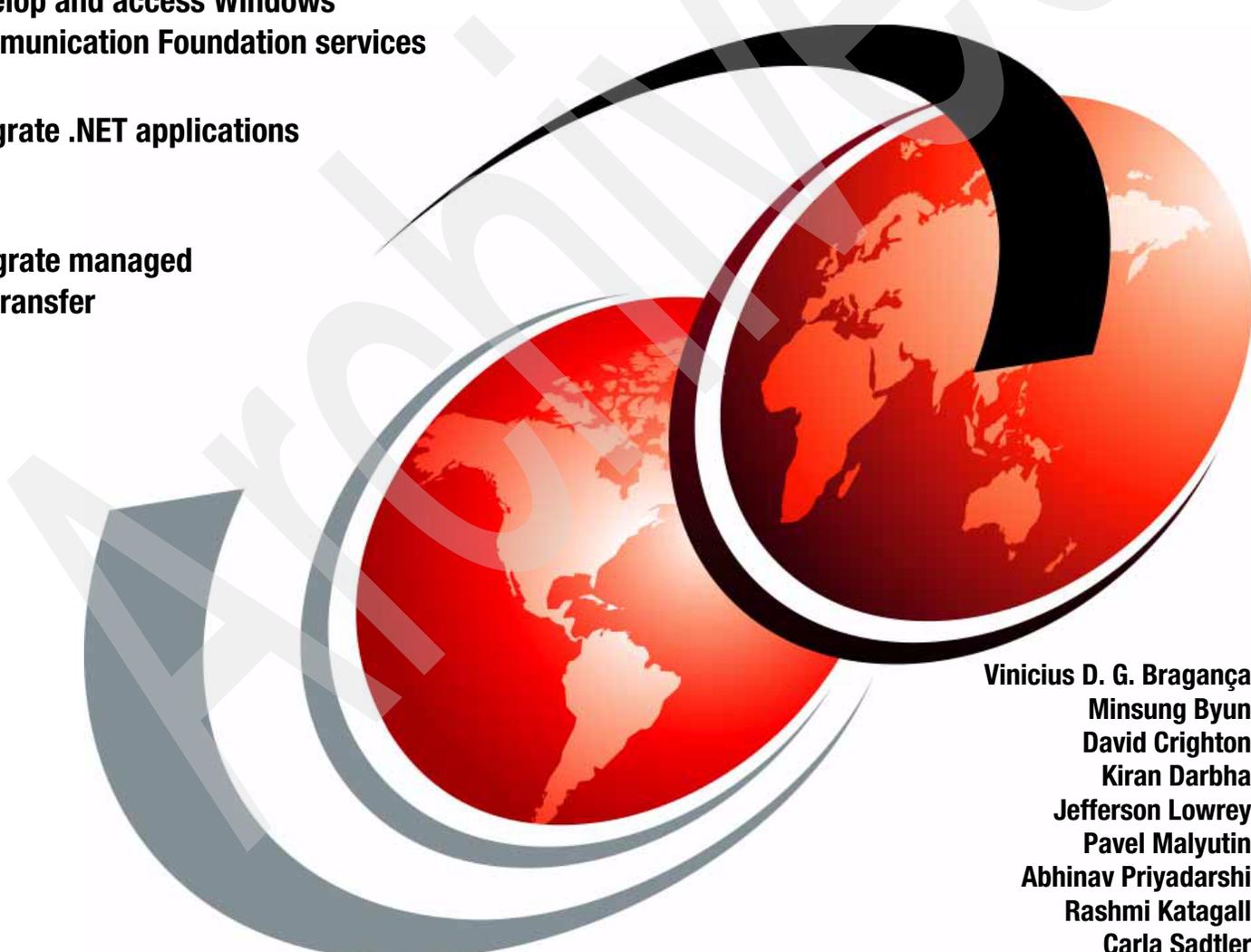


Using WebSphere Message Broker V8 in Mid-Market Environments

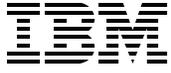
Develop and access Windows Communication Foundation services

Integrate .NET applications

Integrate managed file transfer



Vinicius D. G. Bragança
Minsung Byun
David Crighton
Kiran Darbha
Jefferson Lowrey
Pavel Malyutin
Abhinav Priyadarshi
Rashmi Katagall
Carla Sadtler



International Technical Support Organization

Using WebSphere Message Broker V8 in Mid-Market Environments

August 2012

Archived

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

Archived

First Edition (August 2012)

© Copyright International Business Machines Corporation 2012. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contact an IBM Software Services Sales Specialist



Start SMALL, Start BIG, ... **JUST START** architectural knowledge, skills, research and development . . . **that's IBM Software Services for WebSphere.**

Our highly skilled consultants make it easy for you to design, build, test and deploy solutions, helping you build a smarter and more efficient business. **Our worldwide network of services specialists wants you to have it all!** Implementation, migration, architecture and design services: IBM Software Services has the right fit for you. We also deliver just-in-time, customized workshops and education tailored for your business needs. You have the knowledge, now reach out to the experts who can help you extend and realize the value.

For a WebSphere services solution that fits your needs, contact an IBM Software Services Sales Specialist:
ibm.com/developerworks/websphere/services/contacts.html

Archived

Contents

Contact an IBM Software Services Sales Specialist	iii
Notices	xi
Trademarks	xii
Preface	xiii
The team who wrote this book	xiii
Now you can become a published author, too!	xv
Comments welcome	xvi
Stay connected to IBM Redbooks	xvi
Chapter 1. WebSphere Message Broker for mid-market solutions	1
1.1 WebSphere Message Broker	2
1.1.1 Message processing	2
1.1.2 Connectivity options	4
1.1.3 WebSphere Message Broker editions	6
1.2 WebSphere Message Broker for midmarket	7
1.2.1 Microsoft .Net integration	7
1.2.2 File processing and transfer	8
1.3 Additional resources	8
Chapter 2. Introduction to WebSphere Message Broker V8	9
2.1 Runtime architecture of WebSphere Message Broker	10
2.1.1 Broker	10
2.1.2 Execution groups	10
2.2 Development environment of WebSphere Message Broker	11
2.2.1 WebSphere Message Broker Toolkit	11
2.2.2 Applications and libraries	12
2.2.3 Message flows	13
2.2.4 Message	15
2.2.5 Message modeling	15
2.2.6 Pattern instances	17
2.2.7 Microsoft Visual Studio 2010 integration	18
2.3 Connectivity options	18
2.3.1 WebSphere MQ	18
2.3.2 Java Message Service (JMS) 1.1	19
2.3.3 WebSphere MQ/JMS	20
2.3.4 SOAP-based web services	21
2.3.5 HTTP and HTTPS	22
2.3.6 File	23
2.3.7 Java Connector Architecture (JCA) Adapters and WebSphere adapters	25
2.3.8 TCP/IP	25
2.3.9 Service Component Architecture	26
2.4 Transformation interfaces	27
2.4.1 Compute node	27
2.4.2 JavaCompute node	28
2.4.3 PHPCompute node	28
2.4.4 .NETCompute node	29
2.5 Administering WebSphere Message Broker	29

2.5.1	WebSphere Message Broker Explorer	30
2.5.2	Broker sets	31
2.5.3	Web Administration and record and replay	31
2.6	Deploying applications	33
2.7	Getting started	33
2.7.1	Opening the Message Broker Toolkit	34
2.7.2	Creating a new application	35
2.7.3	Creating a new message flow	36
2.7.4	Creating a broker	38
2.7.5	Managing brokers	43
2.7.6	Creating a BAR file for deployment	46
2.7.7	Deploying the BAR file to a broker	48
2.7.8	Executing WebSphere Message Broker commands	49
2.8	Creating WebSphere MQ queue managers and queues	50
2.8.1	Creating queue managers	50
2.8.2	Creating queues and queue managers using commands	50
2.8.3	Creating queue managers and queues from the WebSphere MQ Explorer	51
Chapter 3. Using the tools for WebSphere Message Broker and .NET Integration ..		53
3.1	Creating a .NETCompute node in a message flow	54
3.1.1	Creating the messaging flow	54
3.1.2	Creating the code for the .NETCompute node	56
3.1.3	Setting the properties for the .NETCompute node	59
3.1.4	Deploying the application	60
3.2	Hot swap deploys	62
3.3	Using the Visual Studio debugger	63
Chapter 4. Scenario: Bridging WebSphere MQ and Microsoft Message Queuing ..		67
4.1	Scenario overview	68
4.2	Receiving messages from MSMQ	69
4.2.1	Constructing a simple utility to read and write MSMQ messages	69
4.2.2	Constructing and testing a basic message flow	78
4.3	Mapping headers between MSMQ and WebSphere MQ	88
4.4	Adding logging to your code	89
4.5	Handling exceptions	93
4.6	Sending messages to MSMQ	94
4.6.1	Constructing the message flow	94
4.6.2	Configuring the message flow	95
4.6.3	Creating the .NET code	96
4.6.4	Writing the code	96
4.7	Distributed transactional coordination	103
4.7.1	Manual transaction sample	103
4.7.2	Testing the message flow	105
4.8	Conversions	108
Chapter 5. Scenario: Calling Microsoft Dynamics CRM from a message flow		109
5.1	Scenario overview	110
5.1.1	SAP Request node for SAP Software	110
5.1.2	Microsoft Dynamics CRM Online	111
5.1.3	Prerequisites for this scenario	111
5.2	Creating the message flow	111
5.2.1	Setting up the environment	114
5.2.2	Creating and connecting the nodes	115
5.2.3	Configuring the node properties	116

5.2.4	Broker configuration for SAP request nodes	127
5.2.5	Writing the code for the SAP nodes	128
5.2.6	Coding the ESQL for the Compute nodes	128
5.2.7	Coding the Filter Request .NETCompute node	131
5.2.8	Writing the code for the database operations	132
5.2.9	Writing the code for accessing Microsoft Dynamics CRM Online	135
5.2.10	Writing the code for the .NETCompute node (Create) - DB<->CreateCRMOnlineCustomer	142
5.3	Deploying the message flow	144
5.4	Testing the message flow	147
5.4.1	Windows Forms Project: GUI Based	147
5.4.2	Writing the code for the SubscribeToService form	151
5.4.3	Running the test	157
5.4.4	Using the test client from the Message Broker Toolkit	157
5.5	Troubleshooting tips	161
Chapter 6. Scenario: Integration Windows Communication Foundation in message flows - Part 1		163
6.1	ClaimsProcessingWcfService overview	164
6.2	Windows Communication Foundation	165
6.2.1	WCF ABCs	166
6.2.2	WCF operation	168
6.3	Developing the WCF Service	168
6.3.1	Creating a Microsoft Visual Studio 2010 project for the WCF Service	168
6.3.2	Setting the namespace for the project	169
6.3.3	Modifying the classes created by the New Project Wizard	170
6.3.4	Creating the required datatypes	172
6.3.5	Creating private methods to serialize and deserialize the Datatypes	180
6.3.6	Designing and implementing the service contract	184
6.3.7	Building the WCF Service	229
6.3.8	Configuring the WCF service	230
6.3.9	Testing the WCF Service using the WCF Test Client	235
6.4	Obtaining information about the service using MEX	246
6.5	Generating WCF client code from a MEX enabled WCF service	248
6.6	WCF hosting options	251
6.6.1	Hosting using Visual Studio	251
6.6.2	Hosting using a stand-alone application	251
6.6.3	Hosting as a service	257
6.6.4	Hosting in Internet Information Services (IIS)	265
Chapter 7. Scenario: Integrating Windows Communication Foundation in message flows - Part 2		273
7.1	Scenario overview	274
7.1.1	Prerequisites	275
7.2	Creating the message flow	276
7.2.1	Preparing message models	276
7.2.2	Creating the message flow	278
7.2.3	Creating and connecting the nodes	280
7.2.4	Configuring node properties	284
7.2.5	Queue names	290
7.3	Using the Mapping node to transform the input to the Canonical Message Format	290
7.4	Transforming the SOAP input message	296
7.5	Creating a .NETCompute node to consume the ClaimsProcessingWcfService	298
7.5.1	Creating a constant to hold the namespace prefix	301

7.5.2	Adding the MEX-generated WCF client to the project	302
7.5.3	Developing helper methods to interact with the WCF service.	304
7.5.4	Concurrency	304
7.5.5	Handling exceptions when invoking the WCF service	307
7.5.6	Ensuring that the serviceClient is closed on flow deletion	309
7.5.7	Creating methods to add datatypes to the message tree	309
7.5.8	Creating a helper method to build Customer object from the Logical Message Tree	311
7.5.9	Preparing the ClaimsProcessingRequest object.	312
7.5.10	Adding the information from the WCF response into the message tree	314
7.5.11	The CallWCFNode.Evaluate() method	316
7.5.12	Handling failed tree navigations in the Evaluate() method	320
7.5.13	Handling FaultExceptions in the Evaluate() method.	321
7.5.14	Verifying the completed CallWCFNode.	322
7.6	Routing the output.	326
7.6.1	Additional processing	326
7.6.2	Routing the reply message	328
7.7	Writing a .NETCompute node class to generate a payment message	330
7.8	Processing incomplete results for the ViewOutstandingClaims operation	333
7.9	Creating a Word document for email and print	337
7.9.1	Preparing a letter template.	338
7.9.2	Creating a user-defined property for the template location.	340
7.9.3	Creating .NET code to generate a Word document using a template.	341
7.9.4	Using the EmailOutput node to send an email to the customer	347
7.9.5	Using the FileOutput node to send the file for printing	348
7.10	Using the Mapping node to transform the Canonical message to the output format	349
7.11	Using a Compute node to transform the Canonical Message into a SOAP message	353
7.12	Handling exceptions in the flow.	353
7.13	Building and deploying the Visual Studio project	358
7.13.1	Building a solution.	358
7.14	Updating the properties for the .NETCompute nodes	359
7.15	Building and deploying the Message Broker application.	359
7.15.1	Creating the BAR file	360
7.15.2	Deploying the BAR file	360
7.16	Testing the WebSphere Message Broker application	361
7.16.1	Testing the MQ interface.	362
7.16.2	Testing the SOAP Interface	381
7.17	Altering the scenario to use a SOAP over HTTP based binding.	383
7.17.1	Altering the WCF Service to expose an http based binding	383
7.17.2	Altering a message flow to use an HTTP based binding	385
Chapter 8. Integrating file transfer with WebSphere MQ FTE into the message flow		391
8.1	Scenario overview.	392
8.2	Overview of the WebSphere MQ File Transfer edition	393
8.2.1	Using WebSphere Message Broker as a bridge for FTE networks.	394
8.3	Preparing the broker environment for this scenario	395
8.3.1	Creating the database.	396
8.3.2	Configuring the ODBC data source.	401
8.3.3	Preparing the file system structure	402
8.4	Applications emulated in this scenario	403
8.4.1	Branch application (FTE)	403
8.4.2	Report Data Warehouse	409
8.4.3	Volume Analysis System.	414

8.5	Creating the main message flow	418
8.5.1	Message flow overview	419
8.5.2	Creating and connecting the nodes	420
8.5.3	Creating a message definition	422
8.5.4	Using FTE nodes	429
8.5.5	Using aggregation in WebSphere Message Broker	431
8.5.6	Producing multiple messages from the compute node	434
8.5.7	Accessing databases from the database nodes	435
8.5.8	Creating the queues for the FTESample application	438
8.6	Running the scenario	438
8.6.1	Troubleshooting tips	439
8.6.2	Exploring the work of the scenario	440
8.7	Extending the scenario	441
Chapter 9. Integrating file transfer using Sterling Connect:Direct with your message flow		443
9.1	Scenario overview	444
9.2	Overview of the Sterling Connect:Direct	445
9.3	Using WebSphere Message broker with Connect:Direct	446
9.4	Preparing the Connect:Direct environment	447
9.5	Preparing the logging database	453
9.6	Configuring WebSphere Message Broker	456
9.6.1	Creating a configurable service on WebSphere Message Broker	456
9.6.2	Creating the message flow	458
9.6.3	Creating the main message flow	459
9.6.4	Creating the simulation flow	469
9.6.5	Deploying the message flows and preparing the environment	472
9.6.6	Sample file transfer data	473
9.6.7	Testing the message flow	473
Appendix A. Additional material		485
	Locating the Web material	485
	Using the Web material	485
	Downloading and extracting the Web material	486
Related publications		487
	IBM Redbooks	487
	Online resources	487
	Help from IBM	488

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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com) are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

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

AIX®
CICS®
DataPower®
DB2®
developerWorks®

IBM®
IMS™
Informix®
MVS™
Redbooks®

Redbooks (logo) ®
solidDB®
Tivoli®
WebSphere®
z/OS®

The following terms are trademarks of other companies:

Connect:Direct, Sterling Commerce, and N logo are trademarks or registered trademarks of IBM International Group B.V., an IBM Company.

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

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

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.

Preface

IBM WebSphere® Message Broker is a lightweight, advanced enterprise service bus (ESB) that provides a broad range of integration capabilities that enable companies to rapidly integrate internal applications and connect to partner applications. Messages from business applications can be transformed, augmented and routed to other business applications. The types and complexity of the integration required will vary by company, application types, and a number of other factors.

Processing logic in WebSphere Message Broker is implemented using message flows. Through message flows, messages from business applications can be transformed, augmented, 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 perform tasks that are associated with message routing, transformation, and enrichment. Message flows are created and tested using the Message Broker Toolkit, a sophisticated, easy-to-use programming tool that provides a full range of programming aids.

This IBM® Redbooks® publication focuses on two specific integration requirements that apply to many midmarket companies. The first is the ability to use WebSphere Message Broker to integrate Microsoft.NET applications into a broader connectivity solution. WebSphere Message Broker V8 introduces the ability to integrate with existing Microsoft .NET Framework applications. A .NET assembly can be called from within a message flow and the WebSphere Message Broker runtime can host and run .NET code. Solutions explored in this book cover connectivity to applications using Windows Communications Framework (WCF), Microsoft Message Queuing, Microsoft Dynamics CRM, and other Microsoft applications.

The second is the ability to integrate WebSphere Message Broker with file transfer networks, specifically with WebSphere MQ File Transfer Edition and IBM Sterling Connect Direct.

The team who wrote this book

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

Vinicius D. G. Bragança has worked at IBM for eleven years in various Information Technology (IT) areas. Before joining the IBM Software Group, he worked for IBM Global Services as an Integration Architect. He has over 14 years of experience working on IBM Integration Technologies and has been involved with the planning, design, implementation, management, and problem analysis of IBM solutions. Now he is an I/T Architect Advisor for the IBM WorldWide WebSphere Business Partners Organization at the Latin America team in Sao Paulo, Brazil.

Minsung Byun is a Software Engineer at Research Triangle Park, North Carolina. He has been at IBM for nine years and the last six years of which were as a member of the WebSphere Message Broker Level 2 support team. He has a Bachelor's degree in Chemical Engineering from Inha University and a Masters of Science in Computer Science from Akron University.

David Crighton is the WebSphere Message Broker Level 3 Technical Delivery Lead working in the IBM Hursley Laboratory in the United Kingdom (UK). He has worked at IBM for six years spending time in the development, test, and service teams for WebSphere Message Broker. He provides final line support to WebSphere Message Broker customers including establishing best practices, problem analysis and resolution, defect support, and delivery of new function for in service releases.

Kiran Darbha is an Advisory Software Engineer working for IBM India Software Labs. He has nine years of experience in the IT industry with expertise in the Windows platform, Global Transactions, and WebSphere Service Oriented Middleware. Kiran is the Technical Leader for the WebSphere MQ .NET Stack and is currently involved with design and development of the .NET component for MQ to enable .NET developers access to WebSphere messaging. Apart from his primary responsibilities, he is actively involved in Innovation, New Technology developments, and Product consumability. Kiran has wide expertise and works extensively on various Microsoft .NET Technologies.

Jefferson Lowrey has more than 20 years of IT experience, ranging from desktop support through server administration and network configuration to architecting, developing, and maintaining enterprise applications and business-critical file transfer systems. Jeff has deep experience with a broad range of products in the WebSphere family, with a concentration on messaging and connectivity. Jeff has worked for financial services companies, mid-sized retailers, insurance companies, medical and pharmaceutical companies, and government agencies.

Jeff started programming computers at age 10, and got his first professional programming job at age 19 working on a team that wrote data analysis and collection software for cryogenic equipment at NASA Goddard Space Flight center. He went on to job roles in help desk and various other support and programming roles on the Macintosh platform until he started writing Windows software to encrypt, decrypt, and transfer files over FTP and SMTP for a student loan company. This led to him learning WebSphere MQ and Message Broker as a means of routing and identifying data to be sent to the correct external and internal parties. Within five years, Jeff was an internationally recognized expert on these products, and was hired by IBM as a consultant with a focus on Message Broker.

Jeff has experience working with a wide range of WebSphere products, including App Server, Portal Server and with IBM DataPower® to a lesser extent. Jeff brings deep skills in software architecture, design, implementation, administration, and management to all of his projects.

Pavel Maljutin is an IBM Certified Level 2 (Master) IT Specialist at IBM Russia, Moscow. He has over 12 years of experience in IT, and he joined IBM six years ago. Pavel has a Specialist degree in Information Science & Software from the Russian National Research University of Electronic Technology/ MIET. His areas of expertise include Java/C++ programming, building distributed systems using WebSphere branded software, consulting customers from Russia and CIS countries, and conducting workshops for WebSphere developers and administrators.

Abhinav Priyadarshi works for the WebSphere Message Broker Development team at IBM India, Bangalore. He has 11 years of experience with IBM. Abhinav develops new features and nodes for WebSphere Message Broker's new releases. He also provides consultancy on WebSphere Message broker proof of concepts (POCs), EAI design, and WebSphere Message Broker implementation for various customers in India and ASIA Pacific. He successfully completed POCs in various customer engagements and performed the implementation using WebSphere Message Broker and related WebSphere products in various EAI customer solutions.

Rashmi Katagall works with the WebSphere Message Broker development team at IBM India in Bangalore, with over four years of experience. She is responsible for testing the new

features that go into WebSphere Message Broker's new releases. She is also the author of the IBM Developer Works article “Configure secure communications with WebSphere Application Server and WebSphere Message Broker using SAML 2.0 tokens and Tivoli® Federated Identity Manager”.

Carla Sadtler is a Consulting IT Specialist at the ITSO, Raleigh Center. She writes extensively about WebSphere products and solutions. Before joining the ITSO in 1985, Carla worked in the Raleigh branch office as a Program Support Representative, supporting IBM MVS™ customers. She has a degree in Mathematics from the University of North Carolina at Greensboro.

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

Stephen Smith
Deana Coble
Shari Deiana
International Technical Support Organization, Raleigh Center

Mallanagouda B Patil
IBM India

John A Reeve
IBM UK

Matthew Golby-Kirk
IBM UK

Matthew Sunley
IBM UK



Figure 0-1 The team on site in RTP from left to right: Minsung Byun, Kiran Darbha, Vinicius Dias Gomes de Braganca, David Crighton, Carla Sadtler, Pavel Malyutin, Jeff Lowrey

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your

network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Obtain 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 publications in one of the following ways:

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

ibm.com/redbooks

- ▶ Send your comments in an email 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

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:

<http://www.facebook.com/IBMRedbooks>

- ▶ Follow us on Twitter:

<http://twitter.com/ibmredbooks>

- ▶ Look for us on LinkedIn:

<http://www.linkedin.com/groups?home=&gid=2130806>

- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>

- ▶ Stay current on recent Redbooks publications with RSS Feeds:

<http://www.redbooks.ibm.com/rss.html>



WebSphere Message Broker for mid-market solutions

WebSphere Message Broker is a lightweight, advanced enterprise service bus (ESB) that enables the integration of data sources from a wide range of platforms across both service-oriented architecture environments (SOA) and non-SOA environments.

With WebSphere Message Broker, organizations of any size can eliminate point-to-point connections and batch processing. Moving forward in this way increases business flexibility and smart interoperability of systems regardless of platform, protocol, or data format.

WebSphere Message Broker is frequently associated with large enterprise solutions because of the capabilities for connecting to enterprise information systems, such as SAP, Siebel, IBM CICS®, and others. However, IBM also recognized a growing need for ESB capabilities in midmarket companies.

With this in mind, new features and editions of WebSphere Message Broker provide targeted capability for this growing segment.

1.1 WebSphere Message Broker

You can use IBM WebSphere Message Broker to connect applications together, regardless of the message formats or protocols that they support.

This connectivity means that your diverse applications can interact and exchange data with other applications in a flexible, dynamic, and extensible infrastructure. WebSphere Message Broker routes, transforms, and enriches messages from one location to any other location:

- ▶ The product supports a wide range of protocols: WebSphere MQ, JMS 1.1, HTTP and HTTPS, Web Services (SOAP and REST), File, Enterprise Information Systems (including SAP and Siebel), and TCP/IP.
- ▶ It supports a broad range of data formats: Binary formats (C and COBOL), XML, and industry standards (including SWIFT, EDI, and HIPAA). You can also define your own data formats.
- ▶ It supports many operations, including routing, transforming, filtering, enriching, monitoring, distribution, collection, correlation, and detection.

Message Broker works with various hardware and software to match, route, convert, transform, identify, and distribute data. Figure 1-1 provides an overview of these functions.

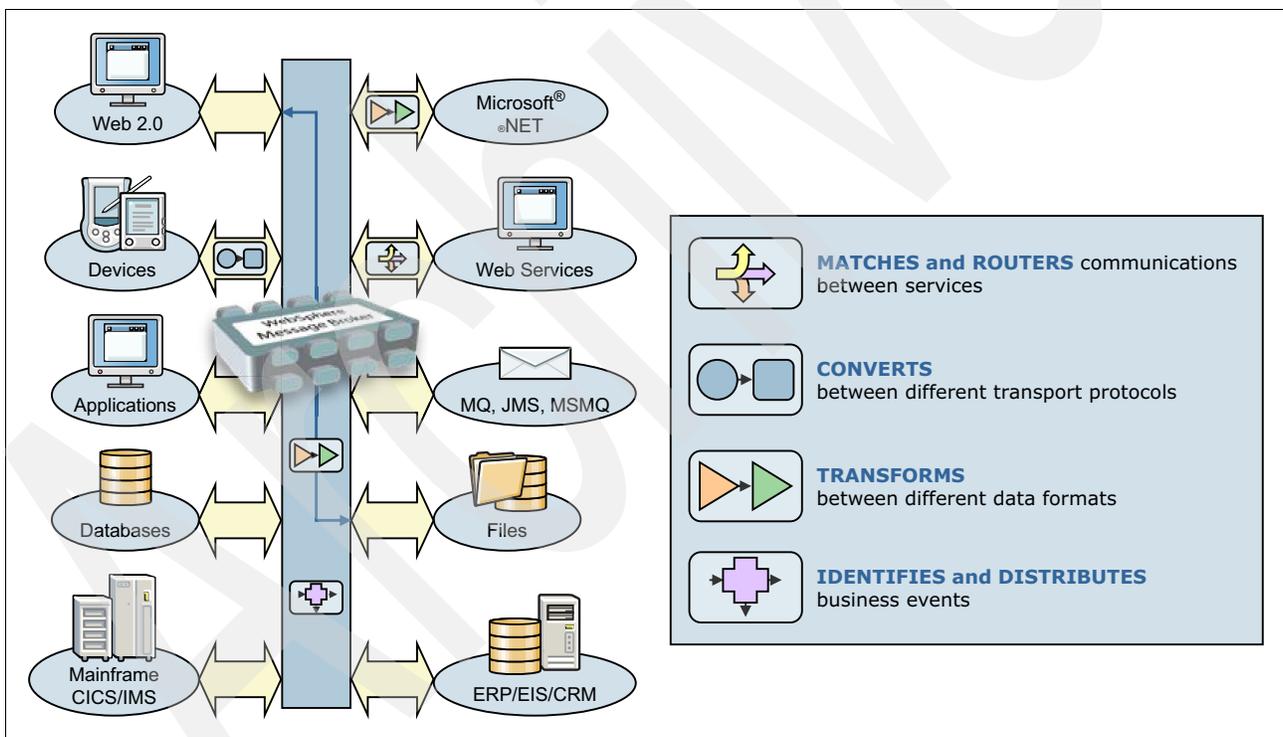


Figure 1-1 Message Broker overview

1.1.1 Message processing

Processing logic in WebSphere Message Broker is implemented using *message flows*. Through message flows, messages from business applications can be transformed, augmented, 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 perform tasks that are associated with message routing,

transformation, and enrichment. Message flows are created and tested using the WebSphere Message Broker Toolkit, which is a sophisticated, easy-to-use programming tool that provides a full range of programming aids.

The base capabilities of WebSphere Message Broker are enhanced by SupportPacs that provide a wide range of additional enhancements.

Message routing

WebSphere Message Broker provides 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 based on matching the content of the message with a pattern that is specified in a node.

Aggregation is an advanced form of message routing. With aggregation, a request message is received, and *multiple* new 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.

Collection is another advanced form of message routing. With collection, a set of related individual messages is assembled through a time-independent process and a single message or response structure is created. WebSphere Message Broker handles all of the details of storing incoming messages and correlating them together to produce a completed structure based on the requirements. Further transformation logic can then be applied to the results to produce the necessary output.

Transport protocol conversion

WebSphere Message Broker enables simplified connectivity between applications or business processes that use disparate transport protocols, including Web Services (SOAP and REST), HTTP(S), Java Message Service (JMS) 1.1, WebSphere MQ, file transfer protocols, CICS, IBM IMS™, TCP/IP, Service Component Architecture (SCA), enterprise information systems (SAP, Siebel, JD Edwards, PeopleSoft), and user-defined transports.

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. Enriched in-flight messages also provides a powerful mechanism to unify organizations because business information can now be distributed to applications that handle completely separate 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 Broker Toolkit contains facilities for defining messages to the WebSphere Message Broker. Message Broker can import and reuse message definitions that are developed using XML schemas and message definitions from previous versions of the product. Message Broker version 8 introduces a new parser and message definition model using the Data Format Description Language (DFDL). DFDL can create models that represent complex text and binary message formats including COBOL records, CSV structures, industry standard (SWIFT, HIPAA) messages, and more. DFDL can be used to model any message that is not defined using an XML schema. The DFDL parser and modeler are faster and more advanced than the previous TDS and CWF functions in WebSphere Message Broker version 7.

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), Java, and PHP facilities. These facilities are provided in the Message Broker Toolkit or using a .Net program developed using Microsoft Visual Studio 2010.

WebSphere Transformation Extender can be integrated into the WebSphere Message Broker ESB solution to further extend the existing capabilities.

1.1.2 Connectivity options

WebSphere Message Broker provides support for a variety of messaging transport protocols through a series of built-in nodes. These nodes enable robust and scalable applications to be developed without having to write code to support individual transport functions. The available nodes for connectivity include:

- ▶ WebSphere MQ

The WebSphere MQ Enterprise Transport supports WebSphere MQ applications that connect to WebSphere Message Broker to benefit from message routing and transformation options.

- ▶ Java Message Service (JMS) 1.1

The WebSphere Message Broker JMS Transport sends and receives JMS messages using destinations that are accessible through a JMS provider. The built-in JMS nodes act as JMS clients. The JMS nodes work with WebSphere MQ JMS provider, WebSphere Application Server, and any other JMS provider that conforms to the Java Message Service Specification V1.1.

- ▶ WebSphere MQ/JMS

WebSphere Message Broker provides support that includes the ability to transform MQ messages to JMS and vice versa.

- ▶ SOAP-based Web services

The WebSphere Message Broker Web services support includes both provider and consumer scenarios, WS-Security and WS-Addressing support, WS-RM (Web Services Reliable Messaging), and integration with DataPower appliances for Web service security. Integrated support for WebSphere Service Registry and Repository is also provided. Message flows can access, use, and select specific service registry entities dynamically at runtime.

- ▶ HTTP and HTTPS

The WebSphere Message Broker HTTP Transport connects applications that use the HTTP protocol. Message flows can use the HTTP transport to work with SOAP-based Web services, other Web services standards, such as REST or XML-RPC, and general HTTP messaging, where the payload might or might not be XML.

- ▶ File

File processing support is provided for:

- File processing of local (to the broker) files or remote files using FTP and SFTP
- Managed file transfer using WebSphere MQ File Transfer Edition
- Managed file transfer using IBM Sterling Connect:Direct®

- ▶ **Java Connector Architecture (JCA) Adapters**
The WebSphere Adapters nodes can be used to communicate with enterprise information systems (EIS). WebSphere Message Broker provides message flow nodes that support connectivity with an SAP server, TwineBall, Siebel Business Applications server, PeopleSoft Enterprise, and JD Edwards EnterpriseOne applications.
- ▶ **TCP/IP**
WebSphere Message Broker provides support for TCP/IP connectivity of existing applications that use raw TCP/IP sockets for transferring data.
- ▶ **Service Component Architecture**
SCA nodes allow interoperability with WebSphere Process Server. The SCA nodes support inbound and outbound scenarios with WebSphere Process Server and WebSphere Enterprise Service Bus.
- ▶ **CICS Transaction Server for IBM z/OS®**
Message flows can create a client connection with CICS Transaction Server for z/OS either directly or through CICS Transaction Gateway for Multiplatforms. Message flows can call programs that are running in the target CICS region. CICS can also be accessed through web services or over WebSphere MQ.
- ▶ **IBM Information Management System (IMS)**
Message flows can create a client connection with IMS using IMS Connect, WebSphere MQ-IMS bridge, and the IMS SOAP Gateway.
- ▶ **Email**
Message flows can send or receive emails. Email can be delivered to an email server that supports SMTP. Email can be received from an email server that supports Post Office Protocol 3 (POP3) or Internet Message Access Protocol (IMAP).
- ▶ **CORBA**
Use CORBA nodes in message flows to connect to CORBA Internet Inter-Orb Protocol (IIOP) applications.
- ▶ **Databases**
Message flows can access user databases on IBM DB2®, IBM Informix®, Microsoft SQL Server, Oracle, IBM solidDB®, and Sybase. Transactional (XA) and non-transactional connections are supported. Both ODBC and JDBC type 4 connections are supported.
- ▶ **Routing nodes**
Use the Route node to direct messages that meet certain criteria down different paths of a message flow. Route nodes can be directed to filter, publish, subscribe, label, aggregate, collect, sequence, and re-sequence.
- ▶ **Transformation nodes**
Use the transformation nodes to transform a message from one type to another and support integration. Transformation nodes include Microsoft .Net, mapping, XSL Transform, Compute, JavaCompute, and PHPCompute.
- ▶ **Construction nodes**
Use construction nodes to manage and create subflows. These subflows can control the order of messages, force message flow during error processing or exceptions, parsing, and versions of subflow. Construction nodes include input, output, trace, throw, trycatch, floworder, pass-through, and reset content descriptor.

- ▶ **Validation nodes**
Use the validation node to check that the message that arrives is as expected (message domain, message set, and message type) and to check that the content of the message is correct by selecting message validation. The check node is deprecated in Version 6 and higher and can be replaced with validation nodes.
- ▶ **Security nodes**
Use the SecurityPEP node to invoke the message flow security manager at any point in the message flow. These can be placed between input, output, and request nodes.
- ▶ **Timer nodes**
The TimeoutControl node processes an input message with a timeout request. It validates the timeout request message, stores the message, and propagates the message (unchanged) to the next node in the message flow. The TimeoutNotification node can be paired with a TimeoutControl node for events that must happen at a specific time, or stand-alone based on the timing function desired.

1.1.3 WebSphere Message Broker editions

WebSphere Message Broker is now available in a number of different editions. Each edition is designed to provide targeted capability to address the key integration requirements of companies within a particular market segment.

WebSphere Message Broker also supports a seamless upgrade from one edition of the product to the next without the need to reinstall. The ability to seamlessly upgrade provides businesses with the flexibility to change their original choice of edition as their integration needs grow. The following list outlines the various WebSphere Message Broker Editions:

- ▶ *WebSphere Message Broker Trial Edition*

The 90-day Trial Edition can be downloaded from the web at no charge. This edition enables you to evaluate all of the capabilities of WebSphere Message Broker to decide which edition is most appropriate for your needs.

- ▶ *WebSphere Message Broker Express Edition*

The Express Edition provides simple, robust connectivity for small businesses or single departments. This connectivity includes the ability to integrate common applications and data sources. It also includes the ability to perform basic data transformation and mapping.

A list of the message flow nodes that are available in this edition are in the following WebSphere Message Broker Information Center topic:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an67720_.htm.

- ▶ *WebSphere Message Broker Standard Edition*

The Standard Edition augments Express to provide additional integration capabilities required by medium-sized businesses. These integration capabilities include the ability to perform more complex operations on data, for example, to aggregate data from multiple sources or sequence messages to ensure that data arrives in a consistent manner. It also includes connectivity to mainframe systems, such as CICS or IMS, and to CRM, ERP, EIS systems.

- ▶ *WebSphere Message Broker*

WebSphere Message Broker augments Standard to provide the performance, scalability, and management required by large enterprises. This includes the ability to handle large volumes of business-critical traffic and provide separation of individual department's data for security, maintenance, or other operational management purposes.

- ▶ *WebSphere Message Broker Remote Adapter Deployment*

The Remote Adapter Deployment Edition provides a specific solution for businesses that require basic connectivity to a broad range of endpoints, including applications, such as ERP or EIS systems.

A list of the message flow nodes that are available in this edition are in the following WebSphere Message Broker Information Center topic:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an67720_.htm.

1.2 WebSphere Message Broker for midmarket

WebSphere Message Broker provides a broad range of integration capabilities that enable companies to rapidly integrate internal applications and connect to partner applications. The types and complexity of the integration required vary by company, application types, and a number of other factors.

This book focuses on two specific integration requirements that apply to many midmarket companies:

- ▶ The first is the ability to use WebSphere Message Broker to integrate Microsoft .NET applications into a broader connectivity solution.
- ▶ The second is the ability to use WebSphere Message Broker in conjunction with WebSphere MQ File Transfer Edition to transfer and use business data held in files.

1.2.1 Microsoft .Net integration

WebSphere Message Broker V8 introduces the ability to integrate with existing Microsoft .NET Framework applications. A .NET assembly can be called from within a message flow and the WebSphere Message Broker runtime can host and run .NET code.

A new node called the .NETCompute node was added to allow the message data to be processed using a Common Language Runtime (CLR)-compliant .NET programming language. Some CLR compliant languages options are C#, Visual Basic (VB), F#, or C++/Common Language Infrastructure (CLI). This new node can be used to build messages, interact with .NET or COM applications, transform messages from one format to another, copy messages, route messages, and modify messages. The node must be configured with a .NET assembly file that contains the code that performs the logic of the node. The Message Broker Toolkit provides integration with Microsoft Visual Studio 2010, allowing you to develop .NET solutions in Visual Studio and integrate them into the message flow.

The WebSphere Message Broker runtime hosts and runs .NET code that is associated with a WebSphere Message Broker solution. The .NET code is not executed in a stand-alone process. Message flows containing .NET code can only be deployed and run successfully on brokers that are on Windows platforms.

WebSphere Message Broker can also integrate with Microsoft .NET applications that expose a Windows Communication Framework web service interface. This is less efficient than direct invocation through the .NETCompute node, but enables non-Windows brokers to remotely invoke .NET business logic.

WebSphere Message Broker can act as a bridge between IBM messaging technology (WebSphere MQ) and Microsoft messaging technology (MSMQ). This bridge enables the creation of solutions that span the two messaging environments.

1.2.2 File processing and transfer

Many business applications and user-based processes consume and save data as files. It is a common practice for companies of all sizes to create lots of small applications and processes to ensure that files created by one application are moved to the correct places to be consumed by other applications and users. This leads to a complicated IT infrastructure and multiple, different solutions for management and operational awareness of issues with these transfers.

WebSphere Message Broker provides the capability to integrate file-based business applications, replacing manual, script-based or batch file processing. Files can be read and written locally or remotely using FTP or SFTP. WebSphere MQ File Transfer Edition augments this by providing secure, reliable delivery of files to and from WebSphere Message Broker in place of FTP or SFTP.

1.3 Additional resources

- ▶ IBM for Midsize Business on Facebook
http://www.facebook.com/MidmarketIBM?v=app_228046053890511#!/MidmarketIBM?sk=wall
- ▶ Using Microsoft .NET in WebSphere Message Broker V8 - DeveloperWorks tutorials
http://www.ibm.com/developerworks/views/websphere/libraryview.jsp?search_by=Using+Microsoft+.NET+in+WebSphere+Message+Broker+V8



Introduction to WebSphere Message Broker V8

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 functionality.

Specifically, we cover the following topics:

- ▶ Runtime architecture of WebSphere Message Broker
- ▶ Development environment of WebSphere Message Broker
- ▶ Connectivity options
- ▶ Transformation interfaces
- ▶ Administering WebSphere Message Broker
- ▶ Deploying applications
- ▶ Getting started
- ▶ Creating WebSphere MQ queue managers and queues

2.1 Runtime architecture of WebSphere Message Broker

WebSphere Message Broker consists of a development environment where message flows and message sets are designed and developed and a runtime environment where the message flows executes. Figure 2-1 shows an overview of the WebSphere Message Broker architecture.

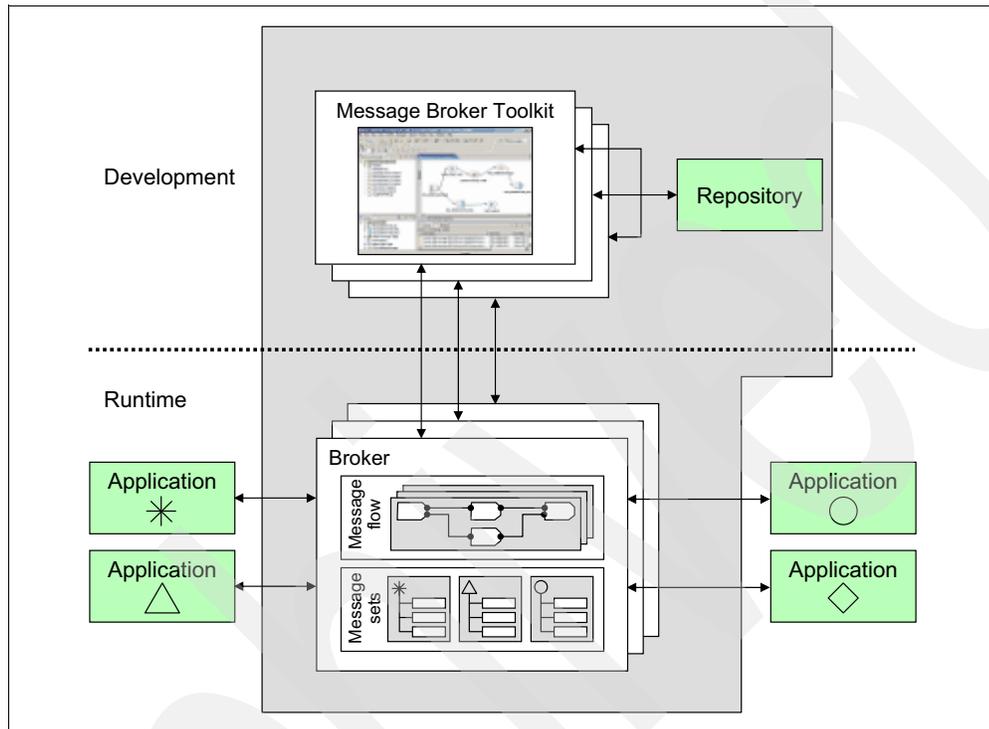


Figure 2-1 Architecture of WebSphere Message Broker

2.1.1 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 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 an internal repository in the local file system to store the message flows, configuration data, and the message sets that are deployed to it.

2.1.2 Execution groups

Execution groups are processes that host message flows. The execution groups facilitate the grouping of message flows within the broker with respect to functionality, load balancing, or other necessary qualifications. Each broker contains a main execution group. Additional execution groups can be created if 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. Execution

groups can also be used to scale the broker, for example, in multi-processor platforms, different execution groups can be assigned to various processors.

Message flow applications are deployed to a specific execution group. To enhance performance, additional message flow instances can be deployed to an execution group. Each additional instance allocates an additional thread to host the message flow application. The number of threads can be increased using a per message flow basis, based on the processing requirements of that message flow. For each additional instance, an additional thread is started in the execution group to process the incoming messages at run time.

Additional information: For more information about designing for performance in WebSphere Message Broker, refer to the SupportPacs IP04 at:

http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24006518&loc=en_US&cs=utf-8&lang=en

2.2 Development environment of WebSphere Message Broker

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

2.2.1 WebSphere Message Broker Toolkit

The WebSphere Message Broker Toolkit is an integrated development environment (IDE) and graphical user interface (GUI) based on the Eclipse platform. Application developers can use the Message Broker Toolkit to create message flows and the associated artifacts and to deploy them to the execution groups.

Broker Application Development perspective

The Broker Application Development perspective is the default perspective that is displayed the first time that you start the Message Broker Toolkit. Application developers work in this perspective to develop and modify message sets and message flows. Figure 2-2 on page 12 shows the Broker Application Development perspective with a message flow open in the Message Flow editor.

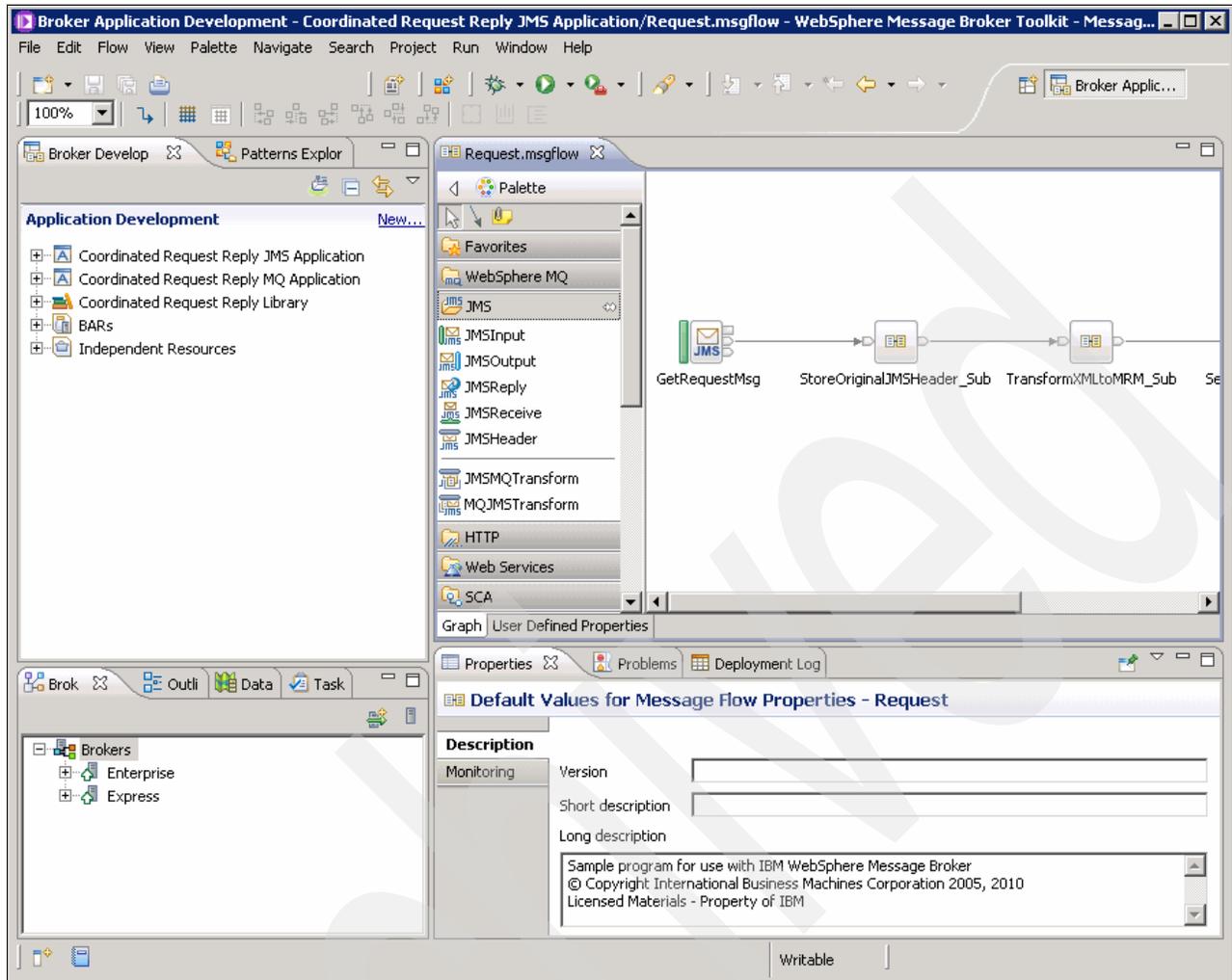


Figure 2-2 Application development perspective

Figure 2-2 shows that the JMS folder within the Palette is open in the Message Flow Editor.

The flow, shown in Figure 2-2, also starts with a JMSInput node. The middle terminal of this node, which is the *out* terminal, is wired to the *input* terminal on a subflow. The out terminal of that subflow was again wired to the *in* terminal of another subflow. These wires indicate the logical steps that the message flow performs. Each node or subflow is invoked by a connection coming in to one of its input terminals and forwards a response to an output terminal, based from the logic performed inside the node itself.

2.2.2 Applications and libraries

Message Broker Version 8 introduces a new concept of packaging broker solutions. Every deployable artifact now belongs to an application or a library. Resources can be developed outside of an application or library but when deployed they are still part of a default application. Applications and libraries provide runtime isolation of source code and message models for an increased level of change control and fault isolation.

2.2.3 Message flows

Message flows are created by adding nodes to the workspace, defining properties for these nodes, and wiring the nodes together in a logical flow. Nodes are dragged onto the message flow document from the palette located on the right. Individual nodes have terminals that represent connection points to these nodes. Flow logic is built by adding lines between the terminals of each of the nodes in the flow to direct the order of processing. Each node then performs a specific function or type of function within the flow. Table 2-1 lists the nodes that the Message Broker Toolkit provides 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/v8r0m0/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 2-1 Available node types in a WebSphere Message Broker

Category	Input/output type node	Flow development nodes
WebSphere MQ	MQInput MQOutput	MQReply MQGet MQHeader MQOptimizedFlow (deprecated)
JMS	JMSInput JMSOutput	JMSReply JMSReceive JMSHeader MQJMSTransform JMSMQTransform
HTTP	HTTPInput HTTPRequest HTTPReply	HTTPHeader
Routing		Filter Label Publication RouteToLabel Route AggregateControl AggregateRequest AggregateReply Collector Resequence Sequence
Transformation		.NetCompute Mapping XSLTransform Compute JavaCompute PHPCompute

Category	Input/output type node	Flow development nodes
Construction		Input Output Throw Trace TryCatch FlowOrder Passthrough ResetContentDescriptor node
Database	DatabasInput	Database DatabaseRetrieve DatabaseRoute
Validation		Check (Deprecated) Validate
Timer		TimeoutControl TimeoutNotification
Corba		CorbaRequest
SCA	SCAInput	SCAReply SCARequest SCAAsyncRequest SCAAsyncResponse
Web Services	SOAPInput	SOAPReply SOAPRequest SOAPAsyncRequest SOAPAsyncResponse SOAPEnvelope SOAPExtract Registry Lookup Endpoint Lookup
WebSphere Adapters	PeopleSoftInput SAPInput SiebelInput JDEdwardsInput TwineBallInput	PeopleSoftRequest SAPRequest SAPReply SiebelRequest JDEdwardsRequest TwineBallRequest
File	FileInput FileOutput FTEInput FTEOutput CDInput CDOOutput	FileRead
Email	EmailInput EmailOutput	
TCP/IP	TCPIPClientInput TCPIPClientOutput TCPIPServerInput TCPIPServerOutput	TCPIPClientReceive TCPIPServerReceive
CICS		CICSRequest
IMS		IMSRequest

Category	Input/output type node	Flow development nodes
Security		SecurityPEP

2.2.4 Message

A *message* is a collection of data that one application sends to another. This data can be business data or elements that are arranged in a predefined structure or a sequence of bytes.

When a message arrives at a message flow input node in a broker, the parser that is configured at the input node interprets and creates a logical tree representation. This tree representation is created from the bitstream of the message data, which is known as the message assembly. The tree format can then be easily manipulated and updated within the message flow using ESQL, Java, PHP or .NET. After the message is processed, the parser converts it back into a bitstream at the output. The message assembly consists of four tree structures:

- ▶ **Message tree structure:** The message tree includes all of the headers in the message, properties subtree, and the message body.
- ▶ **Environment tree structure:** The environment tree stores information in the form of variables as the message passes through the message flow. It is typically used to store information regarding accounting and statistics or correlation attributes that are associated with broker monitoring events. The environment tree is always included in the message and its contents are retained in the output message.
- ▶ **Local environment tree structure:** This tree also stores information in the form of variables; however, these variables can be referenced or updated in the message flow. Local environment tree structure is composed of several subtrees, such as variables, destination, file, SOAP, service-registry, wildcard, adapter, TCP/IP, or a written destination.
- ▶ **Exception list tree structure:** This tree contains information about exceptions that can occur during message processing. The exception list tree is typically built when an exception is thrown and the message processing is suspended.

Additional information: For more information about the Logical tree structure, refer to the following topic in the Information Center:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp?topic=%2Fcom.ibm.etools.mft.doc%2Fac00490_.htm

2.2.5 Message modeling

Message modeling is a way to predefine the message formats that are used by applications in the broker infrastructure. Modeling is usually used for message formats that are not self-defining because the parser must have access to the predefined model that describes the message. Modeling might also be used for runtime validation of incoming messages and enhanced parsing of XML messages. Message models construct the logical message tree from raw data and serialize a logical message tree into a bitstream.

WebSphere Message Broker V8 includes a new modeler and parser built around the open standard Data Format Description Language (DFDL). Previous models built using Message Sets can be reused in WebSphere Message Broker V8 but cannot be created.

Message models in version 8 are stored in XML schema files. DFDL describes the contents of non-XML formatted data using XML schemas, so all message models can be stored and deployed in a common format.

WebSphere Message Broker V8 includes sophisticated tooling around DFDL, including the ability to test a DFDL model directly in the Message Broker Toolkit. This function allows you to test parsing a message and serializing a message tree. This can significantly shorten the time it takes to develop and prove out a message model.

Message modeling concepts

All messages arrive in the broker as binary data. To understand them, the broker uses parsers that split the message into logical parts and create a *logical message model* instance. The logical message model instance in the broker is always represented as a tree, with one root element that corresponds to the name of the parser used.

There are two different kind of parsers in WebSphere Message Broker:

- ▶ **Programmatic:** These parsers understand the message model because the message model is built into them. Each message format needs a specific parser. Examples of these parsers are:
 - **MIME:** Parses internet MIME messages form.
 - **BLOB:** Treats messages as binary data. This is the default parser.
 - **XMLNSC:** Creates a logical structure from an XML message. If you do not specify a message set, all elements and attributes are treated as string, so additional casting is required during processing.
- ▶ **Descriptive:** With this type of parser, the message format is encoded in a model. A general purpose parser program uses the model when parsing. The model can be used as-is or generated into code. Examples model-driven parsers include:
 - **DFDL:** Data Format Description Language is an open-standard XML-based language used to define the structure of formatted data in a way that is independent from the data format itself. The DFDL domain can be used to parse and write a wide variety of message formats, and is primarily intended for non-XML message formats. WebSphere Message Broker provides tooling for modeling and testing DFDL data at design time. You can create a logical model of the message or import it from a file, test how it will be serialized into binary format, and test how binary data will be parsed into the logical model. Because all of these steps are executed at design time and do not require the broker, DFDL development is easier and faster than using MRM. DFDL is not intended to be used to model XML documents. Use normal XML Schema files to model XML.
 - **MRM:** Message Repository Manager (MRM) is the predecessor of DFDL and uses similar techniques for describing the structure of formatted data using a type of XSD schema, except that it can also model XML data. The MRM format of data definition was only used in WebSphere Message Broker.
 - **XMLNSC with XML schema:** The XMLNSC parser can execute a validation of the message against the message model and, optionally, can cast all data from string representation to actual data type from an XML schema. The XMLNSC parser can be used both as a programmatic parser (a generic XML parser) and as a descriptive parser. This is used when you create the XML schema for the message, deploy it to the broker, and select message validation in the message flow.

2.2.6 Pattern instances

Patterns are reusable solutions that encapsulate a tested approach in solving common architectural, deployment, and design tasks in a particular context. Patterns can be used to generate customized solutions to a recurring problem in a more effective way. Patterns in WebSphere Message Broker can be referenced to provide guidance in implementing solutions. Pattern specifications typically include the description of identified solutions. Patterns greatly reduce the development cycles because the resources can be generated from a set of predefined templates, thus increasing the efficiency. Patterns improve quality through asset reuse and the implementation of error handling and logging functions.

The Message Broker Toolkit displays pattern instances in the Patterns Explorer view, which provides a list of patterns and details about each of the patterns. The description in the help can guide you towards a suitable pattern to solve a particular problem. Figure 2-3 shows the list of available patterns that are provided in the Pattern Explorer.

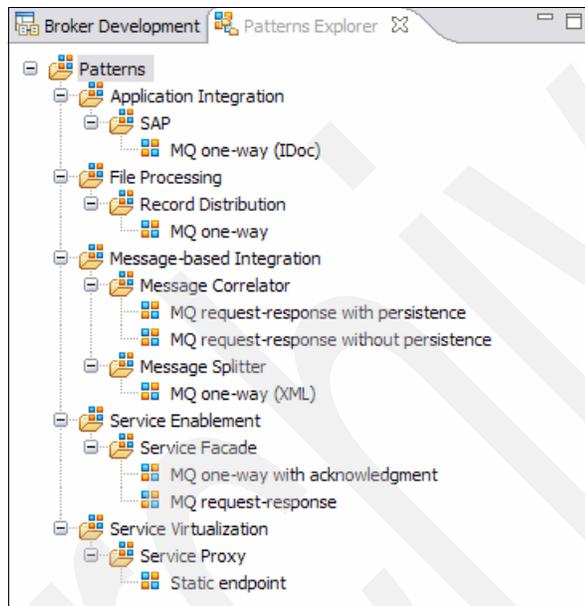


Figure 2-3 Patterns Explorer

Patterns: For more information about using patterns and pattern categories, search *patterns* in the WebSphere Message Broker Information Center at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp?topic=/com.ibm.etools.mft.doc/ac67850_.htm

You can get additional information about WebSphere Message Broker Patterns from the following developer works articles:

- ▶ <http://www.ibm.com/developerworks/wikis/display/esbpatterns/>
- ▶ <http://www.ibm.com/developerworks/library/ws-enterpriseconnectivitypatterns/index.html>
- ▶ http://www.ibm.com/developerworks/websphere/library/techarticles/0910_phillips/0910_phillips.html

2.2.7 Microsoft Visual Studio 2010 integration

The Message Broker Toolkit integrates with Microsoft Visual Studio 2010 for developing applications that use the .NET Common Language Runtime either from ESQL procedures or from a .NETCompute node.

The Message Broker Toolkit integrates the ability to rapidly open Visual Studio solutions that are associated with .NETCompute nodes by simply double-clicking the node. The Visual Studio debugger can also be attached to the broker execution group process and used in conjunction with the broker debugger to perform step-through debugging of flows and code. The Message Broker Toolkit also allows you to drag and drop DLLs associated with Visual Studio solutions onto the message flow to create a .NETCompute node configured for the solution.

2.3 Connectivity options

WebSphere Message Broker provides support for a variety of messaging transport protocols. These protocols include WebSphere MQ, Java Message Service (JMS), WebSphere MQ/JMS, SOAP-based Web services, HTTP and HTTPS, File and Managed File Transfer, Java Connector Architecture (JCA) adapters for SAP, Siebel and PeopleSoft, TCP/IP, and Service Component Architecture (SCA).

In this section, we introduce the various transport protocols that WebSphere Message Broker supports. The appropriate connection points are described for each transport type that client applications can use to both send messages to and receive messages from the WebSphere Message Broker.

2.3.1 WebSphere MQ

The WebSphere MQ Enterprise Transport supports WebSphere MQ applications that connect to WebSphere Message Broker to benefit from message routing and transformation options.

The WebSphere MQ nodes put and get messages from the queue manager that is associated with the broker that is running the flow. If you need to read or write messages from other queue managers, use WebSphere MQ configuration to connect the queue managers and move messages over the WebSphere MQ Enterprise Transport. You can also configure the JMS nodes to use WebSphere MQ as a JMS provider and use a client connection to a remote queue manager.

MQInput and MQGet nodes

Use the MQInput node if the messages arrive at the broker on a WebSphere MQ queue and the node is to be at the start of a message flow.

Use the MQGet node to retrieve a message from a WebSphere MQ queue, if you intend to get the message after the start of the message flow.

MQOutput and MQReply nodes

Use the MQOutput node if the target application expects to receive messages on a WebSphere MQ queue.

Use the MQReply node if the target application expects to receive messages on the WebSphere MQ reply-to queue that is specified in the input message MQMD.

Figure 2-4 shows a sample message flow that makes use of the MQInput, MQGet, MQReply, and MQOutput nodes.

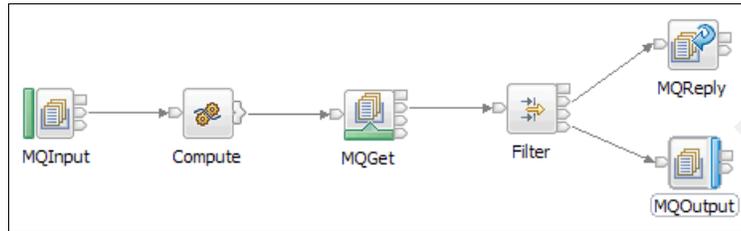


Figure 2-4 The MQ nodes

For additional information about configuring and using each of the WebSphere MQ nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to the following topics:

- ▶ MQInput node (ac04560_)
- ▶ MQGet node (ac20806_)
- ▶ MQOutput node (ac04570_)
- ▶ MQReply node (ac04580_)

2.3.2 Java Message Service (JMS) 1.1

The WebSphere Message Broker JMS Transport sends and receives JMS messages using destinations that are accessible through a JMS provider. The built-in JMS nodes act as JMS clients.

The JMS nodes work with WebSphere MQ JMS provider, WebSphere Application Server (V6 and subsequent) service integration bus, and any other JMS provider that conforms to the Java Message Service Specification V1.1. WebSphere Message Broker supports Java 6 (Version 1.6), so the JMS provider classes must be compliant with that version.

JMSInput node

Use the JMSInput node if a JMS application sends the messages. The JMSInput node acts as a JMS message consumer and can receive all six message types that are defined in the JMS Specification V1.1.

JMSOutput and JMSReply nodes

Use the JMSOutput node, if the messages are for a JMS destination. The JMSOutput node acts as a JMS message producer and can publish all six message types that are defined in the JMS Specification V1.1.

