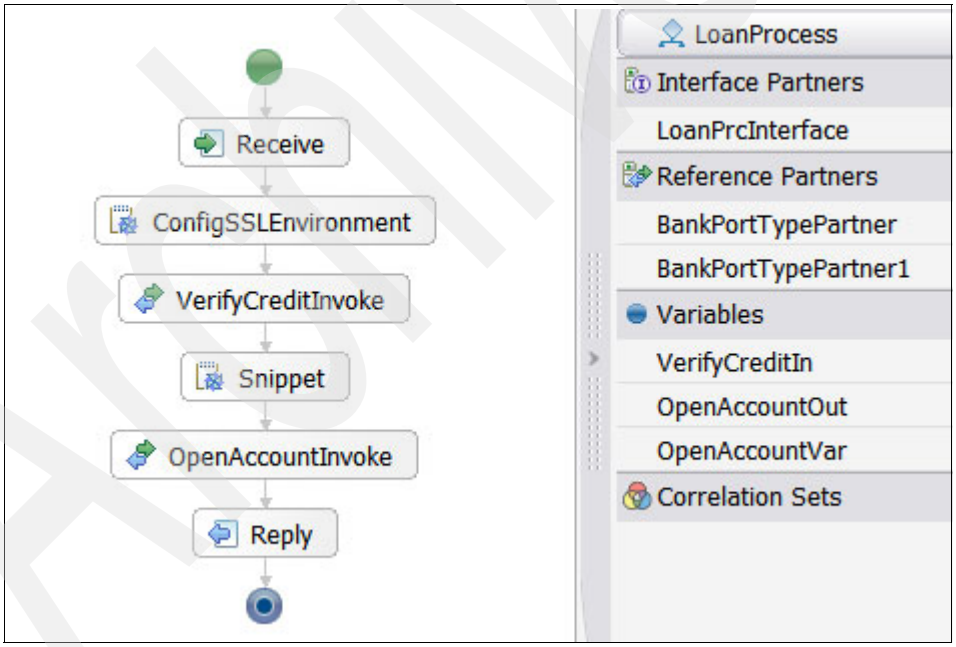


Figure 7-17 Loan business process

The properties for this process are the default properties.

Archive

Interface and partner references

The business process uses one interface partner and one reference partner. You can see these listed to the right of the business process in Figure 7-17 on page 189.

An *interface partner* is the process interface and exposes operations that can be called by external partners. The LoanPrcInterface interface is defined as an interface partner for the interaction between the business process and its clients.

Reference partners specify the interface that is used in the invocation of another service. In this scenario, two partner references are defined, one for each message flow to be executed in WebSphere Message Broker:

- BankPortTypePartner (Figure 7-18) specifies that the BankPortType interface is to be used to call the service. In the assembly diagram, this reference partner is wired to the VerifyCredit import component.

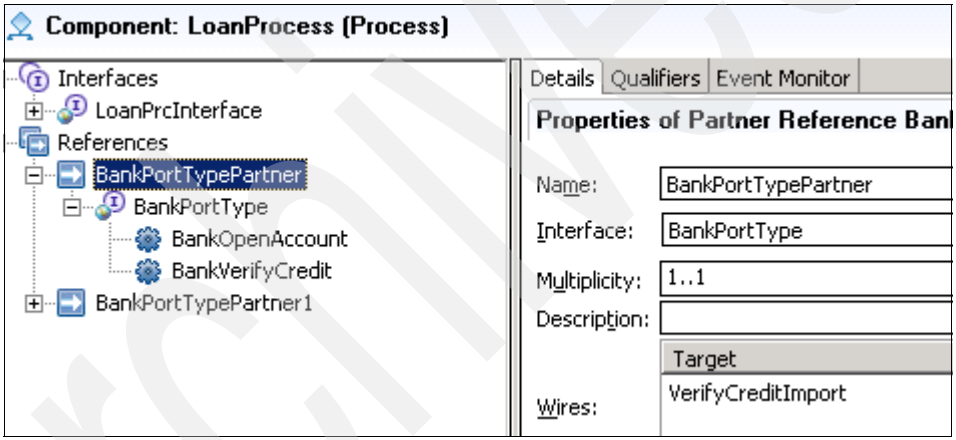


Figure 7-18 Business process reference

Archived

- BankPortTypePartner1 (Figure 7-19) also uses the BankPortType interface to call the service. However, in this case, the reference partner is wired to the OpenAccountImport.

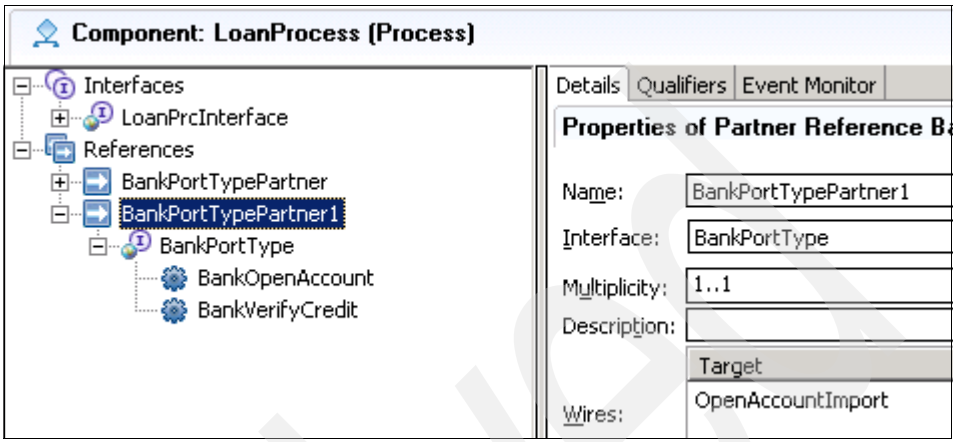


Figure 7-19 Business process reference

Receive activity

The Receive activity (Figure 7-20) receives the business process input from an external client. It is associated with the LoanPrcInterface interface.

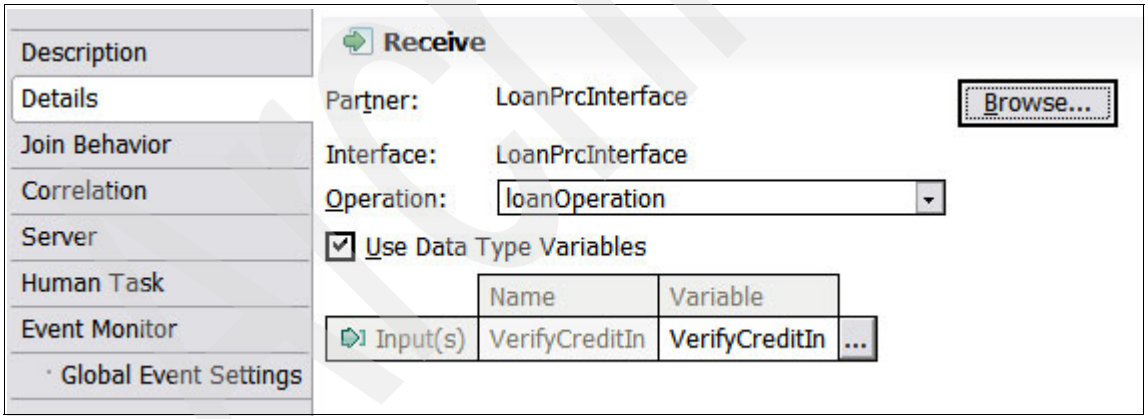


Figure 7-20 Receive activity configuration

ConfigSSLEnvironment Java Snippet activity

The interface that is used by LoanProcess to connect to the message flows in WebSphere Message Broker is bound by using an HTTP secure channel. Most of the HTTPS configuration is done at the application server level, but a few configuration steps are required at the business process level.

Before an import can make an external call, a few runtime properties must be defined to the application JRE environment. Those variables can be defined as entry values during the application server initialization (-D properties), or directly with Java code.

In this scenario, before the bound ports can be executed on an external HTTPS service call, a Java snippet activity (Figure 7-21) with the SSL configuration is defined. This activity is added just after the Receive activity call and before an import can be executed.

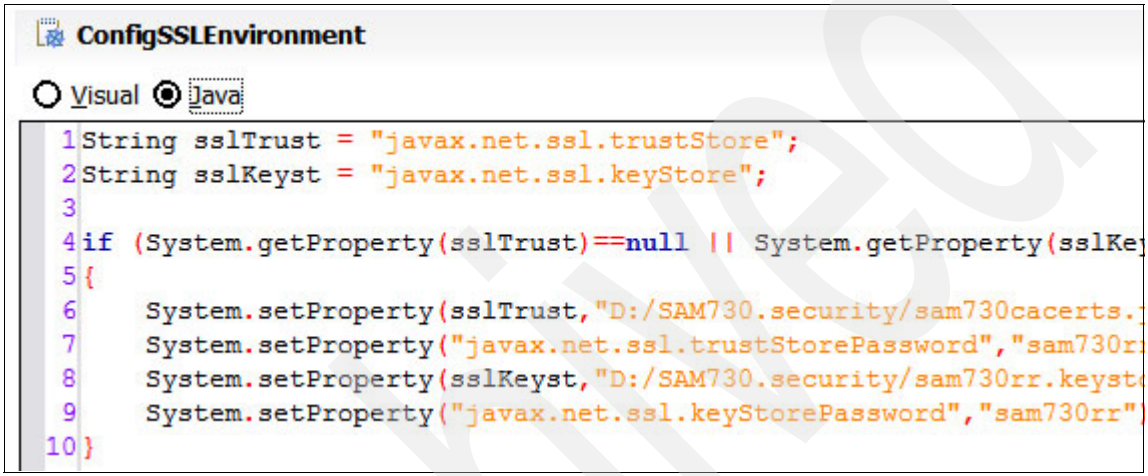


Figure 7-21 Java snippet used to set the SSL properties into the JRE environment

This Java snippet code defines the required SSL properties to use HTTPS (Example 7-9).

Example 7-9 JRE SSL environment configuration

```
...
System.setProperty("javax.net.ssl.keyStore", <<keystore path>>);
System.setProperty("javax.net.ssl.keyStorePassword",<<password>>);
System.setProperty("javax.net.ssl.trustStorePassword",<<password>>);
System.setProperty("javax.net.ssl.trustStore", <<truststore path>>);
...
```

Keystore and truststore file system definitions: The file system definitions for the keystore and truststore must reflect the same stores that are both used in the WebSphere application server as in the WebSphere Message Broker.

Archived

VerifyCreditInvoke Invoke activity

The VerifyCreditInvoke Invoke activity (Figure 7-22) is associated with the BankPortTypePartner partner reference. It uses the BankPortType interface to send an HTTPS request to the WebSphere Message Broker.

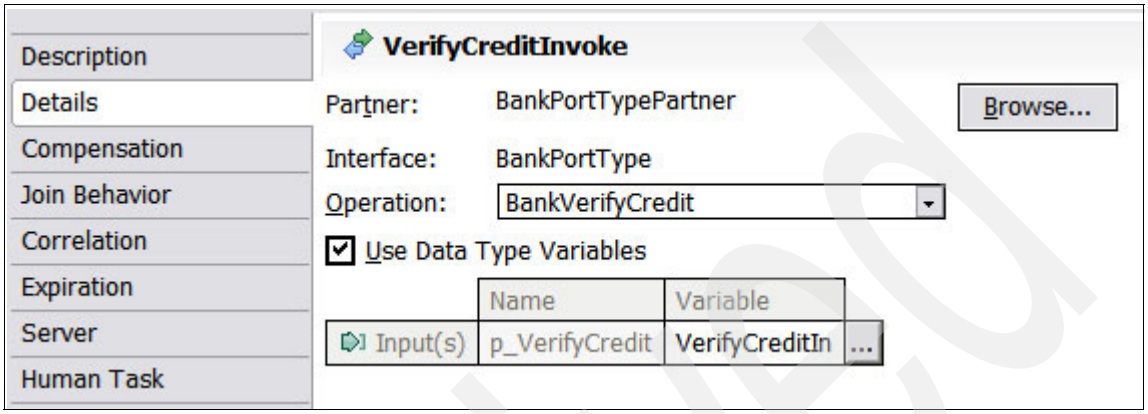


Figure 7-22 VerifyCredit Invoke activity

Snippet activity

The Snippet Java snippet activity (Figure 7-23) is used to generate the content that is expected by the OpenAccount business object.

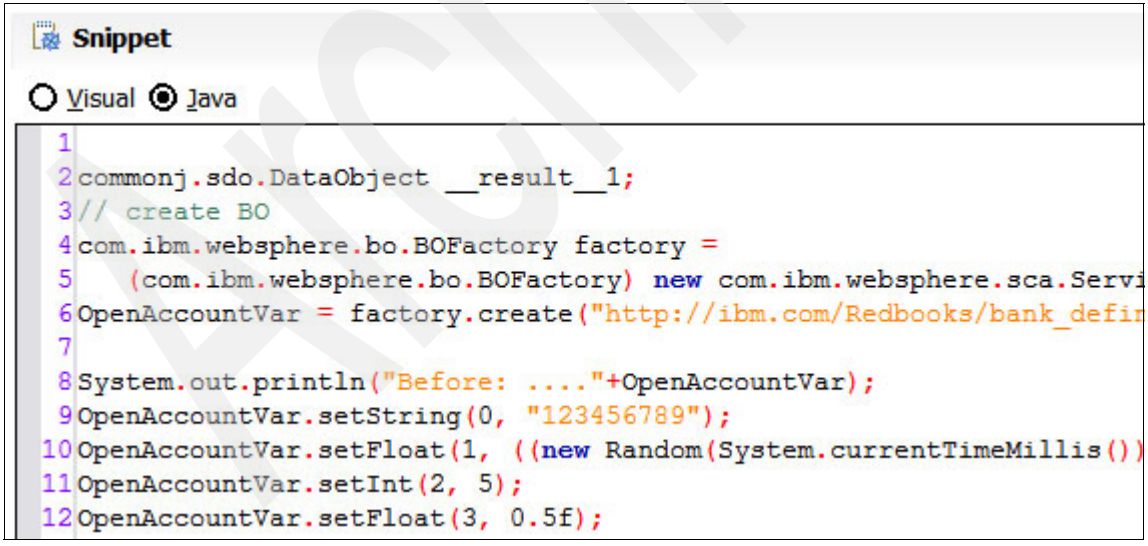


Figure 7-23 Dynamic generating content for the OpenAccount business object

Archived

OpenAccountInvoke activity

The OpenAccountInvoke Invoke activity (Figure 7-24) is associated with the BankPortTypePartner1 reference partner. It uses the BankPortType interface configuration to send the request to WebSphere Message Broker.

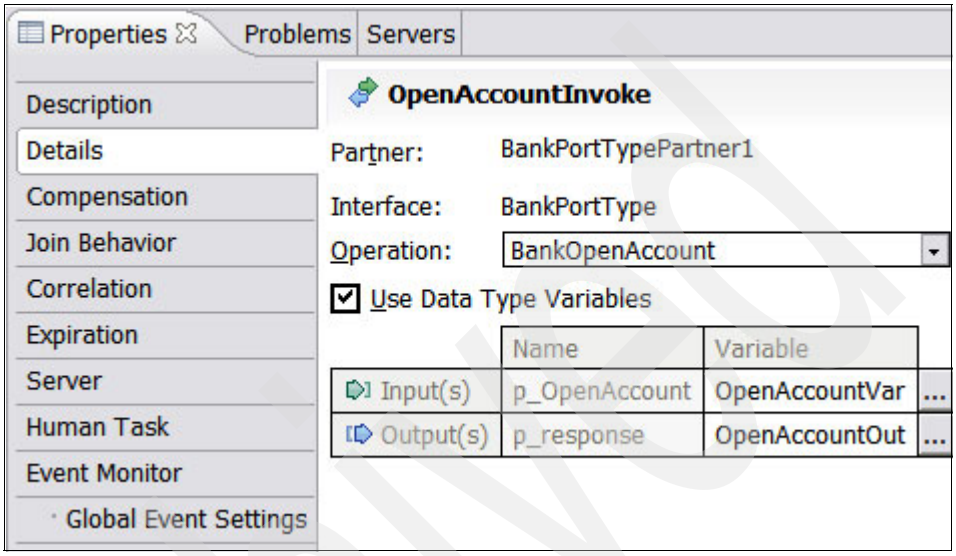


Figure 7-24 OpenAccountInvoke activity configuration

Reply activity

The last part of the business process execution is the Reply activity (Figure 7-25). The Reply activity is used to return a response to the calling client. It uses the LoanPrcInterface interface.

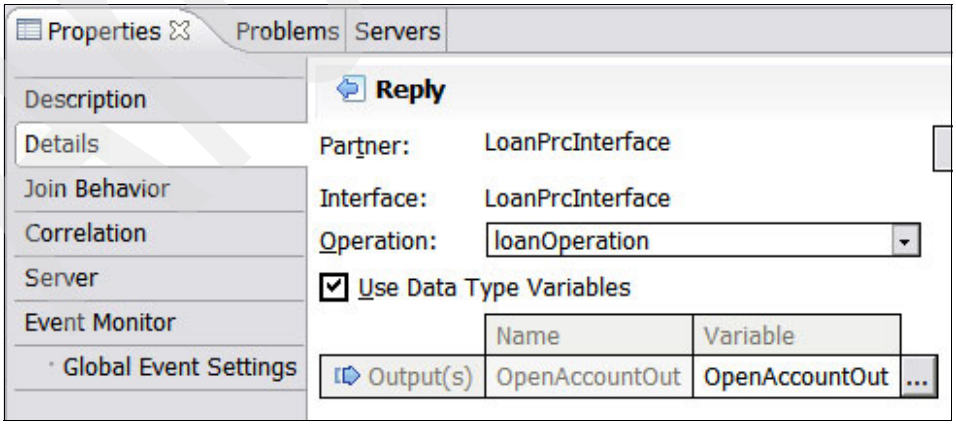


Figure 7-25 Reply activity configuration

7.5.4 Import components and binding

The business integration module uses two import components, VerifyCreditImport and OpenAccountImport, both with a SOAP/HTTP Web service binding. As the names imply, VerifyCreditImport is used to invoke the VerifyCredit message flow, and OpenAccountImport is used to invoke the OpenAccount message flow.

Figure 7-26 shows the properties for VerifyCreditImport.

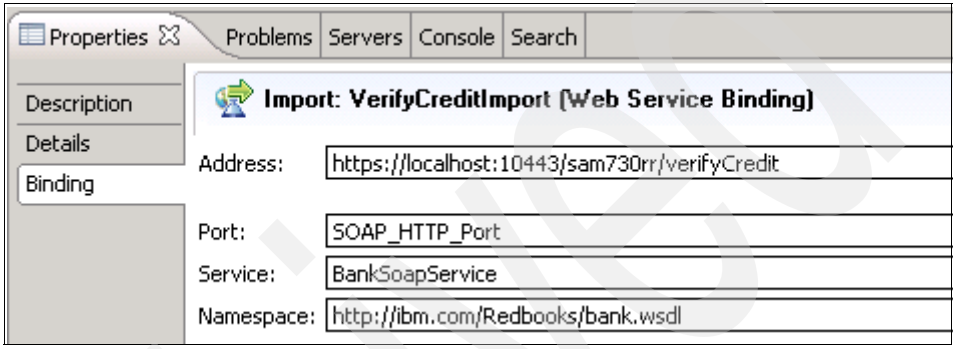


Figure 7-26 VerifyCreditImport component binding properties

Figure 7-27 shows the properties for OpenAccountImport.

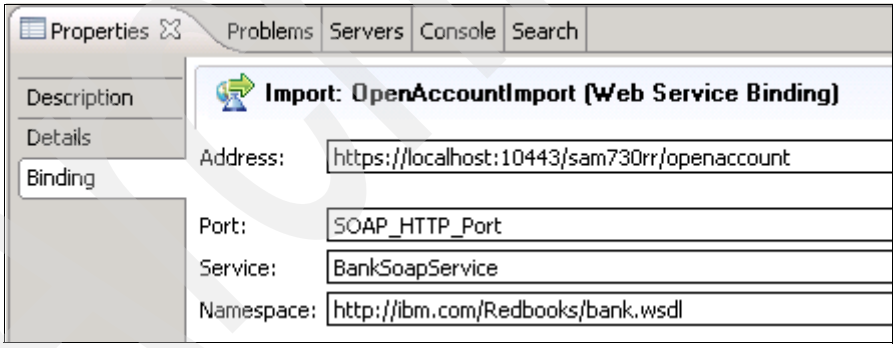


Figure 7-27 OpenAccountImport component binding properties

Advanced

7.6 Message flows

In this scenario, we use two message flows, the VerifyCredit message flow and the OpenAccount message flow, for the ESB function. The enterprise information systems are also emulated using message flows.

7.6.1 Message set

Both message flows use a message set (Figure 7-28) that contains message definitions to transform and parse the MRM message structure during the flow.

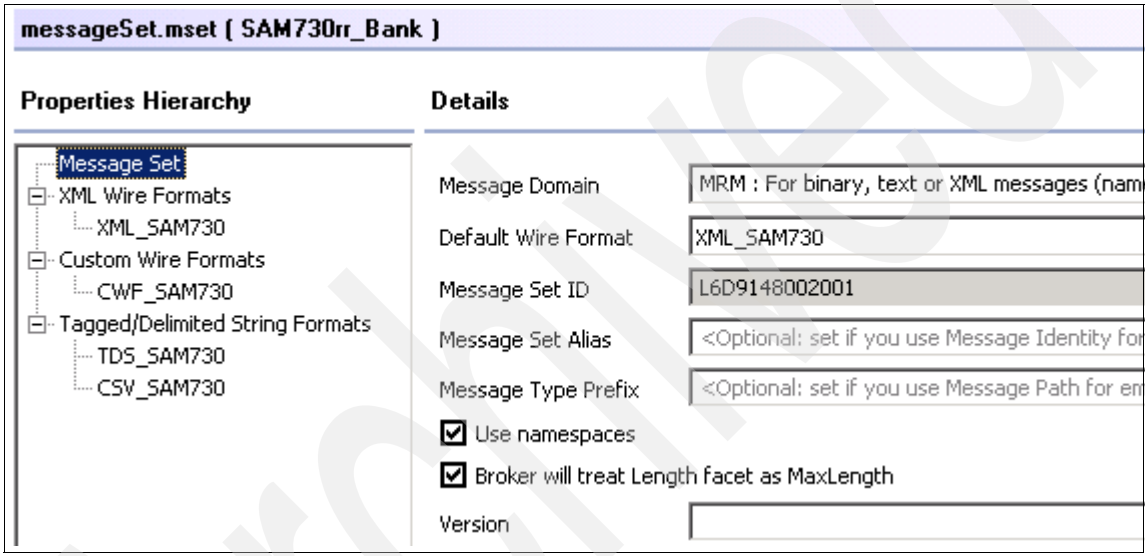


Figure 7-28 Message set SAM730rr_Bank

The messages are parsed using two physical formats:

- TDS

  The TDS format is used to parse and wire messages that are sent to the enterprise information systems in the VerifyCredit message flow.

- XML

  The XML format is used to transform messages for the enterprise information systems and to parse the SOAP envelope in the OpenAccount message flow.

Archived

TDS format for Legacy 1

The XML_2_TDS.mxsd message definition file shown in Figure 7-29 defines the proprietary string structure that is required for use with Legacy 1.

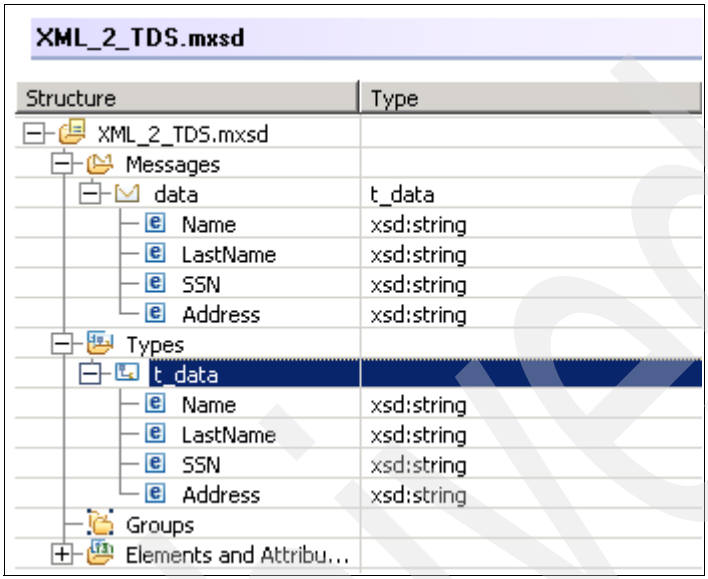


Figure 7-29 Message structure for Legacy 1 - XML_2_TDS.mxsd

Archived

The physical format for the t_data type, defined by TDS_SAM730, is used for proprietary string messages used with Legacy 1. You can see this format by selecting the t_data type in the message definition file and viewing its properties. Figure 7-30 shows the TDS_SAM730 physical format.

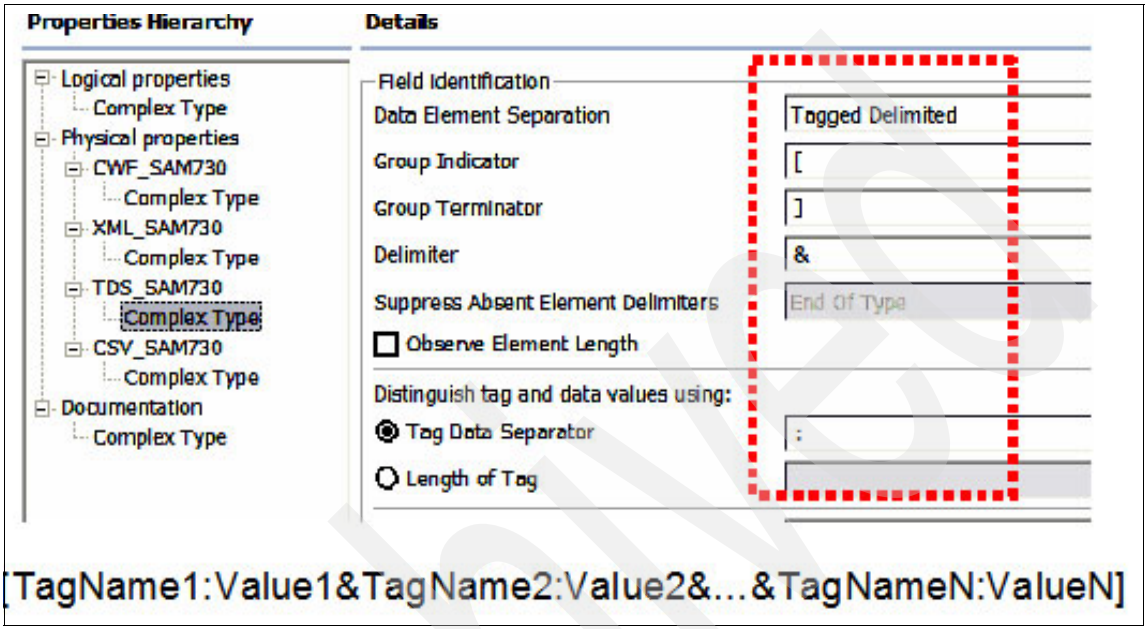


Figure 7-30 TDS_SAM730 physical format

Archive

TDS format for Legacy 2

The XML_2_X12.mxsd message definition shown in Figure 7-31 defines the structure that is required to parse an X12 string for Legacy 2.

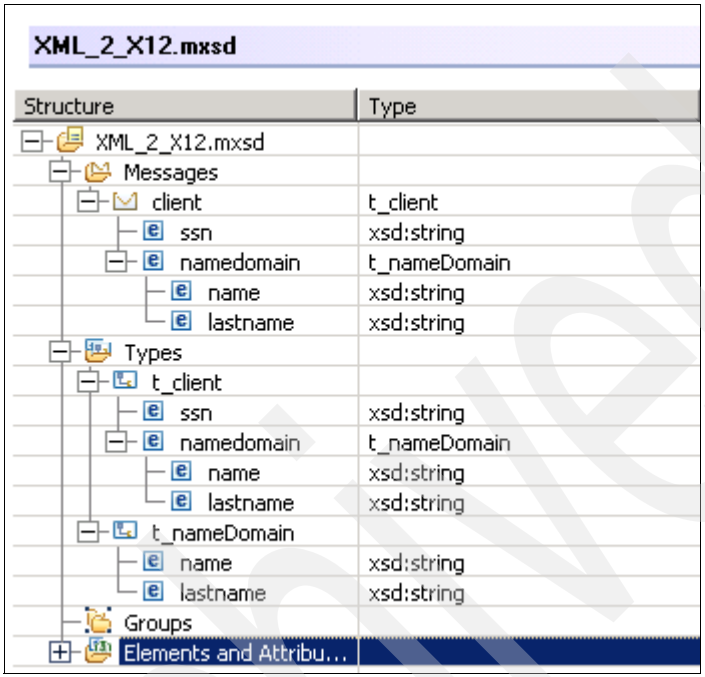


Figure 7-31 Message structure for Legacy 2 - XML_2_X12.mxsd

Archived

Two types are defined: t_client and t_nameDomain. Each type defined in this definition has a different TDS structure and organization. The message is parsed according to the specification in the message definition and TDS structure defined by the CSV_SAM730 format (Figure 7-32).

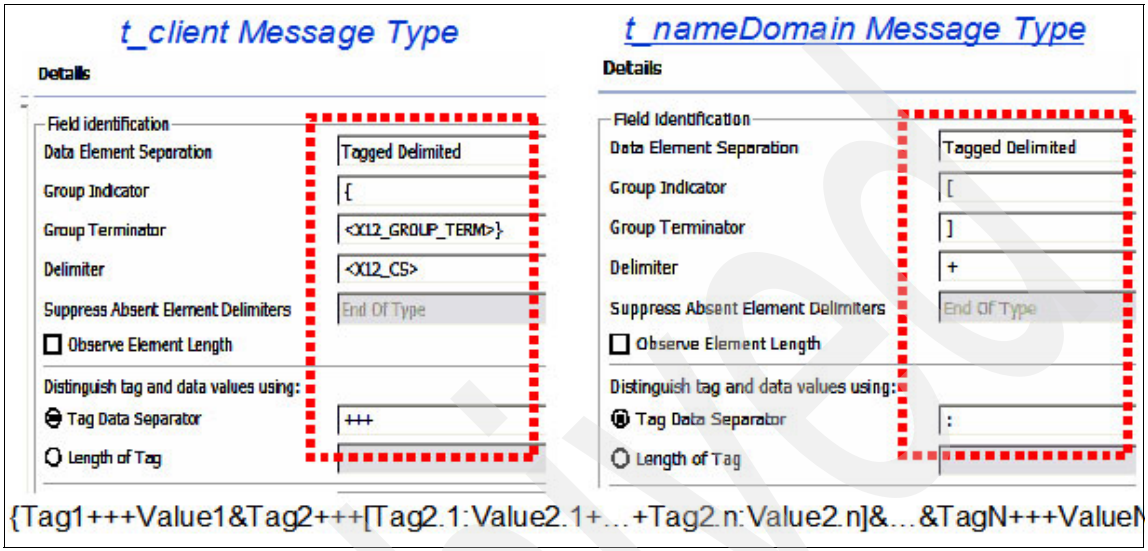


Figure 7-32 CSV_SAM730 physical format

Archived

XML

The XML message format is called XML_SAM730. It contains message definitions that are created by using the Create Message Definition from... wizard (Figure 7-33). The message definitions were created from the bank.wsdl file (BankPortType interface) and from both the service definitions schema, bank_definitions.xsd, and legacy-service (plain XML service definitions) schema, legacy.xsd.

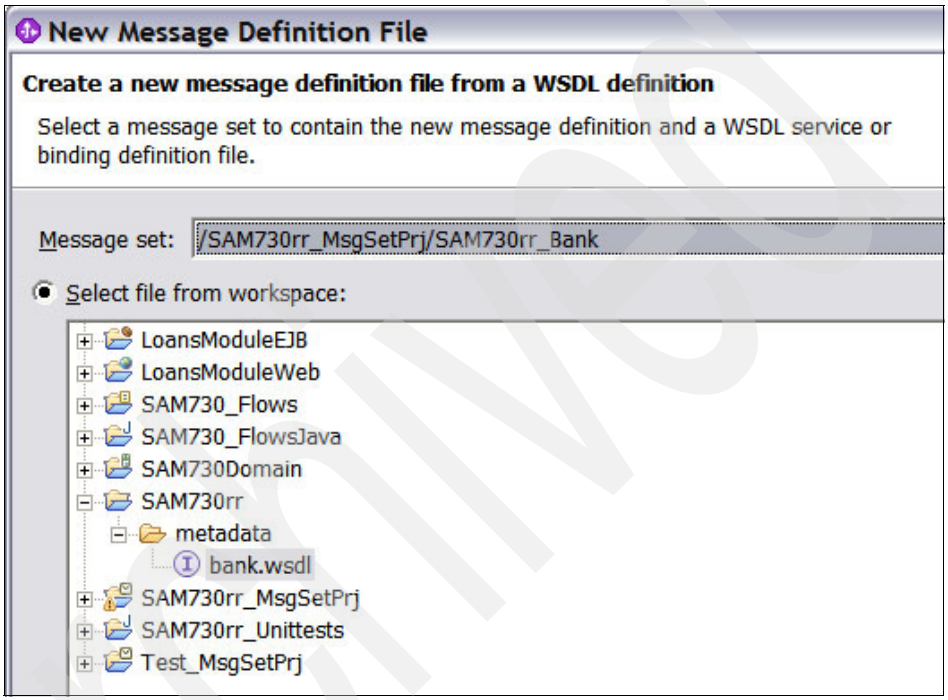


Figure 7-33 Create new message definition from a WSDL wizard

When the MRM domain receives a SOAP message through an input node, the HTTPInput node, it parses the message using the Envelope message type that was created automatically by the wizard. Therefore, the broker cannot parse business elements inside the SOAP Body tag.

MRM domain: The following steps are necessary only if you are using the MRM domain as the message structure parser.

Archived

To resolve this issue after the wizard completes:

1. Open the **soapenv11.mxsd** definition in the org.xmlsoap.schemas.soap.envelope namespace.
2. Select the **soapenv11.mxsd** file in the navigation view and then select the **Properties** view. See Figure 7-34.

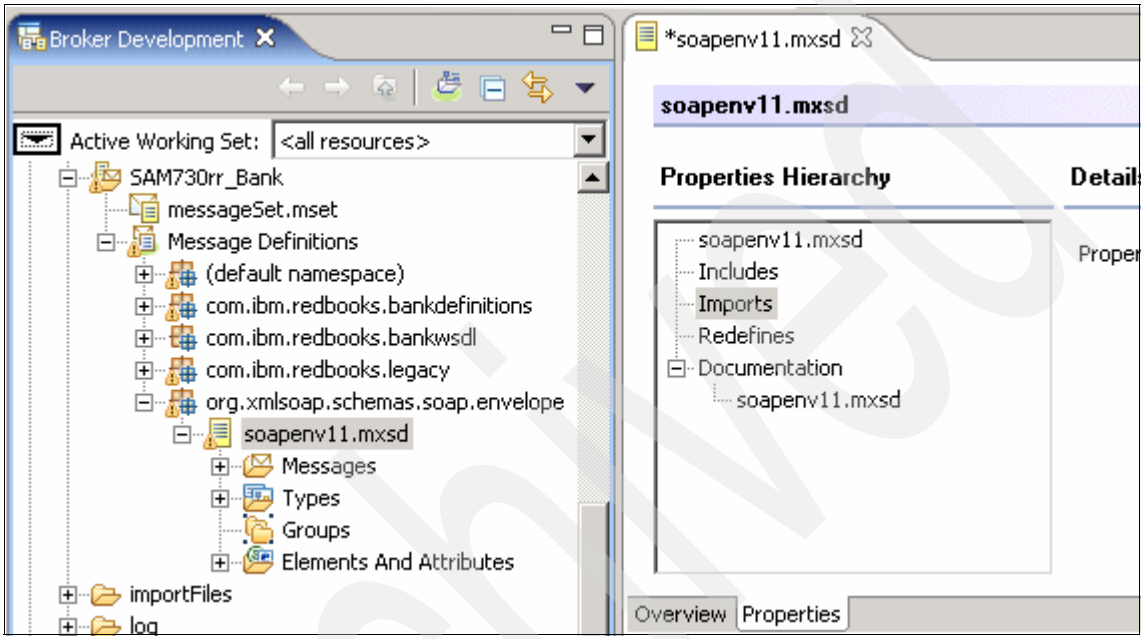


Figure 7-34 Properties view of soapenv11.mxsd

3. Right-click **Imports** and select **Add Import**.

Archived

4. Select the **bank_definitions.mxsd** file as shown in Figure 7-35.

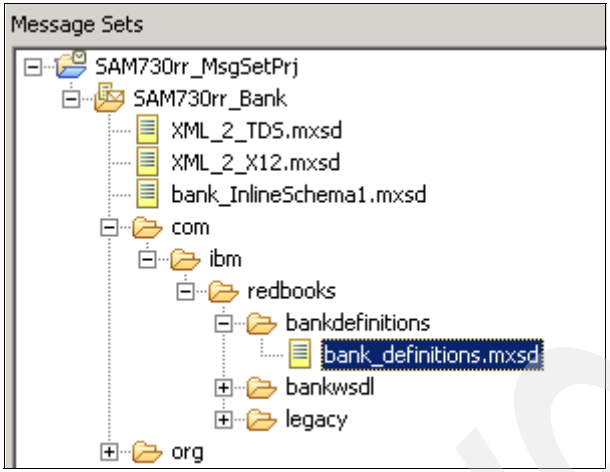


Figure 7-35 Select bank_definitions.mxsd

Figure 7-36 shows the results.

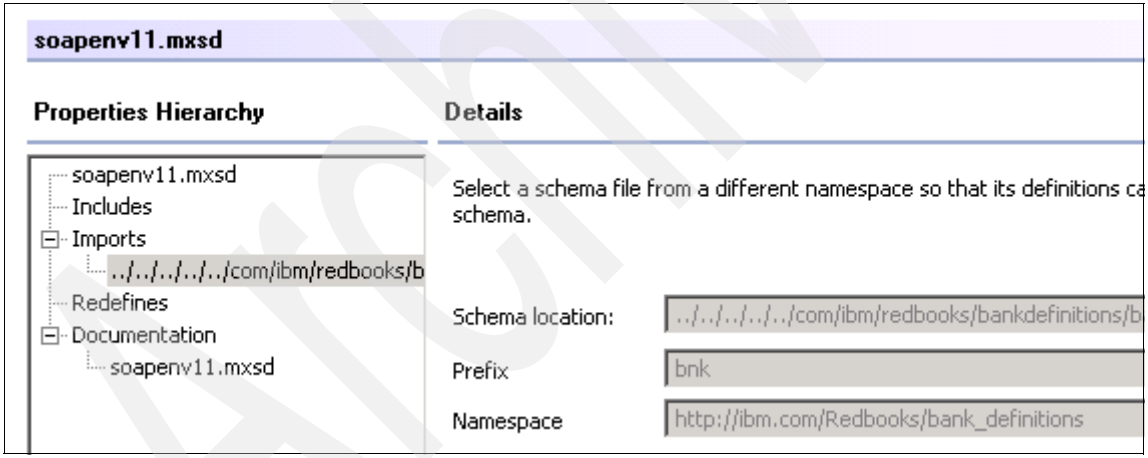


Figure 7-36 Business elements imported to the soapenv.mxsd file

Archived

5. Add the elements to the SOAP Body message as required, and all the incoming services are validated on the business level. See Figure 7-37.

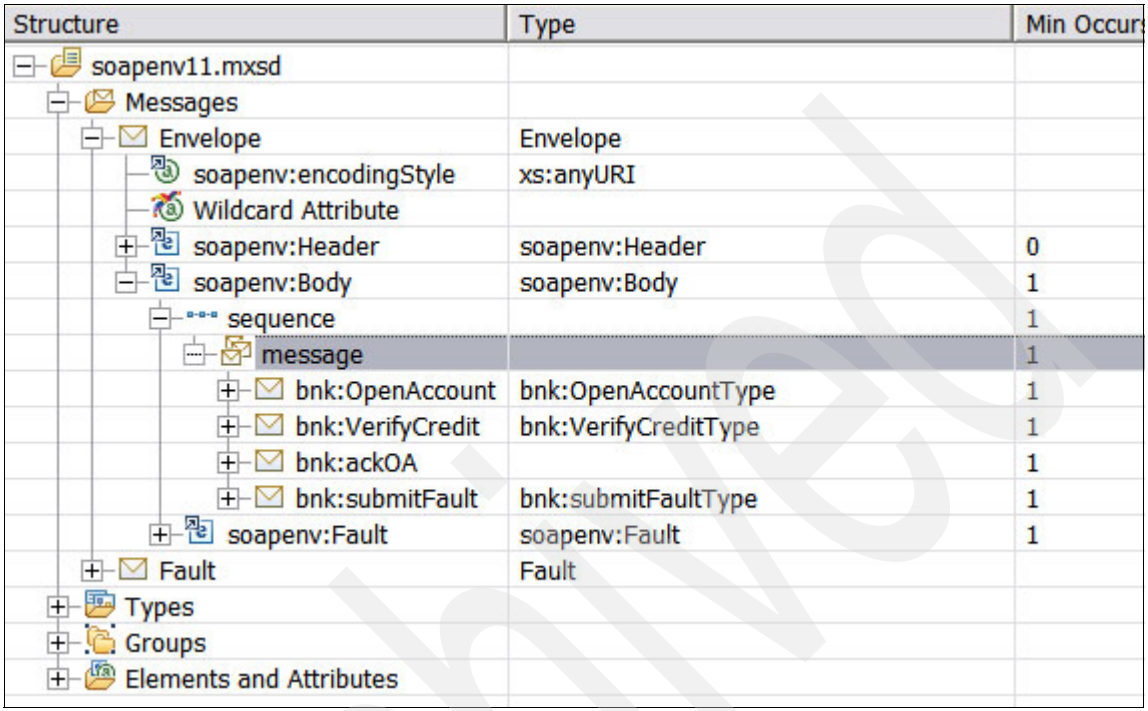


Figure 7-37 Imported business elements inside the soapenv.mxsd file

The business validation can also be executed by using the XML/XMLNS domain, removing the SOAP envelope part from the message, and then adding the client MessageType within a Validate node.

7.6.2 The com.ibm.redbooks.sam730.BankUtils class reference

All the JavaCompute nodes that are used in the message flows use a utilitary class called `BankUtils.java`. This class is responsible for maintaining all the auxiliary methods (functions), such as copying headers, that are used. Table 7-2 on page 205 provides a brief description of this methods for this class.

Archived

Table 7-2 BankUtils methods

Method name	Description
`void formatFaultMessage(...)`	Creates a SOAP fault message structure that is based on the ExceptionList that is passed as a parameter.
`void copyHeaders(...)`	Copies header elements from the InputRoot to the OutputRoot. A string can be passed to this method that specifies header elements to be excluded from the copy.
`void copyChildElements(...)`	Copies child elements in the RootElement. A string is passed to the method specifying which child elements to copy and where to copy them.
`boolean verifyHeader(...)`	Verifies if the header name that is passed as a parameter exists on the Input/OutputRoot structure.
`boolean createMQMD(...)`	Creates a MQMD header on the OutputRoot message structure.

Tip: For performance, all external method calls from the JavaCompute node itself must only be from runtime static methods, not from a newly instantiated class.

7.6.3 Enabling the HTTPInput node for SSL

Both the VerifyCredit message flow and OpenAccount message flow are bound using an HTTP Secure connection. Therefore, a secure session, cipher, and handshake must be created before the requestor can access and execute the services that are defined on WebSphere Message Broker.

After defining the broker runtime configuration for the SSL handshake, the message flow must also be configured to listen to the right context root (Table 7-3). This configuration is defined on the Input node (HTTPInput node).

Table 7-3 Bound context roots

Message flow	Bound context root
VerifyCredit message flow	/sam730rr/verifyCredit
OpenAccount message flow	/sam730rr/openaccount

In the Basic properties for the HTTPInput node, select the **Use HTTPS** check box (Figure 7-38).

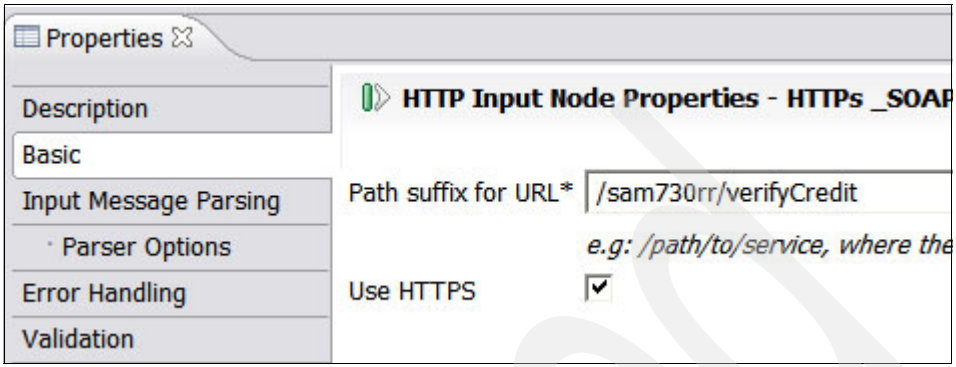


Figure 7-38 HTTPS configuration

HTTPReply node: No SSL configuration is needed for the HTTPReply node.

Archived

7.6.4 VerifyCredit message flow

Figure 7-39 shows the VerifyCredit message flow.