The JMSReply node has a similar function to the JMSOutput node, but the JMSReply node sends JMS messages to only the reply destination that is supplied in the JMSReplyTo header field of the JMS message tree. Use the JMSReply node to treat a JMS message that is produced from a message flow as a reply to a JMS input message and when you have no other routing requirements.

For additional information about configuring and using each of the JMS nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to the following topics:

- ▶ JMSInput node (ac24820_)
- ▶ JMSOutput node (ac24830_)
- ▶ JMSReply node (ac37130_)

2.3.3 WebSphere MQ/JMS

WebSphere Message Broker provides support that includes the ability to transform MQ messages to JMS and vice versa.

WebSphere Message Broker can receive messages that have a WebSphere MQ JMS provider message tree format and transform them into a format that is compatible with messages that are to be sent to a JMS client application.

Conversely, WebSphere Message Broker can transform a message with a JMS message tree structure into a message that is compatible with the format of messages that are produced by the WebSphere MQ JMS provider.

MQJMSTransform node

Use the MQJMSTransform node if the input message is of the WebSphere MQ JMS provider format and the output message is to be sent to a JMS client application.

Figure 2-5 shows a sample message flow, which takes an MQ input message, transforms the message using the MQJMSTransform node, and outputs a JMS message.



Figure 2-5 The MQJMSTransform node

JMSMQTransform node

Use the JMSMQTransform node if the input message is a JMS message and the output message must be in the format of messages that are produced by the WebSphere MQ JMS provider.

Figure 2-6 shows a sample message flow, which takes a JMS input message, transforms the message using the JMSMQTransform node, and outputs an MQ message.



Figure 2-6 The JMSMQTransform node

For additional information about configuring and using each of the JMSMQ/MQJMS Transform nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to the following topics:

- ▶ MQJMSTransform node (ac24850_)
- ▶ JMSMQTransform node (ac24840_)

2.3.4 SOAP-based web services

The WebSphere Message Broker Web services support includes provider and consumer scenarios, WS-Security and WS-Addressing support, and integration with DataPower appliances for Web service security. The standards that are supported include SOAP 1.1/1.2, WSDL 1.1, MTOM/XOP, SOAP with Attachments, and are Basic Profile 1.1 compliant.

Integrated support for WebSphere Service Registry and Repository (WSRR) is also provided. Message flows can access, use, and select specific service registry entities dynamically at runtime.

SOAP input and request nodes

The SOAPInput, SOAPRequest, and SOAPAsyncRequest nodes can be used to process client SOAP messages and to configure the message flow to behave like a SOAP Web Services provider.

The SOAPInput node processes client SOAP messages, thereby enabling the broker to be a SOAP Web Services provider. It is typically used in conjunction with the SOAPReply node.

The SOAPRequest node sends a SOAP request to a remote Web service. The node is a synchronous request and response node and blocks after sending the request until the response is received.

The SOAPAsyncRequest node sends a Web services request, but the node does not wait for the associated Web service response to be received. It does, however, wait for the HTTP 202 acknowledgment before continuing with the message flow and blocks if the acknowledgement is not received. The SOAPAsyncRequest node is used with the SOAPAsyncResponse node to construct a pair of message flows that call a Web service asynchronously.

SOAP reply and response nodes

The SOAPReply node is used if the target application expects to receive SOAP messages in response to a message that is sent to the SOAPInput node.

The SOAPAsyncResponse node is used to deliver a SOAP message in response to a message that is received by the SOAPAsyncRequest node.

Figure 2-7 shows a sample Web Service Provider message flow, which makes use of the SOAPInput and SOAPReply nodes.

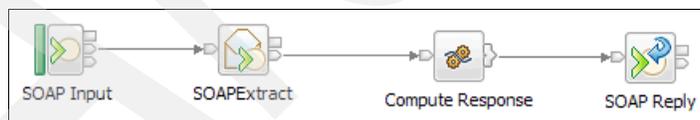


Figure 2-7 SOAP Web Service Provider message flow

Figure 2-8 shows a sample Web Service Consumer message flow, which makes use of the SOAPRequest node.



Figure 2-8 SOAP Web Service Consumer message flow

Figure 2-9 shows a sample Asynchronous Consumer message flow, which makes use of the SOAPAsyncRequest and SOAPAsyncResponse nodes.

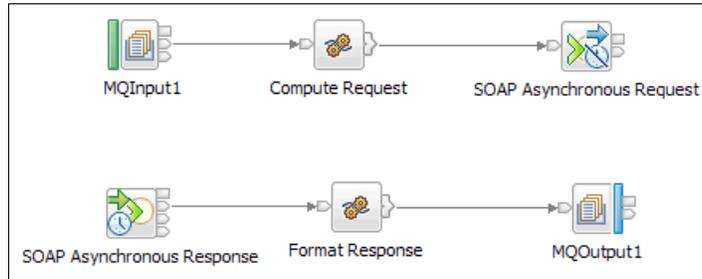


Figure 2-9 SOAP Asynchronous Consumer message flow

For additional information about configuring and using each of the SOAP nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ SOAPInput node (ac56170_)
- ▶ SOAPRequest node (ac56190_)
- ▶ SOAPAsyncRequest node (ac56200_)
- ▶ SOAPReply (ac56180_)
- ▶ SOAPAsyncResponse (ac56210_)

2.3.5 HTTP and HTTPS

The WebSphere Message Broker HTTP Transport connects applications that use the HTTP protocol. Message flows can use the HTTP transport to work with SOAP-based Web services, other Web services standards, such as REST or XML-RPC, and general HTTP messaging, where the payload might or might not be XML.

In the case of SOAP-based Web services, several advantages exist if you use the SOAP domain instead of the HTTP transport nodes. Using the SOAP domain and nodes provides the following advantages:

- ▶ Support for WS-Addressing, WS-Security, and SOAP headers
- ▶ A common SOAP logical tree format that is independent of the actual bitstream
- ▶ Run time checking against WSDL

HTTP input and request nodes

The HTTPInput node receives an HTTP message from an HTTP client for processing by a message flow. If you include an HTTPInput node in a message flow, you must either include an HTTPReply node in the same flow or pass the message to another flow that includes an HTTPReply node.

The HTTPRequest node interacts with a web service using all, or part, of the input message as the request that is sent to that service. This node can be used in a message flow that does or does not contain an HTTPInput or HTTPReply node. The HTTPRequest node is synchronous, which means it will wait until the HTTP service being contacted returns a reply.

If you use the HTTPInput node with the HTTPReply and HTTPRequest nodes, the broker can act as an intermediary for web services. Web service requests can also be transformed and routed in the same way as other message formats that WebSphere Message Broker

supports. Web service requests can be received either in standard HTTP (1.0 or 1.1) format or in HTTP over SSL (HTTPS) format. You can select the **Use HTTPS** property to handle either HTTP or HTTPS requests.

HTTPReply node

The HTTPReply node returns a response from the message flow to an HTTP client. This node generates the response to an HTTP client from which the input message was received by the HTTPInput node and waits for the confirmation that it was sent.

Figure 2-10 shows a sample Web Service Provider message flow, which makes use of the HTTPInput and HTTPReply nodes.

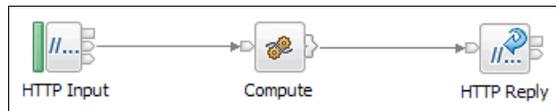


Figure 2-10 HTTP Web Service Provider message flow

Figure 2-11 shows a sample web service consumer message flow, which makes use of the HTTPRequest node.

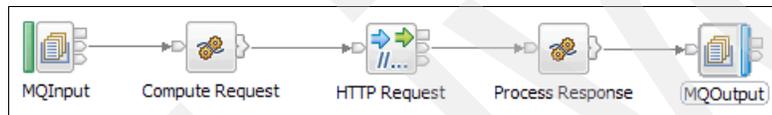


Figure 2-11 HTTP web service consumer message flow

For additional information about configuring and using each of the HTTP nodes, visit the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ HTTPInput node (ac04565_)
- ▶ HTTPRequest node (ac04595_)
- ▶ HTTPReply node (ac04585_)

2.3.6 File

File processing support, including FTP and SFTP, is delivered through built-in nodes. This support allows large files to be handled without using excessive storage. It provides comprehensive support for record detection and sophisticated record identification that allows specific records to be identified in a larger flow. All of the file nodes that act as Input nodes provide functionality to propagate individual records as new messages based on message model, fixed delimiters, or fixed length information. All of the file nodes that act as output nodes can assemble individual records into a complete file before file transfer is performed.

FileInput node

The FileInput node can be used in any message flow that must accept messages in files. One or more messages can be read from a single file, and each message is propagated as a separate flow transaction. A file can be a single record or a series of records.

Select **FTP** → **Transfer protocol property** to choose either FTP or SFTP.

In version 8, the FileInput node has increased ability to work with files located in subdirectories, so it can process files from different senders based on the subdirectory. However, each FileInput node can only be associated with one top-level directory. If multiple business partners are expected to submit data, it must all be delivered into folders that have a common root path.

FileOutput node

The FileOutput node can be used to write messages to files. The node writes files as a sequence of one or more records. Each record is generated from a single message that is received on the in terminal of the node. The FileOutput node will create a file in an identified location on the broker runtime's file system. It can be configured to then move the file to a remote location using FTP or SFTP.

A sample file transport message flow has a FileInput node connected to a FileOutput node. The FileInput node monitors the input directory and sends the data to a FileOutput node that writes the file to the output directory.

FTEInput node

The FTEInput node acts as a destination agent endpoint for files sent over the WebSphere MQ File Transfer Edition protocol. The FTEInput will receive files from any file transfer agent across the entire file transfer network.

FTEOutput node

The FTEOutput node acts as a source agent endpoint for files that are sent over the WebSphere MQ File Transfer Edition protocol. The FTEOutput node will output data to any file transfer agent across the entire file transfer network based on the destination information configured in the node properties or by the message flow logic.

CDInput node

The CDInput node acts as a receiving endpoint for files sent using IBM Sterling Connect:Direct. The node is configured to know the location of a specific Connect:Direct server. The node can be configured to process all of the files or a subset of the files transferred to that server based on the directory and file name.

CDOOutput node

The CDOOutput node acts as a sending endpoint for files that are sent using IBM Sterling Connect:Direct. The CDOOutput node is configured to know the location of a Primary Connect:Direct server (PNODE in Connect:Direct terminology) and a Secondary Connect:Direct server (SNODE in Connect:Direct terminology). The primary server will receive the file data from the CDOOutput node and then transfer it to the secondary server using Connect:Direct file transfer services.

For additional information about configuring and using each of the file nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ FileInput node (ac55150_)
- ▶ FileOutput node (ac55160_)
- ▶ FTEInput node(bc34034_)
- ▶ FTEOutput node(bc34030_)
- ▶ CDInput node(bc14020_)
- ▶ CDOOutput node(bc14015_)

2.3.7 Java Connector Architecture (JCA) Adapters and WebSphere adapters

The WebSphere Adapters nodes can be used to communicate with enterprise information systems (EIS). WebSphere Message Broker provides message flow nodes that support connectivity with the SAP, Siebel, JDEdwards and PeopleSoft adapters.

Enterprise Metadata Discovery (EMD) is used for key data structure discovery and accelerated message set generation. The WebSphere Adapters nodes need an adapter component to access the EIS. A license for the adapter components is required.

The WebSphere Adapters nodes can be used to interact with the Enterprise Information System. The SAP, Siebel, JDEdwards, and PeopleSoft adapters are supported by the following message flow nodes in WebSphere Message Broker:

- ▶ SAPInput node
- ▶ SAPReply node
- ▶ SAPRequest node
- ▶ SiebelInput node
- ▶ SiebelRequest node
- ▶ PeopleSoftInput node
- ▶ PeopleSoftRequest node
- ▶ JDEdwardsInput node
- ▶ JDEdwardsRequest node

The WebSphere Adapters' input nodes monitor an EIS for a particular event. When that event occurs, business objects are sent to the input node. The input nodes construct a tree representation of the business object, which the rest of the message flow can use.

The WebSphere Adapters' request nodes can send and receive business data. They request information from an EIS and propagate the data to the rest of the message flow.

For additional information about configuring and using each of the WebSphere Adapters nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ SAPInput node (ac37290_)
- ▶ SAPReply node (bc22000_)
- ▶ SAPRequest node (ac37300_)
- ▶ SiebelInput node (ac37310_)
- ▶ SiebelRequest node (ac37320_)
- ▶ PeopleSoftInput node (ac37330_)
- ▶ PeopleSoftRequest node (ac37340_)
- ▶ JDEdwardsInput node (bc22690_)
- ▶ JDEdwardsRequest node (bc22700_)

2.3.8 TCP/IP

WebSphere Message Broker provides support for TCP/IP connectivity of existing applications that use raw TCP/IP sockets for transferring data.

TCP/IP Client nodes

The TCP/IPClientInput node can be used to create a client connection to a raw TCP/IP socket and to retrieve data over that connection.

The TCPIPClientOutput node can be used to create a client connection to a raw TCP/IP socket and to send data over that connection to an external application.

The TCPIPClientReceive node can be used to receive data over a client TCP/IP connection.

TCP/IP Server nodes

The TCPIPServerInput node can be used to create a server connection to a raw TCP/IP socket and to retrieve data over that connection.

The TCPIPServerOutput node can be used to create a server connection to a raw TCP/IP socket and to send data over that connection to an external application.

The TCPIPServerReceive node can be used to receive data over a server TCP/IP connection.

For additional information about configuring and using each of the TCP/IP nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ TCPIPClientInput node (ac67300_)
- ▶ TCPIPClientOutput node (ac67310_)
- ▶ TCPIPClientReceive node (ac67320_)
- ▶ TCPIPServerInput node (ac67330_)
- ▶ TCPIPServerOutput node (ac67340_)
- ▶ TCPIPServerReceive node (ac67350_)

2.3.9 Service Component Architecture

Service Component Architecture (SCA) simplifies creating and integrating SOA business applications by separating the business logic from its implementation, thus using the developer to focus on assembling an integrated application without knowing the details of its implementation.

WebSphere Message Broker provides built-in message flow nodes that support interoperability with WebSphere Process Server. The SCA nodes support inbound and outbound scenarios with WebSphere Process Server and WebSphere Enterprise Service Bus.

The nodes are configured by a Broker SCA definition. The Broker SCA Definition wizard creates a Broker SCA definition that includes objects that define the binding, interface, and message format information to permit interoperation between WebSphere Message Broker and WebSphere Process Server.

SCAInput node

The SCAInput node listens for SCA inbound requests from WebSphere Process Server to WebSphere Message Broker.

SCAReply node

The SCAReply node is used to send a message from the broker to the originating client in response to a message that a SCAInput node receives.

SCARequest node

The SCARequest node is used to send a request to WebSphere Process Server. The node is configured using a Broker SCA Definition (.outsca) file. Depending on the contents of the.outsca file, requests can be either of the following types:

- ▶ Two-way, synchronous: The node sends the request and then blocks until it receives a response or the time out period is exceeded.
- ▶ One-way: The node sends a request only.

SCAAsyncRequest node

The SCAAsyncRequest node is used to send an SCA outbound request to a service component that runs in WebSphere Process Server.

SCAAsyncResponse node

The SCAAsyncResponse node receives the response from a business process that is running in WebSphere Process Server and to which the previous asynchronous request was made. The SCAAsyncResponse node can be in the same message flow or in a separate message flow.

For additional information about configuring and using each of the SCA nodes, go to the WebSphere Message Broker Information Center at:

<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>

In the WebSphere Message Broker Information Center, refer to these topics:

- ▶ SCAInput node (ac68510_)
- ▶ SCAReply node (ac68520_)
- ▶ SCARequest (bc34002_)
- ▶ SCAAsyncRequest (ac68530_)
- ▶ SCAAsyncResponse (ac68540_)

2.4 Transformation interfaces

The most significant function of Message Broker is its wide range of transformation interfaces that open it to act as a robust data transformation engine. Data in messages received by input nodes are parsed into a *logical message tree* that works with code. Together the tree and code can alter the values and location of individual fields without having to know anything about the physical representation of the data in the message. This powerful concept shifts an application rapidly from receiving data through an MQInput node to receive data over HTTP using an HTTPInput node without any significant changes to business logic.

2.4.1 Compute node

The Compute node allows for the creation, editing, and execution of transformation logic written in the ESQL language. ESQL is a simplified language that supports the full range of broker function and is easy to write and easy to read.

Message data is accessed using a dot-notation path structure that is built up of the names of the fields in the message tree. Individual parsers provide named constants to indicate data that has a special meaning with the parser structure, for example, namespace declarations in XML data.

ESQL code is stored in text files with the file extension of *.esql*. The Message Broker Toolkit provides a full featured code editor with content assist, error identification, and flagging along with full step-through debugger support. An ESQL stub file for holding message flow logic can be simply created by double-clicking a **Compute node** that was added to a message flow. The same ESQL file can contain logic associated with more than one Compute node, and ESQL files can load and reference other ESQL files as code libraries. ESQL also supports a schema syntax that allows for separation of function into logical namespaces. This supports the same modular development of code and methods as other full-featured programming languages, such as Java, PHP, and .NET CLR languages. ESQL can natively invoke functions that are coded in Java, .NET, and in database stored procedures.

ESQL is compiled into an internal representation and deployed to the Broker as part of an application or library. ESQL files can also be deployed individually to an application or library, including as stand-alone files in the default application.

2.4.2 JavaCompute node

The JavaCompute node allows for the creation, editing, and execution of transformation logic that is written in the Java language. Java is a full functioning programming language that supports an object-oriented development model and is widely used for applications of all kinds across all levels and areas of business.

WebSphere Message Broker provides a full featured set of Java classes that present an object-oriented view onto the logical message tree giving a Java programmer the ability to develop WebSphere Message Broker solutions with a minimum of additional training. The Java API and class library for the broker supports full control over the logical message tree. The JavaCompute node can be used for integrating with applications and other endpoints that are not using transports available with the built in nodes. The JavaCompute node also can reuse existing Java code that performs business logic.

The Message Broker Toolkit provides a full-featured Java development environment based on the Eclipse tooling. This environment includes content assist, debugging aids, error marking, and the creation of stub classes and methods. The same Java classes can be used by multiple JavaCompute nodes, and Java objects can be shared across Java code running in the same execution group.

Java code is compiled into jar files and deployed as part of an application or library through the normal broker deployment tooling. The Java code can be manually deployed to the broker's file system and made available using a JavaClassLoader Configurable Service. Java code runs inside a Java Virtual Machine that is part of each individual execution group process.

2.4.3 PHPCompute node

The PHPCompute node opens the creation, editing, and execution of transformation logic written in the PHP language. PHP is a full-functioning programming language that supports an object-oriented development model. PHP also allows a script-like programming model that does not require an object-oriented approach. PHP is widely used for web-based applications and in development environments where simplified programming models are important.

WebSphere Message Broker provides a full-featured PHP API and set of classes that present an object-oriented view onto the logical message tree. The PHP API in the broker provides a convenient path navigation mechanism that is natural to the way that data is accessed by PHP code. The PHPCompute node can be used to integrate with applications and endpoints using transports that are not fully supported by WebSphere Message Broker built in nodes,

and existing PHP code can be reused. The PHPCompute node can call the evaluate method of a PHP class or it can invoke a stand-alone PHP script file that returns a modified message tree.

The Message Broker Toolkit does not provide immediate support for PHP code editing; however, Eclipse-based tooling can be installed into the Message Broker Toolkit environment. Message Broker Toolkit also does not support step-through debugging of PHP code.

PHP code is compiled into Java bytecodes and deployed into an application or library. It is executed in the execution group JVM.

2.4.4 .NETCompute node

The .NETCompute node allows for creating, editing, and executing of transformation logic in any of the programming languages supported by the Microsoft Common Language Runtime at version 4.0. This includes Visual Basic, C#, JScript, F#, and more than 40 other programming languages.

The .NETCompute node can only be deployed to a broker that is running on a Windows platform.

The .NETCompute node allows for simple and fast integration with Visual Studio, which is the main development environment for .NET code outside of WebSphere Message Broker. The broker leverages this power and ease of use to provide a common development environment. Code is created, edited, compiled, and debugged using Visual Studio. The .NETCompute node holds a reference to the location of compiled code and the broker runtime loads the assemblies and executes the code.

WebSphere Message Broker provides a full featured .NET API and set of classes that represents an object-oriented view on the logical message tree. This is similar to the Java and PHP APIs, however the .NET library provides several additional helper classes and some additional simplified ways to access the logical message tree. The .NET API can use the Microsoft LINQ APIs to query the tree and interact with the results, and provide a comprehensive path navigation mechanism that is natural to the way data and objects are accessed in .NET Code. Because the .NET API is built against the Common Language Runtime, full support for these features exists in all programming languages that work with the CLR V4.

WebSphere Message Broker includes project templates that are installed into Visual Studio, making it easy to write .NET code to perform logic inside the broker. The Visual Studio Debugger also allows for the step-through of .NET code that is deployed inside the broker. This supports comprehensive debugging in a development environment.

Scenarios in this IBM Redbooks Publication focus on several important aspects of writing .NET code that is used by the .NETCompute node to access external systems and transform or interpret message data.

2.5 Administering WebSphere Message Broker

In this section, we provide an introduction to administering WebSphere Message Broker. We cover WebSphere Message Broker Explorer, broker sets, and aspects of web administration console and the functions of record and replay.

2.5.1 WebSphere Message Broker Explorer

The WebSphere Message Broker Explorer is a tool for administrators and provides the capability for enhanced WebSphere Message Broker monitoring and management.

WebSphere Message Broker Explorer is installed as a plug-in for WebSphere MQ Explorer. The *Brokers* view is added to the WebSphere MQ Explorer Navigator pane. The broker administration tasks can then be performed from this Brokers view.

Using the WebSphere Message Broker Explorer, as shown in Figure 2-12, the user can administer their brokers and WebSphere MQ queue managers, in the same tool.

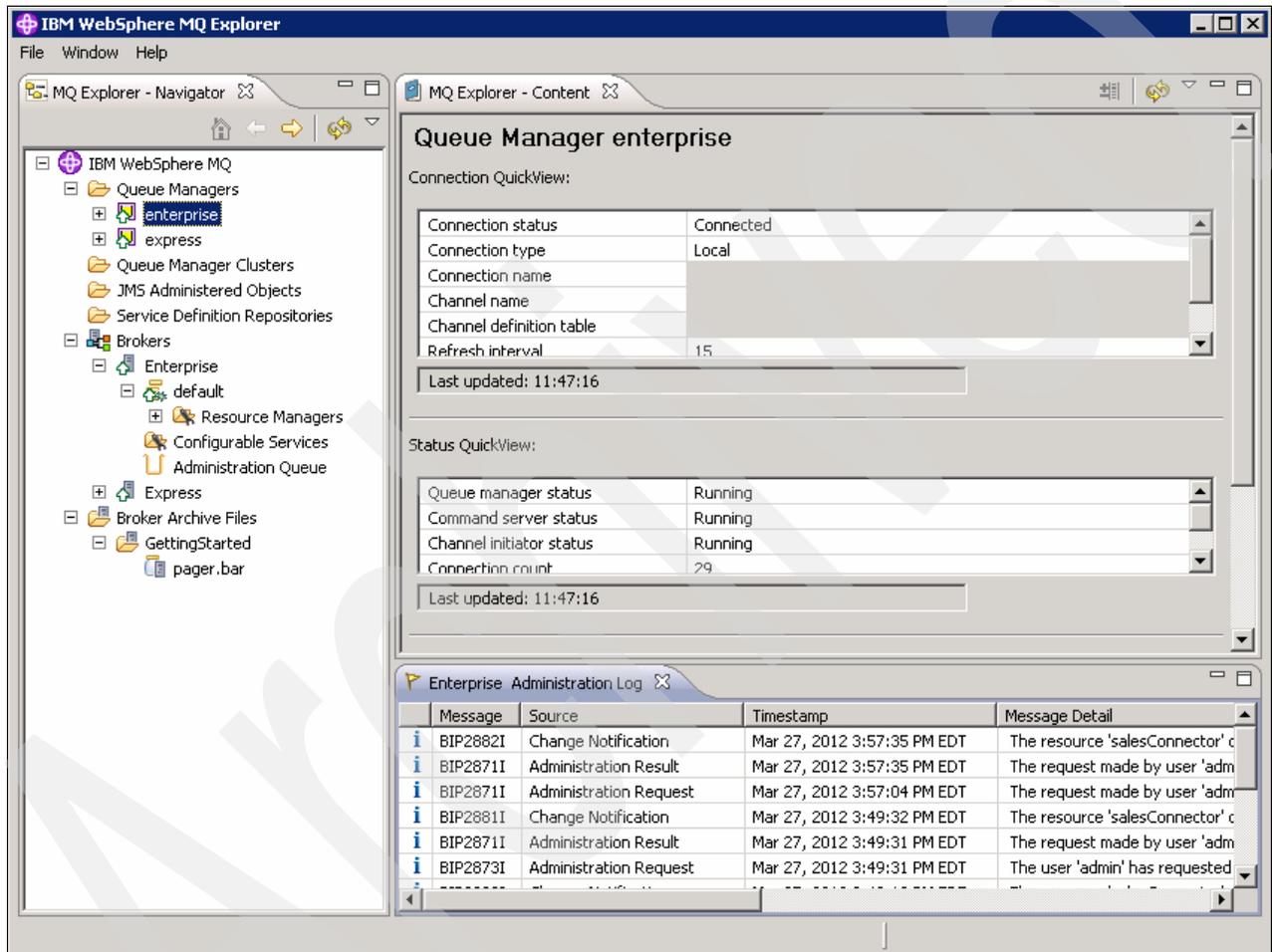


Figure 2-12 WebSphere Message Broker Explorer

Additionally, the new WebSphere Message Broker Explorer provides the following new capabilities:

- ▶ Create, delete, start, and stop local brokers without using the command line.
- ▶ Show the relationships between the brokers and queue managers.
- ▶ Deploy a broker archive file to multiple execution groups in one step.
- ▶ Visualize a brokers accounting and statistics data.
- ▶ Configure broker properties, including creating and modifying services for broker communication with external services, such as SMTP, JMS providers, and adapters.

- ▶ View the broker administration queue and remove pending tasks that were submitted to the broker.
- ▶ Connect and configure DataPower security settings.
- ▶ View the Administration Log showing all activity that occurred on a given broker.

The WebSphere Message Broker Explorer also provides a number of QuickViews that you can use to view the properties of brokers and their resources. These views are automatically displayed in the Brokers folder in the WebSphere MQ Explorer Navigator view. There is also a QuickView for viewing the details of broker archive files that are imported into the WebSphere Message Broker Explorer.

2.5.2 Broker sets

You can use a broker set to visually group your brokers in the WebSphere Message Broker Explorer. If you have a large number of brokers displayed, it might be helpful to group the brokers using broker sets. Use broker sets to separate the development, test, QA, and production brokers into logical groups. You can create manual broker sets, automatic broker sets, or a combination of both.

A manual broker set is empty until you add brokers to the broker set from the *All* broker set or another broker set. You can add or remove brokers from the manual broker set at any time.

An automatic broker set uses broker tags to dynamically add and remove brokers from a set that is based on values that you provide or the current state of brokers.

2.5.3 Web Administration and record and replay

WebSphere Message Broker V8 also comes with a web-based administrative console that provides a read-only view into the broker runtime environment. The console can also be used as an end-user console for the record and replay functions for messages that pass through the Message Broker. The record and replay functions allow a broker to store copies of messages that are passed through the system into a database and then the web administration console can be used to retrieve and resend the needed messages.

This function must be specifically enabled by the broker administrators to work. Figure 2-13 on page 32 shows the web administration console.

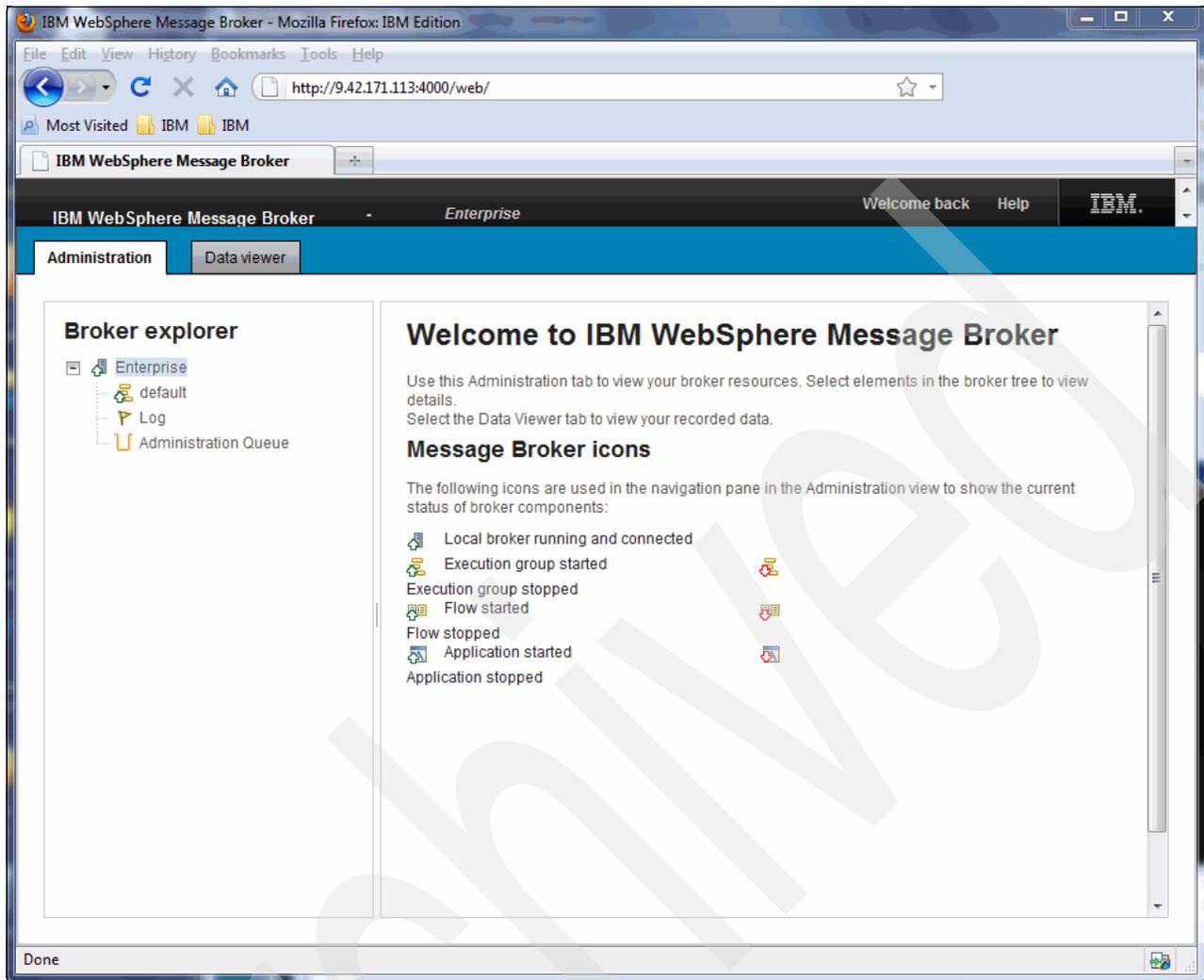


Figure 2-13 WebSphere Message Broker web administration console

The web administration console also exposes a full REST API for accessing resources and performing record or replay features. These functions can then be easily integrated into existing or custom applications.

In addition to the WebSphere Message Broker Explorer and the read-only web administration console, there are command-line and API-based utilities that you can use to accomplish broker administration tasks. For example, the CMP API Exerciser sample that WebSphere Message Broker provides utilizes the Java Administration API. You can use the CMP API Exerciser to view and manage brokers and their execution groups. You can also use it to create, modify, and delete services that can be configured and general broker administration tasks.

The Java Administration API is also available to users for scripting broker administration tasks.

2.6 Deploying applications

WebSphere Message Broker applications and libraries contain message flows and the message models that are used to describe the data used by message flows. Applications and libraries are independently deployable containers. An application can contain other libraries. WebSphere Message Broker resources can be deployed to the execution groups of the brokers by first adding the application or library to a *broker archive file* (bar file) and then deploying the bar file to the broker's execution group. You can deploy the bar files using the command line or with the Message Broker Toolkit.

You can create a bar file using a graphical interface in the Message Broker Toolkit. The interface allows you to select the components to include for deployment. If properties, such as queue names or other configurable properties, are promoted to the bar level, the broker archive editor offers options to configure these properties.

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

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/ac00640_.htm

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

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an09020_.htm

You can also deploy a bar file using WebSphere Message Broker Explorer. Drag and drop the bar file from the Broker Archive Files folder to the execution group. The WebSphere Message Broker Explorer also supports customizing bar files using overrides and promoted properties.

You can also develop an application written in Java that uses the Administration API to deploy bar files.

Deploying resources without creating a BAR file: The Message Broker Toolkit allows you to deploy a message flow, application, or library directly by dragging it to an execution group (without creating a BAR file). In this case, the dependent resources are deployed too.

2.7 Getting started

As you read through the scenarios, you will encounter instructions for performing tasks in the Message Broker Toolkit. These instructions center on creating and deploying message flows. This section provides the information needed to perform these tasks.

2.7.1 Opening the Message Broker Toolkit

The Message Broker Toolkit is used to develop and deploy message flows and to manage brokers. It is supported on Windows and Linux on x86. The scenarios in this book all use Message Broker Toolkit installed on a Windows system:

1. Open the Message Broker Toolkit. In a default installation, this can be done by selecting **Start** → **All Programs** → **IBM WebSphere Message Broker Toolkit** → **IBM WebSphere Message Broker Toolkit 8.0** → **WebSphere Message Broker Toolkit 8.0**.
2. In the Workspace Launcher dialog, enter a location for your workspace in the Workspace field, and press **OK**, as shown in Figure 2-14.

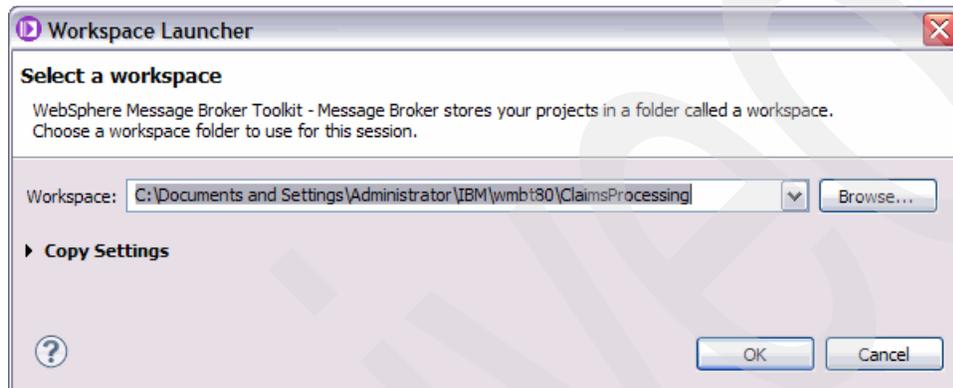


Figure 2-14 Creating a new workspace

3. The IBM WebSphere Message Broker Welcome window will then display. Click **Go to the Message Broker Toolkit**, as shown in Figure 2-15.



Figure 2-15 Go to Message Broker Toolkit

You are now in the Broker Application Development perspective with an empty workspace ready for new projects to be added and created (shown in Figure 2-16 on page 35).

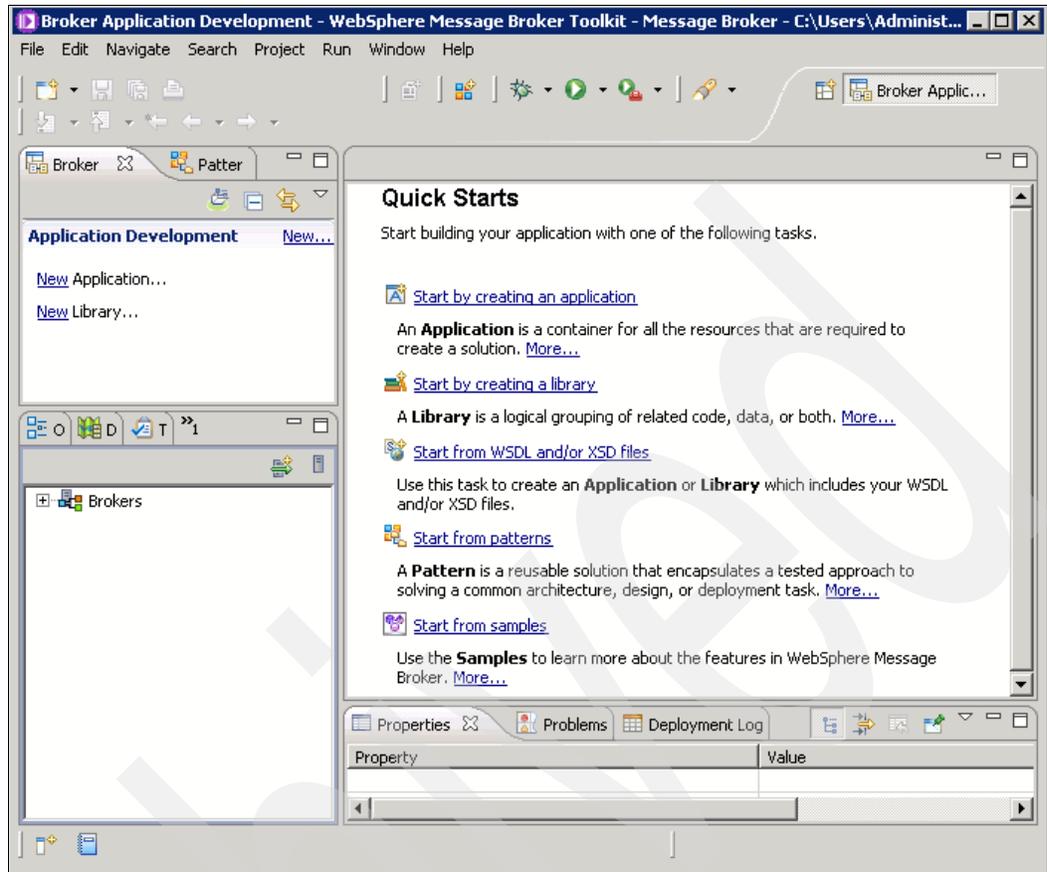


Figure 2-16 Empty workspace in the Broker Application Development perspective

2.7.2 Creating a new application

Use the following procedure to quickly outline how to create a new application. These easy steps create efficiency, accuracy, and help standardize the process:

1. There are several ways to start the New Application wizard:
 - Select **New Application**, Figure 2-16 (see in the Broker Development view if this option is available).
 - Select **File** → **New** → **Application**.
 - Right-click in an empty area of the Broker Development view and select **New** → **Application**.

Any of these actions will start the New Application wizard.
2. Name the application, and click **Finish** (see Figure 2-17 on page 36).

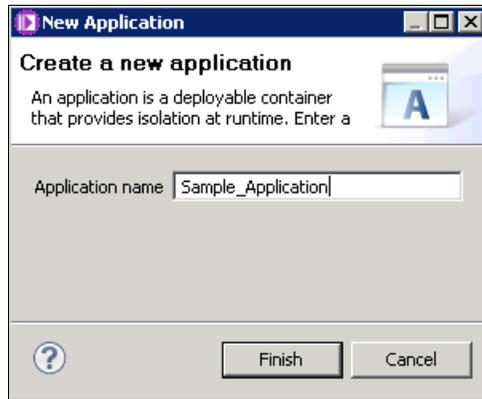


Figure 2-17 New Application wizard

The new application will appear in the Broker Development view (see Figure 2-18).

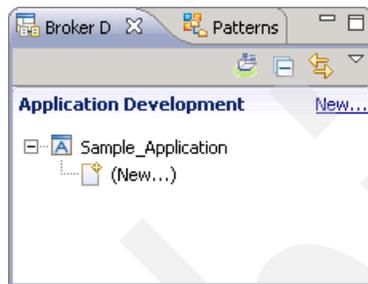


Figure 2-18 A new application

2.7.3 Creating a new message flow

Message flows contain the logic of the application. To create a new message flow in the application follow these steps:

1. The following list shows several ways to start the New Message Flow wizard:
 - Click **New**, under the application name, as shown in Figure 2-18, and select **Message Flow**.
 - Select **File** → **New** → **Message Flow**.
 - Right click in an empty area of the Broker Development view and select **New** → **Message Flow**.
2. In the Container field, enter the application name. Enter a name for the new message flow and click **Finish**, as shown in Figure 2-19 on page 37.

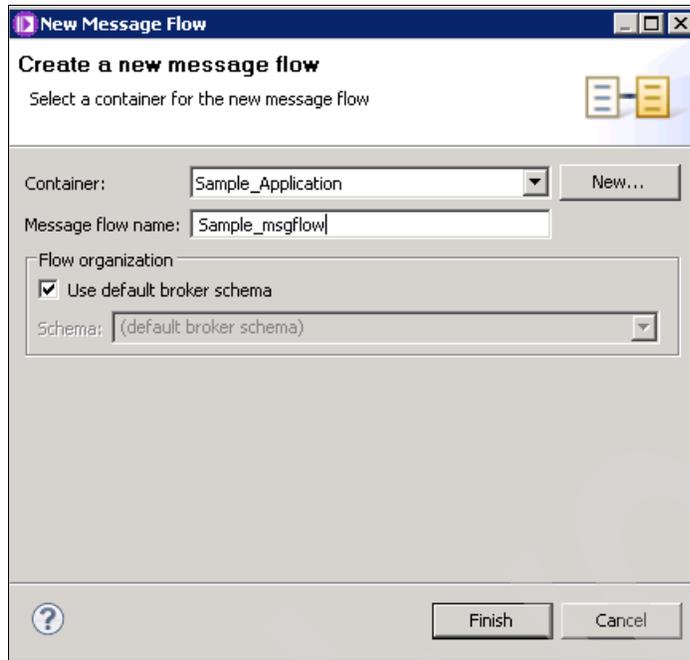


Figure 2-19 Create a new message flow

3. The new message flow is created and opened in the Message Flow editor, as shown in Figure 2-20.

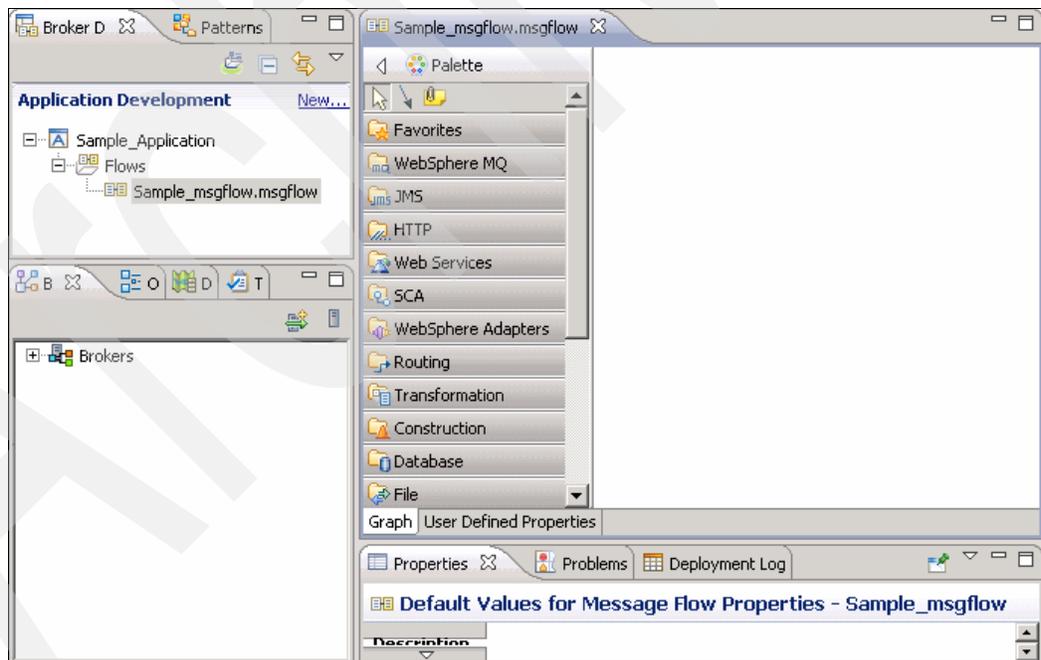


Figure 2-20 New message flow opened in the Message Flow editor

4. A message flow is built by dragging nodes from the Palette to the canvas, setting the properties for the nodes, and wiring them together to form a flow for messages to traverse. Each drawer in the Palette can be opened by clicking it, to show the nodes in that drawer.

In Figure 2-21, the HTTPInput node and .NETCompute node were dragged from the Palette located to the left over to the canvas located to the right. The HTTPInput node is selected in the canvas, and its properties are displayed. The Properties view has multiple tabs to the left that allow you to update the properties by category, for example, input message parsing options.

The *out* terminal of the HTTPInput node was wired to the *in* terminal of the .NETCompute node. A wire is created by clicking the terminal of one node, and then dragging the connection to the input node of another terminal.

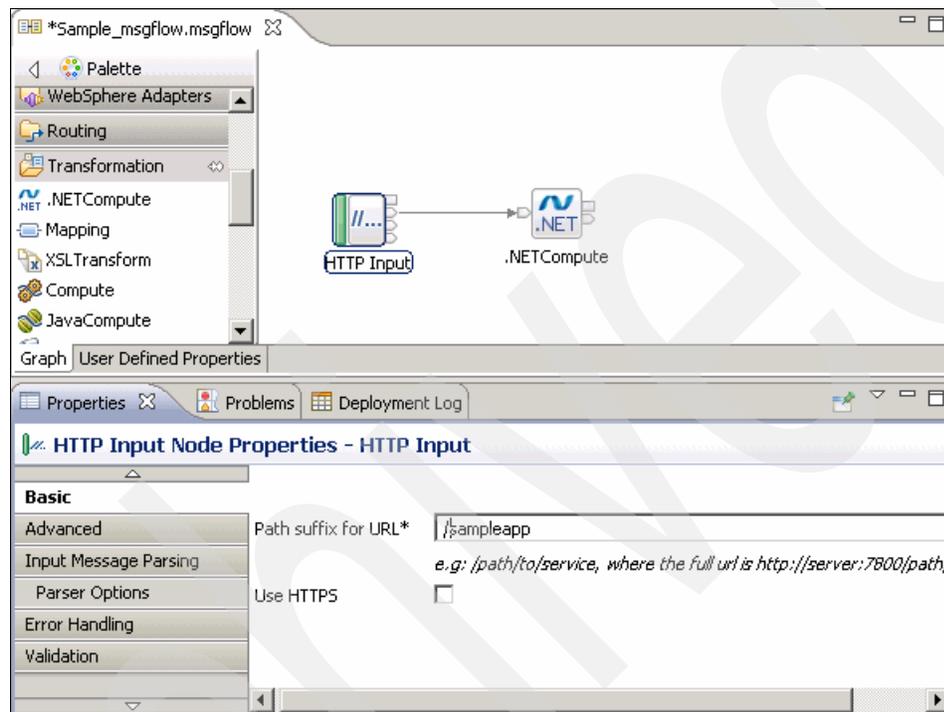


Figure 2-21 The HTTPInput node selected within the canvas and properties opened

5. To save the message flow, click **CTRL-S**, or select **File → Save**.

2.7.4 Creating a broker

A broker can be created from the Message Broker Toolkit, the WebSphere Message Broker Explorer or when using the WebSphere Message Broker `mqsicreatebroker` command.

Using the Message Broker Toolkit

Brokers can be created, deleted, and managed from the Message Broker Toolkit. To create a broker:

1. Open the Message Broker Toolkit.
2. In the Brokers view, right-click **Brokers**, and select **New → Local Broker**, as shown in Figure 2-22 on page 39.

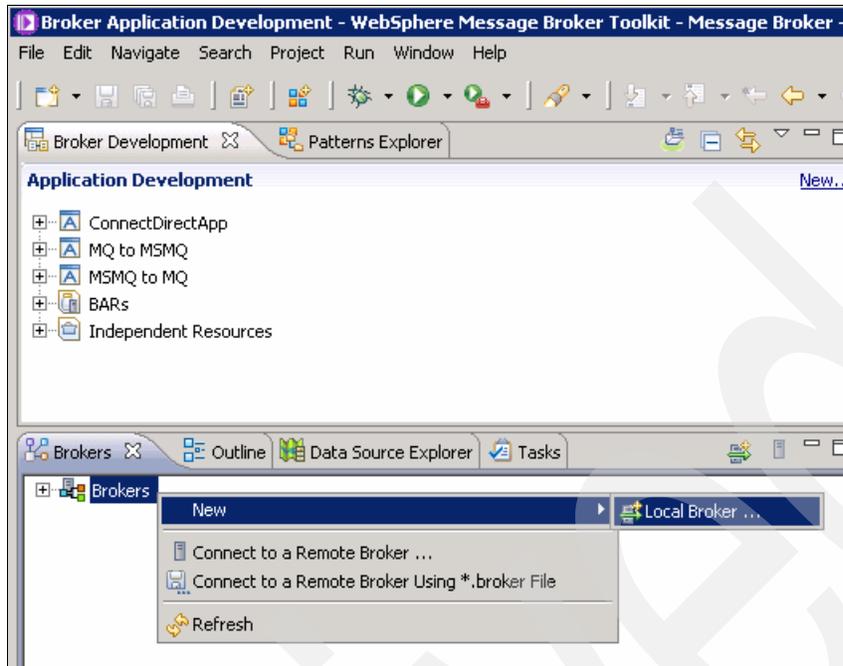


Figure 2-22 Creating a new local broker in the Brokers view

3. Enter a name for the broker. Enter an existing queue manager or a new name for the broker queue manager. Finally, enter the execution group name. Click **Finish**. This prompt is shown in Figure 2-23.

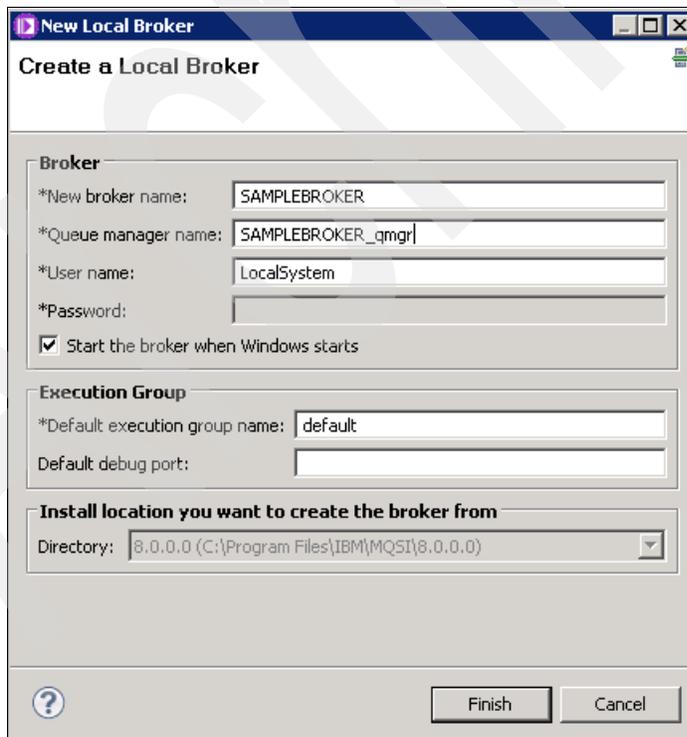


Figure 2-23 Entering names for broker, queue manager, and execution group

- The broker and queue manager are created. The results are shown in the Progress Information window. Click **Close**. See an example of the results window in Figure 2-24.

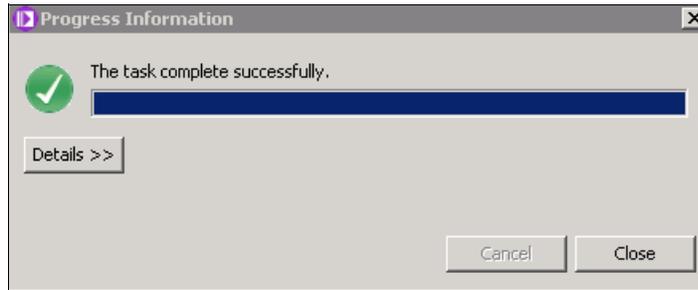


Figure 2-24 Update bar reflecting successful completion

The new broker and execution group will be started and will appear in the list of brokers.

Using the WebSphere Message Broker Explorer

Brokers can be created, deleted, and managed from the WebSphere Message Broker Explorer. Follow these steps to create a broker:

- Open the WebSphere Message Broker Explorer by right-clicking the **WebSphere MQ icon** in the Windows system tray and selecting **WebSphere Message Broker Explorer**. See Figure 2-25 for an example.

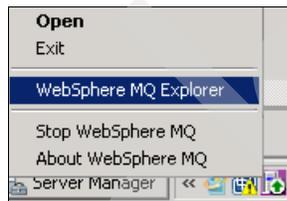


Figure 2-25 Open the WebSphere Message Broker Explorer

- Right-click **Brokers**, and select **New** → **Local Broker**, as shown in Figure 2-26.

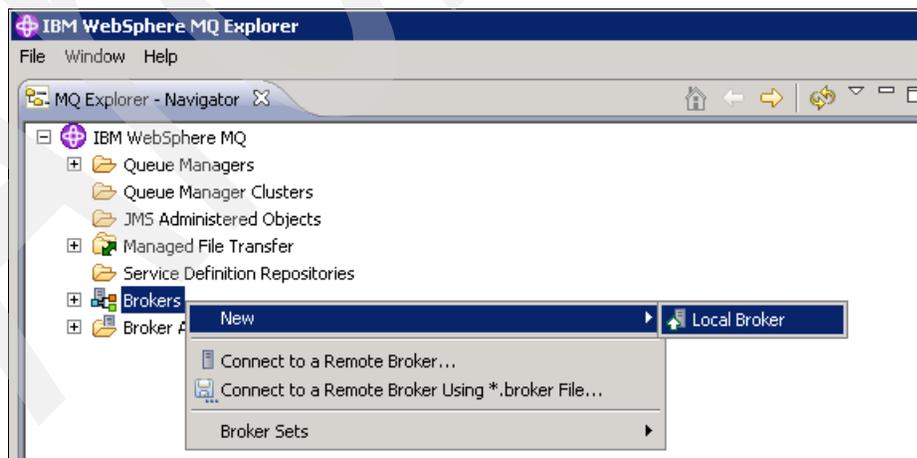


Figure 2-26 Creating a Local Broker

- Enter a name for the new broker, and click **Next**, as shown in Figure 2-27 on page 41.



Figure 2-27 Name the new broker

4. Enter the queue manager name for the broker. You can enter an existing name or a new name and the wizard will create the queue manager for you. You can also enter the name of the execution group or accept *default* as the name. Click **Next**. See Figure 2-28 on page 42 for an example.

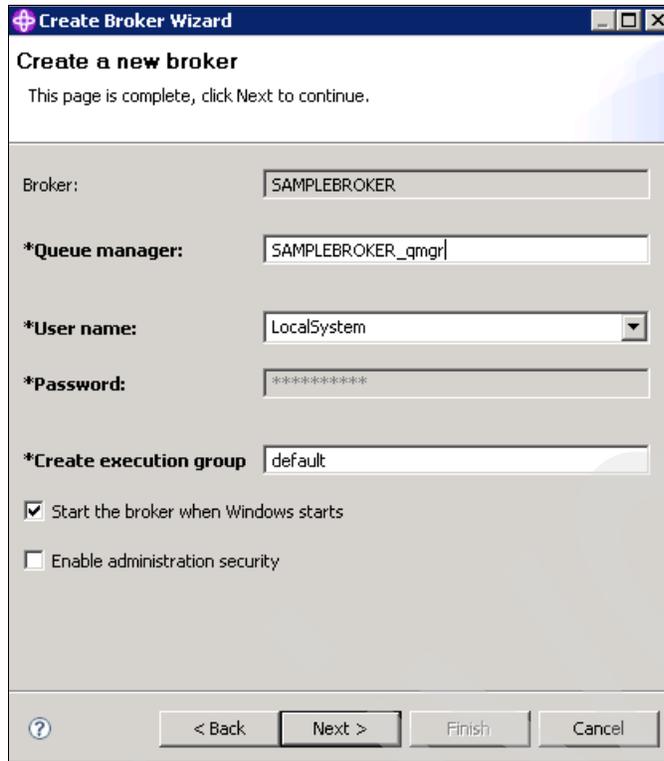


Figure 2-28 Prompt to enter new queue manager name

5. The new broker and queue manager is created and the results shown in the Summary window. Click **Finish**. Figure 2-29 shows an example of this window.



Figure 2-29 Results of the Create Broker wizard

The new broker and execution group will be started and will appear in the list of brokers.

Using a command

The broker can also be created using the `mqsicreatebroker` command. Use the following syntax with basic parameters for this command:

```
mqsicreatebroker brokername -i LocalSystem -q queuemanagername
```

For example, the following command creates a broker named SAMPLEBROKER and its queue manager SAMPLEBROKER_qmgr:

```
mqsicreatebroker SAMPLEBROKER -i LocalSystem -q SAMPLEBROKER_qmgr
```

WebSphere Message Broker commands are in the **Broker_Install/bin** directory.

A complete list of the options for this command are available at the following address:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an28125_.htm

2.7.5 Managing brokers

Brokers can also be managed with the Message Broker Toolkit, WebSphere Message Broker Explorer, and commands. The following examples show approaches for the Message Broker Toolkit and WebSphere Message Broker. A link is provided for managing brokers with commands.

WebSphere Message Broker Toolkit

The Brokers view in the Message Broker Toolkit lists the brokers, shows their status, and provides options for starting, stopping, and deleting existing brokers.

Right-click **Brokers** to access options to create a new broker or connect to a remote broker, as shown in Figure 2-30.

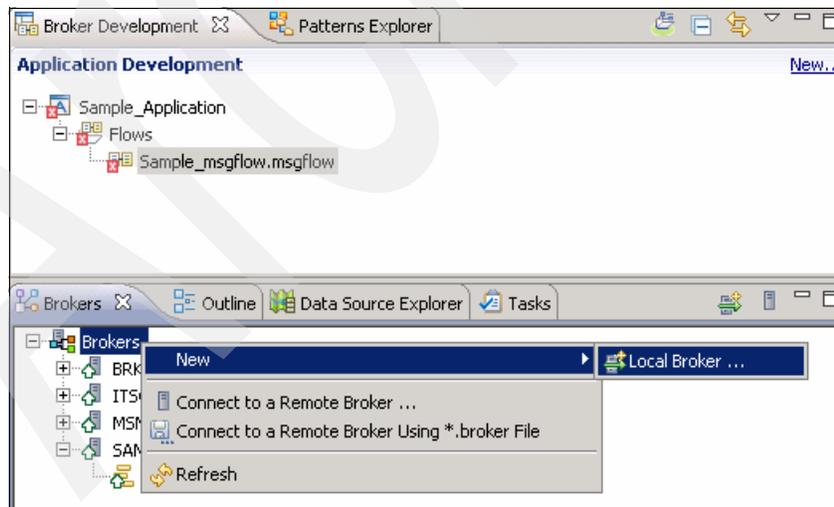


Figure 2-30 Options window for creating or connecting to a local broker

Right click a broker to start or stop the broker, delete the broker, or to create a new execution group, as shown in Figure 2-31 on page 44.

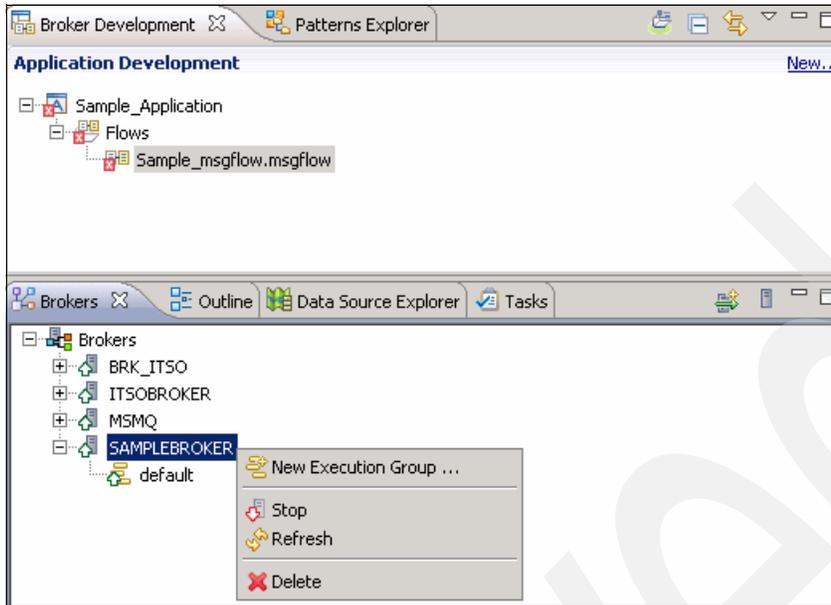


Figure 2-31 Options menu for the local broker or creating a new execution group

And finally, right-click an execution group to deploy artifacts to it or to stop, start, and delete it. The options are shown in Figure 2-32.

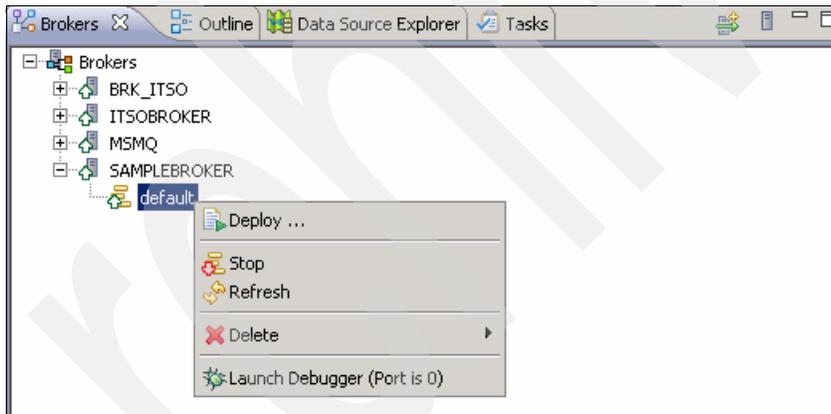


Figure 2-32 Options menu for the execution group

WebSphere Message Broker Explorer

WebSphere Message Broker Explorer provides similar functions for brokers and execution groups. Figure 2-33 on page 45 shows the options available for brokers.

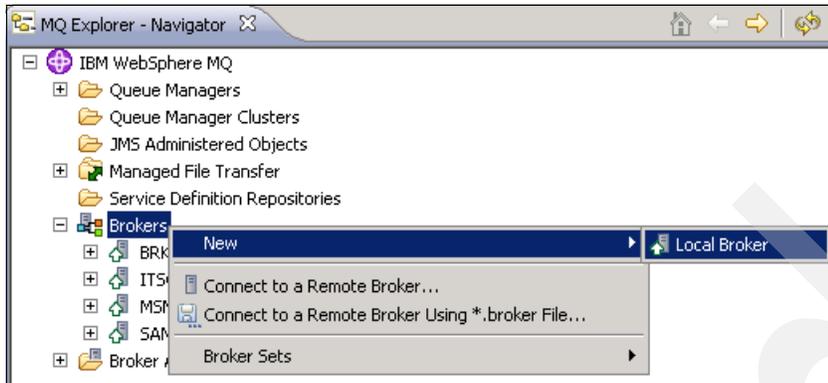


Figure 2-33 Options available for brokers in WebSphere Message Broker Explorer

Figure 2-34 shows the options for brokers that are available in WebSphere Message Broker Explorer.

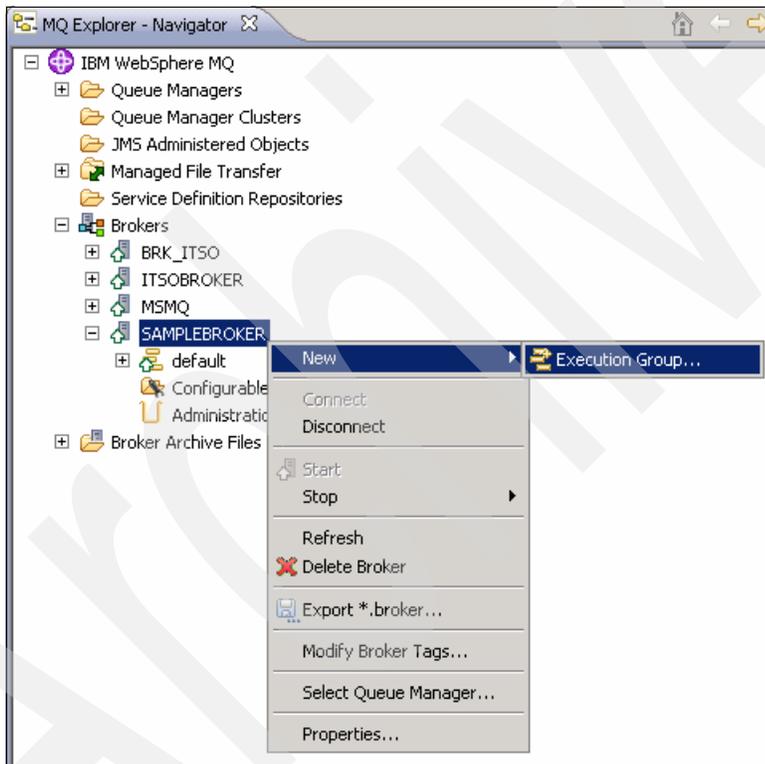


Figure 2-34 Options for brokers or to create a new execution group

Figure 2-35 on page 46 shows the options available for execution groups within WebSphere Message Broker Explorer.

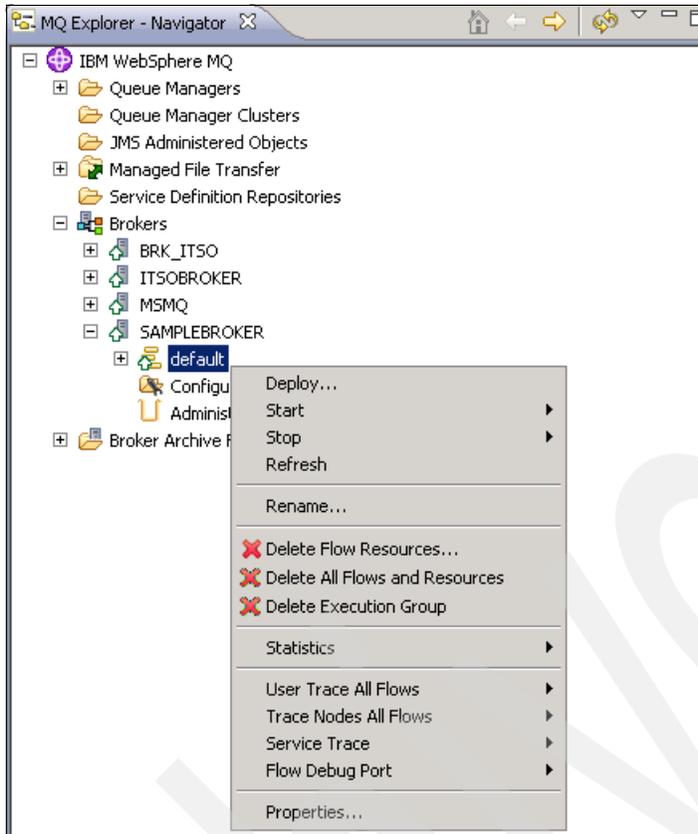


Figure 2-35 Execution group options

Commands

Brokers and execution groups can be managed using mqsi commands. For a complete list of commands, follow this link:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an07060_.htm

2.7.6 Creating a BAR file for deployment

WebSphere Message Broker applications are contained in broker archive (BAR) files for deployment. To create a BAR file for deployment of the application:

1. Right-click the **application** in the Broker Development view, and select **New** → **BAR File**. For an example, see Figure 2-36 on page 47.

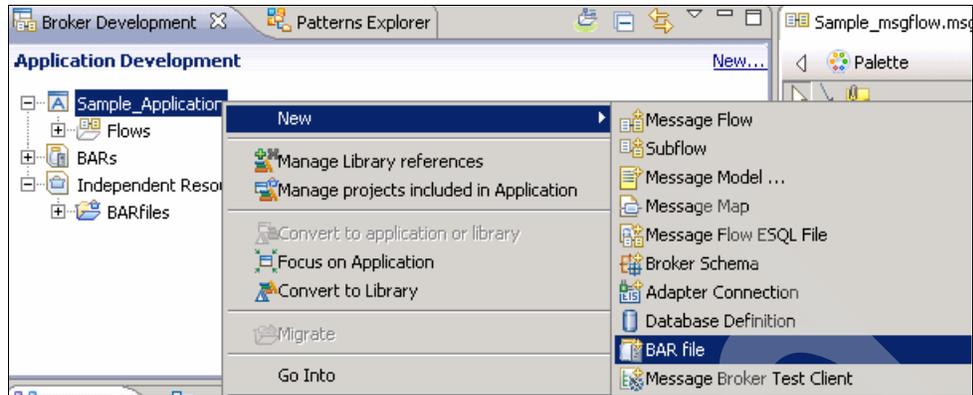


Figure 2-36 Creating a new BAR file

2. Enter a name for the BAR file, as shown in Figure 2-37, and click **Finish**.

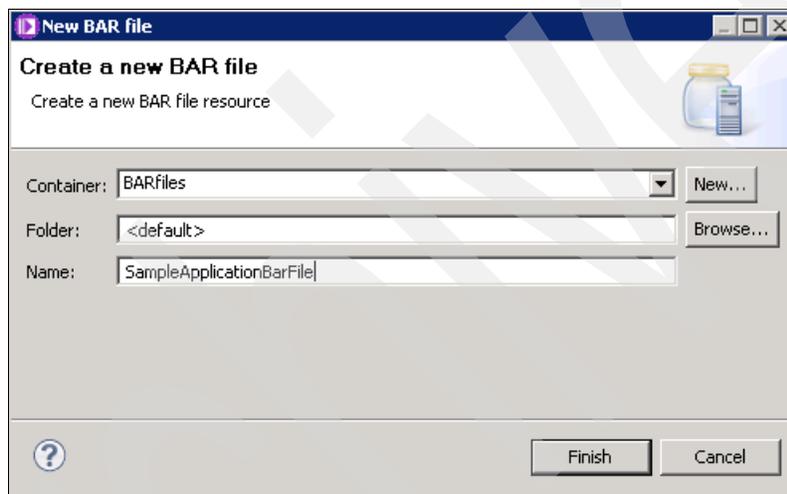


Figure 2-37 Naming the new BAR file

3. The new BAR file opens in the Broker Archive editor with multiple tabs for configuration. On the Prepare tab, select the **application** then click the **Build and Save** button. This procedure is shown in Figure 2-38 on page 48.

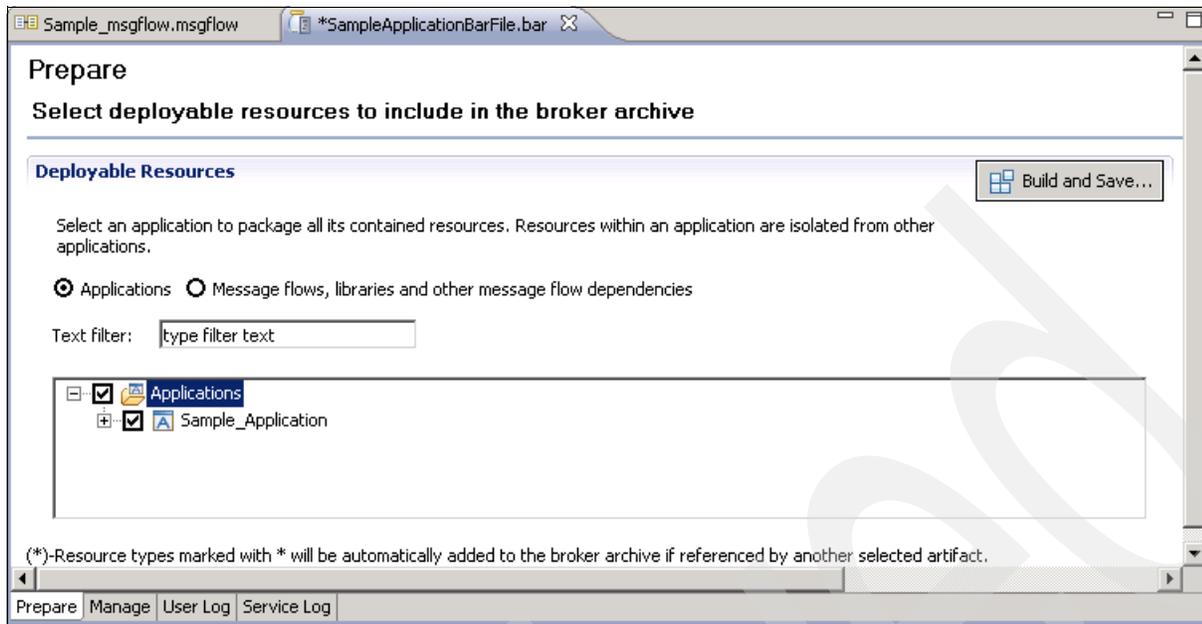


Figure 2-38 Configuring the new BAR file

Configurable properties: If there are configurable properties for the message flow, select the Manage tab, expand the application in the view, and select the **message flow**. The Properties view can then be used to configure the properties.

Parallel processing for input nodes: The broker always creates a separate thread for each input node in a message flow. For example, a message flow with three input nodes will be processed by three threads by default. You can specify additional threads for the whole flow using the Additional Instances configuration parameter of the BAR file.

To see this parameter, select the Manage tab, and select the compiled message (*message_flow.cmf* file). The setting will be in the Properties view on the Configure tab.

These additional thread instances are distributed between input nodes on a first-come first-served basis. If you need more even distribution for your input nodes, you can use a separate thread pool for each of them. Select the pool from the Instances tab on the Properties view for each input node during design time.

4. Close and save the BAR file.

Deleting BAR files: You will notice in the Message Broker Toolkit that right-clicking a BAR file does not give you the option to delete it. Instead, highlight the BAR file, and press the delete key.

2.7.7 Deploying the BAR file to a broker

BAR files are deployed to execution groups by dragging and dropping from within the Message Broker Toolkit or the WebSphere Message Broker Explorer.

Figure 2-39 shows the application deployed for the default execution group in the broker. The broker is named, SAMPLEBROKER.

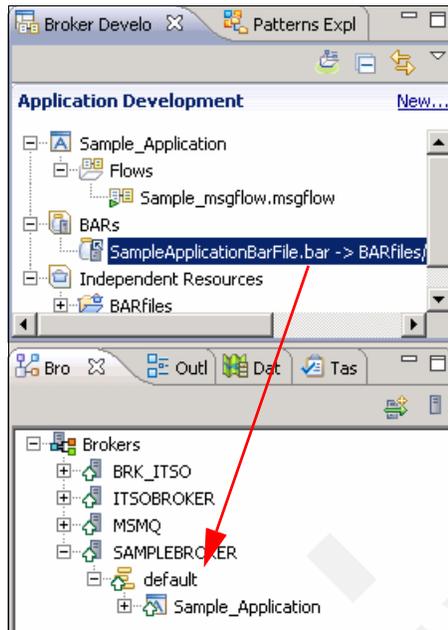


Figure 2-39 The application is deployed within the broker (the red arrow represents a drag and drop)

2.7.8 Executing WebSphere Message Broker commands

WebSphere Message Broker includes a set of commands that can be used to quickly operate or configure a broker. The commands are located in the *Broker_Install/bin* directory. To configure a broker execute these commands:

1. Open a command prompt.
2. Change the directory to the *Broker_Install/bin* directory.
3. Execute the `mqsiprofile` command to prepare the user environment for executing commands.

Example 2-1 shows an example of the commands to prepare the environment.

Example 2-1 Executing WebSphere Message Broker commands

```
c:\Program Files\IBM\MQSI\8.0.0.0\bin
C:\Program Files\IBM\MQSI\8.0.0.0\bin>mqsiprofile

MQSI 8.0.0.0
C:\Program Files\IBM\MQSI\8.0.0.0
```

A list of commands and links to help can be found at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp?topic=%2Fcom.ibm.etools.mft.doc%2Fan07060_.htm

2.8 Creating WebSphere MQ queue managers and queues

The WebSphere Message Broker infrastructure depends on WebSphere MQ. Brokers have a queue manager and queues to support them. These queue managers and queues are created when you create a broker.

Message flows also can contain WebSphere MQ nodes that require a queue manager and queue to be defined. This section provides you two methods for creating queue managers and queues for use by WebSphere Message Broker.

2.8.1 Creating queue managers

There are two simple ways to create a queue manager. You can create the queue manager with a WebSphere MQ command or from the WebSphere MQ Explorer.

2.8.2 Creating queues and queue managers using commands

The `crtmqm` command creates a queue manager, for example, to create a queue manager named `qmgr1`, use the following command:

```
c:\>crtmqm qmgr1
```

The `strmqm` command can be used to start the queue manager.

```
c:\>strmqm qmgr1
```

Example 2-2 shows the syntax for creating and starting a queue manager.

Example 2-2 Creating and starting a queue manager

```
C:\Users\Administrator>crtmqm qmgr1
WebSphere MQ queue manager created.
Directory 'C:\Program Files (x86)\IBM\WebSphere MQ\qmgrs\qmgr1' created.
Creating or replacing default objects for qmgr1.
Default objects statistics : 68 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.

C:\Users\Administrator>strmqm qmgr1
WebSphere MQ queue manager 'qmgr1' starting.
5 log records accessed on queue manager 'qmgr1' during the log replay phase.
Log replay for queue manager 'qmgr1' complete.
Transaction manager state recovered for queue manager 'qmgr1'.
WebSphere MQ queue manager 'qmgr1' started.
```

After you have an active queue manager, you can use the `RUNMQSC` command to issue commands interactively for the specified queue manager:

1. You will first issue the `runmqsc` command against the queue manager:

```
c:\>runmqsc Queue Manager Name
```

2. Enter the commands to execute against the queue manager. The `DEFINE QLOCAL` command can be used to create a local queue:

```
DEFINE QLOCAL(queue name)
```

3. When you are done, end the session:

END

For an example, review the setup shown in Example 2-3.

Example 2-3 Using the RUNMQSC command to issue commands to the queue manager

```
C:\Users\Administrator>runmqsc qmgr1
5724-H72 (C) Copyright IBM Corp. 1994. ALL RIGHTS RESERVED.
Starting MQSC for queue manager qmgr1.
```

```
DEFINE QLOCAL(queue1)
  2 : DEFINE QLOCAL(queue1)
AMQ8006: WebSphere MQ queue created.
END
  3 : END
2 MQSC commands read.
One command has a syntax error.
All valid MQSC commands were processed.
```

```
C:\Users\Administrator>
```

For more information about the DEFINE QLOCAL command visit the following link:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/sc11210_.htm

2.8.3 Creating queue managers and queues from the WebSphere MQ Explorer

Queue managers can be created in WebSphere MQ Explorer by selecting **Queue Managers**. Queue managers can be created in the Navigator view of the WebSphere MQ Explorer by right-clicking and selecting **New** → **Queue Manager**. After that selection, you will enter the name of the queue manager, and click **Finish**.

Queues can be created in the WebSphere MQ Explorer by selecting the **queue manager** in the WebSphere MQ Explorer, Navigator view, and right-clicking the **Queues** folder. Select **New** → **Queue**, as shown in Figure 2-40. Enter the queue name, and click **Finish**.

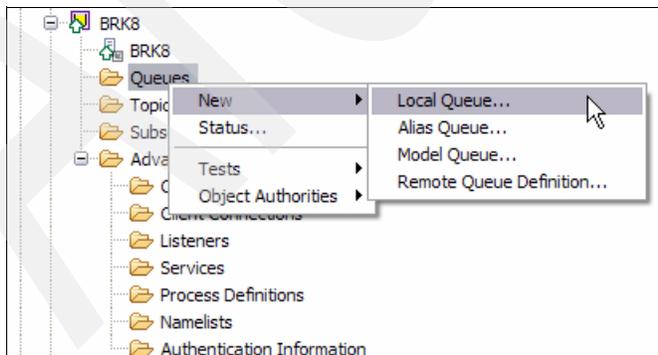


Figure 2-40 Creating a New Queue

Archived



Using the tools for WebSphere Message Broker and .NET Integration

The Microsoft Visual Studio 2010 development environment is a best of breed solution for developing .NET platform code. To fully use the capability of this powerful IDE development of .NETCompute nodes, this IBM Redbooks publication features the tight integration between the WebSphere Message Broker Toolkit and Visual Studio. The Message Broker Toolkit is where you develop the flow structure and properties, and Visual Studio is where you develop .NET code to run inside the WebSphere Message Broker runtime.

This chapter takes a closer look at how you can use the Message Broker Toolkit and Microsoft Visual Studio to integrate .NET applications into a message flow.

3.1 Creating a .NETCompute node in a message flow

This section demonstrates how to use Message Broker Toolkit and Visual Studio together to develop a simple flow that counts the number of characters in an input message.

The tasks are to create a message flow with the .NETCompute node and then to create the implementation code for the node using Visual Studio.

3.1.1 Creating the messaging flow

To create a simple message flow that includes a .NETCompute node:

1. Open the Message Broker Toolkit, and create a new application called CountCharsApp.
2. Create a new message flow in the application called CountCharsFlow. The blank message flow will now be open in the canvas. The message processing nodes are displayed in the palette to the left of the canvas.
3. In this example, we use three nodes, the MQInput node, the MQOutput node, and the .NETCompute node. The function of each of these nodes, in this scenario, is defined in the following list:
 - MQInput node: Reads (GETs) a message from an MQ queue using the BLOB (binary) parser and constructs an in-memory representation of this message, which can then be transformed (the logical tree).
 - .NETCompute node: Runs a .NET class implemented in C# that counts the number of characters in the input message and creates a new output message with the total characters listed.
 - MQOutput node: Writes (PUTs) a message to an MQ queue with the content defined in the logical message tree.

To add the nodes to the message flow follow these steps:

- a. Drag the **MQInput node** from the WebSphere MQ drawer of the palette onto the canvas.
- b. Drag the **MQOutput node** from the WebSphere MQ drawer of the palette onto the canvas
- c. Drag the **.NETCompute node** from the Transformation drawer of the palette onto the canvas.

Arrange the message flow as shown in Figure 3-1 and described in step 4.



Figure 3-1 The message flow, before wiring the terminals

4. The execution path taken by a message through the message flow is defined by wiring the terminals of the nodes together. In this example, the execution flow is in the following order: MQInput node → .NETCompute node → MQ Output node.

To wire the flow:

- a. Click the **Out terminal of the MQInput node**, and drag the wire to the **In terminal of the NETCompute node**.

- b. Click the **Out terminal of the .NETCompute node**, and drag the wire to the **In terminal of the MQOutput node**.

The wired flow looks like the example in Figure 3-2.

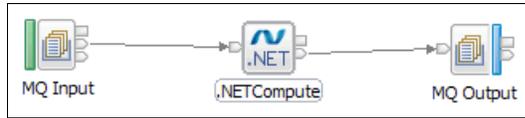


Figure 3-2 The wired message flow

5. The MQInput and MQOutput nodes read and write from MQ queues. The queue used by each node is set in the node properties:
 - a. Click the **MQInput node** in the canvas to open the properties. Click the **Basic tab**, and enter **COUNT_IN** as the queue name, as shown in Figure 3-3.

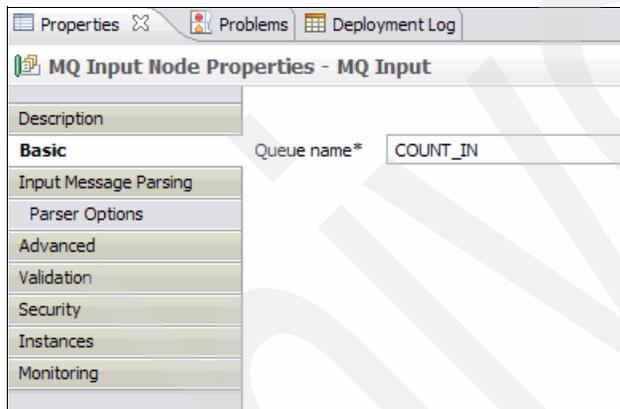


Figure 3-3 Setting the input queue to **COUNT_IN**

- b. Click the **MQOutput node** in the canvas to open the properties. Click the **Basic tab**, and enter **COUNT_OUT** as the queue name, as shown in Figure 3-4.

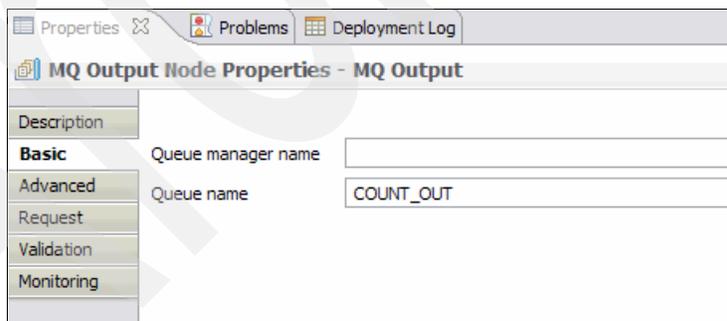


Figure 3-4 Setting the output queue to **COUNT_OUT**

Note: Before the flow is deployed the **COUNT_IN** and **COUNT_OUT** queues must be created on the broker's queue manager.

3.1.2 Creating the code for the .NETCompute node

To develop the .NET portion of the example, we take advantage of the tight integration between the Message Broker Toolkit and Visual Studio.

During installation of WebSphere Message Broker V8, a number of wizards are added to, which can be used to create skeleton code for .NETCompute nodes. Skeleton code is available for the following common node functions:

Creating a message This project is used when the node will create a new logical message tree that is not based on the input message tree.

Filtering a message The node will not modify the message but will route it or filter it based on examining the content of the logical tree.

Modifying a message The node will modify the input data, transforming it into a new structure or enriching it with new data.

Integration steps: The templates for creating WebSphere Message Broker .NET artifacts are installed by default as long as the Message Broker Toolkit is installed after Visual Studio is installed. If Visual Studio is installed after Message Broker Toolkit, you must execute a specific file from the Message Broker Toolkit installation directory to install the Broker .NET templates. This file is called `IBM.Broker.DotNet.vsix` and is located in the `wmbt` subfolder of the Message Broker Toolkit install. Double-click this file and wait until it responds back that the templates are installed. This might not provide any visible feedback until the completion notice, so some patience is required.

In this example, we create a new message tree to hold the number of characters in the message.

To create a new Visual Studio project, to hold the .NET code for the .NETCompute node, follow these steps:

1. In the Message Broker Toolkit, right-click the **.NETCompute node**, and select **Open Microsoft Visual Studio**, as shown in Figure 3-5. Visual Studio launches automatically.

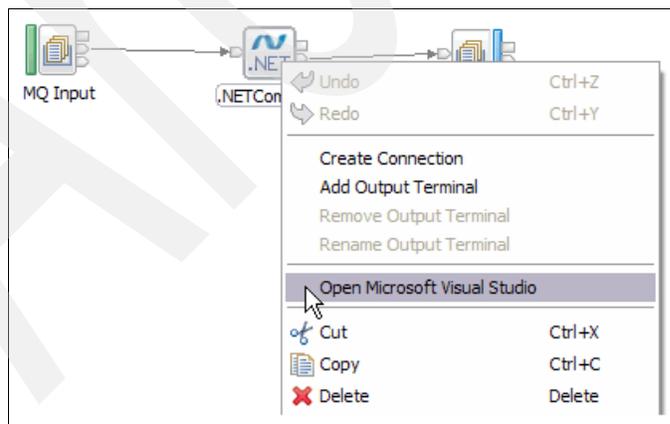


Figure 3-5 Opening Visual Studio

2. Create a new Visual Studio project using the following steps:
 - a. From the Visual Studio menu, select **File** → **New** → **Project**.

- b. From the Installed Template Panel, navigate to **Visual C#** → **Message Broker**.
- c. In the center pane, select **Project to create a Message Broker message**.
- d. Fill in a Name for your project in the Name panel, in this case, CountCharsNode.

The completed dialog looks like the example shown in Figure 3-6.

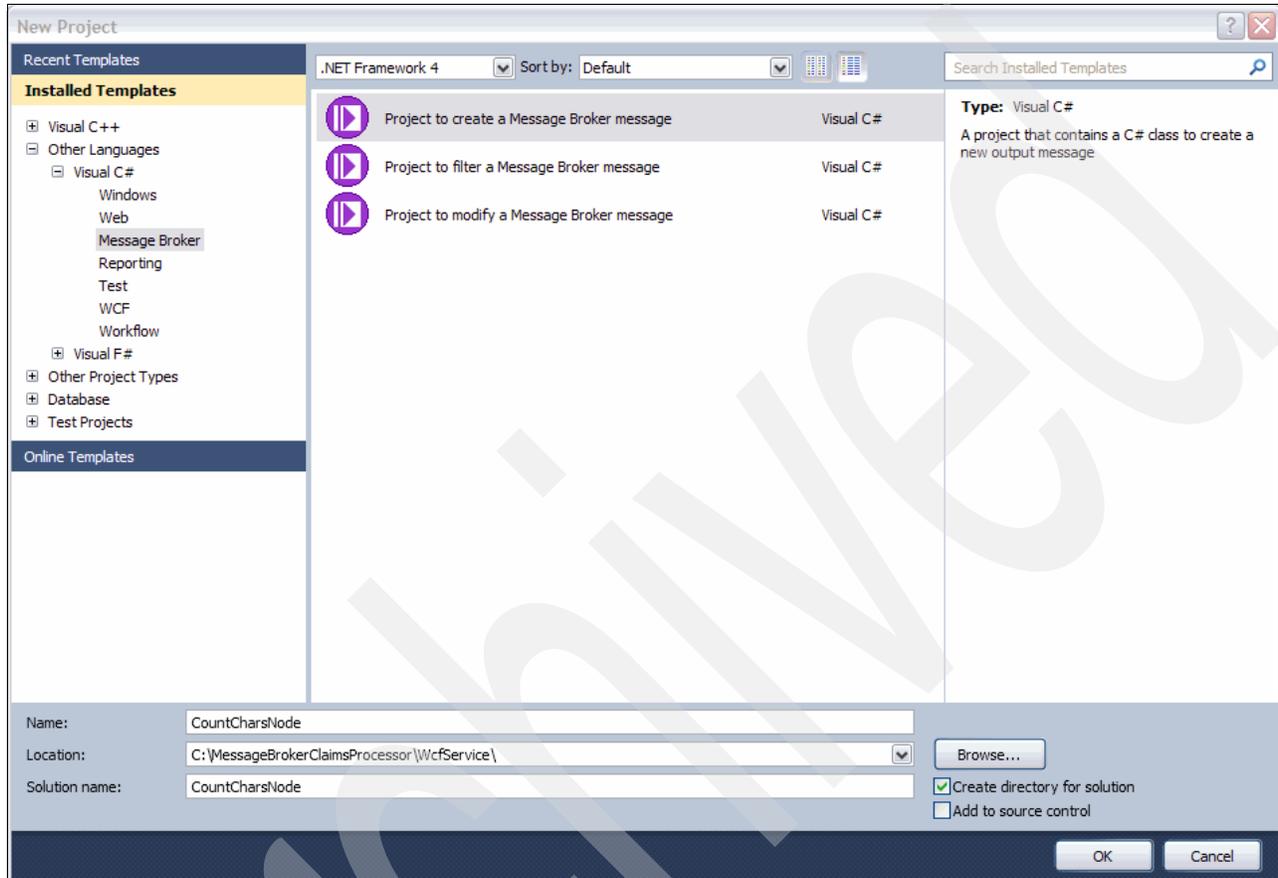


Figure 3-6 Creating a new Visual Studio project

3. Click **OK**. The new project is created, and a new CreateNode.cs class is opened in the editor. See Figure 3-7 on page 58 for an example.

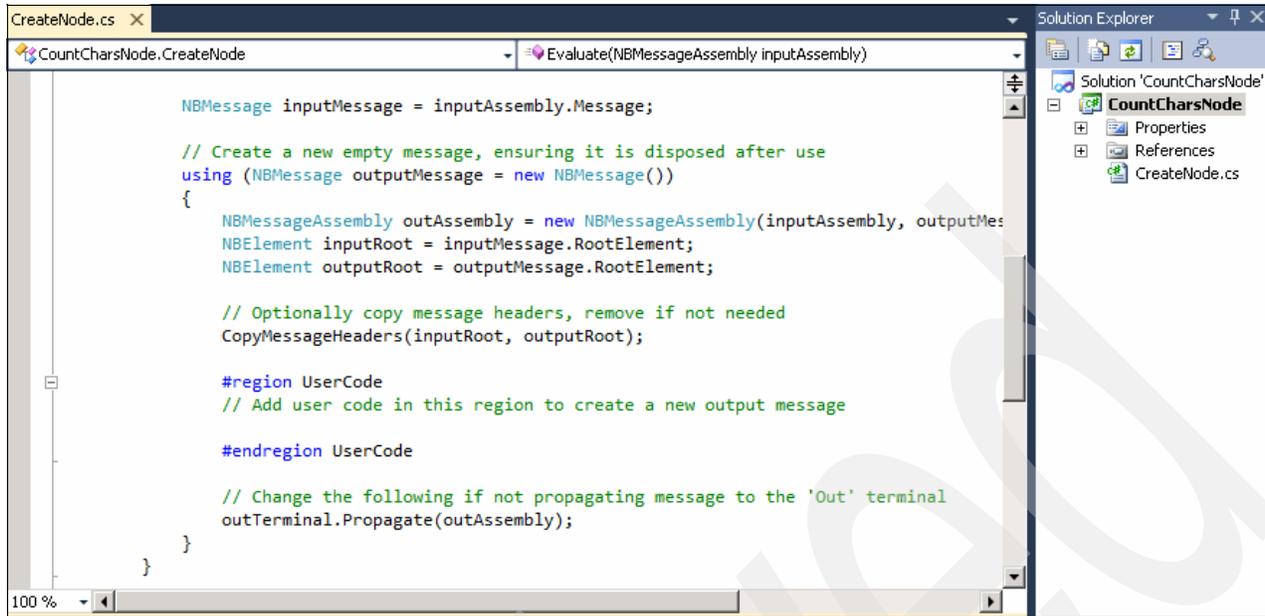


Figure 3-7 The new CreateNode.cs class is opened in the editor automatically

- In WebSphere Message Broker, all .NETCompute nodes are derived from the NBComputeNode class. The New Project wizard will create a new class derived from NBComputeNode and provide a skeleton implementation of the Evaluate() method. The Evaluate() method is invoked each time a message is propagated to the Input terminal of the .NETCompute node. If the message must be propagated to any nodes connected to the .NETCompute node terminals, the Evaluate() method itself must propagate the message using the NBOutputTerminal.Propagate() method.

The skeleton code ensures that this Propagate() method is called and provides a UserCode region for you to add your own code.

Add the code shown Example 3-1 into the UserCode #region of the CreateNode.cs file.

Example 3-1 CreateNode.cs UserCode

```

int length = inputRoot["BLOB"].LastChild.GetByteArray().Length;
NBElement xmlnscOutputElement =
    outputRoot.CreateLastChildUsingNewParser("XMLNSC");
NBElement xmlRoot = xmlnscOutputElement.CreateFirstChild("Output");
xmlRoot.CreateLastChild("Count").SetValue(length);

```

This code references the BLOB parser in the input message tree and obtains its value as a byte array. The length of this byte array is stored in the length integer. A new XMLNSC parser is used to construct an XML message body for the output tree and the length filed is used to populate the Output/Count element.

- After you modify the code for the CreateNode.cs file, save the file and build the project using the **Build** → **Build Solution** menu item. This converts the code from a .cs file into a .NET assembly dll file which can be used by the WebSphere Message Broker runtime.

Note: One instance of CreateNode.cs is created for each copy of the node in the flow. So, if you are running with Additional Instances set, all threads will run against the same CreateNode class.

Care must therefore be taken that the Evaluate() method is threadsafe and reentrant.

Tip: Keep the following points in mind when you define nodes:

- ▶ The Assembly name must include the fully qualified path.
- ▶ The Class name must not include a .cs extension.
- ▶ Alternatively, you can strongly name the dll and load it into the Global Assembly Cache, and on node properties refer to the dll from the cache. More information about this is at the following locations in the WebSphere Message Broker Information Center:
 - Identifying the .NET assembly at run time:
http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.e.tools.mft.doc/bc34225_.htm
 - mqsiAssemblyInstall command:
http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.e.tools.mft.doc/bn34224_.htm

4. Save the message flow.

Tip: There is an alternative to adding a .NETCompute node to the message flow, building the .NET assembly, and then updating the node properties to point to the assembly. If you create the .NET code first and build the solution, you can simply drag and drop the assembly file dll file to the message flow canvas. A .NETCompute node is added, and you are prompted for the class name if more than one class exists.

3.1.4 Deploying the application

To deploy the application to your the broker, drag the **CountCharsApp** icon onto an empty execution group in the Brokers panel of the Message Broker Toolkit, as shown in Figure 3-10.

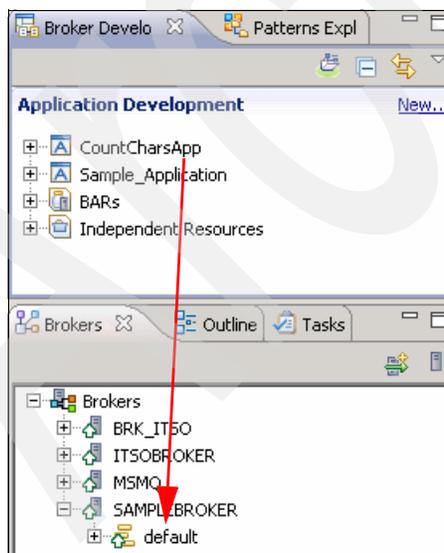


Figure 3-10 Deploying the CountCharsApp application to the default execution group

After the application is successfully deployed, you can use the WebSphere Message Broker Explorer to put a test message into the COUNT_IN queue by following these steps:

1. In the WebSphere Message Broker Explorer, navigate to the queue manager associated with your broker in the Navigator view and select the **Queues** folder.
2. Right-click the **COUNT_IN queue**, and select **Put Test Message**, as shown in Figure 3-11.

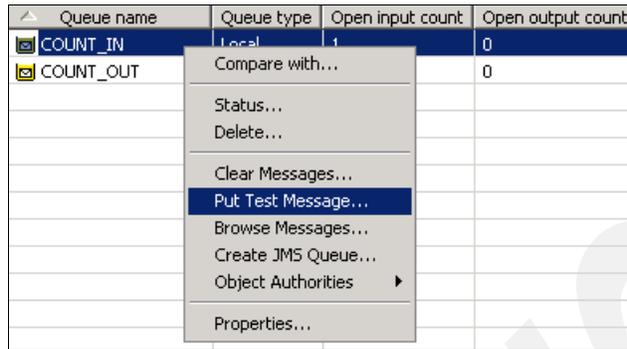


Figure 3-11 Putting a test message

3. Type any stream of characters in the message data box, and click **Put message**. Click **Close**. In the example shown in Figure 3-12, we used the string 123456.



Figure 3-12 Sending a test message

4. Select the **COUNT_OUT queue**, right-click, and select **Browse Messages**. The Message Browser shows the output XML message and shows that the input message was six characters long, as shown in Figure 3-13. Click **Close**.

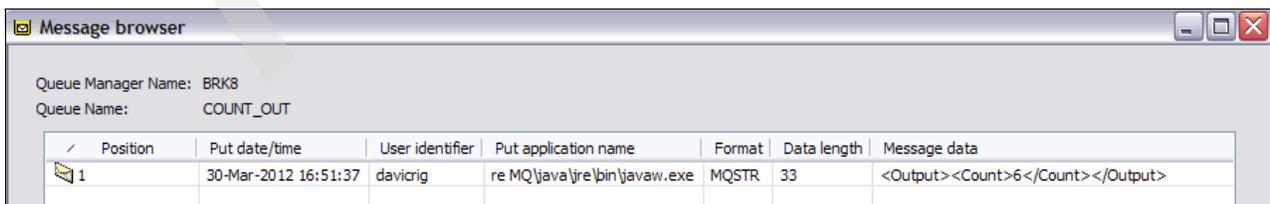


Figure 3-13 Displaying the output Message

3.2 Hot swap deploys

When you are developing a .NETCompute node, it is likely that as a result of testing you must make some changes to the code. The round trip cycle of changing the code, building the assembly, and redeploying to the broker can make this a time consuming process.

To counteract this time loss, the .NETCompute node introduces a feature known as hot swap deploys. This feature ensures that the WebSphere Message Broker runtime will monitor a .NET assembly that the hot swap deploy or node or broker uses for changes. If the WebSphere Message Broker runtime detects any changes to this dll file, it automatically reloads the node and changes will take effect.

To test the hot swap deploy feature:

1. In Visual Studio, modify CreateNode.cs so that the length variable has 1 added to it. You must modify the line shown in Example 3-2.

Example 3-2 Modifying the code in CreateNode.cs

```
int length = inputRoot["BLOB"].LastChild.GetByteArray().Length + 1 ;
```

2. Save CreateNode.cs, and rebuild the project.

In the Windows Event Viewer for applications, you will see a BIP 7451 message indicating that the broker reloaded an assembly, as shown in Figure 3-14.

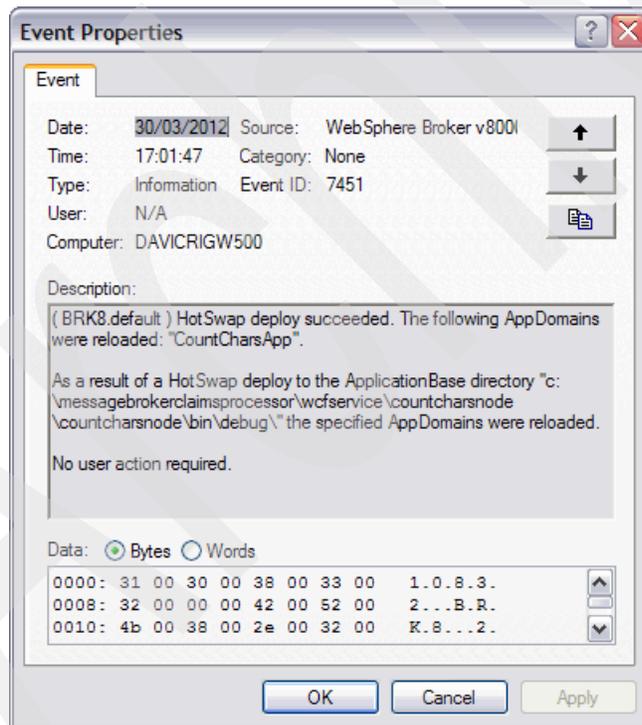


Figure 3-14 A hot swap deploy

3. If you resend the same input message, you can see that the output message contains a count of seven, as shown in Figure 3-15 on page 63.

Position	Put date/time	User identifier	Put application name	Format	Data length	Message data
1	30-Mar-2012 16:51:37	davicrig	re MQ\java\jre\bin\javaw.exe	MQSTR	33	<Output><Count>6</Count></Output>
2	30-Mar-2012 17:03:24	davicrig	re MQ\java\jre\bin\javaw.exe	MQSTR	33	<Output><Count>7</Count></Output>

Figure 3-15 The new input message has a count of 7

3.3 Using the Visual Studio debugger

Another advantage of the integration between Visual Studio and WebSphere Message Broker is that developers of .NETCompute nodes can use the first class debugging capabilities of Visual Studio. This integration saves developers time and gives tools to diagnose problems in their .NET code.

To debug the example:

1. To set a breakpoint in the CreateNode.cs file, click in the **gray margin** in the code editor window, as shown in Figure 3-16. A second click will delete the breakpoint.

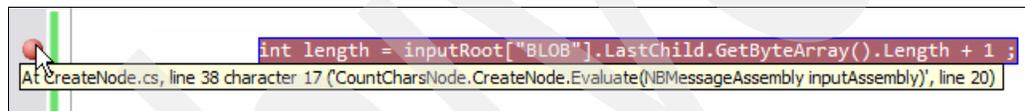


Figure 3-16 Setting a breakpoint

2. Attach the Visual Studio debugger to the running execution groups by selecting **Debug** → **Attach to Process**.
3. In the Attach To Process dialog, select the **DataFlowEngine** process, and click **Attach**. Select the **Show processes from all users** and the **Show processes in all sessions** to see the DataFlowEngine option. There is a DataFlowEngine for each broker that is running. An example of attaching to a process is shown in Figure 3-17 on page 64.

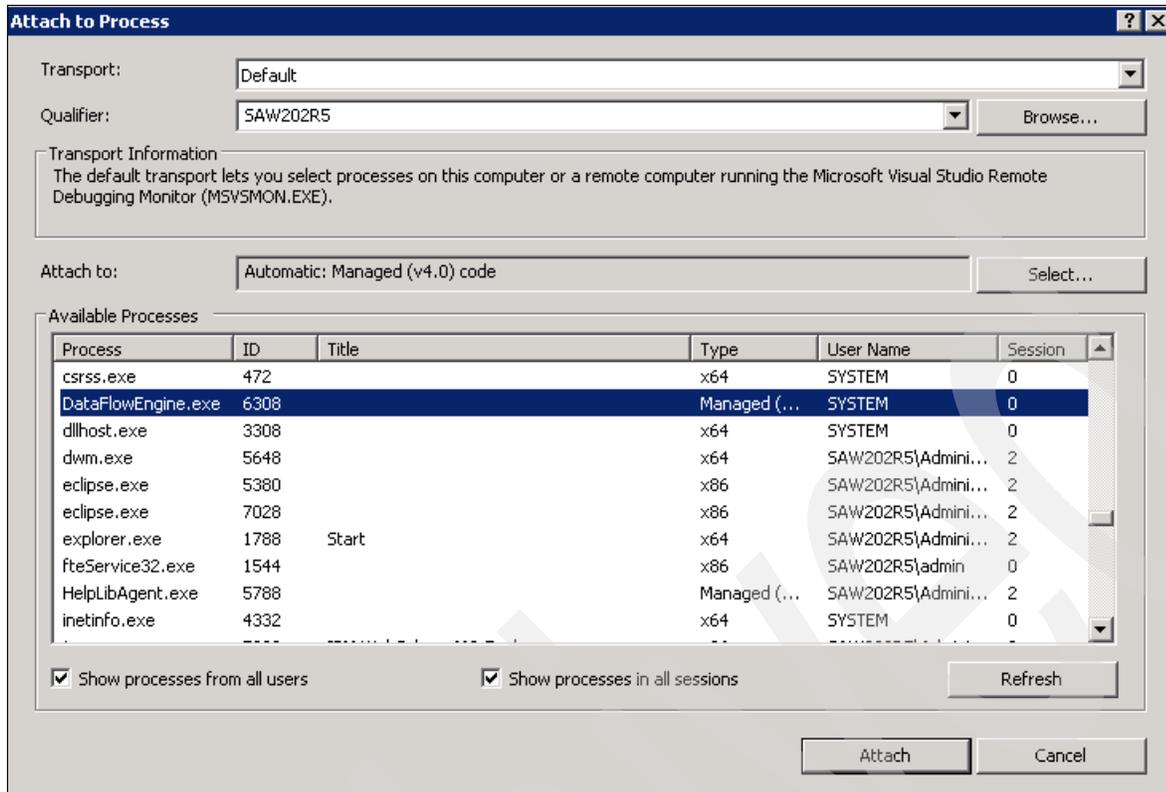


Figure 3-17 Attaching the DataFlowEngine to a process

Execution will now halt whenever the breakpoint is passed. To trigger the debugger, send a third message to the **COUNT_IN queue**. The debugger will then halt, as shown in Figure 3-18 on page 65.

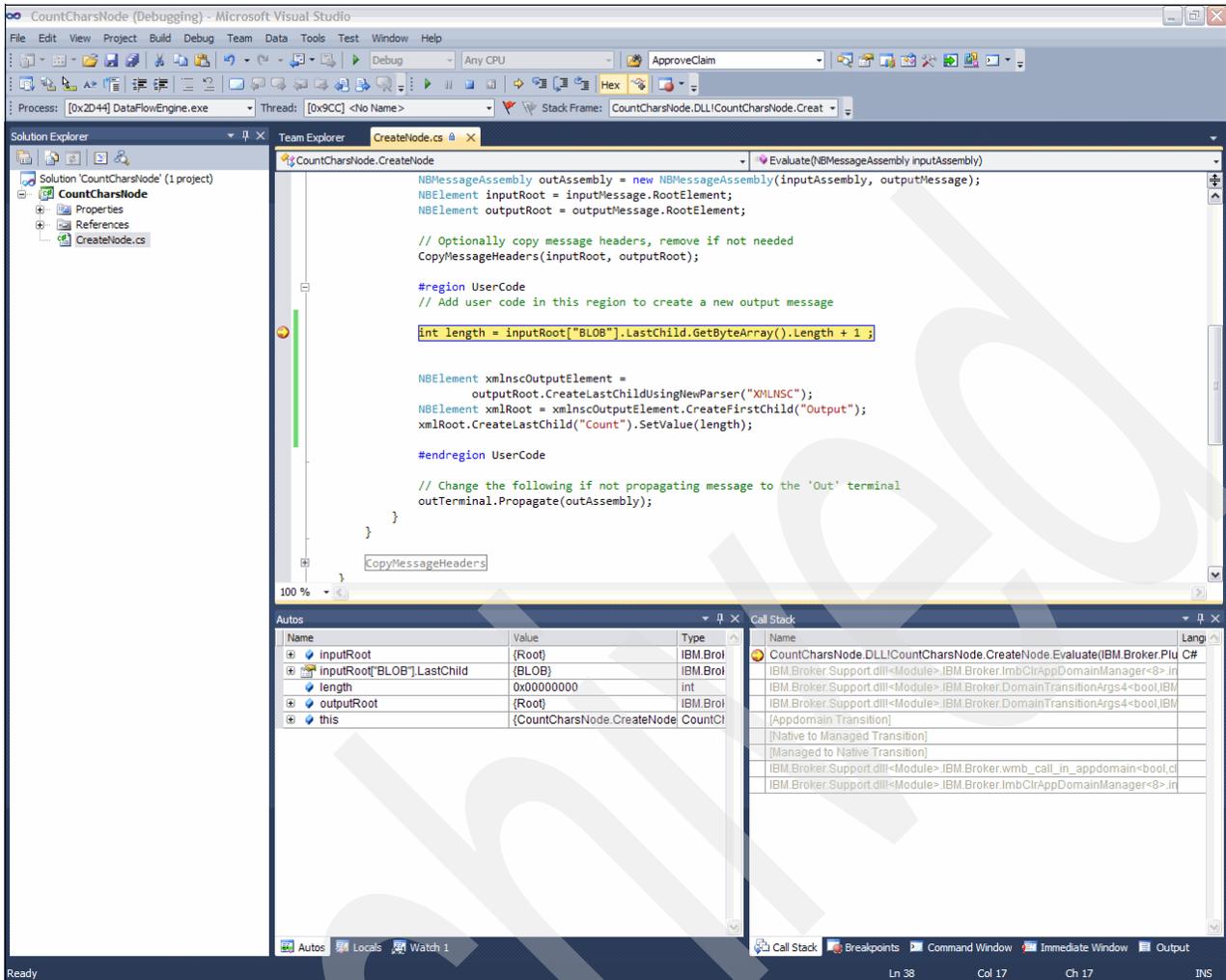


Figure 3-18 Execution halts at the breakpoint

After execution stops, the .NET developer can examine the content of variables, examine the stack backtrace, and single step through execution exactly as they would with a command line .NET application.

You can stop the debug, by selecting **Debug → Stop Debugging**.

Archived



Scenario: Bridging WebSphere MQ and Microsoft Message Queuing

The scenario in this chapter demonstrates a number of best practice development techniques using WebSphere Message Broker and .NET applications. We use the relatively simple case of constructing a messaging bridge between WebSphere MQ and Microsoft Message Queuing (MSMQ). The scenario features a customer who has a sales application that produces and receives data as messages on MSMQ queues and the customer must move these messages to an Enterprise Service Bus (ESB) that uses WebSphere MQ.

This scenario features the following techniques:

- ▶ Receiving messages from MSMQ into a message flow and sending them to WebSphere MQ
- ▶ Receiving messages from WebSphere MQ and sending them to MSMQ
- ▶ Handling transactions against MSMQ objects
- ▶ Using logging from .NET code in a message flow
- ▶ Managing exceptions in .NET code

Additional materials: The WebSphere Message Broker project interchange file and the .NET class code for this scenario can be downloaded from the IBM Redbooks publication web site. See Appendix A, “Additional material” on page 485 for more information.

4.1 Scenario overview

This scenario creates a message flow to bridge between WebSphere MQ and MSMQ, as shown in Figure 4-1.

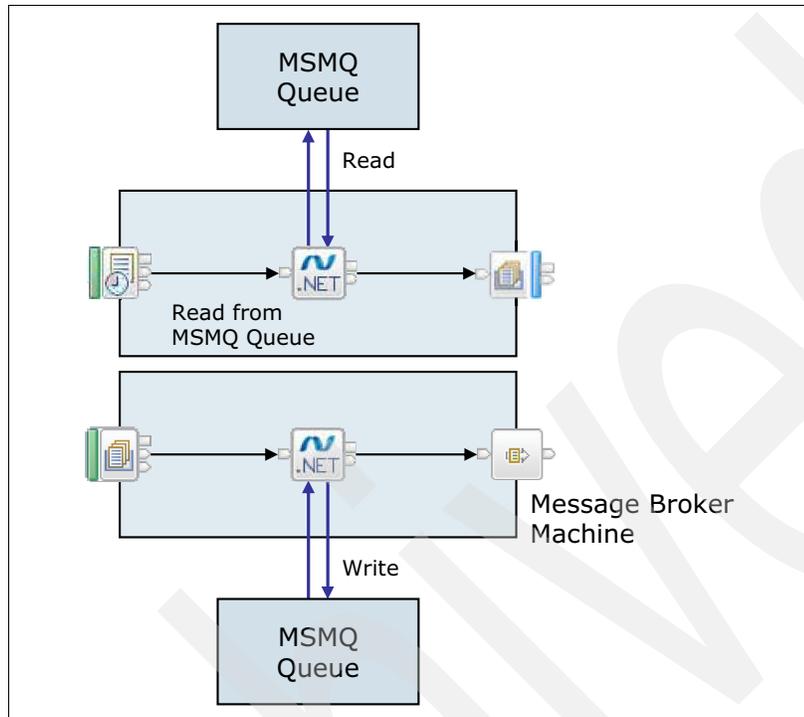


Figure 4-1 WebSphere MQ to MSMQ bridge

A broker flow uses a TimeoutNotification node to initiate message processing on a repeating basis. This process drives a .NETCompute node that reads individual messages from a MSMQ queue until the queue is empty. Each message is read in a transaction and propagated to an MQOutput node. When the MQOutput node completes, the .NETCompute will complete the transaction and read the next message from the queue.

A second flow uses an MQInput node to read individual messages from MQ. These messages are then transformed and written into a MSMQ queue in a transaction. This MSMQ transaction will complete successfully when the MQInput node completes its transaction. The MSMQ will be named by a user-defined property, and the flow will access this and use it.

The .NET code in these nodes is written to expose WebSphere Message Broker development best practices in terms of .NET persistent object management, appropriate exception catching and handling mechanisms, and any needed logging.

This scenario uses the following software:

- ▶ MSMQ is enabled on the same system where WebSphere Message Broker is installed. To enable MSMQ on Windows 2008 Server:
 - a. Select **Start** → **Server Manager**.
 - b. In the Server Manager window, click **Features**, and click **Add Features**.
 - c. Expand **Message Queuing** → **Message Queuing Services**.
 - d. Select **Message Queuing Server**.
 - e. Click **Next**, and then click **Install**.
 - f. After the feature is installed, click **Close**.

- ▶ Microsoft Visual Studio 2010 is installed.
- ▶ WebSphere MQ 7.0.1.3 is installed.
- ▶ WebSphere Message Broker V8 is installed:
 - A broker and its queue manager was created.
 - Queues named ESB.IN and ESB.OUT have been defined on the broker's queue manager.

4.2 Receiving messages from MSMQ

In the first part of this scenario, you will:

- ▶ Construct a message flow that reads messages from a MSMQ queue and forward them to an MQ queue.
- ▶ Construct a simple desktop application to read and write MSMQ messages to provide for some basic testing utilities and to get comfortable with the .NET coding mechanisms for MSMQ.

4.2.1 Constructing a simple utility to read and write MSMQ messages

This section tells you how to construct a Microsoft Forms application to read and write simple MSMQ messages. You can use this application to do some basic testing of your message flows to ensure that they work.

Creating a new Visual Studio project

Open Visual Studio and use the following procedure to create a new project.

1. Select **File** → **New** → **Project**, as shown in Figure 4-2 on page 70.

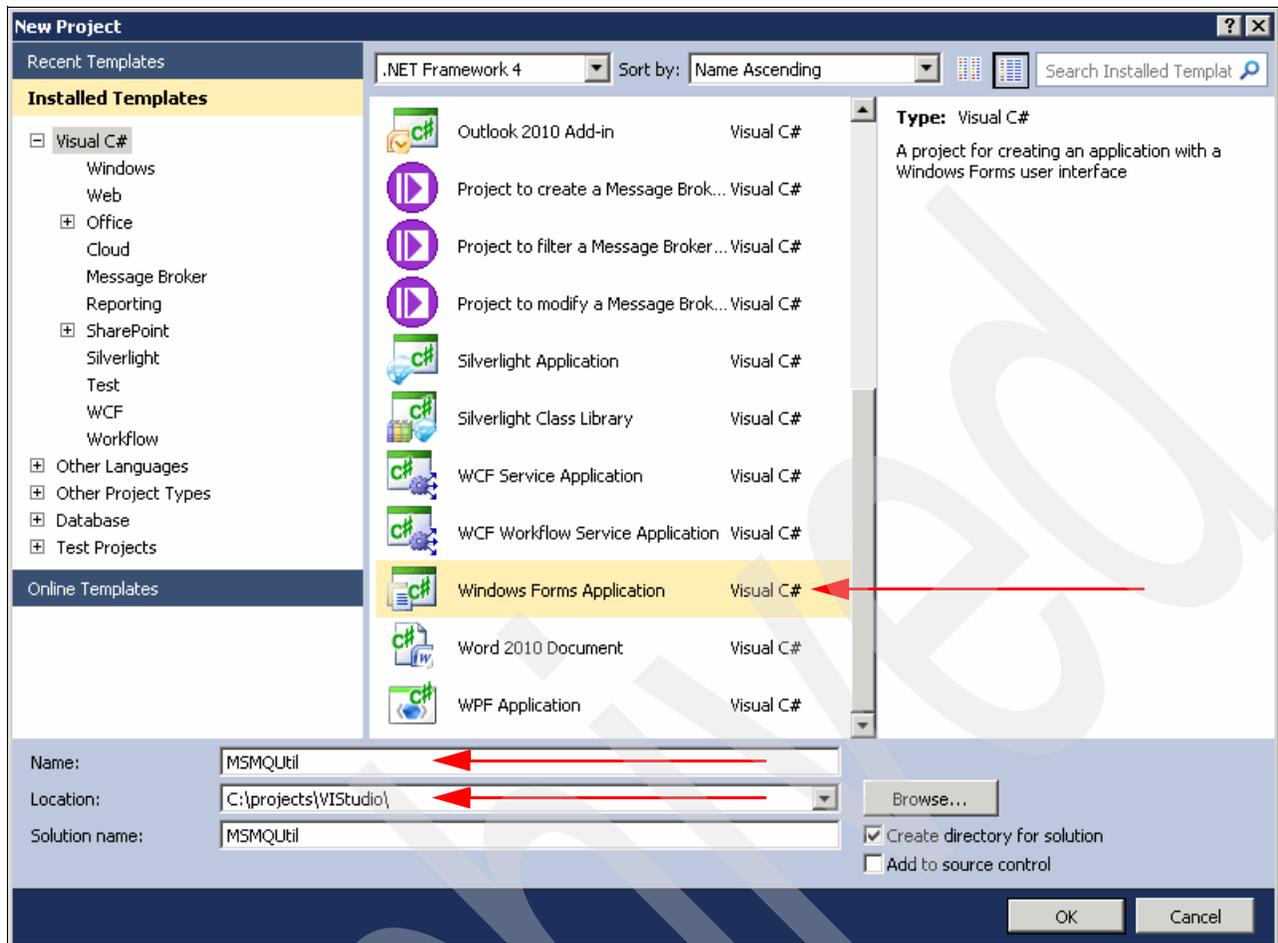


Figure 4-2 Create a new Windows Forms Application

Perform the following steps:

- In the Recent Templates column, select **Visual C#** and in the list of templates, select **Windows Forms Application**.
- In the Name field, enter `MSMQUtil`.
- In the Location field, enter `c:\projects\VISudio`.

Note that the location of the project is moved from the default directory for Visual Studio projects to provide a more convenient location for keeping the .NET code and the WebSphere Message Broker code together. The WebSphere Message Broker code will be kept in a workspace under `c:/projects` as well.

In the Solution field, `MSMQUtil` is automatically filled in for you.

- Click **OK**. The new form displays, as shown in Figure 4-3 on page 71.

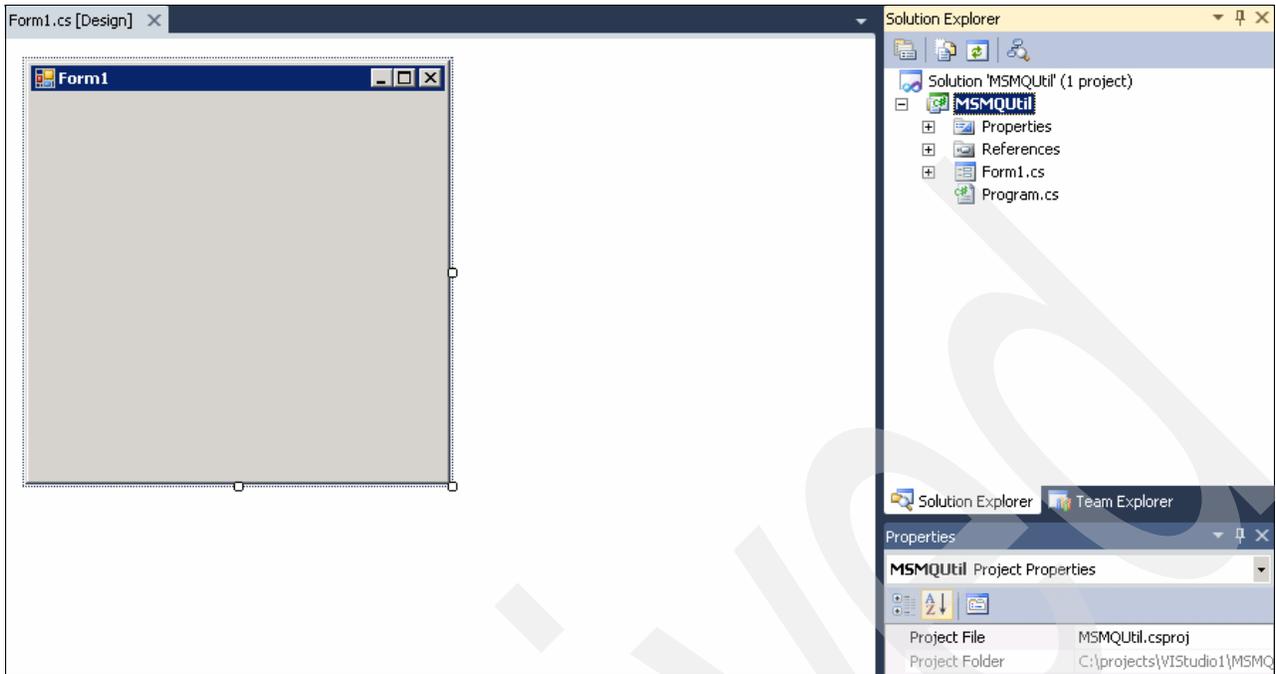


Figure 4-3 New Forms Application after creation

Constructing the form

The next step is to construct the form:

1. Rename the form. In the Solution Explorer, right-click **Form1.cs** and select **Rename**. Enter `MSMQUtil.cs` as the new name.
2. At the prompt shown in Figure 4-4, click **Yes**.

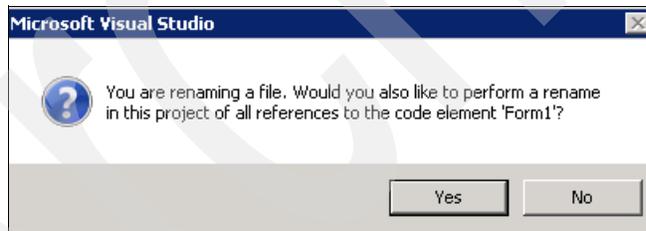


Figure 4-4 Rename prompt

3. Use the Visual Studio Designer to create this form. You can access the Designer by selecting the `MSMQUtil.cs` file in the Solution Explorer and clicking the **View Designer** button, as shown in Figure 4-5.



Figure 4-5 Select View Designer

4. In the Toolbox palette on the left side of the window, expand **Common Controls**, and use your cursor to drag-and-drop the necessary control objects into the appropriate locations to create the form shown in Figure 4-6. You can also use your cursor to size the form and the control objects in the form.

If you do not see the Toolbox, select **View** → **Toolbox**.