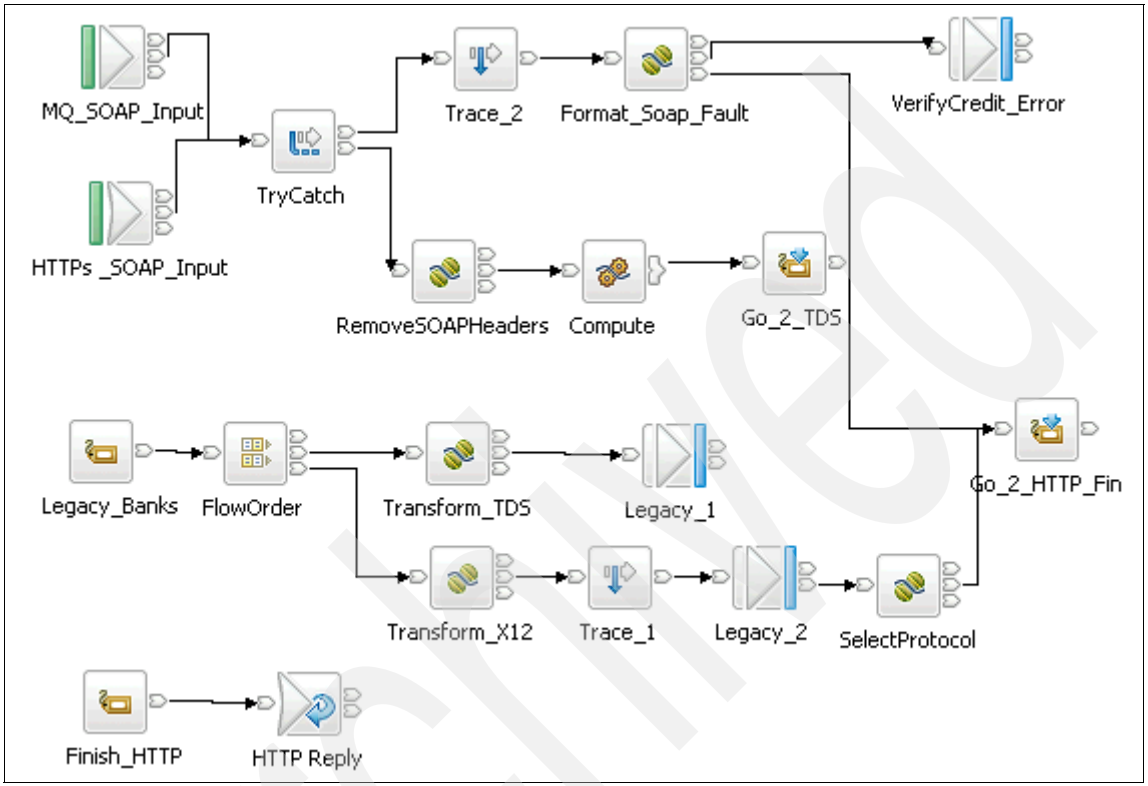


Figure 7-39 VerifyCredit message flow

The message flow has two input nodes. An HTTPInput node called *HTTPs_SOAP_Input* is the production node. The import component in WebSphere Process Server has a Web service binding that uses the URL of this node. An MQInput node called *MQ_SOAP_Input* was used for testing.

Every time the service requestor, which in this case is WebSphere Process Server, initiates the VerifyCredit message flow, a protocol selection occurs inside the flow. When the broker receives a request, it uses a message tree structure to route the flow to the appropriate nodes (Figure 7-40 on page 208) depending on the incoming request protocol (HTTP or MQ).

Archived

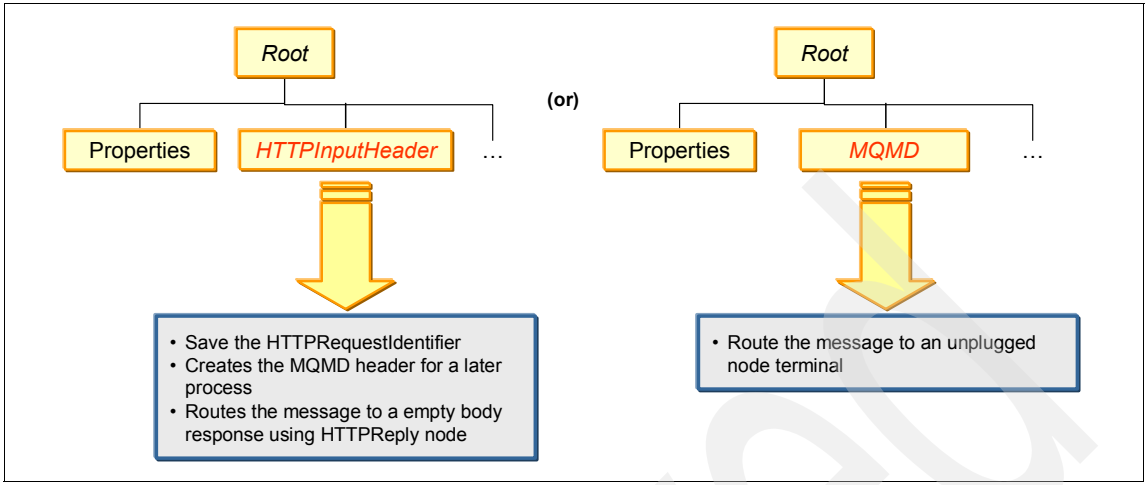


Figure 7-40 Dynamic routing and switching of protocol

HTTPs_SOAP_Input node

The HTTPs_SOAP_Input HTTPInput node receives the message from WebSphere Process Server. The incoming message is parsed using the MRM domain with the SOAP envelope message type. This configuration is defined directly on the HTTPInput node and is necessary for the correct creation of the message structure (Figure 7-41).

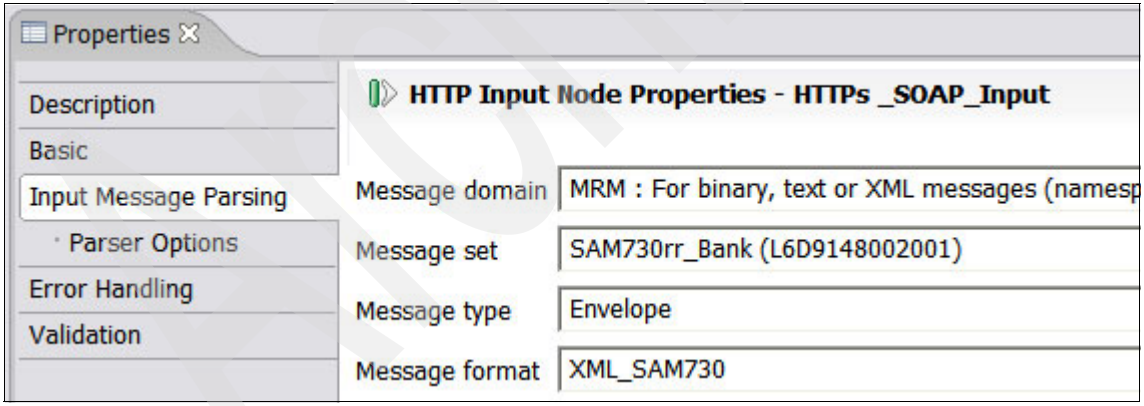


Figure 7-41 HTTPInput node message parsing definition

The Use HTTPS option is enabled as discussed in 7.6.3, “Enabling the HTTPInput node for SSL” on page 205.

RemoveSOAPHeaders JavaCompute node

When the message arrives from the input node, the SOAP envelope elements must be removed from the original structure. This is done by using a JavaCompute node (Example 7-10).

Example 7-10 Removing the SOAP envelope part from the message structure

```
...
// Copy all headers excluding the MRM message structure
BankUtils.copyHeaders(inMessage, outMessage, new String[]{"MRM"});

MbElement inRootElement  = inMessage.getRootElement();
MbElement outRootElement = outMessage.getRootElement();
// Select all the chields of body from the InputRoot
String bodyChields = "/MRM/Body";

MbElement parser = outRootElement.createElementAsLastChild("MRM");
// Retrives the soap body chields from the InputRoot //
MbElement element = inRootElement.getFirstElementByPath(bodyChields);

// Copy all the chields of the body element to the output message //
if (element!=null)
   parser.copyElementTree(element);
// Just copy everything...
else
   outRootElement.copyElementTree(inRootElement);
...
```

Compute and RouteToLabel nodes

After removing the SOAP envelope elements (Header/Body), the message is routed using a RouteToLabel node to a the Legacy_Banks secondary flow. This routing is configured using a simple Compute node that adds a new labelname attribute (Example 7-11) in the LocalEnvironment structure. The RUNTDS label identifies the Legacy_Banks Label node.

Example 7-11 Compute node

```
...
SET OutputLocalEnvironment = InputLocalEnvironment;
-- Go to the label called RUNTDS
SET
OutputLocalEnvironment.Destination.RouterList.DestinationData[1].labeln
ame = 'RUNTDS';
...
```

The Compute mode property must be set to **LocalEnvironment and Message** (Figure 7-42).

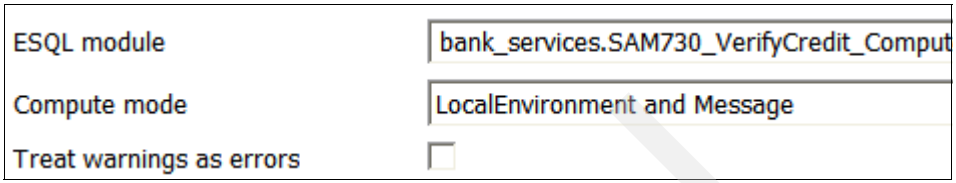


Figure 7-42 Compute node configuration

FlowOrder node

This secondary flow uses a FlowOrder node to sequence the incoming message to two different endpoints, Legacies 1 and 2.

The first execution goes to a JavaCompute node called *Transform_TDS*. When processing is complete, the message is sent to another JavaCompute node called *Transform_X12*. These two nodes prepare the message for delivery to the enterprise information systems.

Transform_TDS JavaCompute node

The Transform_TDS JavaCompute node transforms the incoming message to a format that is expected by Legacy 1 using the XML_2_TDS.mxsd message definition (Figure 7-43).

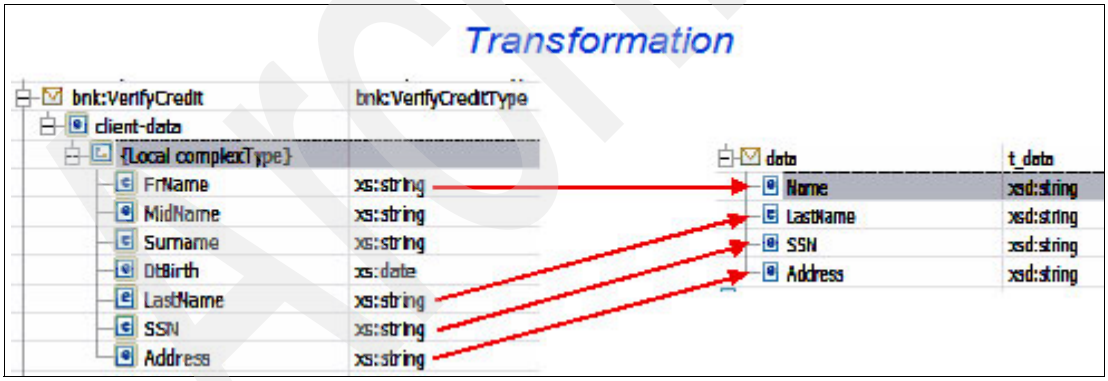


Figure 7-43 VerifyCredity to data type message transformation

Archive

The transformation is done using XPath (Example 7-12).

Example 7-12 XPath transformation on a JavaCompute node: Transform_TDS code

```
...
// Prepare the MQMD header if it doesn't exist //
BankUtils.createMQMD(outRootElement);

// Creates the parser //
MbElement parser = outRootElement.createElementAsLastChild("MRM");
// Creates the data message that will be used by the message set //
MbElement dataElement =
   parser.createElementAsLastChild(MbElement.TYPE_NAME, "data", null);

// Define the TDS element structure using XPath //
MbXPath xpName =
   new MbXPath("/?$Name[set-value($client_name)]", dataElement);
MbXPath xpLName =
      new MbXPath("/?$LastName[set-value($client_lname)]",
dataElement);
MbXPath xpSSN =
   new MbXPath("/?$SSN[set-value($ssn)]", dataElement);
MbXPath xpAddr =
   new MbXPath("/?$Address[set-value($addr)]", dataElement);
// Populate the variables defined above //
xpName.assignVariable("client_name", ...);
xpLName.assignVariable("client_lname", ...);
xpSSN.assignVariable("ssn", ...);
xpAddr.assignVariable("addr", ...);

// Evaluate the XPath definitions //
outMessage.evaluateXPath(xpName);
outMessage.evaluateXPath(xpLName);
outMessage.evaluateXPath(xpSSN);
outMessage.evaluateXPath(xpAddr);
...
```

The following actions occur in the Transform_TDS code:

1. The code creates the root message structure that is expected by the MessageSet /Root/MRM/data/.
2. The XPath expressions are defined by using the “data” element in the message as the root element for the new message structure (set in the dataElement variable). The second argument of the MbXpath constructor defines dataElement as the root element.

a. The code creates all the elements if they do not exist and populates their values with XPath signed variables, for example:

```
/?$SSN[set-value($ssn)]
```

b. The code assigns values for all the variables, for example:

```
xpSSN.assignVariable("ssn", "123456789");
```

3. The code evaluates all XPath expressions on the Output message object, for example:

```
outMessage.evaluateXPath(xpSSN);
```

After transforming the message using XPath expressions, the node defines the parser parameters for the TDS_SAM730 physical format (Example 7-13) inside the flow's MessageSet.

Example 7-13 TDS_SAM730 MessageSet configuration

```
...
MbElement props = outRootElement.getFirstElementByPath("Properties");
props.getFirstElementByPath("MessageSet").setValue("L6D9148002001");
props.getFirstElementByPath("MessageType").setValue("data");
props.getFirstElementByPath("MessageFormat").setValue("TDS_SAM730");
...
```

These changes are made inside the Properties message tree structure (Table 7-4).

Table 7-4 MessageSet configuration

MessageSet property name	MessageSet property value
MessageSet	SAM730rr_Bank (L6D9148002001)
MessageType	data
MessageFormat	TDS_SAM730

Transform_X12

The Transform_X12 JavaCompute node transforms the incoming message to a format that is expected inside the XML_2_X12.mxsd message definition (Figure 7-44).

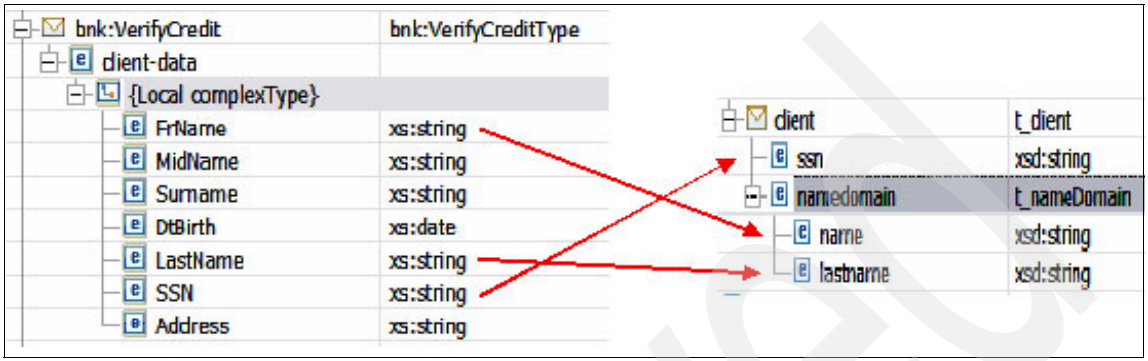


Figure 7-44 VerifyCredit to client type data transformation

Example 7-14 shows the transformation of XPath to the X12 format.

Example 7-14 XPath transformation to the X12 format: Transform_X12 code

```
// Prepare the MQMD header if he doesn't exist //
BankUtils.createMQMD(outRootElement);
// Creates the MRM message parser //
MbElement mrmElement = outRootElement.createElementAsLastChild("MRM");
MbElement cliElement =
   mrmElement.createElementAsLastChild(MbElement.TYPE_NAME, "client", null);
MbElement ndElement  =
   cliElement.createElementAsLastChild(MbElement.TYPE_NAME, "namedomain", null);

// Evaluate XPath for the X12 legacy //
MbXPath xpSSN = new MbXPath("/?$ssn[set-value($ssn)]", cliElement);
MbXPath xpClientLName = new MbXPath("/?$lastname[set-value($lname)]", ndElement);
MbXPath xpClientName  = new MbXPath("/?$name[set-value($fname)]", ndElement);
// define a value for the XPath variables //
xpSSN.assignVariable("ssn", ...);
xpClientLName.assignVariable("lname", ...);
xpClientName.assignVariable("fname", ...);
// EVALUATE XPATH...
outMessage.evaluateXPath(xpSSN);
outMessage.evaluateXPath(xpClientLName);
outMessage.evaluateXPath(xpClientName);
...
```

Archived

The following actions occur in the Transform_X12 code:

1. The first Java instruction creates an MQMD header inside the OutputRoot if the incoming request came from HTTPS.
2. The code creates the root message structure expected by the MessageSet: `/Root/MRM/client/` and `/Root/MRM/client/namedomain/`.
3. The second argument of the MbXPath constructor is the root element for the XPath expression. All the XPath expressions are defined by using the `client` and `namedomain` messages as the root elements for the message composition. The MbXpath statements perform the following tasks:
   a. Create the elements if they do not exist and populate the values with XPath signed variables, ssn:

      ```
      /?$ssn[set-value($ssn)]
      ```

   b. Assign values for all the variables:

      ```
      xpSSN.assignVariable("ssn", "123456789");
      ```

4. The XPath expressions are evaluated on the Output message object:

   ```
   outMessage.evaluateXPath(xpSSN);
   ```

Once again, the MessageSet configuration defined inside the Properties message tree must be changed (Example 7-15).

Example 7-15 Configuring the client message type and the CSV_SAM730 physical wire

```
...
MbElement props = outRootElement.getFirstElementByPath("Properties");
props.getFirstElementByPath("MessageSet").setValue("L6D9148002001");
props.getFirstElementByPath("MessageType").setValue("client");
props.getFirstElementByPath("MessageFormat").setValue("CSV_SAM730");
...
```

XPath support: The current version of WebSphere Message Broker only supports the XPath 1.1 specification.

Archive

The broker must combine the logical XML_2_X12.mxsd message definition with the CSV_SAM730 physical format to transform the incoming message before sending that data to the Legacy 2 system (Table 7-5).

Table 7-5 XML_2_X12 MessageSet configuration

MessageSet property name	MessageSet property value
MessageSet	SAM730rr_Bank (L6D9148002001)
MessageType	client
MessageFormat	CSV_SAM730

Malformed messages

Before the message reaches the MQOutput nodes for Legacy 1 and Legacy 2, the logical (message definition) structures are parsed by the run time (Figure 7-45) using the physical formats that are defined for each endpoint (TDS_SAM730 and CSV_SAM730). The broker run time constructs and formats the message before it is delivered to MQ. In the case of a malformed message, an exception is thrown, an exception list is added to the message tree, and the message is delivered to the preceding JavaCompute node.

When an exception occurs, the message is rolled back to the last well-formed structure that was defined before the error occurred.

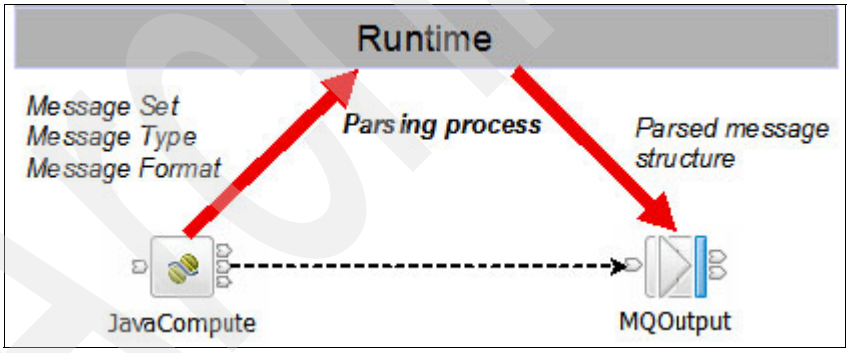


Figure 7-45 Message transformation

SelectProtocol JavaCompute node

The SelectProtocol JavaCompute node (Example 7-16 on page 216) is used to route the message using the predefined connection terminals of Out and Alternate. If the message originated as an HTTP request, the flow is dispatched using the Out terminal to a RouteToLabel node, which produces an empty HTTP response to the requestor (one-way interaction service) using a secondary

response. If this message came from MQ, the flow is delivered to an unconnected terminal (the Alternate terminal), and the process ends.

Example 7-16 Message routing using JavaCompute terminals

```
...
// Verify if this is a HTTP request //
if (BankUtils.verifyHeader(inMessage.getRootElement(),
                           "HTTPInputHeader")) // HTTP
{
   MbMessage localEnvMsg = outAssembly.getLocalEnvironment();
   String xpStr = // Must have the Destination element already defined
"(//Destination/?RouterList/?$DestinationData/?$@labelname)[set-value($
name)]";
   // Defines the route: labelName using XPath (LocalEnv must exist)
   MbXPath xpRoute = new MbXPath(xpStr, localEnvMsg.getRootElement());
   // Sets the value of the variable name
   xpRoute.assignVariable("name", NODE_ROUTE_NAME);
   // Defines the routing part of the message //
   localEnvMsg.evaluateXPath(xpRoute);
////////////////////////////////////////////////////////////////////

   // Creates a empty body for the SOAP response //
   MbElement mrm = outRootElement.createElementAsLastChild("MRM");
   mrm.createElementAsLastChild(MbElement.TYPE_NAME, "Body", null);

   // Define the MessageSet to create the message in a CWF format
   // Message set: SAM730rr_Bank
   MbElement props =
      outRootElement.getFirstElementByPath("Properties");
   props.getFirstElementByPath("MessageType").setValue("Envelope");
   props.getFirstElementByPath("MessageFormat").setValue("XML_SAM730");

   out.propagate(outAssembly);
}
else // MQ
{
   // Ends the message...
   alt.propagate(outAssembly);
}
...
```

Archive

The following actions occur in the SelectProtocol code:

1. The code verifies if this is an HTTP request by using the `verfyHeader` method. If this is an MQ request, the flow is automatically directed to the alternative terminal and finishes the execution. Otherwise it continues:

   ```
   if (BankUtils.verifyHeader(inMessage.getRootElement(),
   "HTTPInputHeader")) {...}
   ```

2. After selecting the protocol as a HTTP request, an XPath expression is created to define an attribute inside the LocalEnvironment structure. This attribute is used by the RouteToLabel node to dispatch the message to the correct destination:

   ```
   (//Destination/?RouterList/?$DestinationData/?$@labelname)[set-value
   ($name)]
   ```

3. To finish, a SOAP envelope is defined as a MessageType, and the flow is propagated to the connected Out terminal:

   ```
   props.getFirstElementByPath("MessageType").setValue("Envelope");
   props.getFirstElementByPath("MessageFormat").setValue("XML_SAM730");
   ...
   out.propagate(outAssembly);
   ```

TryCatch node

If an exception occurs during the flow execution, the error is caught by a TryCatch node that was defined at the beginning of the flow and handled by a JavaCompute node called *Format_Soup_fault*. This node uses the last child of the ExceptionList message structure to format a SOAP Fault message to the requestor (Example 7-17).

Example 7-17 BankUtils.formatFaultMessage method to format SOAP Fault message

```
...
// Format the Fault message //
BankUtils.formatFaultMessage(domain, exElementRoot, inRootElement);
...
```

Depending on the protocol of the requester, the message is either routed to an MQ error queue or returned directly to the open HTTPS handshake.

Archived

7.6.5 OpenAccount message flow

After the VerifyCredit message flow is invoked by the business process, the process calls the OpenAccount message flow (Figure 7-46). This flow sends information that is specified by the client to a queue for the enterprise information system. If the request loan amount is bigger than $5,000.00, the request goes to the default application, Legacy 1. Otherwise, the request goes to Legacy 2.

The flow expects to receive a positive or negative response from the service. Every external interaction in this scenario is secured using a public key infrastructure (PKI) defined for the HTTPS listener. The legacy messages are also secured using a 64-bit encryption with the MQ listeners.

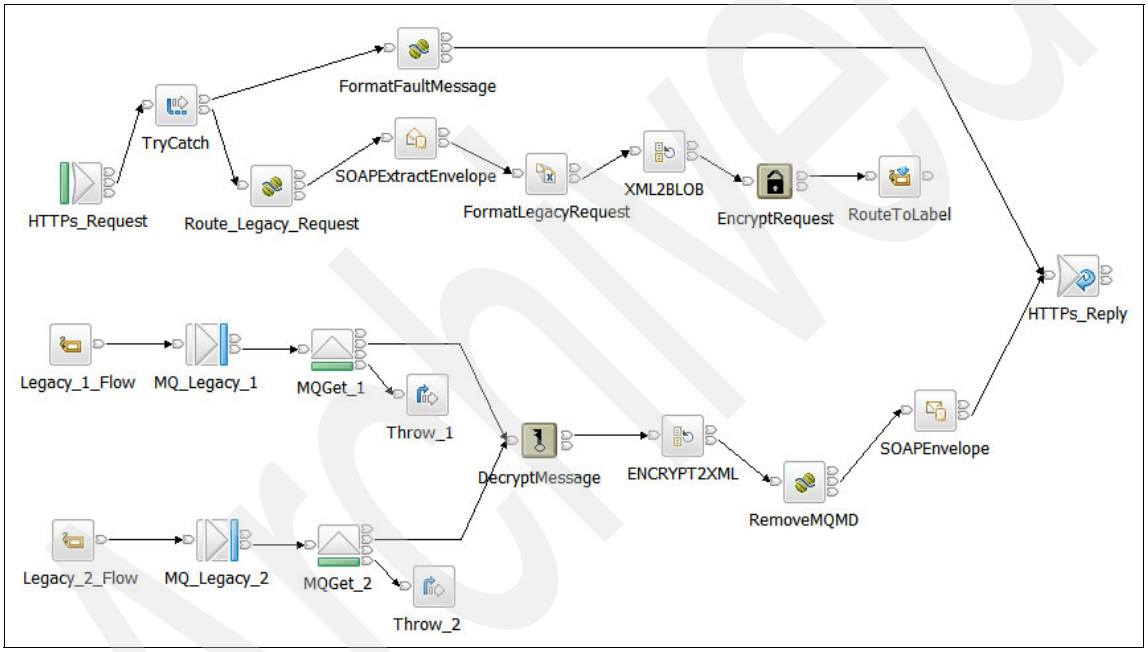


Figure 7-46 OpenAccount message flow

Archive

HTTPs_Request HTTPInput node

When a SOAP message arrives on the HTTPInput node, it is parsed using the MRM domain and the SOAP Envelope MessageType (Figure 7-47).

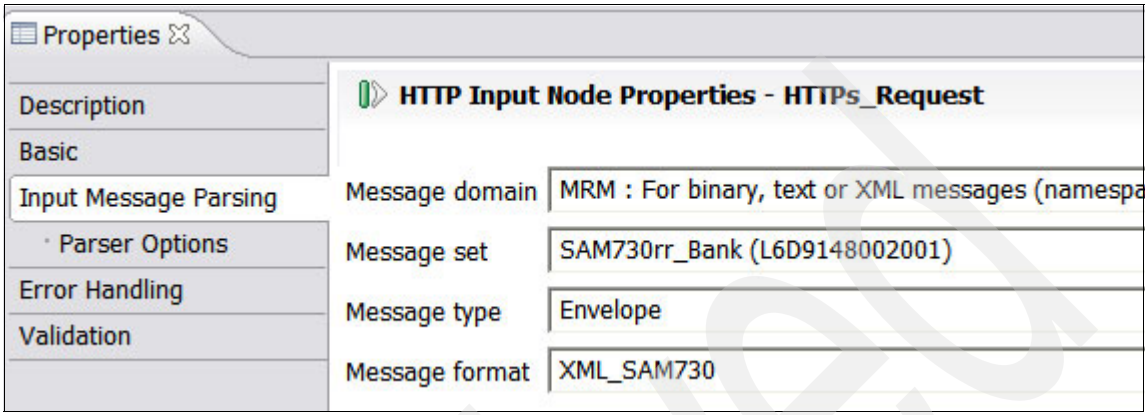


Figure 7-47 OpenAccount HTTPInput parser configuration

Route_Legacy_Request JavaCompute node

The message is routed by using an element value that is defined in the message. To perform this routing, the flow uses two nodes, a JavaCompute node called *Route_Legacy_Request* to prepare the routing, and a RouteToLabel node to dispatch the message to the right execution. However, additional processing is performed before the RouteToLabel node is reached in the flow.

The JavaCompute node uses the `<AskedAmount/>` element to determine the destination of the message (Example 7-18).

Example 7-18 Route_Legacy_Request code

```
...
String path = "/MRM/Body/OpenAccount/AskedAmount";
// Selects the amount specified by the requestor //
float amount = ((Number)
   inRootElement.getFirstElementByPath(path).getValue()).floatValue();
////////////////////////////////////////////////////////////////////////
// Selects the routing direction based on the amount passed by the
requestor
////////////////////////////////////////////////////////////////////////
String direction = (amount > 5000) ? "PORT_1" : "PORT_2";

// Define the routing label used by the RouteToLabel Node //
String xlabel =
```

```
"(/?Destination/?RouterList/?$DestinationData/?$@labelname)[set-value($
name)]";
// Defines the route: labelName attribute using XPath (LocalEnv must
exist)
MbXPath xpRoute = new MbXPath(xlabel, lenvMessage.getRootElement());
xpRoute.assignVariable("name", direction); // Sets the value of the
variable name
// Defines the routing part of the message //
lenvMessage.evaluateXPath(xpRoute);
////////////////////////////////////////////////////////////////////
...
```

The following actions occur in the Route_Legacy_Request code:

1. The first portion of the code returns the amount value that is specified in the <AskedAmount/> element:

   ```
   float amount = ...
   ```

2. A Route selection is then made using the selected amount. If the value is greater than $5,000.00, the flow is dispatched to the PORT_1 label (Legacy_1_Flow). Otherwise it goes to PORT_ 2 (Legacy_2_Flow). Each legacy subflow puts the message in a unique queue to be picked up and processed by the appropriate enterprise information system:

   ```
   String direction = (amount > 5000) ? "PORT_1" : "PORT_2";
   ```

3. The code uses an XPath expression to define the `labelname` attribute in the LocalEnvironment structure. This attribute is used later by the RouteToLabel node:

   ```
   (/?Destination/?RouterList/?$DestinationData/?$@labelname)[set-value
   ($name)]
   ```

4. The XPath `$name` variable, which is defined as the `labelname` attribute value, is populated before the evaluation of the entire expression:

   ```
   xpRoute.assignVariable("name", direction);
   ```

5. The XPath expression is evaluated, and the entire message structure is created for use by the RouteToLabel node:

   ```
   lenvMessage.evaluateXPath(xpRoute);
   ```

Archived

SOAPExtractEnvelope node

Next, the SOAP envelope parts (Body and Header) are removed from the message structure and saved in the LocalEnvironment structure by using a SOAPExtract node from the IA9O SupportPac (Figure 7-48). The XML namespace for the SOAP envelope is saved in the path $LocalEnvironment/SOAP/Envelope. The namespace can be used later in the message flow to construct a SOAP envelope response to the requestor using the same namespaces that are defined on the request.

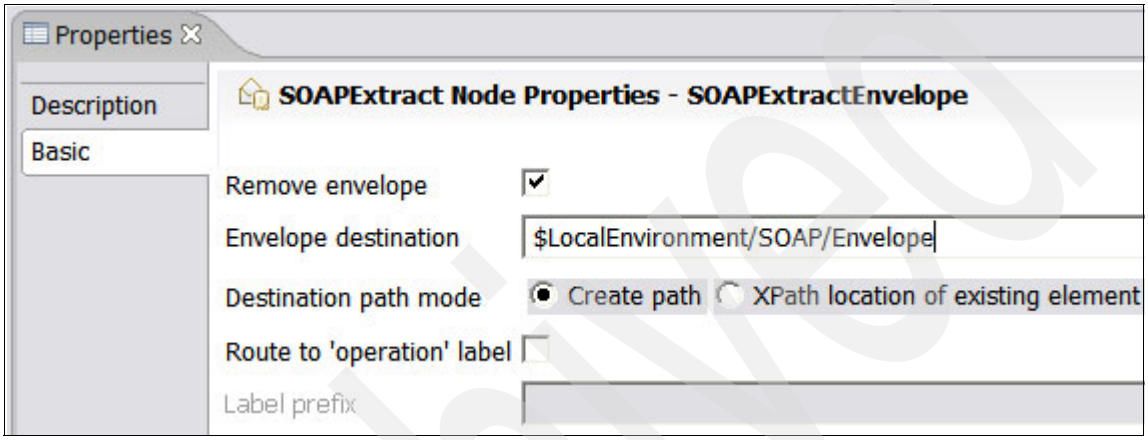


Figure 7-48 SOAPExtract node configuration

FormatLegacyRequest XMLTransformation node

An XMLTransformation node is used to transform the incoming message to the expected structure that is used by the enterprise information system. To use this node, an XSL file must be created to map the conversion from one XML file to another.

An XSL file is defined by creating an XML-to-XML mapping file (*.xmx). The incoming message is filtered and changed to a new XML plain format using an XMX file called *req2legacy.xmx* (Figure 7-49 on page 222).

Archived

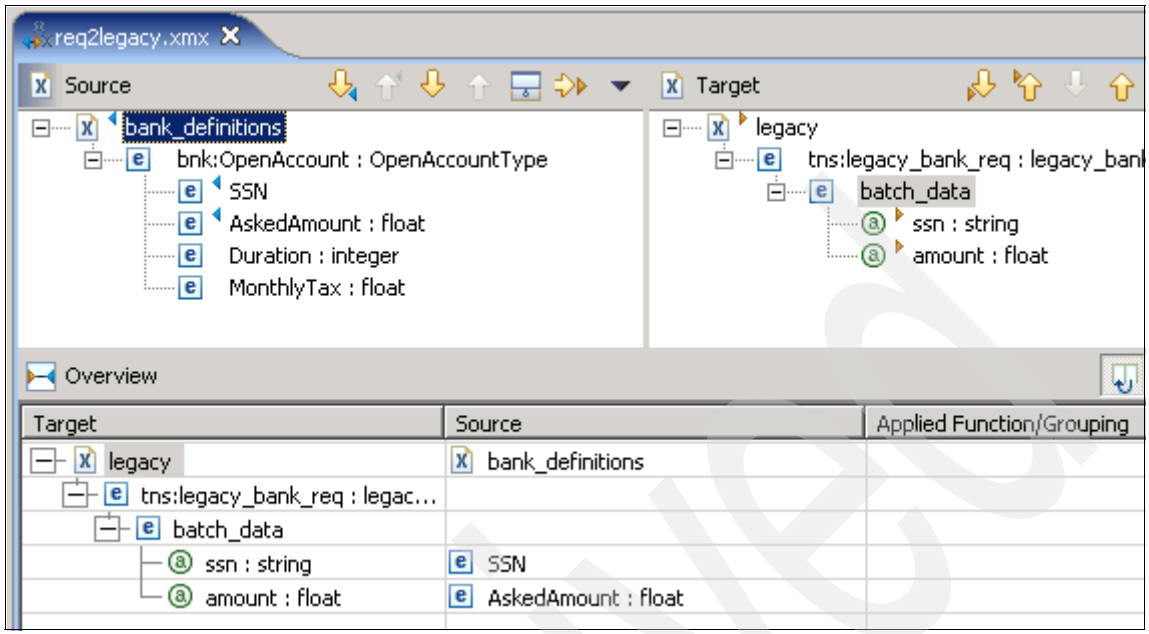


Figure 7-49 XMX legacy request

Each generated XMX file (legacy request and response) must be exported as a new XSL configuration. Each newly created transformation object (XSL) must be defined inside an XMLTransformation node (Figure 7-50).

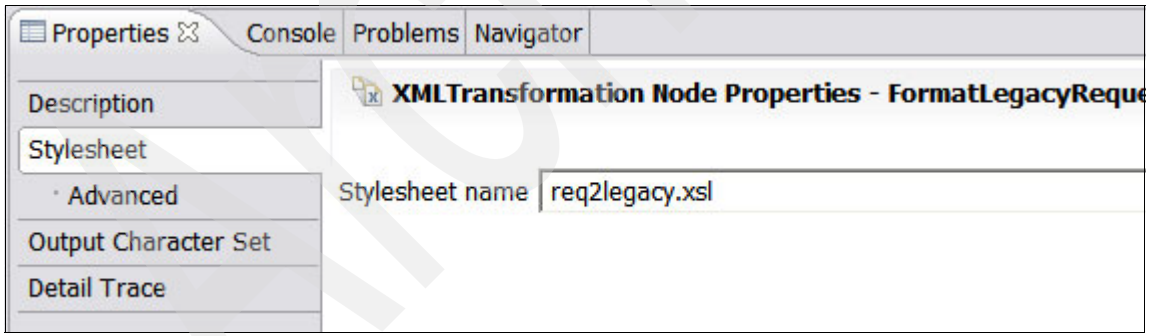


Figure 7-50 XMLTransformation node configuration

XSL file from an XMX file: To generate an XSL file from an XMX file, right-click the XMX file and select the **Generate XSLT** option.

XML2BLOB ResetContentDescriptor node

After the XML transformation is performed, the newly created message structure must be encrypted to the expected security algorithm before it is sent to the enterprise information system. The encryption node only supports binary large object (BLOB) bit streams. Therefore, the message structure must be converted to a BLOB domain using a ResetContentDescriptor node. This node is responsible for resetting the message content (MRM) to a BLOB bit stream (Figure 7-51).

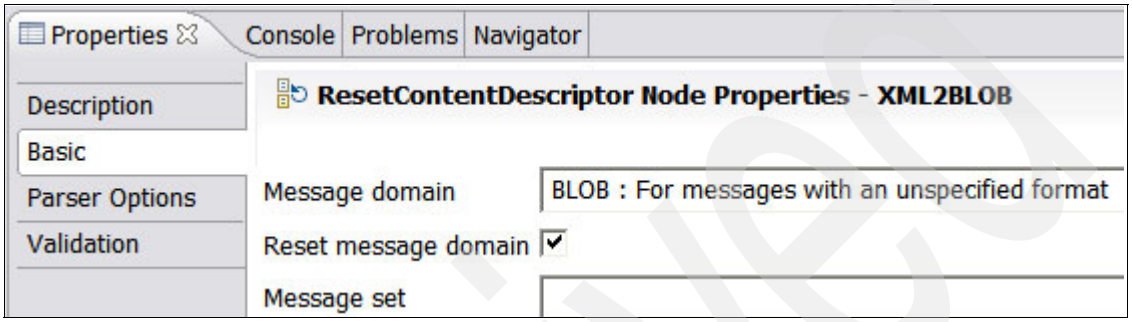


Figure 7-51 Resetting the MRM domain to a BLOB bit stream

EncryptRequest PasswordBasedEncryption node

Next, the message is encrypted using the cryptographic SupportPac (PasswordBasedEncryption and PasswordBasedDecryption nodes). A 64-bit DES algorithm is used to encrypt the bit stream (Figure 7-52), and a password is used to decrypt the response.

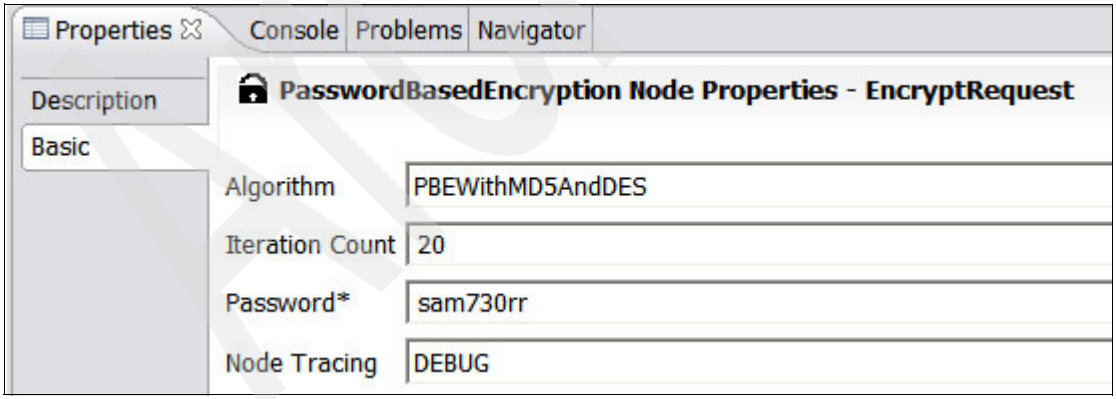


Figure 7-52 Encryption property configuration

Note: You can download the IA9W: WebSphere Message Broker - Encryption/Decryption Nodes SupportPac from the Web at:

http://www-1.ibm.com/support/docview.wss?rs=171&uid=swg24016431&loc=en_US&cs=utf-8&lang=en

MQ_Legacy_1 and MQ_Legacy_2 MQOutput nodes

The MQOutput nodes deliver the encrypted message to the correct destination.

MQ_Legacy_1 delivers the message to the SC3Q.INPUT.LEGACY_1 queue (Figure 7-53).

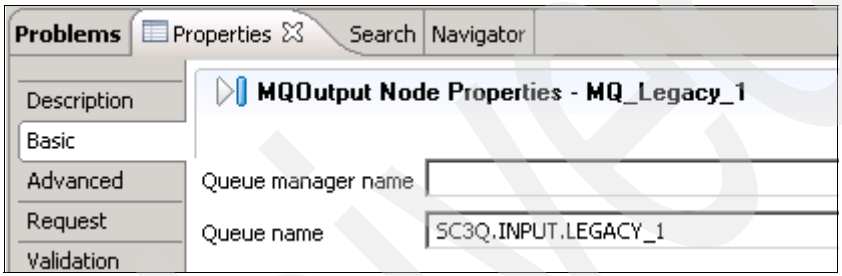


Figure 7-53 MQ_Legacy_1 MQOutput node

MQ_Legacy_2 delivers the message to the SC3Q.INPUT.LEGACY_2 queue (Figure 7-54).

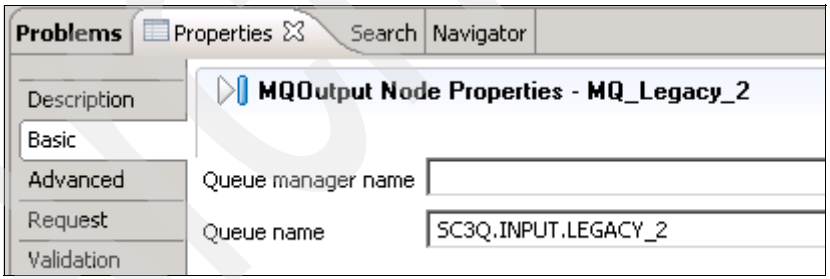


Figure 7-54 MQ_Legacy_2 MQOutput node

Archived

MQGet1, MQGet2, and Throw_2 nodes

The MQGet nodes are used to collect the response from the enterprise information system. If no response is produced in 5 seconds, the MQGet node throws an exception using the Throw node, specifying that no response was delivered on an expected time (Figure 7-55).

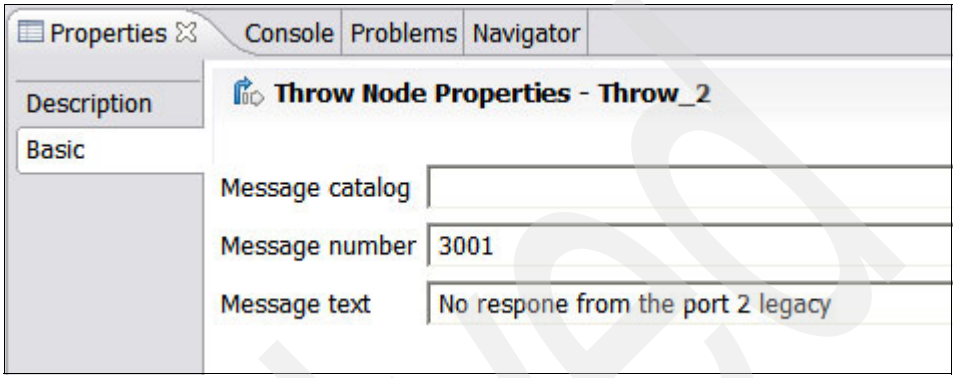


Figure 7-55 Throwing an exception with the message defined

The MQGet nodes parse the message using a BLOB domain (Figure 7-56). The applications of the enterprise information system also encrypt their response bit stream and must be decrypted by a PasswordBasedDecryption node that is defined later on the message flow.

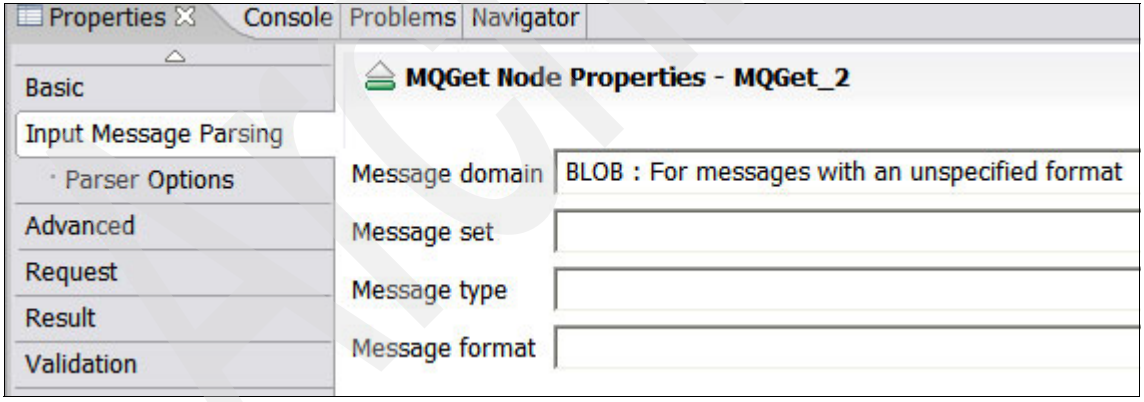


Figure 7-56 A BLOB domain must be defined inside the MQGet parser options

PasswordBasedDecryption node

After the message is received by the MQGet node, the BLOB bit stream is decrypted using a PasswordBasedDecryption node, and the secure message cycle between the runtime broker and the enterprise information system is complete (Figure 7-57).

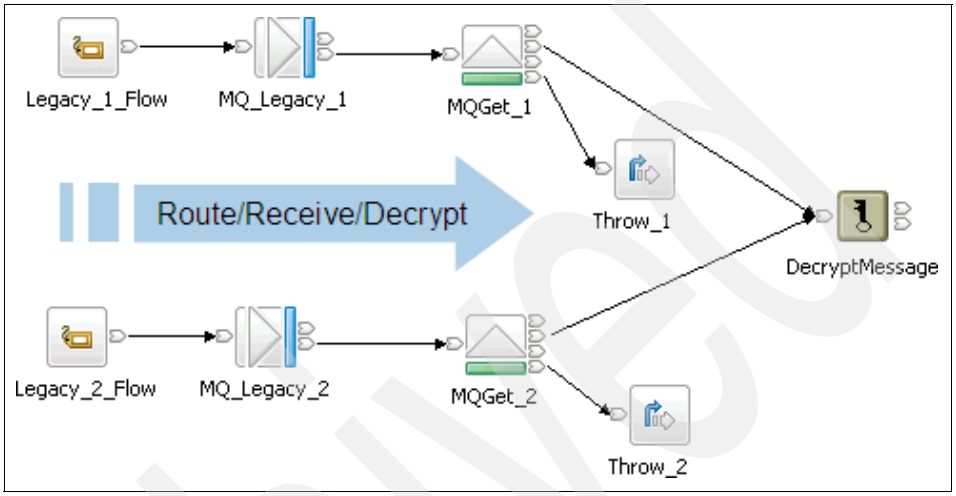


Figure 7-57 Secure message cycle

ENCRYPT2XML ResetContentDescriptor node

The PasswordBasedDecryption node decrypts and saves the new message bit stream inside a BLOB domain. As a result, a ResetContentDescriptor node is necessary to reset the message bit stream into an XMLNS structure. The XMLNS domain is used to convert the BLOB bit stream into a new XML message structure on the flow (Figure 7-58).

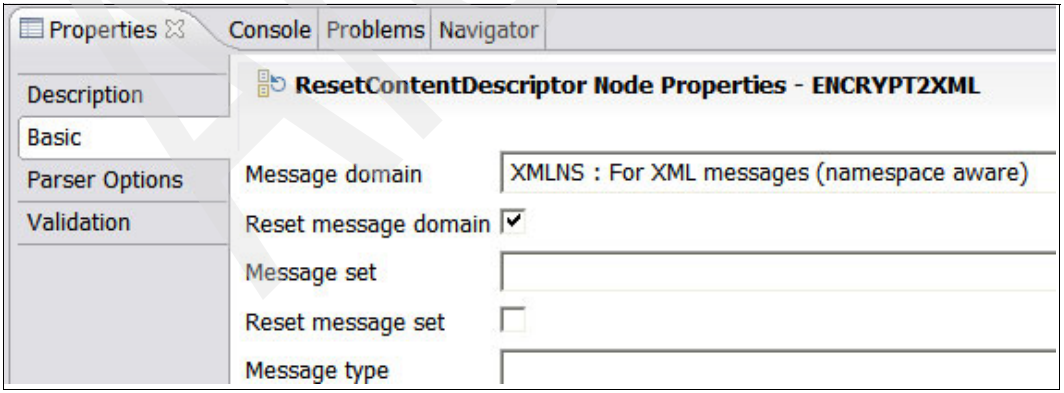


Figure 7-58 ResetContentDescriptor XMLNS configuration properties

RemoveMQMD JavaCompute node

Before the message can be returned to the requester, the generated MQMD header must be removed from the message structure. The MQMD removal is done by using a JavaCompute node, which has the following significant line:

```
BankUtils.copyHeaders(inMessage, outMessage, new String[]{"MQMD"});
```

This code copies all the headers from the InputRoot message to the OutputRoot, excluding the MQMD structure that was created by the MQOutput node.

SOAPEnvelope node

When the flow removes the MQMD header from the message tree, the SOAP Envelope structure is then added again to the message using the SOAPEnvelope node. After receiving the new SOAP envelope, the message is finally returned to the requestor, finishing the message flow.

Error with OpenAccount message flow: As with the VerifyCredit message flow, if an error occurs during the OpenAccount message flow execution, a SOAP fault message is formatted by a JavaCompute node at the beginning of the message structure.

Archived

Archived

8

Scenario: Aggregation

In the scenario presented in this chapter, we demonstrate the following concepts:

- Protocol transformation

  WebSphere Message Broker accepts a SOAP/HTTP request through a firewall, communicates with MQ clients using the MQ transport behind the firewall, and responds back to the requester using SOAP/HTTP.

- Data enrichment through aggregation

  WebSphere Message Broker aggregates responses from two sources to form a single message, which is delivered to the requester.

- Request-reply interaction pattern

  A request is sent from WebSphere Process Server to WebSphere Message Broker. A reply is expected.

- Connectivity using a SOAP/HTTP Web service binding WebSphere Process Server to WebSphere Message Broker using SOAP/HTTP

  An HTTPInput node is used in WebSphere Message Broker to receive the message from WebSphere Process Server. An HTTPReply node is used to send the response back to WebSphere Process Server.

© Copyright IBM Corp. 2008. All rights reserved.

8.1 Scenario overview

The business process of WebSphere Process Server in this scenario is responsible for processing an order. The order consists of a single item number. The business process must find suppliers to fulfill this order.

The systems that contain the supplier information are located behind a firewall and can only communicate using MQ. WebSphere Message Broker is located with these systems behind the firewall.

In order to fulfill requests for suppliers, the business process makes a SOAP/HTTP call through the firewall to WebSphere Message Broker. WebSphere Message Broker requests supplier information from two suppliers, aggregates the responses of the suppliers, and responds to the business process using SOAP/HTTP.

Aggregation: Aggregation is the generation and fan-out of related requests that are derived from a single input message. It is also the fan-in of the corresponding replies to produce a single aggregated reply message.

Figure 8-1 shows the basic flow of this scenario.

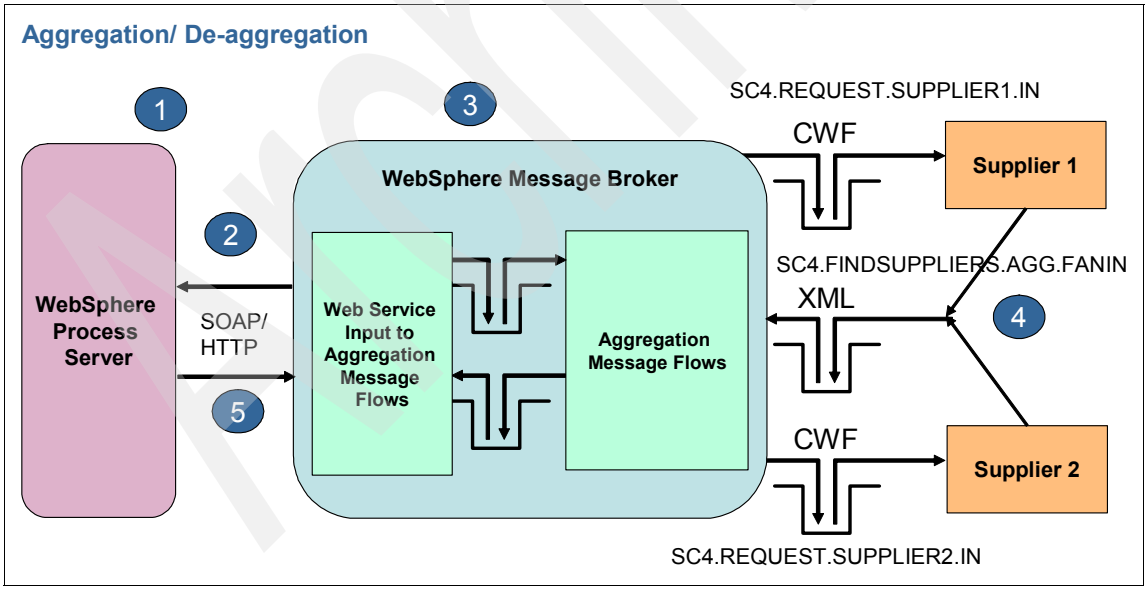


Figure 8-1 Scenario overview

We describe the highlights of this scenario, numbered in Figure 8-1 on page 230, as follows:

1. WebSphere Process Server hosts a business process that sends a message to WebSphere Message Broker using a request-response operation.
2. The message is sent over an import component with a SOAP/HTTP Web service binding.
3. A message flow in WebSphere Message Broker receives the request and converts it from HTTP to the MQ transport protocol. The message is modeled as a Custom Wire Format (CWF) using a message definition.

   The message flow requests responses from two suppliers using the MQ transport. It then aggregates the responses of the two suppliers into a single message, converts the messaging protocol back to HTTP, and responds to the HTTP requester.
4. The requests made of Supplier 1 and Supplier 2 are in a predefined CWF message format and are sent using MQ as the messaging transport. Each supplier system is emulated by a message flow, which accepts a CWF message and responds with a SOAP message that contains the corresponding supplier information that corresponds to an XML formatted message model.
5. The response is returned to WebSphere Process Server using SOAP/HTTP.

8.2 Environment

This scenario was developed in an environment with the following software:

- Operating system: Microsoft Windows XP 2002 Professional Edition with Service Pack 2
- WebSphere MQ v6.0.0.0
- WebSphere Message Broker v6003 with the fix for APAR IC51074 applied.
- WebSphere Integration Developer v6.0.2
- WebSphere Message Brokers Toolkit v6.0.2 with Interim fix 008

Archive

8.3 Queue manager implementation of WebSphere MQ

WebSphere Message Broker requires a queue manager. For this scenario, the following queues must be defined on the broker's queue manager using the default options. Keep in mind that queue names are case-sensitive.

- SC4.FINDSUPPLIERS.AGG.FANIN
- SC4.FINDSUPPLIERS.AGG.FANOUT
- SC4.FINDSUPPLIERS.WS.RESPONSE.IN
- SC4.REQUEST.SUPPLIER1.IN
- SC4.REQUEST.SUPPLIER2.IN

8.4 Business integration module

In this section, we focus on the implementation of the business integration module. The process receives a request that contains an item number as a string. It sends the message to a WebSphere Message Broker message flow. The message flow sends the message to multiple suppliers to see if the item is in stock. The message flow compiles a list of supplier responses and sends the list back to the requestor.

8.4.1 Business integration module components

The business process is implemented in a business integration module called *ProcessOrder*. Figure 8-2 illustrates the assembly diagram for the ProcessOrder module.

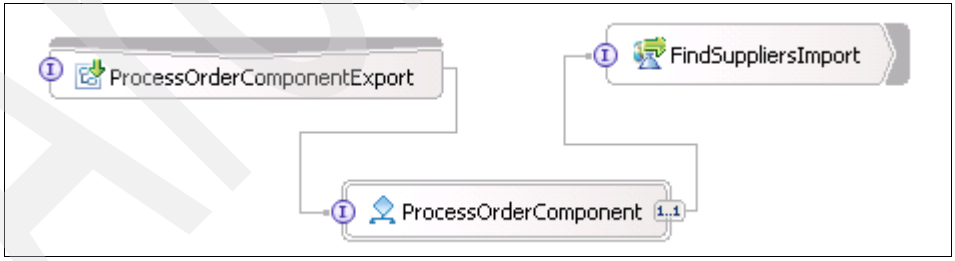


Figure 8-2 ProcessOrder module assembly diagram

Archived

The ProcessOrder module includes the following components:

- *ProcessOrderComponentExport* is the export component that the client uses to access the business process.
- *ProcessOrderComponent* contains the business process implementation.
- *FindSuppliersImport* is the import component that sends messages to a service via WebSphere Message Broker.

All three components use the FindSuppliersInterface interface.

8.4.2 Business objects

The module uses two business objects:

- The Supplier business object is used to model the information that pertains to a supplier.
- The Suppliers business object is a container to model the information that pertains to a list of Supplier business objects.

Item: The request that is received by the business process is the *item*. The item is a string and not modeled as a business object.

Archived

Figure 8-3 shows the Supplier business object, which consists of string attributes and uses the namespace `http://ProcessOrder`.

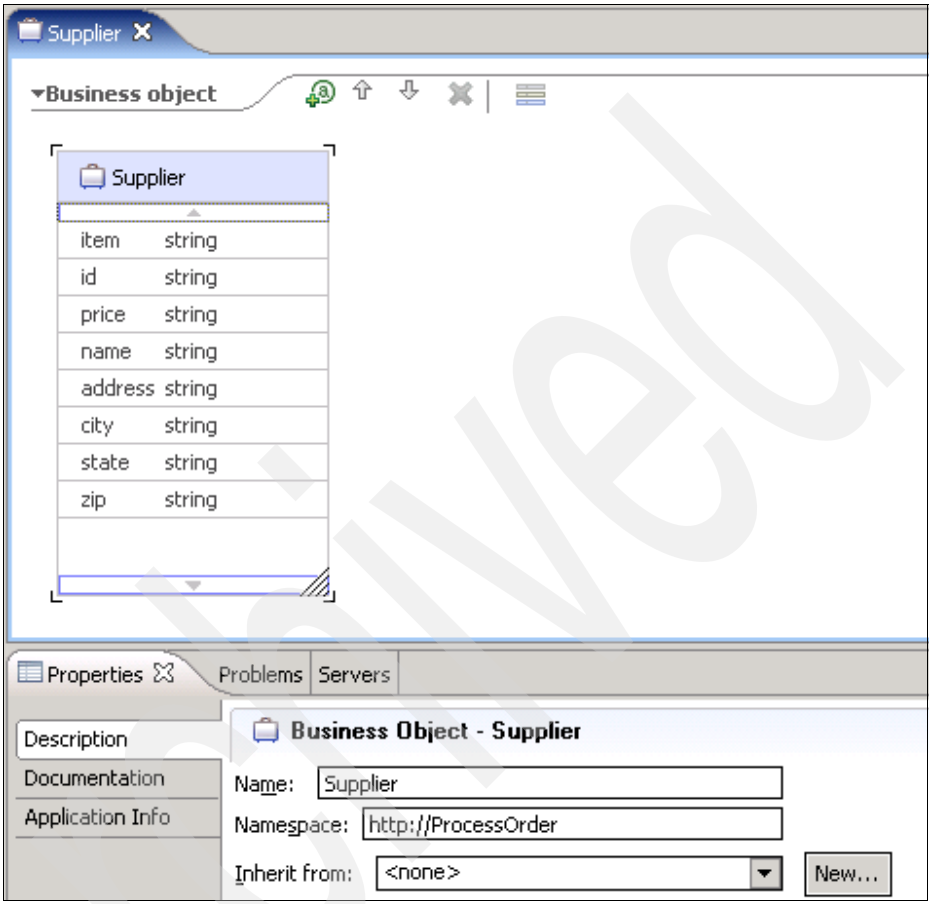


Figure 8-3 The Supplier business object

Archived

Figure 8-4 shows the Suppliers business object. This object is simply a container for the list of Supplier business objects that are expected in the response. The namespace for the Suppliers business object is the same as the Supplier business object.

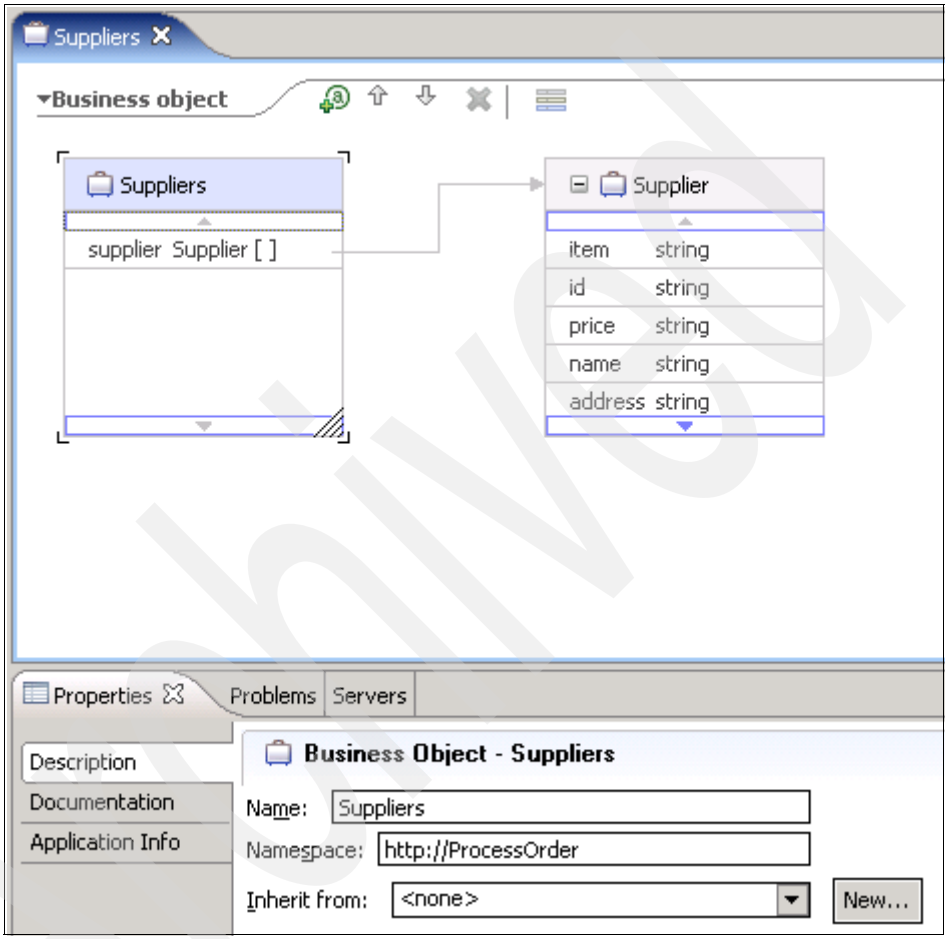


Figure 8-4 The Suppliers business object

Archived

8.4.3 Interfaces

The ProcessOrder module uses one interface, FindSuppliersInterface, which is shown in Figure 8-5. This interface has one request-response operation that sends the string, which is `item`, and receives a Suppliers business object in response.

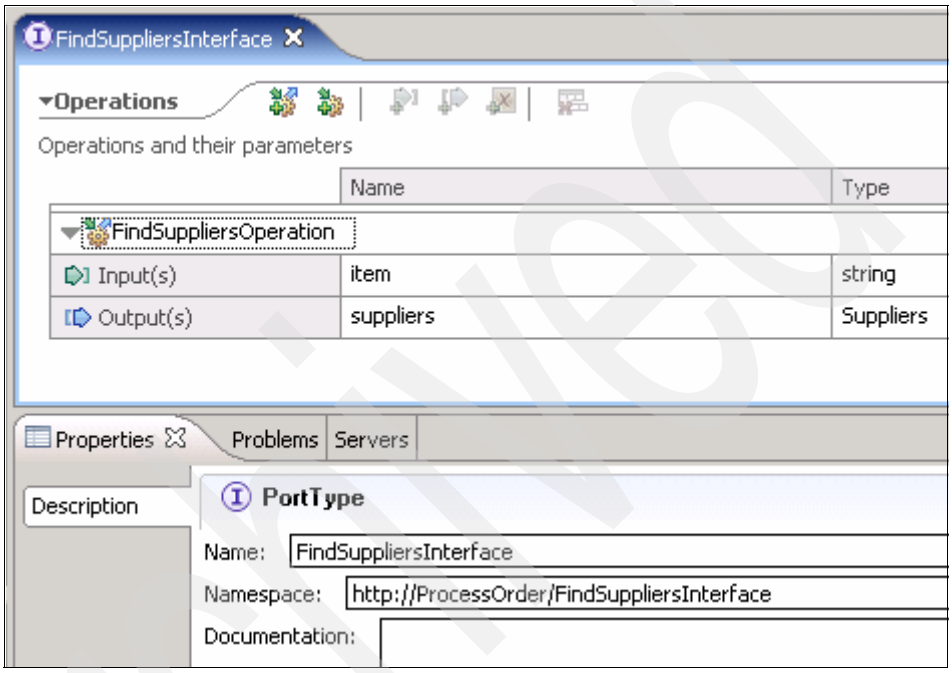


Figure 8-5 FindSuppliersInterface

The FindSuppliersInterface uses the namespace `http://ProcessOrder/FindSuppliersInterface`.

8.4.4 Business process

In this section, we describe the business process of the ProcessOrderComponent. The ProcessOrderComponent business process (Figure 8-6 on page 237) receives a Web service request from the client and sends a message to a message flow in WebSphere Message Broker. The process waits for a response and then responds to the requester.

The ProcessOrderComponent properties were not configured beyond the default configuration.

Archived

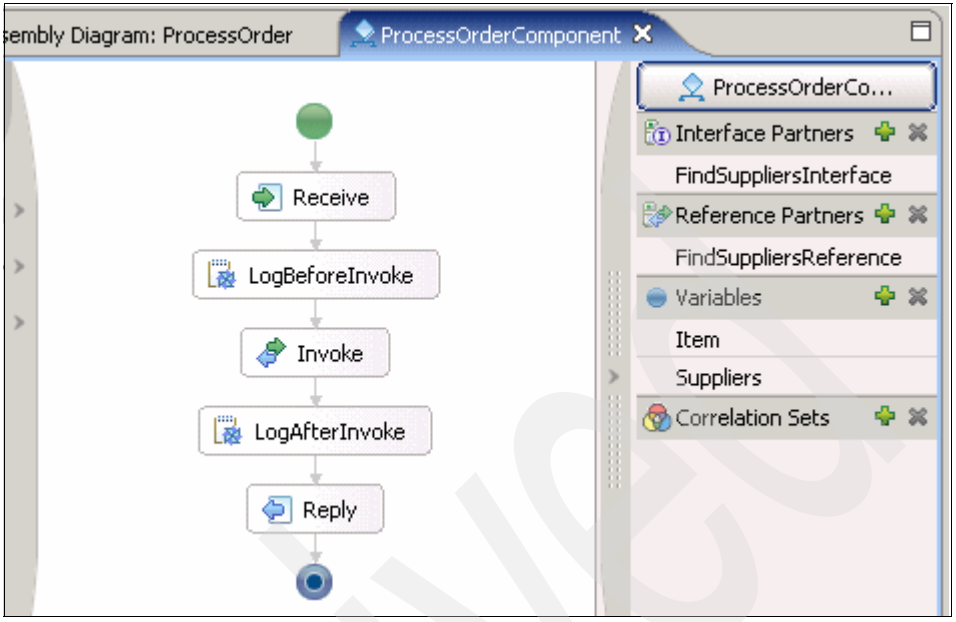


Figure 8-6 ProcessOrderComponent process

Interface and partner references

The ProcessOrderComponent process has one interface partner and one reference partner. You can see these listed to the right of the business process in Figure 8-6.

The FindSuppliersInterface interface partner (Figure 8-7) is the process interface. It exposes operations that can be called by external partners.

Figure 8-7 Business process interface partner

Archive

Reference partners specify the interface that is used in the invocation of another component. FindSuppliersReference (Figure 8-8) is defined as a reference partner to the FindSuppliersImport component. It specifies that FindSuppliersInterface is to be used as the interface.

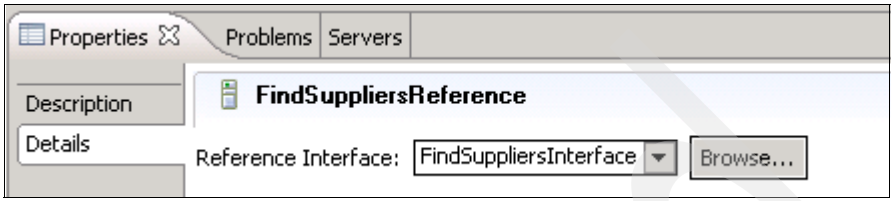


Figure 8-8 FindSuppliersReference

Receive activity

The Receive activity (Figure 8-9) receives the input to the business process from a client. It is associated with FindSuppliersInterface.

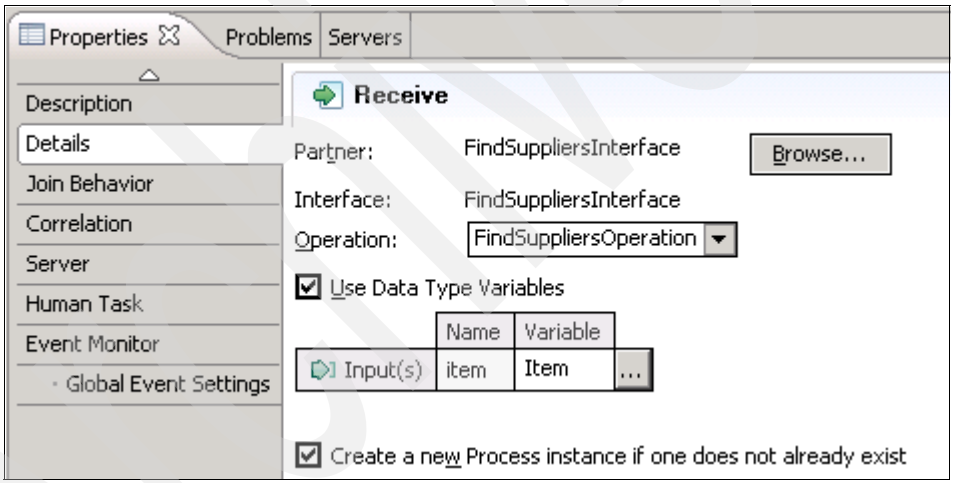


Figure 8-9 Receive activity

Invoke activity

The Invoke activity sends a message to the message flow. The Invoke activity communicates with the message flow as defined in the FindSuppliersReference partner definition. The FindSuppliersReference partner uses the FindSuppliersInterface to define that operation that will be used and how the data will be formatted. The FindSuppliersOperation of the FindSuppliersInterface is used. The FindSuppliersOperation is a simple request-response operation. The data formats, input and output, are displayed in Figure 8-10 on page 239.

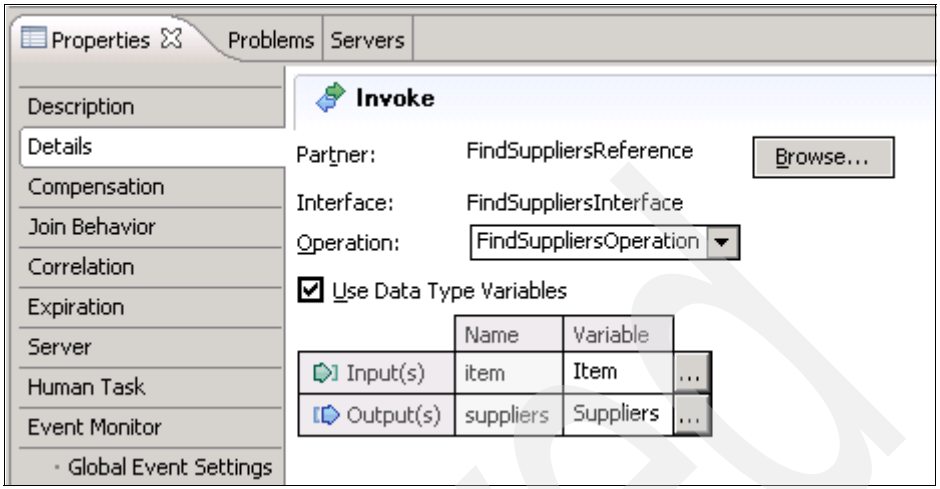


Figure 8-10 Invoke activity

Reply activity

The Reply activity (Figure 8-11) returns the response to the business process client. The business process is connected to the import and the export component in the assembly diagram.

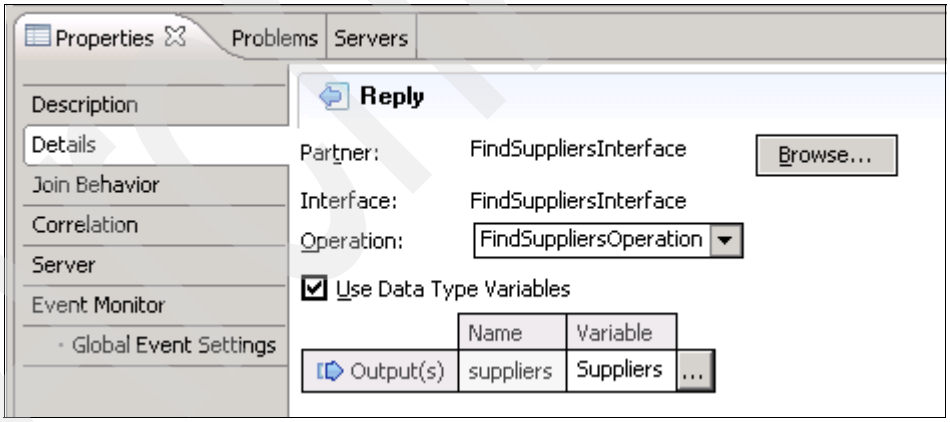


Figure 8-11 Reply activity

Archived

8.4.5 Import component and binding

The business process uses the FindSuppliersImport import component to send messages to the message flow in WebSphere Message Broker. The import component has the FindSuppliersInterface interface.

The import has a Web service binding. Figure 8-12 shows the binding properties.

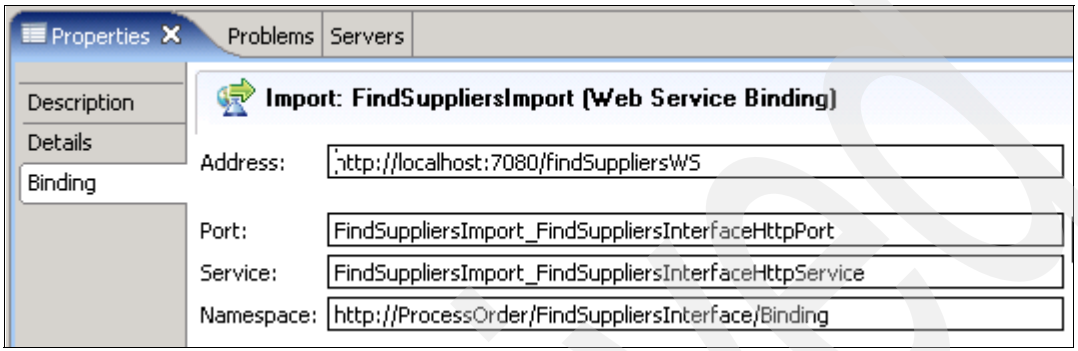


Figure 8-12 The FindSuppliersImport Web service binding properties

The Address field corresponds to the HTTPInput node in the message flow (Figure 8-18 on page 245). The HTTPInput node defines the URL path suffix, `/findSuppliersWS`, for the URL in its basic properties. The port corresponds to the port listened to by the broker's HTTP listener. Port 7080 is the default.

8.5 Message flows

In this scenario, we include six message flows and one message set. We describe how aggregation is implemented using message flows. Then we describe the message set and message flows that are used in this scenario.

8.5.1 Aggregation implementation overview

WebSphere Message Broker provides three message flow nodes that support aggregation:

- AggregateControl node
- AggregateRequest node
- AggregateReply node

When you include these nodes in your message flows, the multiple fan-out requests are issued in parallel from within a message flow. The standard

operation of the message flow is for each node to perform its processing in sequence.

You can also use these aggregation nodes to issue requests to applications outside the broker environment. Messages can be sent asynchronously to external applications or services. The responses are retrieved from those applications and are combined to provide a single response to the original request message.

These nodes can help to improve response time, because slow requests can be performed in parallel, and they do not need to follow each other sequentially. If the subtasks can be processed independently, and do not need to be handled as part of a single unit of work, they can be processed by separate message flows.

The message flow in Figure 8-13 demonstrates a fan-out flow that takes the incoming request message, generates four unique request messages, sends each message to a specific destination using a request-reply interaction pattern, and starts the tracking of the aggregation operation.

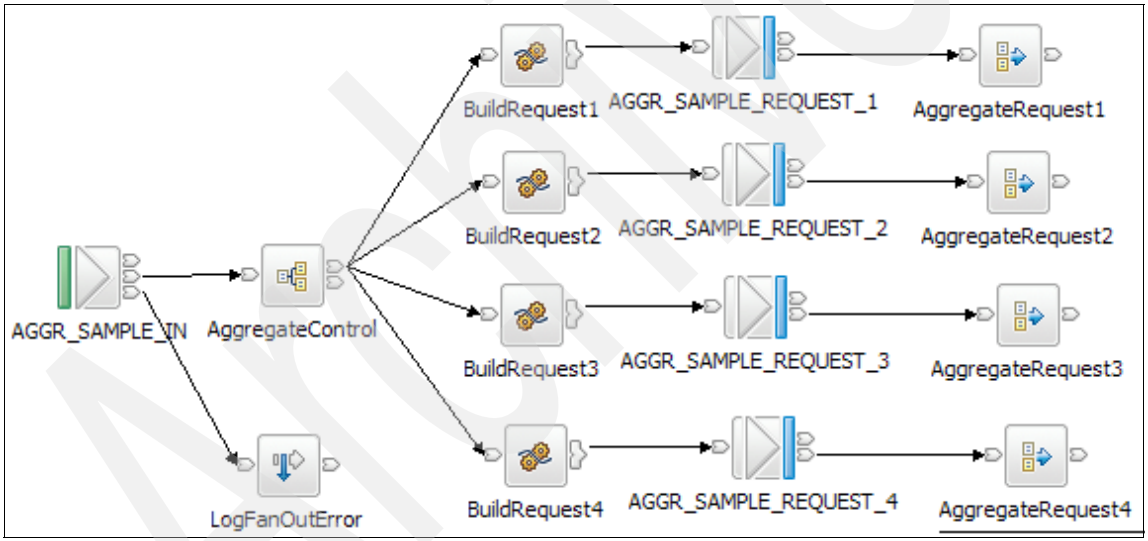


Figure 8-13 A fan-out flow

Archive

This fan-out flow is followed by a fan-in flow (Figure 8-14 on page 242) that gathers the responses from the service providers and aggregates them together into a single output message. The output message from the AggregateReply node cannot be output by an MQOutput node. Therefore, a Compute node is added to tweak the data into a format where it can be written out to a queue.

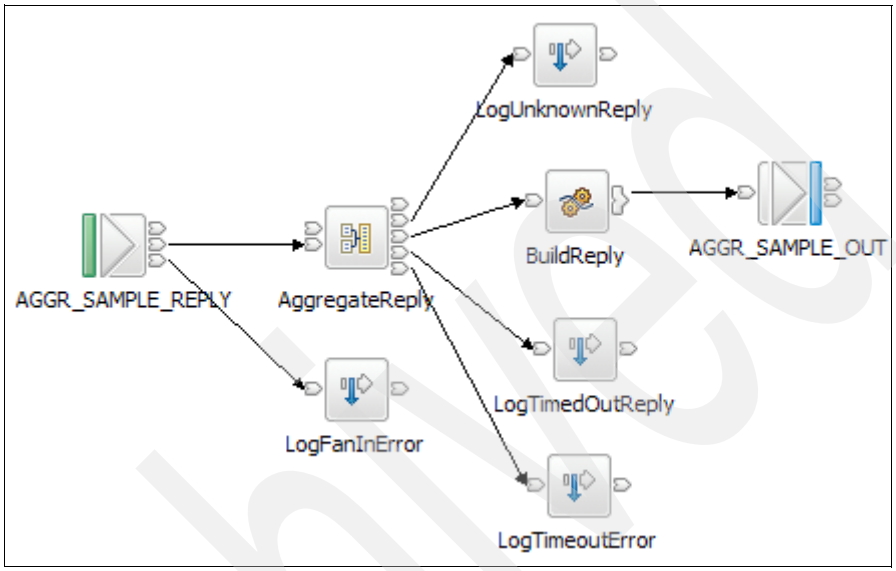


Figure 8-14 A fan-in flow

Archived

8.5.2 FindSuppliersMSetprj message set

The FindSuppliersMSetPrj message set (Figure 8-15) contains the message definitions that model the messages that are involved in the flow. The message definitions were generated from the Web Services Description Language (WSDL) file that describes the FindSuppliersInterface interface (Figure 8-5 on page 236).

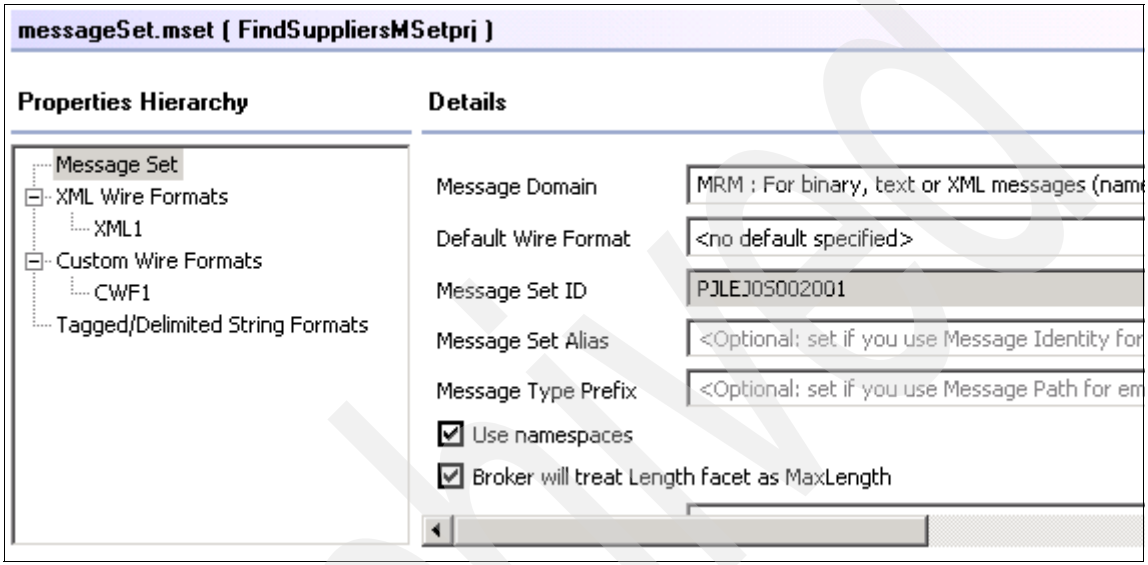


Figure 8-15 FindSuppliersMSetprj message set

The message set is identified by the message set ID. Any message to be modeled by a message definition within this message set must have the Message Set ID set in the Properties folder of the message tree. MRM is the message domain for this message set. The message set supports XML and custom wire formats.

Archived

Message definitions

The FindSuppliersInterface_InlineSchema1.mxsd message definition file (Figure 8-16) shows the structure of how the message will be modeled. Each of the two messages, FindSuppliersOperation and FindSuppliersOperationResponse, can be modeled in the CWF or XML format.

FindSuppliersInterface_InlineSchema1.mxsd

Structure	Type	Min Occurs	Max Occurs
FindSuppliersInterface_InlineSchema1.mxsd			
Messages			
FindSuppliersOperation			
{Local complexType}			
item	xsd:string	1	1
FindSuppliersOperationResponse			
{Local complexType}			
suppliers	bons1:Suppliers	1	1
supplier	bons1:Supplier	0	-1
item	xsd:string	0	1
id	xsd:string	0	1
price	xsd:string	0	1
name	xsd:string	0	1
address	xsd:string	0	1
city	xsd:string	0	1
state	xsd:string	0	1
zip	xsd:string	0	1

Figure 8-16 The FindSuppliersInterface_InlineSchema1.mxsd

8.5.3 FindSuppliersWSInput message flow

Figure 8-17 shows the FindSuppliersWSInput message flow.

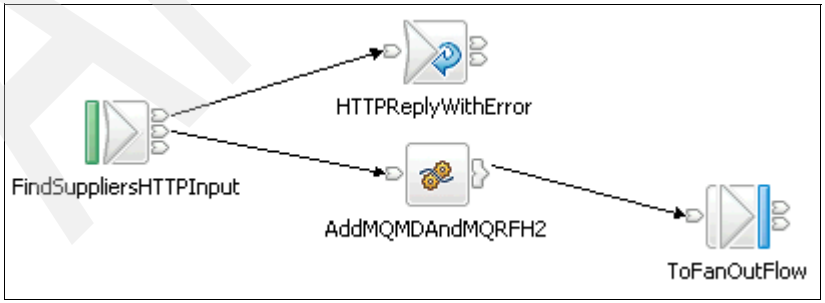


Figure 8-17 FindSuppliersWSInput.msgflow

The message flow receives the request from WebSphere Process Server using SOAP/HTTP. The message is transformed to an MQ format and validated against the WebSphere Message Broker message definition in the message set.

The flow changes the messaging transport from HTTP to MQ and submits the request to the FindSuppliersAggFanIn flow.

FindSuppliersHTTPInput node

The basic properties of the FindSuppliersHTTPInput node (Figure 8-18) define the URL selector in the node, which in this case is findSuppliersWS.

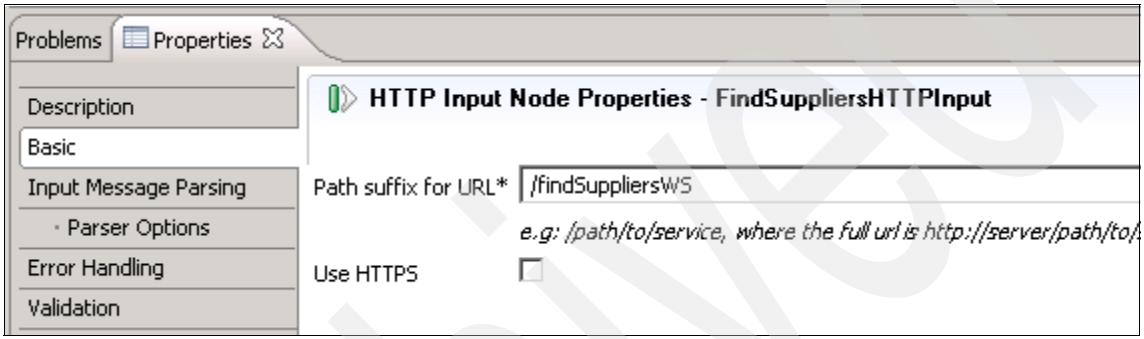


Figure 8-18 Basic properties of the FindSuppliersHTTPInput node

The input message parsing properties (Figure 8-19) define how the message is modeled. The Message set field contains the ID for the message set that is defined in Figure 8-17 on page 244.

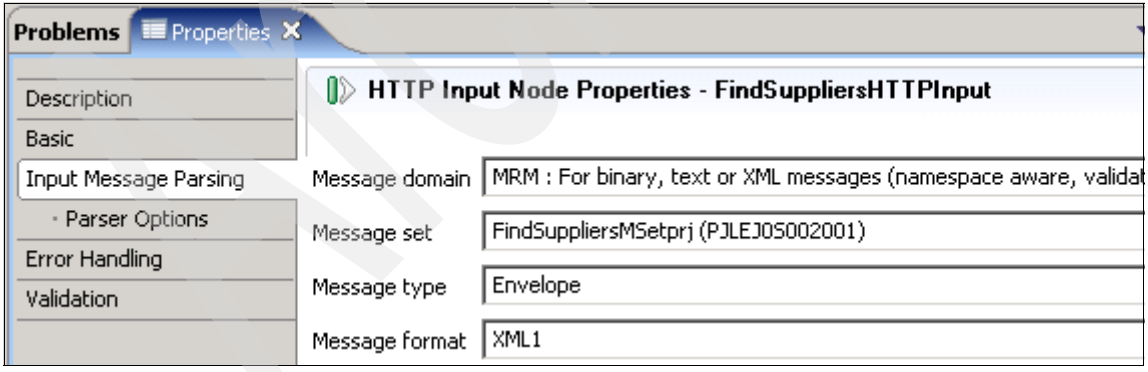


Figure 8-19 Input message parsing properties of the FindSuppliersHTTPInput node

The validation properties (Figure 8-20) indicate that validation is to be done for both content and value. The message will be validated against the Envelope message type in the FindSuppliersMSetprj message set as it is parsed.

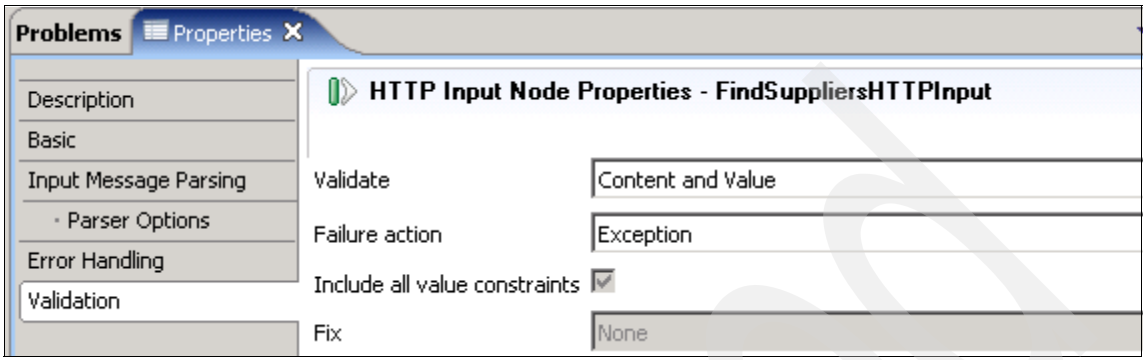


Figure 8-20 Validation properties of the FindSuppliersHTTPInput node

If the validation fails, a SOAP exception is appended to the message and sent to the failure terminal of this node. The failure terminal is connected to the HTTPReplyWithError node, which relays the exception to the HTTP requester.

If the validation is successful, the message is passed to the AddMQMDAndMQRFH2 node.

HTTPReplyWithError node

The HTTPReplyWithError node is an HTTPReply node that responds back to the HTTP requester should any failure occur within this flow. This node is not configured beyond its default values.

AddMQMDAndMQRFH2 Compute node

The AddMQMDAndMQRFH2 node is a Compute node that removes the HTTP header and adds an MQMD header since the message will be delivered to MQ for aggregation. The HTTPRequestIdentifier is stored in the usr folder of the MQRFH2 header for the duration of the aggregation. This identifier allows the broker to respond to the correct HTTP requester.

Archived

The basic properties of the node (Figure 8-21) define the location of the extended SQL (ESQL) module used to implement the logic. No other properties are configured beyond default values.