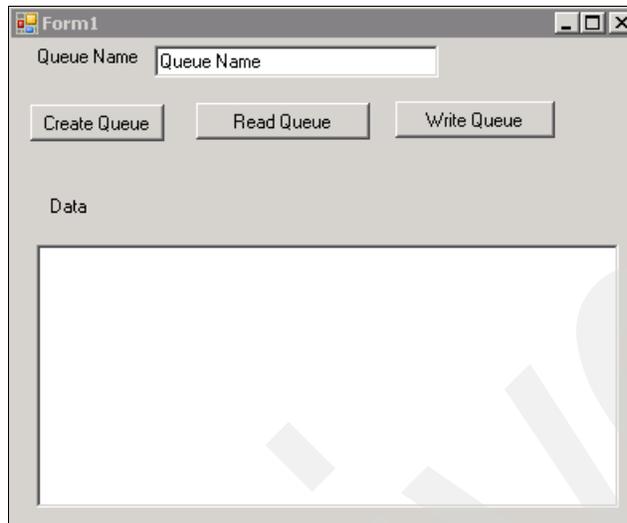


Figure 4-6 The basic MSMQUtil form

To create the form:

- a. Add a Label and a Textbox at the top of the form to allow the user to input the name of the MSMQ queue. Enter Queue Name as the text for both the label and the textbox.
The visible attributes of the control objects are set in the properties pane of Visual Studio, including the names of the objects. If the properties pane is not open and visible, select one of the control objects, and right-click to choose **Properties**.
- b. Add three buttons to the form.
- c. Use the Properties view to add the text shown in Figure 4-6 to the Text property for each button.
- d. Use the Properties view to name the buttons createQueue, readQueue, and writeQueue respectively. The name is set in the (Name) property in the Design section of the properties.
- e. Add another Label called Data below the row of buttons and a RichTextBox to hold the contents of the message.

Save the design and close the Properties view.

5. Add a reference in the project to the System.Messaging component. In the Solution Explorer, right-click **References** and select **Add Reference**.
6. In the Add Reference window, shown in Figure 4-7 on page 73, select the **.NET** tab and then select the **System.Messaging** component. Click **OK**. This reference allows your code to find the necessary runtime libraries when it executes. It also allows Visual Studio to validate the code you're writing against those runtime libraries.

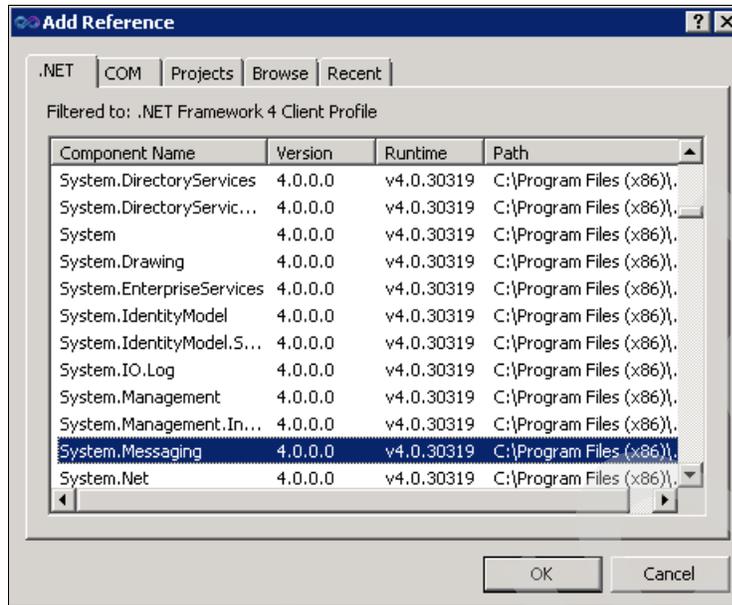


Figure 4-7 Add References

7. Save the design by pressing Ctrl+S.

Writing the code

To construct the three methods that will be called by the different buttons:

1. Double-click the form design to open the `MSMQUtil.cs` code.
2. Add the `System.IO` and `System.Messaging` namespaces to the code. Add the statements shown in Example 4-1 to the list of using statements at the top of the file.

Example 4-1 Add the using statement for `System.Messaging` namespace

```
using System.Messaging;
using System.IO;
```

3. Declare some class variables (shown in bold in Example 4-2). Create a variable to hold the contents of the message we are going to write, a variable to hold the Message Queue object, and a variable to hold the name of the queue we are going to open.

Example 4-2 Class variables for `MSMQUtil`

```
namespace MSMQUtil
{
    public partial class MSMQUtil : Form
    {
        private String myText;
        private MessageQueue myQ;
        private String myName;
        public MSMQUtil()
    }
}
```

4. Add the code for the methods that will get performed when the user clicks each of the three buttons. The `createQueue_Click` method will open or create the queue. Add the code shown in Example 4-3 on page 74.

To create the stub of this method and ensure that it is invoked when the button is pushed, double-click each of the three buttons in the Designer view. It will create and associate a new method with each button.

Example 4-3 Create Queue method for MSMQUtil

```
private void createQueue_Click(object sender, EventArgs e)
{
    myName = textBox1.Text;
    if (!MessageQueue.Exists(myName))
    {
        myQ = MessageQueue.Create(myName);
    }
    else
    {
        myQ = new MessageQueue(myName);
    }
}
```

5. The readQueue_Click method opens the queue, if it is not already open (and create it if it is not created), and then extracts the first message and puts the contents of the message into the text box. It then displays a user message to indicate that the reading was successful. Add the code shown in Example 4-4 to the stub method created by double-clicking this button.

Example 4-4 Read a message from the queue and populate the text box

```
private void readQueue_Click(object sender, EventArgs e)
{
    myName = textBox1.Text;
    if (!MessageQueue.Exists(myName))
    {
        createQueue_Click(sender, e);
    }
    else
    {
        myQ = new MessageQueue(myName);
    }

    System.Messaging.Message myMsg;
    myMsg = myQ.Receive();
    if (myMsg != null)
    {
        Byte[] myBytes = new Byte[(int)myMsg.BodyStream.Length];
        StreamReader myTR = new StreamReader(myMsg.BodyStream);
        myText = myTR.ReadToEnd();
        richTextBox1.Text = myText;
        MessageBox.Show(this, "Read the message from queue " + myName +
            "successfully", "MSMQ UTIL",
            MessageBoxButtons.OKCancel);
    }
}
```

The code uses a StreamReader to get the contents of the message body as a single-String field.

- The `writeQueue_Click` method will also create and open the queue. It then puts whatever text is in the text box into a local string variable and writes that into a new message on the queue. It then displays a user message indicating that the message was successfully put on the queue. Add the code shown in Example 4-5 to the stub method generated by double-clicking the button.

Example 4-5 Write the contents of the text box as an XML Message to the queue

```
private void writeQueue_Click(object sender, EventArgs e)
{
    myName = textBox1.Text;
    try
    {
        if (!MessageQueue.Exists(myName))
        {
            createQueue_Click(sender, e);
        }
    }
    else
    {
        myQ = new MessageQueue(myName);
    }
    myText = richTextBox1.Text;
    System.Messaging.Message myMsg = new System.Messaging.Message() ;
    StreamWriter mySW = new StreamWriter(myMsg.BodyStream);
    mySW.Write(myText);
    mySW.Flush();
    if (myText != null)
    {
        myQ.Send(myMsg);
    }

    MessageBox.Show(this, "Wrote the message to queue " + myName +
        "successfully", "MSMQ UTIL",
        MessageBoxButtons.OKCancel);
    richTextBox1.Text = "";
}
catch (Exception)
{
    throw;
}
}
```

This code uses a `StreamWriter` to insert the contents of the `String` into the message body, and then flushes the buffer to ensure that the string is written. This method produces a plain text message on the queue.

- Close the `MSMQUtil.cs` file and the form design.

Testing the program

To execute some basic testing to ensure that the code works:

- Launch the program by clicking **Start Debugging** in Visual Studio, shown in Figure 4-8.



Figure 4-8 Start Debugging button

- In the Queue Name box, enter the value `.\private$\test`. Click the **Create Queue** button. The application does not notify you that this method succeeds, but you can check it using the Server Manager Console (see step 4).
- Enter some text into the Data box, such as the text shown in Figure 4-9. Click the **Write Queue** button.

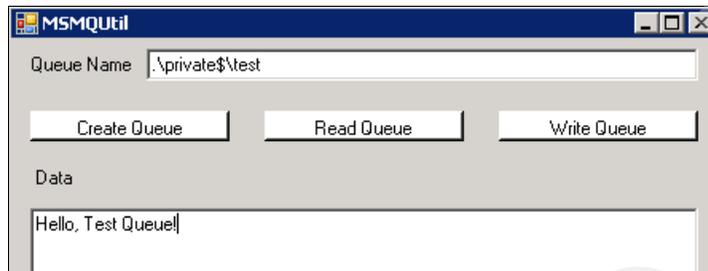


Figure 4-9 Testing the Write Queue button

The success message box is displayed, as shown in Figure 4-10.



Figure 4-10 Test message sent successfully

- After the dialog box indicating that the message was written successfully is displayed, open the Windows Server Manager console, and navigate to your new private queue in the MSMQ section, as shown in Figure 4-11 on page 77.

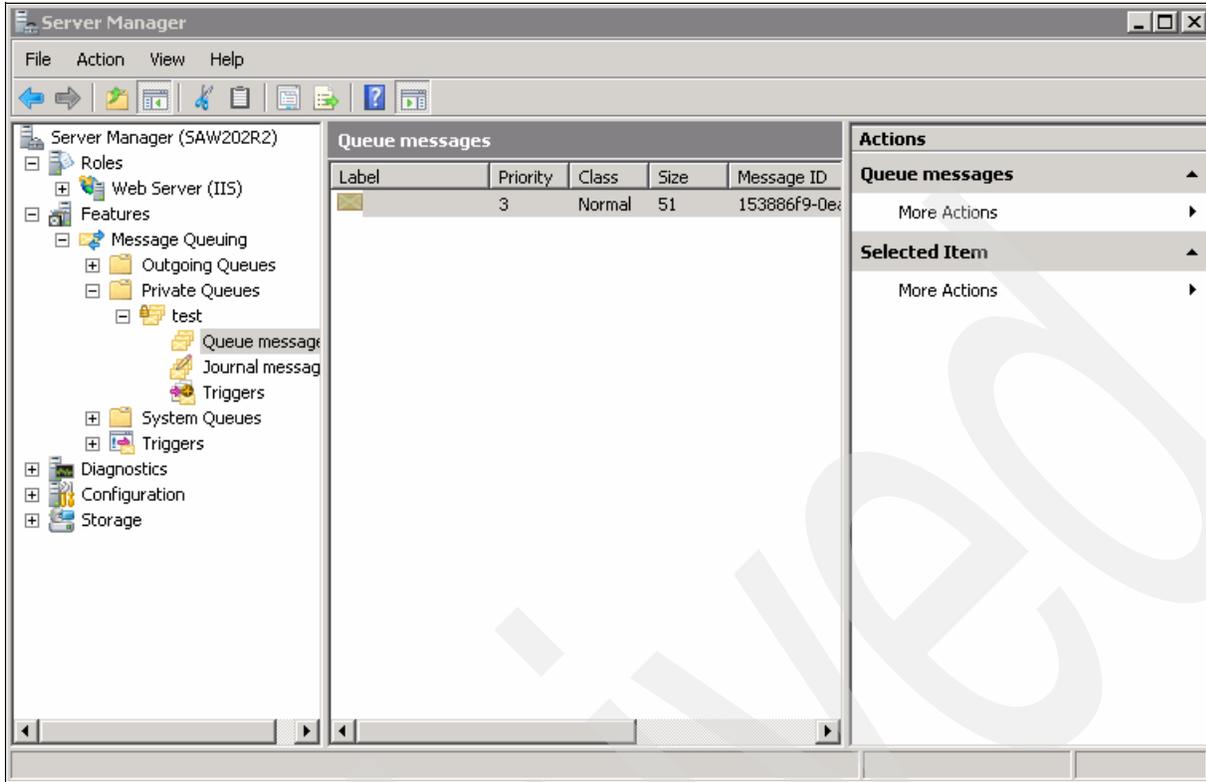


Figure 4-11 MSMQ Server Manager view

5. Right-click the message, and select **Properties** in the Body tab to view the message you sent in plain text, as shown in Figure 4-12.

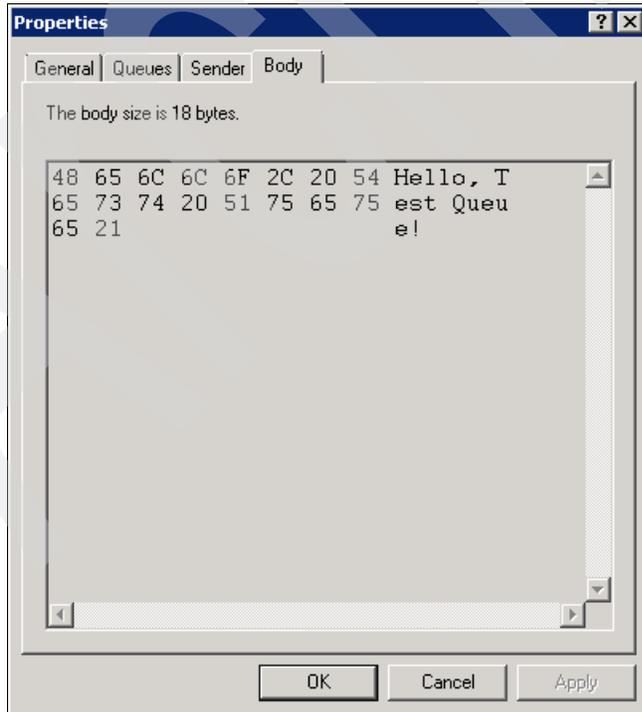


Figure 4-12 MSMQ message properties showing the Body

4.2.2 Constructing and testing a basic message flow

The next step is construct a message flow that will use .NET code to read messages that are delivered to an MSMQ queue. WebSphere Message Broker V8 does not currently provide a means for .NET code to initiate a message flow. There is not a .NETInput node, and you cannot create a custom Input plug-in using .NET code. However, the Standard and Advanced Editions of WebSphere Message Broker V8 provides the *TimeoutNotification* node. This node propagates an empty message to its out terminal at a configurable interval.

Before starting this procedure, make sure that WebSphere Message Broker V8 is running.

Constructing the message flow

To create the message flow:

1. Open the WebSphere Message Broker Toolkit.
2. Create a new application called salesConnector.
3. Create a new message flow called readSales. (See 2.7, “Getting started” on page 33)
4. Construct the message flow using the following steps:
 - a. Drag-and-drop the TimeoutNotification node in the Timer folder on the Palette to the Canvas.
 - b. Drag-and-drop the .NETCompute node in the Transformation folder on the Palette to the Canvas.
 - c. Drag-and-drop the MQOutput node from the WebSphere MQ folder on the Palette to the Canvas.
 - d. Wire the out terminal of the TimeoutNotification node to the in terminal of the .NETCompute node.
 - e. Wire the out terminal of the .NETCompute node to the in terminal of the MQOutput node.

Your flow now looks like the flow in Figure 4-13.

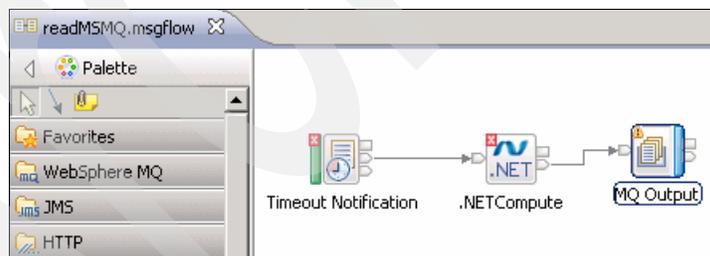


Figure 4-13 Message Flow to read MSMQ

The red boxes and yellow warning flags are present because the nodes have not yet been configured.

Configuring the message flow

To configure the relevant properties on the message flow, including adding the necessary user-defined properties:

1. Select the **TimeoutNotification** node on the canvas and, in the Properties view Basic tab, set the Unique identifier to the value readSales.
2. Promote the Timeout interval (sec) property to the message flow.

- a. Right-click the TimeoutNotification node, and select **Promote Property**. This process allows the property to be set on the flow along with the user-defined properties and allows the timeout interval to be altered at runtime without redeploying the flow.
- b. In the Available node properties column, expand the Basic section, select **Timeout Interval (sec)**, and click **Promote**. The Target Selection screen displays.
- c. Click **OK** to put the property in the readSales group.

The results are shown in Figure 4-14.

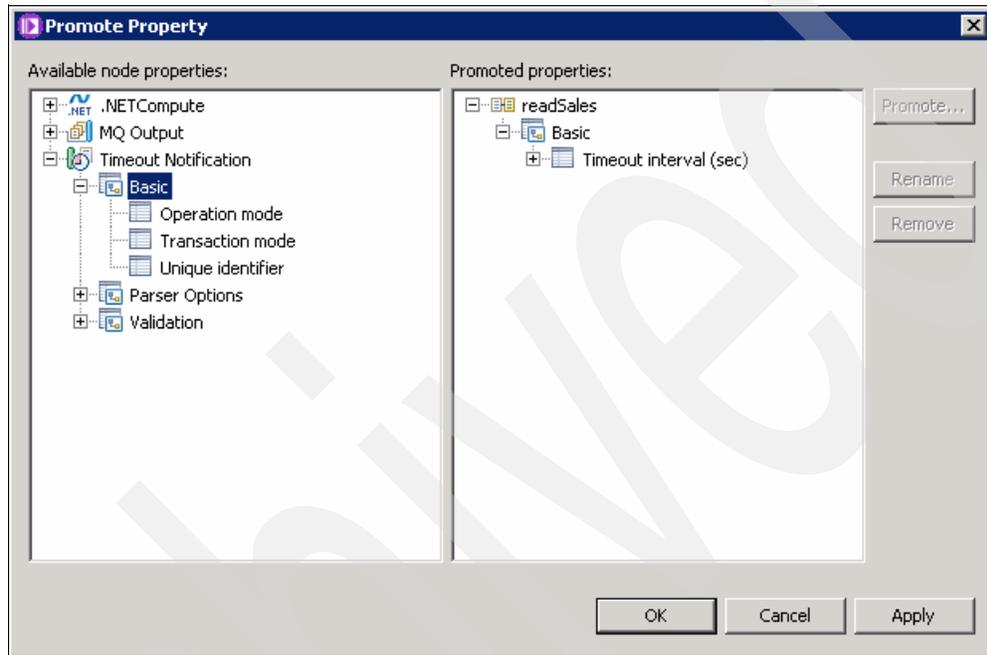


Figure 4-14 Promoted properties

- d. Click **OK** again to close the window. Figure 4-15 shows the resulting properties.

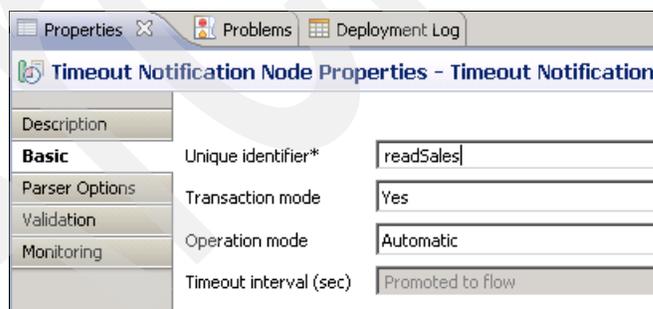


Figure 4-15 Timeout Node properties

3. Select the **MQ Output** node on the Canvas and, in the Basic tab of the Properties view, specify a Queue name of **ESB.IN**, as shown in Figure 4-16 on page 80.

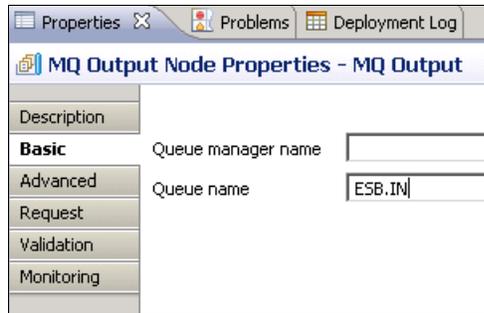


Figure 4-16 MQOutput properties

4. Create two user-defined properties, one to hold the name of the MSMQ queue and one to hold the number of seconds to wait for a new message. Both properties are String type because non-String properties are not currently supported by the runtime.

To create the properties:

- a. Select **User Defined Properties** at the bottom of the Message Flow Editor.
- b. Select **Basic**, and click the **Add Property** button, as indicated by the arrow in Figure 4-17. Enter `SalesQueueName` and select **String** as the type to define the name of the queue.
- c. Click the **Add Property** button again. Enter `SalesWaitInterval`, and select **String** as the type to define the wait interval.

Figure 4-17 shows the two new user properties.

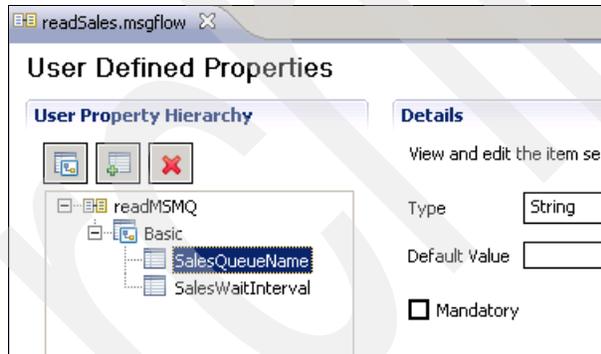


Figure 4-17 User defined properties

5. Return to the Graph view.

Creating the .NET code

The next step is to create the Visual Studio project and the .NET code associated with the .NETCompute node. The Message Broker Toolkit makes it easy to do this.

To create the Visual Studio project and the .NET code associated with the .NETCompute node:

1. Double-click the **.NETCompute** node in the message flow to launch Visual Studio.
2. Select **File** → **New** → **Project** to create a new project.

Follow these steps to configure the project:

- a. In the Recent Templates column, select **Visual C# → Message Broker**. In the list of templates, select **Project To Create A Message Broker message**.
 - b. In the Name field, enter ReadQueue.
 - c. In the Location field, enter the location to store the project, for example, c:\projects\VISstudio.
 - d. In the Solution field, ReadQueue is automatically populated.
 - e. Click **OK**.
3. An editor with the new code template is displayed. Add a reference in the project to the System.Messaging component.
 - a. In the Solution Explorer, right-click **References**, and select **Add Reference**.
 - b. In the Add Reference window (see Figure 4-7 on page 73), select the .NET tab, and select the **System.Messaging** component.
 - c. Click **OK**.
 4. Click the **CreateNode.cs** tab to display the code, add the System.Messaging namespace, and add the System.Globalization namespace to the code by adding the following syntax. Use statements from the list at the top of the file:

```
using System.Messaging;  
using System.Globalization;
```
 5. Save your changes by pressing Ctrl-S.

Writing the code

To create the code for the .NETCompute node:

1. The class that is created has a default name of CreateNode. Even though it is put into a namespace of ReadQueue after your Solution name, we change the name of the class to ReadQueueNode instead:
 - a. In the Solution Explorer, right-click **CreateNode.cs**, and select **Rename**.
 - b. Rename the class to ReadQueueNode.cs.
 - c. A Visual Studio message is displayed that asks if you want to also rename all references to this element. Click **Yes**.
2. Add the code, shown in bold in Example 4-6, to the ReadQueueNode class in ReadQueueNode.cs. to create the following class variables:
 - The salesQueue variable is the Microsoft Messaging Queue object representing the queue.
 - The salesQName holds the actual name of the queue.
 - The salesWaitInterval is the time in seconds that the flow waits for a new message to arrive before terminating the flow.

Example 4-6 ReadQueueNode class variable

```
public class ReadQueueNode : NBComputeNode  
{  
    MessageQueue salesQueue;  
    String salesQName;  
    int salesWaitInterval;  
}
```

- d. Create a constructor for the node. Add the code in Example 4-7 directly below the last variable from Example 4-6 on page 81.

Tip: If you start typing `public override`, the code completion in Visual Studio displays a pop-up of the methods of the class to override. Select **OnInitialize** to obtain the skeleton of the method.

```
public override |
/// <summary>
/// Evaluate Met
/// </summary>
/// <param name=
Equals(object obj)
GetHashCode()
OnDelete()
OnInitialize()
```

Example 4-7 OnInitialize method of the node class

```
public override void OnInitialize()
{
    base.OnInitialize();
    salesQName = GetUserDefinedPropertyAsString("SalesQueueName");
    String myInt = GetUserDefinedPropertyAsString("SalesWaitInterval");
    try
    {
        salesWaitInterval = Convert.ToInt32(myInt, CultureInfo.InvariantCulture);
    }
    catch (System.FormatException )
    {
        salesWaitInterval = 30;
        //Default value of 30, if the User Defined Property is not a valid integer.
    }
    catch (System.OverflowException )
    {
        salesWaitInterval = 30;
        //Default value of 30, if the User Defined Property is not a valid integer.
    }
    if (!MessageQueue.Exists(salesQName))
    {
        salesQueue = MessageQueue.Create(salesQName);
        salesQueue.SetPermissions("Users", MessageQueueAccessRights.FullControl);
    }
    else
    {
        salesQueue = new MessageQueue(salesQName);
    }
}
```

You now have code that reads the user-defined properties that you created on the message flow. This code uses the name property to create and open the MSMQ queue to be used during the main flow execution.

We also use a basic try/catch block to ensure that the `SalesWaitInterval` user-defined property on the flow is convertible to an integer value. If it is not convertible to an integer value, we ignore the reason and set the timeout to a fixed 30-second interval. WebSphere Message Broker does not currently support user-defined properties that are not String values; therefore, we cannot enforce this value to be an integer at deployment time, so we use the try/catch logic to enforce it during runtime.

If the broker code created the queue, it also explicitly grants all permissions to all members of the Windows group “Users”. This process ensures that our desktop application can interact with the same queue. Remember that your broker is not going to run as the same user that you are logged in as, and otherwise does not have sufficient access.

You can also manually create the MSMQ queue before the message flow tries to access it. If you do this, you must ensure that you set the needed permissions. The desktop application that we wrote in 4.2.1, “Constructing a simple utility to read and write MSMQ messages” on page 69 does not set any permissions on the queue, so it is automatically private to the user running the application. You can modify the application to add the same code to set all permissions, or you can use the Microsoft Server Manager console to create the queue and set whatever permissions are appropriate.

For the purposes of this scenario, the broker creates the queue and thus all users have full control.

3. Having created the constructor, you must create the destructor to make sure that the queue is closed when we are finished. Add the code shown in Example 4-8 directly below the OnInitialize() method.

Example 4-8 OnDelete method of the node class

```
public override void OnDelete()  
{  
    salesQueue.Close();  
    base.OnDelete();  
}
```

4. The next step is to construct the method that is invoked when the TimeoutNotification node passes control to the in terminal of the .NETCompute node. This is the Evaluate method. A skeleton of this method is already created for you that handles the main parts of creating a new message and sending it to the out terminal.

We must modify this method so that it repeatedly reads messages from the MSMQ queue, puts the data from each message into the logical message tree, and forwards this to the out terminal for writing to an MQ queue.

To accomplish this, perform the following steps:

- a. Create the skeleton of the output message. This is created once for each time the .NETCompute node is invoked, and then reused for each MSMQ message read.
- b. Create a loop that gets each message from the queue until the queue indicates that it is currently empty.
- c. Read a message from the queue, and specify a timeout using the value of the user-defined property that was read in the constructor method. If the message is not returned until the timeout expires, an IOException is thrown. If the IOException is not thrown, read the bytes from the BodyStream of the MSMQ message object, insert them into the NBElement in the logical message tree, and then propagate out of the node.

Use the code in Example 4-9. Add this code to the user region in ReadQueueNode.cs.

Example 4-9 UserCode region of Evaluate() method

```
#region UserCode  
// Add user code in this region to create a new output message  
// Create the output message structure for a BLOB message.  
  
    outputRoot.CreateLastChildUsingNewParser(NBParsers.BLOB.ParserName);
```

```

    NBElemet blob = outputRoot[NBParsers.BLOB.ParserName].CreateLastChild(null,
"Blob");

    // Create the object to hold each MSMQ message.
    System.Messaging.Message myMsg = new Message();

    // read the queue until it's empty.
    while (myMsg != null)
    {
        try
        {
            //Read the message from the queue.
            myMsg = salesQueue.Receive(new TimeSpan(0, 0,0, salesWaitInterval));

            //Construct and populate a Byte array holding the data in the msg body
            Byte[] myBytes = new Byte[(int)myMsg.BodyStream.Length];
            myMsg.BodyStream.Read(myBytes, 0, (int)myMsg.BodyStream.Length);

            //Add data to the NBElemet that represents the body of the blob message
            blob.SetValue(myBytes);

            //Send the message to the next node in the message flow.
            outTerminal.Propagate(outAssembly);
        }
        catch (System.Messaging.MessageQueueException e)
        {
            if (e.MessageQueueErrorCode == MessageQueueErrorCode.IOTimeout)
                myMsg = null;
            else
                throw;
        }
    }
}
#endregion UserCode

```

5. Most of the code that is outside of the marked user region can be left alone. The line of code immediately below the UserCode region that calls propagate (in bold in Example 4-10) is redundant, so comment it out.

Example 4-10

```

#endregion UserCode
// Change the following if not propagating message to the 'Out' terminal
//outTerminal.Propagate(outAssembly);

```

6. Save and close the ReadQueueNode.cs file.
7. Select **Build** → **Build Solution** to test the code.

Select the Output tab at the bottom of the screen to view the messages from the build and to ensure that the build was successful.

Updating the .NETCompute node

Now that you have the code, you must update the message flow to use it:

1. In the Message Broker Toolkit, click the **.NETCompute** node.

2. In the Properties view, select the **Basic** tab:
 - a. In the Assembly name field, browse to the following location, and select the ReadQueue.dll file:


```
C:\projects\VIStudio\ReadQueue\ReadQueue\bin\Debug\ReadQueue.dll
```
 - b. Click **Open**.
 - c. Enter the ReadQueueNode as the class name.
3. Select the **Advanced** tab in the Properties view, and enter ReadSales as the AppDomain name. This process ensures that this .NETCompute node's DLL is loaded in a unique .NET application domain and does not interfere with other flows in the same broker application.
4. Save the message flow.

Testing the message flow

To test the message flow:

1. Create a BAR file to deploy your message flow application. In the Message Broker Toolkit, right-click your application, and chose **New** → **BAR File**. The window shown in Figure 4-18 is displayed.
2. Enter a name for the BAR file and either put the file in the normal BAR Files container or a project of your choosing.

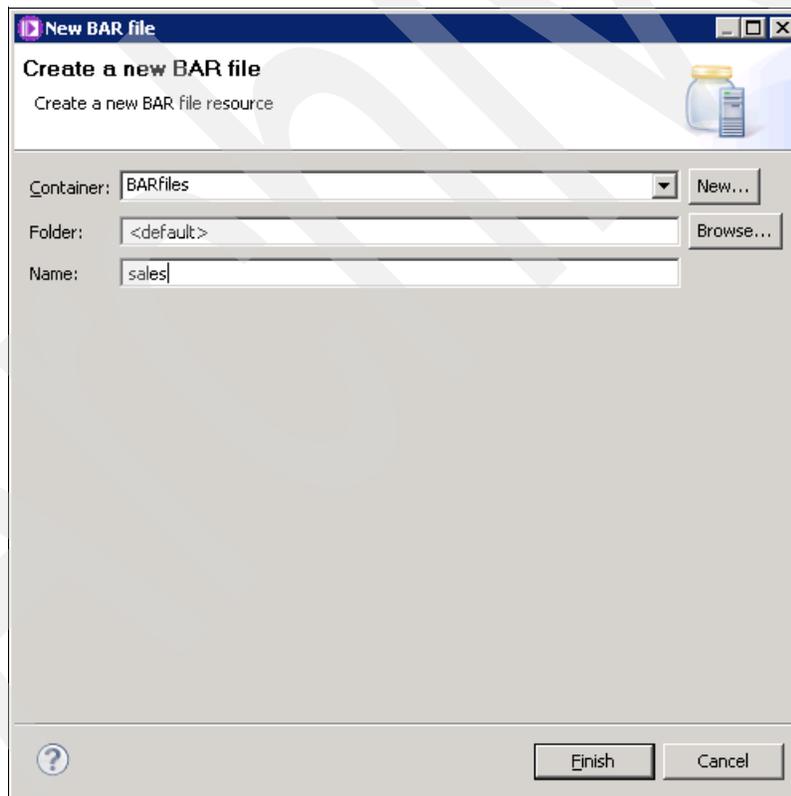


Figure 4-18 Create a BAR file for deployment

3. Click **Finish**.
4. Add your application to the BAR file by selecting the check boxes on the Prepare tab, as shown in Figure 4-19 on page 86.

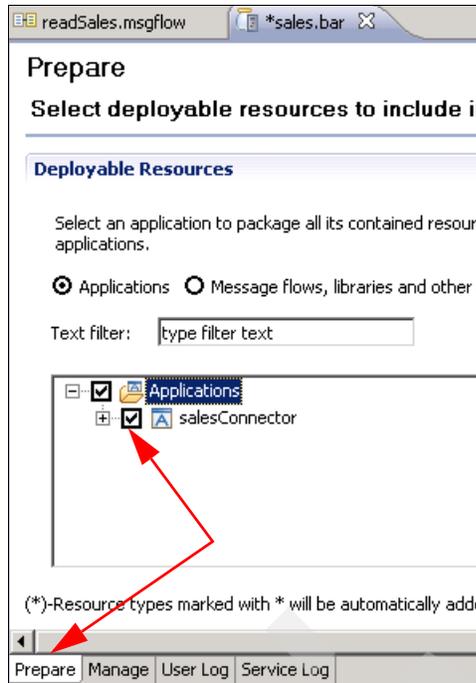


Figure 4-19 Add the application to the BAR file

5. Click the **Build and Save** button to build the BAR file, and then click **OK**.
6. Select the **Manage** tab. Expand your application, and select the readSales compiled message flow.

Name	Type
salesConnector	Application
readSales.cmf	Compiled message flow
readSales	
.NETCompute	

Figure 4-20 Select the message flow to edit the deployable properties

7. In the Properties view, set the SalesQueueName to a valid UNC path that names the correct queue. For the purposes of this scenario, use `.\private$\SALESOUT` to indicate a queue on the local machine in the private area named salesout.
8. Set the SalesWaitInterval and the TimeoutNotification node timeout intervals to 30 seconds, as shown in Figure 4-21.

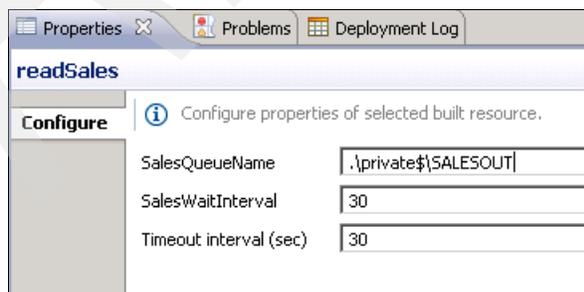


Figure 4-21 Edit the deployable properties

9. Save the BAR file, and deploy it by dragging it from the Broker Development view onto the execution group in your local broker.

If the BAR file does not deploy, you receive a message with the option to view the details of the error. The information gives you an idea of the problem. If the .NETCompute node is called out as the problem, review the settings in “Updating the .NETCompute node” on page 84. If so, update the settings and then rebuild and deploy the bar file.

Your queue was created in the Server Manager view, as indicated by the red arrow in Figure 4-22. (You might need to refresh the view).



Figure 4-22 New Queue visible in Server Manager

Note that MSMQ forced the name of the queue into lowercase.

10. You can now use the test application we built in 4.2.1, “Constructing a simple utility to read and write MSMQ messages” on page 69 to write messages to this queue.

Enter the name `.\private$\SALESOUT` into the queue name box in the MSMQUtil, and then enter the text `Hello Sales Message Flow` into the data box, and click the **Write Queue** button, as shown in Figure 4-23.

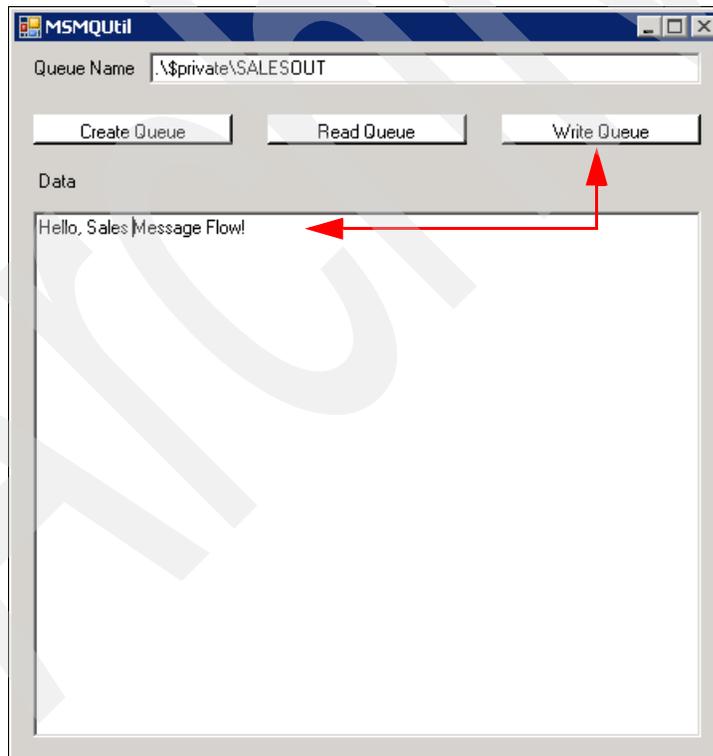


Figure 4-23 Write data to the queue

11. Use the WebSphere MQ Explorer to view the message on the output queue.

Expand the queue manager, and select **Queues**. Right-click the ESB.IN queue in the Content pane, choose **Browse Messages**. Select the first message in the viewer, and choose **Properties**. Figure 4-24 is the message that is displayed.

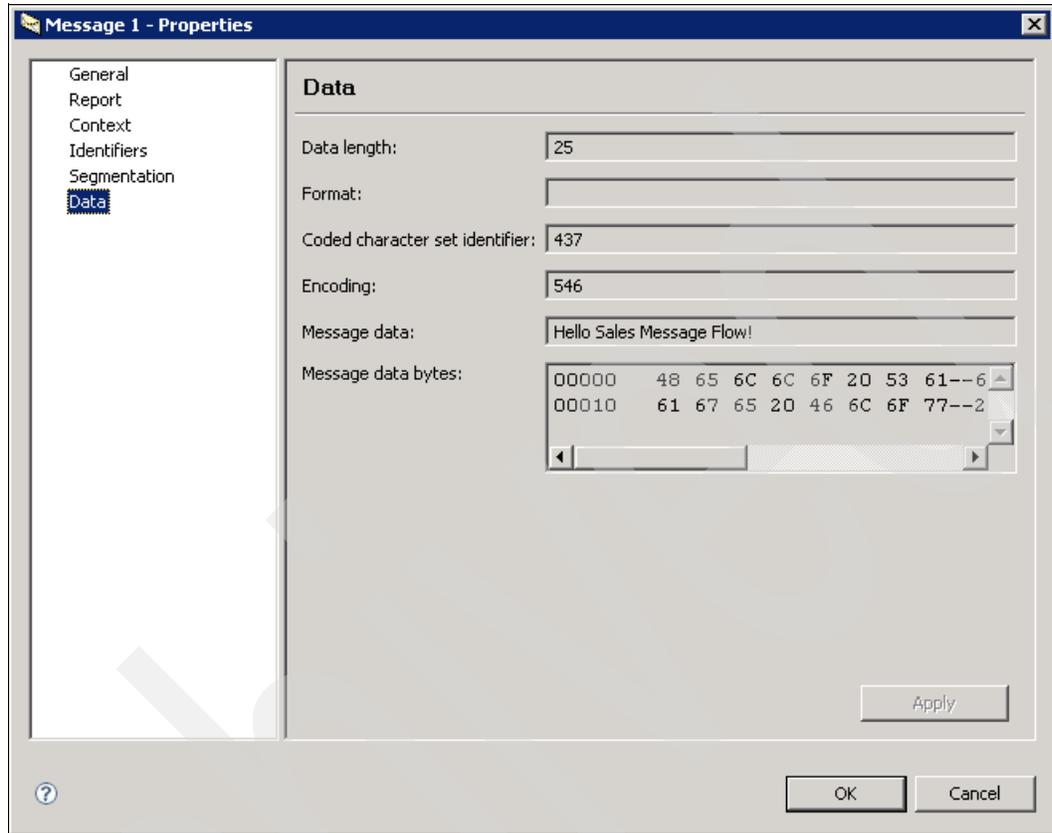


Figure 4-24 Verify the message arrived at WebSphere MQ

Again, we see that the message is plain text.

4.3 Mapping headers between MSMQ and WebSphere MQ

WebSphere MQ Messages have a number of important headers on them and a set of associated metadata that forms a logical context around the data in the message. When moving messages from a WebSphere MQ messaging network onto an MSMQ network or the other way around, it is important to understand the context and to carry the relevant parts of it through.

Table 4-1 shows the properties that are available on an MSMQ message that can be directly mapped to properties on a WebSphere MQ message.

Table 4-1 MQ message mapping properties

MSMQ	WebSphere MQ
Id	MessageID
CorrelationID	CorrelationID
Priority	Priority

MSMQ	WebSphere MQ
ResponseQueue	ReplyToQueue
SentTime	PutTime and PutDate
Recoverable	Persistence

Most of the remaining properties that exist on a given Microsoft Messaging Queue message can be stored without change into WebSphere MQ message properties.

4.4 Adding logging to your code

There are several kinds of code-level instrumentation that can be used with a .NET solution in WebSphere Message Broker. You want to ensure that you can provide adequate information about the normal operation of your flow. You also want to ensure that you can debug your code effectively in a production environment without requiring significant updates.

The .NET V4 framework provides the System.Diagnostics namespace and the Systems.Diagnostics.Trace, System.Diagnostics.TraceSource, and System.Diagnostics.Debug classes to allow you to easily instrument your own code.

You can continue to use these methods within your .NET code running inside a WebSphere Message Broker .NETCompute node. However, be aware of a couple of things:

- ▶ WebSphere Message Broker has a separate standard output console and standard error console for each execution group. In Windows, there is a single file for each execution group that contains both the standard output console and the standard error console. This file is kept in the broker's working storage area. The file is named `console.txt` and is in the following location:

```
C:\ProgramData\IBM\MQSI\components\<Broker Name>\<ExecutionGroup Unique ID>\
```

<Broker Name> is the name of the Broker, and *<ExecutionGroup Unique ID>* is a string that contains the unique identifier of each ExecutionGroup. See the topic for Directory Structures After Installation in the WebSphere Message Broker Information Center for further details:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp?%2Fcom.ibm.etools.mft.doc%2Fbh44510_.htm

- ▶ If you use the DefaultTraceListener and you enable it to write to Console.Out or you otherwise use standard print statements to write to Console.Out or Console.Error, this is the file your statements will end up in. This can lead to a disk full error or otherwise cause additional noise in this log file that will make it harder to debug other problems.

Important: Avoid using standard write statements and TraceListeners and other loggers that unilaterally output to the standard Console.Out and Console.Error.

The WebSphere Message Broker .NET APIs also provide the NBlog class to allow your .NETCompute node to output data into the standard areas that WebSphere Message Broker writes out information. WebSphere Message Broker flows typically output logging and tracing information into two locations: The Windows Event log (or other system log on non-windows systems) and an internal set of user trace files that can be read and processed using the broker commands.

The NLog class provides methods to allow your .NETCompute node to access both of these functions. You can write a custom TraceListener object to enable the System.Diagnostics frameworks to output to the WebSphere Message Broker standard locations. Example 4-11 shows a simple example.

Example 4-11 TraceListener that knows about broker trace facilities

```
using System.Diagnostics;
using System.Text;
using IBM.Broker.Plugin;

public class NLogListener : TraceListener
{
    //2951 is the default value, it is the number of the lowest message
    //in the block of reserved numbers for user messages.

    public int MessageNumber = 2951;
    System.Diagnostics.BooleanSwitch writeToEventLog = new
        System.Diagnostics.BooleanSwitch("WriteToEventLog",
            "Enables output to the event log");

    public override void Write(string message)
    {
        if (NLog.UserTraceActive)
        {
            NLog.LogUserTrace(MessageNumber, message);
        }
        if (writeToEventLog.Enabled)
        {
            NLog.LogEvent(MessageNumber, NBSeverity.Information, message);
        }
    }

    public override void WriteLine(string message)
    {
        if (NLog.UserTraceActive)
        {
            NLog.LogUserTrace(MessageNumber, message);
        }
        if (writeToEventLog.Enabled)
        {
            NLog.LogEvent(MessageNumber, NBSeverity.Information, message);
        }
    }
}
```

This listener creates a class variable to hold the message number for the error message being reported under the WebSphere Message Broker catalog. It initializes this to the default message number for the first error in the range of numbers set aside by the broker for user-created error messages.

It creates a second class variable that holds a Boolean Trace Switch object. If this trace switch is true, the broker outputs the provided trace message into the Event Viewer Application Log. This TraceListener also checks the NLog.UserTraceActive field to determine if the broker user tracing is enabled. If user tracing is enabled, this class also outputs the message to the user trace.

To use this TraceListener in your .NETCompute node, you can write code similar to the code in Example 4-12.

Example 4-12 Using a new TraceListener

```
// Create and add a new trace listener.  
NBLogListener brokerListener;  
brokerListener = new NBLogListener();  
Trace.Listeners.RemoveAt(0); //Remove the default listener  
Trace.Listeners.Add(brokerListener);  
//Log a message  
System.Diagnostics.Debug.WriteLine('Hello, Broker Log!');
```

Any of your other code that is already instrumented with the System.Diagnostics Tracing frameworks also outputs data to the Event Log or the user trace as configured.

You can control the state of the writeToEventLog Trace Switch by using an application configuration file. This file is deployed to the same location as your .NETCompute node's DLL. Visual Studio allows you to create a configuration file for your application and provides context-sensitive help for editing it.

Follow these steps:

1. Right-click our project folder in the Solution Explorer. Select **Add → New Item**.
2. Select **Application Configuration File** from the Visual C# items in the Installed Templates, as shown in Figure 4-25.

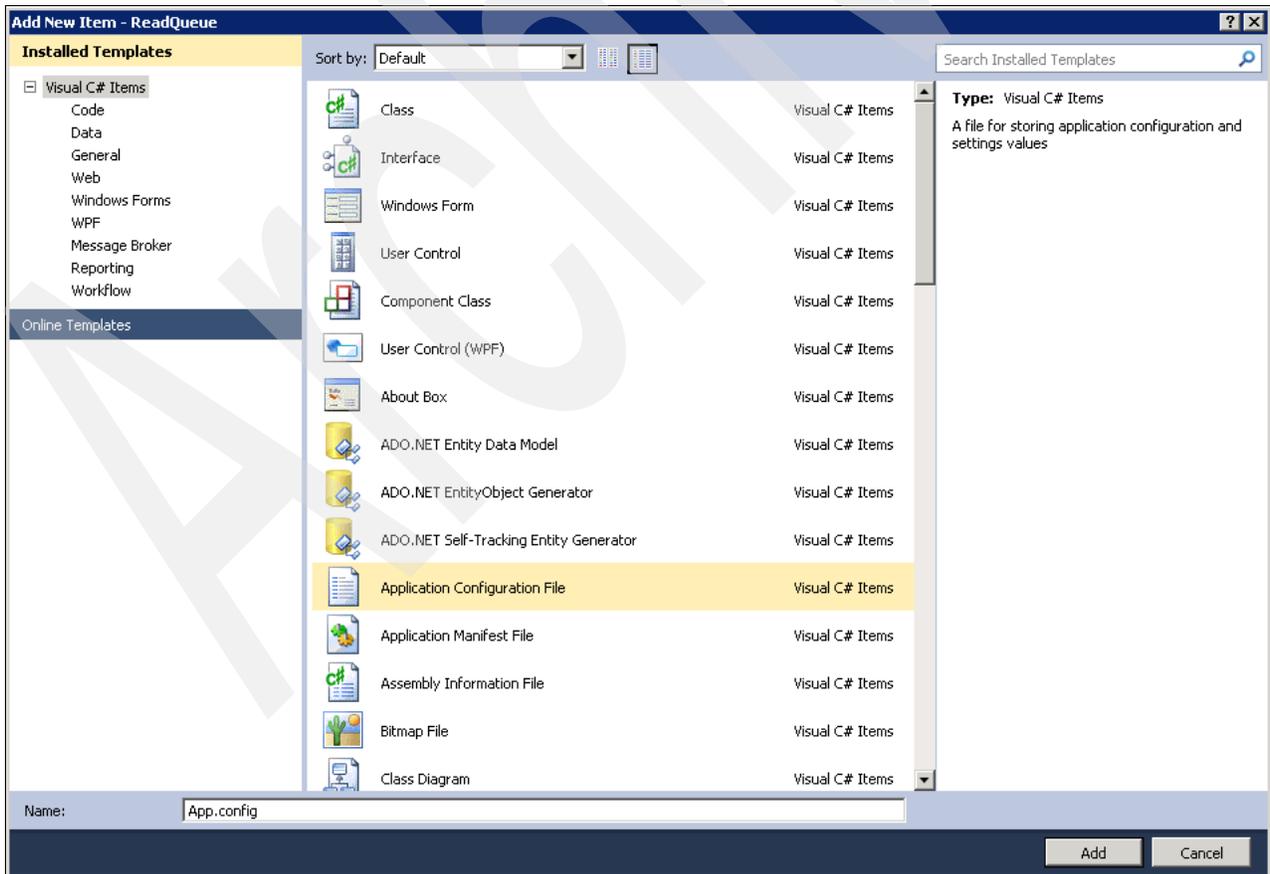


Figure 4-25 Creating a new XML file

3. Click **Add** to create a new XML file named `App.config`, as shown in Figure 4-26.

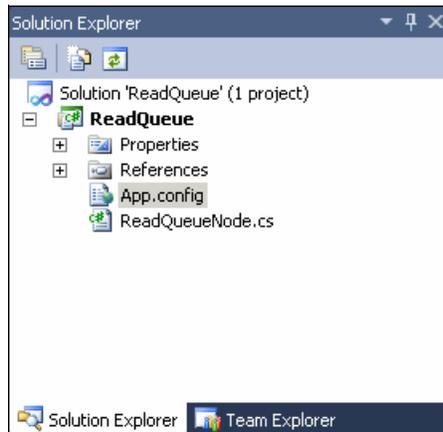


Figure 4-26 `App.config` file

Example 4-13 shows the file contents.

Example 4-13 Initial Application Configuration file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

4. Open and edit the file, as shown in bold in Example 4-14.

Example 4-14 Application Configuration File for `writeToEventLog`

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <switches>
      <add name="WriteToEventLog" value="0"></add>
    </switches>
  </system.diagnostics>
</configuration>
```

The name of the `<add>` tag is the name given to the switch constructor, not the name of the variable representing the switch. The value 0 on the switch indicates that the `BooleanSwitch` is false. Any other value indicates that the switch is true.

5. Rebuild the project. If your flow is deployed, rebuilding the project will cause a hot-deploy.
6. In the project folder, under the `debug/bin` folder, there is now a new `.config` file named after the DLL. Open this file with a text editor or double-click to open it in Visual Studio.
7. Change the value of the `WriteToEventLog` switch to 1. Save the file. A hot-deploy occurs and the switch is now active. Calls to the logger will output messages to the Event Log.

You can also use a `TextWriterTraceListener` class and output your logging information into an application-specific file in a known location that is not inside the broker's working storage area. You can configure this file location and enable or disable it in the application configuration file, rather than strictly in your code.

Sample code to add a `TextWriterTraceListener` is shown in Example 4-15 on page 93.

Example 4-15 TextWriterTraceListener code example

```
Stream logFile = File.Create("c:\\AppLogFile.txt");
TextWriterTraceListener tx= new
TextWriterTraceListener(logFile,"myListener");
Trace.Listeners.Add(tx);
```

You can also add and manage TextWriterTraceListeners in the Application Config, as shown in Example 4-16.

Example 4-16 TextWriterTraceListener configuration example

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="false" indentsize="4">
      <listeners>
        <add name="myTextListener"
            type="System.Diagnostics.TextWriterTraceListener"
            initializeData="c:\\textWriterOut.log" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Using full versus relative paths: The default working directory for your .NET code is the Broker's install directory, by default this is C:\Program Files\IBM\MQSI\8.0.0.0\bin. You must ensure that all filenames you use in your .NET code use full paths rather than relative paths. Otherwise, you run a strong risk of interfering with the proper functioning of the broker by filling up the disk or overwriting files.

You can also use any other kinds of TraceListener objects, provided they do not write into areas that can impact the correct running of the broker.

4.5 Handling exceptions

Message Broker V8 uses an exception-based programming model. Nodes that experience errors create an exception and pass the exception backwards through the flow by throwing it until it is caught by some part of the flow or it reaches the input node that started the flow.

The .NET API provided for the broker introduces sophisticated features for identifying and managing exceptions, both the ones that occur within the .NETCompute node or .NET method itself and exceptions that occur downstream of the .NETCompute node.

Exceptions that are created within a .NETCompute node are treated and caught and handled by the native methods in the language being used. Exceptions that are thrown by code inside a .NETCompute node and not caught within the node itself are automatically translated into Message Broker Exceptions stored in the ExceptionList tree.

If a .NETCompute node put a try/catch block around the propagate() call that transfers control of the message processing to the next node in the flow, any exceptions that occur downstream and are not caught get caught by this try/catch block. You can see an example of this in Example 4-9 on page 83. The propagate() is included in the try/catch block, so any errors that occurred downstream cause the catch code to be executed. In the example, the

catch block only tests to see if the error is a specific error that is related to reading the message from the Microsoft Messaging Queue.

If the MQOutput node in the readSales flow throws an exception, it is caught by the .NETCompute node. The Message Broker Exception List is parsed and automatically turned into an NBException object of the relevant type. The NBException class is a subclass of the Microsoft .NET standard Exception class, so it is handled exactly like all other exceptions in any .NET program.

A .NETCompute node can be aware of the failures of processing that occurs downstream of it and can make decisions about how to respond to the failure of a message flow as a whole or any results that are needed. A .NETCompute node can be wired to the Catch terminal of an input node and handle all of the exceptions that occur in the message flow, including rolling back any local transactions that are not controlled by the broker or freeing any resources that might be locked or affected by the flow.

You can modify the sample code for the readSales or writeSales code to catch all exceptions and then check to see if the caught exception is an NBException object. You can then use this object to identify the error that occurred and perform the necessary actions to handle it or rethrow it.

4.6 Sending messages to MSMQ

In this part of the scenario, we show you how to construct a message flow that gets messages (even using different parsers) from a WebSphere MQ queue and forward them onto a MSMQ queue.

4.6.1 Constructing the message flow

To create a new message flow in your application:

1. In the Message Broker Toolkit, create a new message flow in your application and name it writeSales.
2. Construct the message flow using the following steps:
 - a. Add an MQInput node from the WebSphere MQ on the Palette.
 - b. Add a .NETCompute node from the Transformation folder on the Palette.
 - c. Add a Trace node from the Construction folder on the Palette.
 - d. Wire the out terminal of the MQInput node to the in terminal of the .NETCompute node.
 - e. Wire the failure terminal of the .NETCompute node to the in terminal of the Trace node. We will use this just to capture any exceptions that occur.

Your flow now looks like Figure 4-27.

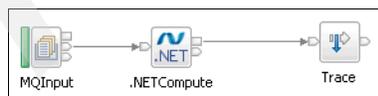


Figure 4-27 Message Flow to write to MSMQ

4.6.2 Configuring the message flow

The next step is to configure the relevant properties on the message flow, including adding the necessary user-defined properties:

1. Select the **MQ Input** node on the canvas, and specify a queue name of `ESB.OUT`, as shown in Figure 4-28.

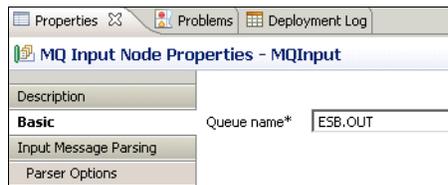


Figure 4-28 MQInput properties

2. Create a user-defined property to hold the name of the MSMQ queue. This value is read in the `OnInitialize()` method:
 - a. Select **User Defined Properties** at the bottom of the Message Flow Editor.
 - b. Select **Basic**, and click the **Add Property** button. Enter `SalesQueueName`, and select **String** as the type to define the name of the queue, as shown in Figure 4-29.

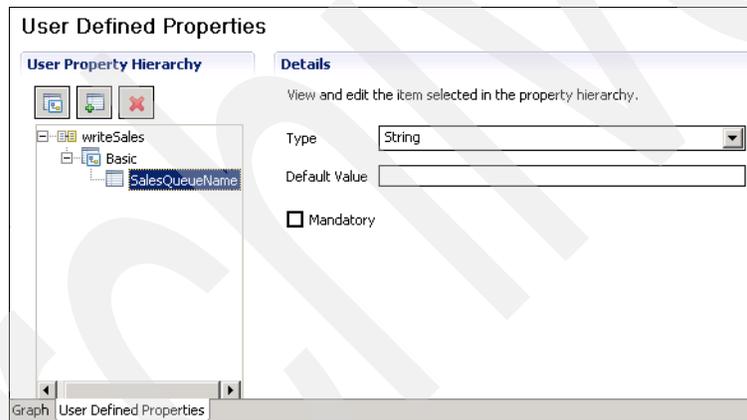


Figure 4-29 Add SalesQueueName as a user defined property

- c. Switch back to the Graph view.
3. Select the **Trace** node on the canvas, and configure the Trace node properties, as shown in Figure 4-30.

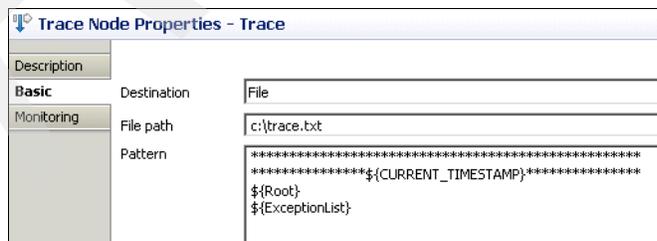


Figure 4-30 Trace node properties

4.6.3 Creating the .NET code

The next step is to create the VisualStudio project and the .NET code associated with the .NETCompute node:

1. Double-click the **.NETCompute** node in the message flow to launch Visual Studio.
2. Select **File** → **New** → **Project** to create a new project. Follow these steps:
 - a. In the Recent Templates column, select **Visual C# > Message Broker**.
 - b. In the list of templates, select **Project To Create A Message Broker Message**.
 - c. In the Name field, enter `WriteQueue`.
 - d. In the Location field, enter the path to store the project, for example, `c:\projects\VISudio`.
 - e. In the Solution field, `WriteQueue` is automatically populated.
 - f. Click **OK**. An editor with the new code template opens.
3. Add a reference in the project to the `System.Messaging` component.

In the Solution Explorer, right-click **References**, and select **Add Reference**. In the Add Reference window, shown in Figure 4-7 on page 73, select the **.NET** tab, and then select the **System Messaging** component. Click **OK**.
4. In the `CreateNode.cs` code, add the `System.Messaging` namespace and the `System.Globalization` namespace to the code using statements shown in Example 4-17 to the list of using statements at the top of the file.

Example 4-17 Using statements

```
using System.Messaging;  
using System.Xml;  
using System.IO;
```

5. Save the design by pressing `ctrl-S`.

4.6.4 Writing the code

To create the code for the .NETCompute node:

1. The class that is created has a default name of `CreateNode`. We change the name of the class to `WriteQueueNode` instead. In the Solution Explorer, right-click **CreateNode.cs** and select **Rename**. Enter `WriteQueueNode.cs`, and press `Enter`.

When prompted to perform a rename of all references, click **Yes**.

2. Create the class variables.

The `salesQueue` variable is the MSMQ object representing the queue, and the `salesQName` will hold the actual name of the queue. Add the code shown in bold in Example 4-18 to the `WriteQueueNode` class.

Example 4-18 WriteQueueNode method

```
public class WriteQueueNode : NBComputeNode  
{  
    MessageQueue salesQueue;  
    String salesQName;
```

3. Create a constructor for the node, as shown in Example 4-19 on page 97, directly below the variables.

Example 4-19 OnInitialize method of the node class

```
public override void OnInitialize()
{
    base.OnInitialize();

    salesQName = GetUserDefinedPropertyAsString("SalesQueueName");
    if (!MessageQueue.Exists(salesQName))
    {
        salesQueue = MessageQueue.Create(salesQName);
        salesQueue.SetPermissions("Users",
MessageQueueAccessRights.FullControl);
    }
    else
    {
        salesQueue = new MessageQueue(salesQName);
    }
}
```

This code reads the user-defined properties that you created on the message flow, and then uses the name property to create and open the MSMQ queue to be used during the main flow execution.

If the broker code created the queue, it also explicitly grants all permissions to all members of the Windows group Users. You can still manually create the MSMQ queue before the broker flow tries to access it. If you do, ensure that you set the needed permissions. The desktop application in 4.2.1, “Constructing a simple utility to read and write MSMQ messages” on page 69 does not set any permissions on the queue, so it is automatically private to the user running the application. You can modify the application to add the same code to set all permissions, or you can use the Microsoft Server Manager console to create the queue and set the appropriate permissions.

For the purpose of this scenario, the broker creates the queue, giving all users full control.

4. Having created the constructor, you must create the destructor to make sure that the queue is closed when we are finished, as shown in Example 4-20.

Example 4-20 OnDelete method of the node class

```
public override void OnDelete()
{
    salesQueue.Close();
    base.OnDelete();
}
```

5. Construct the method you will use to write the message. A skeleton of the Evaluate method is already created for you. The Evaluate method handles the messages that arrive at the .NETCompute’s Input terminal and allows you to work with the data. When you are finished, send it to the MSMQ queue.

In this step, we show two alternative code options. The first one shows how to receive a BLOB message and send it to a MSMQ queue. This option is useful if you want to use WebSphere Message Broker as a router with no parsing, simply focusing on performance.

The second option shows how to code to work with XMLNSC parsed messages. This option is useful when you need to handle the data.

In both code options, we use a basic try/catch block to handle the exceptions.

If you choose to work within the BLOB domain, remember to configure the Input Message Parsing properties of the MQInput node. Figure 4-31 shows how to configure it to parse messages using the BLOB parser.

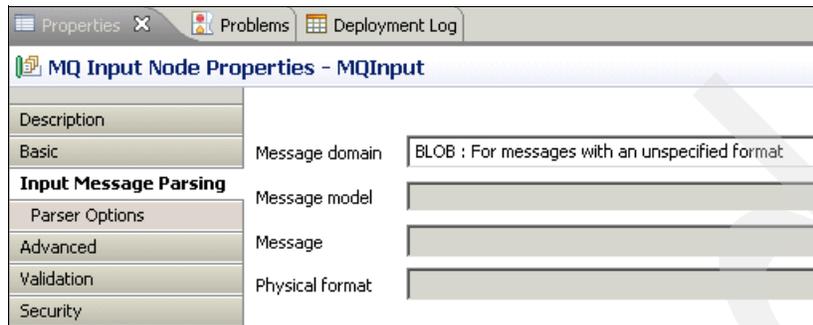


Figure 4-31 Configuring MQInput to read messages with the BLOB parser

Select the option to use and add the code in the UserCode region of WriteQueueNode.cs. Example 4-21 shows the code for BLOB messages.

Example 4-21 Working with BLOB messages