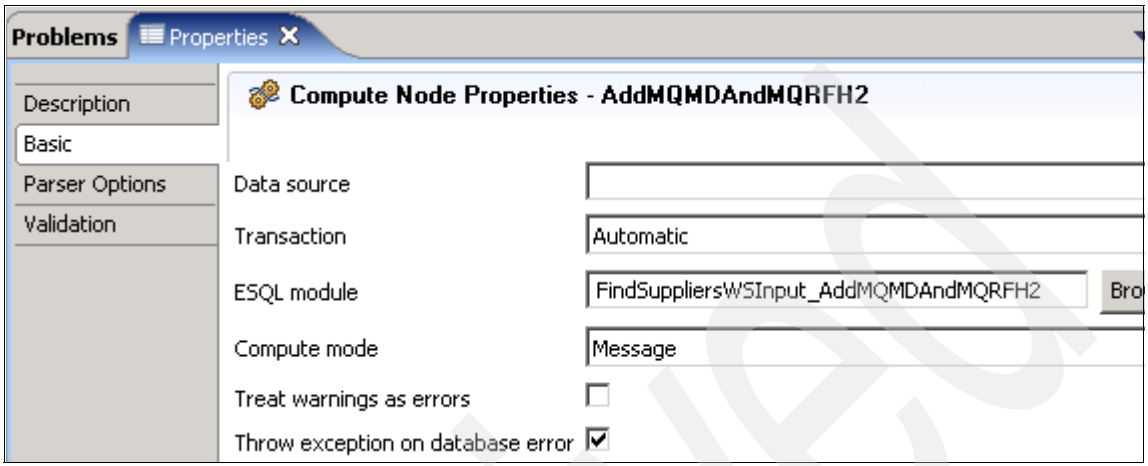


Figure 8-21 AddMQMDAndMQRFH2 basic properties

The ESQL, shown in Example 8-1, facilitates the change in messaging transport from HTTP to MQ. The MQRFH2 header is added to store the HTTPRequestIdentifier of the HTTP requester. The MQRFH2 header is maintained throughout the aggregation request, so that the response can be sent to the correct requester.

Example 8-1 ESQL for the AddMQMDAndMQRFH2 node

```
CREATE COMPUTE MODULE FindSuppliersWSInput_AddMQMDAndMQRFH2
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN

      -- First, we need the majority of this message and so
      -- we will copy the input message to the output message
      CALL CopyEntireMessage();

      -- Next, we remove HTTP headers and add the MQMD since the
      -- output is MQ
      SET OutputRoot.HTTPInputHeader = null;
      CREATE NEXTSIBLING OF OutputRoot.Properties DOMAIN ('MQMD');

      -- We need to keep track of the HTTPRequestIdentifier so we will
      -- know where to send the response
      -- To keep track of HTTPRequestIdentifier, we place it in MQRFH2's
      -- usr folder
```

Archived

```
      -- usr.HTTPRequestIdentifier will remain untouched for the duration
      -- of the aggregation and will be used in the FindSupplierWSOut flow
      -- to send the HTTP response back to the corresponding requester
      CREATE NEXTSIBLING OF OutputRoot.MQMD DOMAIN ('MQRFH2');
      SET OutputRoot.MQRFH2.usr.HTTPRequestIdentifier =
      CAST (InputLocalEnvironment.Destination.HTTP.RequestIdentifier AS CHARACTER);
      RETURN TRUE;
   END;

   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

ToFanOutFlow MQOutput node

The ToFanOutFlow MQOutput node creates a message ID for the newly created MQ message and puts the message on the SC4.FINDSUPPLIERS.AGG.FANOUT queue. The basic properties of the node (Figure 8-22) define to which queue the message is delivered. The queue is the input queue for the FindSuppliersAggFanOut.msgflow.

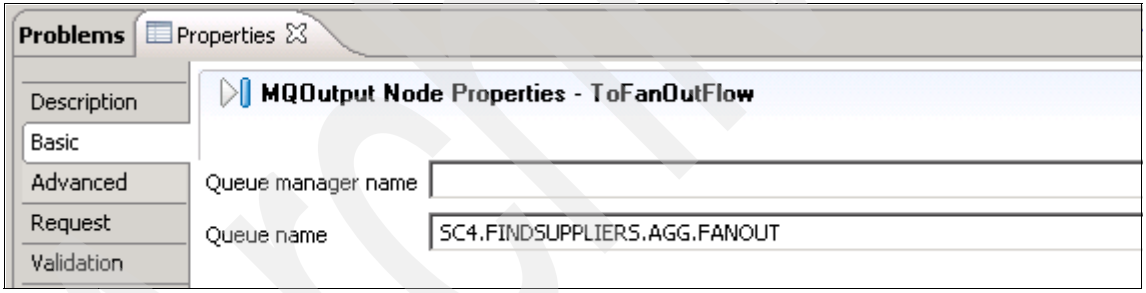


Figure 8-22 ToFanOutFlow basic properties

Archived

The advanced properties (Figure 8-23) of the instruct the node to create a new message ID for the message and to pass only the default message context.

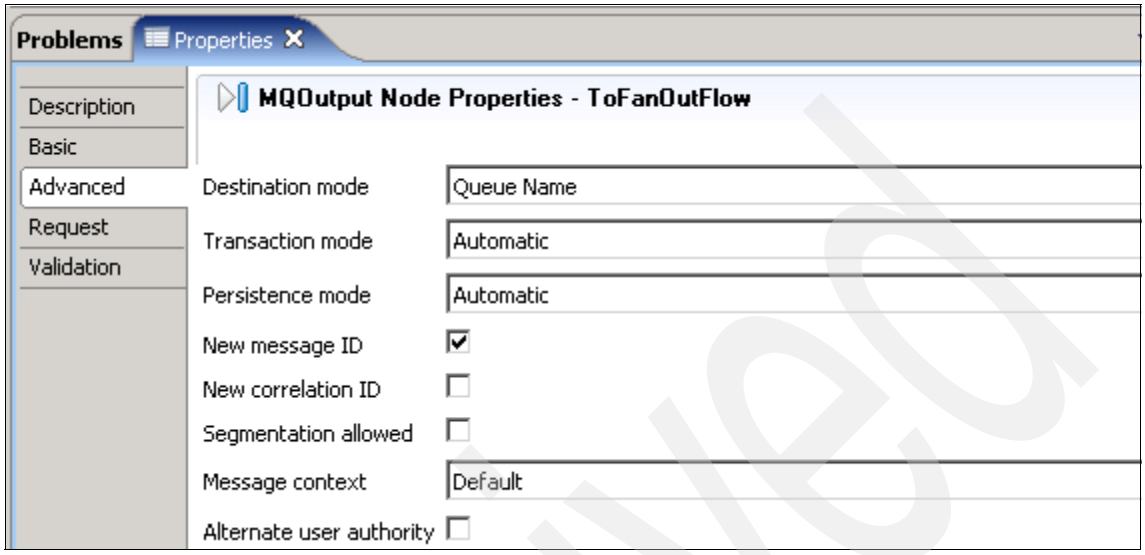


Figure 8-23 ToFanOutFlow advanced properties

8.5.4 FindSuppliersAggFanOut message flow (aggregation fan-out)

The FindSuppliersAggFanOut.msgflow (Figure 8-24) is the aggregation fan-out flow. This flow takes the output from the FindSuppliersWSInput.msgflow and performs the fan-out of the message to multiple suppliers.

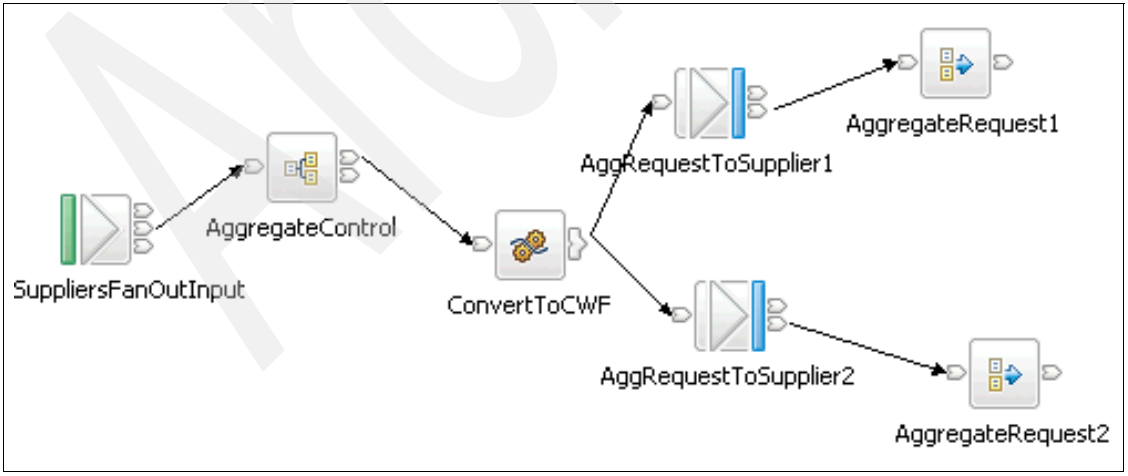


Figure 8-24 FindSuppliersAggFanOut message flow

Archived

The AggregateControl node marks the beginning of the aggregation. A Compute node converts the message from SOAP to CWF. Next, MQOutput nodes place the request message on multiple queues, one for each supplier. The new requests all specify the input queue for the aggregation response flow (FindSuppliersAggFanIn message flow) as the response queue. The last node in each path is an AggregateRequest node that stores information about the aggregate request.

FindSuppliersFanOutInput node

The FindSuppliersFanOutInput MQInput node receives the request from the FindSuppliersWSInput.msgflow message flow. The basic properties (Figure 8-25) define the queue that holds the input messages for this flow. This queue is defined on the ToFanOutFlow node (Figure 8-22 on page 248).

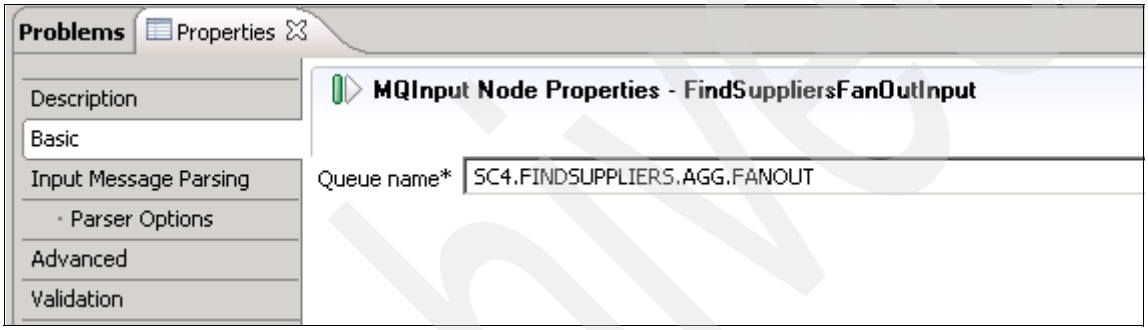


Figure 8-25 Basic properties of the FindSuppliersFanOutInput node

The input message parsing properties (Figure 8-26) define how the message is modeled. No other properties are configured beyond default values. The message set points to the FindSuppliersMSetprj message set (Figure 8-15 on page 243).

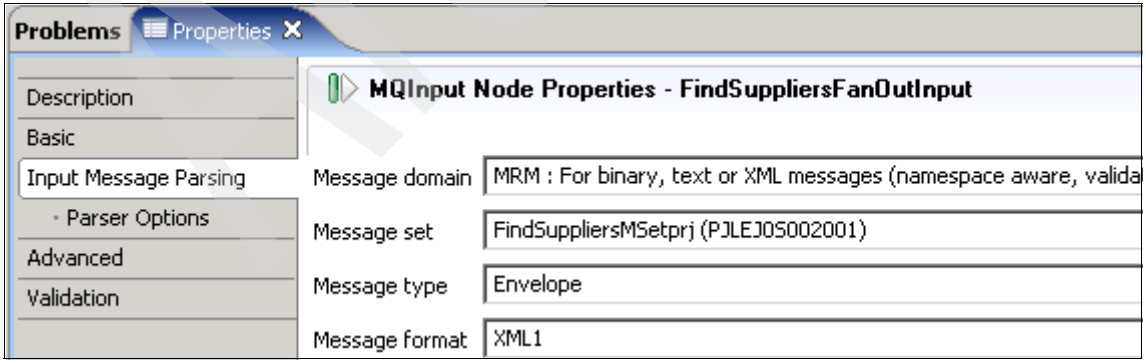


Figure 8-26 Input message parsing properties of the FindSuppliersFanOutInput node

AggregateControl node

The AggregateControl node marks the beginning of the aggregation request. The node is configured by setting an aggregate name to be associated with the aggregate requests. The basic properties (Figure 8-27) define the aggregate name to be assigned to requests that come through this aggregate control node.

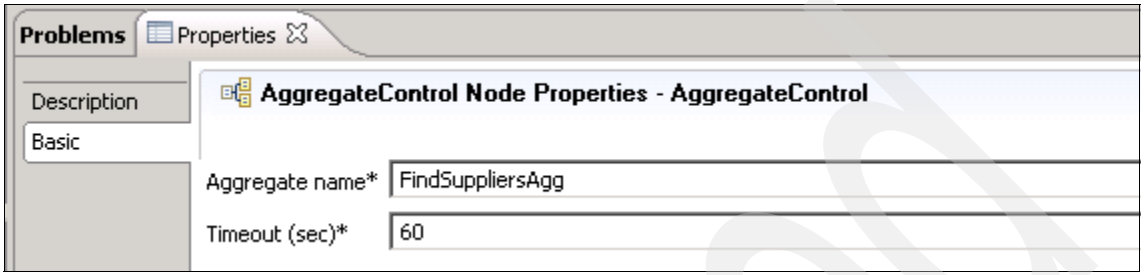


Figure 8-27 Basic properties of the AggregateControl node

ConvertToCWF Compute node

The ConvertToCWF Compute node converts the message from SOAP to CWF, so that the format of the message is understood by the supplier systems. After the message is converted, the node sends a message to each supplier input queue via the corresponding MQOutput nodes. The basic properties (Figure 8-28) define the location of the ESQL module that is used to implement the logic of this node.

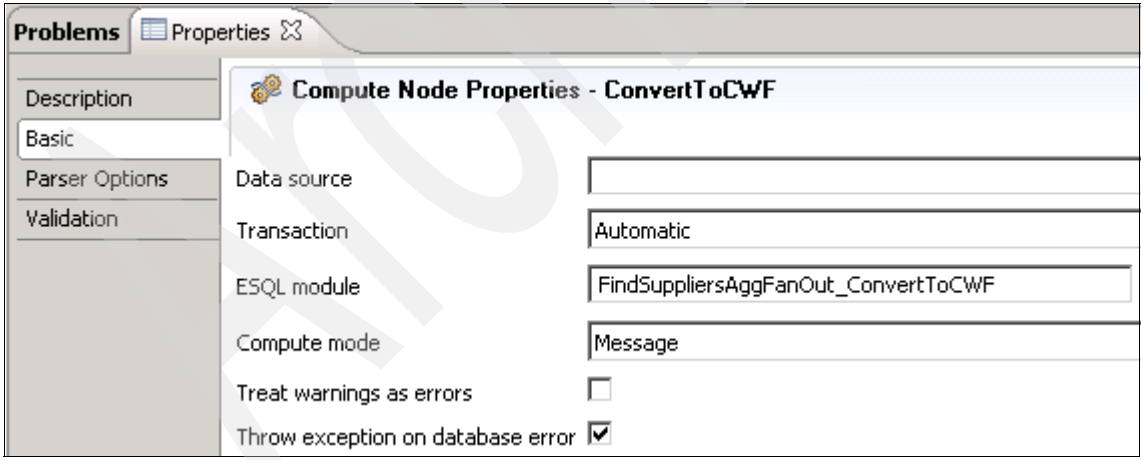


Figure 8-28 Basic properties of the ConvertToCWF node

Example 8-2 shows the ESQL for the node.

Example 8-2 ESQL code for the ConvertToCWF node

```
CREATE COMPUTE MODULE FindSuppliersAggFanOut_ConvertToCWF
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN

      -- First, we copy the message headers
      CALL CopyMessageHeaders();

      -- Next, we create MRM to store the request
      CREATE NEXTSIBLING OF OutputRoot.MQRFH2 DOMAIN 'MRM';

      -- Next, we define the properties of this message
      -- Note, the output message is CWF and so we set the properties
      -- according to the CWF format of our message set
      SET OutputRoot.Properties.MessageDomain = 'MRM';
      SET OutputRoot.Properties.MessageSet = 'PJLEJOSO02001';
      SET OutputRoot.Properties.MessageType = 'FindSuppliersOperation';
      SET OutputRoot.Properties.MessageFormat = 'CWF1';

      -- Next, we populate the MRM CWF item of the output root with the
      -- value of the input root's MRM Soap item
      SET OutputRoot.MRM.item =
InputRoot.MRM.ns:Body.null2:FindSuppliersOperation.item;
      RETURN TRUE;
   END;

   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;

END MODULE;
```

Archived

AggRequestToSupplier1 MQOutput node

The AggRequestToSupplier1 MQOutput node is used to output the request message to the supplier 1 queue. The basic properties (Figure 8-29) define the queue to use to send the request to Supplier1.

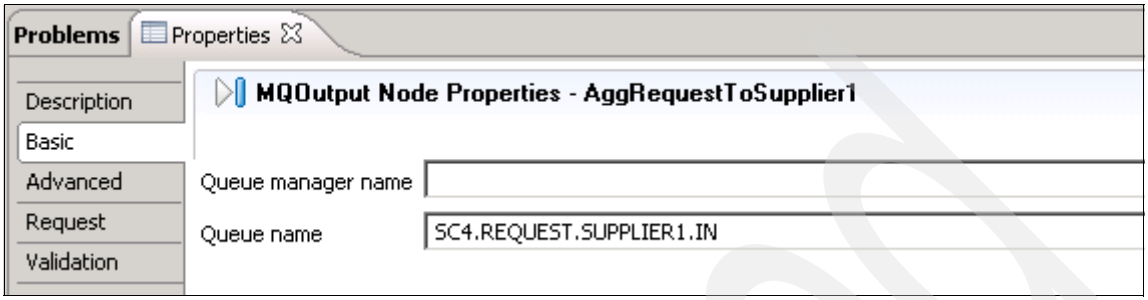


Figure 8-29 Basic properties of the AggRequestToSupplier1 node

The advanced properties (Figure 8-30) are configured to set a new message ID for this message while specifying that only the default message context is passed.

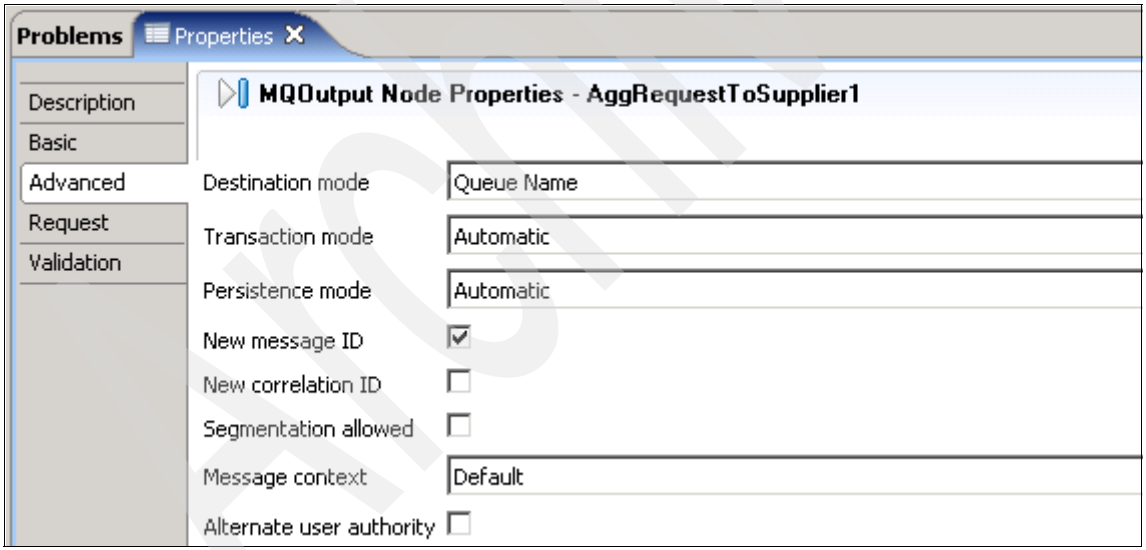


Figure 8-30 Advanced properties of the AggRequestToSupplier1 node

Archived

The request properties (Figure 8-31) mark this message as a request message requiring a response. The reply-to queue for the response is the input queue for the FindSuppliersAggFanIn.msgflow message flow, which aggregates the responses from the two suppliers.

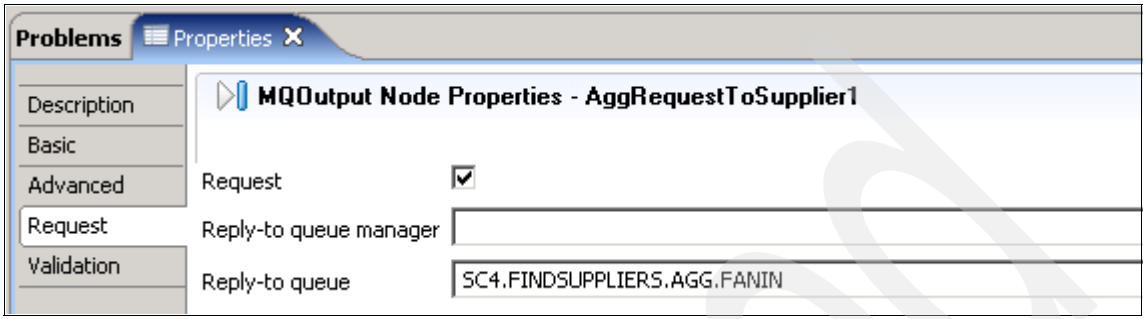


Figure 8-31 Request properties of the AggRequestToSupplier1 node

AggRequestToSupplier2 node

The AggRequestToSupplier2 MQOutput node is identical to the AggRequestToSupplier1 node with the exception of the queue name. A new message ID is set and the reply-to queue is also set to SC4.FINDSUPPLIERS.AGG.FANIN. Figure 8-32 shows the basic properties.

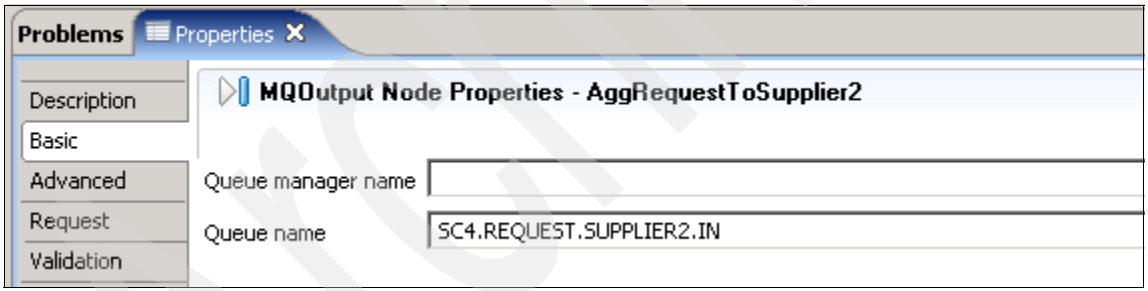


Figure 8-32 Basic properties of the AggRequestToSupplier2 node

AggregateRequest1 node

The AggregateRequest node serves the following purposes:

- It stores internal information regarding the request that was sent to the output queue.
- It assigns the response to this message to a particular folder within the message tree of the aggregated message in the FindSuppliersAggFanIn.msgflow. This helps the developer identify how to distinguish the response to this aggregate request from others by storing the response message in the named folder within the message tree.

Figure 8-33 shows the basic properties of the AggregateRequest1 node.

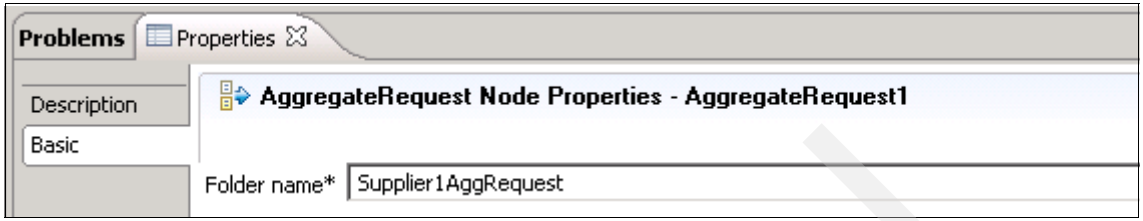


Figure 8-33 Basic properties of the AggregateRequest1 node

AggregateRequest2 node

The AggregateRequest2 node is similar to the AggregateRequest1 node, but specifies a different folder for the response. Figure 8-34 shows the properties of the AggregateRequest2 node.

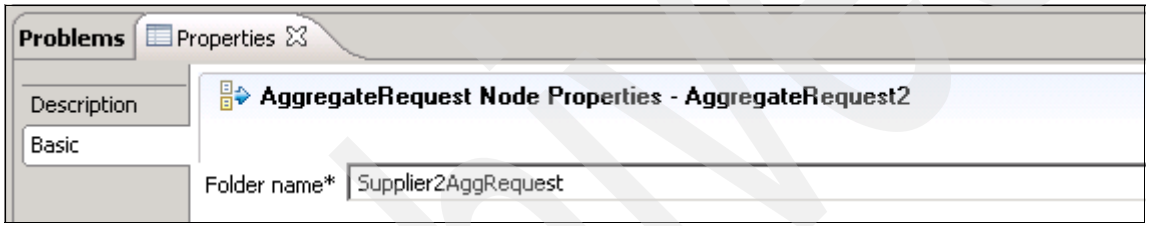


Figure 8-34 Basic properties of the AggregateRequest2 node

Message flow: At the end of this flow, the aggregated messages are sent to the services. The responses from the services are placed on the SC4.FINDSUPPLIERS.AGG.FANIN queue. The next message flow in the aggregation process is the FindSuppliersAggFanIn message flow.

In our test lab, we used message flows to emulate the supplier enterprise information systems. You can see these message flows in 8.5.7, “Supplier1 message flow” on page 266, and 8.5.8, “Supplier2 message flow” on page 272.

8.5.5 FindSuppliersAggFanIn message flow (aggregation fan-in)

The FindSuppliersAggFanIn message flow (Figure 8-35) receives the responses from the suppliers, aggregates the responses. It also sends the aggregated message to the FindSuppliersWSResponse flow, which delivers the response to the HTTP requester.

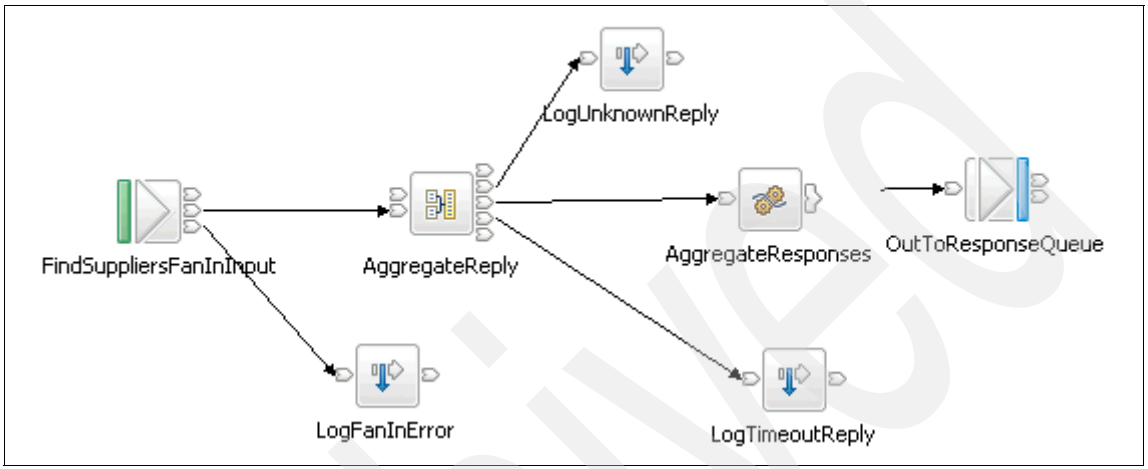


Figure 8-35 FindSuppliersAggFanIn message flow

The FindSuppliersAggFanIn message flow has four primary nodes. In addition, three nodes are used to report errors and terminate the flow processing:

- The *FindSuppliersFanInInput MQInput* node receives the responses from the suppliers. It validates that each message is in an XML wire format that conforms to the list of properties that are contained in the XML1 wire format label of the message set. If the validation fails, an exception is appended to the message and sent to the LogFanInError trace node, which outputs the exception to the local error log.
- The *AggregateReply* node checks that the responses are valid and have not timed out. It then organizes the responses into separate folders within a single message. The folder names are determined by the folder name that is specified in the AggregateRequest node. The single message is then sent to the AggregateResponses Compute node.
- The *AggregateResponses* node parses the message and combines the data, building a single response. The response is sent to the FindSuppliersWSResponse message flow via the OutToResponseQueue node.

Archived

FindSuppliersFanInInput node

The basic properties (Figure 8-36) of the FindSuppliersFanInInput MQInput node define the queue that this flow uses to received messages.

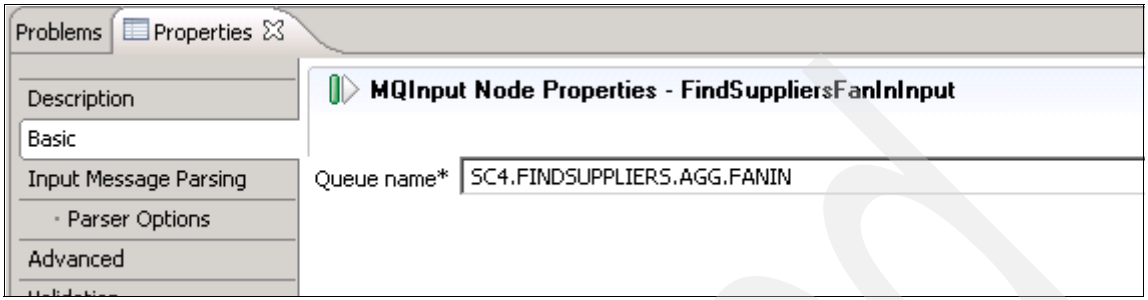


Figure 8-36 Basic properties of the FindSuppliersFanInInput node

The input message parsing properties (Figure 8-37) define how this message should be modeled. The message set is the FindSuppliersMSetprj message set in Figure 8-15 on page 243.

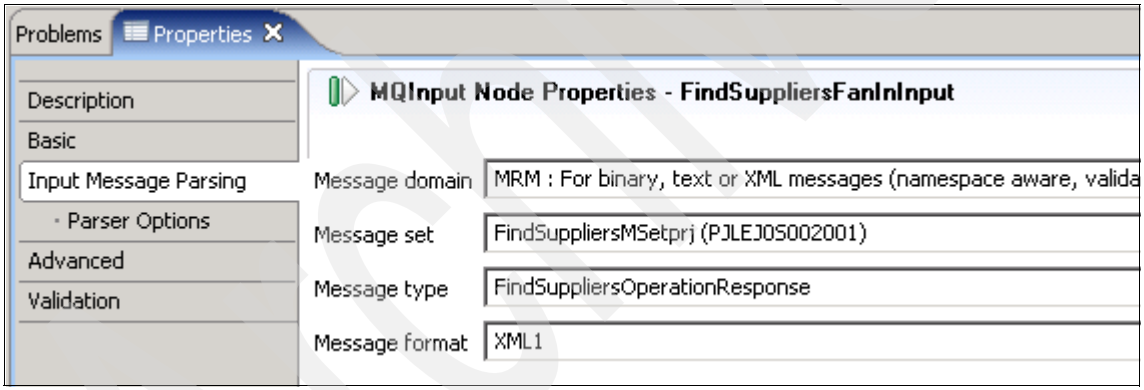


Figure 8-37 Input message parsing properties of the FindSuppliersFanInInput node

Archivo

The validation properties (Figure 8-38) dictate that the input message should be checked for content and value. If the validation fails, an exception is appended to the message and sent to the failure terminal of this node. The failure terminal is connected to the LogFanInError node, which logs the exception to the system log. If validation is successful, the message is passed to the AggregateReply node for further processing.

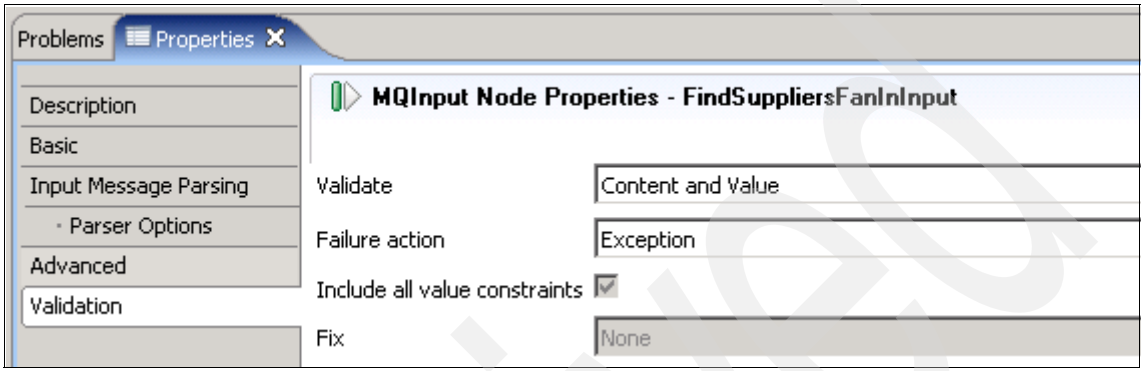


Figure 8-38 Validation properties of the FindSuppliersFanInInput node

LogFanInError node

The basic properties of this Trace node are set to write any exception to the local error log. This node is invoked should an error occur in the FindSuppliersAggFanIn message flow.

The LogFanInError Trace node has the same properties as the Supplier1Error Trace node (see “Supplier1Error Trace node” on page 269).

Archived

AggregateReply node

The basic properties (Figure 8-39) of the AggregateReply node of define which aggregate name is expected for these messages. The aggregate name that is specified here should match the aggregate name that is specified in the AggregateControl node (Figure 8-27 on page 251).

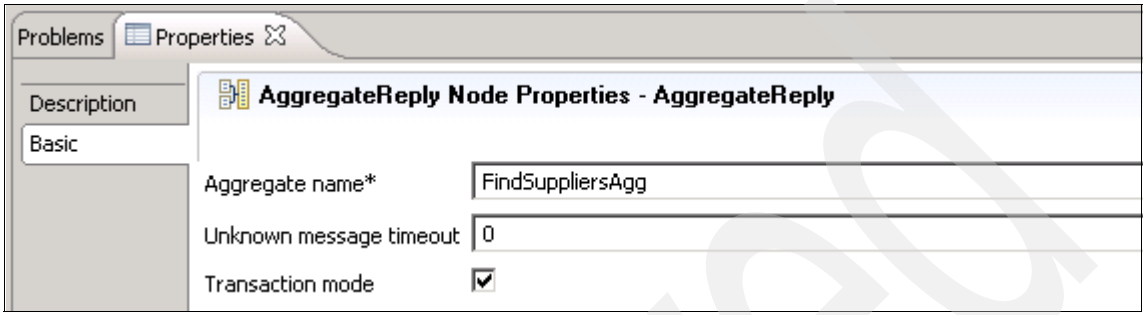


Figure 8-39 Basic properties of the AggregateReply node

LogTimeoutReply Trace node

The basic properties of the LogTimeoutReply Trace node are set to write any exception to the local error log. This node is invoked should a time-out condition occur in the message flow. The time-out period is defined in the AggregateControl node (Figure 8-27 on page 251).

The LogTimeoutReply node has the same properties as the Supplier1Error node (see “Supplier1Error Trace node” on page 269).

LogUnknownReply Trace node

The basic properties of the LogUnknownReply Trace node are set to write any exception to the local error log. This node is invoked should an unknown reply be detected by the AggregateReply node. An *unknown reply* is a message that is delivered without an aggregate name matching the aggregate name that is specified in the AggregateReply node.

The LogUnknownReply Trace node has the same properties as the Supplier1Error Trace node (see “Supplier1Error Trace node” on page 269).

Archived

AggregateResponses Compute node

The basic properties (Figure 8-40) of the AggregateResponses Compute node define the location of the ESQL module used to implement the logic of this node.

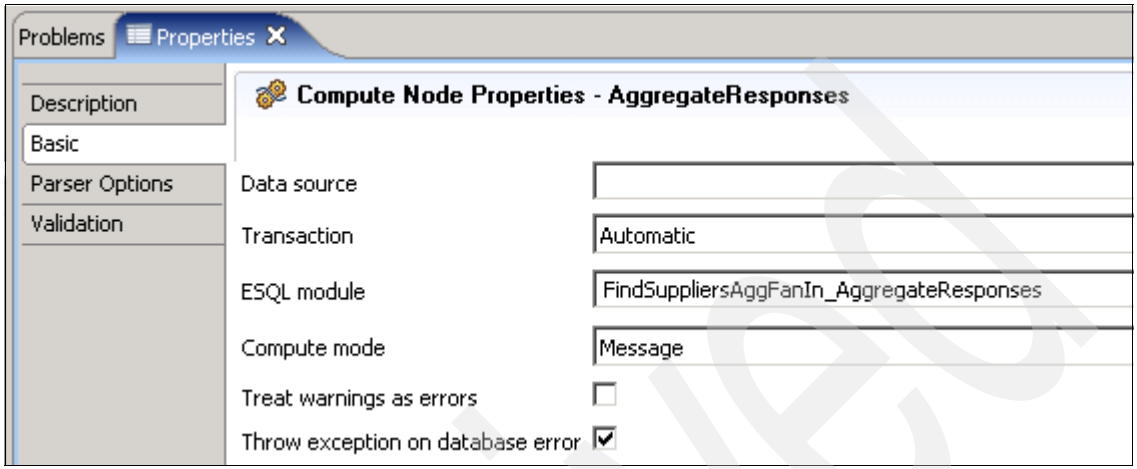


Figure 8-40 Basic properties of the AggregateResponses node

The ESQL shown in Example 8-3 looks for the response data in the ComIbmAggregateReplyBody of the message tree. The response data is copied into the output message and conforms to the XML wire format characteristics and the Envelope message type. The Envelope message type is SOAP, which corresponds to the message type that is understood by the HTTP requester. The MQRFH2 header is also copied from the response to the output message, since it contains the HTTPRequestIdentifier of the original HTTP requester.

Upon successful processing of the message, the message is propagated to the OutToResponseQueue node, which is the input for the FindSuppliersWSResponse flow. This flow responds to the HTTP requester.

Example 8-3 The ESQL of the AggregateResponses node

```
CREATE COMPUTE MODULE FindSuppliersAggFanIn_AggregateResponses
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN

      -- First, we copy the message headers
      CALL CopyMessageHeaders();

      -- Next, we create the MQMD in the output message
      CREATE NEXTSIBLING OF OutputRoot.Properties DOMAIN 'MQMD';
```

```
      -- Next, we populate
--    SET OutputRoot.MQMD.CorrelId =
InputRoot.ComIbmAggregateReplyBody.Supplier1AggRequest.MQMD.CorrelId;
--    SET OutputRoot.MQMD.MsgId =
InputRoot.ComIbmAggregateReplyBody.Supplier1AggRequest.MQMD.CorrelId;

      -- Copy the MQRFH2 from one of the Aggregate response
      CREATE NEXTSIBLING OF OutputRoot.MQMD DOMAIN 'MQRFH2';
      SET OutputRoot.MQRFH2 =
InputRoot.ComIbmAggregateReplyBody.Supplier1AggRequest.MQRFH2;
      --Build the body of the response message
      CREATE NEXTSIBLING OF OutputRoot.MQRFH2 DOMAIN 'MRM';
      SET OutputRoot.Properties.MessageSet = 'PJLEJOS002001';
      SET OutputRoot.Properties.MessageType = 'Envelope';
      SET OutputRoot.Properties.MessageFormat = 'XML1';
      SET
OutputRoot.MRM.ns:Body.null2:FindSuppliersOperationResponse.suppliers.supplier[1] =
InputRoot.ComIbmAggregateReplyBody.Supplier1AggRequest.MRM.suppliers[1].supplier;
      SET
OutputRoot.MRM.ns:Body.null2:FindSuppliersOperationResponse.suppliers.supplier[2] =
InputRoot.ComIbmAggregateReplyBody.Supplier2AggRequest.MRM.suppliers[1].supplier;

      RETURN TRUE;
   END;

   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;

END MODULE;
```

Archived

OutToResponseQueue node

The OutToResponseQueue MQOutput node is used to output the aggregated response message to the input of the Web service response flow, FindSuppliersWSResponse. The basic properties (Figure 8-41) define the queue where this flow outputs the response message.

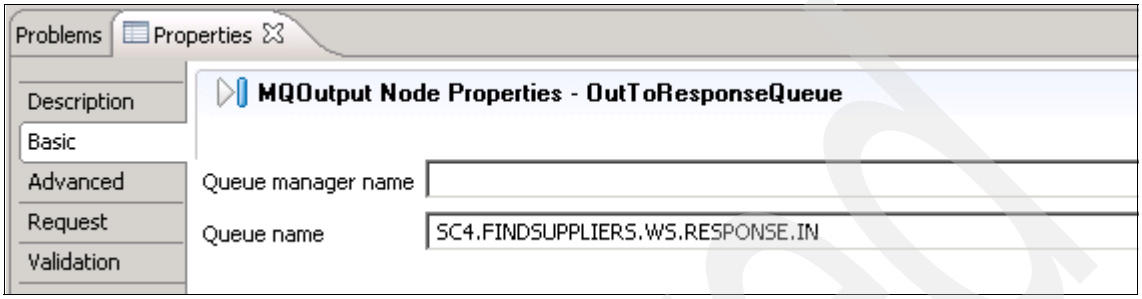


Figure 8-41 Basic properties of the OutToResponseQueue node

8.5.6 FindSuppliersWSResponse message flow

The FindSuppliersWSResponse message flow (Figure 8-42) responds to the HTTP requester.

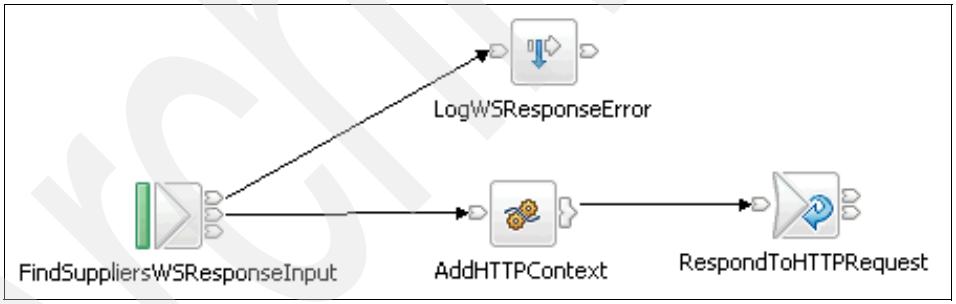


Figure 8-42 FindSuppliersWSResponse message flow

The FindSuppliersWSResponse message flow contains the following main nodes as well as one error node:

- The *FindSuppliersWSResponseInput* MQInput node receives the response from the FindSuppliersAggFanIn flow via a queue.
- The *AddHTTPContext* Compute node parses the HTTPRequestIdentifier and stores it in the LocalEnvironment portion of the message tree as required by the RespondToHTTPRequest HTTPReply node.

Archived

- The *RespondToHTTPRequest* node initiates the switch in the messaging transport from MQ to HTTP by propagating the response to the broker's HTTP listener, which responds to the HTTP requester with the response message contents.

FindSuppliersWSResponseInput node

The FindSuppliersWSResponseInput MQInput node receives the aggregated response message from the FindSuppliersAggFanIn message flow, parses the message, and validates that the message is of the Envelope type. The basic properties (Figure 8-43) define the queue that is used to receive messages.

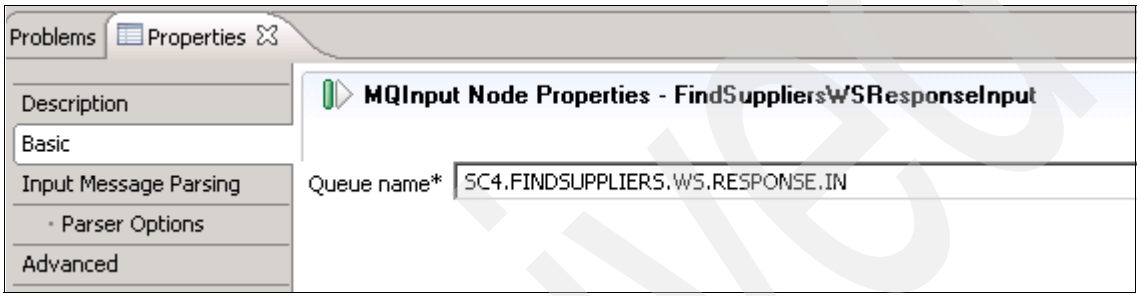


Figure 8-43 Basic properties of the FindSuppliersWSResponseInput node

The input message parsing properties (Figure 8-44) define how the message should be modeled upon input.

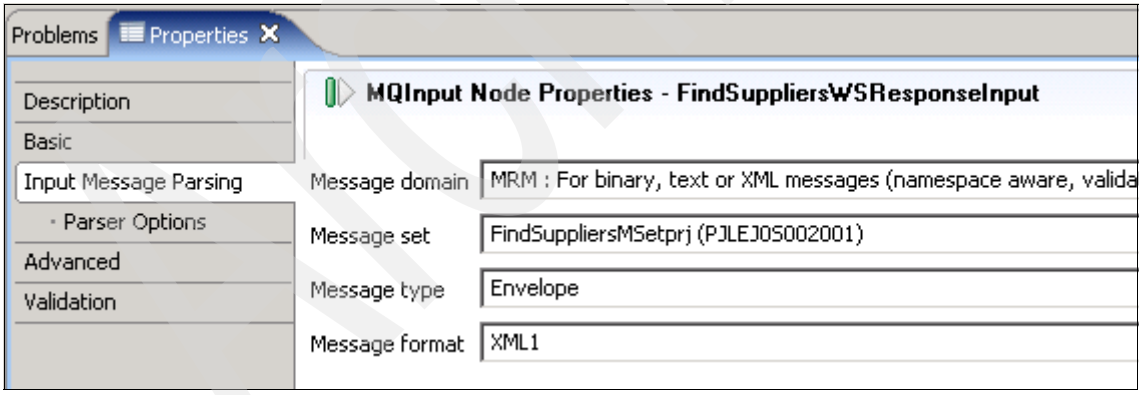


Figure 8-44 Input message parsing properties of the FindSuppliersWSResponseInput node

The validation properties (Figure 8-45) dictate that the input message should be checked for content and value. If validation fails, an exception is appended to the message and sent to the failure terminal of this node. The failure terminal is connected to the LogWSResponseError Trace node, which logs the exception to the local error log. If validation is successful, the message is passed to the AddHTTPContext Compute node for further processing.

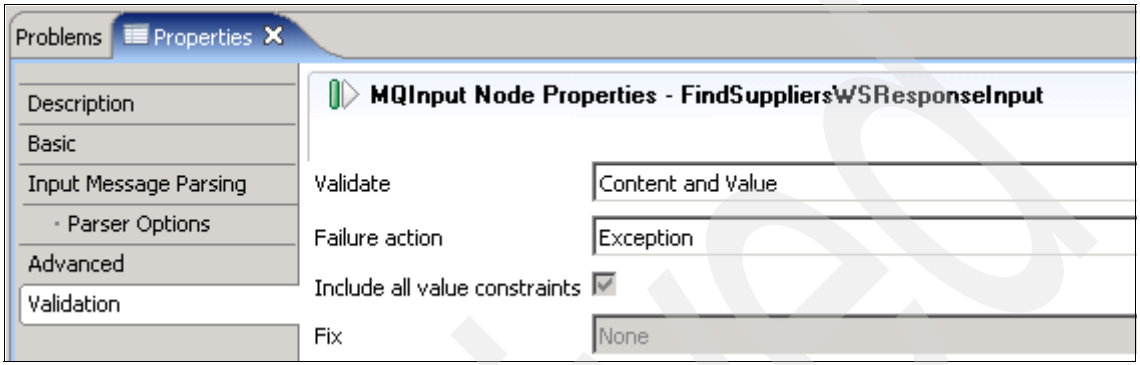


Figure 8-45 Validation properties of the FindSuppliersWSResponseInput node

LogWSResponseError node

The basic properties of the LogWSResponseError Trace node are set to write any exception to local error log. This node is invoked should an error occur in the FindSuppliersWSResponse message flow.

The LogWSResponseError Trace node has the same properties as the Supplier1Error Trace node (see “Supplier1Error Trace node” on page 269).

AddHTTPContext Compute node

The AddHTTPContext Compute node copies the HTTPRequestIdentifier from the MQRFH2 usr folder into the LocalEnvironment of the message tree. The RespondToHTTPRequest HTTPReply node uses the HTTPRequestIdentifier in the LocalEnvironment to send the response to the original requester.

Archived

The basic properties (Figure 8-46) define the location of the ESQL module that is used to implement the logic of this node.

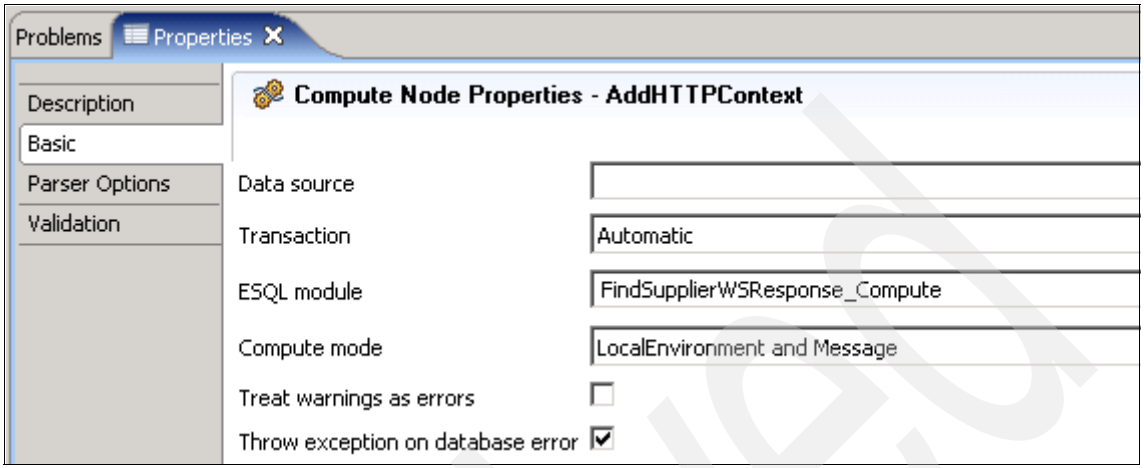


Figure 8-46 Basic properties of the AddHTTPContext Compute node

Example 8-4 shows the ESQL.

Example 8-4 ESQL for the AddHTTPContext Compute node

```
CREATE COMPUTE MODULE FindSupplierWSResponse_Compute
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      -- CALL CopyMessageHeaders();
      CALL CopyEntireMessage();
      SET OutputLocalEnvironment.Destination.HTTP.RequestIdentifier =
CAST (InputRoot.MQRFH2.usr.HTTPRequestIdentifier AS BLOB);
      SET OutputRoot.MQMD = NULL;
      SET OutputRoot.MQRFH2 = NULL;
      RETURN TRUE;
   END;

   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;
```

Archived

```
   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

RespondToHTTPRequest node

The RespondToHTTPRequest HTTPReply node (Figure 8-47) replies to the original HTTP requester.