```
#region UserCode
// Add user code in this region to create a new output message
// Create the output message structure for a BLOB message.
try
{
    NBElement blobRoot = inputRoot["BLOB"]["BLOB"];
    Message myMessage = new Message();

    // Saving the original CCSID and Encoding from the MQ Message
    int oCcsid = (int)inputRoot["MQMD"]["CodedCharSetId"]; // Original CCSID
    int oEncoding = (int)inputRoot["MQMD"]["Encoding"]; // Original Encoding

    myMessage.BodyStream = new MemoryStream(
        inputRoot[NBParsers.BLOB.ParserName].AsBitStream(oCcsid,oEncoding));

    salesQueue.Send(myMessage);
}
catch (System.Messaging.MessageQueueException e)
{
    throw e;
}
}
#endregion UserCode
```

Alternatively, if you choose to work with the XMLNSC domain, remember to configure the Input Message Parsing properties of the MQInput node. Figure 4-32 on page 99 shows how to configure it.

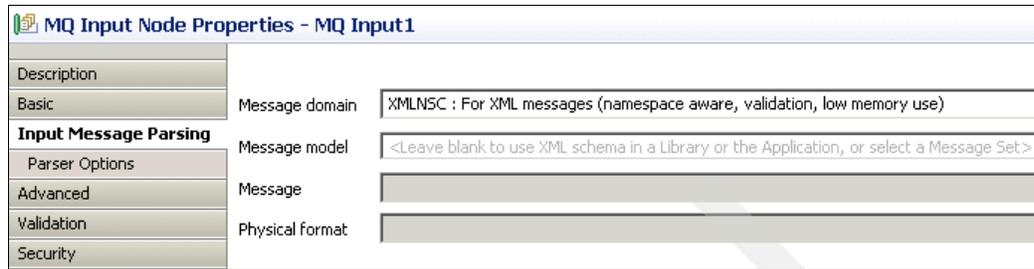


Figure 4-32 Configuring MQInput to read messages with the XMLNSC parser

Example 4-22 shows the code for XMLNSC messages.

Example 4-22 Working with XMLNSC messages

```
#region UserCode
// Add user code in this region to create a new output message
// Create the output message structure for a XML message.
try
{
    NBElement xmlRoot = inputRoot["XMLNSC"];
    XmlDocument myMessage = new XmlDocument();

    // Saving the original CCSID from the MQ Message
    int oCcsid = (int)inputRoot["MQMD"]["CodedCharSetId"];

    // Loading the xml
    myMessage.LoadXml(Encoding.UTF8.GetString(
        inputRoot[NBParsers.XMLNSC.ParserName].AsStream(0,
oCcsid)));

    salesQueue.Send(myMessage);
}
catch (System.Messaging.MessageQueueException e)
{
    throw e;
}
}
#endregion UserCode
```

6. Save and close the WriteQueueNode.cs file.
7. Select **Build** → **Build Solution**.

Updating the .NETCompute node

Now that you have the code, you must update the message flow to use it:

1. In the Message Broker Toolkit, click the **.NETCompute** node.
2. In the Properties view, select the **Basic** tab:
 - a. In the Assembly name field, browse to the following location and select the WriteQueue.dll file:

C:\projects\VIStudio\WriteQueue\WriteQueue\bin\Debug\WriteQueue.dll

Click **Open**.
 - b. Enter WriteQueueNode as the class name.

3. Select the **Advanced** tab in the Properties view, and enter WriteSales in the AppDomain name. This process ensures that this .NETCompute node's DLL is loaded in another .NET application domain and does not interfere with other flows in the same Broker Application.
4. Save the message flow by pressing Ctrl-S.

Testing the message flow

To test the message flow:

1. Update the BAR file to deploy your message flow application. Open the BAR file created in "Testing the message flow" on page 85 in your Broker Application Development perspective, as shown in Figure 4-33. On the Prepare tab, both message flows are listed.

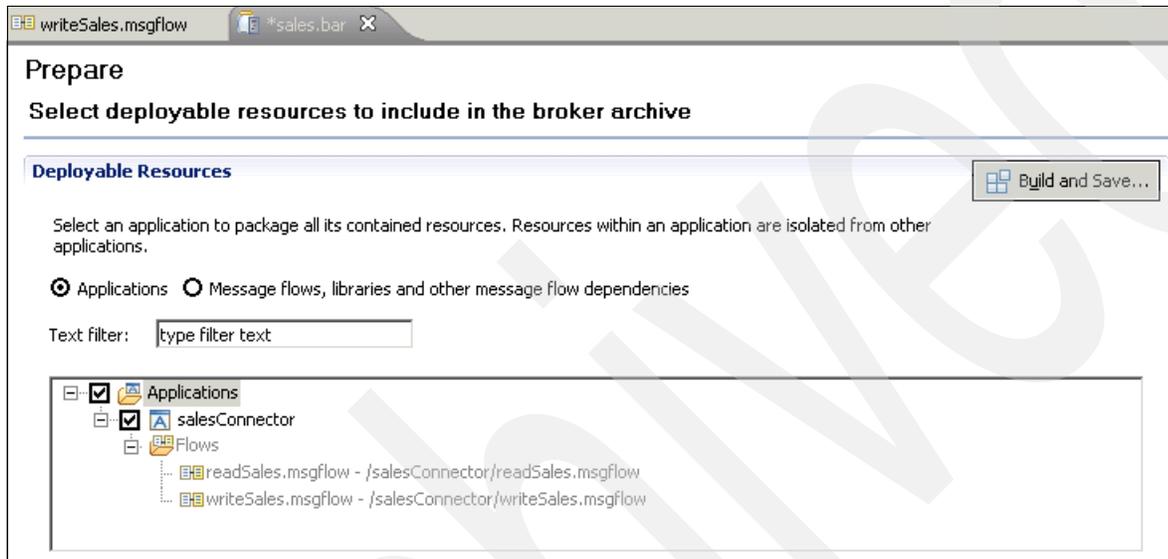


Figure 4-33 Updating the sales.bar File

Click **Build and Save**.

2. Switch to the **Manage** tab. Expand your application, and select the **writeSales** compiled message flow, as shown in Figure 4-34.

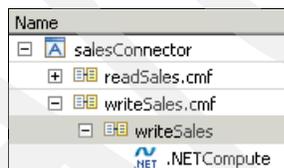


Figure 4-34 Select the message flow to edit the deployable properties

3. In the Properties view, set the SalesQueueName to `.\private$\SALESIN` to indicate an MSMQ queue on the local machine in the private area named "salesin", as shown in Figure 4-35 on page 101.



Figure 4-35 Edit the deployable properties

4. Save the BAR file and redeploy it by dragging it onto the execution group in your local broker.

If the BAR file does not deploy, a message is displayed with the option to view the details of the error. The information gives you an idea of the problem. If the .NETCompute node is called out as the problem, review the settings in “Updating the .NETCompute node” on page 99. If so, update the settings and then rebuild and deploy the bar file.

You can see that your queue is created in the Server Manager view, as shown in Figure 4-36. (You might need to refresh the view.)



Figure 4-36 New queue visible in Server Manager

Note that MSMQ forced the name of the queue into lowercase.

5. Use the WebSphere MQ Explorer to put a message into the ESB.OUT queue.

In the WebSphere MQ Explorer - Navigator view, expand the queue manager, and select **Queues**. In the Content pane, right-click **ESB.OUT** and select **Put Test Message**.

Example 4-23 shows a sample message that works for both domains. Enter the message, and click **Put message**.

Example 4-23 Sample queue message

```
<salesorder><product number="12345"><price>150</price><description>A beautiful  
new leather wallet</description></product></salesorder>
```

Note: MSMQ messages are limited to 4 Mb of data and do not support message segmentation, unlike IBM WebSphere MQ. For messages larger than 4MB, you must split the message into smaller messages using your own coding techniques. Make sure that both sender and receiver know about the split mechanism.

6. Use the Server Manager to view the message on the queue. Right-click the **salesin** queue, and select **Refresh**, as shown in Figure 4-37 on page 102.

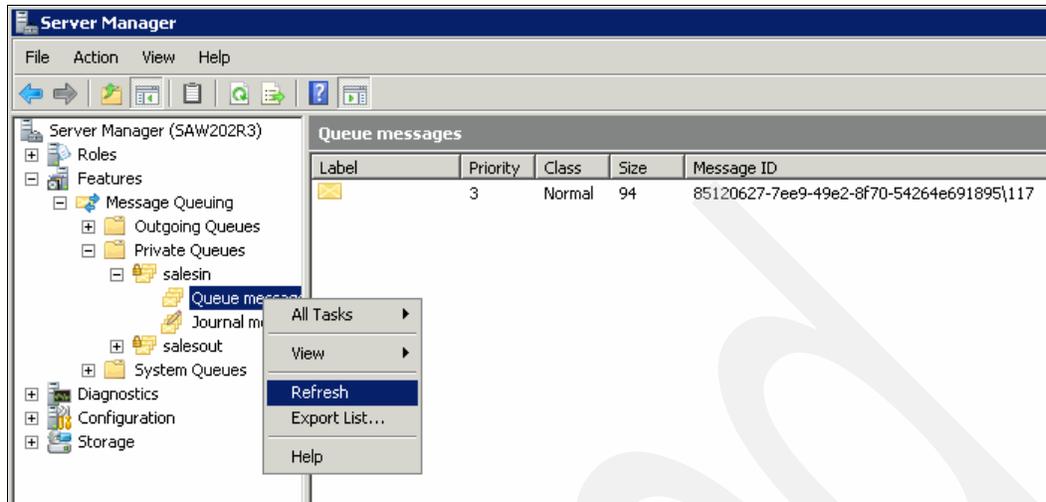


Figure 4-37 Checking the message in the salesin queue

If you do not see a message on the queue, make sure that your input message parsing property on the MQ Input node matches the code type that you selected for your .NETCompute node (BLOB or XMLNSC). Also make sure that you used valid data, as shown in Example 4-23 on page 101.

7. Double-click the message in the viewer to check the body of the message, as shown in Figure 4-38.

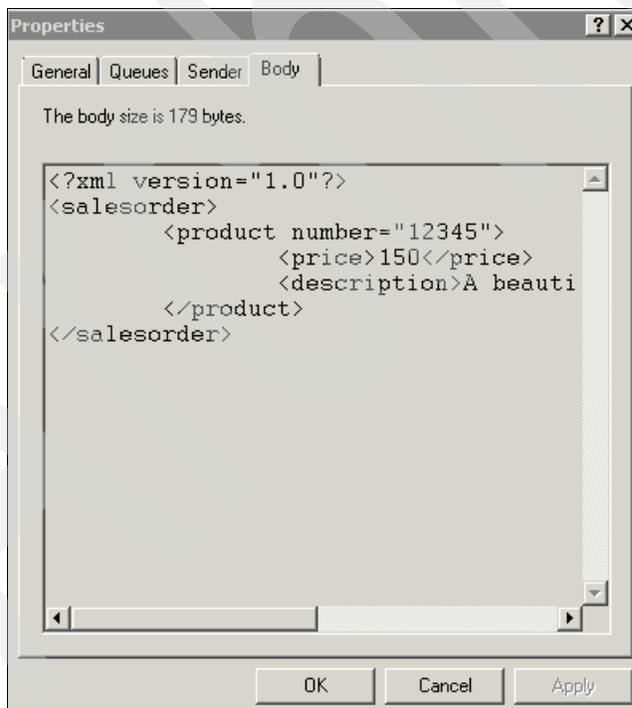


Figure 4-38 Checking the Body of the message

4.7 Distributed transactional coordination

A *business transaction* is a set of tasks that either succeed or fail as a unit. Although tasks that are associated with a transaction are operationally independent, they share a common intent as a unit. By performing only a subset of these operations, the system can compromise the overall intent of the transaction.

To guarantee the “all or nothing” aspect of transactions, the IT systems must handle participating operations in a transactional context within the overall scope. If a single participant of the transaction fails, all changes to data within the scope of the transaction are rolled back to a previous stage. All participating operations must guarantee that any change to data is permanent and will persist despite system crashes or other unanticipated events.

Although updates made using the .NET languages do not participate in WebSphere Message Broker transactions, it is possible to use transactions with MSMQ manually. This means that the developer controls the transaction and must be aware of the constraints. We will show how to do a manual transaction by performing some changes to our message flow.

4.7.1 Manual transaction sample

Our example uses the writeSales message flow to illustrate how to commit/rollback a transaction:

1. Open the writeSales message flow, and add another .NetCompute node after the first .NETCompute node, and rename it to Commit.
2. Wire the out terminal of the .NETCompute node to the input terminal of the Commit node. Wire the failure terminal of the Commit node to the Trace node. Your flow now looks like Figure 4-39.

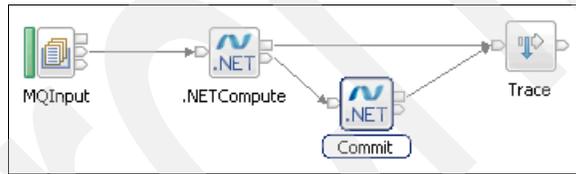


Figure 4-39 Changing the flow to add manual transaction

3. Declare a static `MessageQueueTransaction` variable inside the `WriteQueueNode` class, as shown in bold in Example 4-24.

Example 4-24 Defining the static `MessageQueueTransaction` variable

```
public class WriteQueueNode : NBComputeNode
{
    MessageQueue salesQueue;
    String salesQName;
    public static MessageQueueTransaction tx;

    public override void Evaluate(NBMessageAssembly inputAssembly)
    {
        (...)
    }
}
```

4. To determine that our MSMQ queue supports transactions, change the constructor of the node, adding a new parameter with value true in the Create method, as shown in bold in Example 4-25.

Example 4-25 OnInitialize method of the node class

```
public override void OnInitialize()
{
    base.OnInitialize();

    salesQName = GetUserDefinedPropertyAsString("SalesQueueName");
    if (!MessageQueue.Exists(salesQName))
    {
        salesQueue = MessageQueue.Create(salesQName, true);
        salesQueue.SetPermissions("Users",
MessageQueueAccessRights.FullControl);
    }
    else
    {
        salesQueue = new MessageQueue(salesQName);
    }
}
```

5. Add code to begin the transaction before you send the message. The code is highlighted in Example 4-26.

Example 4-26 Doing manual transaction between WebSphere Message Broker and MSMQ

```
#region UserCode
// Add user code in this region to create a new output message

try
{
    NBElement xmlRoot = inputRoot["XMLNSC"];
    XmlDocument myMessage = new XmlDocument();

    int oCcsid = (int)inputRoot["MQMD"]["CodedCharSetId"];
    myMessage.LoadXml(Encoding.UTF8.GetString(
        inputRoot[NBParsers.XMLNSC.ParserName].AsStream(0,
oCcsid)));

    tx = new MessageQueueTransaction();
    tx.Begin();

    salesQueue.Send(myMessage, tx);
}
catch (System.Messaging.MessageQueueException e)
{
    throw e;
}

#endregion UserCode
```

6. Add a new class to the Project:
 - a. Select **Project** → **Add Class**.
 - b. Select **Message Broker** → **Class to create a Message Broker**.

- c. Name the new class `CommitNode`.
- d. Click **Add**.
7. In the Solution Explorer, right-click **CreateNode.cs** and select **Rename**. Enter `CommitNode.cs` and press Enter.
8. Enter the code shown in Example 4-28 in the `UserCode` region to test committing the transaction.

Example 4-27 Defining the static `MessageQueueTransaction` variable

```
#region UserCode
// Add user code in this region to modify the message
// Manually commits the transaction
WriteQueueNode.tx.Commit();
#endregion UserCode
```

9. Alternatively, you can test the rollback scenario. Enter the code in Example 4-28 to rollback the transaction.

Example 4-28 Defining the static `MessageQueueTransaction` variable

```
#region UserCode
// Add user code in this region to modify the message
// Manually rollbacks the transaction
WriteQueueNode.tx.Abort();
#endregion UserCode
```

10. Save and close the `CommitNode.cs` file.
11. Select **Build** → **Build Solution**.
12. In the Message Broker Toolkit, select the **Commit .NETCompute node**, and on the basics tab, complete the assembly name field with a pointer to the `WriteQueue.dll`.

4.7.2 Testing the message flow

At this point, you can execute tests committing or rolling back your messages. You can check the Computer Management Console in Windows to see if the messages are being committed or rolled back:

1. Update the BAR file to deploy your message flow application. Open it in your Broker Application Development perspective. See Figure 4-40 on page 106. Click **Build and Save**.

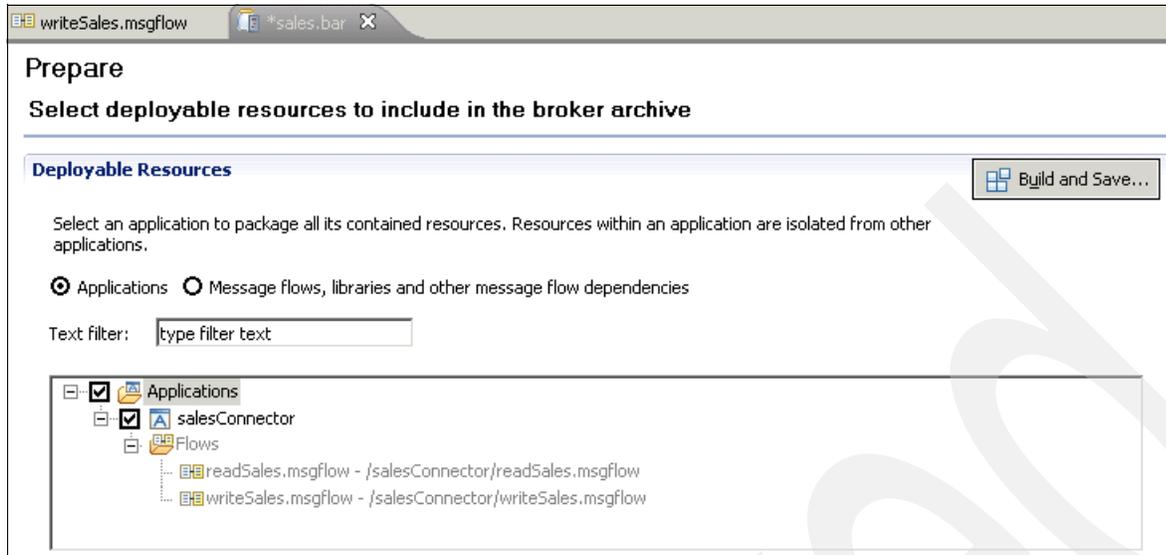


Figure 4-40 Updating the sales.bar File

2. Switch to the Manage tab. Expand your application, and select the writeSales compiled message flow. See Figure 4-41.



Figure 4-41 Select the message flow to edit the deployable properties

3. In the Properties view, set the SalesQueueName to a valid UNC path that names the correct queue. For the purposes of this scenario, use `.\private$\SALESIN.TX` to indicate a transactional queue on the local machine in the private area named salesin.tx. See Figure 4-42.



Figure 4-42 Edit the deployable properties

4. Save the bar file, and deploy it by dragging it onto the execution group in your local broker.
5. You can see that your queue is created in the Server Manager view, as shown in Figure 4-43 on page 107. (You might need to refresh the view.)

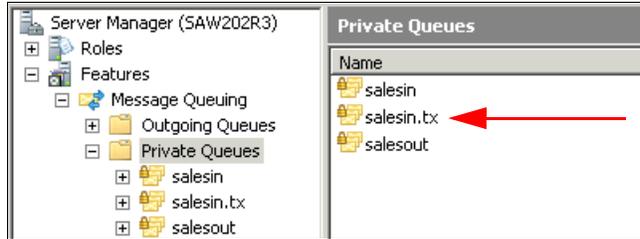


Figure 4-43 New Queue visible in Server Manager

- You can now put a message into the ESB.OUT queue. Example 4-29 shows a sample message.

Example 4-29 Sample queue message

```
<salesorder><product number="12345"><price>150</price><description>A beautiful
new leather wallet</description></product></salesorder>
```

- Use the Server Manager to view the message on the queue. Right-click the **salesin.tx** queue and select **Refresh**, as shown in Figure 4-44.

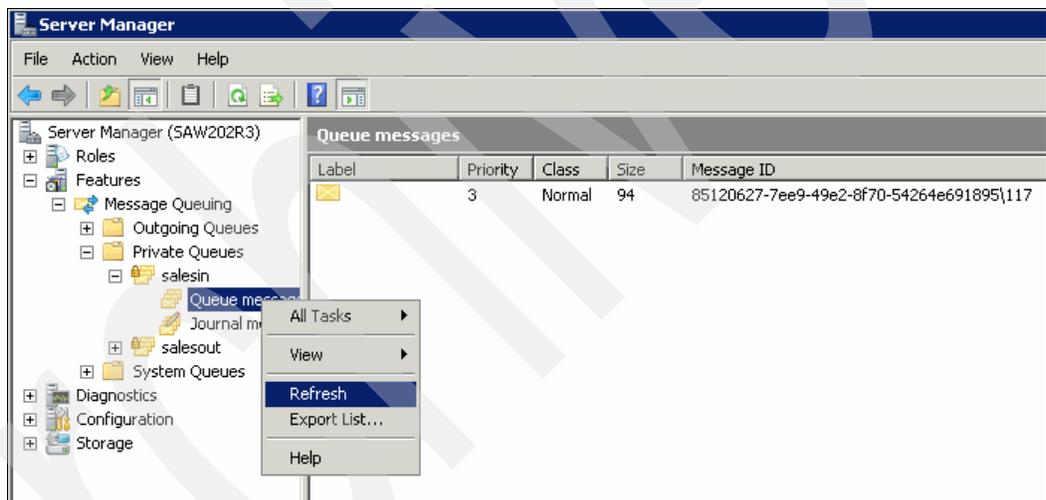


Figure 4-44 Checking the message in the salesin queue

- Double-click the message in the viewer to check the message body. See Figure 4-45 on page 108.



Figure 4-45 Checking the Body of the message

If you see the message, the commit worked.

4.8 Conversions

Messages coming from Z/OS, IBM AIX®, Linux, Windows, or any other platform that is supported by IBM WebSphere MQ or coming from other countries with code pages that are specific to national languages or operating systems can require encoding and CCSID conversions. Basically, CCSID conversion is used to convert character data and control characters, while encoding conversion does it for numeric data. You can find more information about CCSID and encoding conversions in the WebSphere MQ documentation at:

<http://www-01.ibm.com/support/docview.wss?uid=swg27005729>



Scenario: Calling Microsoft Dynamics CRM from a message flow

This chapter explores the integration capabilities of message flows and Microsoft Dynamics CRM. This scenario is based on a WebSphere Message Broker message flow that connects to an SAP system, a database, and Microsoft Dynamics CRM Online.

This chapter contains the following topics:

- ▶ Scenario overview
- ▶ Creating the message flow
- ▶ Deploying the message flow
- ▶ Testing the message flow
- ▶ Troubleshooting tips

Additional materials: The WebSphere Message Broker project interchange file and the .NET class code for this scenario can be downloaded from the IBM Redbooks publication web site. See Appendix A, “Additional material” on page 485 for more information.

and from the SAP System. The adapter supports SAP integration interfaces, such as BAPI and ALE, and SAP remote function call (RFC) function modules.

5.1.2 Microsoft Dynamics CRM Online

Microsoft Dynamics CRM 2011/Online is customer relationship management software from Microsoft for managing interactions, orders, bills, audits, and satisfaction surveys with the sales, clients, and customers. It offers a new level of productivity to sales, services, and marketing organizations by providing familiar, intelligent, and connected experiences with rich integration features.

Microsoft Dynamics CRM offers several programming models. In this chapter, we demonstrate the WebSphere Message Broker connectivity to Microsoft Dynamics CRM Online version by exploring its late-bound programming model.

5.1.3 Prerequisites for this scenario

Before reading this chapter, you must be familiar with the following products:

- ▶ IBM WebSphere Message Broker
- ▶ IBM WebSphere MQ
- ▶ SAP Software
- ▶ Microsoft Dynamics CRM

To follow the instructions for the technical examples described in this chapter, you must have a development environment set up and running that includes:

- ▶ WebSphere Message Broker Runtime and WebSphere Message Broker Toolkit V8.0
- ▶ WebSphere MQ V7.0.1
- ▶ Access to an SAP environment
- ▶ Access to Microsoft Dynamics CRM Online
- ▶ Microsoft CRM SDK
<http://www.microsoft.com/en-us/download/details.aspx?id=24004>
- ▶ Windows Identity Foundation
<http://www.microsoft.com/download/en/details.aspx?id=17331>
- ▶ Microsoft Visual Studio 2010

5.2 Creating the message flow

The message flow in this scenario handles customer requests to subscribe to a service with the online store.

If the customer is an existing customer, the message flow verifies the customer records from SAP and the database and create a new CRM Customer record by tagging-in the account numbers from SAP and the database.

If the customer is a new customer, the message flow creates a new SAP customer record. The message flow then verifies the customer record in the database and creates a new CRM Customer record by tagging-in the account numbers from SAP and the database.

Integration scenarios and options vary based on business needs, for example, this scenario explores connectivity to Microsoft Dynamics CRM Online using a late-bound programming model. You can also use an early-bound or other programming models. WebSphere MQ is used as the transport between WebSphere MQ and Microsoft Dynamics CRM. Based on the front-end architecture and connectivity, you might consider designing the front-end application to send SOAP requests or HTTP requests instead of WebSphere MQ messages. It is also possible to use Microsoft Message Queuing (MSMQ) as the messaging engine for this flow. For more information about calling WebSphere Message Broker integration with MSMQ, see Chapter 4, “Scenario: Bridging WebSphere MQ and Microsoft Message Queuing” on page 67.

Figure 5-2 shows the WebSphere Message Broker flow for this scenario.

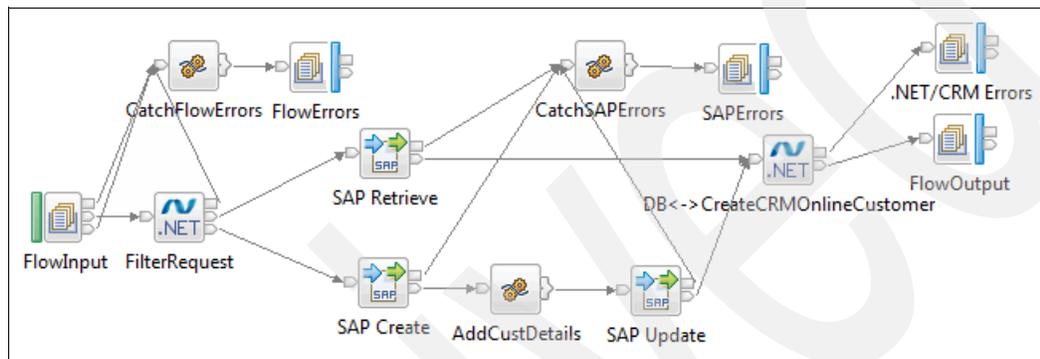


Figure 5-2 WebSphere Message Broker flow

The following list describes the behavior of each node.

1. WebSphere MQ messages enter the flow through the FlowInput node.
2. The FilterRequest node determines if the request is from an existing or new customer. If it is from a new customer the message is sent to the SAP Create node. If it is from an existing customer, the message is sent to the SAP Retrieve node.
3. The SAP Retrieve node retrieves the Customer records from the SAP system based on a customer number search.
4. The SAP Create node creates a new customer record on the SAP system based on a reference customer.
5. The SAP Update node updates the newly created customer record with field values.
6. The AddCustDetails node constructs the SAP request to update the customer details.
7. The CatchSAPErrors node processes SAP errors and creates a meaningful exception message.
8. The CatchFlowErrors node processes flow errors and creates a meaningful exception message.
9. The FlowErrors node stores the error messages that occur during flow execution.
10. The SAPErrors node stores the error messages that occur during SAP system interactions.
11. The .NET/CRM Errors node stores the error messages that occur during database and CRM interactions.
12. The FlowOutput node stores the message flow execution output.
13. The DB CreateCRMOnlineCustomer node interacts with the database and creates the customer on CRMOnline.

Additionally, Figure 5-2 on page 112 shows the following flow details:

1. When an MQ message with data conforming to XMLNSC format is placed on the input queue, the FlowInput node picks up the message and passes it to the .NETCompute FilterRequest node.
2. The FilterRequest node checks for the type of request (New Customer / Existing Customer) and routes the message accordingly to the SAP Retrieve or SAP Create nodes.
3. If the message is sent to the SAP Retrieve node:
 - a. The SAP Retrieve node uses the customer number in the incoming message and constructs a SapBapiCustomerGetdetail1 request. It issues a BAPI call and gets the customer details and passes them to the .NETCompute node DB<->CreateCRMOnlineCustomer.
 - b. The DB<->CreateCRMOnlineCustomer node checks if the customer exists in the database by performing a search based on the customer email ID. If the customer exists, it retrieves the DB account number and then creates a new CRM online customer with a reference to SAP and the database account numbers.
 - c. The DB<->CreateCRMOnlineCustomer node sends an output message of format XMLNC to the FlowOutput node which places the new message on the output queue. The CRM online record identifier is included in the message.
4. If the message is sent to the SAP Create node:
 - a. The SAP Create node uses the data from the incoming message to construct a SapBapiCustomerCreatefromdata1 request. The node issues a BAPI call, gets the customer details, and passes that information to the Compute Node AddCustDetails.
 - b. AddCustDetails constructs a SapBapiCustomerChangefromdata1 request with the new customer details and sends the request to the SAP Update node.
 - c. The SAP Update node uses the information coming in from the Compute node and issues a BAPI call to update the Customer details, then passes the returned values to the .NETCompute node DB<->CreateCRMOnlineCustomer.
 - d. The DB<->CreateCRMOnlineCustomer node checks if the given customer exists in the database by performing a search based on the customer email ID. If the customer exists, it retrieves the database account number and then creates a new CRM online customer with a reference to SAP and the database account numbers.

In the following sections, we implement these steps by creating the message flow from the beginning and exploring the techniques used to integrate with SAP and Microsoft Dynamics CRM. We also test our message flow to see if it can retrieve/create data from SAP to Microsoft Dynamics CRM Online.

This section tells you how to create this message flow using the following steps:

- ▶ Setting up the environment
- ▶ Creating and connecting the nodes
- ▶ Configuring the node properties
- ▶ Broker configuration for SAP request nodes
- ▶ Writing the code for the SAP nodes
- ▶ Coding the ESQL for the Compute nodes
- ▶ Coding the Filter Request .NETCompute node
- ▶ Writing the code for the database operations
- ▶ Writing the code for accessing Microsoft Dynamics CRM Online
- ▶ Writing the code for the .NETCompute node (Create) - DB<->CreateCRMOnlineCustomer

5.2.1 Setting up the environment

Use this section to help you create the WebSphere MQ, WebSphere Message Broker, SQL Server, and SAP environments for messaging.

WebSphere Message Broker

Use the Message Broker Toolkit or the WebSphere Message Broker Explorer to create a new broker instance called SAMPLEBROKER. Specify SAMPLEQM as the queue manager.

WebSphere MQ

Use the WebSphere MQ Explorer to create the following local queues on SAMPLEQM.

- ▶ INPUT_MSGS_Q
- ▶ FLOW_ERROR_MSGS
- ▶ SAP_ERROR_MSGS
- ▶ DotNet_CRM_ERROR_MSGS
- ▶ OUTPUT_MSGS_Q

SQL Server database

Working with SQL Server: This section assumes that you have SQL Server running on the same system as Visual Studio. Instructions for enabling SQL Server database are in 8.3.1, “Creating the database” on page 396.

This scenario uses a Microsoft SQL Server 2008 database called ITSOCRM that has the following two tables.

Customers table with the columns shown in Table 5-1.

Table 5-1 Customers table

Column	Datatype
customerid	uniqueidentifier, primary key, DEFAULT NEWSEQUENTIALID()
firstname	varchar(100)
lastname	varchar(100)
address1	varchar(100)
address2	varchar(100)
city	varchar(50)
state	varchar(50)
country	varchar(50)
postal code	varchar(10)
email	varchar(250)
phone number	integer(10)
modified date	DateTime

Accounts table with the columns in Table 5-2.

Table 5-2 Accounts table

Column	Datatype
account number	UNIQUEIDENTIFIER DEFAULT NEWSEQUENTIALID() PRIMARY KEY
customerid	UNIQUEIDENTIFIER DEFAULT NEWSEQUENTIALID() PRIMARY KEY
email	varchar(250)
modified date	DateTime

SAP environment

Obtain the SAP Java connector files (sapjco3.jar, sapjco3.dll) from the SAP system or the SAP administrator and copy them to your local drive. In our scenario, we used C:\SAP_JCO as our location.

5.2.2 Creating and connecting the nodes

Use the following procedure to construct the skeleton of the message flow using the Message Broker nodes:

1. Create a directory file that is created or copied for the message flow. In our scenario, we created the directory C:\SampleFlow.
2. Start Message Broker Toolkit V8.0, and create a new application and message flow.
3. Using the information in Table 5-3, add and wire the nodes to the message flow. As you build the message flow, use Figure 5-2 on page 112 as a reference.

Table 5-3 Nodes used to create the message flow

Palette drawer / node type	Node name	Wiring information (terminal to node)
WebSphere MQ MQInput	FlowInput	<ul style="list-style-type: none"> ▶ Out: FilterRequest ▶ Catch: CatchFlowErrors ▶ Failure: CatchFlowErrors
Transformation .NETCompute	FilterRequest	<ul style="list-style-type: none"> ▶ Failure: CatchFlowErrors ▶ Out: SAP Retrieve ▶ altOut¹: SAP Create
WebSphere Adapters SAPRequest	SAP Retrieve	<ul style="list-style-type: none"> ▶ Failure: CatchSAPErrors ▶ Out: DB<->CreateCRMOnlineCustomer
WebSphere Adapters SAPRequest	SAP Create	<ul style="list-style-type: none"> ▶ Failure: CatchSAPErrors ▶ Out: AddCustDetails
Transformation Compute Node	AddCustDetails	<ul style="list-style-type: none"> ▶ Out: SAP Update
<p>¹ You must add the altOut terminal to the node. Right-click the node, and select Add Output Terminal. Enter the name of the new terminal.</p>		

Palette drawer / node type	Node name	Wiring information (terminal to node)
WebSphere Adapters SAPRequest	SAP Update	<ul style="list-style-type: none"> ▶ Failure: CatchSAPErrors ▶ Out: DB<->CreateCRMOnlineCustomer
Transformation Compute Node	CatchSAPErrors	<ul style="list-style-type: none"> ▶ Out: SAPErrors
Transformation Compute Node	CatchFlowErrors	<ul style="list-style-type: none"> ▶ Out: FlowErrors
WebSphere MQ MQOutput	FlowErrors	(none, end of flow)
WebSphere MQ MQOutput	SAPErrors	(none, end of flow)
WebSphere MQ MQOutput	.NET/CRM Errors	(none, end of flow)
WebSphere MQ MQOutput	FlowOutput	(none, end of flow)
Transformation .NETCompute	DB<->CreateCRMOnlineCustomer	<ul style="list-style-type: none"> ▶ Failure: .NET/CRM Errors ▶ Out: FlowOutput
<p>¹ You must add the altOut terminal to the node. Right-click the node, and select Add Output Terminal. Enter the name of the new terminal.</p>		

5.2.3 Configuring the node properties

The next step is to configure the node properties.

WebSphere MQ nodes

Use the following procedure to configure the FlowInput node.

1. Select the **FlowInput** node on the canvas, and open the Properties view.
2. On the Basic tab, enter INPUT_MSGS_Q as the queue name, as shown in Figure 5-3.

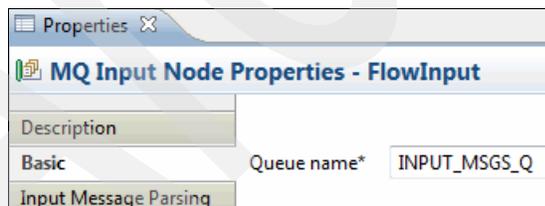


Figure 5-3 MQInput node Queue Name

3. On the Input Message Parsing tab, select **XMLNSC: For XML messages (namespace aware, validation, low memory use)** from the drop-down list on the Message domain field, as shown in Figure 5-4 on page 117.

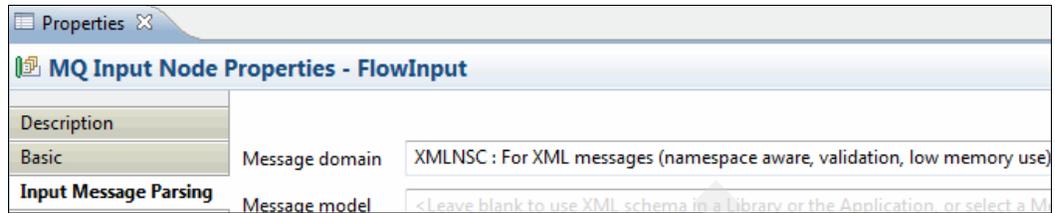


Figure 5-4 MQInput node Message domain

4. Select each MQ Output node, and on the Basic tab of the Properties view, enter their queue name:
 - FlowErrors node: queue name FLOW_ERROR_MSGS
 - SAPErrors node: queue name SAP_ERROR_MSGS
 - .NET/CRM Errors node: queue name DotNet_CRM_ERROR_MSGS
 - FlowOutput node: queue name OUTPUT_MSGS_Q
5. Press Ctrl+S to save your changes.

.NETCompute nodes

The .NETCompute node is configured with the location of the .NET assembly. In our scenario, we use a .NET assembly called MBCreateCRMOnline and its classes, FilterRequest and CreateCRMOnline.

1. Double-click the **FilterRequest** node to open Visual Studio.
2. In Visual Studio, select **File** → **New** → **Project**.
 - a. Select **Visual C#** > **Message Broker** and **Project to filter a Message Broker message**.
 - b. Enter MBCreateCRMOnline as the project and solution name.
 - c. Enter a location for the project.
 - d. Click **OK**.
3. From the Solution Explorer, rename the Filternode.cs to FilterRequest.cs.
4. From the Solution Explorer, right-click the project, and select **Add** → **Class**. The Add New Item screen displays:
 - a. Select **Message Broker** and **Class to create a Message Broker message**.
 - b. Enter CreateCrmOnline.cs in the Name field.
 - c. Click **Add**.

The Project tree in the Solution Explorer now looks like Figure 5-5 on page 118.

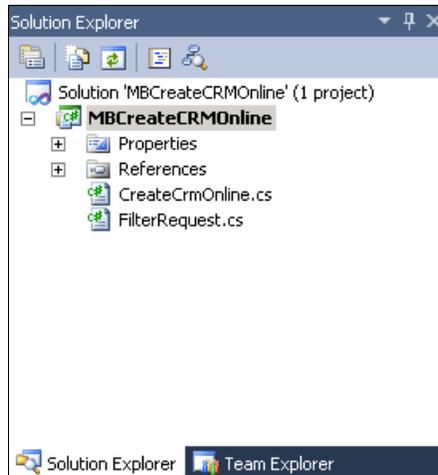


Figure 5-5 CreateCrmOnline.cs class

5. Save all files by pressing Ctrl+Shift+S, and build the project by selecting **Build** → **Build Solution**. This action generates a .NET assembly called MBCreateCRMOnline.dll under the project\bin\debug directory.

The Output view shows the full directory path for the output.

```
----Build started: Project: MBCreateCRMOnline, Configuration: Debug Any CPU----
MBCreateCRMOnline ->
C:\SampleFlow\MBCreateCRMOnline\MBCreateCRMOnline\bin\Debug\MBCreateCRMOnline.dll
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =====
```

The code for the .NET project is created later. For now, all we need to configure the .NETCompute node is an existing dll file.

6. Switch back to the Message Broker Toolkit.
7. In the message flow, select the **FilterRequest** node. In the Properties view of the Basics tab, enter the Assembly name. This is the fully qualified path of the dll file:

```
C:\SampleFlow\MBCreateCRMOnline\MBCreateCRMOnline\bin\Debug\MBCreateCRMOnline.dll
```

Enter FilterRequest as the Class name.

The results are shown in Figure 5-6. Save your changes by pressing Ctrl+S.

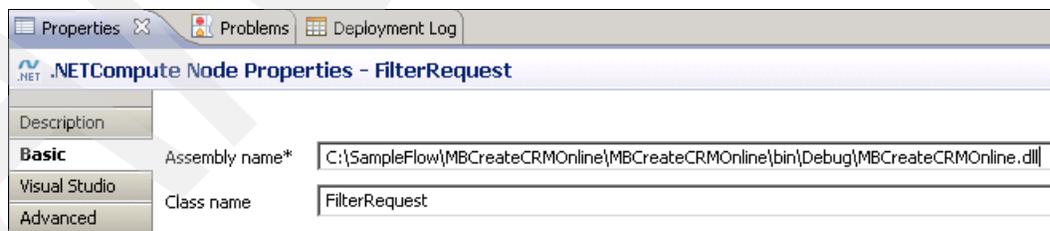


Figure 5-6 FilterRequest properties

8. Repeat step 7 to define the properties for the **DB<->CreateCRMOnlineCustomer** node. Use the same value for the Assembly name field and enter the Class name as CreateCrmOnline.

Compute node

Use the following procedure to create projects and define properties for the following Compute nodes we defined for this scenario:

- ▶ AddCustDetails
- ▶ CatchFlowErrors
- ▶ CatchSAPErrors

1. For each Compute node, open the Properties view, and select the **Basic** tab.
2. Select the Compute mode as **LocalEnvironment and Message**.
3. Leave all of the other default settings, and press Ctrl+S to save the properties for each node.

The ESQL for the Compute nodes is created in later sections.

SAPRequest nodes

WebSphere Adapters allow the message flow to communicate with Enterprise Information Systems, including SAP. The SAPRequest nodes used in this flow support the use of the WebSphere Adapter for SAP.

To configure the SAP request nodes:

1. Right-click the **SAPRetrieve** node, and select **Open Adapter Component** to open the Adapter Component Select panel shown in Figure 3.

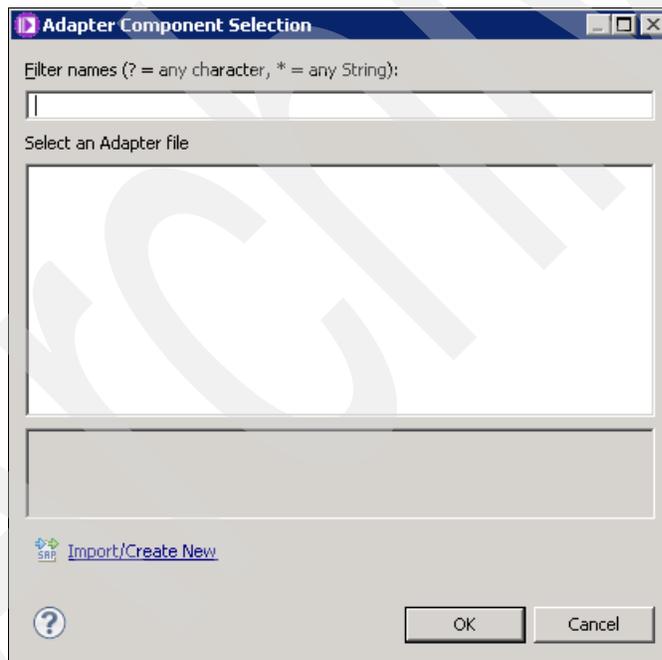


Figure 5-7 Select an adapter component

2. Click the **Import/Create New** link at the bottom of the screen in Figure 5-7.
3. In the Figure 5-8 on page 120, select **IBM WebSphere Adapter for SAP Software with transaction support**, and click **Next**.

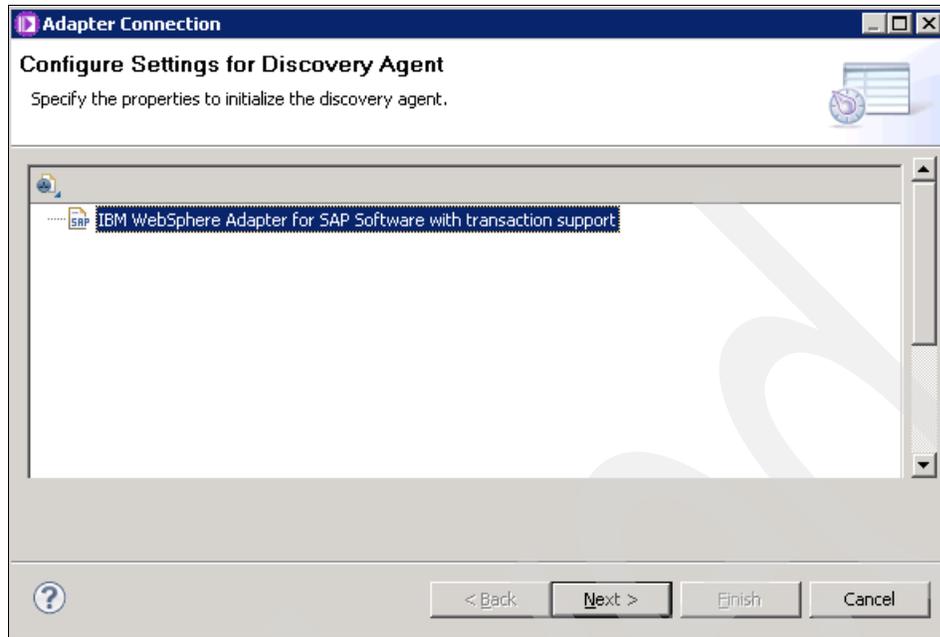


Figure 5-8 Create an adapter connection

4. In the Connector Import screen (Figure 5-9), accept the name provided in the Connector project field, and click **Next**.

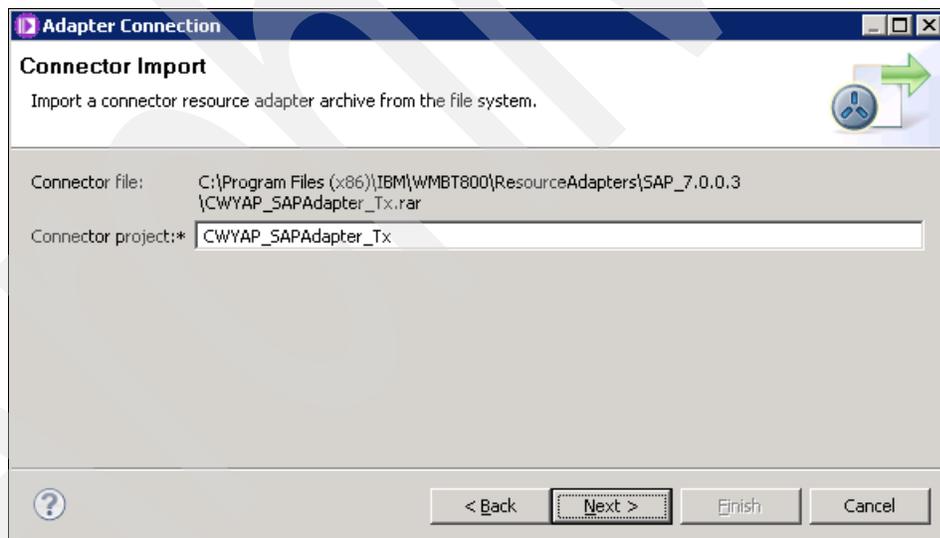


Figure 5-9 Create a connector project

5. The connector is built and Figure 5-10 on page 121 displays. Enter the paths and file names for the SAP jar and dll file, as shown. Click **Next**.

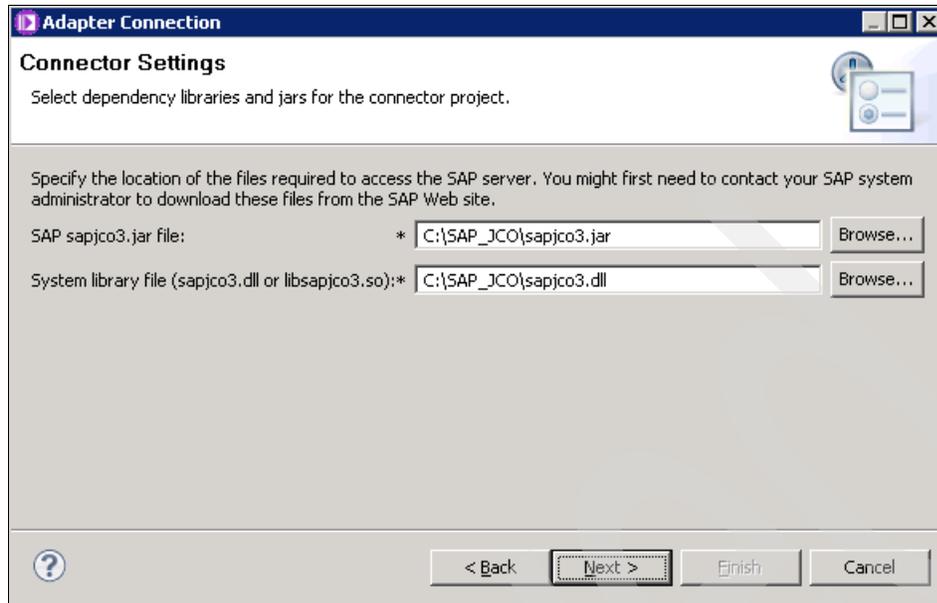


Figure 5-10 Connector files

6. Enter the host name, user name, and password required to connect to the SAP system. Specify **BAPI** as the SAP interface name, as shown in Figure 5-11. Click **Next**.

The values in Figure 5-11 are example values. Host name, system number, client, user name, and password are specific to your SAP system.

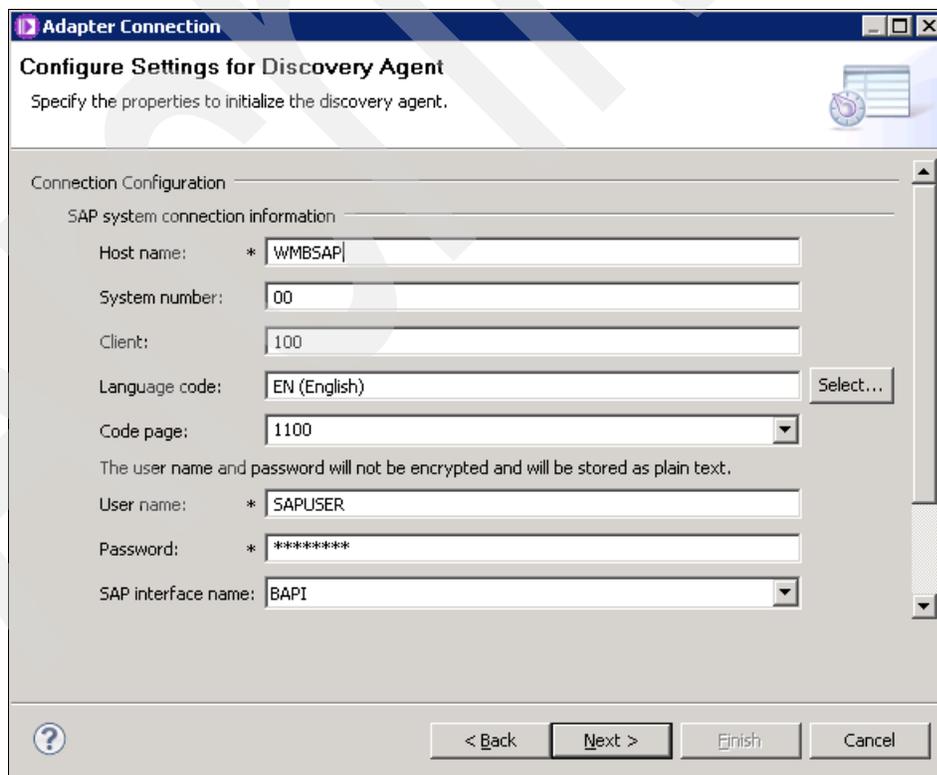


Figure 5-11 Configure the connector

7. On the next screen, Figure 5-12, you discover services that are available on the system that you are connected to. In this scenario, the SAP nodes tries to retrieve or create customer information using the BAPI interface, which is available in the Business Object Repository (BOR) object.

Select **BOR**, and click the **Filter** button. Apply a filter to narrow your search on the BOR. In this case, we use Customer1533. Click **OK**.

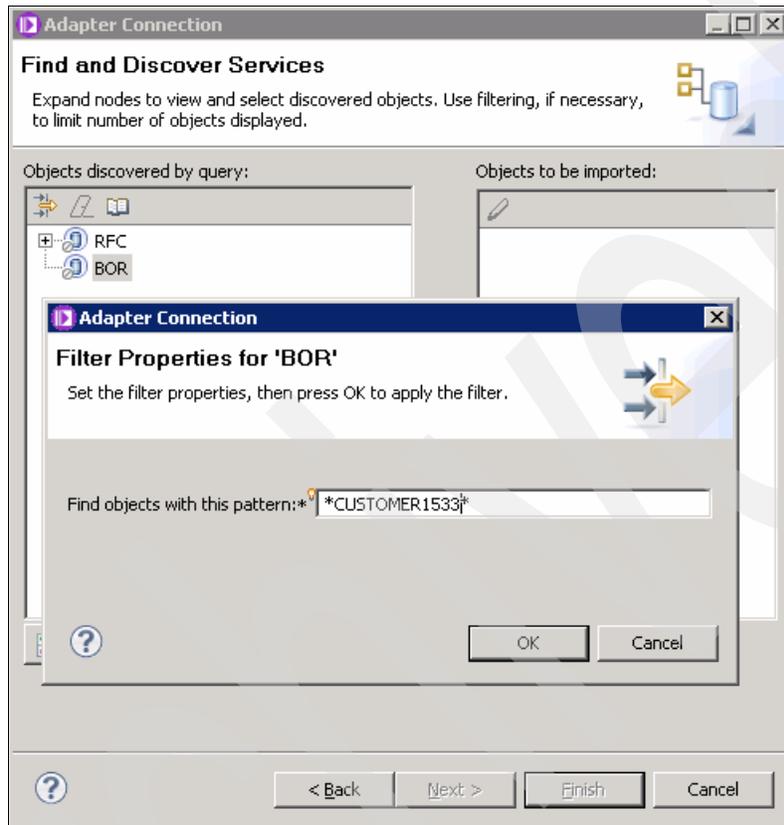


Figure 5-12 Apply a filter

8. In the Objects discovered by query column, click **expand BOR** to see the objects discovered by the search.
9. Select the desired object and expand the view on it.

In our example, we need BAPIs to create, get, and change the customer details from the SAPCreate, SAPRetrieve, and SAPUpdate nodes.

Select the BAPI objects and move them to the Objects to be imported column, as shown in Figure 5-13 on page 123. As you click the  button to move each object, a configuration panel opens. Click **OK** on that panel:

- BAPI_CUSTOMER_CHANGEFROMDATA1
- BAPI_CUSTOMER_CREATEFROMDATA1
- BAPI_CUSTOMER_GETDETAIL1

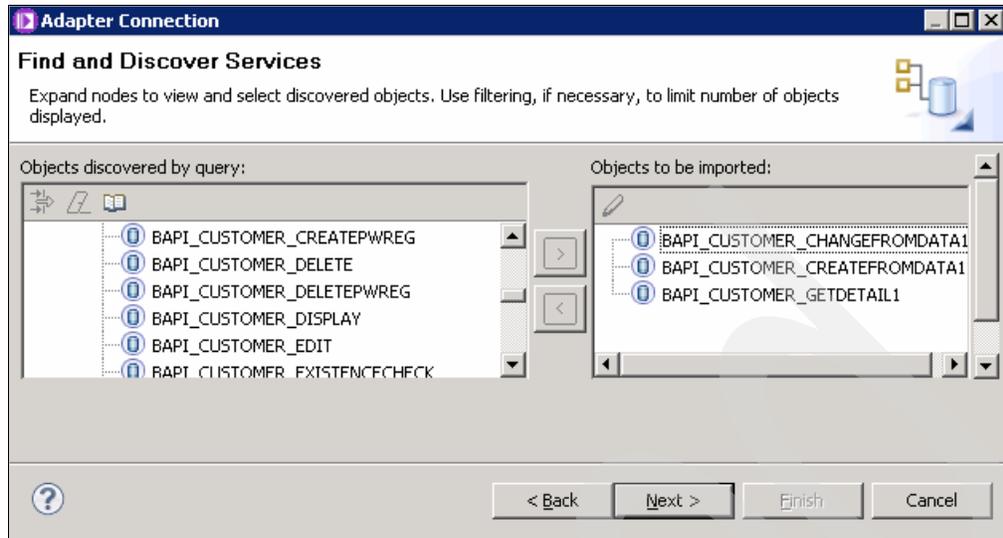


Figure 5-13 Select the BAPI objects to be imported

After you move all of the objects to the right column, click **Next**.

10. In the Configure Objects screen, verify the business object namespace, and select **Synchronous RFC** for the SAP call type, as shown in Figure 5-14. Click **Next**.

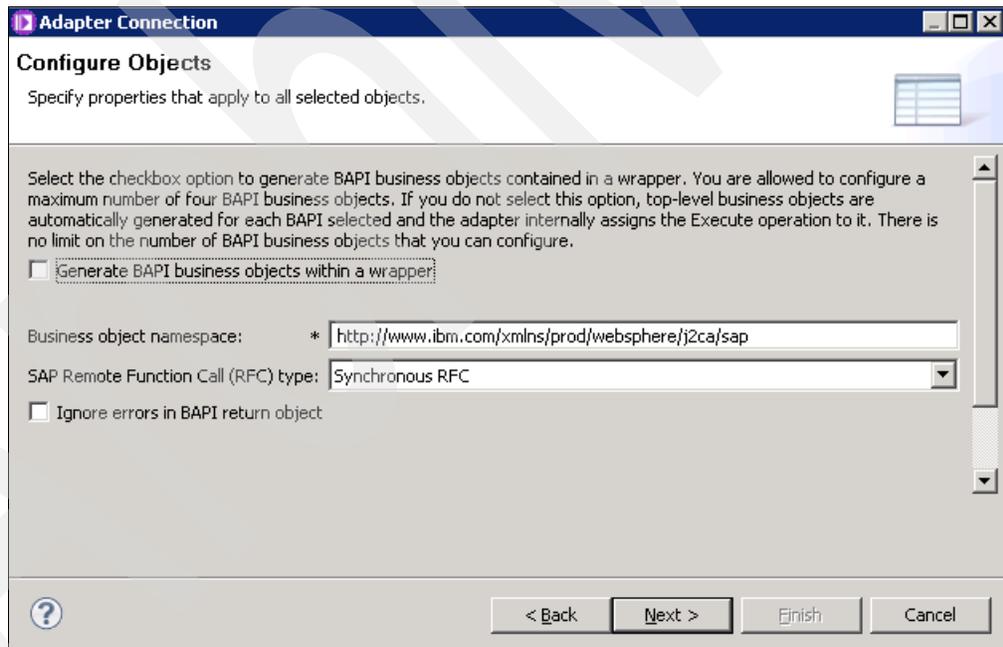


Figure 5-14 Specify object configuration properties

11. On Figure 5-15 on page 124, you set properties for service generation and deployment. Verify and complete the connection properties to be used for the SAP system at run time, and click **Next**.

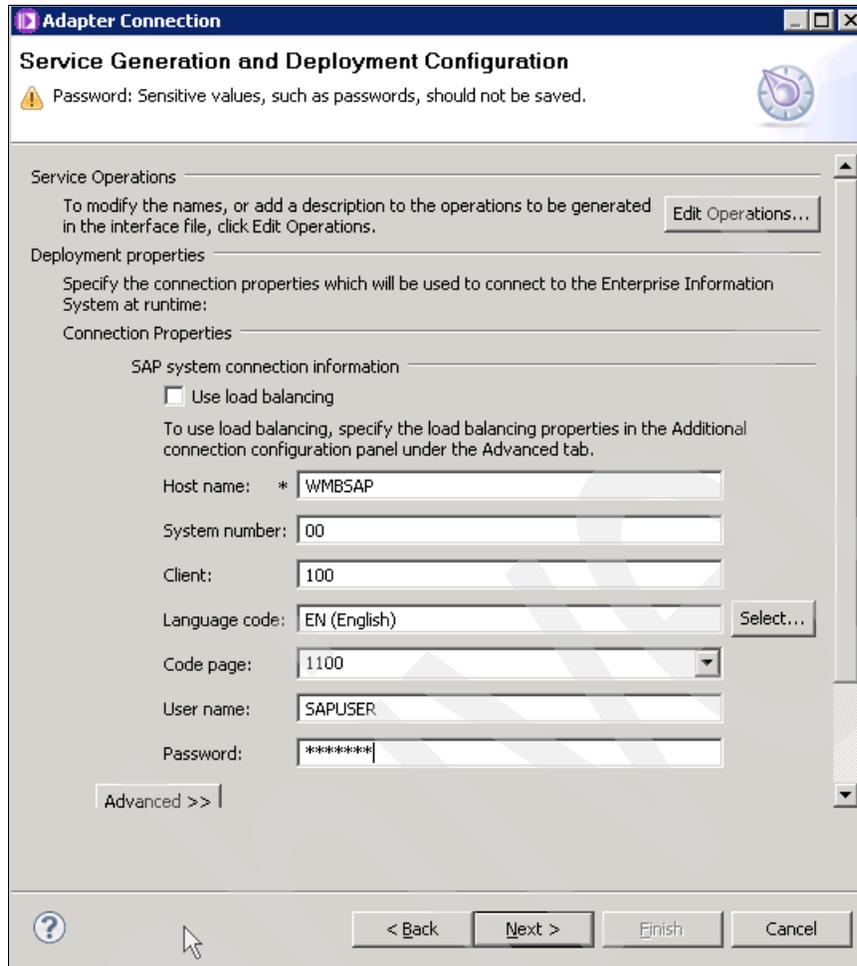


Figure 5-15 Service generation and deployment configuration properties

12. On the last screen of the wizard, Figure 5-16 on page 125, set properties for creating and running the connector. Set the adapter component name. In this scenario, we use SapCustomerOutbound. Verify the remaining properties, and click **Finish**.

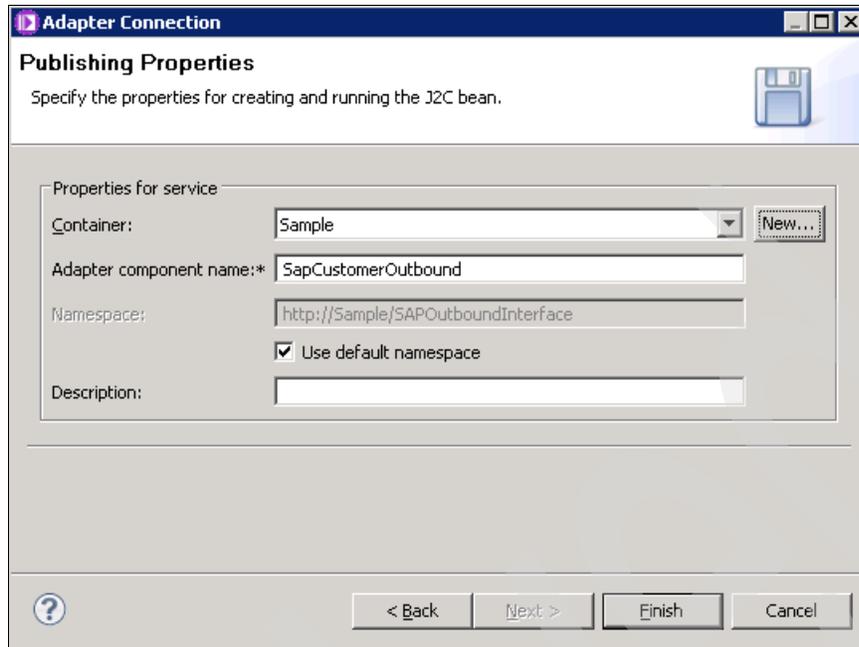


Figure 5-16 Publishing properties for the adapter

13. After the adapter connection wizard is complete, update the SAPRequest nodes (SAP Retrieve, SAP Create, and SAP Update) on the message flow with the adapter connection information.

To do this, select each node and open the Basic tab in the Properties view:

a. Select the **SAP Retrieve** node, as shown in Figure 5-17.

The Browse button provides a list of adapter components to choose from. The values are determined by the publishing properties that you set in Figure 5-16.

Select **Adapters/SAP/SapCustomerOutbound.outadapter**.

b. After you select the adapter, the drop-down list for the Default Method provides a list of methods to choose from. Select **executeSapBapiCustomerGetdetail1**.

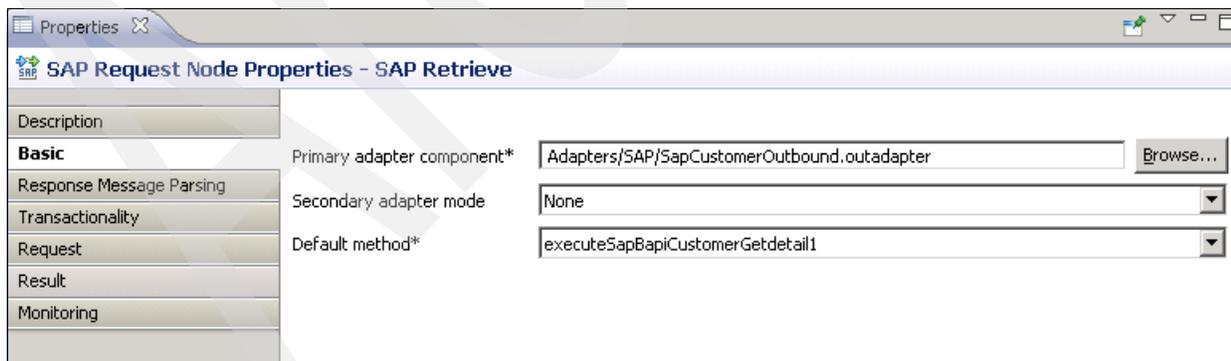


Figure 5-17 Define SAP Retrieve node properties

c. Open the **SAP Create node** properties, as shown in Figure 5-18 on page 126. The primary adapter component is the same as the primary adapter component for the SAP Retrieve node. Select **executeSapBapiCustomerCreatefromdata1** as the default method.

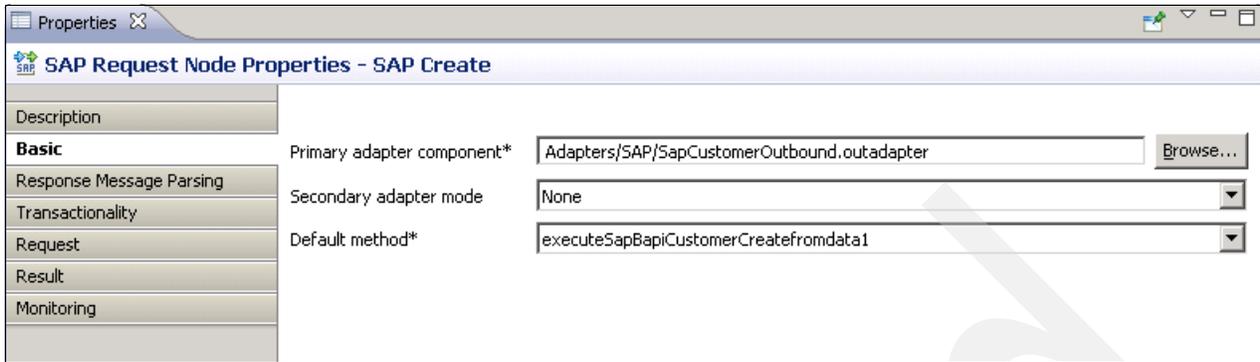


Figure 5-18 Define SAP Create node properties

- d. Open the **SAP Update** node properties, as shown in Figure 5-19. The primary adapter component is the same as the primary adapter component for the SAP Retrieve node. Select **executeSapBapiCustomerChangefromdata1** as the default method.

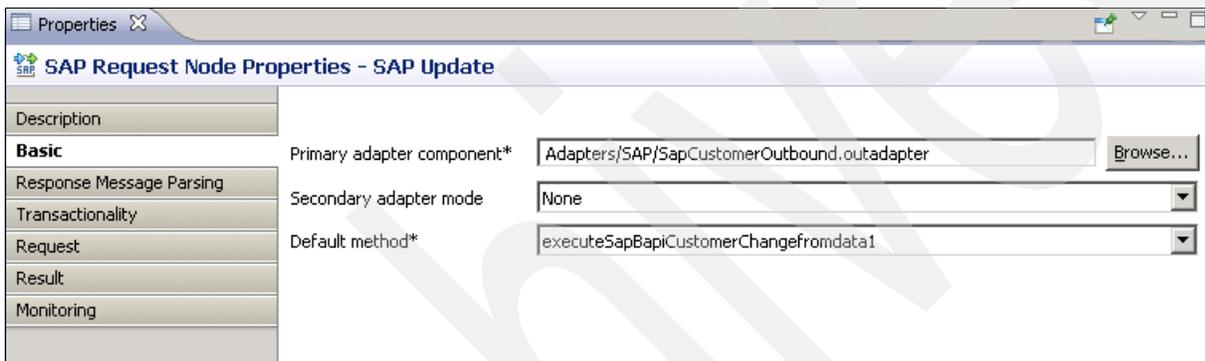


Figure 5-19 Define SAP Update node properties

After all of the SAPRequest nodes are updated, the list of schema definitions for the application in the Broker Development view will look like Figure 5-20 on page 127.

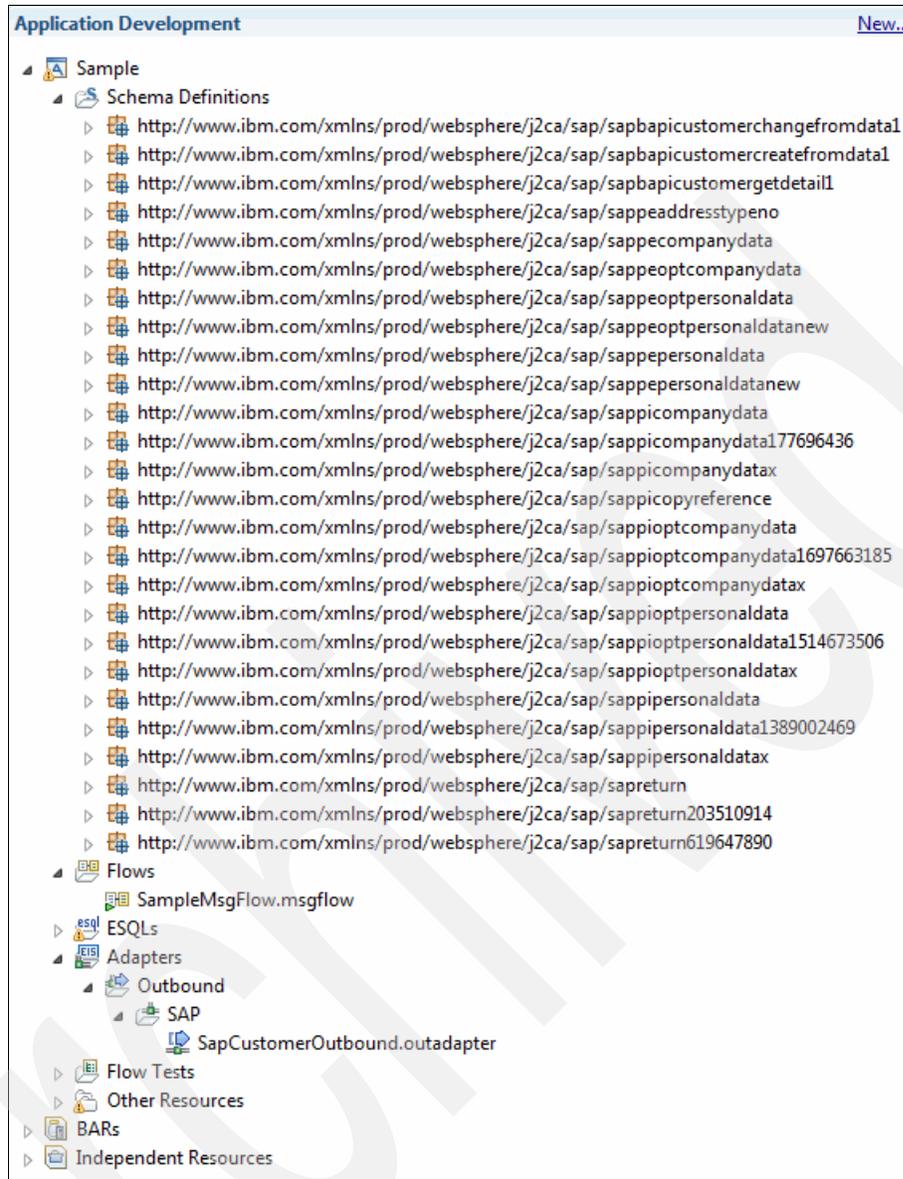


Figure 5-20 Application schema definitions

5.2.4 Broker configuration for SAP request nodes

In this section, we modify the broker properties to refer to the correct location for the Jco Jars and native libraries at run time.

Setting the path for SAP

First, we use the `mqsichangeproperties` command to update the location of the JAR file (`sapjco3.jar`) and native library (`sapjco3.dll`) for the SAP adapter.

1. To use this command, open a command prompt and switch to the bin directory for WebSphere Message Broker, for example, `C:\Program Files\IBM\MQS\8.0.0.0\bin`.
2. Enter `mqsiprofile` to prepare the environment for commands.

3. The following command is used to set the path for the JAR file:

```
mqsischangeproperties BrokerName -c EISProviders -o SAP -n jarsURL -v Jco loc
```

In this scenario, we enter the following command:

```
mqsischangeproperties SAMPLEBROKER -c EISProviders -o SAP -n jarsURL -v  
C:\SAP_JCO
```

4. The following command is used to set the path for the SAP native library:

```
mqsischangeproperties BrokerName -c EISProviders -o SAP -n nativeLibs-v Jco loc
```

In this scenario, we enter the following command:

```
mqsischangeproperties SAMPLEBROKER -c EISProviders -o SAP -n nativeLibs -v  
C:\SAP_JCO
```

UserID and Password for adapter connection

Associate the userID and password values with the adapter component used by the SAP Request nodes:

1. Stop the broker:

```
mqsisstop SAMPLEBROKER
```

2. Set the user ID and password for the component. The command format is:

```
mqsisetdbparms brokerName -n eis::Adaptername -u userid -p password
```

In this example, we enter:

```
mqsisetdbparms SAMPLEBROKER -n eis::SapOutboundCustomer.outadapter -u MyUserID  
-p MyPassword
```

3. Restart the broker with the following command:

```
mqisstart SAMPLEBROKER
```

5.2.5 Writing the code for the SAP nodes

After the adapter connection is configured, you will find a number of schema definitions created by the SAPRequest node. These schemas provide the message model for the BAPI queries and are generated based on the entity descriptions with BAPI. You can change these element names and descriptions to be more meaningful and accessible.

5.2.6 Coding the ESQL for the Compute nodes

Compute nodes are used to process the errors and to add some data to update the SAP Customer record.

CatchFlowErrors and CatchSAPErrors Compute nodes

These nodes are used to capture errors during the flow and put them out as MQ messages. In this simple scenario, the same code will be used for both compute nodes:

1. Double-click the **CatchFlowErrors** node to open the ESQL. Replace the SampleFlow_Compute module with the code in Example 5-1.

Example 5-1 ESQL for handling errors

```
CREATE COMPUTE MODULE SampleMsgFlow_Proc_Errors  
CREATE FUNCTION Main() RETURNS BOOLEAN  
BEGIN
```

```

        CALL CopyMessageHeaders();
        -- Create an error message detailing the broker exception message
        SET OutputRoot.XMLNSC.Failed='FailureData';
        SET OutputRoot.XMLNSC.Failed.Body=InputBody;
        SET OutputRoot.XMLNSC.Failed.Exception=InputExceptionList;
        -- 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;

```

2. Save and close the ESQL.
3. In the Basic tab of the Properties view for both the CatchFlowErrors and CatchSAPErrors, change the ESQL module name to SampleMsgFlow_Proc_Errors.
4. Save the message flow.

AddCustDetail Compute node

This node is used to construct a request message to update the customer information about the SAP system. In the SAP environment, SapBapiCustomerCreatefromdata1 creates a customer record with the details of the referenced customer. This record can be updated later using SapBapiCustomrChangefromdata1:

1. Double-click the AddCustDetails node in the message flow to open the ESQL.
2. Replace the default module generated with the code in Example 5-2.

Example 5-2 Compute node example

```

DECLARE ns3 NAMESPACE
'http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/sapbapicustomercreatefromdata1';
DECLARE ns5 NAMESPACE
'http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/sapbapicustomerchangefromdata1';
DECLARE ns8 NAMESPACE
'http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/sapbapicustomergetdetail1';

CREATE COMPUTE MODULE SampleMsgFlow_AddCustDetails
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyMessageHeaders();
    -- Copy the customerno into Environment tree.

```

```

CREATE NEXTSIBLING OF Environment.Root TYPE NameValue NAME 'Customerno' VALUE
InputBody.ns3:SapBapiCustomerCreatefromdata1.Customerno;

-- Copy the customerno into Output root.
SET OutputRoot.XMLNSC.ns5:SapBapiCustomerChangefromdata1.Customerno =
    InputBody.ns3:SapBapiCustomerCreatefromdata1.Customerno;

-- Create a reference for the values being updated
DECLARE custref REFERENCE TO OutputRoot.XMLNSC.ns5:SapBapiCustomerChangefromdata1;

-- Update the values
SET custref.PiDistrChan = '01';
SET custref.PiDivision = '01';
SET custref.PiSalesorg = '0001';
SET custref.SapPiPersonaldata.Firstname = 'Jane';
SET custref.SapPiPersonaldatax.Firstname = 'X';
SET custref.SapPiPersonaldata.Lastname = 'Doe';
SET custref.SapPiPersonaldatax.Lastname = 'X';
SET custref.SapPiPersonaldata.City = 'Carlisle';
SET custref.SapPiPersonaldatax.City = 'X';
SET custref.SapPiPersonaldata.PostlCod1 = '94010';
SET custref.SapPiPersonaldatax.PostlCod1 = 'X';
SET custref.SapPiPersonaldata.EMail = 'john@john.com';
SET custref.SapPiPersonaldatax.EMail = 'X';
SET custref.SapPiPersonaldatax.Tel1Numbr = '999-999-9999';
SET custref.SapPiPersonaldata.Country = 'US';
SET custref.SapPiPersonaldatax.Country = 'X';
SET custref.SapPiPersonaldata.Region = 'CA';
SET custref.SapPiPersonaldatax.Region = 'X';
SET custref.SapPiPersonaldata.LanguP = 'EN';
SET custref.SapPiPersonaldatax.LanguP = 'X';
SET custref.SapPiPersonaldata.Currency = 'USD';
SET custref.SapPiPersonaldatax.Currency = 'X';

-- 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;

```

3. Save and close the ESQL.
4. In the Basic tab of the Properties view for the AddCustDetails node, change the ESQL module name to SampleMsgFlow_AddCustDetails.
5. Save the message flow.

5.2.7 Coding the Filter Request .NETCompute node

Use the following steps to write this code:

1. Open the solution for MBCreateCRMOnline in Visual Studio.
2. In the Solution Explorer, double-click `FilterRequest.cs` to open it.
3. Replace the `Evaluate()` method with the code shown in Example 5-3.

Example 5-3 .NETCompute node filter request

```
public override void Evaluate(NBMessageAssembly inputAssembly)
{

    //Make connection to output terminals
    NBOutputTerminal outSAPRetrieve = OutputTerminal("Out");
    NBOutputTerminal outSAPCreate = OutputTerminal("altOut");
    NBOutputTerminal failureTerminal = OutputTerminal("Failure");

    NBMessage inputMessage = inputAssembly.Message;
    NBElement root = inputMessage.RootElement;
    try
    {
        NBElement t1s = root;
        NBElement key = root[NBParsers.XMLNSC.ParserName].FirstChild;

        // Add user code in this region to filter the message
        switch (key.Name)
        {
            case "SapBapiCustomerGetdetail1":
                // Existing Customer.
                // Extract details from SAP, create an account into CRM.
                inputAssembly.Environment.RootElement.AddLastChild(key);
                outSAPRetrieve.Propagate(inputAssembly);
                break;

            case "SapBapiCustomerCreatefromdata1":
                // New customer. Create record in SAP and CRM.
                inputAssembly.Environment.RootElement.AddLastChild(key);
                outSAPCreate.Propagate(inputAssembly);
                break;

            default:
                //Unrecognized flow. Move it to Failure.
                failureTerminal.Propagate(inputAssembly);
                break;
        }
    }
    catch (Exception)
    {
        failureTerminal.Propagate(inputAssembly);
    }
}
```

The `Evaluate` method is invoked by `DataFlowEngine` when there is a message in the `FlowInput` node.

In the Evaluate method, we check the first child of the XML message:

- ▶ If it is "SapBapiCustomerGetdetail1", we assume it is an existing customer and route it to SAPRetrieve node.
- ▶ If it is "SapBapiCustomerCreatefromdata1", we assume it is a new customer and route it to SAPCreate.

We add the incoming message to the Environment as the last child to retrieve the original message again later in the message flow.

5.2.8 Writing the code for the database operations

The DB<->CreateCRMOnlineCustomer node contains a lot of activity, including database operations, Microsoft Dynamics CRM operations, and code related to the broker. We categorize this sections for each activity.

We start by adding a new application configuration file (App.Config) to the MBCreateCRMOnline project in Visual Studio:

1. In the Solution Explorer, right-click the project, and select **Add** → **New Item**.
2. Select **Visual C# Items** in the Installed Templates column, and select **Application Configuration File**.
3. Leave the name as App.config, and click **Add**.
4. The App.config file is used to store the credentials and URIs and any volatile data that can be accessed from it during application execution. Replace the generated XML in App.config with the XML in Example 5-4.

Example 5-4 Contents of app.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <clear/>
    <add key="DBConnectionString" value="Server=ITSOSQLServer;
                                     Database=ITSOCRM;
                                     user id=MyUser;
                                     password =MyPw;
                                     Trusted_Connection = no;
                                     connection timeout = 60;" />
    <add key ="StoredProcedureName" value = "dbo.GetAccNumberByEmail" />
    <add key ="CRMURI" value="https://myid.crm.dynamics.com"/>
    <add key ="CRMUser" value ="MyUser" />
    <add key ="CRMUserPwd" value="MyPW"/>
  </appSettings>
</configuration>
```

5. Save and close App.config.

Code for interactions with the database

Add a new class to the project called DBOperations. We use this class to manage all database-related operations. To add the DBOperations class to the project:

1. Add a new reference to the project. Right-click **References** in the Solution Explorer, and select **Add Reference**. Click the .NET tab, select **System.Configuration**, and click **OK**.

2. Select **Project** → **Add Class**.
3. Select **Class**, enter `DBOperations.cs` as the name, and click **Add**.
4. Replace the generated code with the code shown in Example 5-5.

Example 5-5 Code for DBOperations

```

using System;
using System.Collections;
using System.Linq;
using System.Text;
using System.Data.SqlClient;
using System.Configuration;

namespace MBCreateCRMOnline
{
    /// <summary>
    /// Performs all the Database related operations.
    /// Its major function is to access a stored procedure
    /// in Database and get AccountNumber.
    /// </summary>
    internal class DBOperations
    {
        SqlConnection dbConnection = null;
        String connString = null;
        String procedureName = null;
        SqlCommand query;

        /// <summary>
        /// Constructor. Opens the Database connection.
        /// </summary>
        public DBOperations()
        {
            if (ConfigurationManager.AppSettings.Count > 0)
            {
                connString =
                (String)ConfigurationManager.AppSettings["DBConnectionString"];
                procedureName =
                (String)ConfigurationManager.AppSettings["StoredProcedureName"];
            }

            //Open Database.
            dbConnection = new SqlConnection(connString);
            dbConnection.Open();
        }

        /// <summary>
        /// Check Database Record and return Account number.
        /// </summary>
        /// <param name="email"></param>
        /// <returns></returns>
        internal String CheckDBRecordsFromSAP(String email)
        {
            String accNo = null;

            try

```

```

        {
//Create a new instance of sql command.
query = new SqlCommand(procedureName, dbConnection);
query.CommandType = System.Data.CommandType.StoredProcedure;

        //Input parameter.
        SqlParameter email_ = new SqlParameter("@email",
        System.Data.SqlDbType.VarChar, 240);
        email_.Value = email;

        //Output parameter.
        SqlParameter accNum = new SqlParameter("@AccNo", System.Data.SqlDbType.VarChar,
        15);

        accNum.Direction = System.Data.ParameterDirection.Output;

        query.Parameters.Add(email_);
        query.Parameters.Add(accNum);

        //Execute.
        query.ExecuteNonQuery();

        //Grab the retrn value.
        if (accNum.Value != null)
            accNo = accNum.Value.ToString();

        //Release the query resources.
        query.Dispose();

        }
        catch (Exception)
        {
            throw;
        }
        return accNo;
    }
}
}

```

This code queries the database and checks to see if a customer with the given email address already exists. If the customer exists, it retrieves the account identifier and tags it on to the CRM account.

The code uses a stored procedure on the SQL Server to return the Customer Account Number. The stored procedure takes one String type input parameter Email and returns the String type AccountNumber, shown in Example 5-6 on page 135.

Create the following stored procedure for the SQL database instance that the message flow is using. Stored procedures can be added from Visual Studio in the Server Explorer view.

Example 5-6 SQL server database query

```
CREATE PROCEDURE GeAccNumberByEmail (@Email NVARCHAR(50),@AccNo NVARCHAR(15)
OUTPUT)
AS
Select @AccNo = AccountNumber from dbo.Accounts where CustomerID IN(Select
CustomerID from dbo.Customers where EmailAddress=@Email)
GO
```

5.2.9 Writing the code for accessing Microsoft Dynamics CRM Online

Microsoft Dynamics CRM offers an early-bound entity data model, late-bound entity data model, REST endpoints, and WSDL interfaces for client connectivity. Programming scenarios in earlier versions of Microsoft Dynamics CRM were based on WSDL with early-bound classes for each entity and a `DynamicEntity` class for late-bound programming scenarios. Microsoft Dynamics CRM 2011 no longer uses WSDL but now provides assemblies to reference and connect to Microsoft Dynamics CRM 2011 for both early-bound and late-bound programming scenarios.

In Microsoft Dynamics CRM, entities are used to model and manage business data. For example, entities, such as *account* and *contact*, can be used to track and support customer activities. Each entity has an array of attributes of a particular data type. For example, account and contact are entities to model customer data and contain attributes that define such things as Name, Email, Phone, and so on.

► Late-bound

This programming model allows applications to access the entities that are not yet defined and allows code to be written in a generic manner. The code can contain and define custom entities and their attributes that are not available during code compilation. When initialized, the Entity class must have a logical name (to represent an entity, such as contact or account) and a property context of the entity's attributes.

► Early-bound

The Microsoft Dynamics CRM code generation tool (`CrmSvcUtil`) creates early-bound entity classes that are used to access the business data. These classes include one class for each entity in your installation, including custom entities. Any time customizations are made to the system, these classes must be regenerated. Classes created by the code generation tool include all entity attribute and relationships.

Other programming models are:

- REST endpoint for AJAX and Silverlight clients to access Microsoft Dynamics CRM directly rather than invoking a SOAP-based web service.
- WSDL for the use by non .NET applications and does not depend on using Dynamics CRM assemblies.

More information about these programming models are at:

<http://msdn.microsoft.com/en-us/library/gg327971.aspx>

Programming model for the scenario: This scenario demonstrates interactions with Microsoft Dynamics CRM Online using the late-bound programming model.

You can also use the early-bound programming model. In that case, the corresponding managed code must be generated using the CrmSvcUtil tool and must be added and referenced by the CreateCrmCustomer.cs class. You must create the Customer(Account,Entity) data using the syntax and semantics of the early-bound programming model.

Using web services

Microsoft Dynamics CRM provides two web services for accessing CRM Online, one to identify your organization and another to access the CRM data.

IDiscoveryService web service

A single Microsoft Dynamics CRM installation can host multiple organizations on multiple servers. Therefore, it is important to specify which organization you want to access. The IDiscoveryService web service returns a list of organizations that the specified user belongs to and the URL endpoint address for each organization. The service only returns those organizations to which a user has access. For more information about the IDiscoveryService web service, see:

<http://msdn.microsoft.com/en-us/library/gg328127.aspx>

IOrganizationService web service

The IOrganizationService primary web service is for accessing data and metadata in Microsoft Dynamics CRM 2011. For more information about IOrganization web service, see:

<http://msdn.microsoft.com/en-us/library/gg309449.aspx>

Add Microsoft assemblies to your project

Using Microsoft Dynamics CRM SDK assemblies to access the CRM data is the recommended developer scenario for Dynamics CRM 2011 and Online. These assemblies are in the SDK\Bin folder of the Microsoft Dynamics CRM SDK along with XML Intellisense files for Visual Studio. For more information about the assemblies, their contents, and usage, see:

<http://msdn.microsoft.com/en-us/library/gg334621.aspx>

Authenticate users with Microsoft Dynamics CRM web services

Microsoft Dynamics CRM supports multiple authentication models. The type of authentication interface that is used depends on the type of deployment your application is accessing, Microsoft Dynamics CRM Online or Microsoft Dynamics CRM 2011. Microsoft CRM Online-based deployment supports two security models: Claim-based authentication and Active Directory authentication.

Authentication is performed with the help of proxy classes. This scenario uses the OrganizationServiceProxy and DiscoveryServiceProxy classes for authenticating with CRM Online. The four-parameter constructor of these classes supports Microsoft Dynamics CRM Online Windows Live ID deployments. These proxy classes automatically handle Claims or Active Directory authentication for the user. More information about this can be found at:

<http://msdn.microsoft.com/en-us/library/gg334502.aspx>

Getting started accessing CRM data

This section provides a summary of the steps you need to follow to access CRM data.

Before creating your code, make sure you meet the following prerequisites:

- ▶ Make sure you have access to a Microsoft Dynamics CRM server, and note the server name.
- ▶ If you are accessing Microsoft Dynamics CRM Online, install Windows Identity Foundation.
- ▶ Have your Windows Live ID, userID, and password available.

To get started:

1. Create a New Project, or open your existing project in Visual Studio.
2. Add references in your project for Microsoft Dynamics CRM assemblies (recommended approach).
3. Create an App.Config file with entries to contain the CRM server name, Live ID, and password.
4. Create an instance of the DiscoveryService and access the list of Organizations you are authorized to access.
5. Create an instance of OrganizationService to access specific a Organization on CRM to create, update, retrieve, and delete the data.

Example

Use the following procedure to practice accessing CRM data. Note that we use the late-bound programming model:

1. Access Microsoft Dynamics CRM Online and obtain the server name (for example, <https://id.crm5.dynamics.com>).
2. Use your Live ID for CRM Online authentication.
3. Open the MBCreateCRMOnline solution in Visual Studio.
4. Add the references highlighted in Figure 5-21 to Project references.

The System. references can be found in the .NET tab. However, the microsoft.crm.sdk.proxy and microsoft.xrm.sdk references are found by using the Browse tab to locate the following dll files in the Microsoft CRM SDK installation:

- Microsoft.Crm.Sdk.Proxy: `sdk_install\sdk\bin\microsoft.crm.sdk.proxy.dll`
- Microsoft.Xrm.Sdk: `sdk_install\sdk\bin\microsoft.xrm.sdk.dll`

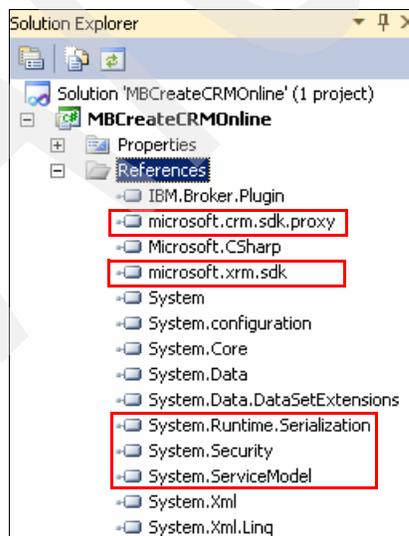


Figure 5-21 CRM libraries

5. While authenticating, you must register a computing device with Windows Live ID through the generation of a Device ID and password. There is a helper class `deviceidmanager.cs` available in the `sdk_install\sdk\samplecode\cs\helpercode` to do this job.

You can add this class to your project or write your own code for doing this job. In our scenario, we use the helper class for device registration.

Add the `deviceidmanager.cs` class to `MBCreateCRMOnline`:

- a. Right-click the project in the Solution Explorer, and select **Add** → **Existing Item**.
 - b. Browse for the helper class `deviceidmanager.cs` in `sdk_install\sdk\samplecode\cs\helpercode`. Select it, and click **Add**.
6. Create a new class `CreateCrmCustomer.cs`:
 - a. Right-click the project in the Solution Explorer, and select **Add** → **Class**. Name the class `CreateCrmCustomer.cs`, and click **Add**.
 - b. Replace the using statements at the top of the file with those shown in Example 5-7.

Example 5-7 Using statements for

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using System.Reflection;
using System.Collections;
using System.Configuration;
using IBM.Broker.Plugin;
// These namespaces are found in the Microsoft.Xrm.Sdk.dll assembly
// located in the SDK\bin folder of the SDK download.
using Microsoft.Xrm.Sdk;
using Microsoft.Xrm.Sdk.Query;
using Microsoft.Xrm.Sdk.Client;
using Microsoft.Xrm.Sdk.Discovery;
using Microsoft.Crm.Sdk.Messages;
using Microsoft.Crm.Services.Utility;
```

- c. Create code for the following actions:
 - Access the user/userid/password values for CRM Online from the `App.Config` file.
 - Create an instance of `IDiscoveryService`.
 - Authenticate on `IDiscoveryService`.
 - Retrieve the list of organizations you are authorized to access.
 - Create an Instance of `IOrganizationService`.
 - Create a `Customer(Account,Contact)` on CRM Online.

Example 5-8 shows the code snippet that performs these actions. Replace the generated code for the `CreateCrmCustomer` class with the code in Example 5-8.

Example 5-8 CRM Online code snippet

```
/// <summary>
/// Demonstrates connectivity to Microsoft Dynamics CRM Online.
/// Action to Create a Microsoft Dynamics CRM Online record.
/// </summary>
internal class CreateCrmCustomer
{
    //Key fields to access Microsoft Dynamics CRM Online.
    private String _server = "OrganizationUniqueName";
    private String _userName = "username@mydomain.com";
    private String _password = "password";
```

```

//Setter and Getters
private OrganizationDetail currentOrganizationDetail { get; set; }
private IDiscoveryService DisService { get; set; }
private IOrganizationService OrgService { get; set; }
private ClientCredentials clientCredentials { get; set; }
private ClientCredentials deviceCredentials { get; set; }
private bool IsLiveIDAuthType { get; set; }

/// <summary>
/// Class Constructor
/// Looks up for uri,uid and pwd from App.Config
/// Creates IDiscoveryService proxy
/// Authenticates
/// Create proxy instance of IOrganization Context.
/// </summary>
public CreateCrmCustomer()
{
    if (ConfigurationManager.AppSettings.Count > 0)
    {
        _server = (String)ConfigurationManager.AppSettings["CRMURI"];
        _userName = (String)ConfigurationManager.AppSettings["CRMUser"];
        _password = (String)ConfigurationManager.AppSettings["CRMUserPwd"];
    }
    Uri discoveryUri = GetDiscoveryServiceUri(_server);

    //Create IDiscoveryService instance.
    IServiceConfiguration<IDiscoveryService> dinfo =
        ServiceConfigurationFactory.CreateConfiguration<IDiscoveryService>(discoveryUri);
    clientCredentials = new ClientCredentials();
    DiscoveryServiceProxy disSrvProxy = null;
    if (GetServerAuthenticationType(discoveryUri) == AuthenticationProviderType.LiveId)
    {
        clientCredentials.UserName.UserName = _userName;
        clientCredentials.UserName.Password = _password;
        deviceCredentials = DeviceIdManager.LoadOrRegisterDevice();
        disSrvProxy = new DiscoveryServiceProxy(discoveryUri, null, clientCredentials, deviceCredentials);
        IsLiveIDAuthType = true;
    }
    //Authenticate.
    disSrvProxy.Authenticate();
    RetrieveOrganizationsRequest orgRequest = new RetrieveOrganizationsRequest();
    RetrieveOrganizationsResponse orgResponse = disSrvProxy.Execute(orgRequest) as
RetrieveOrganizationsResponse;

    //Create IOrganizationService instance.
    currentOrganizationDetail = orgResponse.Details[0];
    Uri orgServiceUri = new Uri(currentOrganizationDetail.Endpoints[EndpointType.OrganizationService]);
    if (IsLiveIDAuthType)
    {
        OrgService = new OrganizationServiceProxy(orgServiceUri, null, clientCredentials,
deviceCredentials);
    }
    else
    {
        IServiceConfiguration<IOrganizationService> orgConfigInfo =
            ServiceConfigurationFactory.CreateConfiguration<IOrganizationService>(orgServiceUri);
        OrgService = new OrganizationServiceProxy(orgConfigInfo, clientCredentials);
    }
}
}

```

```

/// <summary>
/// From the Server provided in App.Config. Construct a meaningful
/// URI for DiscoveryService.
/// </summary>
/// <param name="serverName"></param>
/// <returns></returns>
public Uri GetDiscoveryServiceUri(string serverName)
{
    string uriSuffix = @"/XRMServices/2011/Discovery.svc";
    return new Uri(string.Format("{0}{1}", serverName, uriSuffix));
}
/// <summary>
/// Authentication Type for this Deployment.
/// </summary>
/// <param name="uri"></param>
/// <returns></returns>
public AuthenticationProviderType GetServerAuthenticationType(Uri uri)
{
return ServiceConfigurationFactory.CreateConfiguration<IDiscoveryService>(uri).AuthenticationType;
}

/// <summary>
/// Creates a Crm Customer(account,contact).
/// Makes the contact as the primary contact of the account.
/// </summary>
/// <param name="input">List of attributes(xml form)</param>
/// <param name="custNumber">value from SAP Records</param>
/// <returns></returns>
internal Guid CreateCrmCustomerRelation(NBElement input, String custNumber)
{
    Entity contact = new Entity("contact");
    Entity account = new Entity("account");
    if (input != null)
    {
        contact["firstname"] = input.FirstChild.ValueAsString;
        NBElement sibling = input.FirstChild.NextSibling;
        while (sibling != null)
        {
            switch (sibling.Name.ToLower())
            {
                case "lastname":
                    if (!sibling.ValueIsNull)
                    {
                        contact["lastname"] = sibling.ValueAsString;
                    }
                    break;
                case "jobtitle":
                    if (!sibling.ValueIsNull)
                    {
                        contact["jobtitle"] = sibling.ValueAsString;
                    }
                    break;
                case "street":
                    if (!sibling.ValueIsNull)
                    {
                        contact["address1_line1"] = sibling.ValueAsString;
                        account["address1_line1"] = sibling.ValueAsString;
                    }
                    break;
                case "city":

```

```

        if (!sibling.ValueIsNull)
        {
            contact["address1_city"] = sibling.ValueAsString;
            account["address1_city"] = sibling.ValueAsString;
        }
        break;
    case "regionstateprovincecounty":
        if (!sibling.ValueIsNull)
        {
            contact["address1_county"] = sibling.ValueAsString;
            account["address1_county"] = sibling.ValueAsString;
        }
        break;
    case "countrykey":
        if (!sibling.ValueIsNull)
        {
            contact["address1_country"] = sibling.ValueAsString;
            account["address1_country"] = sibling.ValueAsString;
        }
        break;
    case "citypostalcode":
        if (!sibling.ValueIsNull)
        {
            contact["address1_postalcode"] = sibling.ValueAsString;
            account["address1_postalcode"] = sibling.ValueAsString;
        }
        break;
    case "emailaddress":
        if (!sibling.ValueIsNull)
        {
            contact["emailaddress1"] = sibling.ValueAsString;
            account["emailaddress1"] = sibling.ValueAsString;
        }
        break;
    case "telnumber":
        if (!sibling.ValueIsNull)
        {
            contact["telephone1"] = sibling.ValueAsString;
            account["telephone1"] = sibling.ValueAsString;
        }
        break;
        default:
            break;
    }
    sibling = sibling.NextSibling;
}
}
}
//Create a Contact firts.
Guid contactid = OrgService.Create(contact);
//Instantiate Account.
account["name"] = "SAP Customer";
account["accountnumber"] = custNumber;

//Associate Account and contact as primary contact.
account.Attributes.Add("primarycontactid", new EntityReference("contact", contactid));
//Create account.
return OrgService.Create(account);
}
}

```

- d. Save CreateCrmCustomer.cs.

5.2.10 Writing the code for the .NETCompute node (Create) - DB->CreateCRMOnlineCustomer

Write the code for the DB->CreateCRMOnlineCustomer node for calling the methods on DBOperations or CreateCrmCustomer.cs classes:

1. In Visual Studio, open the CreateCrmOnline.cs class.
2. Add the code in bold shown in Example 5-9 to the class.

Example 5-9 Code to add to the class

```
namespace MBCreateCRMOnline
{
    /// <summary>
    /// CreateCrmOnline Class
    /// </summary>
    public class CreateCrmOnline : NBComputeNode
    {
        CreateCrmCustomer crmOrg = new CreateCrmCustomer();
        DBOperations dbOp = new DBOperations();
    }
}
```

3. Replace the Evaluate() method with the code in Example 5-10.

Example 5-10 DB->CreateCRMOnlineCustomer code snippet

```
public override void Evaluate(NBMessageAssembly inputAssembly)
{
    //Create output connections
    NBOutputTerminal outTerminal = OutputTerminal("Out");
    NBOutputTerminal failureTerminal = OutputTerminal("failure");
    NBMessage inputMessage = inputAssembly.Message;

    // Create a new empty message, ensuring it is disposed after use
    using (NBMessage outputMessage = new NBMessage())
    {
        NBMessageAssembly outAssembly = new NBMessageAssembly(inputAssembly, outputMessage);
        NBElement inputRoot = inputMessage.RootElement;
        NBElement outputRoot = outputMessage.RootElement;
        try
        {
            // Optionally copy message headers, remove if not needed
            CopyMessageHeaders(inputRoot, outputRoot);

            #region UserCode

            NBElement envRoot = inputAssembly.Environment.RootElement;
            NBElement key = envRoot.FirstChild;
            string notargetnamespace = "";

            string namespaceStore =
            "http://www.ibm.com/xmlns/prod/websphere/j2ca/sap/sapbapicustomergetdetail1";

            Guid response = new Guid();
            if (key.Name.ToLower().StartsWith("sapbapicustomergetdetail1"))
```

```

    {
    NBElement Action = inputRoot[notargetnamespace, "DataObject"][namespaceStore,
    "SapBapiCustomerGetdetail1"];
    NBElement data = Action["SapPePersonaldataNew"];
    NBElement sapid = key.FirstChild;
    response = CreateCrmCustomer(data, sapid.ValueAsString);
    }
else if (key.Name.ToLower().StartsWith("sapbapicustomercreatefromdata1"))
    {
    NBElement custdet = key.FirstChild;
    NBElement sapid = envRoot.LastChild;
    response = CreateCrmCustomer(custdet, sapid.ValueAsString);
    }
//construct the output message.
outputRoot.CreateLastChildUsingNewParser(NBParsers.XMLNSC.ParserName);
outputRoot["XMLNSC"].CreateFirstChild(null, "CrmOnlineCustomer");
outputRoot["XMLNSC"]["CrmOnlineCustomer"].CreateFirstChild(null,
"ActivityId").SetValue(response.ToByteArray());

#endregion UserCode

// Send the message to output terminal.
outTerminal.Propagate(outAssembly);
}
catch (Exception e)
{
//Create and construct an exception message
outputRoot.CreateLastChildUsingNewParser(NBParsers.XMLNSC.ParserName);
outputRoot["XMLNSC"].CreateFirstChild(null, "CrmOnlineCustomer");
outputRoot["XMLNSC"]["CrmOnlineCustomer"].CreateFirstChild(null,
"ExceptionDetails").SetValue(e.ToString());
//Put it to FailureTerminal
failureTerminal.Propagate(outAssembly);
}
}
}
}

```

4. Example 5-11 contains the code for adding a method to invoke database and CRM Online operations. Add the following code below the Evaluate method (added in Example 5-10 on page 142).

Example 5-11 Database and CRM Online operations method

```

/// <summary>
/// Creates a CRM Online Customer(Account,Contact)
/// </summary>
/// <param name="entityAttributes"></param>
/// <param name="custno"></param>
/// <returns></returns>
private Guid CreateCrmCustomer(NBElement entityAttributes, String custno)
{
    //Check if customer exist in Database.
    String dbCustAcc =
    dbOp.CheckDBRecordsFromSAP(entityAttributes["EMail"].ValueAsString);
    if (dbCustAcc != null && dbCustAcc.Length > 0)
    {

```

```

        custno = custno + "_DB" + custno;
    }
    //Create CrmOnline customer.
    return crmOrg.CreateCrmCustomerRelation(entityAttributes, custno);
}

```

5. Save all files by pressing Ctrl+Shift+S, and build the project by selecting **Build** → **Build Solution**.

5.3 Deploying the message flow

Deployment is the process of transferring data to an execution group on a broker so that it can take effect in the broker domain. Message flows and associated resources are packaged in broker archive (BAR) files for deployment.

Creating a broker archive (BAR) file

To create a BAR file:

1. In the Message Broker Toolkit, select **File** → **New** → **Message Broker Archive**.
2. Enter the name of your server project, or select one from the displayed list. The list is filtered to only show projects in the active working set. Enter a name for the BAR file that you are creating, as shown in Figure 5-22. Click **Finish**.

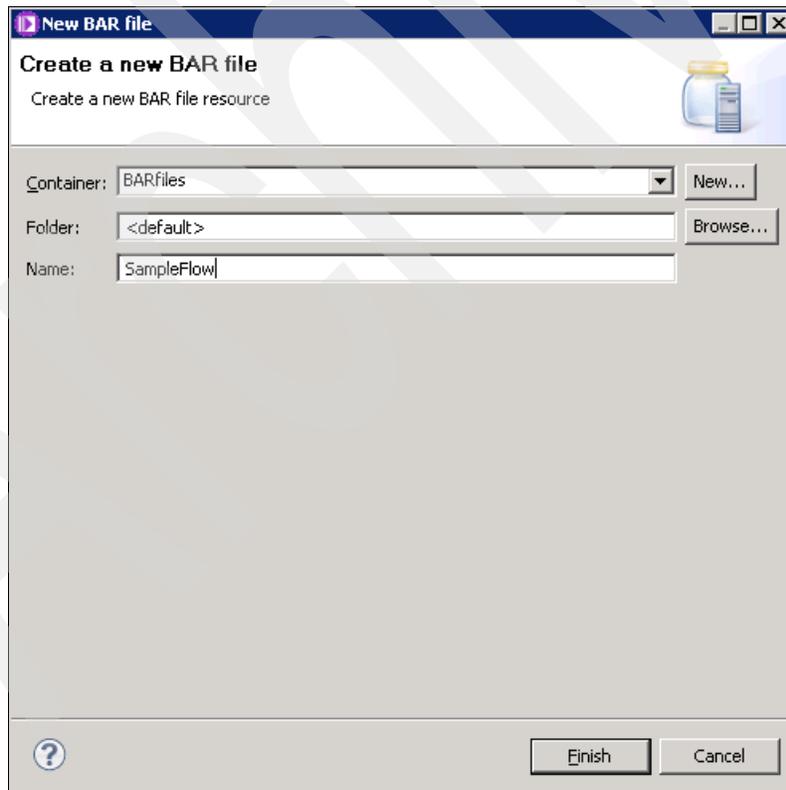


Figure 5-22 Create a new BAR file

3. A file with a .bar extension is created and is displayed in the Broker Administration Navigator view under the Broker Archives folder. See Figure 5-23 on page 145.

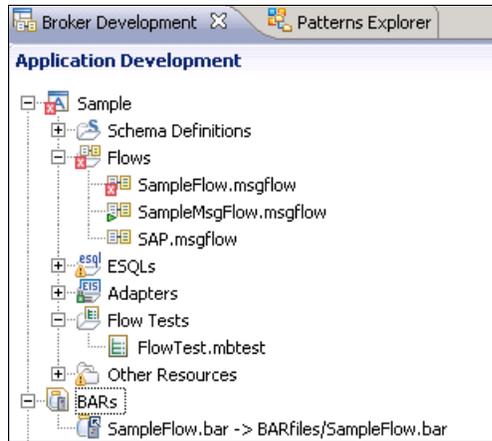


Figure 5-23 New BAR file

4. Double-click the **SampleFlow.bar** file, and in the Content view, select the check box for the Sample application, as shown in Figure 5-24. Click **Build and Save** to add the BAR file.

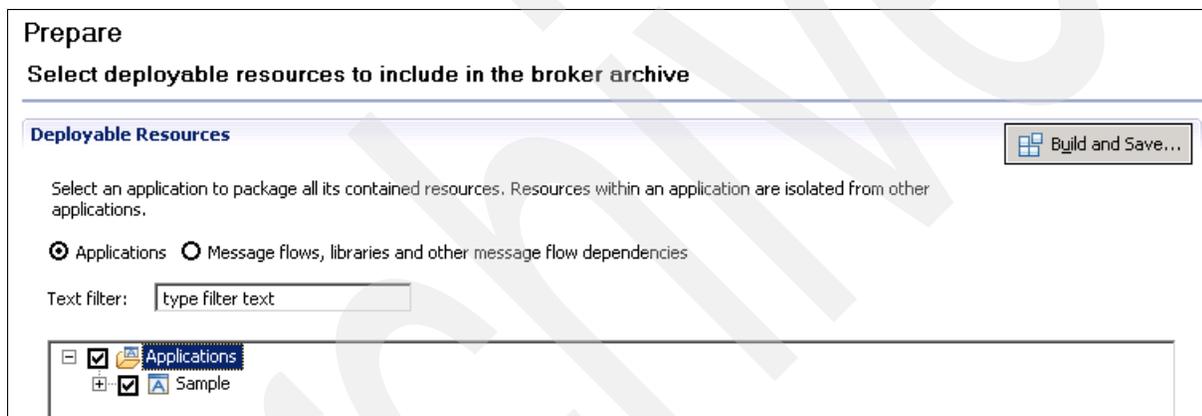


Figure 5-24 Select deployable resources

5. Save and close the bar file.

Deploying the message flow

To deploy the message flow.

1. Right-click the **SampleFlow.bar** file, and select **Deploy**, as shown in Figure 5-25 on page 146.

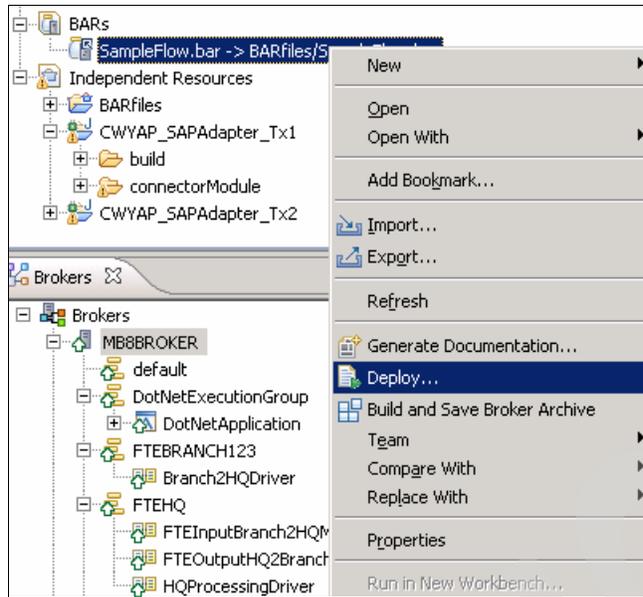


Figure 5-25 Deploy the BAR file

2. Select the broker and execution group to where the BAR is to be deployed. In our scenario, we selected the default, as shown in Figure 5-26. Click **Finish**.

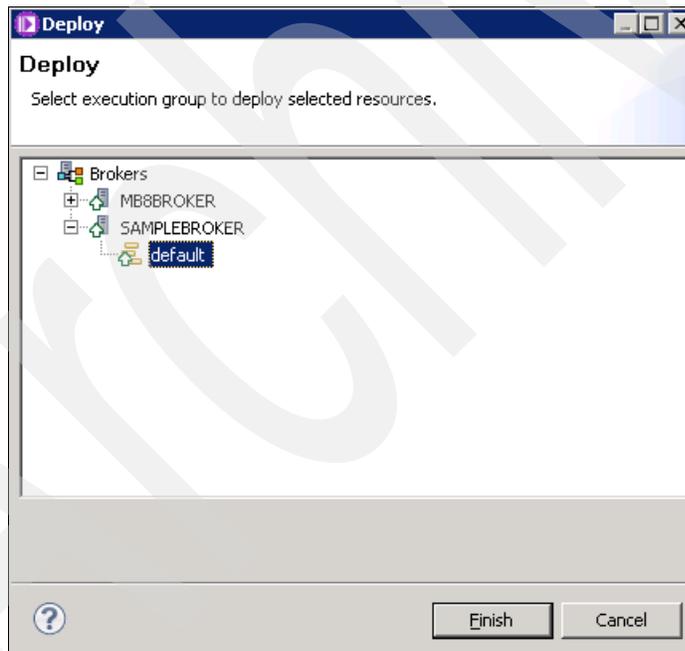


Figure 5-26 Select the execution group for the BAR file

When the deployment is complete, the flow is running, and you can test the flow.

5.4 Testing the message flow

We will create a test client to verify that the message flow is showing the expected behavior and that it is creating a CRM Online customer either by retrieving data from SAP or from user input.

Our scenario demonstrates two options:

- ▶ A graphical user interface application designed using Windows Forms that can collect the user input and triggers the message flow by putting the user details as a WebSphere MQ message onto the FlowInput node.
- ▶ Alternatively you can test the message by simply adding a Flow test project on the Message Broker Toolkit. This can only serve as a unit-testing facility and is not scalable.

The first option is recommended for a scalable and full-length application for triggering message flows and consuming message flow outputs.

5.4.1 Windows Forms Project: GUI Based

In this section, we build a front end GUI for customers who want to subscribe for a service on the Online stores website. The GUI is built using a Visual C# Windows Forms Application project.

Creating a new Visual Studio project

Open Visual Studio and use the following procedure to create a new project:

1. Select **File** → **New** → **Project**:
 - a. In the Recent Templates column, select **Visual C#**, and in the list of templates, select **Windows Forms Application**.
 - b. In the Name field, enter `SubscribeToService`, and click **OK**. The new form will open for edit.
2. In the Solution Explorer, right-click **Form1.cs**, and select **Rename**. Enter `SubscribeToService.cs` as the new name.
3. Use the Visual Studio Designer to create this form. You can access the Designer by selecting the `SubscribeToService.cs` file in the Solution Explorer, and clicking the **View Designer** button.

Use the Toolbox palette on the left side of the screen to add elements to the Form. If you do not see the Toolbox, select **View** → **Toolbox**.

The form has two tabs, as shown in Figure 5-27 on page 148.

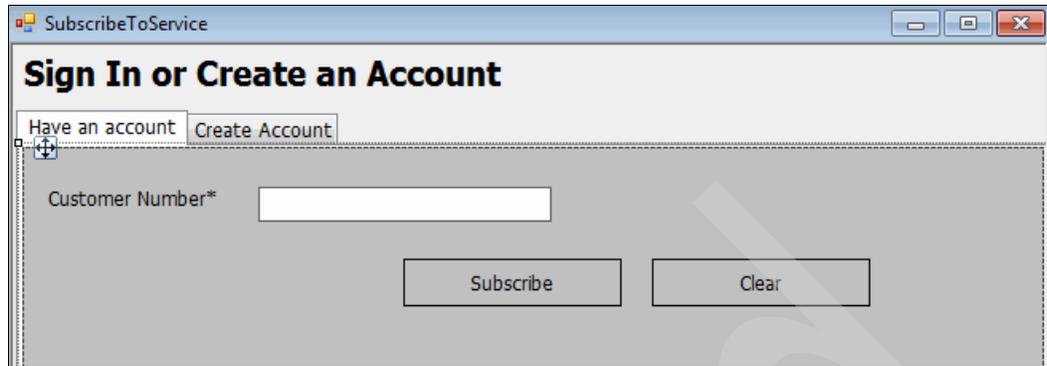


Figure 5-27 Form tabs

This step adds the tabs to the form:

- a. In the Toolbox, expand the Containers section, and drag TabControl to the form. Size the new box appropriately.
- b. Open the Windows Form Properties (**View** → **Properties Window**). From the Control's (topmost) drop-down list, select **tabControl1**.
- c. Select the **TabPages** property in the Behavior section, and click the (Collection) input button as shown in Figure 5-28.

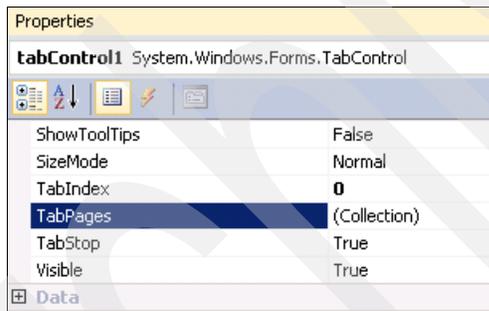


Figure 5-28 TabPages property

- d. This will open the Collection Editor. Select **tabPage1**, and change the following properties, as shown in Figure 5-29:

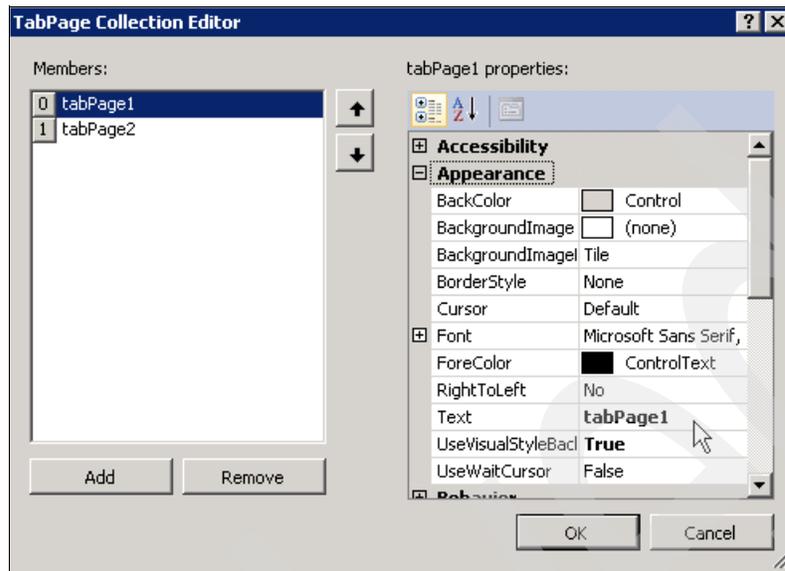


Figure 5-29 Collection Editor

- In the Appearance section, find the Text property, and enter Have an account as the value.
 - In the Design section, find the (Name) property, and enter tbCustomerExist as the value.
- e. Select **tabPage2**, and change the (Name) property value to tbNewCustomer and the value for the Text property to Create Account.
 - f. Click **OK**.
4. Build the **Have an account** tab. Select the tab on the form, and drag-and-drop the necessary control objects into the appropriate locations to create the Label and Text box for the Have an account tab, as shown in Figure 5-27 on page 148.

Perform the following steps to create the form:

- a. Expand the Common Controls section in the Toolbox, and drag a Label and a Textbox to the top of the form to allow the user to input the customer account number:
 - Enter Customer Number* as the text for the label.
 - Enter txtCustNo as the (Name) property for the textbox

The visible attributes of the control objects are set in the properties pane of Visual Studio, including the names of the objects. If the properties pane is not open and visible, select one of the control objects, and right-click to choose **Properties**.

- b. Add a button to the form, and use the Properties view to set the following properties:
 - Text: Subscribe
 - (Name): btnExistCustSub
- c. Add a second button to the form, and use the Properties view to set the following properties:
 - Text: Clear
 - (Name): btnExistCustClear

5. Build the Create Account tab. Select the tab on the form, and drag-and-drop the necessary control objects into the appropriate locations to create the Label and Text box for the tab, as shown in Figure 5-30.

Figure 5-30 New customer sign in panel

Perform the following steps to create the elements of the tab:

- a. Add a Label and a Textbox at the top of the form for each of the input fields shown in Figure 5-30, and set the (Name) property for each text box, as shown in the following list.

Table 5-4 Textbox labels and Name properties

Label	Textbox (Name) property
Title	cbtitle
FirstName	tbNewCustFName
LastName	tbNewCustLName
Email	tbNewCustEmail
Phone	tbNewCustPh
Address1	tbNewCustAddr1
Address2	tbNewCustAddr2
City	tbNewCustCity
State	tbNewCustSt
Country	tbNewCustCountry

Label	Textbox (Name) property
PostalCode	tbNewCustZip

- b. Add a button to the form, and use the Properties view to set the following properties:
 - Text: Subscribe
 - (Name): btnNewCustSubscribe
- c. Add a second button to the form, and use the Properties view to set the following properties:
 - Text: Clear
 - (Name): btnNewCustClear

Save the design.

5.4.2 Writing the code for the **SubscribeToService** form

This application uses WebSphere MQ .NET classes to create an MQ message that carries the customer information (customer number or personal information) as an XML message in the XMLNSC format to the FlowInput queue.

After the application places the message on the input queue, it polls the output queue for the output message. When the message flow completes execution, it places an MQ message on the queue with the CRM customer record identifier as the activityid. The application will pick up this message and display a message box to show the activityid that can be used to trace the current interaction.

The WebSphere MQ connection parameter resource names can be provided using an App.Config file, and the WebSphere MQ operation will be performed in a new class called MQOperations.cs.

App.Config

To add a new application configuration file to your Windows forms project:

1. Right-click the project in the Solution Explorer and then select **Add** → **New item**. Select **Application Configuration file**. Keep App.config as the name, and click **Add**. The new App.config file will open.
2. Replace the generated XML with the XML shown in Example 5-12. This XML will be used to store the WebSphere MQ connection information.

Example 5-12 Application configuration file for storing connection and resource properties

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<appSettings>
<add key="QmgrName" value="SAMPLEQM"/>
<add key="ConnName" value="localhost(5555)"/>
<add key="Channel" value="SYSTEM.DEF.SVRCONN"/>
<add key="InPutQ" value="OUTPUT_MSGS_Q"/>
<add key="OutPutQ" value="INPUT_MSGS_Q"/>
</appSettings>
</configuration>
```

3. Save and close App.config.

WebSphere MQ operations

To create the MQOperations class:

1. Add a new class to your Windows forms project called `MQOperations.cs`.
2. Add a reference to the project for the WebSphere MQ .NET Client library - `amqmdnet.dll`:
 - a. Right-click **References** in the Solution Explorer, and select **Add Reference**.
 - b. Use the Browse tab to find `MQ_install\bin\amqmdnet.dll`. Click **OK**. For example:
C:\Program Files (x86)\IBM\WebSphere MQ\bin\amqmdnet.dll
3. Add a reference (from the .NET tab) to `System.Configuration`.
4. Add the following using statement to your class file.

```
using IBM.WMQ;
using System.Configuration;
```
5. Replace the generated code for the MQOperations class shown in Example 5-13 to perform to the MQOperations class.