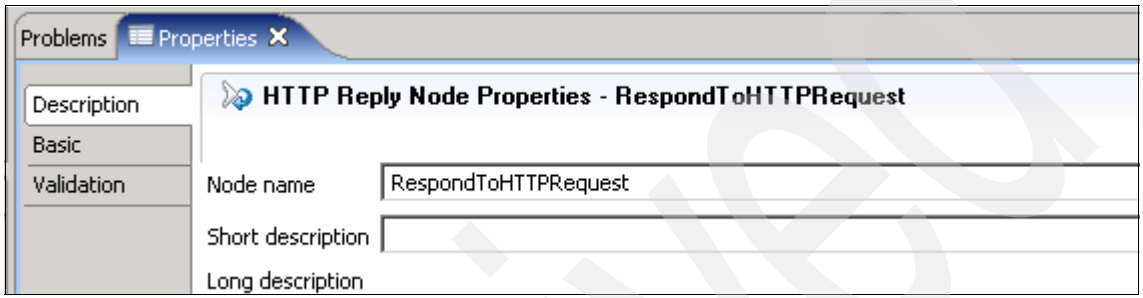


Figure 8-47 The RespondToHTTPRequest node

8.5.7 Supplier1 message flow

Note: The Supplier1 message flow emulates the Supplier1 system. It is not part of the aggregation process, but rather the service that one of the aggregated messages is sent to. In a production system, these services are probably not implemented with message flows. However, we include this because it illustrates converting a CWF message to the XML format.

The Supplier1 message flow (Figure 8-48 on page 267) emulates the Supplier1 system, responding to the aggregate request. The Supplier1 message flow emulates a enterprise information system that communicates using the MQ transport and can only read messages in the CWF1 message format. The Supplier1 message flow demonstrates conversion of the message back to XML format as it builds the response message.

Archived

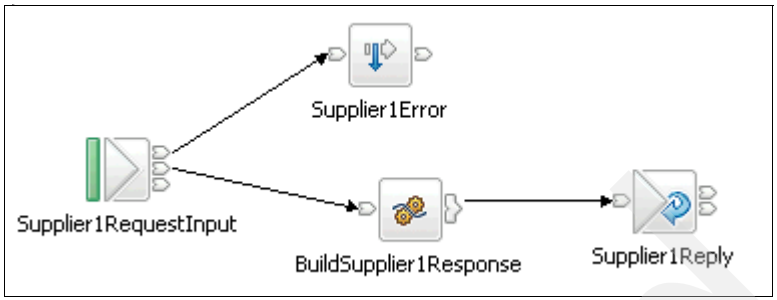


Figure 8-48 Supplier1 message flow

The Supplier1 message flow has the following nodes:

- The *Supplier1RequestInput* MQInput node receives the request and validates that the message is in CWF message format.
- The *BuildSupplier1Response* Compute node takes a message in the CWF message format and uses it to build the response in the XML message format.
- The *Supplier1Reply* MQReply node sends the message to the queue specified in the reply-to queue field in the message. This field was previously set in the AggRequestToSupplier1 MQOutput node in the FindSuppliersAggFanIn.msgflow.

Supplier1RequestInput MQInput node

The basic properties (Figure 8-49) of the Supplier1RequestInput MQInput node define the queue that the flow uses to receive messages.

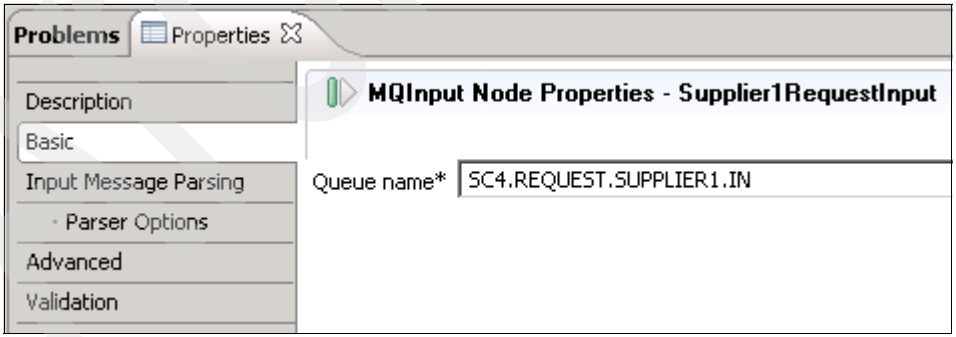


Figure 8-49 Basic properties of the Supplier1RequestInput node

The input message parsing properties (Figure 8-50) define how this message should be modeled upon input.

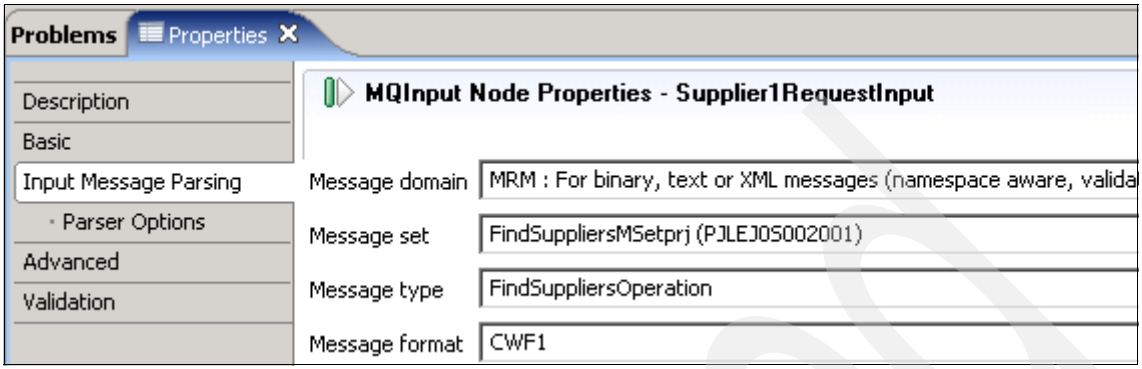


Figure 8-50 Input message parsing properties of the Supplier1RequestInput node

The validation properties (Figure 8-51) dictate that the input message should be checked for content and value. If validation fails, an exception is appended to the message and sent to failure terminal of this node. The failure terminal is connected to the Supplier1Error node, which logs the exception to the system log. If validation is successful, the message is passed to the BuildSupplier1Response node for processing.

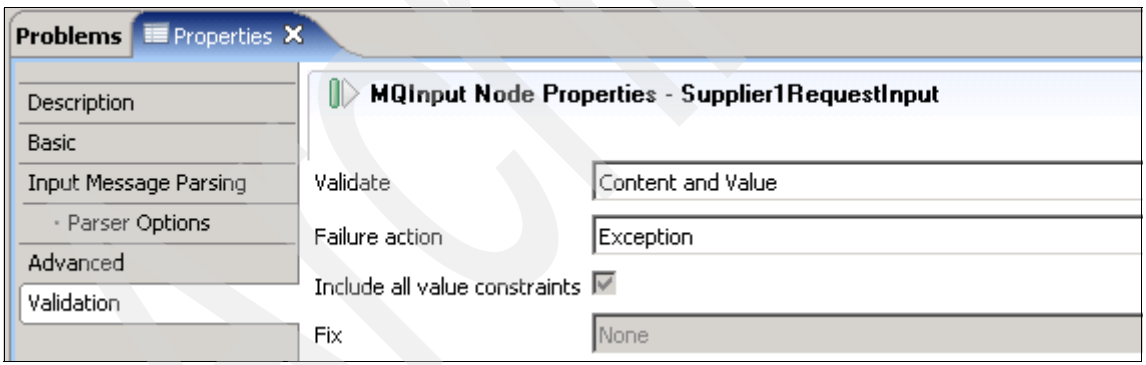


Figure 8-51 Validation properties of the Supplier1RequestInput node

Supplier1Error Trace node

The basic properties (Figure 8-52) of the Supplier1Error Trace node are set to write any exception to a local error log. This node is invoked should an error occur in the Supplier1 message flow.

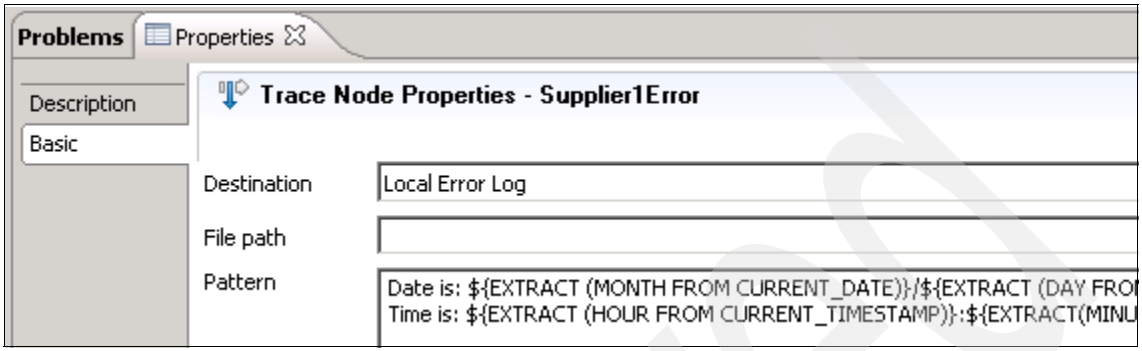


Figure 8-52 Basic properties of the Supplier1Error Trace node

Example 8-5 shows the full code in the Pattern field.

Example 8-5 Pattern of the error written to the local error log

```
Date is: ${EXTRACT (MONTH FROM CURRENT_DATE)}/${EXTRACT (DAY FROM
CURRENT_DATE)}/${EXTRACT (YEAR FROM CURRENT_DATE)}
Time is: ${EXTRACT (HOUR FROM CURRENT_TIMESTAMP)}:${EXTRACT(MINUTE FROM
CURRENT_TIMESTAMP)}

*** A Supplier1 flow error condition occurred. ***

Environment is:
${Environment}

Exception List is:
${ExceptionList}

The Message is:
${Root}
```

Archived

BuildSupplier1Response Compute node

The basic properties (Figure 8-53) of the BuildSupplier1Response Compute node define the location of the ESQL module that is used to implement the logic of this node.

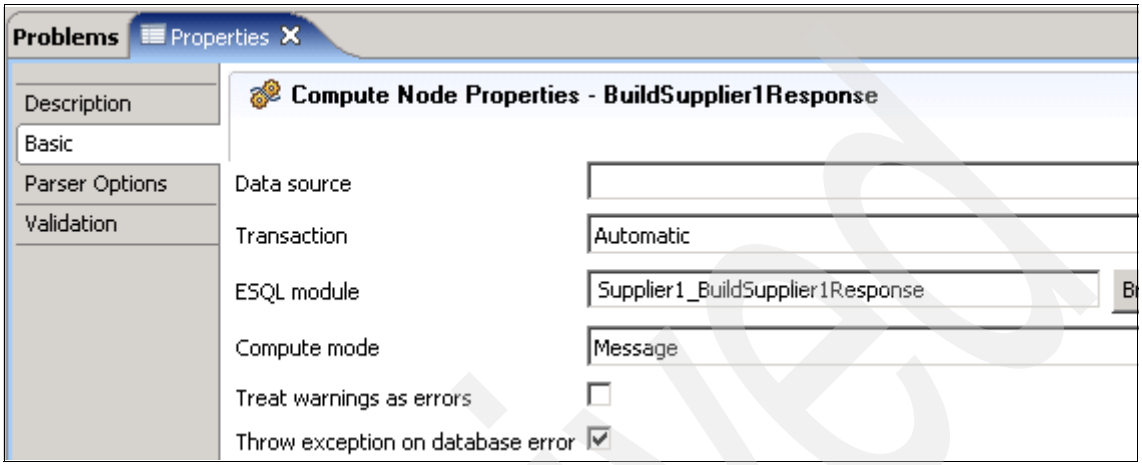


Figure 8-53 Basic properties of the BuildSupplier1Response node

Example 8-6 shows the ESQL, which changes the wire format of the message from CWF to XML as it builds the response. The values are hard coded because it is not the intent of this scenario to demonstrate any logic in the Supplier1 system.

Example 8-6 ESQL for the BuildSupplier1Response node

```
CREATE COMPUTE MODULE Supplier1_BuildSupplier1Response
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN

      -- First, we copy the message headers
      CALL CopyMessageHeaders();

      -- Next, we set the properties of the output
      -- message since we convert from CWF to XML format
      SET OutputRoot.Properties.MessageSet = 'PJLEJOS002001';
      SET OutputRoot.Properties.MessageType = 'FindSuppliersOperationResponse';
      SET OutputRoot.Properties.MessageFormat = 'XML1';

      -- Next, we create the MRM domain to store the response
      CREATE LASTCHILD OF OutputRoot DOMAIN 'MRM';
```

Archived

```
      -- To simplify this scenario, while maintaining the focus, we hardcode
      -- the response from the supplier
      SET OutputRoot.MRM.suppliers.supplier[1].item = InputRoot.MRM.item;
      SET OutputRoot.MRM.suppliers.supplier[1].id = 'supplier1ID';
      SET OutputRoot.MRM.suppliers.supplier[1].price = '10.00';
      SET OutputRoot.MRM.suppliers.supplier[1].name = 'Supplier1';
      SET OutputRoot.MRM.suppliers.supplier[1].address ='123 Supplier1 Street';
      SET OutputRoot.MRM.suppliers.supplier[1].city='RTP';
      SET OutputRoot.MRM.suppliers.supplier[1].state='NC';
      SET OutputRoot.MRM.suppliers.supplier[1].zip='27709';
      RETURN TRUE;
   END;

   CREATE PROCEDURE CopyMessageHeaders() BEGIN
      DECLARE I INTEGER 1;
      DECLARE J INTEGER;
      SET J = CARDINALITY(InputRoot.*[]);
      WHILE I < J DO
         SET OutputRoot.*[I] = InputRoot.*[I];
         SET I = I + 1;
      END WHILE;
   END;

END MODULE;
```

Supplier1Reply MQReply node

The Supplier1Reply MQReply node processes the request message. It sends the message to the queue specified in the reply-to queue set in the AggRequestToSupplier1 node of the FindSuppliersAggFanOut.msgflow. See Figure 8-54.

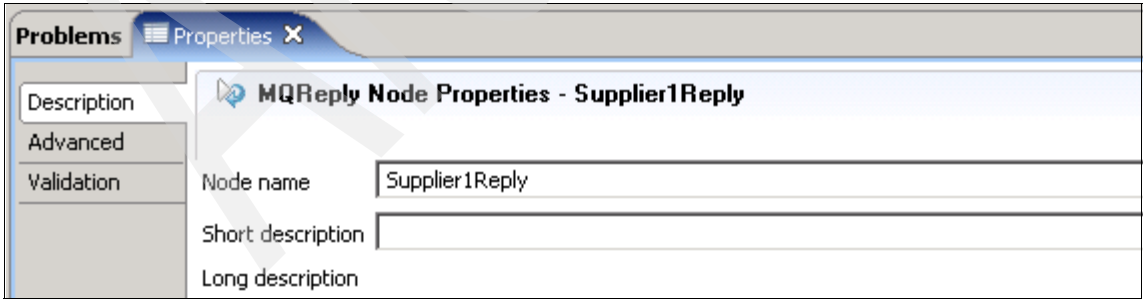


Figure 8-54 Supplier1Reply node

Archived

8.5.8 Supplier2 message flow

The Supplier2 message flow (Figure 8-55) emulates the Supplier2 system. With the exception of the queue for the MQInput node (SC4.REQUEST.SUPPLIER2.IN), it is identical to the message flow for Supplier1.

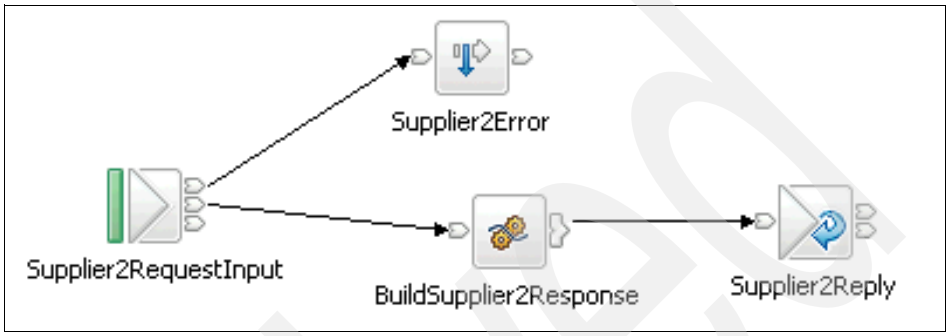


Figure 8-55 Supplier2 message flow

The Supplier2.msgflow message flow has the following nodes:

- The *Supplier2RequestInput* MQInput node receives the request and validates that the message is in the CWF1 message format.
- The *BuildSupplier1Response* Compute node takes a message in the CWF message format and uses it to build the response in the XML message format.
- The *Supplier2Reply* node is a MQReply node, which sends the message to the queue that is specified in the reply-to queue field in the message. This field was previously set in the AggRequestToSupplier2 MQ output node in the FindSuppliersAggFanIn message flow.

Archived

8.6 Testing the flow

In this section, we show the integration testing between the integration module and the message flow in WebSphere Integration Developer.

First, we tested the flow using the Integration Test Client as shown in Figure 8-56.

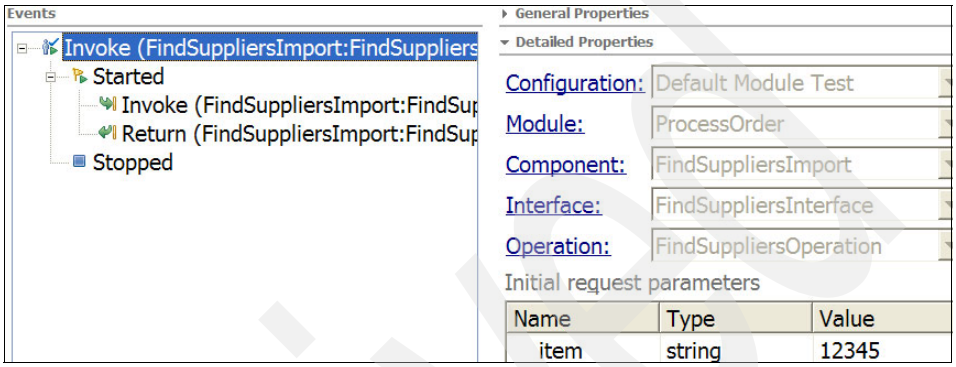


Figure 8-56 Testing the process

Figure 8-57 shows the results of the test.

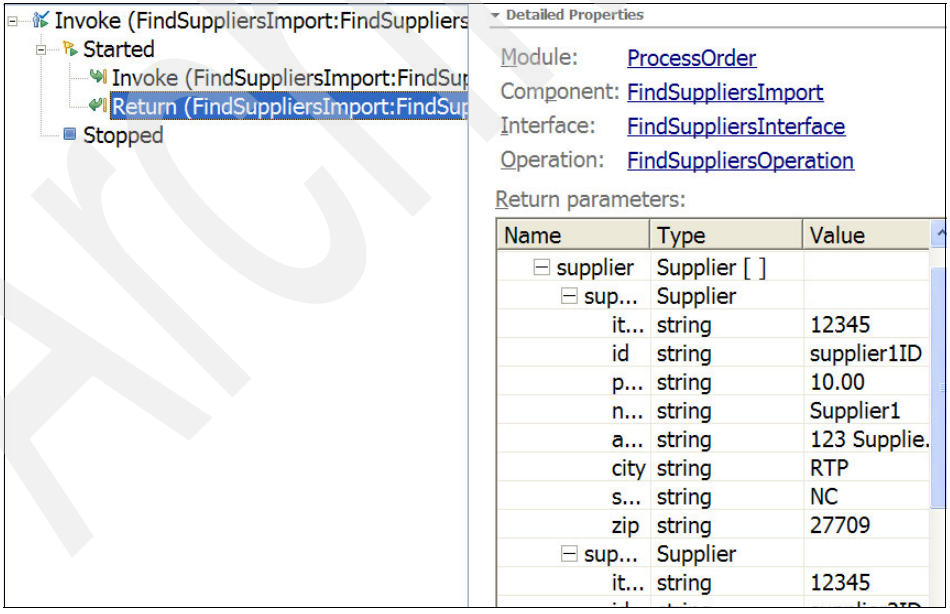


Figure 8-57 Scenario test results

Archive

Archived

9

Scenario: Asynchronous callback

In the scenario presented in this chapter, we demonstrate the following concepts:

- Dynamic routing using WebSphere Service Registry and Repository to find the destination URL
- Implementation of error handling in a message flow using a replay and timeout mechanism
- Asynchronous callback interaction pattern

  The client implements a callback interface that is invoked when the response is ready.
- Message transformation

  The message is delivered to WebSphere Message Broker as an MQ message. The message is sent to a service as a SOAP message. The response from the service is transformed back to an MQ message for delivery to WebSphere Process Server.
- Connectivity using an MQ binding

  The business process uses an import component with an MQ binding to send a message to the message flow. An MQInput node is used in the message flow that receives the message from the process. The response is sent back to the business process using an MQOutput node.

© Copyright IBM Corp. 2008. All rights reserved.

9.1 Scenario overview

Figure 9-1 shows the basic flow of this scenario.

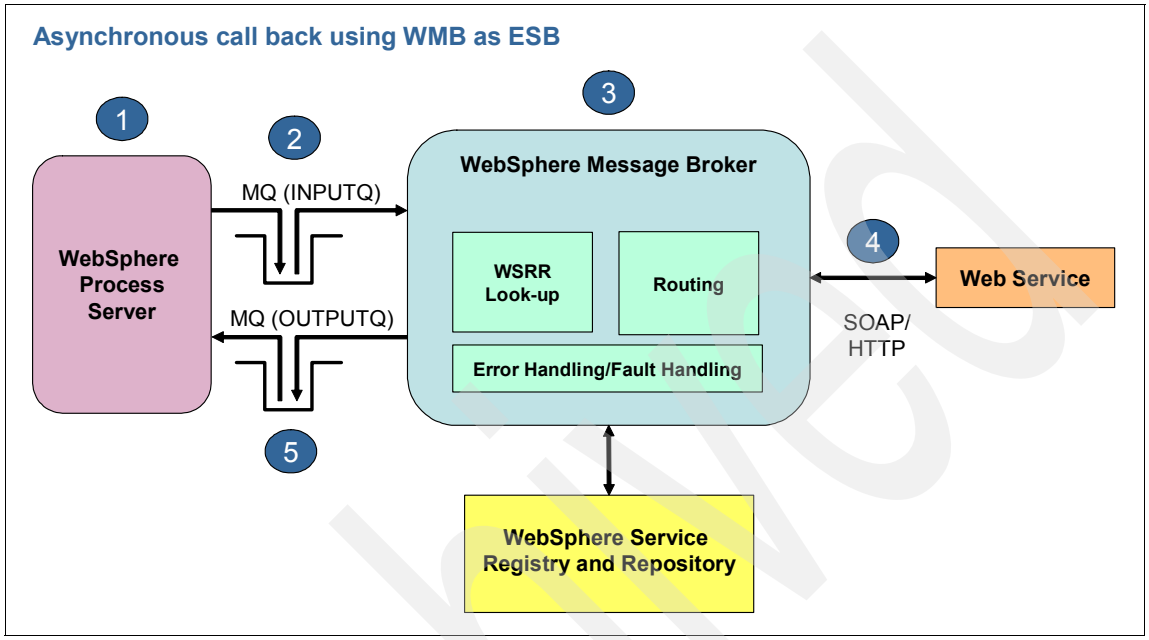


Figure 9-1 Scenario overview

We describe the highlights of this scenario, numbered in Figure 9-1, as follows:

1. A business process hosted in WebSphere Process Server invokes a service using an import component with an MQ binding.
2. The import component puts the message on a WebSphere MQ queue (INPUTQ) where it is picked up and processed by a message flow in WebSphere Message Broker.
3. WebSphere Message Broker uses an SRRetrieveITService node to perform an endpoint lookup for the Web service and dynamically routes the message to the service.
4. The call to the Web service is made using SOAP/HTTP.
5. The reply is returned from WebSphere Message Broker over a queue (OUTPUTQ) to the business process through the callback interface.

Archived

A correlation set for the message is used to match the asynchronous response with the request as illustrated in Figure 9-2:

1. When the business process sends a message to the import component, it initializes the correlation set from the inbound message.
2. The import component then places the message on the queue.
3. The reply is placed on the queue that the export component for the process listens to. The reply has the same correlation set as the sent message.
4. The reply is sent to the business process.

Correlation sets: Correlation sets are used to uniquely identify business processes using business data. If a message must be delivered to a business process, correlation sets are used to identify the particular process instance, with which the message is associated. Correlation sets are initialized with values from process inbound or outbound messages.

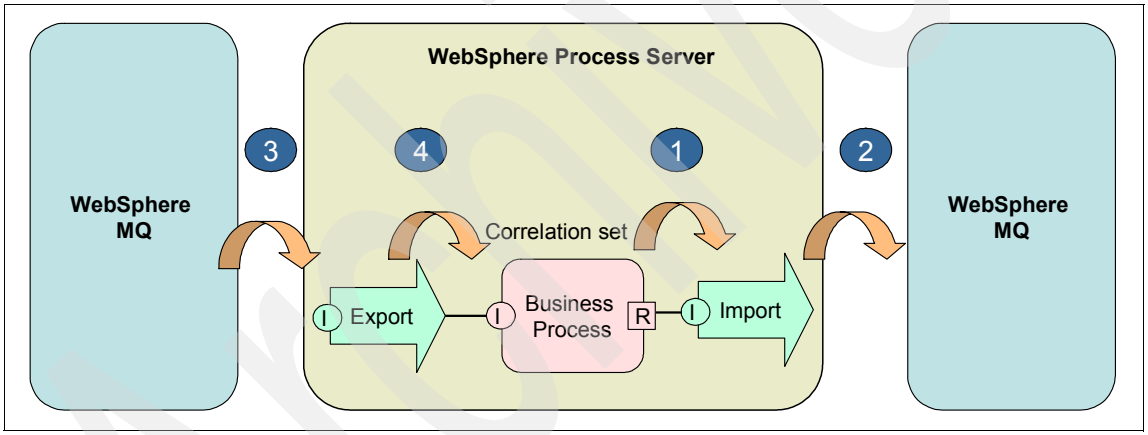


Figure 9-2 Using a correlation set in an asynchronous callback

9.2 Environment

The scenario was developed in an environment with the following components:

- WebSphere Integration Developer 6.0.2.2
- WebSphere Process Server 6.0.2.0 (Build Number o0648.28)
- WebSphere Application Server 6.0.2.17 (Build Number cf170648.10)
- WebSphere Message Broker run time 6.0.0.5

- WebSphere Message Brokers Toolkit 6.0.2 (Interim Fix 009)
- WebSphere Service Registry and Repository V6.0.2
- IA9O SupportPac for WebSphere Message Broker

  This SupportPac provides the SOAPEnvelope and SOAPExtract nodes.
- IA9Q SupportPac

  This SupportPac provides WebSphere Message Broker V6 with nodes and related capabilities that interface with WebSphere Service Registry and Repository to access service metadata for use in a mediation message flow.
- WebSphere MQ 6.0.0.0
- DB2 Enterprise Server 8.1.7.445
- Windows XP Professional Version 2002 SP2

Note: For more information about WebSphere Message Broker SupportPacs, refer to the following Web page:

http://www-1.ibm.com/support/docview.wss?uid=swg27007197

9.3 WebSphere Service Registry and Repository

The target Web service in this scenario is implemented as a Java 2 platform, Enterprise Edition (J2EE) application that is deployed to WebSphere Application Server. The Web service is published to WebSphere Service Registry and Repository.

9.3.1 OrderWebService Web service

The OrderWebService is a simple Web service that takes the Order element that wraps orderID and returns the Invoice element that wraps orderID along with its price. Example 9-1 shows the Web Services Description Language (WSDL) file that describes this service.

Example 9-1 OrderWebService.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:yc="http://www.yourco.com" name="OrderService"
targetNamespace="http://www.yourco.com">
```

```
  <wsdl:types>
    <xsd:schema targetNamespace="http://www.yourco.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Order">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="OrderID" type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Invoice">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="OrderID" type="xsd:string"/>
            <xsd:element name="Price" type="xsd:string"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>

  </wsdl:types>
  <wsdl:message name="getOrderResponse">
    <wsdl:part element="yc:Invoice" name="InvoiceOB"/>
  </wsdl:message>
  <wsdl:message name="getOrderRequest">
    <wsdl:part element="yc:Order" name="OrderOB"/>
  </wsdl:message>
  <wsdl:portType name="OrderPortType">
    <wsdl:operation name="getOrder">
      <wsdl:input message="yc:getOrderRequest"/>
      <wsdl:output message="yc:getOrderResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="OrderPortBinding" type="yc:OrderPortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="getOrder">
      <soap:operation soapAction="http://www.yourco.com/getOrder"/>
      <wsdl:input>
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"/>
      </wsdl:output>
```

Archived

```
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="OrderService">
    <wsdl:port binding="yc:OrderPortBinding" name="OrderPort">
      <soap:address
location="http://localhost:9080/OrderWebService/services/OrderPort"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Tip: This Web service is Web Services Interoperability Organization (WS-I) compliant to promote interoperability across different across platforms, Web service engines, and programming languages.

9.3.2 Defining the registry

The WSDL for the OrderWebService was published to the WebSphere Service Registry and Repository by using the following steps:

1. Using a browser, access the registry at:

   `http://localhost:9080/ServiceRegistry`

2. In the left navigation pane, select **Service Documents** → **LoadDocuments**.
3. In the right pane, complete these tasks:
   a. Browse to the WSDL file and select it.
   b. Select **WSDL** as the document type.
   c. Click **OK**.

Archived

See Figure 9-3.

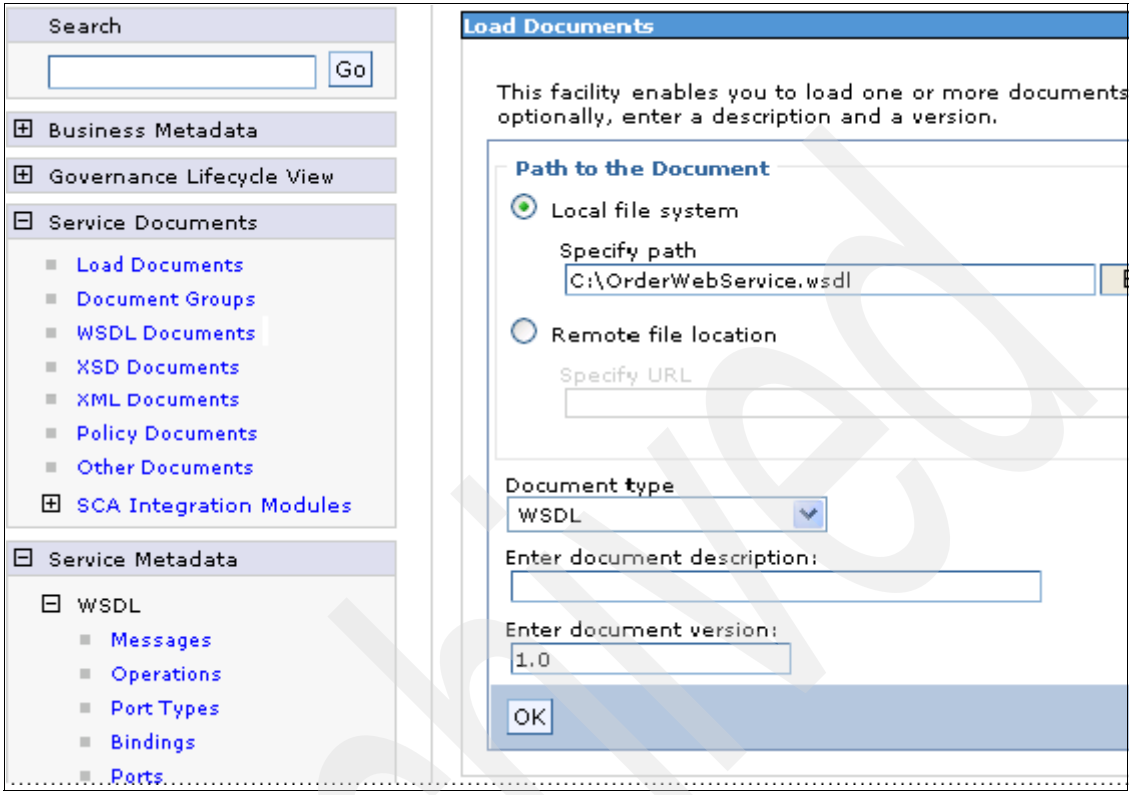


Figure 9-3 Load the WSDL for the service to the registry

4. Click **Finish**.

Next, a user property was added to the OrderWebservice port for use as a search argument for clients looking up the service. In this example, two versions of the order service are published to the registry. A QualityOfService property is added to both versions.

- One version has the QualityOfService property set to a value of `Moderate`.
- The second version of the QualityOfService property is set to a value of `High`.

Therefore, a lookup for the Web service can be performed based on a certain value of the QualityOfService property.

Archived

To add a property:

1. In the left navigation pane, navigate to **Service Metadata** → **WSDL** → **Ports**.
2. Click **OrderPort** → **Properties**.
3. Click **Add** and add a key and value for the property as shown in Figure 9-4.

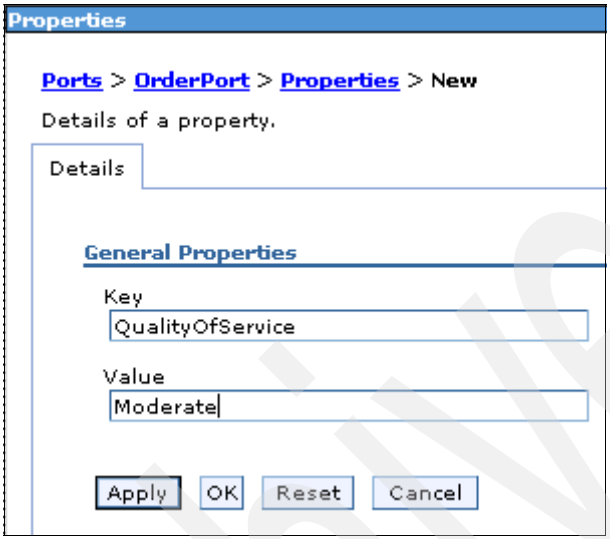


Figure 9-4 Adding a property to a port

4. Repeat steps 1 through 3, with the same WSDL, but change the version to be 2.0 and the value of the QualityOfService property to be **High**.

> **WSDL:** For illustration purposes, the same WSDL (with the same endpoint) is loaded to WebSphere Service Registry and Repository. In a real-world scenario, it will be either a different version of WSDL (with a different endpoint) or a different WSDL.

9.4 WebSphere MQ infrastructure

When you install a Service Component Architecture (SCA) module that contains MQ bindings, the run time is automatically configured to allow connectivity with WebSphere MQ. However, the MQ systems administrator must configure the WebSphere MQ queue managers, because these are not automatically configured.

Archived

The four queues shown in Figure 9-5 are required for the scenario. All queues were defined to WBRK6_DEFAULT_QUEUE_MANAGER by using WebSphere MQ Explorer.

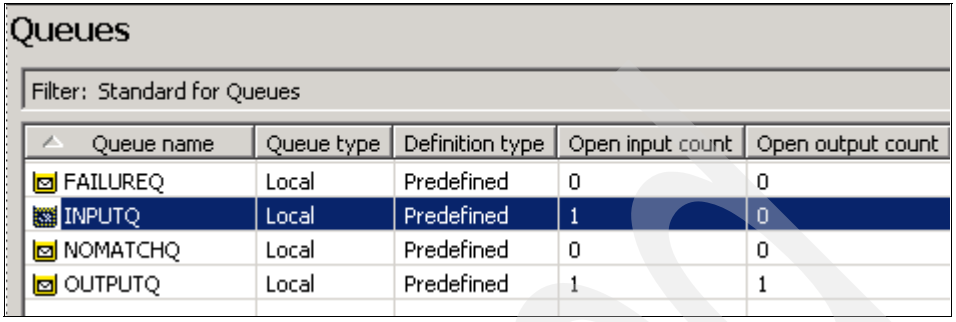


Figure 9-5 WebSphere MQ queues required for the scenario

9.5 Business integration module

In this section, we demonstrate the implementation of a business integration module deployed on WebSphere Process Server. The business process gets a request from a client, sends the message to a queue (to which the message flow listens), and sets a listener to queue (to which the message flow will reply). After a reply arrives on the queue, the business process replies to its client.

9.5.1 Business integration module components

The business process is implemented in a business integration module called *OrderModule*. Figure 9-6 shows the assembly diagram for this module.

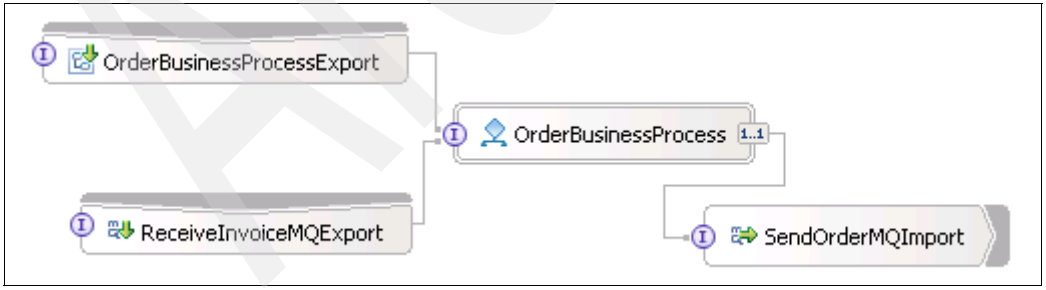


Figure 9-6 Scenario assembly diagram

Archive

The assembly diagram consists of the following components:

- *OrderBusinessProcessExport* is the export component that the client uses to access the business process. It has the OrderProcessInterface interface.
- *ReceiveInvoiceMQExport* is the export component that receives reply messages. It has the ReceiveInterface interface.
- *OrderBusinessProcess* contains the business process implementation. It has two interfaces, ReceiveInterface and OrderProcessInterface.
- *SendOrderMQImport* is the import component that sends request messages to a service via WebSphere Message Broker. It has the SendInterface interface.

Asynchronous and synchronous calls: The asynchronous callback is carried out inside the business process, while the client to the process makes a synchronous call.

The sequence diagram in Figure 9-7 summarizes the interaction between the client, SCA module, queues and WebSphere Message Broker flow.

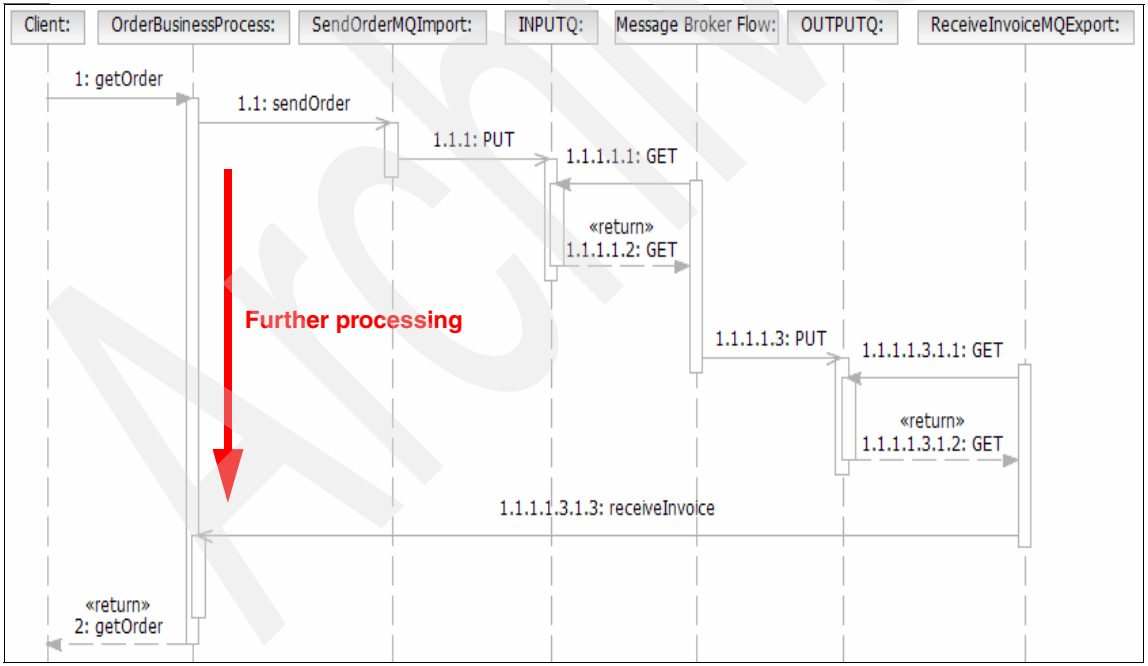


Figure 9-7 Scenario sequence diagram

Archive

9.5.2 Business objects

The module uses the following business objects:

- The Order business object for the request data
- The Invoice business object for the response data

The Order business object has the `OrderID` string attribute and namespace `http://www.yourco.com`. See Figure 9-8.