Example 5-13 MQQueueManager code

```
/// <summary>
/// Performs all the Websphere Operations.
/// </summary>
internal class MQOperations
{
    //WebSphere MQ Resources
    MQQueueManager QMgr = null;
    MQQueue QueueToPut = null;
    MQQueue QueueToGet = null;
    MQMessage subscriptionMsg = new MQMessage();
    MQPutMessageOptions pmo = new MQPutMessageOptions();
    MQGetMessageOptions gmo = new MQGetMessageOptions();
    //local variables
    String qmgrName = null;
    String connName = null;
    String channel = null;
    String inputQ = null;
    String outputQ = null;

    /// <summary>
    /// Constructor.
    /// </summary>
    public MQOperations()
    {
        ReadValuesFromConfig();
        try
        {
            QMgr = new MQQueueManager(qmgrName, channel, connName);
            QueueToPut = QMgr.AccessQueue(outputQ, MQC.MQOO_OUTPUT + MQC.MQOO_INQUIRE);
            QueueToGet = QMgr.AccessQueue(inputQ, MQC.MQOO_INPUT_SHARED + MQC.MQOO_INQUIRE);
        }
        catch (Exception e)
        {
            throw e;
        }
    }

    /// <summary>
    /// Read values from the Config file.
```

```

/// </summary>
private void ReadValuesFromConfig()
{
    if (ConfigurationManager.AppSettings.Count > 0)
    {
        qmgrName = ConfigurationManager.AppSettings["QmgrName"];
        connName = ConfigurationManager.AppSettings["ConnName"];
        channel = ConfigurationManager.AppSettings["Channel"];
        inputQ = ConfigurationManager.AppSettings["InPutQ"];
        outputQ = ConfigurationManager.AppSettings["OutputQ"];
    }
}

/// <summary>
/// PUT a FlowInput message and then pool for FlowOutput message.
/// </summary>
/// <param name="msg"></param>
/// <returns></returns>
public String SendBrokerRequest(String msg)
{
    String result = null;
    subscriptionMsg.Persistence = MQC.MQPER_PERSISTENT;
    subscriptionMsg.CharacterSet = 437;
    subscriptionMsg.WriteString(msg);
    pmo.Options = MQC.MQPMO_SYNCPOINT + MQC.MQPMO_SYNC_RESPONSE;

    //Put the message.
    QueueToPut.Put(subscriptionMsg, pmo);
    QMgr.Commit();
    //Pool for reply.
    while (true)
    {
        if (QueueToGet.CurrentDepth > 0)
        {
            MQMessage response = new MQMessage();
            gmo.WaitInterval = MQC.MQEI_UNLIMITED;
            gmo.Options = MQC.MQGMO_WAIT + MQC.MQGMO_SYNCPOINT;

            //We have a reply now, get it.
            QueueToGet.Get(response, gmo);

            if (response != null)
            {
                //Send back msg to SubscribeToService.
                result = response.ReadString(response.MessageLength);
                break;
            }
        }
    }
    return result;
}
}

```

6. Save MQOperations.cs.

This code performs the following operations:

1. Creates an instance of the WebSphere MQ .NET queue manager class (which results in a connection to the queue manager).
2. Creates an instance of the WebSphere MQ .NET Queue class for the INPUT_MSGS_Q, OUTPUT_MSGS_Q, and Error queues, which results in opening the queues for PUT/GET operations.
3. When the SubscribeToService form's **Subscribe** button is clicked, MQOperations is invoked, which constructs a message with user data and puts it on a queue to trigger the broker message flow.
4. Waits for an output message and when it arrives, gets the message and displays the result in a message window.

Windows Forms: SubscribeToService.cs

Form actions are based on the events fired when one of the buttons (**Subscribe** or **Clear**) are clicked on their respective tabs. There are event handlers for buttons to perform the actions.

During the form load event, we instantiate the MQOperations class, which connects to a queue manager and opens the resources (queues) under it.

The button click event is fired whenever a **Subscribe** button is clicked. In the event handler, we construct XML data based on the user input and invoke the method on MQOperations to put the XML data as a message on the queue and to wait for a reply.

The button click event is fired whenever a **Clear** button is clicked and clears all of the textbox contents on its respective tab:

1. Double-click the form design to open the SubscribeToService.cs code.
2. Add the following using statements:

```
using System.Xml;
using System.Xml.Linq;
```
3. Replace the SubscribeToService class code with the code in Example 5-14. Note that the names assigned to the button on the form correspond to the methods in the code.

Example 5-14 Event handler code

```
/// <summary>
/// Codebehind for the form to handle events from it.
/// </summary>
public partial class SubscribeToService : Form
{
    MQOperations mqop = null;

    public SubscribeToService()
    {
        InitializeComponent();
    }

    /// <summary>
    /// Clear all the Textbox entries on tab.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void btnNewCustClear_Click(object sender, EventArgs e)
    {
```

```

foreach (Control obj in this.tbNewCustomer.Controls)
{
if (obj is TextBox)
{
((TextBox)obj).Clear();
}
}
}
/// <summary>
/// Clear all the Textbox entries on the tab.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnExistCustClear_Click(object sender, EventArgs e)
{
foreach (Control obj in tbCustomerExist.Controls)
{
if (obj is TextBox)
{
((TextBox)obj).Clear();
}
}
}
/// <summary>
/// Existing Customer subscribed for the service. Call the flow.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void btnExistCustSub_Click(object sender, EventArgs e)
{
try
{
String result = "Request Submitted!, Activity Number = ";
Form current = this.FindForm();
current.UseWaitCursor = true;

XElement request = new XElement("SapBapiCustomerGetdetail1", new
XElement("Customerno", txtCustNo.Text.Trim()));
String xml = mqop.SendBrokerRequest(request.ToString());

XmlDocument doc = new XmlDocument();
doc.LoadXml(xml);
result = result + doc.DocumentElement.InnerText;

current.UseWaitCursor = false;
MessageBox.Show(result, "Submitted", MessageBoxButtons.OK);
current.UseWaitCursor = false;

btnExistCustClear_Click(null, null);
}
catch (Exception ex)
{
MessageBox.Show(ex.ToString());
}
}
}

```

```

    }
    }

    /// <summary>
    /// Form load event.
    /// Perform MQ Init operation during form load.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void SubscribeToService_Load(object sender, EventArgs e)
    {
        try
        {
            mqop = new MQOperations();
        }
        catch (Exception ei)
        {
            MessageBox.Show(ei.ToString());
        }
    }

    /// <summary>
    /// New Customer subscribed for the Service. Call the Flow.
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    private void btnNewCustSubscribe_Click(object sender, EventArgs e)
    {
        try
        {
            Form current = this.FindForm();
            current.UseWaitCursor = true;

            XElement request = new XElement("SapBapiCustomerCreatefromdata1",
            new XElement("SapPiPersonaldata1389002469",
            new XElement("Firstname", tbNewCustFName.Text.Trim()),
            new XElement("Lastname", tbNewCustLName.Text.Trim()),
            new XElement("Street", tbNewCustAddr1.Text.Trim()),
            new XElement("City", tbNewCustCity.Text.Trim()),
            new XElement("PostlCod1", tbNewCustZip.Text.Trim()),
            new XElement("Country", tbNewCustCountry.Text.Trim()),
            new XElement("Region", tbNewCustSt.Text.Trim()),
            new XElement("LanguP", "EN"),
            new XElement("LangupIso", "EN"),
            new XElement("Currency", "USD"),
            new XElement("SapPiCopyreference",
            new XElement("Salesorg", "0001"),
            new XElement("DistrChan", "01"),
            new XElement("Division", "01"),
            new XElement("RefCustmr", "0000000011")));

            String msg = mqop.SendBrokerRequest(request.ToString());

            current.UseWaitCursor = false;
            MessageBox.Show(msg, "Submitted", MessageBoxButtons.OK);
        }
    }
}

```

```
btnNewCustClear_Click(null, null);
}
catch (Exception exx)
{
    MessageBox.Show(exx.ToString());
}
}
}
```

The front end application is now ready.

5.4.3 Running the test

Use the following procedure to build and execute the `SubscribeToService` project and to test the Message Broker's flow:

1. Build the `SubscribeToService` project by selecting **Build** → **Build Solution**. Check the output to ensure that there are no errors.
2. Execute the Project by pressing `Ctrl+F5`. This shortcut runs the code in Visual Studio without the debugger. The form will open for input.
3. On the GUI, chose the relevant input tab, either **Have an Account** or **Create Account**.
4. Provide the input. For an existing customer, you need a valid SAP ID to test. Talk to your administrator to get a valid existing SAP customer number.
5. Provide the customer number in the `Customerno` input field, and click **Subscribe**.
6. A message box displays indicating the completion of the operation. If successful, it will display the CRM online activity number. If not, exception details are provided to assist you in recovering from the failure.
7. You can log on to your CRM Online account, and verify if a new account with primary contact with relevant details is created and the data is accurate.
8. If it is a new customer, complete the form, and click **Subscribe** to create a new SAP customer and a CRMOnline customer.
9. A message box displays indicating the completion of the operation. If successful, it displays the CRM online activity number. If not, exception details are provided to assist you in recovering from the failure.
10. Closing the form ends the application execution.

5.4.4 Using the test client from the Message Broker Toolkit

In this section, we create a Message Broker test client on the tool kit to test the flow we just constructed. The steps for creating the Test client are:

1. In the Message Broker Toolkit, right-click the message flow, and select **New** → **Message Broker Test Client**, as shown in Figure 5-31 on page 158.

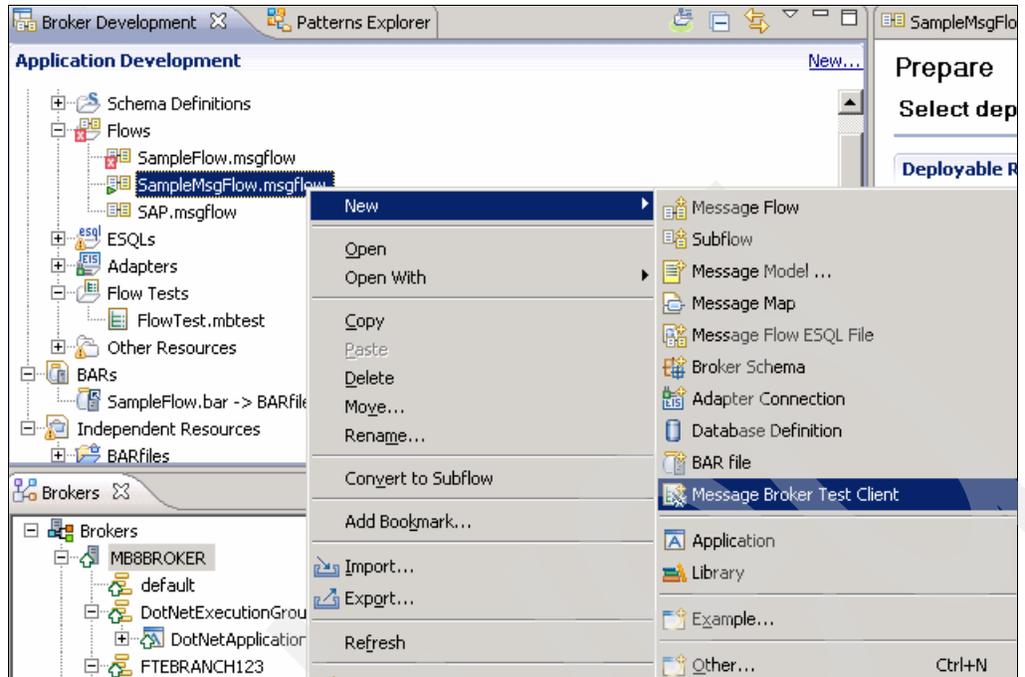


Figure 5-31 Create a Message Broker Test Client

2. Enter a name for the test client. In our scenario, we use **FlowTest**, as shown in Figure 5-32. Click **Finish**.

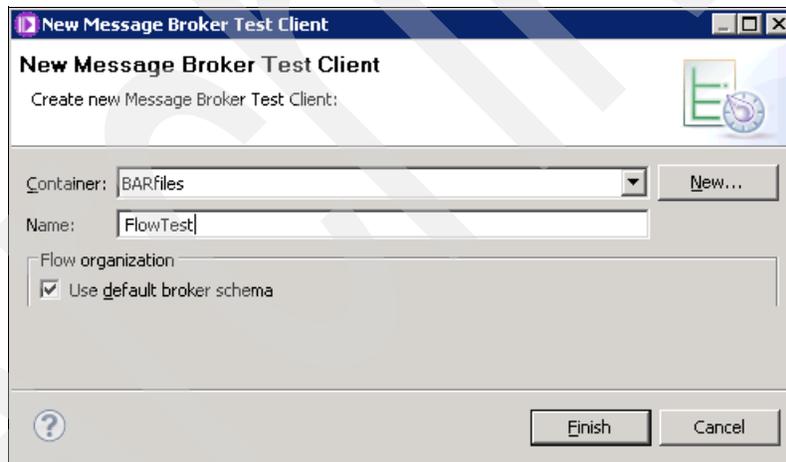


Figure 5-32 Specify a name for the Test Client

3. A new component called Flow Tests and a test client called FlowTest.mbttest are created, as shown in Figure 5-33 on page 159.

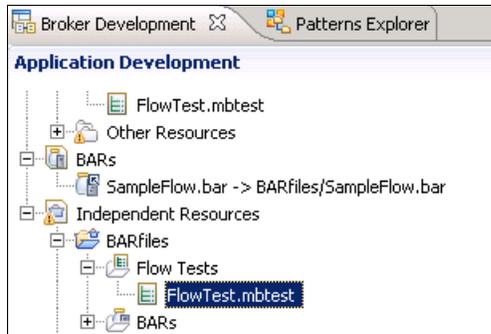


Figure 5-33 New flow test client

4. Double-click **FlowTest.mbstest** to open an Events Page to add the message flow test events. See Figure 5-34.

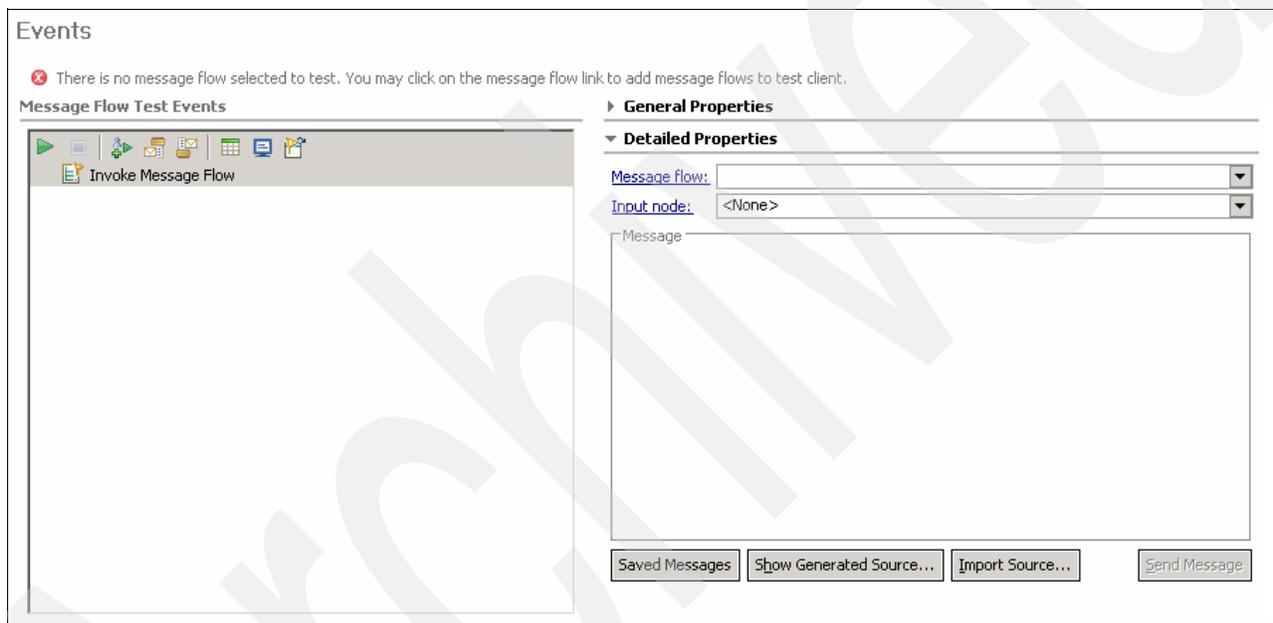


Figure 5-34 Add message flow test events

5. Click the **Enqueue** button (Figure 5-35) to add an Enqueue event, which will put a message on to the WebSphere MQ queue.



Figure 5-35 Enqueue button

6. Click the **Dequeue** button (Figure 5-36 on page 160) three times to add three Dequeue events to retrieve the messages from the queue.



Figure 5-36 Dequeue button

The results are shown in Figure 5-37.

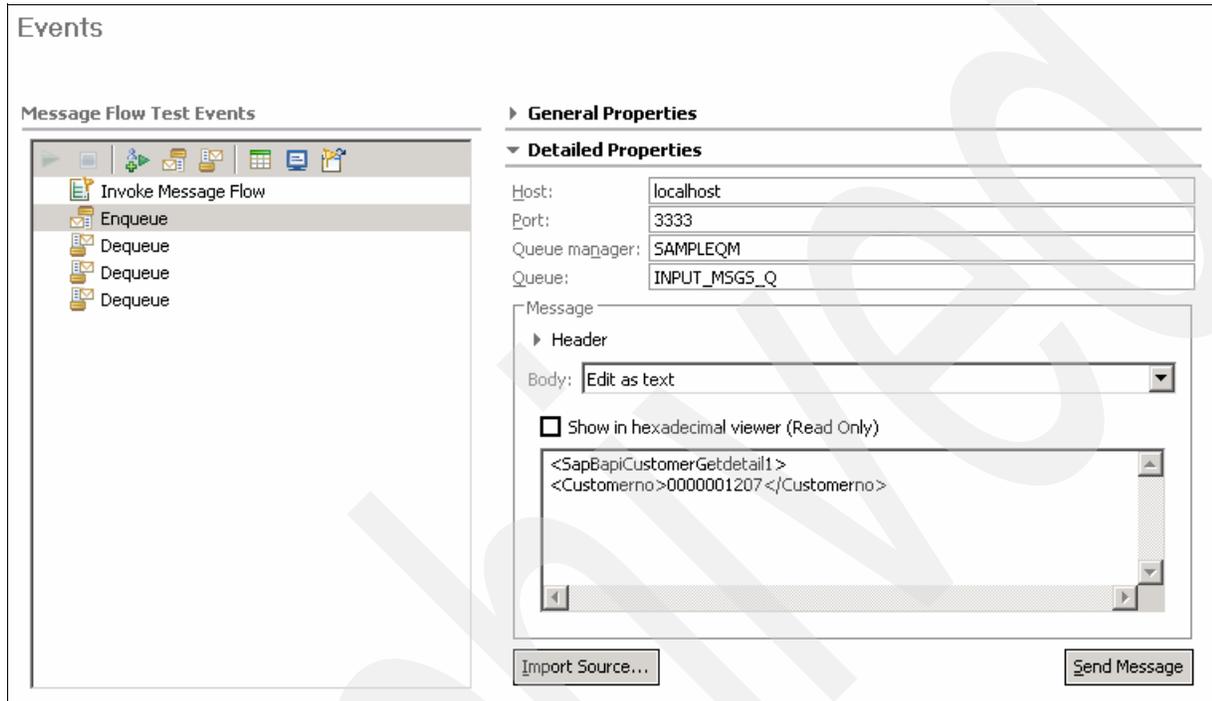


Figure 5-37 Define Enqueue and Dequeue properties

7. Select each Enqueue and Dequeue event, and configure it with the connection and queue information.

There are four events:

- Enqueue: Put an input message into INPUT_MSGS_Q
- Dequeue: Get flow output from OUTPUT_MSGS_Q
- Dequeue: Get Flow Error message from FLOW_ERROR_MSGS
- Dequeue: Get SAP Error message from SAP_ERROR_MSGS

For all the events, only the queue names are different. The rest of the properties remain the same.

8. Now you are ready to test.

Example 5-15 shows sample input data in the case of an existing customer.

Example 5-15 Sample Input data in case of Existing Customer

```
<SapBapiCustomerGetdetail1>
<Customerno>000000</Customerno>
</SapBapiCustomerGetdetail1>
```

Example 5-16 on page 161 shows sample input data in the case of a new customer.

Example 5-16 Sample Input data in case of New Customers

```
<SapBapiCustomerCreatefromdata1>
  <SapPiPersonaldata1389002469>
    <Firstname>xxxx</Firstname>
    <Lastname>xxxx</Lastname>
    <Street>xxxx Rd</Street>
    <City>xxxx</City>
    <PostlCod1>xxxx</PostlCod1>
    <EMail>xxxx@xxxx.com</EMail>
    <Country>xxxx</Country>
    <Region>xxxx</Region>
    <LanguP>xxxx</LanguP>
    <LangupIso>xxxx</LangupIso>
    <Currency>xxxx</Currency>
  </SapPiPersonaldata1389002469>
  <SapPiCopyreference>
    <Salesorg>xxxx</Salesorg>
    <DistrChan>xxxx</DistrChan>
    <Division>xxxx</Division>
    <RefCustmr>xxxxxx</RefCustmr>
  </SapPiCopyreference>
</SapBapiCustomerCreatefromdata1>
```

Select **Enqueue**, copy the sample data into the message box, and click **SendMessage** to trigger the message flow.

9. Use the WebSphere MQ Explorer to monitor the output queues for messages. After a message arrives in any output queue, select **Dequeue** for the relevant queue, and click **Get Message** to display the message.

5.5 Troubleshooting tips

In this section, we discuss some common errors that you might encounter and some tips to overcome them.

Always check if there are any messages in the error queues. Often message data contains exception details that are straightforward, and any issues can be fixed quickly.

Message flow fails to build

If the message flow fails to build:

1. Ensure that all of the properties are correctly set on the nodes. All mandatory properties are suffixed with an asterisk (*). Failing to set any such property causes a message flow build error.
2. Ensure that the queues, assemblies, and adapter connection parameters set on the node properties are valid and can be accessed. If a network resource is used, ensure that you have necessary access to it.
3. Ensure that methods, classes, and module names provided in the node properties match those in the code behind them.
4. Ensure that all in and out terminals are correctly connected.
5. Ensure that all the project prerequisites are met, for example, to use CRM Online you must have installed Windows Identity Foundation.

Message flow fails during deployment

If the message flow fails during deployment:

1. Ensure that the .NETCompute node's assembly property is set to a valid, existing assembly.
2. Ensure that all of the dependencies of the .NET assembly are accessible. The broker tries to load all of the dependency libraries for the given assembly.
3. Ensure that your SAP system is accessible and that you provided a valid user ID and password.
4. Ensure that you set mqsichangeparams and mqsisetdbparams on the broker after creating the SAP Adapter connection.
5. Ensure that there are no unresolved Schemas within the project in the Schema tree.

Failure in SAP nodes

If your message flow experiences a failure in one of the SAP nodes:

1. Ensure that the element names in the message match the element names in the schema. For example Customerno in <Customerno> must match the element name from SapBapiCustomerGetdetail1.xsd in the schema. You might also want to check that the names and mappings are correct in the schema file.
2. Each of the SAP BAPI requests have a few mandatory fields for querying. For example, Firstname, City, PostalCode, and so on are mandatory fields to create a new customer record. Ensure that all of the fields are provided. Refer to SAP BAPI documentation for SapBapiCustomerCreatefromdata1, SapBapiCustomerChangefromdata1, and SapBapiCustomerGetdetailsfromdata1 for a complete list of mandatory fields.
3. Each of the fields in a BAPI request message, such as firstname and lastname in the xsd files under Schema, have fixed lengths. Ensure that the input is within the defined limits.
4. If querying the BOR for BAPI interface runs endlessly, there is either a huge amount of data to discover and network latency is causing delays or you did not apply a correct filter on BOR before querying it. It is recommended to query BOR with a filter; otherwise, it will try to discover all of the objects in the SAP system.

Failure in a .NETCompute node

If you have a failure in a .NETCompute node:

1. Ensure that you provided a relevant class name without a .cs extension in the node properties.
2. Ensure that the database is active and accessible and that the connectionString can trace it.
3. Ensure that you have a valid Windows Live ID that was used to create a CRM Online account.
4. Ensure that you provided the correct CRM Online server name.
5. Ensure that the SAPRetrieve node is providing valid customer details. If not, the .NET node might fail while parsing the SAP output.

Scenario: Integration Windows Communication Foundation in message flows - Part 1

Windows Communication Foundation (WCF) is a distributed application interface (API) that is part of Microsoft's .NET framework. WCF allows independent applications to interoperate across a network through a message passing interface.

In this chapter, we introduce WCF through the implementation of an example service for processing insurance claims. This WCF service is consumed by a WebSphere Message Broker message flow to expose the WCF service to a wide range of enterprise applications as part of an Enterprise Service Bus (ESB). The message flow will be created in Chapter 7, "Scenario: Integrating Windows Communication Foundation in message flows - Part 2" on page 273.

In this chapter, we show you how to implement a WCF service, how to automatically generate WCF client code, how to alter the configuration of the bindings to change how the underlying messaging used by WCF acts, and we also discuss various hosting options that are available for WCF services.

Additional materials: The .NET class code for this scenario can be downloaded from the IBM Redbooks publication web site. See Appendix A, "Additional material" on page 485 for more information.

6.1 ClaimsProcessingWcfService overview

The scenario developed in this chapter (Figure 6-1) represents an application used by a fictional insurance company to process insurance claims. The insurance company provides insurance policy products from a range of financial partners transparently under a single brand.

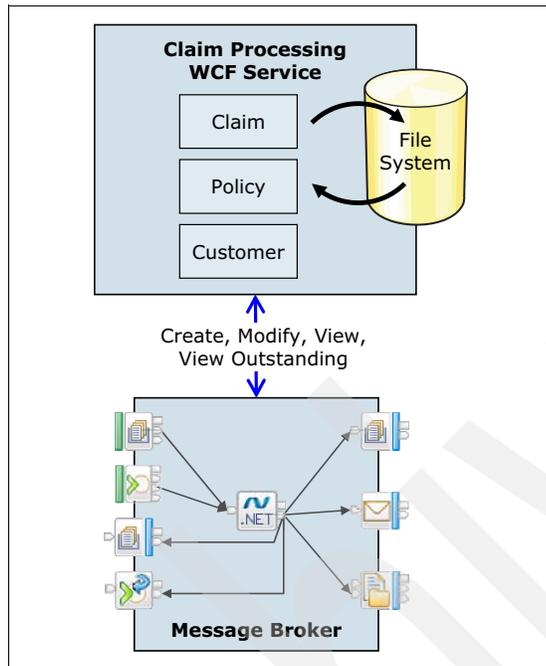


Figure 6-1 The ClaimsProcessingWcfService Overview

The system consists of several business objects that operations can be performed on, which are:

Customers

A logical representation of a customer in the system. These can be customers who actually bought an insurance policy or family members who were added to insurance policies owned by other customers.

Policies

When a customer purchases an insurance policy, all of the information associated with the insurance policy is stored in a Policy object. This includes standard information about the policy, such as the value covered under the policy and any excess that a customer needs to pay when making a claim.

Claims

If a customer needs to make a claim under their policy, perhaps because an accident occurred or their insured asset, such as a vehicle was damaged, they will contact the insurance company. This results in a Claim object being raised that contains all of the details of the claim, such as a description of what happened and the total cost of the claim. A claim can be approved or rejected by agents working at the insurance company.

Payments

If a claim is approved by an agent working for the insurance company, a payment needs to be issued to the customer. Because the insurance company offers a range of products from various financial partners, each Payment object must be sent to the appropriate partner company. The Payment object therefore contains details about both

the customer who is the recipient of the payment and the financial partner who is the underwriter of the payment.

These business objects are kept in a persistent store so that details can be accessed from multiple applications and physical locations. For the purpose of this example, a file system is used as the persistent store; however, in a real enterprise application, it is more likely that a database is used.

When an operation is performed on one of these business objects, the WCF service stores the updated state of the objects and generates appropriate responses to return to the requesting application. The operations supported are:

ViewClaim	Given a claim reference number, returns all of the details associated with a specific Claim object.
ViewPolicy	Given a specific policy reference number, returns all of the details associated with a specific Policy object
ViewOutstandingClaims	Returns the details for all claims that are not yet approved or rejected by an agent.
CreateClaim	Creates a new Claim object in the system. This is used when a customer contacts the insurance company to make a claim against their insurance.
CreatePolicy	Creates a new Policy object in the system. This operation is invoked when a new customer buys a policy from the insurance company.
AcceptClaim	If all of the documentation associated with a claim is present and the claim is determined to be within the scope of the policy, an agent can approve the claims, resulting in a payment being made to the customer.
RejectClaim	If the claim is determined to not be under the scope of the policy or if the documentation attached to the claim is not sufficient, an agent can choose to reject a claim, resulting in no payment to the customer.

6.2 Windows Communication Foundation

Windows Communication Foundation is a framework provided by the Microsoft .NET platform that provides a set of Application Programming Interfaces (APIs) for building network-distributed applications. The WCF programming model is designed to make it easy to implement a service-oriented architecture (SOA) where applications are loosely coupled and communicate through a message passing interface. The WCF API is contained in the `System.ServiceModel` namespace and is available in .NET framework 3.0 and higher.

Note: Because WebSphere Message Broker supports .NET framework 4.0, this chapter covers this version of the framework. Some of the features discussed might vary on lower versions of the .NET platform.

WCF differentiates between two key application types:

- ▶ A service, which provides some reusable unit of business logic that is exposed to other enterprise components. For example, in this chapter you develop a WCF service that exposes various operations for processing insurance claims.

- ▶ A client, which is an application that consumes a service.

Using an SOA-based approach, composite services can be constructed to act as clients of multiple services, exposing rich business functions to the end users of the distributed application.

When developing applications that are distributed in the network, it is important that the communication between clients and services is loosely coupled so that, for example, changes to the service implementation do not require changes to all of the clients consuming the service. This is particularly important in an enterprise where the original author of the service might not even know about all consumers of the service. The WCF programming model ensures that communication between client and service takes place through a defined interface or contract that allows this loose coupling. Further, clients and services do not even need to be developed in the same programming language to interoperate. In some cases, when using web services based bindings, the client does not even need to be a .NET application.

Another common pain point when developing distributed applications is dealing with the transport mechanism. In traditional application development, the choice of underlying transport mechanism is integrated into the application logic itself. This makes it challenging to either integrate the application with diverse clients or to change the underlying transport layer without developing a significant amount of plumbing or infrastructure. In WCF, the choice of underlying transport is abstracted away from the programming model and is determined through the configuration of the application at runtime. This means that the developer of the WCF services or clients does not need to deal with any transport layer code at all. This also has the advantage that the choice of transport can be altered after development of the application, meaning that the deployment strategy does not need to be known up front during development.

6.2.1 WCF ABCs

Although clients and services are loosely coupled, they still need to know enough about each other to interoperate. These details are typically referred to as an *ABC specification*. ABC is a mnemonic for the following three critical pieces of information: address, binding, and contract.

Address

The client needs to know the location of the service to send messages to it. An address is a URI of the form `scheme://<host>:[port]/path` with the following components:

Scheme	A prefix corresponding to the transport being used, for example, the <code>http://</code> prefix indicating a HTTP based transport.
Host	The host name, IP address, or machine name of the system hosting the WCF service.
Port	The port the services is listening on. This parameter is optional because some transports have default ports and others do not use them at all.
Path	The path to the WCF service.

Binding

The binding specifies the transport to be used to communicate between the client and the service. This includes the network protocol and the mechanism for encoding and decoding information about the wire and any specific configuration options. WCF includes the bindings shown in Table 6-1 on page 167.

Table 6-1 Bindings provided by the .Net framework

Binding Name	Description
BasicHttpBinding	SOAP 1.1 Messages over HTTP.
BasicHttpContextBinding	Same as BasicHttpBinding but with additional support for HTTP Cookies
WsHttpBinding	SOAP 1.1 Messages over HTTP with support for additional WS standards, such as WS-Addressing and WS-ReliableMessaging standards.
WsHttpContextBinding	Same as WsHttpBinding but with additional support for sending context information using SOAP headers.
WsDualHttpBinding	Same as WsHttpBinding but suitable for duplex message exchange patterns.
WSFederationBinding	SOAP over HTTP with support for the WS-Federation standard.
NetTcpBinding	TCP/IP based communication with a compact binary message encoding.
NetTcpContextBinding	Same as NetTcpBinding but with additional support for passing context information in headers.
NetNamedPipeBinding	Named Pipe based communication using a compact binary message encoding.
NetPeerTcpBinding	Peer-to-Peer based binding using TCP/IP based wire protocol.
NetMsmqBinding	Binding for enabling MQMS based communication between WCF Applications.
MsmqIntegrationBinding	Binding for integrating WCF applications with existing MSMQ applications.
WebHttpBinding	HTTP based binding using XML message encoding instead of SOAP.

Contract

The contract defines the programmatic interface between the client and the service. As a whole this is referred to as the service contract and is composed of two main parts:

Operation contract The operations exposed by the service. Each operation is equivalent to a method on the service that can be called to invoke some piece of business logic. For example, in this scenario one of the operations defined will be ApproveClaim, which is responsible for approving an outstanding insurance claim and generating a payment to the customer.

Data contract The data contract details which types of data the client and service need to be able to understand to interoperate with each other. In WCF, standard objects can be annotated as part of the data contract so that the framework automatically establishes how these are transmitted between client and service.

6.2.2 WCF operation

After the ABC specification is defined between a client and a service, the WCF Framework can produce a proxy object for the service that the client can use. The proxy object exposes the operations defined in the service contract as method calls. This might look like a Remote Procedure Call (RPC) based model; however, the WCF framework uses a message-passing interface to implement the actual communication.

WCF uses a serializer called the DataContract serializer to convert data sent between the client and the service. Depending on the binding being used, WCF might use a SOAP-based XML format or a binary format on the wire. The important thing to note is that this is completely transparent to both the client and the service. Figure 6-2 shows a high-level architecture diagram of a typical WCF application.

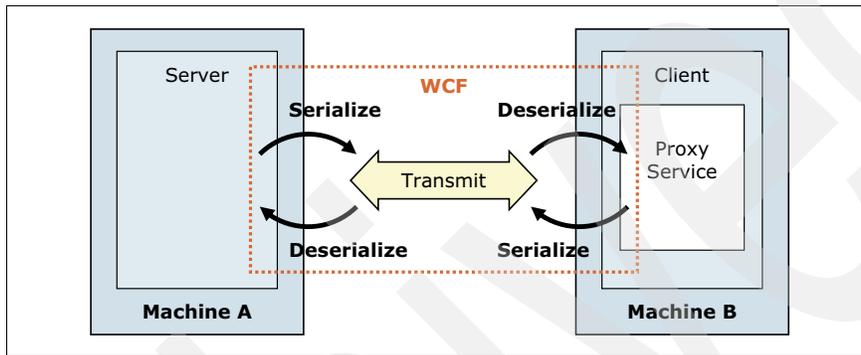


Figure 6-2 WCF service and client architecture overview

6.3 Developing the WCF Service

In this section, you develop a WCF Service that processes policies and claims passed to it from WebSphere Message Broker in its capacity as an Enterprise Service Bus (ESB). You define the service contract that specifies the operations that can be called by a remote .NET caller and the data contract that defines how data is passed back and forth between the WCF Service and the remote WCF client application. You also implement the service itself using flat files as a data storage medium.

6.3.1 Creating a Microsoft Visual Studio 2010 project for the WCF Service

Visual Studio includes several Project templates for developing WCF Services, which automatically reference the System.ServiceModel and System.Runtime.Serialization namespaces required by WCF. Some sample source files and configuration files are also created that are refactored as the service is implemented.

To create a Visual Studio project:

1. Launch Visual Studio.
2. Click **File** → **New Project**.
3. In the dialog shown in Figure 6-3 on page 169, select **Visual C#** → **WCF** and then **WCF Service Library**.
4. Fill in a name for the service and the locations you will be using to save the project and solution files.

The name of the project in this scenario is ClaimsProcessingWcfService.

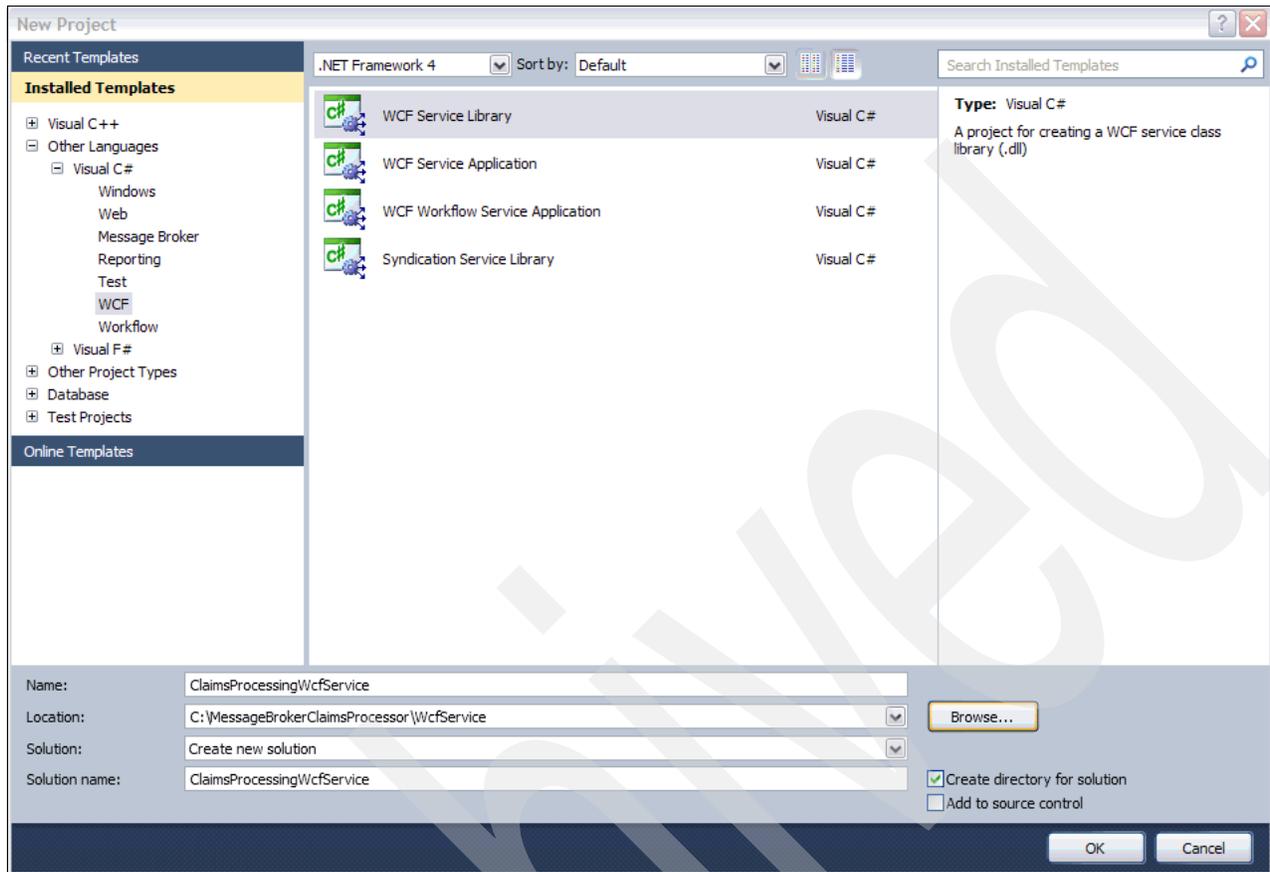


Figure 6-3 New Project dialog

5. Click **OK**.

When you finish, you can see, in the Solution Explorer view, the new project created along with the following default files:

- ▶ App.config
- ▶ IService1.cs
- ▶ Service1.cs

6.3.2 Setting the namespace for the project

To avoid clashes with other .NET assemblies that might be referenced by the solution, set a unique namespace for a project. This namespace, by convention, starts with the name of the company as the root followed by the name of the technology. For this example, the namespace we use is `Ibm.Broker.Example.ClaimsProcessing`.

By default the New Project Wizard creates a default namespace for the project based on the name of the solution. In section 6.3.3, "Modifying the classes created by the New Project Wizard" on page 170, the namespace of existing files is updated to the new value. To ensure that any new resources created in this project are in the correct namespace, set the default namespace for the project by following these steps:

1. In the Solution Explorer, select **ClaimsProcessingWcfService**.
2. Right-click the icon, and select **Properties**.

3. Select the **Application** tab to view the Application Properties.
4. In the Default namespace field, enter `Ibm.Broker.Example.ClaimsProcessing`, as shown in Figure 5.

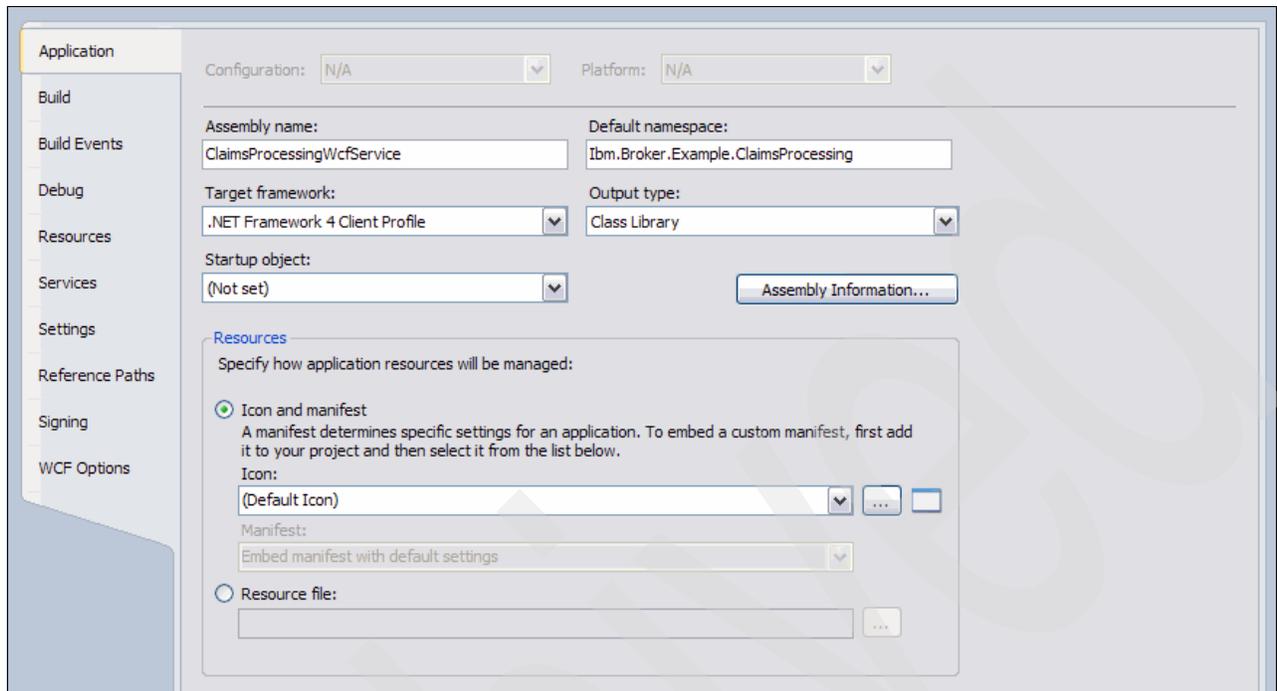


Figure 6-4 Setting the default namespace for the project

5. Save and close the properties file.

6.3.3 Modifying the classes created by the New Project Wizard

By default the New Project Wizard creates a WCF Service called `Service1` and an interface called `IService1`. To increase readability and maintainability of the solution, rename these classes with more descriptive names.

Updating the interface class

To rename the `IService1` interface:

1. In the Solution Explorer, double-click **IService1.cs** to load the file in the editing window:
 - a. Highlight the text `IService1`, in the class declaration, on the line shown in Example 6-1.

Example 6-1 IService1 interface declaration

```
public interface IService1
```

- b. Right-click, and from the context menu, select **Refactor** → **Rename**.
 - c. In the Rename dialogue, enter `IClaimsProcessingWcfService` in the New Name text box.
 - d. Uncheck the Preview reference changes check box.
 - e. Click **OK**.
2. The Refactor menu will not automatically rename the file, so right-click **IService1.cs** in the Solution Explorer, and select **Rename**.

3. Enter `IClaimsProcessingWcsService.cs`, and then press Enter to accept the changes.
4. Alter the namespace of the `IClaimsProcessingWcfService` interface by changing the namespace declaration in the file to match Example 6-2.

Example 6-2 A namespace declaration of `Ibm.Broker.Example.ClaimsProcessing`

```
namespace Ibm.Broker.Example.ClaimsProcessing
```

5. Save the interface file.

Renaming the service class

To rename the `Service1` class:

1. In the Solution Explorer, double-click **Service1.cs** to load the file in the editing window:
 - a. Highlight the **Service1** in the class declaration, on the line shown in Example 6-3.

Example 6-3 Service1 class declaration

```
public class Service1 : IService1
```

- b. Right-click, and from the context menu, select **Refactor** → **Rename**, as shown in Figure 6-5.

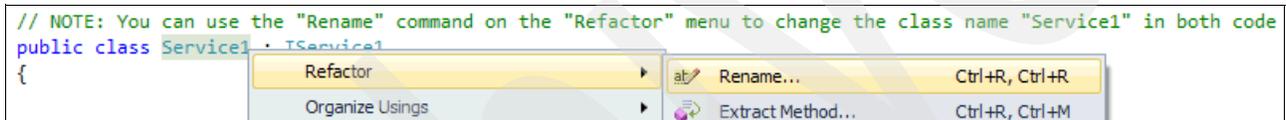


Figure 6-5 Using the Refactor menu to rename the `Service1` class

- c. In the Rename dialogue, enter `ClaimsProcessingWcfService` in the New name text box.
 - d. Uncheck the Preview reference changes check box. One of the advantages of using the Refactor menu is that any references to `Service1` in the project are automatically updated. By unchecking this box all changed references are automatically accepted. Ensure that the completed dialogue box matches the example in Figure 6-6.

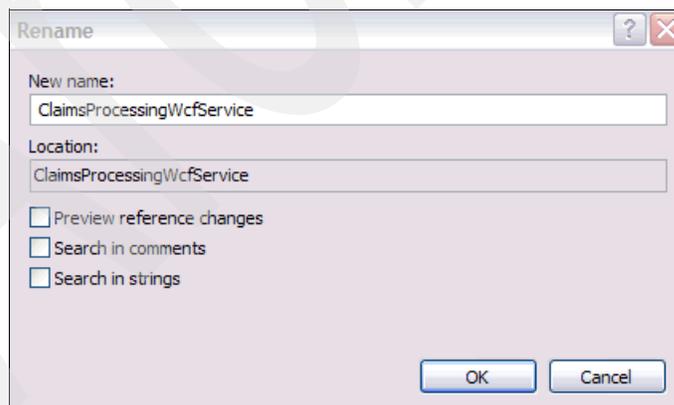


Figure 6-6 Rename dialog

- e. Click **OK**.
 2. The Refactor menu does not automatically rename the file, so right-click `Service1.cs` in the Solution Explorer, and select **Rename**.

3. Enter `ClaimsProcessingWcfService.cs`, and then press `Enter` to accept the changes.
4. Alter the namespace of the `ClaimsProcessingWcfService` class by changing the namespace declaration in the file to match Example 6-4.

Example 6-4 A namespace declaration of `Ibm.Broker.Example.ClaimsProcessing`

```
namespace Ibm.Broker.Example.ClaimsProcessing
```

5. Save the file.

The New Project Wizard generates a comment in both the interface and the service class, indicating that they must be renamed. Because this is completed, you can remove the comment shown in Example 6-5 from both `ClaimsProcessingWcfService.cs` and `IClaimsProcessingWcfService.cs`.

Example 6-5 Generated comment

```
// NOTE: You can use the "Rename" command on the "Refactor" menu to change the  
interface name "IService1" in both code and config file together.
```

Notice that code is generated for some default WCF operations. This code is removed when you define the `DataContract` and `Service Operations` for the service in section 6.3.6, “Designing and implementing the service contract” on page 184.

Updating App.Config

The last step is to update the service name in `App.config` file with the new namespace:

1. In the Solution Explorer, double-click `App.config` to open it in the editor.
2. Replace the service name with the following syntax:

```
<service name="Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService">
```

6.3.4 Creating the required datatypes

Much of the information used by the WCF service is common across operations. It is therefore beneficial to create custom datatypes for some of the common objects that the service will be operating upon. The following datatypes are required for this service:

► **Customer**

This datatype is used to hold all of the required data that represents a customer in the system. The customer is identified throughout the system by a unique ID number.

► **InsurancePolicy**

The `InsurancePolicy` Datatype represents an insurance policy that was purchased by a Customer. This might be one of several predefined policy offerings, identified in this example by the combination of a `Policy Name` and a `Policy Provider`. A policy is uniquely identified in the system by a `Policy Number`.

► **Claim**

This datatype represents a claim made by a Customer against their insurance Policy. A Claim is uniquely identified by a `Claim Reference` number and can be approved or rejected by an agent working for the insurance broker.

► **Payment**

When a Claim is approved, this results in a Payment being generated to the Customer. This payment is processed by another system; however, the WCF service is responsible for generating the information required to make the payment.

Customer Datatype

To create the Customer Datatype:

1. Select **Project** → **Add Class**.
2. In the dialog box, enter `Customer.cs` for the Name field.
3. Click **Add**.
4. A new file named `Customer.cs` is added to the project. The class is automatically created in the WCF Service's name space, and the class is opened in the editor. Replace the generated code with the code shown in Example 6-6 to `Customer.cs`.

Example 6-6 Customer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.Collections;
using System.Xml.Serialization;
using System.Collections.ObjectModel;

namespace IBM.Broker.Example.ClaimsProcessing
{
    [DataContract]
    [XmlRoot("Customer")]
    public class Customer
    {
        //members
        private string _FirstName;
        private string _Surname;
        private Collection<String> _OtherNames;
        private string _Address;
        private string _CustomerId;
        private string _Email;

        public Customer()
        {
            _OtherNames = new Collection<String>();
        }
        [DataMember]
        public string FirstName
        {
            get { return _FirstName; }
            set { _FirstName = value; }
        }
        [DataMember]
        public string Surname
        {
            get { return _Surname; }
        }
    }
}
```

```

        set { _Surname = value; }
    }
    [DataMember]
    [XmlArrayItem("Name")]
    public Collection<String> OtherNames
    {
        get { return _OtherNames; }
        set { _OtherNames = value; }
    }
    [DataMember]
    public string Address
    {
        get { return _Address; }
        set { _Address = value; }
    }
    [DataMember]
    public string CustomerId
    {
        get { return _CustomerId; }
        set { _CustomerId = value; }
    }
    [DataMember]
    public string Email {
        get { return _Email; }
        set { _Email = value; }
    }
    public Boolean Equals(Customer customer)
    {
        if (customer.CustomerId == CustomerId &&
            customer.FirstName == FirstName &&
            customer.Surname == Surname &&
            customer.OtherNames.Count == OtherNames.Count)
        {
            foreach (string Name in OtherNames)
            {
                if (!customer.OtherNames.Contains(Name)) return false;
            }
            return true;
        }
        return false;
    }
}
}

```

5. Save Customer.cs.

Note that the class is marked with the [DataContract] attribute, and the properties are marked with the [DataMember] attribute. These attributes are discussed in more depth in section “Data contracts” on page 185.

Similarly, the class has an [XmlRoot] attribute, and certain attributes have additional attributes relating to the System.Xml.Serialization namespace. These attributes specify details of how these datatypes are serialized to Xml and are discussed in more depth in section “XML serialization attributes” on page 181.

Also note that because this class overrides the boolean `Object.Equals()` method so that two customers can be compared for equality. In this case two customers are treated as equal if their customer IDs match and they have the same first and last names and the complete list of other names matches in any order.

Policy Datatype

To create the Policy Datatype:

1. Select **Project** → **Add Class**.
2. In the dialog box, enter `InsurancePolicy.cs`.
3. Click **Add**.
4. A new file named `InsurancePolicy.cs` is added to the project. The class is automatically created in the WCF Service's name space.

Replace the generated code in `InsurancePolicy` with the code shown in Example 6-7.

Example 6-7 InsurancePolicy.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.Xml.Serialization;
using System.Collections.ObjectModel;

namespace IBM.Broker.Example.ClaimsProcessing
{
    [DataContract][XmlRoot("Policy")]
    public class InsurancePolicy
    {
        // members
        private string _PolicyNumber;
        private Customer _Policyholder;
        private Collection<Customer> _NamedParties;
        private string _PolicyName;
        private int _PolicyCover;
        private int _PolicyExcess;
        private DateTime _PolicyStartDate;
        private DateTime _PolicyExpiryDate;
        private String _Underwriter;

        public InsurancePolicy()
        {
            _NamedParties = new Collection<Customer>();
        }
        [DataMember]
        public string PolicyNumber
        {
            get { return _PolicyNumber; }
            set { _PolicyNumber = value; }
        }
        [DataMember]
        public Customer Policyholder
        {
```

```

        get { return _Policyholder; }
        set { _Policyholder = value; }
    }
    [DataMember][XmlArrayItem("Customer")]
    public Collection<Customer> NamedParties
    {
        get { return _NamedParties; }
        set { _NamedParties = value; }
    }
    public void AddNamedParty(Customer customer)
    {
        _NamedParties.Add(customer);
    }
    [DataMember]
    public string PolicyName
    {
        get { return _PolicyName; }
        set { _PolicyName = value; }
    }
    [DataMember]
    public int PolicyCover
    {
        get { return _PolicyCover; }
        set { _PolicyCover = value; }
    }
    [DataMember]
    public int PolicyExcess
    {
        get { return _PolicyExcess; }
        set { _PolicyExcess = value; }
    }
    [DataMember]
    public DateTime PolicyStartDate
    {
        get { return _PolicyStartDate; }
        set { _PolicyStartDate = value; }
    }
    [DataMember]
    public DateTime PolicyExpiryDate
    {
        get { return _PolicyExpiryDate; }
        set { _PolicyExpiryDate = value; }
    }

    [DataMember]
    public String Underwriter
    {
        get { return _Underwriter; }
        set { _Underwriter = value; }
    }
}
}
}

```

5. Save InsurancePolicy.cs.

Claim Datatype

To create the Claim Datatype:

1. Select **Project** → **Add Class**.
2. In the dialog box, enter Claim.cs.
3. Click **Add**.
4. A new file named Claim.cs is added to the project. The class is automatically created in the WCF Service's name space.

Replace the generated code in Claim.cs with the code shown in Example 6-8.

Example 6-8 Claim.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.Xml.Serialization;

namespace IBM.Broker.Example.ClaimsProcessing
{
    //Claim states
    [DataContract]
    public enum ClaimState
    {
        [EnumMember]
        Approved,
        [EnumMember]
        Rejected,
        [EnumMember]
        Submitted
    };

    [DataContract][XmlRoot("Claim")]
    public class Claim
    {
        //members
        private string _ClaimReference;
        private DateTime _ClaimReceivedDate;
        private DateTime _ClaimDate;
        private int _ClaimValue;
        private Customer _Customer;
        private string _Description;
        private string _PolicyNumber;
        private ClaimState _Status;
        private string _Notes;

        [DataMember]
        public string ClaimReference
        {
            get { return _ClaimReference; }
            set { _ClaimReference = value; }
        }
    }
}
```

```
[DataMember]
public DateTime ClaimReceivedDate
{
    get { return _ClaimReceivedDate; }
    set { _ClaimReceivedDate = value; }
}

[DataMember]
public DateTime ClaimDate
{
    get { return _ClaimDate; }
    set { _ClaimDate = value; }
}

[DataMember]
public int ClaimValue
{
    get { return _ClaimValue; }
    set { _ClaimValue = value; }
}

[DataMember]
public Customer Customer
{
    get { return _Customer; }
    set { _Customer = value; }
}

[DataMember]
public string Description
{
    get { return _Description; }
    set { _Description = value; }
}

[DataMember]
public string PolicyNumber
{
    get { return _PolicyNumber; }
    set { _PolicyNumber = value; }
}

[DataMember]
public ClaimState Status
{
    get { return _Status; }
    set { _Status = value; }
}

[DataMember]
public string Notes
{
    get { return _Notes; }
    set { _Notes = value; }
}
```

```
}  
}
```

5. Save Claim.cs.

This file also defines an enumeration named *ClaimState* that is used to hold the valid values of the Status property. Each member of the enumeration is marked with the [EnumMember] attribute that is required in order for the serializers used in the scenario to correctly serialize the Status property of Claim objects.

Payment Datatype

To create the Payment Datatype:

1. Select **Project** → **Add Class**.
2. In the dialog box, enter Payment.cs.
3. Click **Add**.
4. A new file named Payment.cs is added to the project. The class is automatically created in the WCF Service's name space.

Replace the generated code with the code shown in Example 6-9.

Example 6-9 Payment.cs

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Runtime.Serialization;  
using System.Xml.Serialization;  
  
namespace IBM.Broker.Example.ClaimsProcessing  
{  
    [DataContract]  
    [XmlRoot("Payment")]  
    public class Payment  
    {  
        // members  
        private Customer _Payee;  
        private int _Amount;  
        private string _Underwriter;  
  
        [DataMember]  
        public Customer Payee  
        {  
            get { return _Payee; }  
            set { _Payee = value; }  
        }  
  
        [DataMember]  
        public int Amount  
        {  
            get { return _Amount; }  
            set { _Amount = value; }  
        }  
    }  
}
```

```

[DataMember]
public string Underwriter
{
    get { return _Underwriter; }
    set { _Underwriter = value; }
}
}
}
}

```

5. Save Payment.cs.

6.3.5 Creating private methods to serialize and deserialize the Datatypes

To persist data between invocations, the WCF service reads and writes objects to a local filesystem. The process of writing an object from an in-memory representation to a stream of bytes suitable for being written to disk is called serialization. Similarly the process of converting a series of bytes from an input stream, such as a file into an in-memory representation of an object, is called deserialization.

In this example, objects are serialized using the XmlSerializer, which is part of the System.Xml.Serialization namespace. This means that data is written in XML format.

In memory, a Customer object called cust looks like the example in Figure 6-7.

Name	Value	Type
⊕ this	{Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessing}	Ibm.Br
⊖ cust	{Ibm.Broker.Example.ClaimsProcessing.Customer}	Ibm.Br
⊖ _Address	"1 Generic Street, Any Town, USA"	string
⊖ _CustomerId	"CUST00001"	string
⊖ _Email	"johndoe@itso.example.ibm.com"	string
⊖ _FirstName	"John"	string
⊖ _OtherNames	Count = 0x00000002	System
⊖ [0x00000000]	"James"	string
⊖ [0x00000001]	"Jeffrey"	string
⊕ Raw View		
⊖ _Surname	"Doe"	string
⊖ Address	"1 Generic Street, Any Town, USA"	string
⊖ CustomerId	"CUST00001"	string
⊖ Email	"johndoe@itso.example.ibm.com"	string
⊖ FirstName	"John"	string
⊖ OtherNames	Count = 0x00000002	System
⊖ [0x00000000]	"James"	string
⊖ [0x00000001]	"Jeffrey"	string
⊕ Raw View		
⊖ Surname	"Doe"	string

Figure 6-7 A Customer object viewed in the Visual Studio debugger

When serialized into an XML file on the file system, this same customer object can be represented as shown in Example 6-10.

Example 6-10 A Customer object serialized to XML.

```

<?xml version="1.0"?>
<Customer xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

```

```

<FirstName>John</FirstName>
<Surname>Doe</Surname>
<OtherNames>
  <Name>James</Name>
  <Name>Jeffrey</Name>
</OtherNames>
<Address>1 Generic Street, Any Town, USA</Address>
<CustomerId>CUST00001</CustomerId>
<Email>johndoe@itso.example.ibm.com</Email>
</Customer>

```

Here, each public property is represented as an XML element with a corresponding name and value. The `ArrayList<string> OtherNames` is represented as an XML element called `<OtherNames>` with a child element representing each member of the `ArrayList`.

XML serialization attributes

The `XmlSerializer` automatically serializes all public members. The representation produced by an `XmlSerializer` is controlled by a selection of attributes added to the class declaration and public properties. In this example, the default serialization scheme is used for most purposes, but the `[XmlAttribute]` attribute is used to set the name of the root element of the XML tree. This attribute takes an argument specifying the name of the root element, for example `[XmlAttribute(myRootElement)]`. In most cases, this is set to the same value as the class name itself; however, when you created the `InsurancePolicy` class, shown in Example 6-11, the `[XmlAttribute]` tag was set to create an XML root element named `Policy`.

Example 6-11 Setting the XML root to "Policy" in InsurancePolicy.cs

```

[DataContract][XmlAttribute("Policy")]
public class InsurancePolicy

```

A serialization of an `InsurancePolicy` object therefore produces XML output, as shown in Example 6-12.

Example 6-12 Serialized InsurancePolicy object

```

<?xml version="1.0"?>
<Policy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <PolicyNumber>POLICY0001</PolicyNumber>
  <Policyholder>
    <FirstName>John</FirstName>
    <Surname>Doe</Surname>
    <OtherNames>
      <Name>James</Name>
      <Name>Jeffrey</Name>
    </OtherNames>
    <Address>1 Generic Street, Any Town, USA</Address>
    <CustomerId>CUST00001</CustomerId>
    <Email>johndoe@itso.example.ibm.com</Email>
  </Policyholder>
  <NamedParties>
    <Customer>
      <FirstName>Sam</FirstName>
      <Surname>Doe</Surname>
      <OtherNames>

```

```

        <Name>Stephen</Name>
        <Name>John</Name>
    </OtherNames>
    <Address>1 Generic Street, Any Town, USA</Address>
    <CustomerId>CUST00002</CustomerId>
    <Email>samdoe@itso.example.ibm.com</Email>
</Customer>
<Customer>
    <FirstName>Jane</FirstName>
    <Surname>Doe</Surname>
    <OtherNames>
        <Name>Jill</Name>
        <Name>Jenny</Name>
    </OtherNames>
    <Address>1 Generic Street, Any Town, USA</Address>
    <CustomerId>CUST00003</CustomerId>
    <Email>janedoe@itso.example.ibm.com</Email>
</Customer>
</NamedParties>
<PolicyCover>1000</PolicyCover>
<PolicyExcess>150</PolicyExcess>
<PolicyStartDate>2012-03-22T13:51:58.640625+00:00</PolicyStartDate>
<PolicyExpiryDate>2013-03-22T13:51:58.640625+00:00</PolicyExpiryDate>
<Underwriter>Partner Insurance Company A</Underwriter>
</Policy>

```

Because the Customer datatype is itself serializable, any nested instances of these objects are included in the serialization of the InsurancePolicy object.

Creating serializers for datatypes

Each XmlSerializer object can serialize and deserialize one specific datatype; therefore, to persist Customer, Claim, and Policy objects to disk three different XmlSerializers is created for the WCF service. To reduce computational cost associated with recreating objects, these XmlSerializers are defined as private members of the ClaimsProcessingWcfService class and instantiated in the constructor.

Add the code shown in Example 6-13 to ClaimsProcessingWcfService.cs, starting at the end of the using statements.

Example 6-13 XmlSerializer declaration in ClaimsProcessingWcfService.cs

```

using System.IO;
using System.Xml.Serialization;
using System.Globalization;

private XmlSerializer CustomerSerializer;
private XmlSerializer PolicySerializer;
private XmlSerializer ClaimSerializer;

public ClaimsProcessingWcfService()
{
    CustomerSerializer = new XmlSerializer(typeof(Customer));
    PolicySerializer = new XmlSerializer(typeof(InsurancePolicy));
    ClaimSerializer = new XmlSerializer(typeof(Claim));
}

```

Serializing an object is performed by invoking the `serialize` method on a suitably typed `XmlSerializer` passing in a stream to write to and the object to be serialized. Similarly invoking the `deserialize` method on a suitably typed `XmlSerializer` returns the deserialized object.