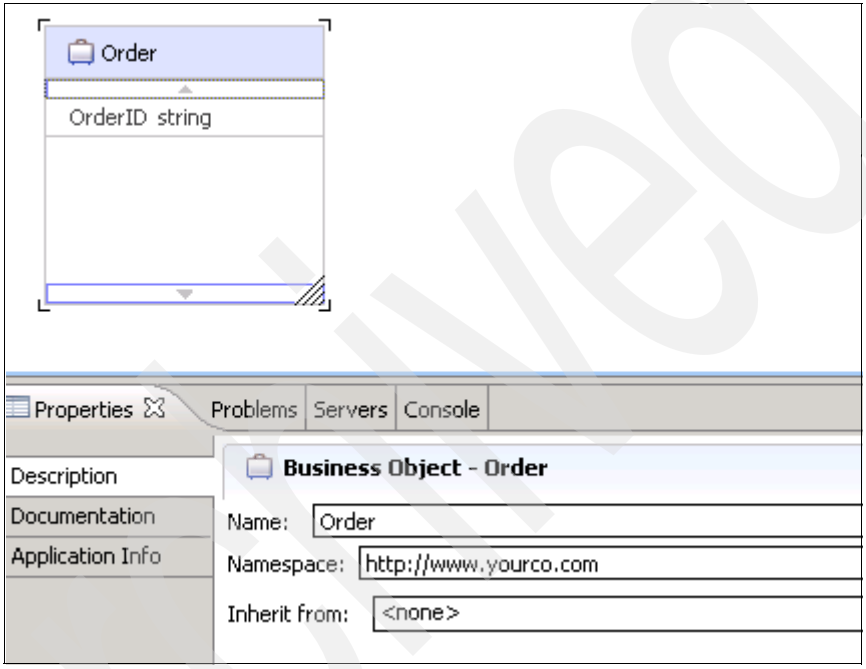


Figure 9-8 Order ID business object

Archived

The Invoice business object has the `OrderID` and `Price` string attributes and also uses namespace `http://www.yourco.com`. See Figure 9-9.

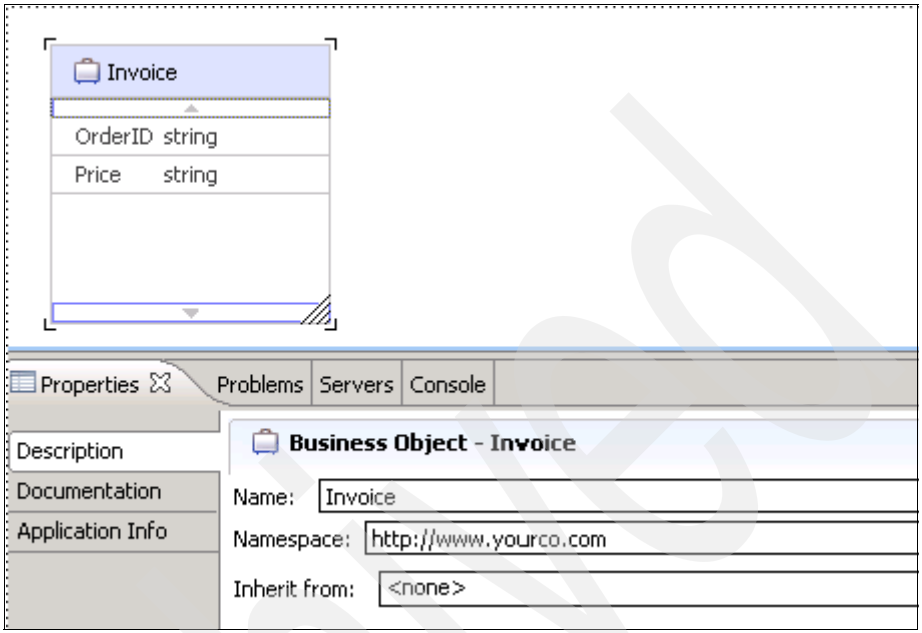


Figure 9-9 Invoice business object

Namespace of the business object: The namespace of the business objects is `http://www.yourco.com`, which is the same namespace of the OrderWebService service elements, because these business objects will be sent from the business process to the message flow with a Web service call. The Web service expects a SOAP message that is qualified with the correct namespace. Therefore, the namespace of the business object must match the schema targetNamespace in the WSDL for the Web service shown in Example 9-1 on page 278.

9.5.3 Interfaces

The module uses three interfaces. The first interface, OrderProcessInterface (Figure 9-10), is the interface for the business process. It has one request-response operation that sends the order using the Order business object and receives the invoice using the Invoice business object.

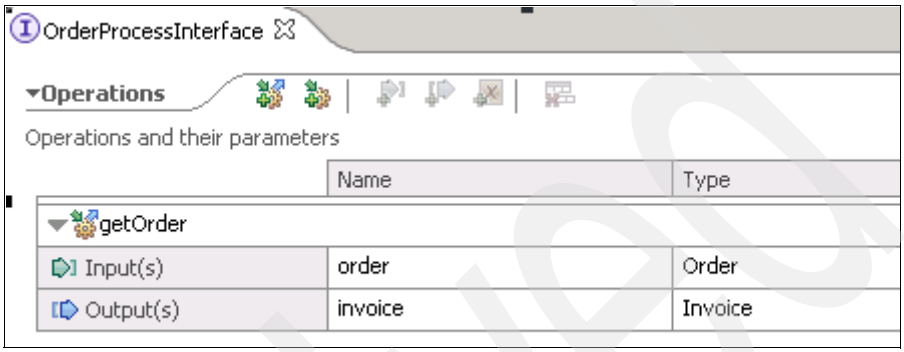


Figure 9-10 OrderProcessInterface

The second interface, SendInterface (Figure 9-11), is the interface for the import component. It has a one-way operation that sends the order using the Order business object.

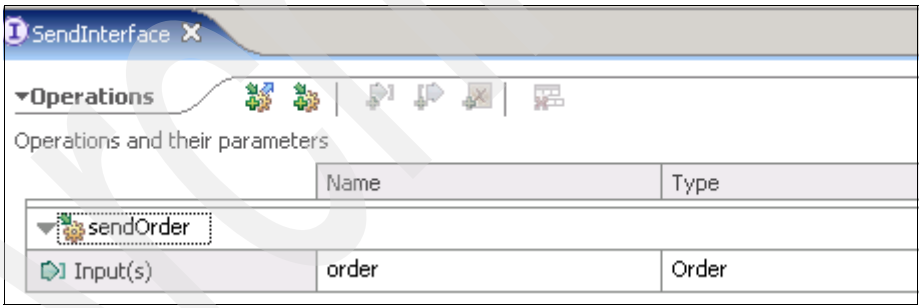


Figure 9-11 SendInterface

Archived

The third interface, ReceiveInterface (Figure 9-12), is the interface for the export component. It has one-way operation that receives the invoice using the Invoice business object.

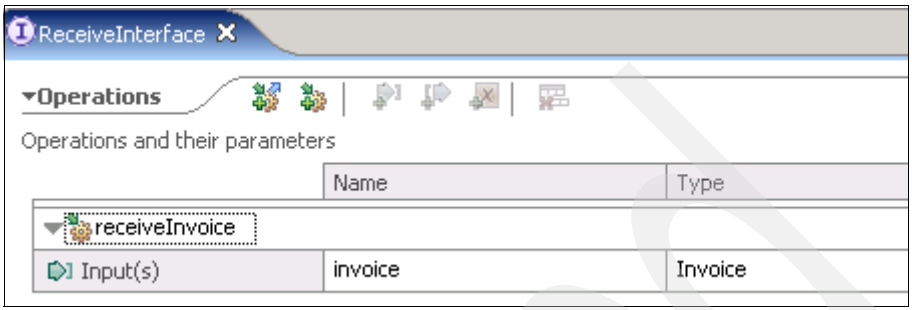


Figure 9-12 ReceiveInterface

9.5.4 Business process

The business process called OrderBusinessProcess has the interface OrderProcessInterface (Figure 9-10 on page 287). In the following sections, we describe the business process that is shown in Figure 9-13.

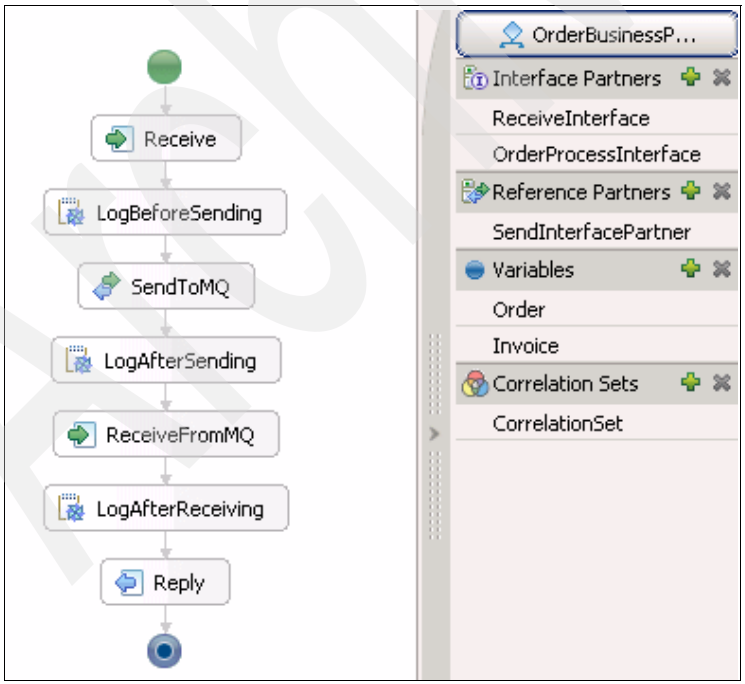


Figure 9-13 OrderBusinessProcess

This business process is a long-running process that receives a request from the client and sends it to a queue using an import. It sets a listener on another queue (a receiver queue) using an export component. When a reply message appears on the other queue, it is picked, and a reply is sent to the client.

Business process properties

The OrderBusinessProcess properties are set to make this a long-running process. See Figure 9-14.

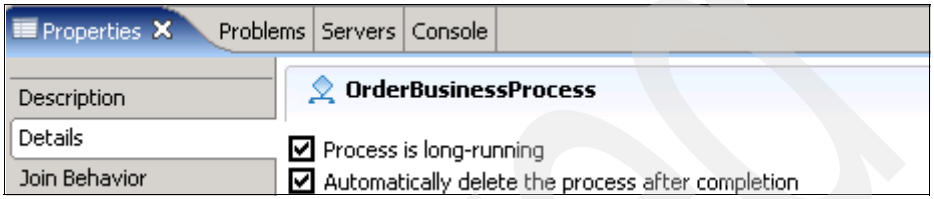


Figure 9-14 Defining the process as long-running

Interface and partner references

OrderBusinessProcess has two interface partners and one reference partner, which are listed to the right of the business process in Figure 9-13 on page 288.

An *interface partner* is the process interface and exposes operations that can be called by external partners. The OrderProcessInterface is defined as an interface partner (Figure 9-15) for the interaction between the business process and its clients.

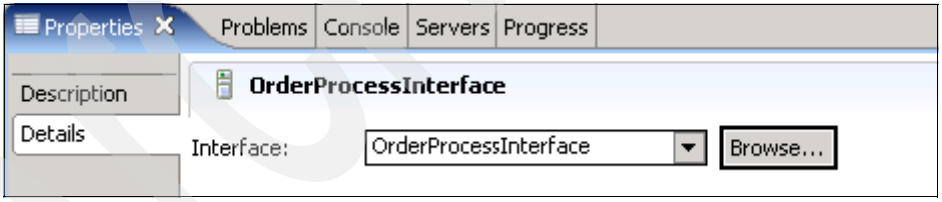


Figure 9-15 OrderProcessInterface interface partner for the business process

Archived

The ReceiveInterface is defined as an interface partner (Figure 9-16) to be used to receive incoming messages from a queue (OUTPUTQ) to the business process.

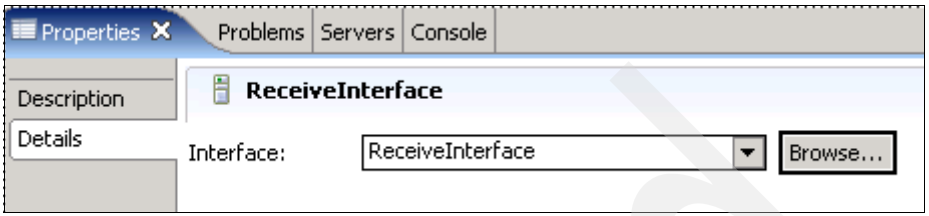


Figure 9-16 ReceiveInterface interface partner for the business process

Reference partners specify the interface that is used in the invocation of another service. In this scenario, the SendInterfacePartner is defined as a *reference partner* (Figure 9-17) and specifies that the SendInterface is to be used to call the service through sending a message to a queue (INPUTQ).

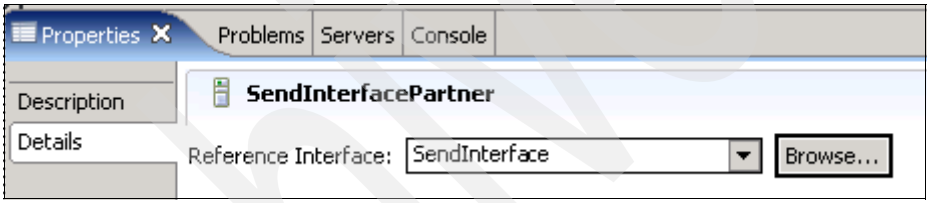


Figure 9-17 SendInterfacePartner reference partner

Correlation sets

Correlation sets are used to uniquely identify business processes using business data. If a message must be delivered to a business process, correlation sets are used to identify the particular process instance, with which the message is associated.

Archive

A correlation set called *CorrelationSet* is added to correlate the `OrderID` that is in the business objects used by SendInterface, ReceiveInterface, and OrderProcessInterface. See Figure 9-18.

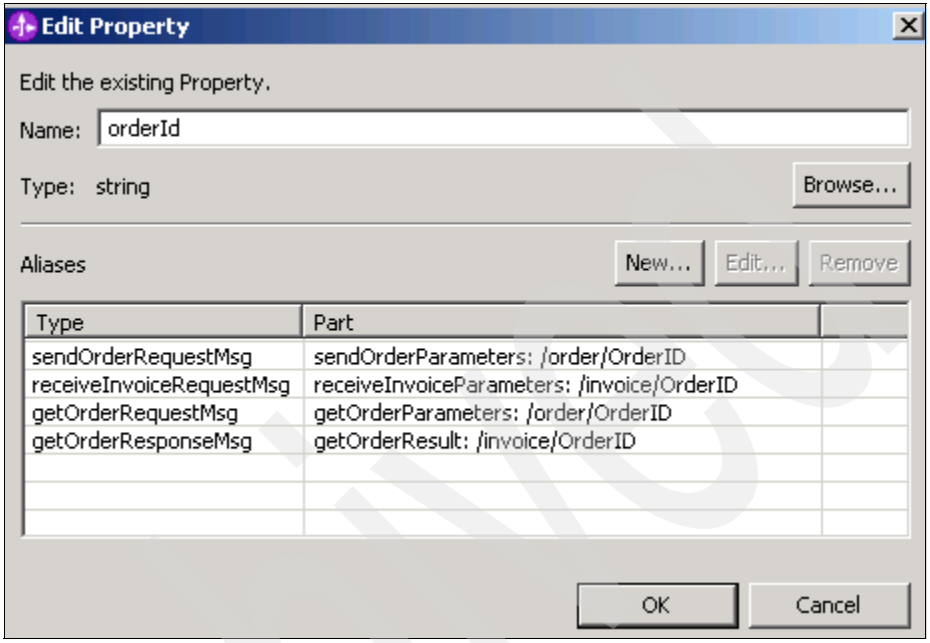


Figure 9-18 Correlation set

Receive activity

The Receive activity receives the input to the business process from a client. It is associated with OrderProcessInterface. See Figure 9-19.

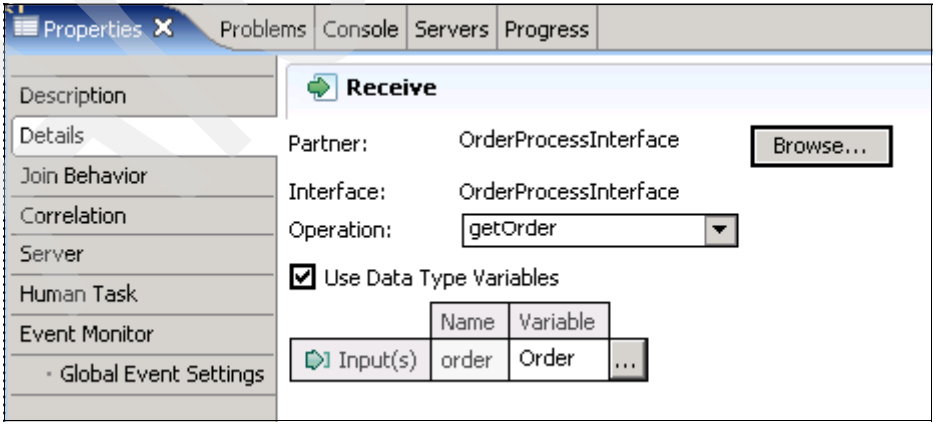


Figure 9-19 Receive activity

SendToMQ Invoke activity

The SendToMQ Invoke activity sends the message to a queue. It is associated with SendInterfacePartner, which means that a message is placed on the INPUTQ queue. See Figure 9-20.

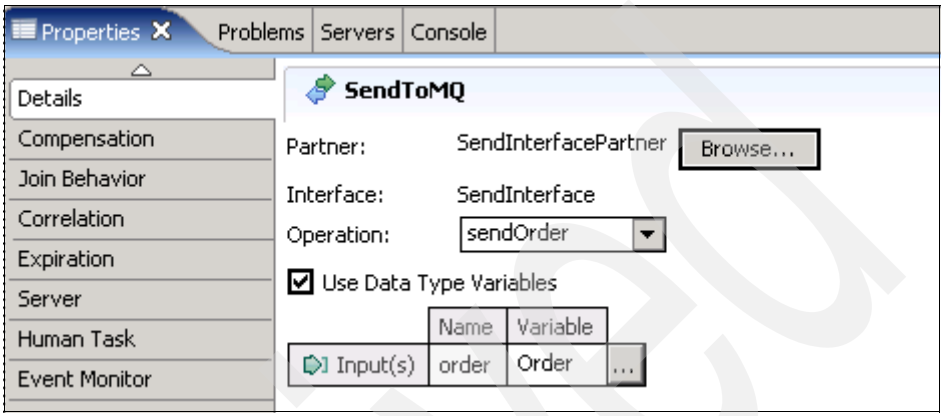


Figure 9-20 SendToMQ activity

The correlation set is added to the SendToMQ Invoke activity, which initiates the correlation set. See Figure 9-21.

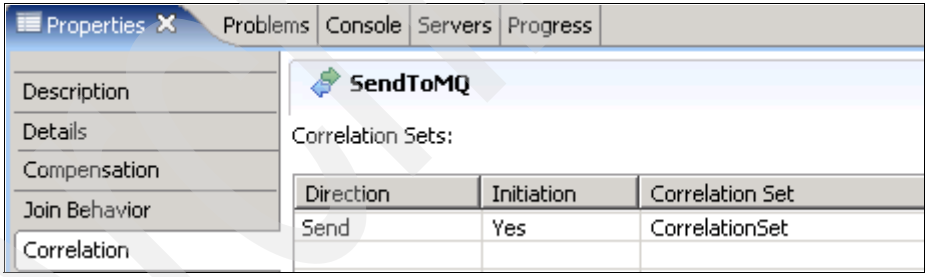


Figure 9-21 A correlation set is associated with the SendToMQ activity

Archive

ReceiveFromMQ Receive activity

The ReceiveFromMQ Receive activity receives a message from a queue. It is associated with ReceiveInterface, meaning that it receives the message from OUTPUTQ. See Figure 9-22.

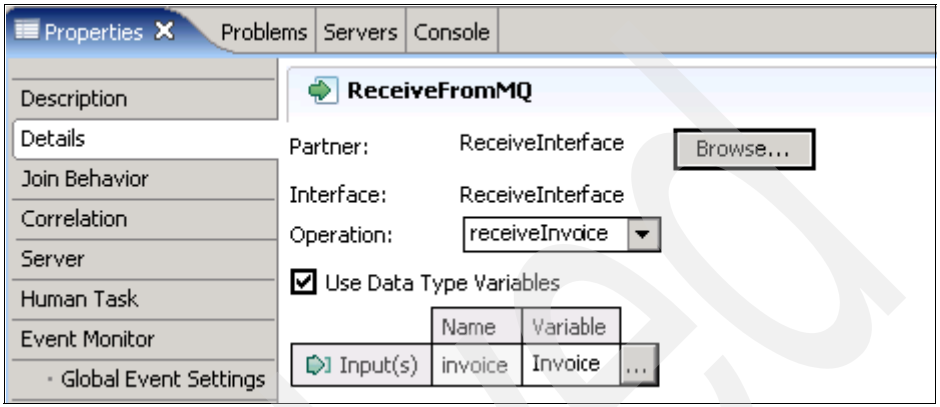


Figure 9-22 ReceiveFromMQ activity

The ReceiveFromMQ Receive activity is also associated with the correlation set to correlate the reply with the request message. See Figure 9-23.

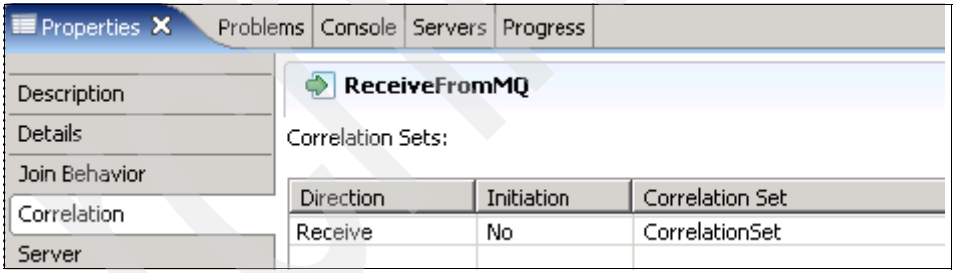


Figure 9-23 Correlation set associated with ReceiveFromMQ

Archive

Reply activity

The Reply activity returns the response to the business process client. See Figure 9-24.

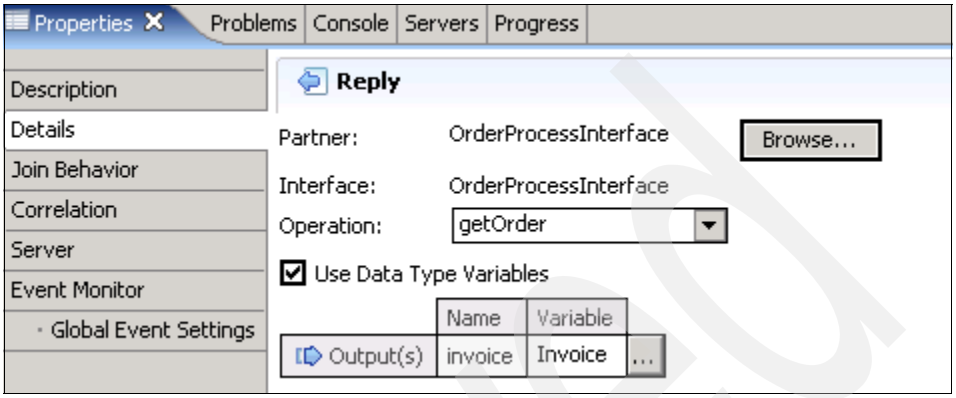


Figure 9-24 Reply activity

The business process is connected to the import component in the assembly diagram.

Java snippets

We included three basic Java snippets in the business process. These snippets simply write a timestamp entry to SystemOut log indicating its place in the process.

9.5.5 Import component and binding

The business process uses the SendOrderMQImport component to send an MQ message to a WebSphere MQ queue called INPUTQ. The message launches the message flow in WebSphere Message Broker. This import component has the SendInterface interface (Figure 9-11 on page 287).

Archived

The import has an MQ binding with the settings shown in Figure 9-25. These settings correspond to the WebSphere MQ configuration.

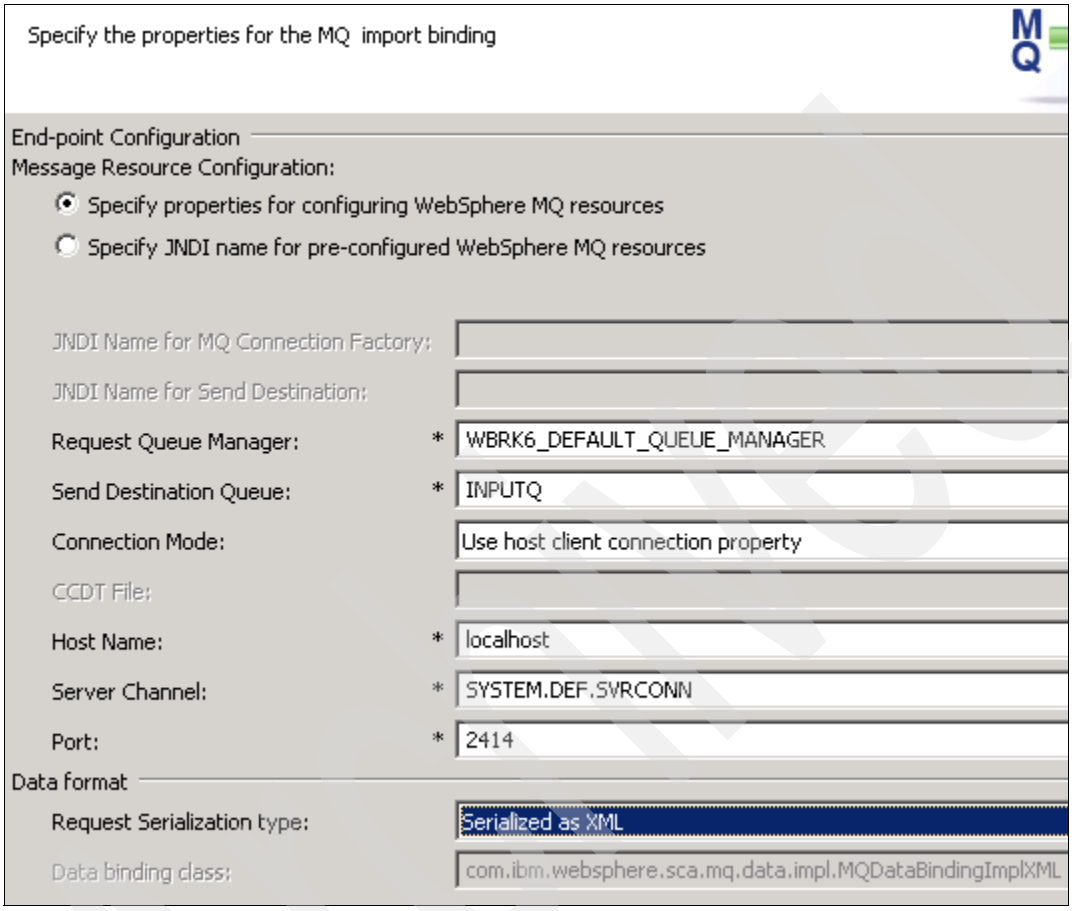


Figure 9-25 MQ import binding properties

Data bindings, specified when you generate the MQ binding, handle the transformation of data passed as an SDO in an SCA application and the native format in WebSphere MQ.

Four body data bindings can be specified for an MQ import using the Request Serialization type field. In this scenario, the request serialization type is set to `Serialized as XML`, meaning that the SDO will be serialized as XML before sending the message to the queue. This option has been chosen because the WebSphere Message Broker message flow is using an MQInput node that has an XMLNS parser.

For more information about serialization types available, see “MQ import and data bindings” on page 59.

9.5.6 Export component and binding

The OrderBusinessProcessExport export component is used by clients to access the business process. It was generated from the business process. The interface is OrderProcessInterface.

The ReceiveInvoiceMQExport export is used to get messages that are placed in the OUTPUTQ queue by the WebSphere Message Broker flow. ReceiveInterface (Figure 9-12 on page 288) is the interface. Figure 9-26 shows the MQ binding settings.

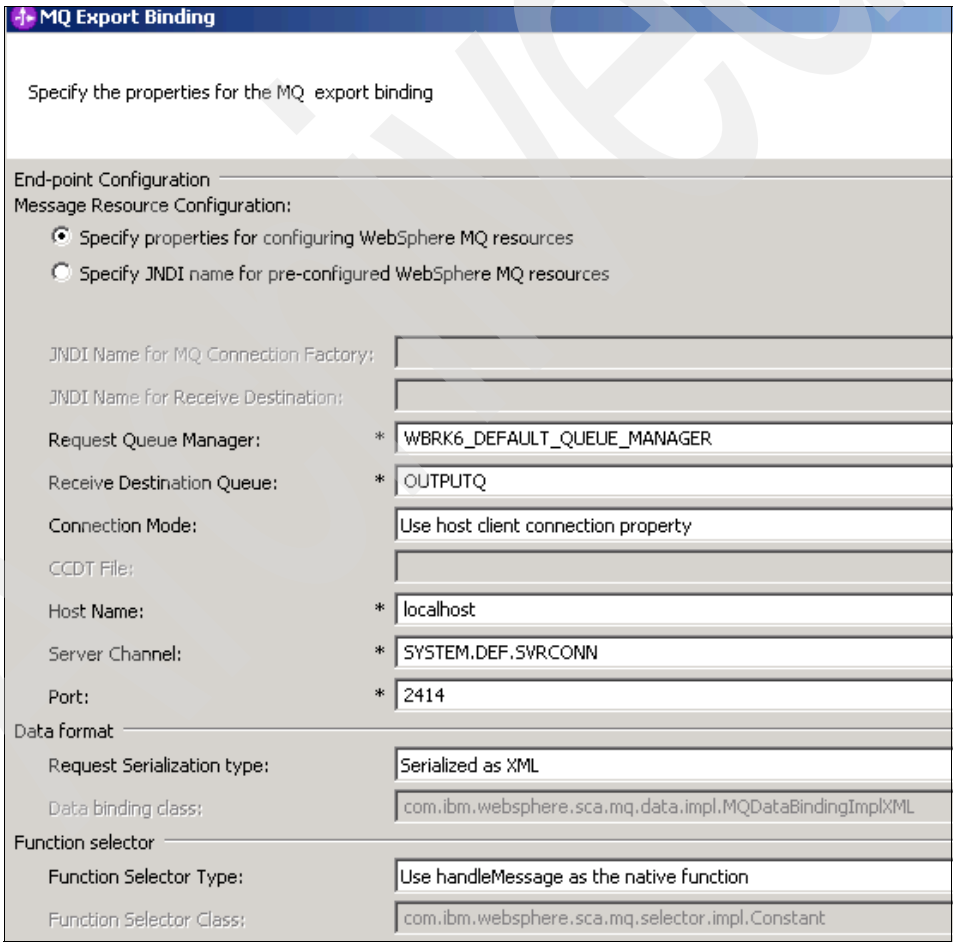


Figure 9-26 MQ bindings settings for ReceiveInvoiceMQExport

9.6 Message flow

In this section, we demonstrate the implementation of the message flows that are deployed on WebSphere Message Broker. The following message flows are used:

- OrderMessageFlow, which is the Original flow
- ErrorHandleFlow, which is a subflow for error/fault handling

9.6.1 OrderMessageFlow

The OrderMessageFlow message flow shown in Figure 9-27 follows this sequence:

1. Gets a message from a WebSphere MQ queue.
2. Performs a lookup for a Web service, specifying a value for a user property that is associated with the Web service in the registry (Figure 9-4 on page 282).
3. Saves the MQ header because it will be lost when the HTTP request to the Web service occurs.
4. Wraps the message in a SOAP envelope.
5. Makes a dynamic call to the Web service using SOAP/HTTP and gets the reply.
6. Extracts the message from the SOAP envelope.
7. Appends the MQ header.
8. Puts the output message to a queue.

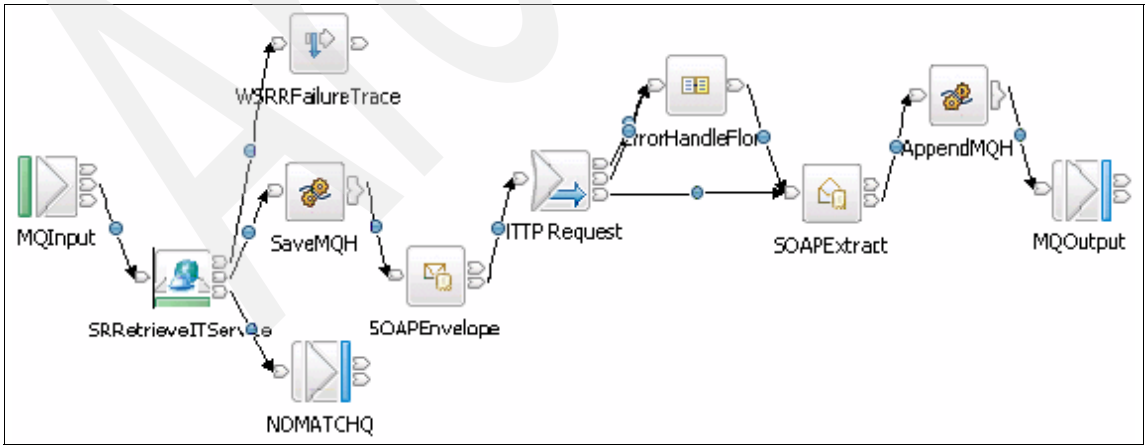


Figure 9-27 OrderMessageFlow

Archived

In the following sections, we show the settings for the nodes of the OrderMessageFlow message flow.

MQInput node

The basic settings (Figure 9-28) of the MQInput node define `INPUTQ` as the queue from which the message flow gets the messages.

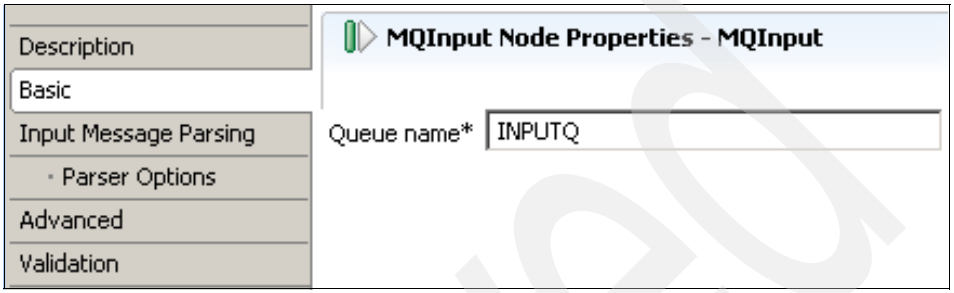


Figure 9-28 MQInput node

In the input message parsing properties (Figure 9-29), the message domain is set to `XMLNS`.

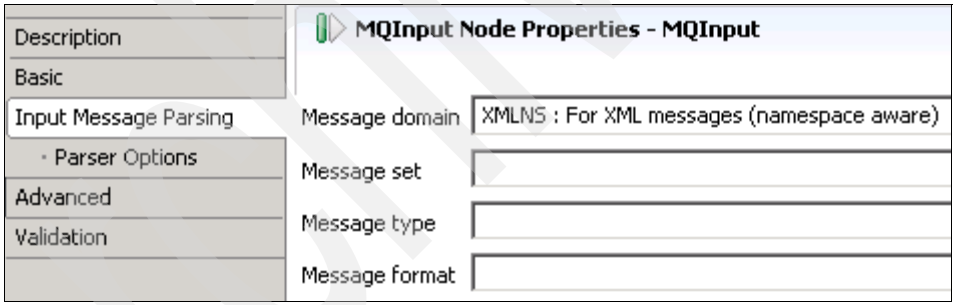


Figure 9-29 MQInput node message domain setting

XMLNS: The message domain is set to XMLNS, and the WebSphere Process Server sends XML namespaced messages.

Archived

SRRetrieveITService node

The SRRetrieveITService node (Figure 9-30) performs a lookup in WebSphere Service Registry and Repository for a service with the following characteristics:

- A PortType name of `OrderPortType`
- Port user properties of `QualityOfService` with a value of `Moderate` (created in Figure 9-4 on page 282)
- A Match Policy equal `One` to return only one service match

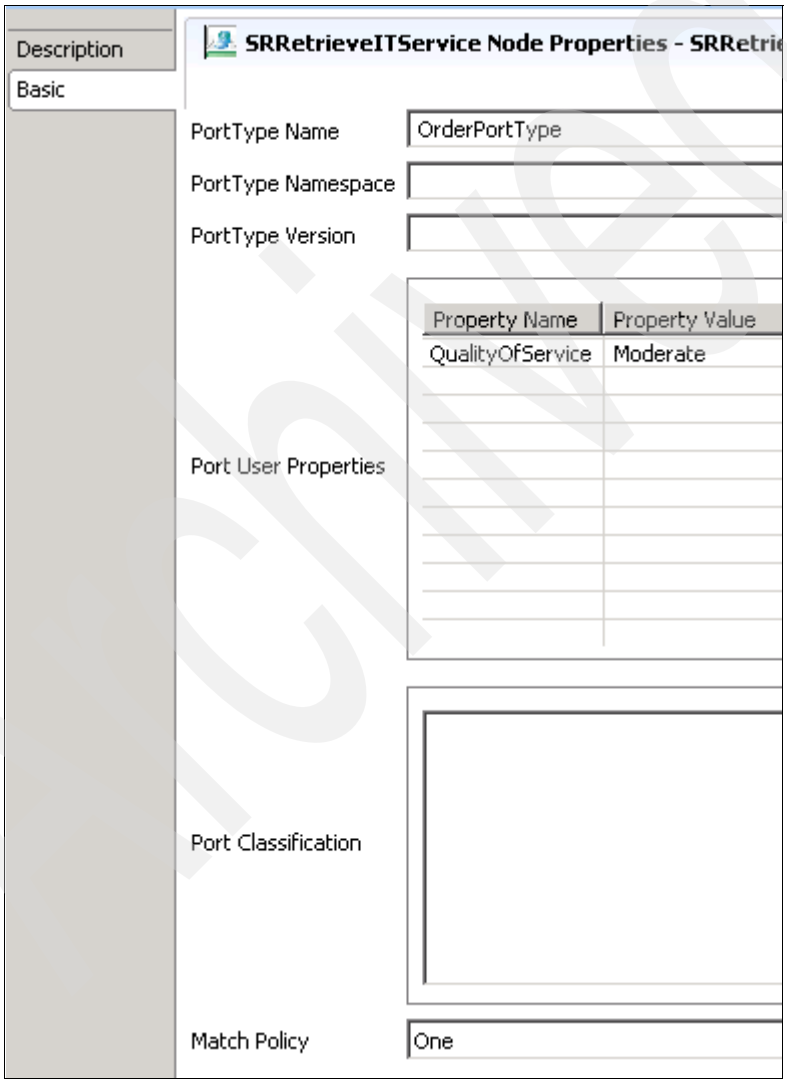


Figure 9-30 SRRetreiveITService node properties

Archived

Wildcard search: The PortType Name attribute in the SRRetrieveITService node does not support a wildcard search criteria, for example, `Order*`. Meanwhile the Query wizard in WebSphere Service Registry and Repository Web user interface does support wildcard search.

Upon successful retrieval of a Web service that matches the specified metadata, the node set the RequestURL in the LocalEnvironment with the Web service endpoint as shown in Figure 9-31.

Figure 9-31 RequestURL set by SRRetrieveITService node

NOMATCHQ node

NOMATCHQ (Figure 9-32) is an MQOutput node that routes the input message to the NOMATCHQ queue if no entity is found in the registry based on the specified values. The flow will end, and the support team should monitor the NOMATCHQ queue and take the appropriate action.

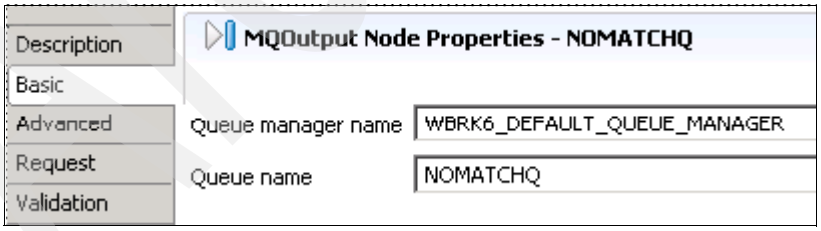


Figure 9-32 NOMATCHQ node properties

Archived

SaveMQH Compute node

When a message is sent using HTTP, the MQ header in the message is destroyed. The SaveMQH Compute node saves the MQMD header in the Environment to preserve it (Example 9-2).

Example 9-2 Saving the MQMD header

```
CREATE COMPUTE MODULE OrderFlow_SaveMQH
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyEntireMessage();
      SET Environment.MQMD = InputRoot.MQMD;
      RETURN TRUE;
   END;
   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

SOAPEnvelope node

The SOAPEnvelope node (Figure 9-33) creates a new SOAP message that wraps the XML message in order to be sent to the HTTPRequest node.

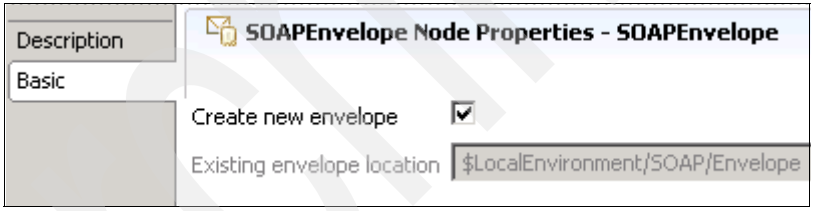


Figure 9-33 SOAPEnvelope node

Archived

HTTPRequest node

The Web service URL (Figure 9-34) in the HTTPRequest node contains a dummy URL. The correct URL is set by using the dynamic registry lookup.

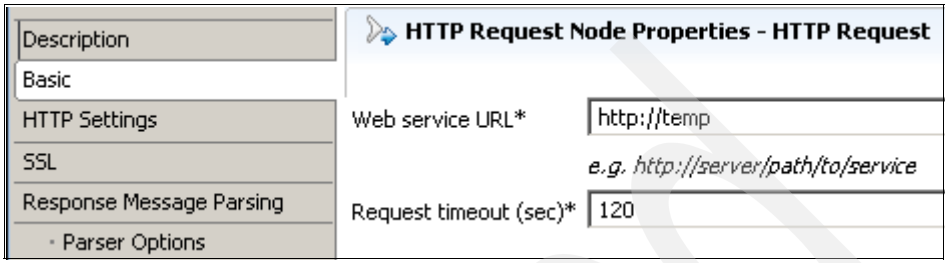


Figure 9-34 HTTP Request node basic properties

Figure 9-35 shows the response message parsing settings.

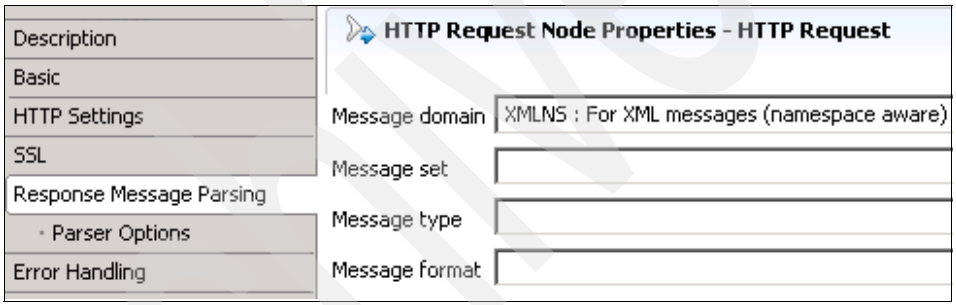


Figure 9-35 HTTP Request node response message parsing settings

In the Error Handling section shown in Figure 9-36, notice that Replace input with error is not selected. The Error Message location is set to `Environment.WSError`. This setup is used to allow proper error handling by using ErrorHandleFlow.

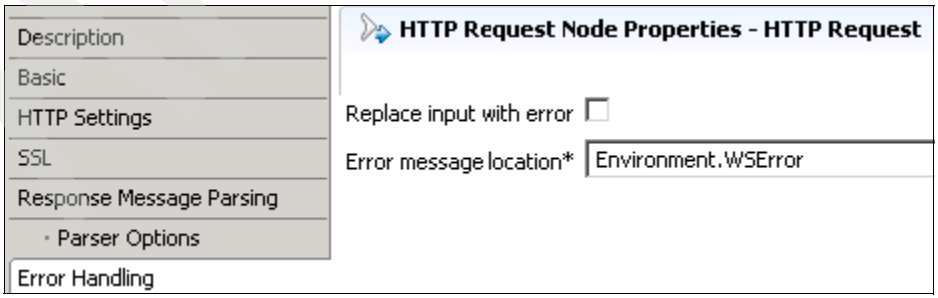


Figure 9-36 HTTP Request node error handling settings

ErrorHandleFlow

In case of error or failure in HTTP request, the message is fed to ErrorHandleFlow subflow, which performs a timeout and replay mechanism for the HTTP request. Refer to 9.6.2, “ErrorHandleFlow” on page 304, in which we discuss this flow.

SOAPExtract node

The SOAPExtract node (Figure 9-37) extracts the XML message from the SOAP envelope.

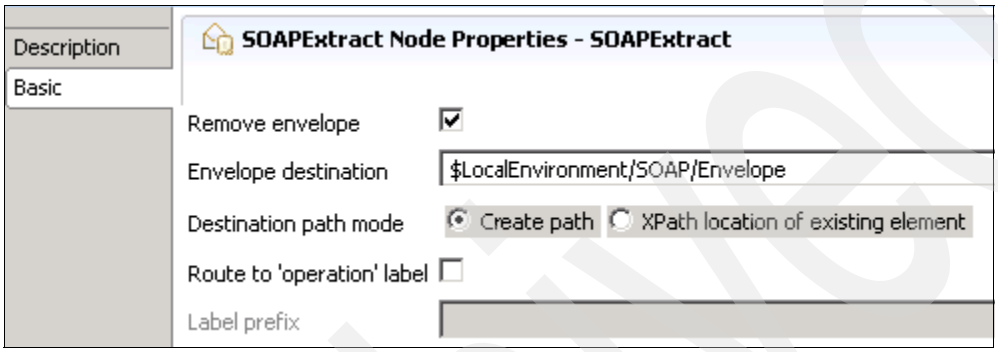


Figure 9-37 SOAPExtract node

AppendMQH node

The AppendMQH Compute node appends the MQMD header back to the message and remove HTTP header. See Example 9-3.

Example 9-3 Adding the MQMD header to the message

```
CREATE COMPUTE MODULE OrderFlow_AppendMQH
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      -- Output is MQ, so remove HTTP headers
      SET OutputRoot.HTTPResponseHeader = NULL;
      -- Set the Properties
      SET OutputRoot.Properties = InputRoot.Properties;
      -- Create an MQMD and restore it to saved values
      CREATE NEXTSIBLING OF OutputRoot.Properties DOMAIN 'MQMD';
      SET OutputRoot.MQMD = Environment.MQMD;
      SET OutputRoot.XMLNS = InputRoot.XMLNS;
      RETURN TRUE;
   END;
```

Archived

MQOutput node

The MQOutput node queue manager is `WBRK6_DEFAULT_QUEUE_MANAGER` and the output queue is `OUTPUTQ`, as shown in Figure 9-38.

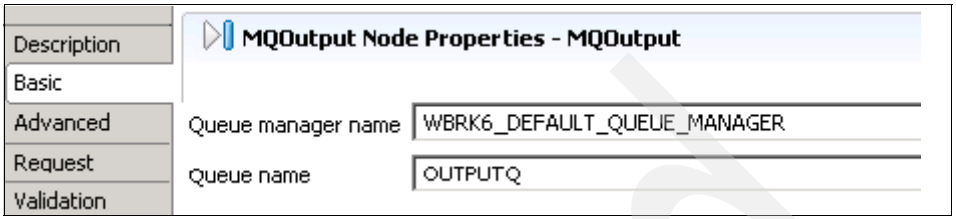


Figure 9-38 MQOutput node

9.6.2 ErrorHandleFlow

The ErrorHandleFlow message flow performs a replay mechanism on the HTTP request message for a certain amount of time with a specific delay between each trial as shown in Figure 9-39.

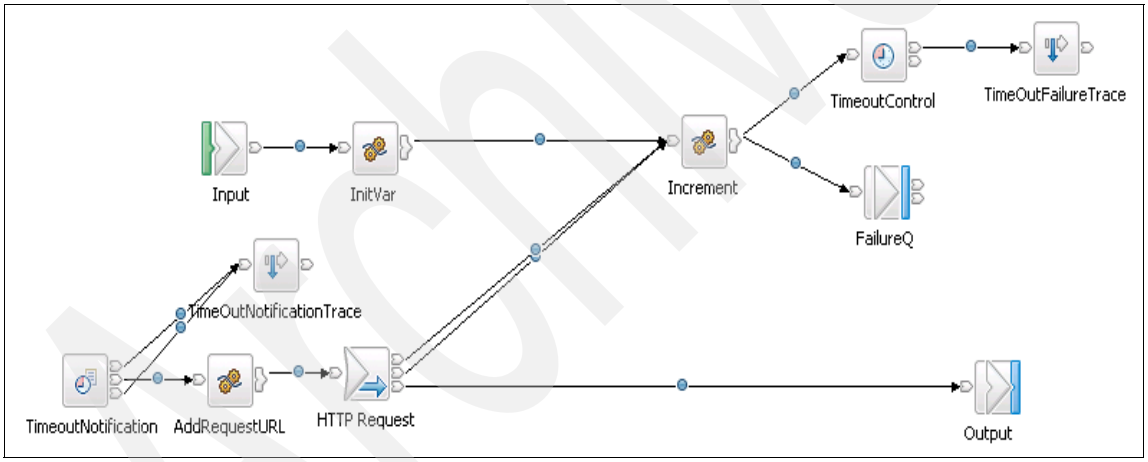


Figure 9-39 ErrorHandleFlow

This flow contains the nodes that are described in the following sections.

Archive

InitVar node

The InitVar Compute node saves the counter and the RequestURL (set by the SRRetrieveITService node) to the SOAP Envelope header (Example 9-4).

Example 9-4 Saving the counter and request URL

```
CREATE COMPUTE MODULE ErrorHandleFlow_InitVar
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyEntireMessage();
      --SET Environment.Counter = 0;
      SET OutputRoot.XMLNS.soapenv:Envelope.soapenv:Header.Counter = 0;
      SET OutputRoot.XMLNS.soapenv:Envelope.soapenv:Header.RequestURL =
InputLocalEnvironment.Destination.HTTP.RequestURL;
      RETURN TRUE;
   END;
   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

Important: The counter and the RequestURL are saved in the SOAP Envelope header and not the Environment because the TimeoutNotification node preserves the message and TimeoutRequest only.

Increment node

The Increment Compute node checks the number of retries. If the counter exceeds the maximum number of retries, the output is propagated to a Compute node that puts the message in the FAILUIREQ. If the counter does not exceed the limit, it is incremented by one and the request timeout is set for the TimeoutNotification node. See Example 9-5.

Example 9-5 Checking and incrementing the retry counter

```
CREATE COMPUTE MODULE ErrorHandleFlow_Increment
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyEntireMessage();

      -- Remove the HTTP response header because its presence causes
problem in TimeOut control processing
      SET OutputRoot.HTTPResponseHeader = NULL;

      -- Check the number of retries
```

Archived

```
      IF InputRoot.XMLNS.soapenv:Envelope.soapenv:Header.Counter <
MAXRetries THEN
         SET OutputRoot.XMLNS.soapenv:Envelope.soapenv:Header.Counter =
CAST(InputRoot.XMLNS.soapenv:Envelope.soapenv:Header.Counter AS INTEGER
ENCODING InputRoot.Properties.Encoding) + 1;
      ELSE
         PROPAGATE TO TERMINAL 'out1';
         RETURN FALSE;
      END IF;

      -- Create the delay interval that will be between each http
request
      DECLARE DELAYINTERVAL INTERVAL;
      SET DELAYINTERVAL = CAST(delay AS INTERVAL SECOND);
      DECLARE DELAYTIMESTAMP TIMESTAMP CURRENT_TIMESTAMP +
DELAYINTERVAL;

      -- Set the time out request that will be used by TimeOut
Notification
      SET Environment.TimeoutRequest.Action = 'SET';
      SET Environment.TimeoutRequest.Identifier = 'RETRY';
      SET Environment.TimeoutRequest.StartDate = CAST(DELAYTIMESTAMP AS
DATE);
      SET Environment.TimeoutRequest.StartTime = CAST(DELAYTIMESTAMP AS
TIME);

      SET Environment.TimeoutRequest.Count = 1;
      SET Environment.TimeoutRequest.IgnoreMissed = TRUE;
      SET Environment.TimeoutRequest.AllowOverwrite = TRUE;

      RETURN TRUE;
   END;
   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

Example 9-6 shows declaring the variables at the beginning of the file.

Example 9-6 Declaring the variables

```
DECLARE MAXRetries EXTERNAL CONSTANT INTEGER 2;
DECLARE delay EXTERNAL CONSTANT INTEGER 5;
Declare soapenv NAMESPACE 'http://schemas.xmlsoap.org/soap/envelope/';
```

Archived

FAILUREQ node

The FAILUREQ MQOutput node (Figure 9-40) routes a message that reached its maximum retries to the FAILUREQ queue and the flow ends. The support team should monitor this queue and take the appropriate action.

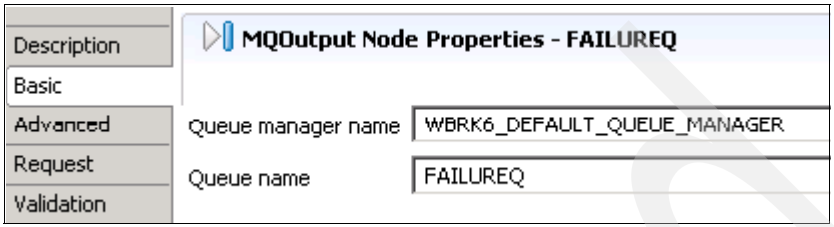


Figure 9-40 FAILUREQ node

TimeoutControl node

The TimeoutControl node has the basic property settings that are shown in Figure 9-41.

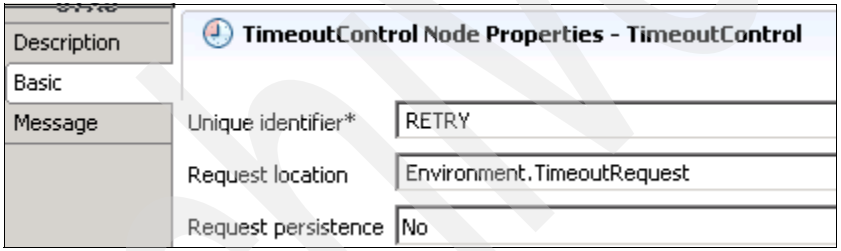


Figure 9-41 TimeoutControl node

Note: The values for the Unique Identifier and Request location fields are the same as the values set in the Increment Compute node.

Archive

TimeoutNotification node

The TimeoutNotification node is paired with the TimeoutControl node. It processes timeout request messages that are sent by the TimeoutControl node and propagates a copy of the messages to the next node in the message flow.

The TimeoutNotification node has the basic property settings that are shown in Figure 9-42.

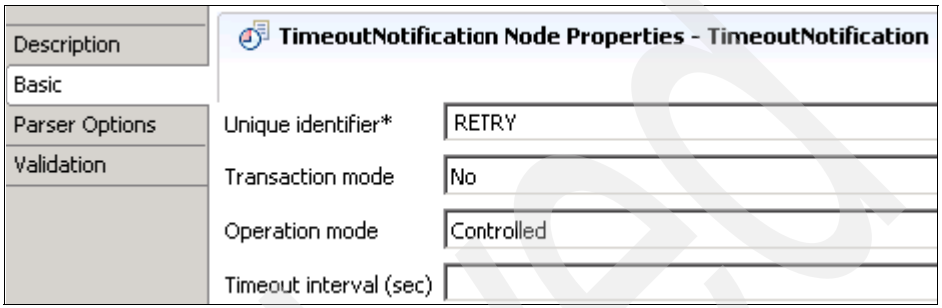


Figure 9-42 TimeoutNotification node

AddRequestURL Compute node

The AddRequestURL Compute node adds the request URL and MQMD header back to the message (Example 9-7). The request URL is needed for the HTTPRequest node that follows. The MQMD header is needed to put the message on a queue. In the case of a failure, this is the FAILUREQ. In the case of success, this is the OUTPUTQ.

Example 9-7 Adding the request URL and MQMD back to the message

```
CREATE COMPUTE MODULE ErrorHandleFlow_AddRequestURL
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyEntireMessage();
      SET OutputLocalEnvironment.Destination.HTTP.RequestURL =
InputRoot.XMLNS.soapenv:Envelope.soapenv:Header.RequestURL;
      SET Environment.MQMD = InputRoot.MQMD;
      RETURN TRUE;
   END;
   CREATE PROCEDURE CopyEntireMessage() BEGIN
      SET OutputRoot = InputRoot;
   END;
END MODULE;
```

The Compute mode is set to `LocalEnvironment` and `Message` because the request URL is set in the LocalEnvironment (Figure 9-43 on page 309).

Archived

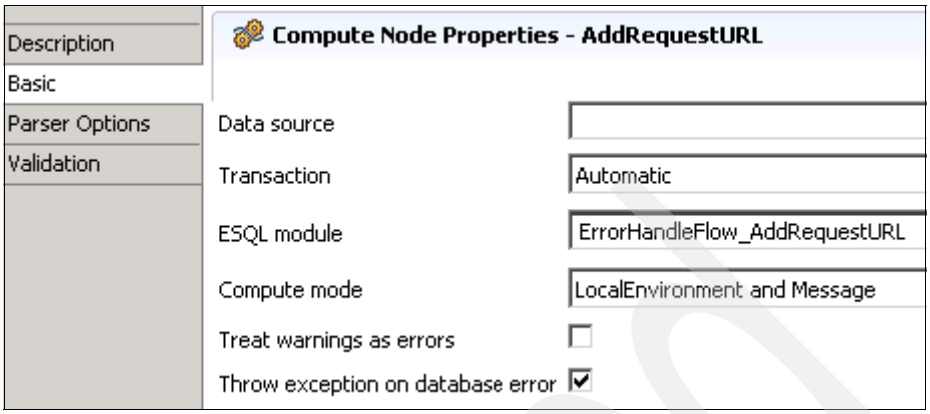


Figure 9-43 Compute node properties

HTTPRequest node

The HTTPRequest node has the same properties set as the HTTPRequest node in the caller flow (Figure 9-34 on page 302).

9.7 Testing the flow

In this section, we show the integration testing between the integration module and the message flow in WebSphere Integration Developer. We test the flow using the Integration Test Client. We start testing from the OrderBusinessProcessExport component (Figure 9-44).

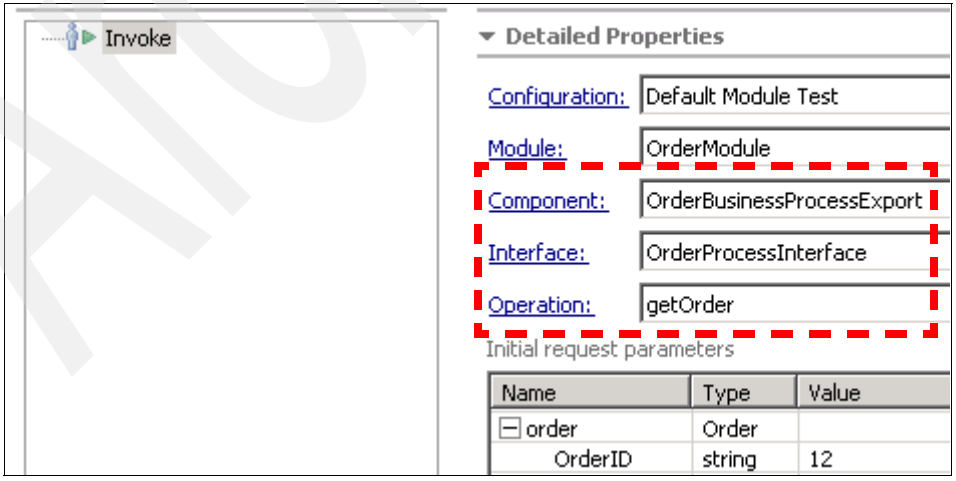


Figure 9-44 Testing the process

Archived

Figure 9-45 shows the results of the test.

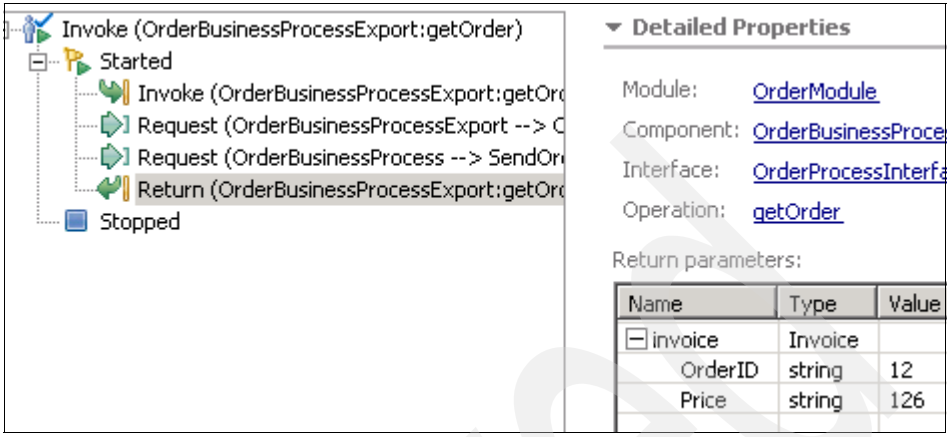


Figure 9-45 Scenario test results

Archived

10

Scenario: Connectivity to third-party JMS providers

In the scenario presented in this chapter, we demonstrate the following concepts:

- Broadcast mediation type

  The message flow performs simple static routing, taking a single message and routing it to two third-party Java Message Service (JMS) providers.

- One-way interaction pattern

  A one-way request is sent from the business process to the message flow.

- Connectivity using a JMS binding

  WebSphere Process Server uses an import component with a JMS binding to send a message to WebSphere Message Broker. A JMSInput node is used in WebSphere Message Broker to receive the message from the default messaging provider of WebSphere Process Server.

WebSphere Message Broker interacts with third-party JMS vendors using JMSOutput nodes.

© Copyright IBM Corp. 2008. All rights reserved.

10.1 Scenario overview

This scenario is that of a stock quote provider sending the latest stock prices to two different stock brokers. Figure 10-1 shows the basic flow of this scenario.

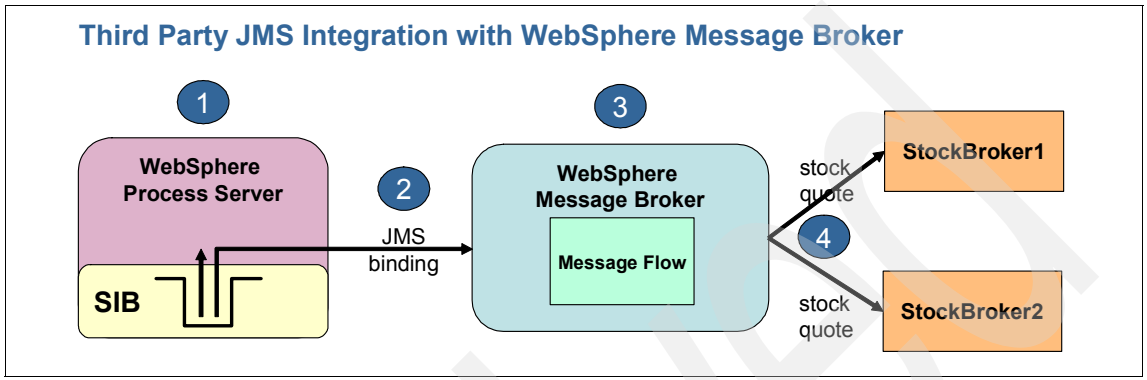


Figure 10-1 Scenario overview

We describe the highlights of this scenario, numbered in Figure 10-1, as follows:

1. WebSphere Process Server hosts a business process that sends a message to a message flow. The message contains a request for a stock quote.
2. The message is sent as a JMS message over an import component with a JMS binding. A destination queue defined on the WebSphere default messaging provider provides the transport. The request is a one-way request, so that no response is expected from the broker.
3. The delivery of the message triggers a message flow in WebSphere Message Broker to begin processing.
4. The message is delivered to two back-end services using third-party JMS providers.

Important: The JMSInput and JMSOutput nodes of WebSphere Message Broker are used in this scenario. These nodes are compatible with any JMS provider that conforms to the Java Message Service Specification, version 1.1. Ensure that the third-party JMS provider you are using complies with the version 1.1. This scenario uses SonicMQ v7.5 and SwiftMQ 7.0.2 as JMS providers.

Archived

10.2 Environment

This scenario was developed in an environment with the following components:

- WebSphere Integration Developer 6.0.2.2
- WebSphere Process Server 6.0.2.1
- WebSphere Application Server 6.0.2.19
- WebSphere Message Broker run time 6.0.0.5
- WebSphere Message Brokers Toolkit 6.0.2 (Interim Fix 006)
- WebSphere MQ 6.0.2.2
- Two third-party JMS providers, Progress SonicMQ and SwiftMQ, running in a Windows environment

10.3 WebSphere Process Server configuration

The items presented in the following sections are defined to WebSphere Process Server for this scenario. The configuration was done using the administrative console of WebSphere Process Server.

Archived

10.3.1 Service integration bus configuration

WebSphere Process Server uses the default messaging provider to host the queue that is used to send the JMS message to WebSphere Message Broker. In this scenario, the following configuration tasks were performed using the WebSphere administrative console:

1. A service integration bus called JMSBUS was created. The application server that hosts the business process (server1) was added to the bus as a member.
2. A queue destination called INPUTQ was defined on the bus (Figure 10-2).

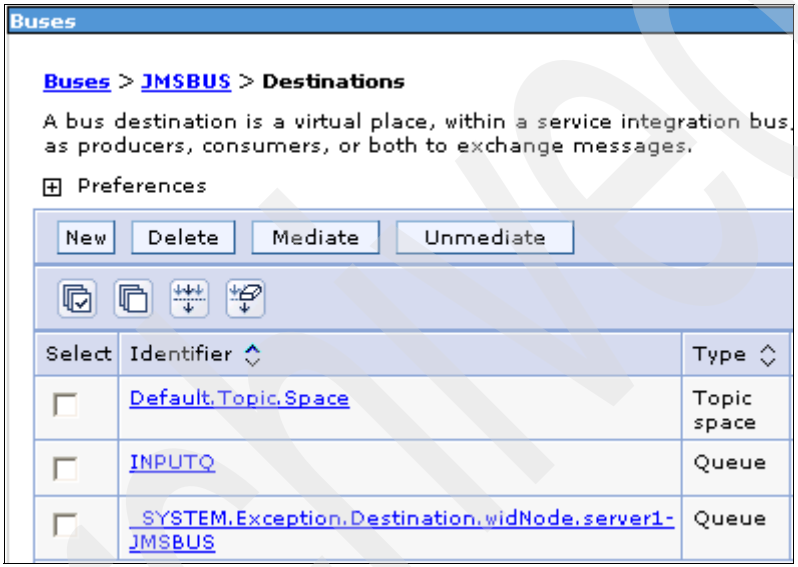


Figure 10-2 Queue destination on the service integration bus

Archived

10.3.2 JMS entries

In this scenario, JMS entries must be defined in WebSphere Process Server. The following configuration tasks were performed using the WebSphere administrative console:

1. A JMS queue connection factory for the default messaging provider was created as shown in Figure 10-3:
   - The queue connection factory is called `QCF`.
   - The Java Naming and Directory Interface (JNDI) name for the queue connection factory is `jms/QCF`.

   You can view JMS queue connection factories in the administrative console by navigating to **Resources** → **Default Messaging Provider** → **JMS Queue Connection Factory**.

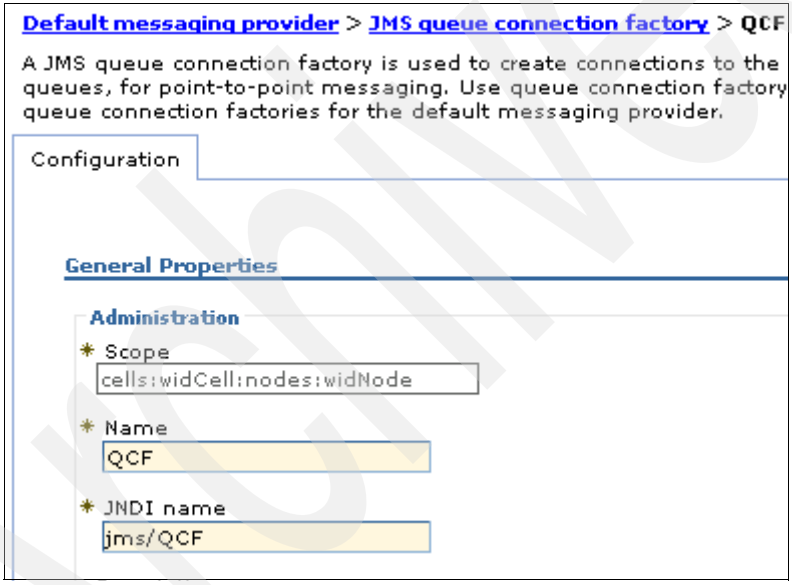


Figure 10-3 JMS queue connection factory

Archived

2. A JMS queue for sending messages was created as shown in Figure 10-4:
   - The JMS queue is associated with `INPUTQ` on `JMSBUS`.
   - The JNDI name is `jms/INPUTQ`.

   You can view JMS queue connection factories in the administrative console by navigating to **Resources** → **Default Messaging Provider** → **JMS Queue**.

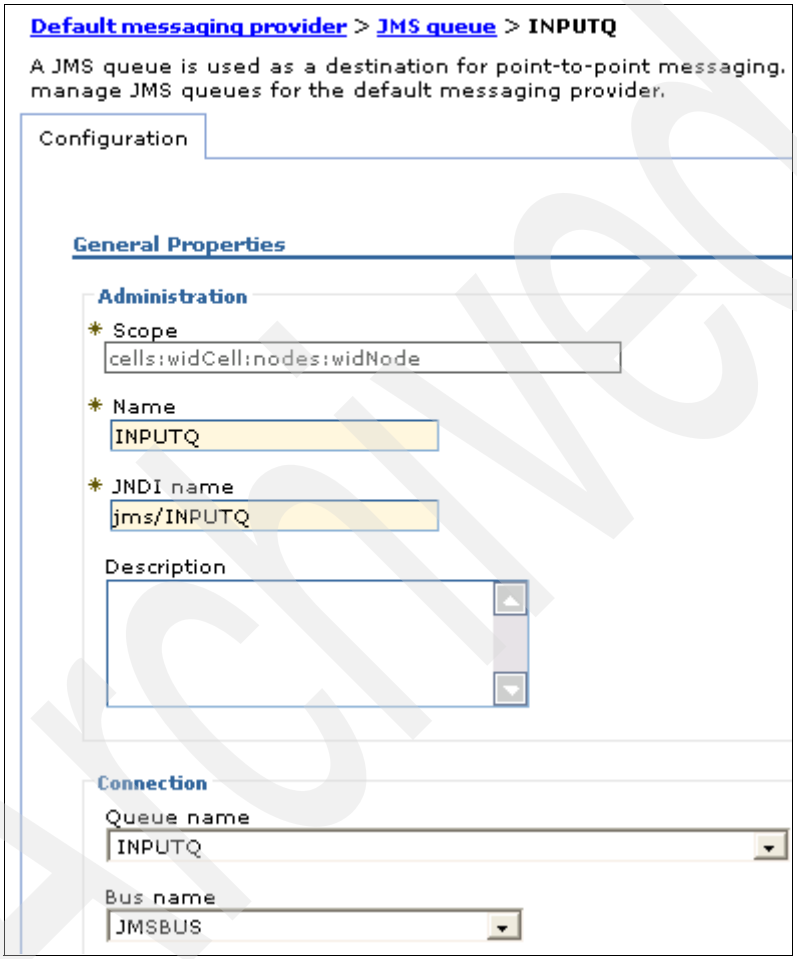


Figure 10-4 JMS queue definition

Archived

10.4 WebSphere Message Broker configuration

In order for a JMS node to connect to a JMS provider, you must copy the provider specific Java archive (JAR) files to the brokers' shared classes folder, so that the classes are available for the broker to load at run time. In this scenario, the JMS provider happens to be the underlying WebSphere default messaging provider.

The article "Connecting the JMS Transport Nodes for WebSphere Message Broker v6 to Popular JMS Providers" at the following Web address describes the steps that are required to configure a JMS node to various JMS providers:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0610_bicheno/0610_bicheno.html?ca=drs-

You do not have to do this if the JMS provider is WebSphere MQ and the JMS node is getting JMS messages from WebSphere MQ.

You must perform the following steps to make the WebSphere Process Server client classes available to WebSphere Message Broker:

1. Download and install the WebSphere Application Server JMS client, which is IBM Client for JMS on J2SE with IBM WebSphere Application Server on the Web at:

   http://www-1.ibm.com/support/docview.wss?uid=swg24012804

   When you install the client, specify the jms_jndi_ibm flag:

   ```
   java -jar sibc_install-<build>.jar jms_jndi_ibm
   c:\jmsnodes\wasclient\
   ```

   This command installs the client into c:\jmsnodes\wasclient\.

2. Copy the two JAR files, sibc.jms.jar and sibc.jndi.jar, from the c:\jmsnodes\wasclient\lib directory to the shared-classes folder of WebSphere Message Broker of C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes\.

Note: You can perform these configuration actions before or after the development of the module.

10.5 WebSphere MQ configuration

A queue manager for the broker called BRK6_DEFAULT_QUEUE_MANAGER is configured for the WebSphere Message Broker. No other configurations are required on the WebSphere MQ side.

10.6 Progress SonicMQ configuration

The following configuration was created on the Progress SonicMQ server using the Sonic Management console:

1. A management broker called *MgmtBroker* accepts connections on port 2506.
2. A queue called *jms_OUTQ* is configured on MgmtBroker using the default properties.
3. JMS administered objects were created for the queue (jms_OUTQ) and the queue connection factory (jms_QCF).

The SonicMQ JMS client classes are made available to WebSphere Message Broker by copying all *.jar files and native libraries for the SonicMQ JMS client from *Sonic_Install_Dir*\MQ7.5\lib to the shared-classes directory of WebSphere Message Broker. For Microsoft Windows, this is typically the C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes directory.

10.7 SwiftMQ configuration

The following steps were performed on the SwiftMQ server:

1. A router called *router2* with a JMSListener on port 4002 was configured.
2. A queue called *ITSOQ* was created on router2. The JNDI configured name for the queue is ITSOQ@router2. This was configured automatically when the queue was created.
3. The SwiftMQ JMS client classes were made available to WebSphere Message Broker by copying all *s.jar files and native libraries for the SwiftMQ JMS client from *Swift_Install_Dir*\jars to the shared-classes directory of WebSphere Message Broker. On a Windows system, this is typically the C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes directory.

   This step ensures that the Java class path for the JMS nodes is set correctly.

10.8 Business integration module

In this section, we demonstrate the implementation of a business integration module that is deployed on WebSphere Process Server. This module receives a message from the export component that triggers the short-running business process. The process sends a one-way JMS message to a queue that is configured in the default messaging system of WebSphere Process Server.

10.8.1 Business integration module components

The business process is implemented in a business integration module called *StockQuoteModule*. Figure 10-5 shows the assembly diagram for this module.

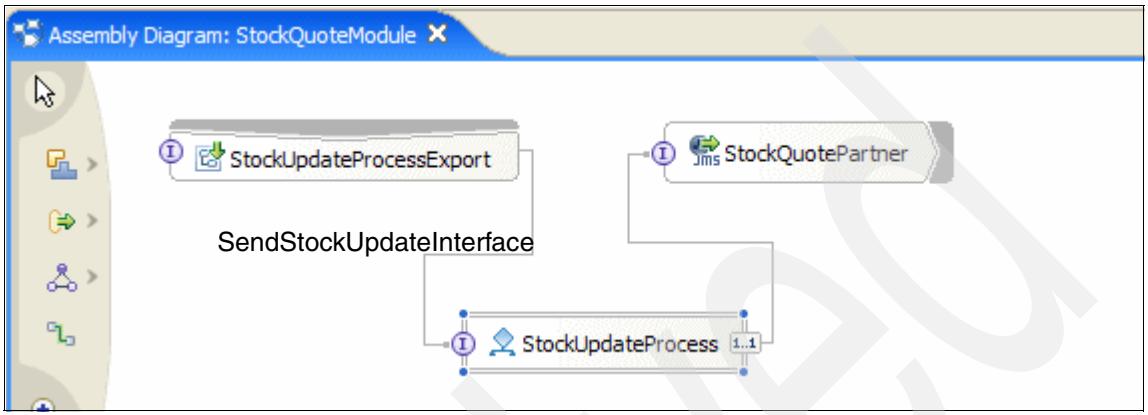


Figure 10-5 The StockQuoteModule

This business integration module consists of the following components:

- *StockUpdateProcessExport* is the export component that the client uses to access the business process.
- *StockUpdateProcess* contains the business process implementation.
- *StockQuotePartner* is the import component that sends request messages to a service via WebSphere Message Broker.

All three components use a single interface, SendStockUpdateInterface. This interface is defined in 10.8.3, “Interfaces” on page 321.

Archived

10.8.2 Business objects

The business integration module uses one business object called *StockQuoteBO*. The business object represents a price of a stock. StockQuoteBO, shown in Figure 10-6, has two string attributes, `StockName` and `Price`.

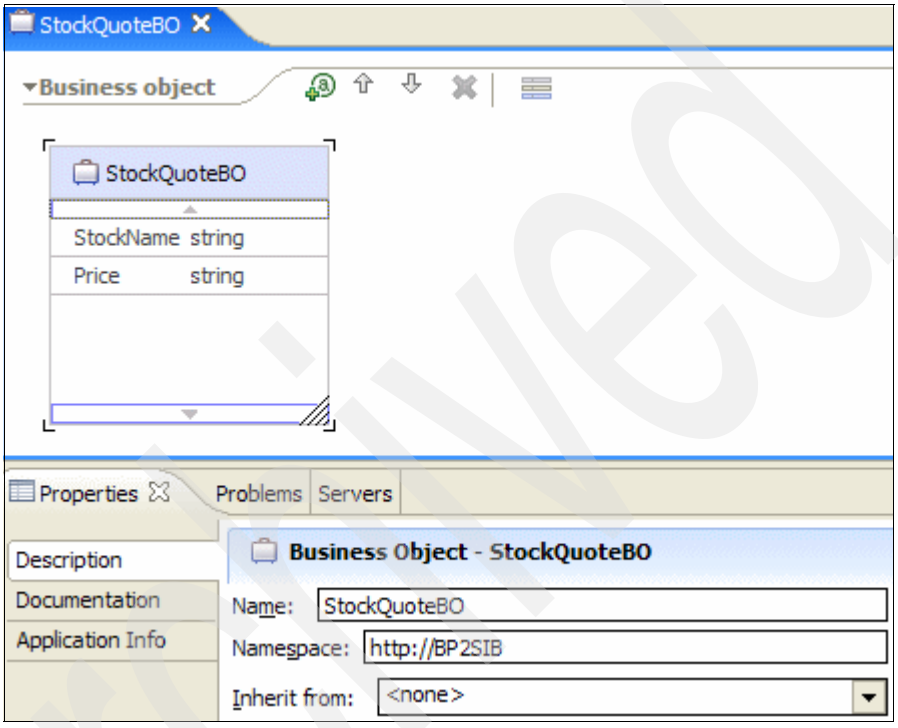


Figure 10-6 StockBO business object

Archived

10.8.3 Interfaces

The module uses one interface, SendStockUpdateInterface, which is shown in Figure 10-7.

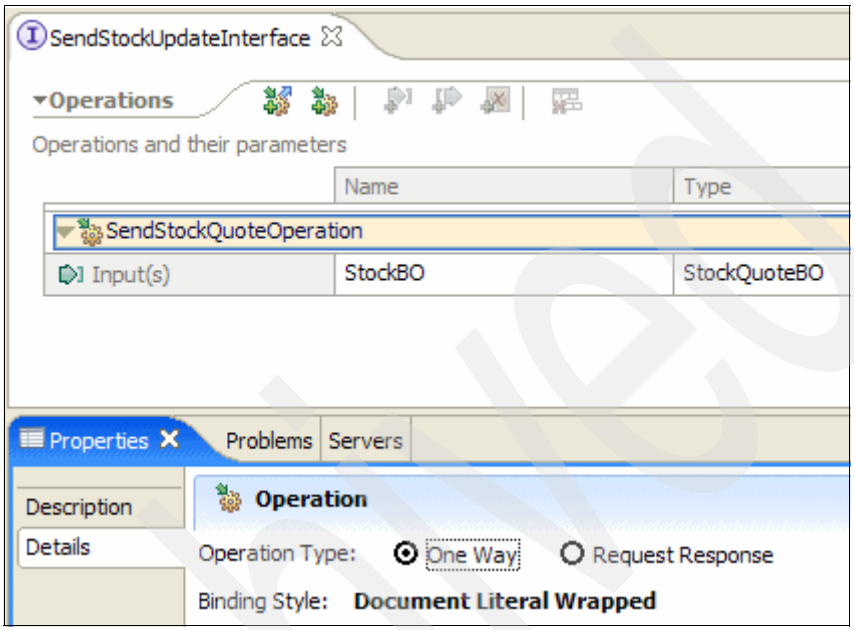


Figure 10-7 SendStockUpdateInterface

The interface has one operation called *SendStockQuoteOperation*. The input to the operation is a StockQuoteBO business object.

Archived

10.8.4 Business process

Figure 10-8 shows the StockUpdateProcess business process.

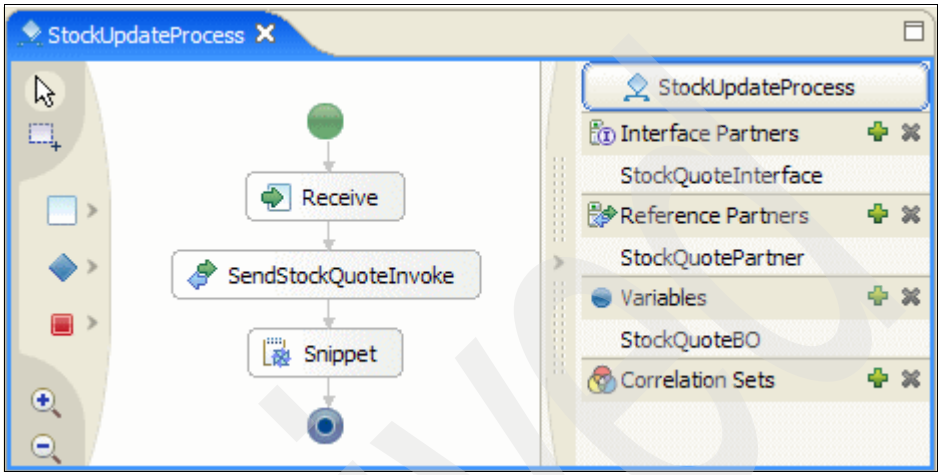


Figure 10-8 The StockUpdateProcess business process

The properties for the business process are the default properties.

Interface and partner references

The business process uses one interface partner and one reference partner. You can see these listed to the right of the business process in Figure 10-8.

An interface partner is the process interface and exposes operations that can be called by external partners. The interface partner for this process, StockQuoteInterface, uses the SendStockUpdateInterface interface.

Archive

Reference partners specify the interface that is used in the invocation of another service. In this scenario, StockQuotePartner is defined as a reference partner and specifies that the SendStockUpdateInterface interface is to be used to call the service (Figure 10-9).

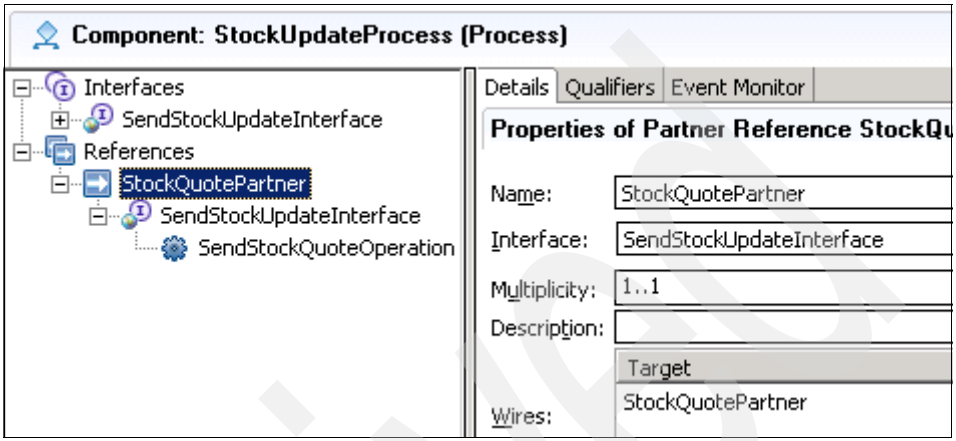


Figure 10-9 Business process reference

Keep in mind that both the interface partner and reference partner use the same interface, because just one operation is being invoked.

Receive activity

The Receive activity (Figure 10-10) receives the request from the client and is associated with the SendStockUpdateInterface.

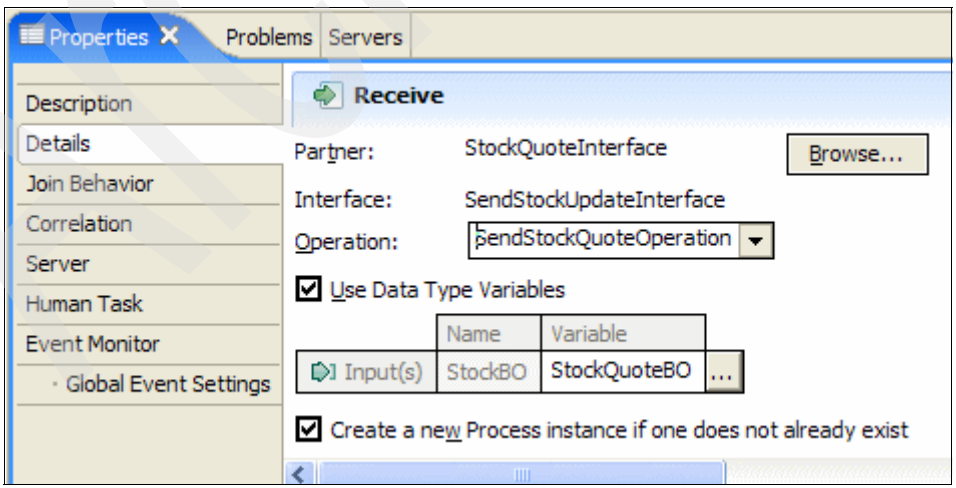


Figure 10-10 Receive activity

SendStockQuoteInvoke Invoke activity

The SendStockQuoteInvoke Invoke activity (Figure 10-11) is responsible for sending the JMS message to the default messaging provider. The Invoke activity is associated with StockQuotePartner. This partner reference is wired to the StockQuotePartner export component.

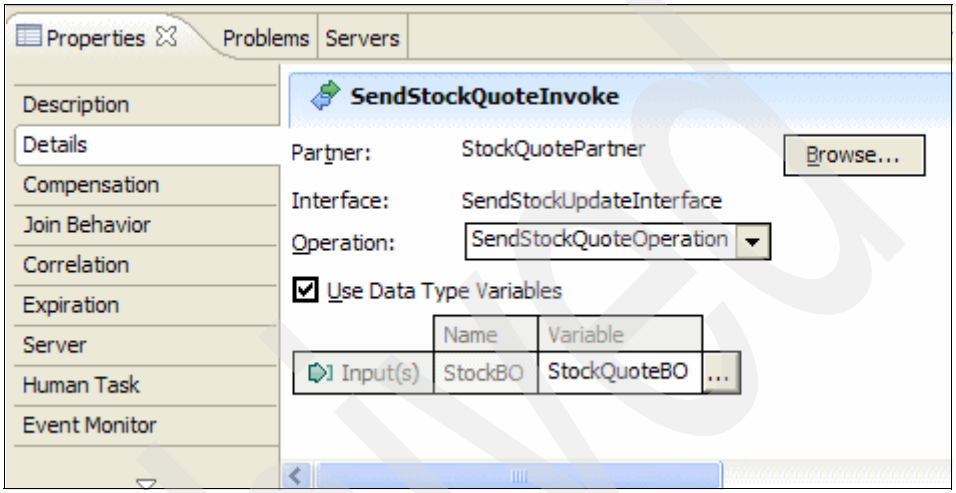


Figure 10-11 SendStockQuoteInvoke Invoke activity

Java snippet

A Java snippet (optional) component displays the results of the invoke operation as shown in Figure 10-12.

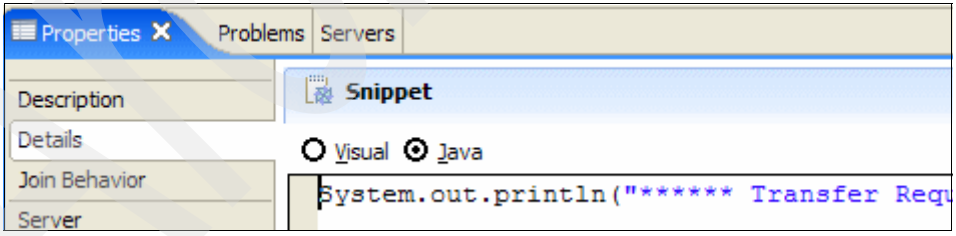


Figure 10-12 Java snippet

10.8.5 Import component and binding

The business process uses an import component to send a JMS message to WebSphere Message Broker. This import component has the SendStockUpdateInterface interface (Figure 10-7 on page 321). The import has a JMS binding with the settings shown in Figure 10-13.

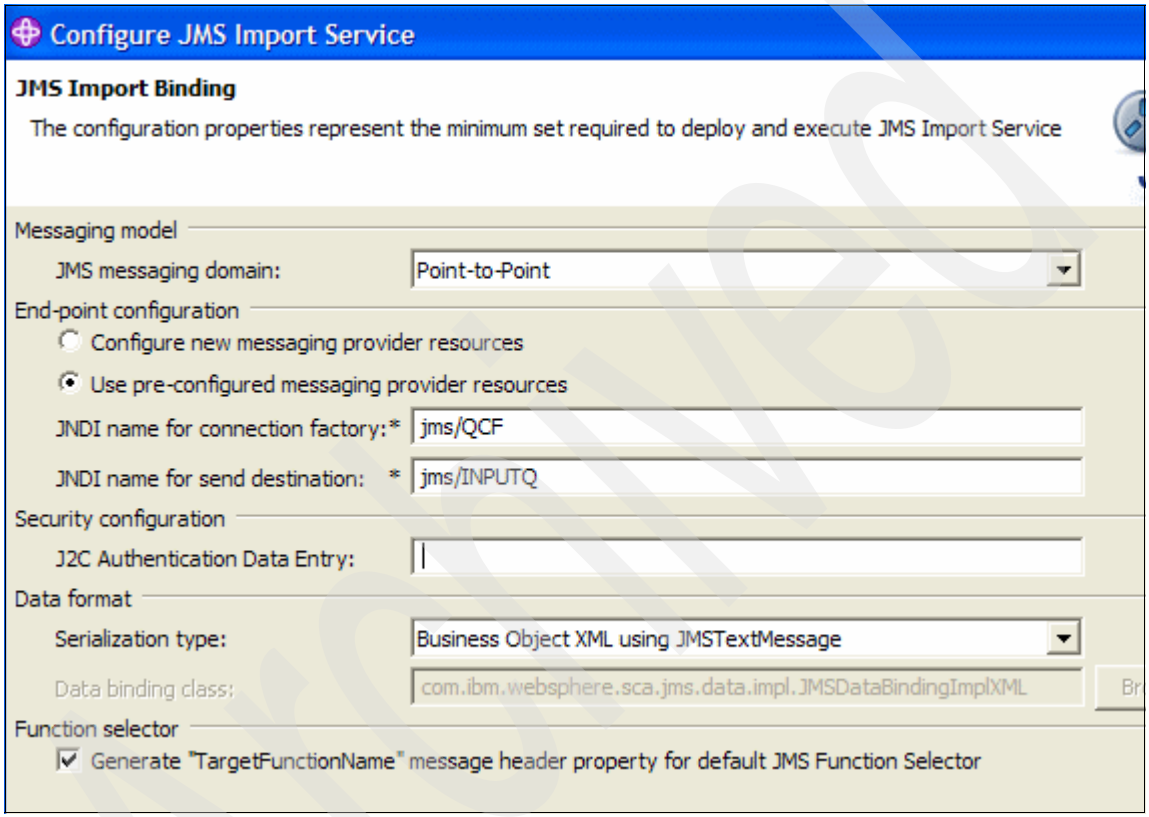


Figure 10-13 JMS binding on the import

The JMS binding has the following settings:

- For JMS messaging domain, select **Point-to-Point**.

  The only other choice here is publish-subscribe.

- For JNDI name for connection factory, type `jms/QCF`.

  This is the JNDI name of the queue connection factory for the default messaging provider (see Figure 10-3 on page 315).

- For JNDI name for send destination, type `jms/INPUTQ`.

  This is the JNDI name of the destination queue that is configured to the default messaging provider (see Figure 10-4 on page 316).

- For Serialization type, select **Business Object XML using JMSTextMessage**.

  This serialization type supports JMSTextMessage. It serializes the DataObject into an XML document that conforms to the schema of the DataObject and sets it to the text field of the JMSMessage.

  This class extends the general purpose data binding class `com.ibm.ws.sca.databinding.impl.DataBindingImplXML`, which provides the function to convert an XML string to a DataObject and vice versa.

  For more information about the various types of serialization options for JMS import bindings, see “JMS import data bindings” on page 68.

10.9 Message flow

In this section, we demonstrate the implementation of a message flow that is deployed on WebSphere Message Broker. The message flow gets its input from a queue that resides on the default messaging provider of WebSphere Process Server. It sends the message to two third-party JMS providers, SwiftMQ and Progress SonicMQ. See Figure 10-14.

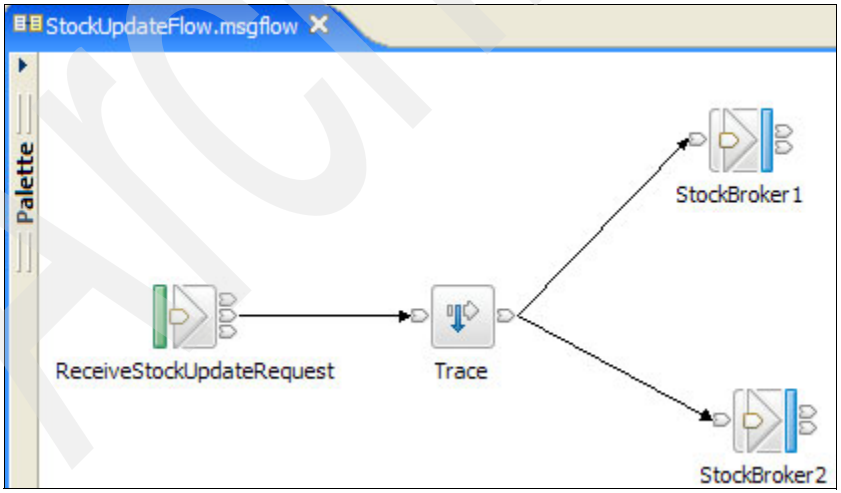


Figure 10-14 The WebSphere Message Broker flow

Archived

ReceiveStockUpdateRequest

The ReceiveStockUpdateRequest JMSInput node is responsible for pulling messages from the default messaging provider queue on the WebSphere Process Server. The JMS connection properties shown in Figure 10-15 must be configured.

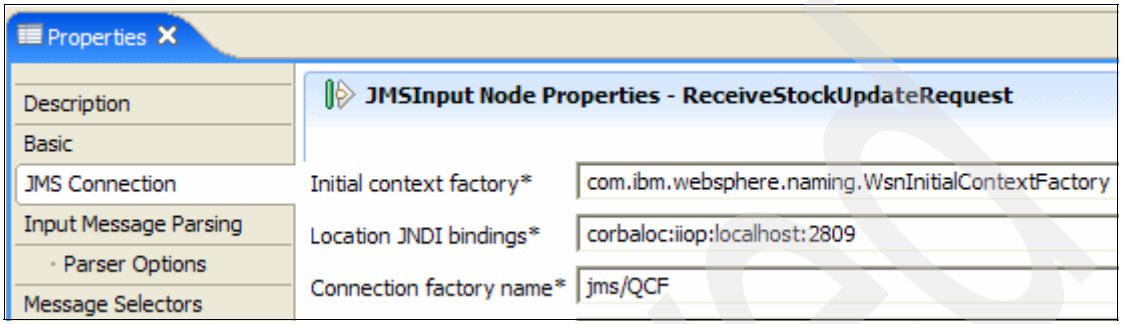


Figure 10-15 ReceiveStockQuoteRequest - JMS input node

The JMS connection properties require the following settings:

- For Initial context factory, type `com.ibm.websphere.naming.WsnInitialContextFactory`.

  A JMS application uses the initial context to obtain and look up the JNDI administered objects for the JMS provider. You must enter it exactly as shown here.

- For Location JNDI bindings, type `corbaloc:iiop:`*`hostname:port`*.

  This value points to the JNDI provider of the application server for WebSphere Process Server. The *hostname* is the host where the application server is executing. *Port* is the TCP/IP port number that the application server uses for the bootstrap address. This value is easily determined by using the WebSphere administrative console to display the port list on the application server configuration page.

- For Connection factory name, type `jms/QCF`.

  This is the JNDI name of the queue connection factory as defined in the default messaging provider. The name was defined in 10.3.2, “JMS entries” on page 315.

The basic properties (Figure 10-16 on page 328) are used to define the JNDI name of the queue from which the JMSInput node gets its message.

For Source queue, type `jms/INPUTQ`, which is the JNDI name of the queue. See 10.3.2, “JMS entries” on page 315.

Archive

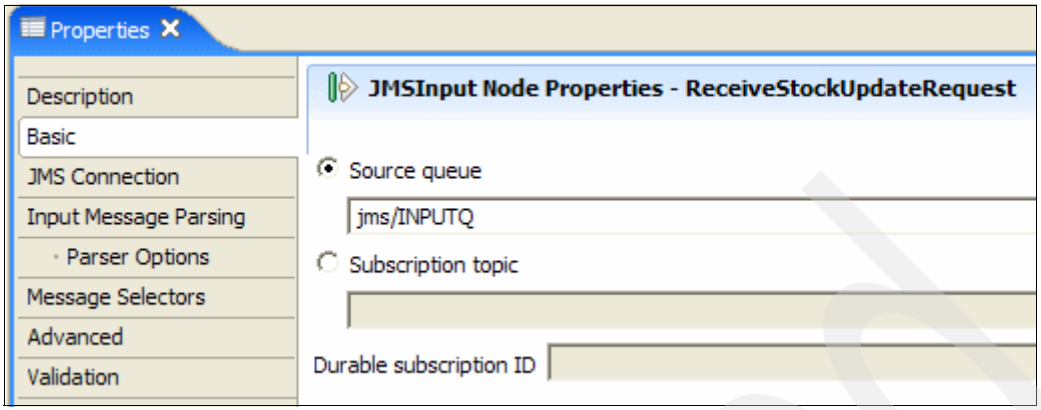


Figure 10-16 RecieveStockUpdateRequest: Basic tab

Trace node (optional)

The Trace node logs the incoming message for tracing purposes.

StockBroker1

The StockBroker1 JMSOutput node (Figure 10-17) is responsible for sending the message to the SonicMQ JMS provider.

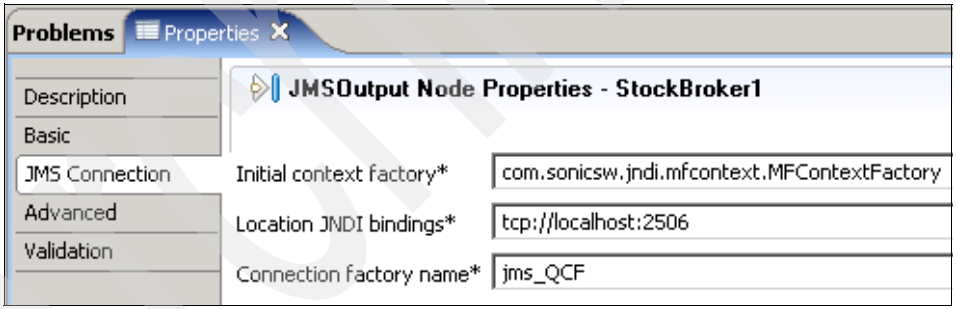


Figure 10-17 JMSOutput node configuration for SonicMQ

The JMS Connection properties require the following settings:

- For Initial context factory, type `com.sonicsw.jndi.mfcontext.MFContextFactory`.

  A JMS application uses the initial context to obtain and look up the JNDI administered objects for the JMS provider. You must enter this value exactly as shown here for SonicMQ.

- For Location JNDI bindings, type `tcp://hostname:2506`.

  This value points to the JNDI name of the JMS provider, and must have the syntax that is shown. The *hostname* is the host on which a SonicMQ management broker is listening, and the *port* is the TCP/IP port number. As described in 10.6, “Progress SonicMQ configuration” on page 318, the Management Broker accepts connections on port 2506.

- For Connection factory name, type `jms_QCF`.

  The JNDI name for the queue connection factory as defined in SonicMQ.

On the Basic properties tab, for Destination queue, type `jms_OUTQ` as shown in Figure 10-18. jms_OUTQ is the JNDI name for the queue as defined in SonicMQ.

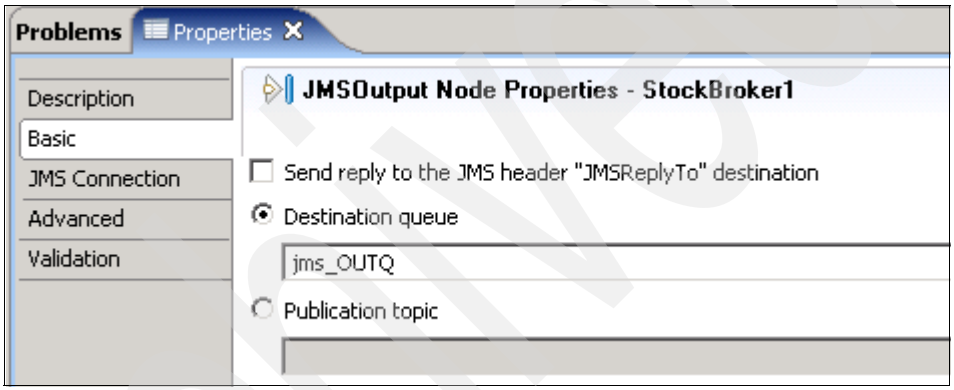


Figure 10-18 Destination queue for the JMSOutput node

StockBroker2

The StockBroker2 JMSOutput node is responsible for sending the message to the SwiftMQ JMS provider. Figure 10-19 shows the JMS connection properties for the JMSOutput node.

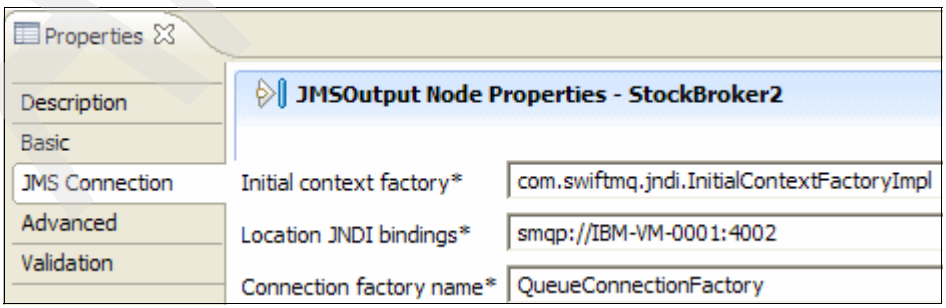


Figure 10-19 JMSOutput Node configurations for SwiftMQ

The JMS Connection properties require the following settings:

- For Initial context factory, type `com.swiftmq.jndi.InitialContextFactoryImpl`.

  A JMS application uses the initial context to obtain and look up the JNDI administered objects for the JMS provider. Enter this value exactly as shown here for SwiftMQ.

- For Location JNDI bindings, type `smqp://`*`hostname`*`:4002`.

  SMQP stands for SwiftMQ Protocol and is the underlying protocol for communication between JMS clients and a SwiftMQ router. *hostname* is the name of the machine where the SwiftMQ provider is running. The port number for router2 is 4002.

- For Connection factory name, type `QueueConnectionFactory`.

  This field contains the name of the queue connection factory as defined in SwiftMQ. A QueueConnectionFactory object was defined in the default configuration and was used in this scenario.

On the Basic properties tab, we set Destination queue to `ITSOQ@router2` as shown in Figure 10-20. ITSOQ@router2 is the JNDI name of the ITSOQ queue. SwiftMQ automatically creates a JNDI reference for a queue of type `queue_name@router_name`.

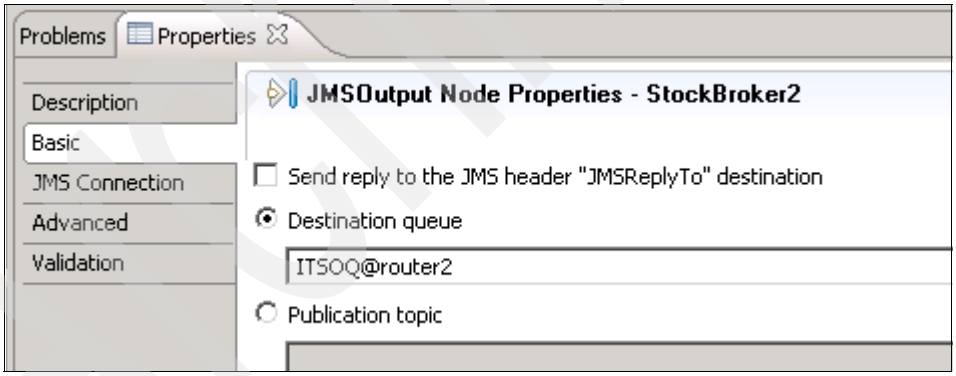


Figure 10-20 Destination queue for the JMSOutput node

10.10 Testing the flow

In this section, we explain the integration testing between the integration module and the message flow in WebSphere Integration Developer. The flow was tested using the Integration Test Client.

The test client is invoked from the context menu of the business integration module, StockQuoteModule, in the Business Integration view.

1. Select **StockQuoteModule**, right-click, and select **Test** → **Test Module**.
2. On the Configuration tab, in the Emulators section, remove **StockQuotePartner**.
3. In the Initial request parameters area (Figure 10-21), enter the values for the StockBO business object attributes.
4. Click **Continue**.

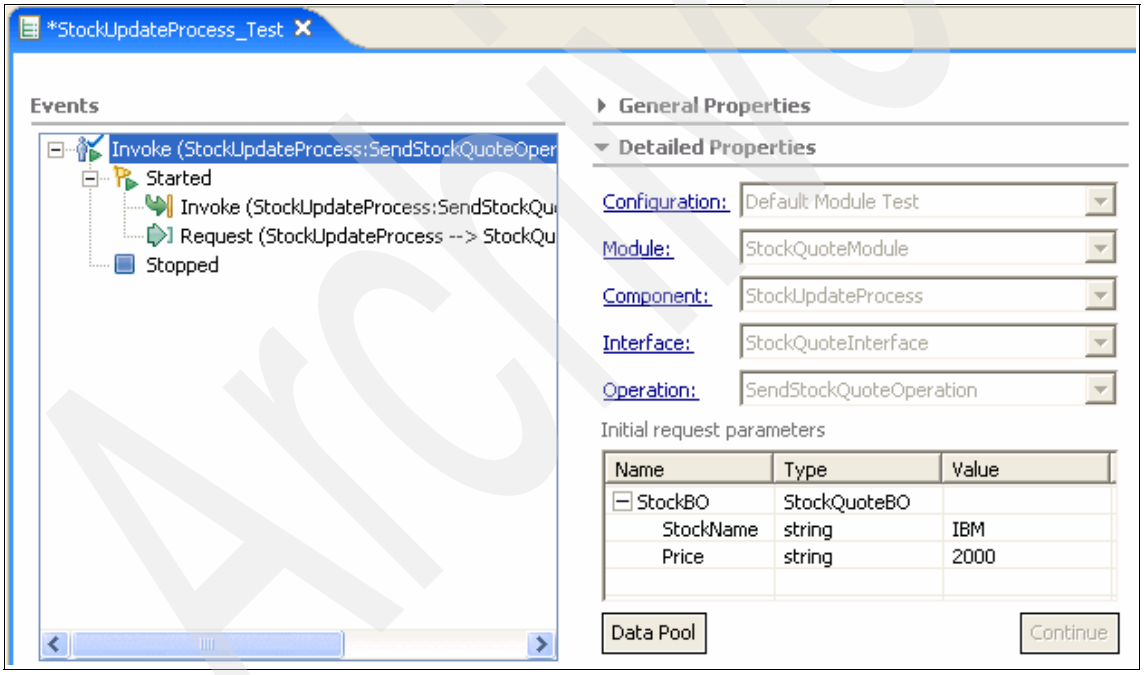


Figure 10-21 Testing the module

5. Since this is a one-way request, no response is returned. The messages are delivered to the JMS providers. You can look in the SystemOut log for the Process Server application server to view messages or errors produced by the business process application (Example 10-1 on page 332).

Archived

Example 10-1 Standard output from the Integration Test Server

```
[9/27/07 9:08:28:078 CDT] 0000008e SibMessage    I   [:] CWSID0020I: The messaging
engine configuration reload is complete.
[9/27/07 9:08:31:031 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding MQQCF as
jms/MQQCF
[9/27/07 9:08:31:109 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding MQINPUTQ as
jms/MQINPUTQ
[9/27/07 9:08:31:188 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding WPS
DataSource_CF as eis/jdbc/WPSDB_CMP
[9/27/07 9:08:31:219 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding QCF as
jms/QCF
[9/27/07 9:08:31:266 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding WPS
DataSource as jdbc/WPSDB
[9/27/07 9:08:31:281 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding
DynamicArtifactRepositoryDataSource as
jdbc/wbiserver/DynamicArtifactRepositoryDataSource
[9/27/07 9:08:31:297 CDT] 000000bb SystemOut     O *** WARNING *** Both transport and
store protocols are undefined; this mail session will be unusable
[9/27/07 9:08:31:297 CDT] 000000bb ResourceMgrIm I   WSVR0049I: Binding
WSRRMailSession as mail/WSRRMailService
[9/27/07 9:08:31:328 CDT] 000000bb ApplicationMg A   WSVR0200I: Starting application:
StockQuoteModuleApp
[9/27/07 9:08:31:656 CDT] 000000bb ArtifactLoade I
com.ibm.ws.al.bootstrap.WASBootstrap appStart(DeployedApplication) CWLAG0205I:
Artifactloader initializing is complete for module StockQuoteModuleApp.
[9/27/07 9:08:31:875 CDT] 000000bb Core          I   SCA_STARTING_APP_MODULE
[9/27/07 9:08:32:203 CDT] 000000bb Core          I   SCA_STARTED_APP_MODULE
[9/27/07 9:08:33:047 CDT] 000000bb EJBContainerI I   WSVR0207I: Preparing to start
EJB jar: StockQuoteModuleEJB.jar
[9/27/07 9:08:33:094 CDT] 000000bb EJBContainerI I   WSVR0037I: Starting EJB jar:
StockQuoteModuleEJB.jar
[9/27/07 9:08:33:188 CDT] 000000bb ActivationSpe E   J2CA0306I: Application
StockQuoteModuleApp#StockQuoteModuleEJB.jar#ServiceSIBusMessageBean has provided no
<activation-config-property> for the ActivationSpec class
com.ibm.ws.sib.ra.inbound.impl.SibRaActivationSpecImpl of ResourceAdapter
cells/widCell/nodes/widNode/resources.xml#J2CResourceAdapter_1180031608156. This may
have undesirable effects.
[9/27/07 9:08:33:266 CDT] 000000bb ApplicationMg A   WSVR0221I: Application started:
StockQuoteModuleApp
[9/27/07 9:09:16:203 CDT] 0000008f SystemOut     O ************** Transfer Request
sent to SIBus
**********************
```

Verifying the output

In SonicMQ, the messages were verified by using the JMSTest Client in the SonicMQ management console to view them. In the test client, the following actions were performed:

1. A connection was created to the broker at `tcp://localhost:2506`.
2. A queue session was created on the connection.
3. A receiver was created on the queue session for the jms_OUTQ queue.
4. After each message was sent to the business process, the queue was checked for messages.

In SwiftMQ, the messages were viewed using the Queue Manager Swiftlet for router2 in the SwiftMQ console. The message count in the usage statistics for the ITSOQ queue contained the messages that were received.

Archived

Archived

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see “How to get Redbooks” on page 337. Note that some of the documents referenced here may be available in softcopy only.

- *Patterns: Integrating Enterprise Service Buses in a Service-Oriented Architecture*, SG24-6773
- *Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB*, SG24-7369
- *WebSphere Message Broker Basics*, SG24-7137
- *WebSphere Message Broker V6, Best Practices Guide: Bullet Proofing Message Flows*, REDP-4043
- *WebSphere MQ V6, WebSphere Message Broker V6, and SSL*, REDP-4140
- *WebSphere MQ V6 Fundamentals*, SG24-7128

Online resources

These Web sites are also relevant as further information sources:

- WebSphere Message Broker Information Center

  http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp
- IBM WebSphere Message Broker Web page

  http://www.ibm.com/software/integration/wbimessagebroker
- WebSphere MQ Information Center

  http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp
- IBM WebSphere MQ Web page

  http://www-306.ibm.com/software/integration/wmqproductline/

© Copyright IBM Corp. 2008. All rights reserved.

- WebSphere MQ Family SupportPacs:

  http://www-1.ibm.com/support/docview.wss?rs=849&uid=swg27007205
- IBM WebSphere Business Process Management Information Center

  http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp
- IBM WebSphere Process Server Web page

  http://www-306.ibm.com/software/integration/wps/
- WebSphere Adapters Web page

  http://www-306.ibm.com/software/integration/wbiadapters/
- IBM Client for JMS on J2SE with IBM WebSphere Application Server

  http://www-1.ibm.com/support/docview.wss?uid=swg24012804
- IBM Education Assistant for WebSphere software products

  http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp
- *Connecting the JMS Transport Nodes for WebSphere Message Broker v6 to Popular JMS Providers*

  http://www.ibm.com/developerworks/websphere/library/techarticles/0610_bicheno/0610_bicheno.html
- Open Group. *Distributed Transaction Processing: The XA Specification*. February 1992, UK. ISBN 1-872630-24-3.

  http://www.opengroup.org/public/pubs/catalog/c193.htm
- OpenSSL

  http://www.openssl.org

Archived

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Archived

Archived

Index

A

ACORD AL3 40
activities 19
adapter 8–9
aggregate name 251, 259
AggregateControl node 28, 34, 240, 250–251, 259
AggregateReply node 28, 35, 240, 242, 256, 259
AggregateRequest node 28, 34, 240, 250, 254, 256
aggregation 34, 230
  data enrichment 229
  fan-in 35, 256
  fan-out 34, 249
  implementation 240–241
  message flow 34–35, 255
  message routing 22, 30
  MQRFH2 header request 247
  nodes 241
AMQMSRVN 160
assembly diagram 16, 18–19
assembly editor 18
asynchronous callback 275, 284
asynchronous messaging 52

B

Backout destination field 148
backout queue 103–104
bar file 49–50, 159
base queue manager 152
binding 296
  import component 143
  information 16, 19
  JMS 325
  SCA 17
  Web service 240
bindings file 129, 148
  JNDI 94, 102
bindings transport 95, 132–133
BLOB 41, 44, 223, 226
  domain 225–226
BPEL (Business Process Execution Language) 13, 19
broker 25
  archive file 49
  database 111
  domain 26
Broker Administration perspective 27
Broker Application Development perspective 26
built-in node 22
business integration 5
business integration module 15, 183
  business process 15, 18
  components 136, 283
  implementation 85, 135, 183, 232, 283, 318
  import components 195
  point-to-point request-response operations 68
  ProcessOrder 232
  Scenario1WPSModule 85
  StockQuoteModule 319
  testing 20, 331
  TransferFund 144
  TransferFundModule 161
Business Integration perspective 16
business object 12, 55
  ackOAType 185
  aggregation 233
  asynchronous callback 285
  component communication 18
  data transformation and security 184
  Invoice 286
  map 13
  OpenAccount 193
  OpenAccountType 186
  ProcessOrdersRequest 88
  StockQuoteBO 320
  Supplier 235
  TranferReqBO 161
  transaction manager 137
Business Object XML using JMS TextMessage 69–70
business process 5
  aggregation 230
  asynchronous and synchronous calls 284
  asynchronous callback 276
  business state machine 14
  component 13, 19
  connectivity to third-party JMS providers 312
  correlation sets 290

© Copyright IBM Corp. 2008. All rights reserved.

data transformation and security 166
development 15, 18
export component and binding 296
external access 188
import component and binding 143, 240, 294, 325
interface and partner references 90, 140, 190, 237, 289
Java component 19
Java snippet 92
LoanPrcInterface 188
LoanProcess 189
message splitting and routing 80
OrderBusinessProcess 288
OrderModule 283
OrderProcessInterface 287
ProcessOrder 232
ProcessOrderComponent 233, 236
real-time monitoring of 13
Receive activity 91, 191, 238, 291
Reply activity 239, 294
SCA implemation 15
service component 17
SOA 4
SplitMessageProcess 85, 90
StockUpdateProcess 319, 322
TransferFundProcess 140
transport protocol 23
WebSphere Message Broker as transaction manager 116
WebSphere Process Server 6, 11, 81
Business Process Execution Language (BPEL) 13, 19
business rules 15, 19
business state machine 14–15
BytesMessage 55

C

C 39, 167
catch terminal 98, 108, 156, 158
CEI (Common Event Infrastructure) 12
channel compression 59
Check node 29
Choice activity 19
CICS EXCI 37
CICSRequest node 37–38
CLASSPATH 128
client transport 95
COBOL 39–41
com.ibm.mqetclient.jar 128, 159
ComIbmAggregateReplyBody 260
COMMAREA 38
Common Event Infrastructure (CEI) 12
compiled style sheet 44
component 4
Compute mode field 105, 210, 308
Compute node 29, 46
AddExceptionListToMsg 108
AddHTTPContext 262, 264
AddMQMDAndMQRFH2 node 246
AddRequestURL 308
AggregateResponses 256, 260
AppendMQH 303
broker properties 48
BuildSupplier1Response 267, 270, 272
ConvertToCWF 251
database access 47
Increment 305, 307
InitVar 305
JavaCompute node 43
labelname attribute 209
LocalEnvironment 35
RouteMsgToDestinationQ 98, 105
SaveMQH 301
Configuration Manager 25–27
connection factory 94, 103, 119, 151–152
Connection factory name field 148, 327
Coordinated Transaction property 159
correlation ID 65, 74, 154
correlation set 277, 290–293
CorrelId 53
CSIv2 inbound authentication 179
CSIv2 outbound transport 180
custom data binding 58, 60
custom wire format 243
Custom Wire Format (CWF) 40–41, 231, 244, 250, 267, 270
CWF (Custom Wire Format) 40–41, 231, 244, 250, 267, 270
CWF to XML transformation 270

D

data binding 56–57, 59, 68, 71, 295, 326
for MQ Export 66
Data perspective 28
database 111

Archive

Database node 29
DataBinding interface 58
DataDelete node 29
DataInsert node 29
DataPower 6
DataUpdate node 29
Debug perspective 27
default messaging provider 67, 311–312, 314–315, 326–327
Destination mode 110
destinations 68
domain objects 27
Dynamic Web project 20

E

EAR (Enterprise Archive) file 20
EDI 9
EDIFACT 39–40
EJB project 20
encoding style 75
Enterprise Application project 20
Enterprise Archive (EAR) file 20
enterprise service bus (ESB) 4, 6, 167
  products 6
  role 6
Envelope message type 201, 260, 263
Environment message tree 43
ESB (enterprise service bus) 4, 6, 167
  products 6
  role 6
ESQL (extended SQL) 23, 32, 39, 42, 46, 105, 109, 251, 260, 265, 270
ExceptionList 43, 45
execution group 25, 49
export binding 57
export component 17, 19, 296
  binding information 57
  development process 18
  JMS 71
  JMS bindings 71
  LoanPrcInterfaceExport1 184
  MQ data bindings 66
  MQ JMS binding 73–74
  OrderBusinessProcessExport 284, 296
  ProcessOrderComponentExport 233
  ReceiveInvoiceMQExport 284
  SCA components 57
  Scenario1_WPStoWMBExport1 85
  StockQuotePartner 324
  StockUpdateProcessExport 319
  TransferFundProcessExport 136
extended SQL (ESQL) 23, 32, 39, 42, 46, 105, 109, 251, 260, 265, 270
Extract node 29

F

failure terminal 258, 268
fan-in flow 242, 256
fan-in process 35
fan-out flow 241–242, 249
fan-out process 34
fan-out request 240
Filter node 28
flat file 49
FlowOrder node 29, 150, 210
function selector 66

G

getDataObject() 58
global transaction 118–119, 124, 151

H

handleMessage 67
HL7 9, 40
HTTP 37
HTTP listener 240, 263
HTTPInput node 28
  basic properties 206
  connectivity 76
  FindSuppliers 245
  HTTP/HTTPS Transport 37
  HTTPs_Request 219
  HTTPs_SOAP_Input 208
HTTPListener 176
HTTPReply node 28, 206
  HTTP/HTTPS Transport 37
  HTTPReplyWithError 246
  RespondToHTTPRequest 262, 264, 266
HTTPRequest node 28, 302, 309
  HTTP/HTTPS Transport 37
HTTPRequestIdentifier 246–247, 260, 262, 264
HTTPS 37, 173, 183, 191–193, 206, 214, 217
  listener 218
human task 14–15, 19
Human task activity 19

Archived

I

IA9O SupportPac 278
IA9Q SupportPac 278
IBM Client for JMS on J2SE with IBM WebSphere Application Server 317
IBM SOA Foundation 4
IBM System z Application Assist Processor (zAAP) 44–45
IDoc SAP 41
import binding 57
import component 17, 19, 56, 325
  aggregation 231, 240
  asynchronous callback 275–276, 294
  binding information 57
  connectivity to third-party JMS providers 311–312, 325
  data transformation and security 195
  development process 18
  FindSuppliersImport 233, 240
  JMS 71
  message splitting and routing 93
  MQ data bindings 66
  MQ JMS binding 73–74
  SendOrderMQImport 284, 294
  SplitMessageImport 85
  StockQuotePartner 319
  transaction manager 143
  TransferFundsPartner 136, 140
  VerifyCredit 190
  WebSphere Message Broker 115
import component binding 143
Increment node 305
initial context 130
initial context factory 103, 328, 330
Initial context factory field 148, 327
INITIAL_CONTEXT_FACTORY 83
input message parsing properties 103, 245, 250, 257, 263, 298
Input node 29
Integration Test Client 112, 161, 273, 309, 331
interface 18–19, 56, 287
  BankPortType 184, 191, 193, 201
  FindSuppliersInterface 233, 236
  LoanPrcInterface 190
  map 13, 15
  OrderProcessInterface 284, 296
  SendStockUpdateInterface 321–322
  TransferFundInterface 139
interface partner 19, 90–91, 140, 190, 237, 289–290, 322–323
Invoke activity 19, 91, 142, 193–194, 238, 292, 324
item 233

J

Java component 19
Java Naming and Directory Interface (JNDI) 315
  JMSInput node 82
  MQ queue name 102
Java snippet 92, 142, 192–193, 294, 324
Java Snippet activity 192
JavaCompute node 29, 43
  BankUtils.java class 204
  broker properties 48
  Compute node 46
  CopyMsgIDToCorrelID 154
  data retrieval 48
  Format_Soup_fault 217
  message transformation 39
  RemoveMQMD 227
  RemoveSOAPHeaders 209
  Route_Legacy_Request 219
  RouteToLabel node 32
  routing capability 31
  SelectProtocol 215
  Transform_TDS 210
  Transform_X12 213
javax.net.ssl.keyStore 182
javax.net.ssl.keyStorePassword 182
javax.net.ssl.trustStore 182
javax.net.ssl.trustStorePassword 182
JMS administered object 129
JMS binding 57, 60, 67–68, 74, 76, 311–312, 325
JMS body data binding 68
JMS default function selector 67
JMS destination 134
JMS export component 71
JMS function selector 71
JMS message format 54
JMS message header 54
JMS message type 55
JMS messaging domain field 325
JMS queue 121–122, 316
jms_jndi_ibm 317
JMSAdmin 83
JMSAdmin.bat 129
JMSAdmin.config 83
JMSBytesBody 60

JMSCorrelationID 71
JMSCorrelationID field 54
JMSDestination field 54
JMSInput node 28, 76, 99, 102, 126, 132, 158, 312, 327
  ReceiveStockUpdateRequest 327
  ReceiveTransferRequest 156, 158
  RecieveTransferRequest 147
JMSMessageID 71
JMSMessageID field 54
JMSMQTransform node 28, 98, 102, 105
JMSOutput node 28, 126, 132, 312
  connectivity to third-party JMS vendors 311
  CreditToBank1 152
  DebitFromBank2 150
  SendStatusFailure 157
  SendStatusSuccess 155
  StockBroker1 328
  StockBroker2 329
JMSReplyTo field 54
JMSSwitch.dll 126
JMSTextBody 60, 62–63
JMSTextMessage 144, 326
JNDI (Java Naming and Directory Interface) 315
  JMSInput node 82
  MQ queue name 102
JNDI bindings file 94, 102
JNDI name for connection factory field 325
JNDI name for send destination field 326
JSSE repertoire 181–182

K

keystore 175–177, 182, 192
keytool 175–176

L

Label node 28, 32, 209
labelname 209
Legacy 3 166
listener port 123–124
local transaction 118
LocalEnvironment 35, 43, 45, 98–99, 105–106, 110, 209–210, 217, 220–221, 262, 264, 300, 308
Location JNDI bindings field 148, 327
long-running process 289

M

malformed message 215
map 13
MapMessage 55
Mapping editor 42
Mapping node 29, 39, 45, 48, 153, 156
Mapping routine 154
MbElement API 44
MbSQLStatement 44
MbXPath 211, 213–214, 216
MbXPath API 44
MDB 123
message
  bit stream 40
  categories 41
  context 249, 253
  correlation 74
  definition 199, 210, 213, 215
  definition file 41, 101, 145, 244
  domain 104, 298
  enrichment 23
  flow 22, 255
  format 104
  ID 65, 248–249, 253–254
  map 153
  model 41
  set 23, 41, 99–100, 144, 196, 240, 243, 245
  set ID 100, 243
  set project 41
  transformation 23, 165
  tree 39, 100, 215, 264
  type 103–104
Message Brokers Toolkit 23, 26, 28, 49
Message Flow editor 26
message-driven bean 123
MessageFormat 212, 214–215
MessageSet 212, 214–215
MessageType 212, 214–215, 217, 219
messaging client 56
messaging provider 52
MIME 41
module 18
module assembly 18
mport 94
MQ
  Client transport 132
  function selector 66
  message format 53
  native body data bindings 59

Archived

MQ binding 57–60, 76, 93, 275–276, 282, 295–296
MQ JMS 73
MQ JMS binding 17, 57, 73–74, 76, 79, 81, 93, 115, 143
MQ to HTTP transformation 263
MQBEGIN 126
MQGet node 28, 48, 225–226
MQInput node 28, 36, 76, 275, 298
  FindSuppliersFanInInput 256–257
  FindSuppliersFanOutInput 250
  FindSuppliersWSResponseInput 262–263
  Supplier1RequestInput 267
  Supplier2RequestInput 272
  XMLNS parser 295
MQJMSTransform node 28
MQMD 53, 61, 214, 227, 301, 303
MQOptimizedFlow node 28
MQOutput node 28, 36, 98, 108, 110–111, 215, 224, 227, 242, 248, 253–254, 262, 275, 300, 304, 307
  AggRequestToSupplier1 267
MQReply node 28, 36, 271
  Supplier1Reply 267
  Supplier2Reply 272
MQRFH 59, 61
MQRFH2 53, 59, 61, 67, 246–247, 260, 264
MQSC 82
mqsiapplybaroverride command 50
mqsichangebroker command 127
mqsichangeproperties command 176
mqsicreatebar command 49
mqsicreatebroker command 127
mqsideploy command 50
MQXAQueueConnectionFactory class 134, 159
MRM 41–42, 44–45, 62, 94, 100, 196, 223, 243
  domain 201, 208, 219
MsgId 53
Multicast transport 37
MUSR_MQADMIN 160
mxsd file 145

N

namespace 185–186, 286
native method 71
node 22

O

ObjectMessage 55
one-way interaction 80, 166
  pattern 311
  service 215
one-way JMS message 318
one-way operation 19, 56–57, 68, 73, 89, 168, 189, 287–288
one-way request 81, 311–312, 331
OpenAccount message flow error 227
openssl 175
operation 56, 321
Output node 29

P

parser 23
  options 104
partner reference 193, 324
Passthrough node 29
PasswordBasedDecryption node 29, 223, 225–226
PasswordBasedEncryption node 29, 223
PATH 128
pattern 80
Plug-in Development perspective 28
port 7080 240
PrjTxnFlowMSets message set 144
ProcessOrderComponentExport 233
protocol conversion 23
protocol transformation 165
PROVIDER_URL 83
Publication node 28, 36–37
publish/subscribe 59, 94

Q

qm.ini file 124
queue connection factory 82, 119, 131, 143, 315, 318, 325, 329–330
queue manager implementation, WebSphere MQ 232
queues 68

R

Real-timeInput node 29, 37
Real-timeOptimizedFlow node 29, 37
Receive activity 91, 141, 191–192, 238, 291, 293, 323
recoverXAQCF 127, 160
Redbooks Web site 337
  Contact us xiii

Archived

reference 15
architecture 4
partner 19, 90–91, 140, 190–191, 194, 238, 289–290, 322–323
relationship instances 13
Reply activity 142, 194, 239, 294
reply-to queue 254, 267, 271–272
ReplyToQ 53
request ID 74
request-reply interaction 22, 34, 115, 123, 166, 188, 229, 241
request-response operation 19, 56–58, 65, 68, 71, 74, 139, 143, 162, 168, 173, 189, 231, 236, 238, 287
RequestURL 300
ResetContentDescriptor node 29, 46, 223, 226
resource manager 117–118, 124–128
Resource Recovery Services (RRS) 125
response correlation schema 65, 74
response message parsing settings 302
RosettaNet 41
RouteToLabel node 28, 32, 209, 215, 217, 219
routing 22, 30, 79, 112
runmqsc command 183

S

SAP BAPI 41
SCA (Service Component Architecture) 12
component 14
implementation 15
interface 15
SCADA 37
SCADAInput node 29
SCADAOutput node 29
schema for simple JMS data bindings 62
SDO (Service Data Objects) 55
business objects 12
Secure Sockets Layer (SSL) 168, 192, 205–206
selector 13, 15, 19
SelectProtocol JavaCompute node 215
serialization type 94, 144, 326
Serialization type field 326
Serialized as XML
data binding type 60–61
serialization type 295
Serialized Business Object using JMSObjectMessage 69
serialized Java object 60
serlialization type 295
service 14
component 13–14, 19
Service Component Architecture (SCA) 12
component 14
implementation 15
interface 15
Service Data Objects (SDO)
business objects 12, 55
service integration 314
service user ID 160
service-oriented architecture (SOA) Foundation 4
setDataObject() 58
shared-classes directory 84, 128, 159
shared-classes folder 317
sibc.jms.jar 317
sibc.jndi.jar 317
Simple JMS BytesMessage Data Binding 69
Simple JMS MapMessage Data Binding 69
Simple JMS Message Data Binding 69
Simple JMS ObjectMessage Data Binding 69
Simple JMS StreamMessage Data Binding 69
Simple JMS TextMessage Data Binding 69
SMQP 330
SOA (service-oriented architecture) Foundation 4
SOAP 250
envelope 76, 208, 219, 221, 305
exception 246
Header 76
message format 75
over HTTP 37
SOAP Body 76
SOAP to CWF conversion 251
SOAP/HTTP 75–76, 166, 229–231, 245, 276, 297
SOAP/HTTPS 75
SOAP/HTTPs 168
SOAP/JMS 75–76
soapenv11.mxsd 202
SOAPEnvelope node 29, 227, 301
SOAPExtract node 29, 221, 303
SonicMQ 312–313, 318, 326, 328–329, 333
SRRetrieveITService node 30, 276, 299–300, 305
SSL (Secure Sockets Layer) 168, 192, 205–206
state machine 19
StreamMessage 55
SupportPac 9, 28, 49, 165, 221, 223
IA9A 99
SWIFT 9, 40
SwiftMQ 312–313, 318, 326, 329–330, 333
switch file 126, 128, 159

sync point 118
synchronous messaging 52
syncpoint coordination 151
syncpoint coordinator 118, 124, 126
syncpoint manager 125
SYSTEM.DEF.SRVCONN 96
System.setProperty 182
SystemOut 92, 113, 294

T

tagged delimited string (TDS) 40, 62, 64, 196, 200
TargetFunctionName 71–72
targetNamespace 185
TDS (tagged delimited string) 40, 62, 64, 196, 200
telemetry 9
TextMessage 55
third-party JMS provider 117, 311
thread 25
ThreadOfControl 127
Throw node 29, 158, 225
TimeoutControl node 29, 307
TimeoutNotification node 29, 305, 308
topic 68
Trace node 29, 258–259, 264, 269, 328
  code 269
transaction 117–118
  coordinator 119
  manager 115, 117–118, 124
  mode 149, 151, 160
transactional support 9
Transform_TDS JavaCompute node 210
Transform_X12 JavaCompute node 213
transport protocols 9
transports 9
truststore 175–176, 182, 192
try terminal 98, 108
TryCatch node 29, 98, 108, 217

U

unit of work 118
unknown reply 259
unstructured binary message 60
unstructured text message 60, 62
URL path suffix 240
Use namespaces 146
User Name Server 26

V

Validate node 29, 204
validation properties 246
verfyHeader 217
VSAMDelete node 37
VSAMInput node 37
VSAMRead node 37
VSAMUpdate node 37
VSAMWrite node 37

W

Warehouse node 29
WCInboundAdminSecure 181
Web service binding 75–76, 229, 231, 240
WebSphere Adapter for Flat Files 49
WebSphere Application Server 5
  Network Deployment 5
WebSphere Broker JMS Transport 37
WebSphere Business Integration Adapters 23
WebSphere Integration Developer Version 6 6, 15
WebSphere Message Broker File Extender 38
WebSphere MQ
  JMS provider 116
  messaging provider 68, 93
  queue manager implementation 232
WebSphere MQ Mobile Transport 36
WebSphere MQ Multicast Transport 36
WebSphere MQ Real-time Transport 37
WebSphere MQ Telemetry Transport 37
WebSphere MQ Transport 36
WebSphere MQ Web services Transport 37
WebSphere Process Server Version 6 5
WebSphere Service Registry and Repository 30, 275, 278, 299–300
WebSphere Transformation Extender 6, 24
  for Message Broker 38
wire format 103
wires 18

X

X12 39–40, 167, 169, 199, 213
XA switch structure 126
XA transaction 115, 117–119, 132, 134, 158, 160
XA/Open transaction call 126
xa_close 127
xa_open 126
XACloseString 127
xahandler.jar 159

Archived

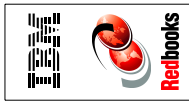
XAJMS_TRACEFILE 160
XAOpen 160
XAOpenString 126
xarecovery.jar 128
XAResourceManager stanza 124–125
XML 40–41, 221
XML domain 41, 44
XML validator plug-in node 99
XMLNS 41–42, 226, 295, 298
  domain 226
XMLNSC 41
XML-to-XML mapping file 221
XMLTransformation node 29, 44, 221–222
XMX file 221–222
XPath 43–44, 211–212, 214, 217
XSLT 39, 44–45

Z

zAAP (IBM System z Application Assist Processor) 44–45

Archived

Archived



Redbooks

Using IBM WebSphere Message Broker as an ESB with WebSphere Process Server

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

INTERNAL

Archived

Archived

Using IBM WebSphere Message Broker as an ESB with WebSphere Process Server

Enterprise service bus capabilities of WebSphere Message Broker

Examples of mediation with message flows

Examples of connectivity options

IBM WebSphere Process Server is a business integration server that was built to support solutions that are based on the service-oriented architecture (SOA). It plays a key role in the IBM SOA Foundation architecture by providing functionality for process services. Another key component of the architecture is the enterprise service bus (ESB). IBM provides two key ESB products: IBM WebSphere Enterprise Service Bus and WebSphere Message Broker.

This IBM Redbooks publication has been written for architects who are planning an SOA solution and application designers who are implementing an SOA solution with WebSphere Process Server and WebSphere Message Broker. In this book, we highlight the ESB capabilities of WebSphere Message Broker and explain how you can leverage them with WebSphere Process Server. In addition, we discuss interoperability and provide examples to illustrate the integration of the two products.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7527-00 ISBN 0738485543