Implementing the private methods for serialization and deserialization

To create a method to serialize `Customer` objects, add the code shown in Example 6-14 to `ClaimsProcessingWcfService.cs`.

Example 6-14 Private method for serializing Customer objects

```
private void WriteCustomerToFile(Customer CustomerToSave, String Dir)
{
    using(Stream fStream = new FileStream(Dir + "\\\" + CustomerToSave.CustomerId,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        CustomerSerializer.Serialize(fStream, CustomerToSave);
    }
}
```

In this method, a new `FileStream` is created based on the `Dir` input parameter concatenated with the `CustomerId`. So for example, if the string `C:\path\to\save` was passed in and the `customerId` was `CUST0001`, the object is serialized to a file called `C:\path\to\save\CUST0001`. Note that because the backslash character is an escape character, we must escape it when entering it into a string.

The `FileStream` object is created with the `FileMode.Create`, `FileAccess.Write`, and `FileShare.None` attributes to ensure that if the file does not exist it will be created. If the file already exists, it is overwritten, and the service has exclusive access to the file.

The `FileStream` class also implements the `IDisposable` interface, so when the using block finishes executing the stream, the underlying file is automatically closed.

To create private methods to serialize `InsurancePolicy` and `Claim` objects using the same pattern, add the code shown in Example 6-15 to `ClaimsProcessingWcfService.cs`.

Example 6-15 Private method for serializing InsurancePolicy and Claim objects

```
private void WritePolicyToFile(InsurancePolicy policyToSave, string dir)
{
    using (Stream fStream = new FileStream(dir + "\\\" + policyToSave.PolicyNumber,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        PolicySerializer.Serialize(fStream, policyToSave);
    }
}

private void WriteClaimToFile(Claim ClaimToSave, string Dir)
{
    using (Stream fStream = new FileStream(Dir + "\\\" + ClaimToSave.ClaimReference,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        ClaimSerializer.Serialize(fStream, ClaimToSave);
    }
}
```

To deserialize an object, a similar pattern is used; however, this time the stream is created using the `File.OpenRead` method, which automatically sets the appropriate `FileMode`, `FileAccess`, and `FileShare` options for reading. Again, scope of the `fStream` object is limited to a single block by the `using` statement to take advantage of the `IDisposable` interface.

Add the code shown in Example 6-16 to `ClaimsProcessing WcfService.cs`. This code creates methods to deserialize `Customer`, `InsurancePolicy`, and `Claim` objects.

Example 6-16 Private methods for deserializing Customer, InsurancePolicy and Claim objects

```
private Customer GetCustomerFromFile(string inputFile)
{
    using (Stream fStream = File.OpenRead(inputFile))
    {
        return (Customer)CustomerSerializer.Deserialize(fStream);
    }
}

private InsurancePolicy GetPolicyFromFile(String inputFile)
{
    using (Stream fStream = File.OpenRead(inputFile))
    {
        return (InsurancePolicy)PolicySerializer.Deserialize(fStream);
    }
}

private Claim GetClaimFromFile(String InputFile)
{
    using (Stream fStream = File.OpenRead(InputFile))
    {
        return (Claim)ClaimSerializer.Deserialize(fStream);
    }
}
```

6.3.6 Designing and implementing the service contract

WCF allows applications to communicate in a transport independent manner by defining a service contract between clients and services. This contract covers what operations are exposed by the service, the operation contract, and the data model to use when calling these operations, the data contract. The types of errors that the client must be prepared to deal with can also optionally be specified using a fault contract, and this is covered in more detail in “Defining the Fault types” on page 208.

Notice that after following the steps in 6.3.3, “Modifying the classes created by the New Project Wizard” on page 170, the New Project Wizard already marked the interface class `IClaimsProcessingWcfService` as defining a service contract by marking the class declaration with the `[ServiceContract]` attribute.

Operation contracts

The operation contract defines what operations the service supports. These are exposed as methods marked with the `[OperationContract]` attribute on the Interface specifying the service contract.

The New Project Wizard created two default operation contracts in `IClaimsProcessingWcfService` when you followed the steps in section 6.3.1, “Creating a

Microsoft Visual Studio 2010 project for the WCF Service” on page 168. The code in Example 6-17 shows the two generated methods marked with the [OperationContract] attribute: GetData and GetDataUsingDataContract.

Example 6-17 Operation contracts in the service interface

```
[OperationContract]
string GetData(int value);

[OperationContract]
CompositeType GetDataUsingDataContract(CompositeType composite);

// TODO: Add your service operations here
```

The GetData declaration specifies that the service must implement an operation called GetData, which takes an int as a parameter and returns a string. Because only primitive types are used in this operation, a data contract is not required.

The second GetDataUsingDataContract method specifies that the service must implement an operation called GetDataUsingDataContract, which takes a CompositeType object as a parameter and returns a CompositeType object to the caller. Because CompositeType is not a primitive type, a data contract is required so that the WCF classes can serialize and deserialize the customer type.

Data contracts

WCF uses the DataContractSerializer class to serialize and deserialize types passed between WCF clients and services. It does this using a data contract, which is an abstract description of the data types to be exchanged.

The type used by the GetDataUsingDataContract operation is automatically defined by the New Project Wizard and is located just below the IClaimsProcessingWcfService interface definition. The definition of the CompositeType class is shown in Example 6-18.

Example 6-18 Data contract

```
[DataContract]
public class CompositeType
{
    bool boolValue = true;
    string stringValue = "Hello ";

    [DataMember]
    public bool BoolValue
    {
        get { return boolValue; }
        set { boolValue = value; }
    }

    [DataMember]
    public string StringValue
    {
        get { return stringValue; }
        set { stringValue = value; }
    }
}
```

The class is marked with the [DataContract] attribute to indicate that the class implements a data contract. For the type to actually implement the data contract, the properties that form part of the DataContract must be marked with the [DataMember] attribute.

Implementation of a service contract

The New Project Wizard also generates a default implementation of the operation contract in the ClaimsProcessingWcfService.cs class. Example 6-19 shows the code that implements the service contract.

Example 6-19 The implementation of the automatically generated service contract

```
public string GetData(int value)
{
    return string.Format("You entered: {0}", value);
}

public CompositeType GetDataUsingDataContract(CompositeType composite)
{
    if (composite == null)
    {
        throw new ArgumentNullException("composite");
    }
    if (composite.BoolValue)
    {
        composite.StringValue += "Suffix";
    }
    return composite;
}
```

These methods implement the interface specified in IClaimsProcessingWcfService.cs and therefore also implement the service contract. Recall that to implement the interface, the calling signature and return type of the concrete method must match the definition in the interface.

Removing the automatically generated service contract and implementation

While the automatically generated code is useful when investigating WCF for the first time, these methods and datatypes are not required in the Claims Processing Application. Therefore, to keep the code clean and make it easier to maintain these, remove methods and datatypes from the project before proceeding.

To remove the automatically generated methods and datatypes from the IClaimsProcessingWcfService.cs and ClaimsProcessingWcfService files:

1. Open the file IClaimsProcessingWcfService.cs by double-clicking the icon in the Solution Explorer.
2. Highlight the code shown in Example 6-17 on page 185. Click the Delete key to remove the highlighted code.
3. Highlight the code shown in Example 6-18 on page 185. Click the Delete key to remove the highlighted code.

IClaimsProcessingWcfService.cs now looks like the code shown in Example 6-20 on page 187.

Example 6-20 IClaimsProcessingWcfService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [ServiceContract]
    public interface IClaimsProcessingWcfService
    {
    }
}
```

4. Save `IClaimsProcessingWcfService.cs`.
5. Select **ClaimsProcessingService.cs** from the Solution Explorer, and double-click it to open the implementation class.
6. Highlight the code shown in Example 6-19 on page 186. Click the Delete key to remove the highlighted code.

`ClaimsProcessingWcfService.cs` now looks like the example in Example 6-21.

Example 6-21 ClaimsProcessingWcfService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;
using System.IO;
using System.Xml.Serialization;
using System.Globalization;
using System.Collections.ObjectModel;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    public class ClaimsProcessingWcfService : IClaimsProcessingWcfService
    {
        private XmlSerializer CustomerSerializer;
        private XmlSerializer PolicySerializer;
        private XmlSerializer ClaimSerializer;

        public ClaimsProcessingWcfService()
        {
            CustomerSerializer = new XmlSerializer(typeof(Customer));
            PolicySerializer = new XmlSerializer(typeof(InsurancePolicy));
            ClaimSerializer = new XmlSerializer(typeof(Claim));
        }

        private Customer GetCustomerFromFile(string inputFile)
        {

```

```

        using (Stream fStream = File.OpenRead(inputFile))
        {
            return (Customer)CustomerSerializer.Deserialize(fStream);
        }
    }

    private void WriteCustomerToFile(Customer CustomerToSave, String Dir)
    {
        using (Stream fStream = new FileStream(Dir + "\\\" +
            CustomerToSave.CustomerId, FileMode.Create,
            FileAccess.Write, FileShare.None))
        {
            CustomerSerializer.Serialize(fStream, CustomerToSave);
        }
    }

    private InsurancePolicy GetPolicyFromFile(String inputFile)
    {
        using (Stream fStream = File.OpenRead(inputFile))
        {
            return (InsurancePolicy)PolicySerializer.Deserialize(fStream);
        }
    }

    private void WritePolicyToFile(InsurancePolicy policyToSave,
        string dir)
    {
        using (Stream fStream = new FileStream(dir + "\\\" +
            policyToSave.PolicyNumber, FileMode.Create,
            FileAccess.Write, FileShare.None))
        {
            PolicySerializer.Serialize(fStream, policyToSave);
        }
    }

    private Claim GetClaimFromFile(String InputFile)
    {
        using (Stream fStream = File.OpenRead(InputFile))
        {
            return (Claim)ClaimSerializer.Deserialize(fStream);
        }
    }

    private void WriteClaimToFile(Claim ClaimToSave, string Dir)
    {
        using (Stream fStream = new FileStream(Dir + "\\\" +
            ClaimToSave.ClaimReference,FileMode.Create,
            FileAccess.Write, FileShare.None))
        {
            ClaimSerializer.Serialize(fStream, ClaimToSave);
        }
    }

```

```

    }
}
}

```

7. Save ClaimsProcessingServiceWcf.cs.

Designing the Claims Processing operation contract

The Claims Processing service supports seven operation types:

- ▶ Create a new claim
- ▶ View an existing claim
- ▶ View outstanding claims
- ▶ Approve a claim
- ▶ Reject a claim
- ▶ View a policy
- ▶ Create a new policy

Each of these operations correspond to a WCF operation contract on the `IClaimsProcessingWcfService` interface. To add the operation contracts to the interface:

1. Open `IClaimsProcessingWcfService.cs`.
2. Modify the code by adding the additional method declarations shown in Example 6-22 after the following lines of code.

```

[ServiceContract]
public interface IClaimsProcessingWcfService
{

```

Example 6-22 The completed `IClaimsProcessingWcfService.cs` file with operation contracts

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [ServiceContract]
    public interface IClaimsProcessingWcfService
    {
        [OperationContract]
        [FaultContract(typeof(ClaimProcessingFault))]
        NewClaimResponse CreateClaim(NewClaimRequest request);

        [OperationContract]
        [FaultContract(typeof(ClaimProcessingFault))]
        ViewClaimResponse ViewClaim(ViewClaimRequest request);

        [OperationContract]
        [FaultContract(typeof(ClaimProcessingFault))]
        [FaultContract(typeof(ViewOutstandingClaimsIncompleteResultsFault))]

```

```

ViewOutstandingClaimsResponse
ViewOutstandingClaims(ViewOutstandingClaimsRequest request);

[OperationContract]
[FaultContract(typeof(ClaimProcessingFault))]
ApproveClaimResponse ApproveClaim(ApproveClaimRequest request);

[OperationContract]
[FaultContract(typeof(ClaimProcessingFault))]
RejectClaimResponse RejectClaim(RejectClaimRequest request);

[OperationContract]
[FaultContract(typeof(ClaimProcessingFault))]
ViewPolicyResponse ViewPolicy(ViewPolicyRequest request);

[OperationContract]
[FaultContract(typeof(ClaimProcessingFault))]
CreatePolicyResponse CreatePolicy(CreatePolicyRequest request);
}
}

```

This code generates errors in the Error list, which we resolve later in “Defining the Fault types” on page 208.

3. Save the file.

The `IClaimsProcessingWcfService.cs` file is now in its completed state. Note that as well as the previously discussed `[OperationContract]` attribute each operation is also decorated with the `[FaultContract]` attribute. This attribute specifies how failures in the service are communicated to the client consuming the service, which we discuss in more detail in “Defining the Fault types” on page 208.

Designing the data contract

In 6.3.4, “Creating the required datatypes” on page 172, several basic datatypes were defined that are already marked with the `[DataContract]` and `[DataMember]` attributes, specifying that they are part of the data contract for this service. The operation contracts defined in “Designing the Claims Processing operation contract” on page 189 do not use these datatypes; however, instead they use a specific data type for the request and response parameters for each operation.

Request and response base classes for data contracts

Many parameters are common across multiple request or response types, so to reduce code duplication, they all inherit from one of two base types. These base types are:

- ▶ `ClaimsProcessingRequest`

This class defines the properties shown in Table 6-2.

Table 6-2 ClaimsProcessingRequest Properties

Property	Purpose
<code>RequestDate</code>	A <code>DateTime</code> value indicating when the request was sent.
<code>ApplicationId</code>	An identifier for the application making the request.
<code>MessageId</code>	A unique identifier for this message.

Property	Purpose
TypeOfRequest	An enumeration value showing what specific type of request this object represents. Including the request type means that reflection is not required to determine the type of a derived class from a base class reference.

► ClaimsProcessingResponse

This class defines the properties shown in Table 6-3.

Table 6-3 ClaimsProcessingResponse Properties

Property	Purpose
ResponseTime	A DateTime indicated when the response was sent from the Claims Processing Service back to the client.
CorrelationId	An identifier allowing the client to correlate a request message with a response message. The CorrelationId for a response is set to the MessageId of the request which generated it.
TypeOfResponse	An enumeration value showing what specific type of response this object represents.

To create the base classes for the requests and responses, add the files shown in Example 6-23 and Example 6-24 on page 192 to your project using the **Project** → **Add class** dialog:

1. Add ClaimsProcessingResponse.cs, and replace the generated code with the code shown in Example 6-23.

Example 6-23 ClaimsProcessingResponse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace IBM.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public enum ResponseType
    {
        [EnumMember]
        ApproveClaim,
        [EnumMember]
        CreatePolicy,
        [EnumMember]
        NewClaim,
        [EnumMember]
        RejectClaim,
        [EnumMember]
        ViewClaim,
        [EnumMember]
        ViewOutstandingClaims,
    }
}
```

```

        [EnumMember]
        ViewPolicy,
        [EnumMember]
        Processing
    }

    [DataContract]
    public class ClaimsProcessingResponse
    {

        private DateTime _ResponseTime;
        private string _CorrelationId;

        [DataMember]
        public DateTime ResponseTime
        {
            get { return _ResponseTime; }
            set { _ResponseTime = value; }
        }

        [DataMember]
        public string CorrelationId
        {
            get { return _CorrelationId; }
            set { _CorrelationId = value; }
        }

        public ClaimsProcessingResponse()
        {
            _ResponseTime = DateTime.Now;
        }

        [DataMember]
        virtual public ResponseType TypeOfResponse
        {
            get { return ResponseType.Processing; }
            //required for WCF serialization
            private set { }
        }
    }
}

```

Tip: The property `TypeOfResponse` is marked as virtual so that it can be overridden by derived classes of this base type.

The `TypeOfResponse` is never set by the code. Ordinarily we can omit the setter for this property making it read only. However, for the serializer to work correctly, the property must have a setter. For this reason, the setter is declared private and does nothing.

2. Add `ClaimsProcessingRequest.cs`, and replace the generated code with the code shown in Example 6-24 on page 192.

Example 6-24 ClaimsProcessingRequest.cs

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public enum RequestType
    {
        [EnumMember]
        ApproveClaim,
        [EnumMember]
        CreatePolicy,
        [EnumMember]
        NewClaim,
        [EnumMember]
        RejectClaim,
        [EnumMember]
        ViewClaim,
        [EnumMember]
        ViewOutstandingClaims,
        [EnumMember]
        ViewPolicy,
        [EnumMember]
        Processing
    }

    [DataContract]
    [KnownType(typeof(ApproveClaimRequest))]
    [KnownType(typeof(CreatePolicyRequest))]
    [KnownType(typeof(NewClaimRequest))]
    [KnownType(typeof(RejectClaimRequest))]
    [KnownType(typeof(ViewClaimRequest))]
    [KnownType(typeof(ViewOutstandingClaimsRequest))]
    [KnownType(typeof(ViewPolicyRequest))]
    public abstract class ClaimsProcessingRequest
    {
        private DateTime _RequestDate;
        private string _ApplicationId;
        private string _MessageId;

        [DataMember]
        public DateTime RequestDate
        {
            get { return _RequestDate; }
            set { _RequestDate = value; }
        }

        [DataMember]
        public string ApplicationId
        {
            get { return _ApplicationId; }
            set { _ApplicationId = value; }
        }
    }
}

```

```

    [DataMember]
    public string MessageId
    {
        get { return _MessageId; }
        set { _MessageId = value; }
    }

    [DataMember]
    virtual public RequestType TypeOfRequest
    {
        get { return RequestType.Processing; }
        //required for WCF serialization
        private set {}
    }
}
}
}

```

The class declaration is decorated with a [KnownType] attribute so that the DataContractSerializer used by WCF recognizes the listed type when serializing or deserializing instances of this object. Without these attributes, WCF does not correctly populate base class properties on derived types.

After the base types are created, each derived type only needs to include additional properties available to that type and relevant to the particular operation.

Data contract for ApproveClaim

To approve a claim, the information shown in Table 6-4 must be provided to complete the request.

Table 6-4 ApproveClaimRequestProperties

Property	Purpose
ClaimReference	The unique ID of the claim to be approved.
ValueApproved	The value of claim approved.
AgentReference	An ID identifying the agent that approved this claim.
TypeOfRequest	Returns a constant value of RequestType.ApproveClaim to indicate that this is an ApproveClaimRequest.

Create a new class in your project named ApproveClaimRequest with the contents shown in Example 6-25.

Example 6-25 ApproveClaimRequest.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ApproveClaimRequest : ClaimsProcessingRequest

```

```

{
    [DataMember]
    public override RequestType TypeOfRequest
    {
        get { return RequestType.ApproveClaim; }
    }
    //ApproveClaimRequest
    private string _ClaimReference;
    private int _ValueApproved;
    private string _AgentReference;

    [DataMember]
    public string ClaimReference
    {
        get { return _ClaimReference; }
        set { _ClaimReference = value; }
    }

    [DataMember]
    public int ValueApproved
    {
        get { return _ValueApproved; }
        set { _ValueApproved = value; }
    }

    [DataMember]
    public string AgentReference
    {
        get { return _AgentReference; }
        set { _AgentReference = value; }
    }
}
}
}

```

When an approval request is submitted there are two possible outcomes. Either the service can reject the approval by throwing a fault or the system can accept the approval and send a response containing payment information to the requesting client. Handling faults is covered in more detail in “Defining the Fault types” on page 208.

The data contract for the ApproveClaimResponse datatype defines the properties shown in Table 6-5.

Table 6-5 ApproveClaimResponse properties

Property	Purpose
Payment	A serialized Payment object containing details of the payment to make to the customer filing the approved Claim.
TypeOfResponse	Returns a constant value of ResponseType.ApproveClaim to indicate that this is an ApproveClaimResponse.

Create a new class in the project named ApproveClaimResponse.cs with the content in Example 6-26.

Example 6-26 ApproveClaimResponse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ApproveClaimResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.ApproveClaim; }
        }

        private Payment _Payment;

        [DataMember]
        public Payment Payment
        {
            get { return _Payment; }
            set { _Payment = value; }
        }

        public ApproveClaimResponse(Payment payment)
        {
            _Payment = payment;
        }
    }
}
```

Data contract for Create Policy

The data contract for the CreatePolicyRequest defines the properties shown in Table 6-6 that are required when creating a new insurance policy in the system.

Table 6-6 CreatePolicyRequest Properties

Property	Purpose
PrimaryCustomer	The Customer who took out the new insurance policy being created.
NamedParties	Any named parties who are also covered by this policy.
PolicyName	The policy name, which corresponds to a product offering defined in the system.
PolicyCover	The total value insured under this policy.
PolicyExcess	The excess payable by the customer on any claim made under this policy.
TypeOfRequest	Returns a constant value of RequestType.CreatePolicy to indicate that this is a CreatePolicyRequest.

Create a new class named `CreatePolicyRequest.cs` in the project with the contents shown in Example 6-27.

Example 6-27 CreatePolicyRequest

```
using System;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class CreatePolicyRequest : ClaimsProcessingRequest
    {
        [DataMember]
        public override RequestType TypeOfRequest
        {
            get { return RequestType.CreatePolicy; }
        }

        private Customer _PrimaryCustomer;
        private Collection<Customer> _NamedParties;
        private string _PolicyName;
        private int _PolicyCover;
        private int _PolicyExcess;

        [DataMember]
        public Customer PrimaryCustomer
        {
            get { return _PrimaryCustomer; }
            set { _PrimaryCustomer = value; }
        }

        [DataMember]
        public Collection<Customer> NamedParties
        {
            get { return _NamedParties; }
        }

        [OnDeserializing]
        private void OnDeserializing(StreamingContext context)
        {
            _NamedParties = new Collection<Customer>();
        }

        [DataMember]
        public string PolicyName
        {
            get { return _PolicyName; }
            set { _PolicyName = value; }
        }

        [DataMember]
        public int PolicyCover
```

```

    {
        get { return _PolicyCover; }
        set { _PolicyCover = value; }
    }

    [DataMember]
    public int PolicyExcess
    {
        get { return _PolicyExcess; }
        set { _PolicyExcess = value; }
    }

    public CreatePolicyRequest()
    {
        _NamedParties = new Collection<Customer>();
    }
}

```

Tip: In the default constructor, the collection is initialized. This is to prevent `NullReferenceExceptions` when a new object is created programmatically. When an object is deserialized by WCF the constructor is not executed. Care therefore needs to be taken when relying on the existence of the `NamedParties` property.

The `CreatePolicyResponse` defines the properties shown in Table 6-7.

Table 6-7 *CreatePolicyResponse Properties*

Property	Purpose
Policy	The full policy details of the newly created policy.
TypeOfResponse	Returns a constant value of <code>ResponseType.CreatePolicy</code> to indicate that this is a <code>CreatePolicyResponse</code> .

Create a new class named `CreatePolicyResponse.cs` in the project with the contents in Example 6-28.

Example 6-28 *CreatePolicyResponse*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class CreatePolicyResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.CreatePolicy; }
        }
    }
}

```

```

//CreatePolicyResponse
private InsurancePolicy _Policy;

[DataMember]
public InsurancePolicy Policy
{
    get { return _Policy; }
    set { _Policy = value; }
}

public CreatePolicyResponse(InsurancePolicy policy)
{
    _Policy = policy;
}
}
}

```

Data contract For NewClaim

The NewClaimRequest datatype defines the properties shown in Table 6-8.

Table 6-8 NewClaimRequest Properties

Property	Purpose
Claim	The full claim details of the newly created claim.
TypeOfRequest	Returns a constant value of RequestType.NewClaim to indicate that this is a NewClaimRequest.

Create a new class named NewClaimRequest.cs with the content shown in Example 6-29.

Example 6-29 NewClaimRequest.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class NewClaimRequest : ClaimsProcessingRequest
    {
        [DataMember]
        public override RequestType TypeOfRequest
        {
            get { return RequestType.NewClaim; }
        }
        private Claim _NewClaim;

        [DataMember]
        public Claim NewClaim
        {
            get { return _NewClaim; }
        }
    }
}

```

```

        set { _NewClaim = value; }
    }

    public NewClaimRequest(Claim newRequest, string messageId,
        string applicationId)
    {
        this.NewClaim = newRequest;
        this.MessageId = messageId;
        this.ApplicationId = applicationId;
        this.RequestDate = DateTime.Now;
    }
}
}
}

```

The `NewClaimResponse` datatype defines the properties shown in Table 6-9.

Table 6-9 *NewClaimResponse.cs*

Property	Purpose
<code>TypeOfResponse</code>	Returns a constant value of <code>ResponseType.NewClaim</code> to indicate that this is a <code>NewClaimResponse</code> .

Create a new file named `NewClaimResponse.cs` in the project with the content in Example 6-30.

Example 6-30 *NewClaimResponse.cs*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class NewClaimResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.NewClaim; }
        }
    }
}
}

```

Tip: The `NewClaimResponse` datatype does not define any additional parameters over the base type of `ClaimsProcessingResponse`; however, it does override the `TypeOfResponse` property. This pattern is repeated in some of the other response datatypes in this section.

Data contract for RejectClaim

The RejectClaimRequest datatype defines the properties shown in Table 6-10.

Table 6-10 *RejectClaimRequest Properties*

Property	Purpose
ClaimReference	A unique ID representing the claim being rejected.
AgentReference	A unique ID representing the Agent responsible for rejecting the claim referred to in the ClaimReference field.
Reason	The reason for rejecting the claim.
TypeOfRequest	Returns a constant value of RequestType.RejectClaim to indicate that this is a RejectClaimRequest.

Create a new class named RejectClaimRequest.cs with the content shown in Example 6-31.

Example 6-31 *RejectClaimRequest.cs*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class RejectClaimRequest : ClaimsProcessingRequest
    {
        [DataMember]
        public override RequestType TypeOfRequest
        {
            get { return RequestType.RejectClaim; }
        }

        private string _ClaimReference;
        private string _AgentReference;
        private string _Reason;

        [DataMember]
        public string ClaimReference
        {
            get { return _ClaimReference; }
            set { _ClaimReference = value; }
        }

        [DataMember]
        public string AgentReference
        {
            get { return _AgentReference; }
            set { _AgentReference = value; }
        }

        [DataMember]
```

```

        public string Reason
        {
            get { return _Reason; }
            set { _Reason = value; }
        }
    }
}

```

The RejectClaimResponse datatype defines the properties shown in Table 6-11.

Table 6-11 *RejectClaimResponse Properties*

Property	Purpose
TypeOfResponse	Returns a constant value of ResponseType.RejectClaim to indicate that this is a RejectClaimResponse.

Create a new class in the project named RejectClaimResponse.cs with the contents shown in Example 6-32.

Example 6-32 *RejectClaimResponse.cs*

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class RejectClaimResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse{
            get { return ResponseType.RejectClaim; }
        }
    }
}

```

Data contract for ViewClaim

The ViewClaimRequest datatype defines the properties shown in Table 6-12.

Table 6-12 *ViewClaimRequest Properties*

Property	Purpose
ClaimReference	A unique ID referencing the claim to view.
TypeOfRequest	Returns a constant value of RequestType.ViewClaim to indicate that this is a ViewClaimRequest.

Create a new class named ViewClaimRequest.cs with the content shown in Example 6-33.

Example 6-33 ViewClaimRequest.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewClaimRequest : ClaimsProcessingRequest
    {
        [DataMember]
        public override RequestType TypeOfRequest
        {
            get { return RequestType.ViewClaim; }
        }

        private string _ClaimReference;

        [DataMember]
        public string ClaimReference
        {
            get { return _ClaimReference; }
            set { _ClaimReference = value; }
        }
    }
}
```

The ViewClaimResponse class defines the properties shown in Table 6-13.

Table 6-13 ViewClaimResponse Properties

Property	Purpose
Claim	The full claim details corresponding to the claim reference passed in the corresponding ViewClaimRequest.
TypeOfResponse	Returns a constant value of ResponseType.ViewClaim to indicate that this is a ViewClaimResponse.

Create a new class named ViewClaimResponse.cs with the content in Example 6-34.

Example 6-34 ViewClaimResponse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewClaimResponse : ClaimsProcessingResponse
```

```

    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.ViewClaim; }
        }

        private Claim _Claim;

        [DataMember]
        public Claim Claim
        {
            get { return _Claim; }
            set { _Claim = value; }
        }

        public ViewClaimResponse(Claim claim)
        {
            _Claim = claim;
        }
    }
}

```

Data contract for ViewOutstandingClaims

The ViewOutstandingClaimsRequest datatype defines the properties in Table 6-14.

Table 6-14 ViewOutstandingClaimsRequest properties

Property	Purpose
TypeOfRequest	Returns a constant value of RequestType.ViewOutstandingClaim to indicate that this is a ViewOutstandingClaimsRequest.

Create a new class named ViewOutstandingClaimsRequest.cs with the content in Example 6-35.

Example 6-35 ViewOutstandingClaimsRequest.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewOutstandingClaimsRequest : ClaimsProcessingRequest
    {
        [DataMember]
        public override RequestType TypeOfRequest
        {
            get { return RequestType.ViewOutstandingClaims; }
        }
    }
}

```

}

The ViewOutstandingClaimsResponse defines the properties in Table 6-15.

Table 6-15 ViewOutstandingClaimsResponse

Property	Purpose
OutstandingClaims	Returns a collection of claims that are in the Submitted state.
TypeOfResponse	Returns a constant value of ResponseType.ViewOutstandingClaim to indicate that this is a ViewOutstandingClaimsResponse.

Create a new class in the project named ViewOutstandingClaimsResponse.cs with the content in Example 6-36.

Example 6-36 ViewOutstandingClaimsResponse.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.Collections.ObjectModel;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewOutstandingClaimsResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.ViewOutstandingClaims; }
        }

        private Collection<Claim> _OutstandingClaims;

        [DataMember]
        public Collection<Claim> OutstandingClaims
        {
            get { return _OutstandingClaims; }
        }

        [OnDeserializing]
        private void OnDeserializing(StreamingContext context)
        {
            _OutstandingClaims = new Collection<Claim>();
        }

        public void AddOutstandingClaim(Claim claim)
        {
            _OutstandingClaims.Add(claim);
        }
    }
}
```

```

public ViewOutstandingClaimsResponse()
{
    _OutstandingClaims = new Collection<Claim>();
}

public ViewOutstandingClaimsResponse(Collection<Claim> outstandingClaims)
{
    _OutstandingClaims = new Collection<Claim>();
    foreach (Claim claim in outstandingClaims)
    {
        _OutstandingClaims.Add(claim);
    }
}
}
}

```

Tip: This class implements a method called `OnDeserializing`, which is decorated with the `[OnDeserializing]` attribute. This attribute specifies that when this class is being deserialized by the `DataContractSerializer` the decorated method is called. This method might have any name but it must take a `StreamingContext` as a parameter.

DataContract for ViewPolicy

The `ViewPolicyRequest` datatype defines the properties shown in Table 6-16.

Table 6-16 ViewPolicyRequest Properties

Property	Purpose
PolicyReference	A unique ID representing the policy to be viewed.
TypeOfRequest	Returns a constant value of <code>RequestType.ViewPolicy</code> to indicate that this is a <code>ViewPolicyRequest</code> .

Create a new class in the project named `ViewPolicyRequest.cs` with the contents in Example 6-37.

Example 6-37 ViewPolicyRequest

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewPolicyRequest : ClaimsProcessingRequest
    {
        private string _PolicyReference;

        [DataMember]
        public string PolicyReference
        {
            get { return _PolicyReference; }
            set { _PolicyReference = value; }
        }
    }
}

```

```

    }

    [DataMember]
    public override RequestType TypeOfRequest
    {
        get { return RequestType.ViewPolicy; }
    }
}
}

```

The ViewPolicyResponse class defines the properties shown in Table 6-17.

Table 6-17 ViewPolicyResponse.cs

Property	Purpose
Policy	The full policy details referenced by the PolicyReference passed in the corresponding ViewPolicyRequest.
TypeOfResponse	Returns a constant value of ResponseType.ViewPolicy to indicate that this is a ViewPolicyResponse.

Create a new class in the project named ViewPolicyResponse.cs with the contents in Example 6-38.

Example 6-38 ViewPolicyResponse.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewPolicyResponse : ClaimsProcessingResponse
    {
        [DataMember]
        public override ResponseType TypeOfResponse
        {
            get { return ResponseType.ViewPolicy; }
        }

        private InsurancePolicy _Policy;

        [DataMember]
        public InsurancePolicy Policy
        {
            get { return _Policy; }
            set { _Policy = value; }
        }

        public ViewPolicyResponse(InsurancePolicy policy)
        {
            _Policy = policy;
        }
    }
}

```

```
    }  
  }  
}
```

Defining the Fault types

The data contract also specifies what kind of error information is recognized by clients consuming the service, which we discuss in more detail in “Defining the Fault types” on page 208.

Create a class named `ClaimProcessingFault.cs` in the project with the content shown in Example 6-39.

Example 6-39 ClaimProcessingFault.cs

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Runtime.Serialization;  
  
namespace Ibm.Broker.Example.ClaimsProcessing  
{  
  
    [DataContract]  
    public class ClaimProcessingFault  
    {  
        private string _CorrelationId;  
        private DateTime _FaultTime;  
        private string _ErrorCode;  
        private string _Message;  
  
        public ClaimProcessingFault(string correlationId)  
        {  
            _CorrelationId = correlationId;  
            _FaultTime = DateTime.Now;  
        }  
  
        [DataMember]  
        public string CorrelationId  
        {  
            get { return _CorrelationId; }  
            set { _CorrelationId = value; }  
        }  
  
        [DataMember]  
        public DateTime FaultTime  
        {  
            get { return _FaultTime; }  
            set { _FaultTime = value; }  
        }  
  
        [DataMember]  
        public string ErrorCode  
        {  
            get { return _ErrorCode; }  
        }  
    }  
}
```

```

        set { _ErrorCode = value; }
    }

    [DataMember]
    public string Message
    {
        get { return _Message; }
        set { _Message = value; }
    }
}
}
}

```

The base class `ClaimsProcessingFault` is used for the majority of application errors in this service; however, there is an exceptional case that has its own type of fault.

When retrieving a list of outstanding claims, it is possible for an exception to be thrown when accessing a particular claim record. However callers can still work with the incomplete list. In this instance, a special fault type can be used to pass the incomplete list of claims back to the client along with a list of the specific faults that occurred.

Create a new class named `ViewOutstandingClaimsIncompleteResultsFault.cs` in the project with the content in Example 6-40.

Example 6-40 ViewOutstandingClaimsIncompleteResultsFault.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.Collections;
using System.Collections.ObjectModel;

namespace Ibm.Broker.Example.ClaimsProcessing
{
    [DataContract]
    public class ViewOutstandingClaimsIncompleteResultsFault :
        ClaimProcessingFault
    {
        private Collection<Claim> _OutstandingResults;
        private Collection<ClaimProcessingFault> _FaultList;

        public ViewOutstandingClaimsIncompleteResultsFault(string correlationId) :
            base(correlationId)
        {
            _OutstandingResults = new Collection<Claim>();
            _FaultList = new Collection<ClaimProcessingFault>();
        }

        [DataMember]
        public Collection<Claim> OutstandingResults
        {
            get { return _OutstandingResults; }
        }
    }
}

```

```

    [OnDeserializing]
    private void OnDeserializing(StreamingContext context)
    {
        _OutstandingResults = new Collection<Claim>();
    }

    public void AddOutstandingClaim(Claim claim)
    {
        _OutstandingResults.Add(claim);
    }

    [DataMember]
    public Collection<ClaimProcessingFault> FaultList
    {
        get { return _FaultList; }
    }

    public void AddFault(ClaimProcessingFault fault)
    {
        _FaultList.Add(fault);
    }
}
}
}

```

The `ViewOutstandingClaimsIncompleteResultsFault` is derived from `ClaimsProcessingFault` so that clients who do not want to work with incomplete claims lists can use their normal fault handling strategy to cope with the failure.

Implementing the `IClaimsProcessingWcfService` interface

In this section, you add methods to the `ClaimsProcessingWcfService` class that implement the service contract. The implementation is loosely coupled to the client using the WCF service contract; therefore, it is not necessary for a consumer of the service to understand the implementation, and indeed the implementation can be replaced without impacting an existing client, provided that the semantics of the service contract are retained.

In this section, you complete the service by implementing the operation contracts.

Adding required data members

Recall that the data used by the service is stored in files on the file system. The implementation of the service must be able to locate the correct location for these files. To do so, add the private data members in Example 6-41 on page 210 to `ClaimsProcessingWcfService`. The code in bold must be added to the class definition outside the scope of any of the class methods.

Example 6-41 Private data members controlling file locations

```

namespace IBM.Broker.Example.ClaimsProcessing
{
    public class ClaimsProcessingWcfService : IClaimsProcessingWcfService
    {
        private string _BaseDir = "C:\\temp\\testData";
        private const string CustomersSuffix = "customers";
    }
}

```

```
private const string ClaimsSuffix = "claims";
private const string PolicySuffix = "policies";
```

The `_BaseDir` variable sets the base location for the files created and accessed by the service. In this example, the variable is set to the directory `c:\temp\testData`. This directory must exist on the file system before starting the completed service; however, you can alter the location to refer to a directory on your local system.

The constants represent subdirectories for storing particular types of data, for example, serialized customer data is stored in `C:\temp\testData\customers` directory. The customers, claims, and policies subdirectories must exist inside your chosen base directory before starting the completed service.

Tip: The `\` character is an escape in C#, so to insert a `\` character into a string, it needs to be preceded by another slash. The `@` prefix can be added to string literals so that the following string is treated as a verbatim string, ignoring escape characters. The `_BaseDir` declaration can therefore also be written as:

```
private string _BaseDir = @"C:\temp\testData"
```

The only exception escape sequence that is recognized in verbatim string literals is the double quote escape sequence `""`.

Example 6-42 shows private data members representing known insurance policy products and partner companies underwriting policies. In a real system, the service might obtain this data from a database or from some other means; however, to keep the service simple, the data is made available directly to the `ClaimsProcessingWcfService` class.

Add the code directly below the code you added in Example 6-41. The code must be within the class definition but outside the scope of any method.

Example 6-42 Private data members storing example product information

```
private enum PolicyProducts { Bronze, BronzePlus, Silver, Gold, Premium,
PremiumPlus }
    private enum UnderWriters { ItsoInsurancePartner1, ItsoInsurancePartner2,
ItsoInsurancePartner3 }
    private static Dictionary<PolicyProducts, UnderWriters>
ProductToProviderMap = new Dictionary<PolicyProducts, UnderWriters>();
```

The `ProductToProviderMap` dictionary is used to look up which company underwrites a particular policy. This is required so that payments are billed to the correct partner company. This dictionary must be populated on service startup by adding the code in Example 6-43 on page 211 to the constructor for `ClaimsProcessingWcfService`. The variable is declared, so the count of the elements in the dictionary is tested before population to prevent duplicate content.

Add the code shown in bold in Example 6-43 to the constructor for the `ClaimsProcessingWcfService` class.

Example 6-43 Initializing the ProductToProvider map

```
public ClaimsProcessingWcfService()
{
    CustomerSerializer = new XmlSerializer(typeof(Customer));
    PolicySerializer = new XmlSerializer(typeof(InsurancePolicy));
```

```

        ClaimSerializer = new XmlSerializer(typeof(Claim));

//populate the provider map if not already done
if (ProductToProviderMap.Count == 0)
{
    //predefine the provider to product map
    ProductToProviderMap.Add(PolicyProducts.Bronze,
                            UnderWriters.ItsoInsurancePartner1);
    ProductToProviderMap.Add(PolicyProducts.BronzePlus,
                            UnderWriters.ItsoInsurancePartner1);
    ProductToProviderMap.Add(PolicyProducts.Silver,
                            UnderWriters.ItsoInsurancePartner2);
    ProductToProviderMap.Add(PolicyProducts.Gold,
                            UnderWriters.ItsoInsurancePartner2);
    ProductToProviderMap.Add(PolicyProducts.Premium,
                            UnderWriters.ItsoInsurancePartner3);
    ProductToProviderMap.Add(PolicyProducts.PremiumPlus,
                            UnderWriters.ItsoInsurancePartner3);
}

```

This code creates the mapping of policy product names to financial partner names shown in Table 6-18.

Table 6-18 Policy Product Name to Underwriter mapping

Policy Product Name	Financial Partner underwriting this policy product.
Bronze	ItsInsurancePartner1
BronzePlus	ItsInsurancePartner1
Silver	ItsInsurancePartner2
Gold	ItsInsurancePartner2
Premium	ItsInsurancePartner3
PremiumPlus	ItsInsurancePartner3

Generating test data

To get started with this scenario, you need some test data to work with. Add the method shown in Example 6-44 to the ClaimsProcessingWcfService to generate three sample customers, a sample policy, and a sample claim.

Example 6-44 Generating test data

```

private void CreateTestData()
{
    Customer cust = new Customer();
    cust.CustomerId = "CUST00001";
    cust.Address = "1 Generic Street, Any Town, USA";
    cust.FirstName = "John";
    cust.Surname = "Doe";
    cust.OtherNames.Add("James");
    cust.OtherNames.Add("Jeffrey");
    cust.Email = "johndoe@itso.example.ibm.com";
}

```

```

WriteCustomerToFile(cust, _BaseDir + "\\\" + CustomersSuffix);

Customer cust2 = new Customer();
cust2.CustomerId = "CUST00002";
cust2.Address = "1 Generic Street, Any Town, USA";
cust2.FirstName = "Sam";
cust2.Surname = "Doe";
cust2.OtherNames.Add("Stephen");
cust2.OtherNames.Add("John");
cust2.Email = "samdoe@itso.example.ibm.com";

WriteCustomerToFile(cust2, _BaseDir + "\\\" + CustomersSuffix);

Customer cust3 = new Customer();
cust3.CustomerId = "CUST00003";
cust3.Address = "1 Generic Street, Any Town, USA";
cust3.FirstName = "Jane";
cust3.Surname = "Doe";
cust3.OtherNames.Add("Jill");
cust3.OtherNames.Add("Jenny");
cust3.Email = "janedoe@itso.example.ibm.com";

WriteCustomerToFile(cust3, _BaseDir + "\\\" + CustomersSuffix);

InsurancePolicy p = new InsurancePolicy();
p.NamedParties.Add(cust2);
p.NamedParties.Add(cust3);
p.PolicyCover = 1000;
p.PolicyExcess = 150;
//add one year
p.PolicyExpiryDate = DateTime.Now + new TimeSpan(365, 0, 0, 0, 0);
p.Policyholder = cust;
p.PolicyNumber = "POLICY0001";
p.PolicyStartDate = DateTime.Now;
p.Underwriter = "Partner Insurance Company A";

WritePolicyToFile(p, _BaseDir + "\\\" + PolicySuffix);

Claim c = new Claim();
c.Customer = cust3;
c.ClaimDate = DateTime.Now - new TimeSpan(20, 0, 0, 0, 0);
c.ClaimReceivedDate = DateTime.Now;
c.ClaimReference = "CLAIM0001";
c.ClaimValue = 400;
c.PolicyNumber = "POLICY0001";
c.Description = "Accidental damage to car, no other parties involved";
c.Status = ClaimState.Submitted;

WriteClaimToFile(c, _BaseDir + "\\\" + ClaimsSuffix);

}

```

This code must be called only when the test data does not already exist. This condition is tested for by checking if there are any files present in the customers subdirectory before

creating the test data. Add the code shown in Example 6-45 to the constructor for the ClaimsProcessingWcfService class.

Example 6-45 Creating test data if it does not already exist

```
//if there is no data present then create some test data
if (Directory.GetFiles(_BaseDir + "\\\" + CustomersSuffix).Length == 0)
{
    CreateTestData();
}
```

The completed constructor for ClaimsProcessingWcfService now looks like Example 6-46.

Example 6-46 The completed constructor for ClaimsProcessingWcfService

```
public ClaimsProcessingWcfService()
{
    CustomerSerializer = new XmlSerializer(typeof(Customer));
    PolicySerializer = new XmlSerializer(typeof(InsurancePolicy));
    ClaimSerializer = new XmlSerializer(typeof(Claim));

    //if there is no data present then create some test data
    if (Directory.GetFiles(_BaseDir + "\\\" + CustomersSuffix).Length == 0)
    {
        CreateTestData();
    }

    //populate the provider map if not already done
    if (ProductToProviderMap.Count == 0)
    {
        //predefine the provider to product map
        ProductToProviderMap.Add(PolicyProducts.Bronze,
                                UnderWriters.ItsoInsurancePartner1);
        ProductToProviderMap.Add(PolicyProducts.BronzePlus,
                                UnderWriters.ItsoInsurancePartner1);
        ProductToProviderMap.Add(PolicyProducts.Silver,
                                UnderWriters.ItsoInsurancePartner2);
        ProductToProviderMap.Add(PolicyProducts.Gold,
                                UnderWriters.ItsoInsurancePartner2);
        ProductToProviderMap.Add(PolicyProducts.Premium,
                                UnderWriters.ItsoInsurancePartner3);
        ProductToProviderMap.Add(PolicyProducts.PremiumPlus,
                                UnderWriters.ItsoInsurancePartner3);
    }
}
```

Implementing the ApproveClaim method

The code for the ApproveClaim method is shown in Example 6-48.

1. Add the new using statement shown at the top of the service class to get access to the Culture.InvariantCulture class, which is required to tell the system that strings in a fault thrown by this class will not be translated.

Example 6-47 Using statement

```
using System.Collections.ObjectModel;
```

2. Add this code to the ClaimsProcessingWcfService.cs file.

Example 6-48 The ApproveClaim method

```
public ApproveClaimResponse ApproveClaim(ApproveClaimRequest request)
{
    ApproveClaimResponse Response;
    try
    {
        //First find out if the claim exists
        if (File.Exists(_BaseDir + "\\\" + ClaimsSuffix + "\\\" + request.ClaimReference))
        {
            Claim ClaimToApprove = GetClaimFromFile(_BaseDir + "\\\" + ClaimsSuffix + "\\\" +
request.ClaimReference);
            if (request.ValueApproved > ClaimToApprove.ClaimValue)
            {
                //Return the error to the caller
                ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
                Fault.ErrorCode = "ERR010";
                Fault.Message = "Could not approve claim. Reason: Approved value " + request.ValueApproved + " is
higher than total value of claim.";
                throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
            }
            else
            {
                //get the policy associated with this claim
                if (File.Exists(_BaseDir + "\\\" + PolicySuffix + "\\\" + ClaimToApprove.PolicyNumber))
                {
                    InsurancePolicy PolicyToCheck = GetPolicyFromFile(_BaseDir + "\\\" + PolicySuffix + "\\\" +
ClaimToApprove.PolicyNumber);
                    if (PolicyToCheck.PolicyCover > ClaimToApprove.ClaimValue)
                    {
                        //Policy can be approved and a payment generated
                        int PaymentValue = ClaimToApprove.ClaimValue - PolicyToCheck.PolicyExcess;
                        if (PaymentValue < 0) PaymentValue = 0;
                        Payment GeneratedPayment = new Payment();
                        GeneratedPayment.Payee = ClaimToApprove.Customer;
                        GeneratedPayment.Amount = PaymentValue;
                        GeneratedPayment.Underwriter = PolicyToCheck.Underwriter;
                        Response = new ApproveClaimResponse(GeneratedPayment);
                        ClaimToApprove.Status = ClaimState.Approved;
                        ClaimToApprove.Notes += System.Environment.NewLine + "[" + DateTime.Now + "] AgentRef: " +
request.AgentReference +
                        System.Environment.NewLine + "APPROVED " + request.ValueApproved;
                        WriteClaimToFile(ClaimToApprove, _BaseDir + "\\\" + ClaimsSuffix);
                    }
                    else
                    {
                        //Return the error to the caller
                        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
                        Fault.ErrorCode = "ERR012";
                        Fault.Message = "Could not approve claim. Reason: Policy cover of " +
PolicyToCheck.PolicyCover +
                        " is less than value of claim " + ClaimToApprove.ClaimValue;
                        throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
                    }
                }
            }
            else
            {

```

```

        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR011";
        Fault.Message = "Could not approve claim. Reason: Claim refers to non-existent policy " +
ClaimToApprove.PolicyNumber;
        throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
}
else
{
    //Return the error to the caller
    ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
    Fault.ErrorCode = "ERR010";
    Fault.Message = "Could not approve claim. Reason: Claim " + request.ClaimReference + " does not
exist.";
    throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
}
}
catch (Exception e)
{
    //Are we an exception type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR009";
        Fault.Message = "Could not approve claim. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}
Response.CorrelationId = request.MessageId;
return Response;
}

```

The ApproveClaim method performs a number of checks before changing the referenced claims to the ClaimState.Approved status. The claim and its associated policy must exist in the system, and the value of the claim must not exceed the total policy cover. If these conditions are met, a new payment is generated for the value of the claim minus the value of any excess associated with the policy, and the claim is moved to the Approved state. A note is added to the claim to indicate the date that the claim was approved and which agent approved the claim.

If the result of any of the checks is false, a fault is thrown that contains a unique error code relating to the error state and an explanation of the root cause of the fault. For example, if the claim does not exist on the file system, the fault in Example 6-49 is thrown.

Example 6-49 A fault indicating that the referenced claim did not exist

```

//Return the error to the caller
ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
Fault.ErrorCode = "ERR010";

```

```

Fault.Message = "Could not approve claim. Reason: Claim " +
                request.ClaimReference + " does not exist.";
throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new
FaultCode(Fault.ErrorCode));

```

As shown in Example 6-50, the whole method is wrapped in a try block so that any exception types that are known to be generated by the code in the method can be converted into a fault that can be consumed by the client. In this case, the code can cause an `IOException`, if there are problems with the underlying file system operations or `InvalidOperationException`, if an attempt is made to write to a `FileStream` that was opened for reading. These exception types are therefore caught and handled by creating a new `ClaimsProcessingFault` to return to the client. Any unexpected Exceptions are rethrown. More detail about exception handling in WCF is in Chapter 7, “Scenario: Integrating Windows Communication Foundation in message flows - Part 2” on page 273.

Example 6-50 Converting exceptions into Faults

```

catch (Exception e)
{
    //Are we an exception type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR009";
        Fault.Message = "Could not approve claim. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}

```

Implementing the CreateClaim method

The `CreateClaim` method is responsible for taking the details sent in the client request and converting these to a `Claim` object that is stored on the file system on the server that hosts the `ClaimsProcessingWcfService`.

Add the method shown in Example 6-51 on page 217 to `ClaimsProcessingWcfService.cs`.

Example 6-51 The CreateClaim method

```

public NewClaimResponse CreateClaim(NewClaimRequest request)
{
    NewClaimResponse Response;
    try
    {
        //check that the claim is in a submitted state
        if (!request.NewClaim.Status.Equals(ClaimState.Submitted))
        {
            ClaimProcessingFault Fault = new
                ClaimProcessingFault(request.MessageId);

```

```

        Fault.ErrorCode = "ERR003";
        Fault.Message = "Could not generate new claim. Reason: " +
            "Invalid Claim Status " + request.NewClaim.Status;
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    //Check if this claim has a claimId
    else if (String.IsNullOrEmpty(request.NewClaim.ClaimReference))
    {
        //generate a new claim reference by listing the file
        String NewClaimRef = "CLAIM" + Guid.NewGuid().ToString("N");
        request.NewClaim.ClaimReference = NewClaimRef;
        WriteClaimToFile(request.NewClaim, _BaseDir + "\\\" + ClaimsSuffix);
        Response = new NewClaimResponse();
    }
    else
    {
        //If the claim already exists we must return an error
        if (File.Exists(_BaseDir + "\\\" + ClaimsSuffix + "\\\" +
            request.NewClaim.ClaimReference))
        {
            ClaimProcessingFault Fault = new
                ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR002";
            Fault.Message = "Could not generate new claim. Reason: " +
                "Duplicate claim reference " + request.NewClaim.ClaimReference;
            throw new FaultException<ClaimProcessingFault>(Fault, new
                FaultReason(new FaultReasonText(Fault.Message,
                    CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
        }
        else
        {
            WriteClaimToFile(request.NewClaim, _BaseDir + "\\\" + ClaimsSuffix);
            Response = new NewClaimResponse();
        }
    }
}
catch(Exception e)
{
    //Are we an exceptin type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR001";
        Fault.Message = "Could not generate new claim. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}

```

```

    }

    //set the correlId
    Response.CorrelationId = request.MessageId;
    return Response;
}

```

This method checks that the claim to be created is in the Submitted state and does not already exist before creating a corresponding object on the file system of the server running the WCF service. If a claim reference is not already provided, one is generated using the Guid class, which generates a 128-bit integer that has a low probability of being duplicated.

Tip: In this example, the Guid created by the Guid.NewGuid() is converted to a string using the toString("N") method invocation. This invocation passes "N" as a format specifier, which produces a 32-digit numeric string representation of the GUID. The following other format specifiers are supported:

- ▶ "D", 32 digits with hyphens separating groups
- ▶ "B", As "D" above but enclosed in braces
- ▶ "P", As "D" above but enclosed in parenthesis
- ▶ "X", Grouped hexadecimal representation

Implementing the CreatePolicy method

The CreatePolicy method also creates customers if the policy contains information about customers that are not already known to the system. To do this it uses a helper method.

Start by adding the method CreateCustomerIfNotExists in Example 6-52 to the class definition for ClaimProcessingWcfService.

Example 6-52 The CreateCustomerIfNotExists method

```

private void CreateCustomerIfNotExists(Customer customerToCreate)
{
    if (customerToCreate.CustomerId == null
        || customerToCreate.CustomerId.Length == 0)
    {
        //Generate a new customer reference
        String NewCustomerRef = "CUST" + Guid.NewGuid().ToString("N");
        customerToCreate.CustomerId = NewCustomerRef;
    }
    if (!File.Exists(_BaseDir + "\\\" + CustomersSuffix + "\\\" +
                    customerToCreate.CustomerId))
    {
        WriteCustomerToFile(customerToCreate, _BaseDir + "\\\" + CustomersSuffix);
    }
    else
    {
        //check if the names match
        Customer existingCustomer = GetCustomerFromFile(_BaseDir + "\\\" +
                    CustomersSuffix + "\\\" + customerToCreate.CustomerId);
        if (!existingCustomer.Equals(customerToCreate))
        {
            //if so resave to update any address info
            WriteCustomerToFile(customerToCreate, _BaseDir + "\\\" + CustomersSuffix);
        }
    }
}

```

```

        else
        {
            throw new IOException("Duplicate customer ID " +
                                  customerToCreate.CustomerId);
        }
    }
}

```

The `CreateCustomerIfNotExists` method checks if a customer file with a matching ID exists on the file system, and if so, uses the `Customer.Equals()` method discussed on page 175 to determine if the customer passed in is the same as a customer found on the file system. An `IOException` is thrown if the customer is passed in with the same ID as an existing customer, but the details do not match the existing customer held on file. This `IOException` is caught by the `CreatePolicy` method and converted into a `ClaimsProcessingFault` to pass the error cause back to the client consuming the service.

Next, add the `CreatePolicyResponse` method, shown in Example 6-53.

Example 6-53 CreatePolicyResponse

```

public CreatePolicyResponse CreatePolicy(CreatePolicyRequest request)
{
    CreatePolicyResponse Response;
    try
    {
        //check if the policy name matches a known product
        if (Enum.IsDefined(typeof(PolicyProducts), request.PolicyName))
        {
            PolicyProducts Product = (PolicyProducts)Enum.Parse(typeof(PolicyProducts), request.PolicyName);
            //check if customers exist and create them if not
            try
            {
                CreateCustomerIfNotExists(request.PrimaryCustomer);
                foreach(Customer NamedParty in request.NamedParties)
                {
                    CreateCustomerIfNotExists(NamedParty);
                }
            }
            //generate a new policy reference
            String NewPolicyRef = "POLICY" + Guid.NewGuid().ToString("N");
            //Create the policy
            InsurancePolicy NewPolicy = new InsurancePolicy();
            foreach (Customer NamedParty in request.NamedParties)
            {
                NewPolicy.AddNamedParty(NamedParty);
            }
            NewPolicy.PolicyCover = request.PolicyCover;
            NewPolicy.PolicyExcess = request.PolicyExcess;
            NewPolicy.PolicyExpiryDate = request.RequestDate + new TimeSpan(365, 0,0,0);
            NewPolicy.Policyholder = request.PrimaryCustomer;
            NewPolicy.PolicyName = request.PolicyName;
            NewPolicy.PolicyNumber = NewPolicyRef;
            NewPolicy.PolicyStartDate = request.RequestDate;
            NewPolicy.Underwriter = ProductToProviderMap[Product].ToString();
            WritePolicyToFile(NewPolicy, _BaseDir + "\\\" + PolicySuffix);
            Response = new CreatePolicyResponse(NewPolicy);
        }
    }
}

```

```

    }
    catch (Exception e)
    {
        //Are we an exceptin type we can't handle
        if (e is InvalidOperationException || e is IOException)
        {
            //Return the error to the caller
            ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR020";
            Fault.Message = "Could not generate new policy. Reason: Problem creating
customers: " + e.Message;
            throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
        }
        else
        {
            throw;
        }
    }
}
else
{
    //Return the error to the caller
    ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
    Fault.ErrorCode = "ERR019";
    Fault.Message = "Could not generate new policy. Reason: Unknown Policy name " +
request.PolicyName;
    throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
}
}
catch (Exception e)
{
    //Are we an exceptin type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR018";
        Fault.Message = "Could not generate new policy. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault, new FaultReason(new
FaultReasonText(Fault.Message, CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}
//set the correId
Response.CorrelationId = request.MessageId;
return Response;
}

```

Implementing the RejectClaim method

The RejectClaim method is used by an agent to move a claim to the Rejected state. Unlike the ApproveClaim method, no payment information is generated for rejected claims although the agent must provide a reason for rejecting the claim, which is logged in to the claims Notes property.

Add the code shown in Example 6-54 to the ClaimsProcessingWcfService class definition.

Example 6-54 The RejectClaim method

```
public RejectClaimResponse RejectClaim(RejectClaimRequest request)
{
    RejectClaimResponse Response;
    try
    {
        //First find out if the claim exists
        if (File.Exists(_BaseDir + "\\\" + ClaimsSuffix + "\\\" +
            request.ClaimReference))
        {
            Claim ClaimToReject = GetClaimFromFile(_BaseDir + "\\\" + ClaimsSuffix + "\\\"
                + request.ClaimReference);
            if (ClaimToReject.Status == ClaimState.Submitted)
            {
                ClaimToReject.Status = ClaimState.Rejected;
                ClaimToReject.Notes += System.Environment.NewLine + "[" + DateTime.Now +
                    "]" AgentRef: " + request.AgentReference +
                    System.Environment.NewLine + "REJECTED: " + request.Reason;
                WriteClaimToFile(ClaimToReject, _BaseDir + "\\\" + ClaimsSuffix);
                Response = new RejectClaimResponse();
            }
            else
            {
                //Return the error to the caller
                ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
                Fault.ErrorCode = "ERR015";
                Fault.Message = "Could not reject claim. Reason: " +
                    "Claim already had status of " + ClaimToReject.Status;
                throw new FaultException<ClaimProcessingFault>(Fault,
                    new FaultReason(new FaultReasonText(Fault.Message,
                        CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
            }
        }
        else
        {
            //Return the error to the caller
            ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR014";
            Fault.Message = "Could not reject claim. Reason: Claim " +
                request.ClaimReference + " does not exist.";
            throw new FaultException<ClaimProcessingFault>(Fault,
                new FaultReason(new FaultReasonText(Fault.Message,
                    CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
        }
    }
    catch (Exception e)
    {

```

```

//Are we an exceptin type we can't handle
if (e is InvalidOperationException || e is IOException)
{
    //Return the error to the caller
    ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
    Fault.ErrorCode = "ERR013";
    Fault.Message = "Could not reject claim. Reason: " + e.Message;
    throw new FaultException<ClaimProcessingFault>(Fault,
        new FaultReason(new FaultReasonText(Fault.Message,
            CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
}
else
{
    throw;
}
}
Response.CorrelationId = request.MessageId;
return Response;
}

```

Like the ApproveClaim method, the ClaimReference in the incoming request is used to check if a claim already exists in the system with this ID. If a claim is found, the state is checked to ensure that the claim is not already Approved or Rejected. If all of these conditions are met, the Claim is moved to the Rejected state and a note is added to the Claim to record the time the claim was rejected, the Agent Reference rejecting the claim, and the reason for rejecting the claim. Finally the altered claim object is serialized to the file system on the server hosting the ClaimsProcessingWcfService.

Implementing the ViewClaim method

The ViewClaim method is responsible for returning the details of a claim when passed a reference ID in the request. This can be used by agents examining a claim or by a service exposed to customers where they might, for example, be able to use a web service to check on the status of their own claim.

Add the ViewClaim method shown in Example 6-55 to the class definition for the ClaimsProcessingWcfService.

Example 6-55 The ViewClaim method

```

public ViewClaimResponse ViewClaim(ViewClaimRequest request)
{
    ViewClaimResponse Response;
    try
    {
        //check if the file exists
        if (File.Exists(_BaseDir + "\\\" + ClaimsSuffix + "\\\" +
            request.ClaimReference))
        {
            Response = new ViewClaimResponse(GetClaimFromFile(_BaseDir + "\\\" +
                ClaimsSuffix + "\\\" + request.ClaimReference));
        }
        else
        {
            //Return the error to the caller
            ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR004";
        }
    }
}

```

```

        Fault.Message = "Could not view claim. Reason: Claim " +
            request.ClaimReference + " does not exist.";
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
}
catch (Exception e)
{
    //Are we an exceptin type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR005";
        Fault.Message = "Could not view claim. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}
Response.CorrelationId = request.MessageId;
return Response;
}

```

This method is simple with most of the work being performed by the four lines shown in Example 6-56. The method checks that the Claim corresponding to the reference on the request object exists, and if so, returns the claim using the ViewClaimResponse(Claim) constructor. The majority of the code in this method is used to provide meaningful error messages to clients consuming the service if an error condition exists.

Example 6-56 Checking the Claim exists and returning it in a ViewClaimResponse object

```

if (File.Exists(_BaseDir + "\\\" + ClaimsSuffix + "\\\" + request.ClaimReference))
{
    Response = new ViewClaimResponse(GetClaimFromFile(_BaseDir + "\\\" +
        ClaimsSuffix + "\\\" + request.ClaimReference));
}

```

Implementing the ViewOutstandingClaims method

The ViewOutstandingClaims method is used by agents to retrieve details for all claims that are in the Submitted state. The agent can then work on a claim and use the appropriate methods to accept or reject the claim. This method might be used by a remote application to construct a queue of claims awaiting an agent's attention.

Create a new method in the ClaimsProcessingWcfService class with the code shown in Example 6-57.

Example 6-57 The ViewOutstandingClaims method

```

public ViewOutstandingClaimsResponse
    ViewOutstandingClaims(ViewOutstandingClaimsRequest request)

```

```

{
    ViewOutstandingClaimsResponse Response;
    try
    {
        Collection<Claim> OutstandingClaims = new Collection<Claim>();
        ViewOutstandingClaimsIncompleteResultsFault Fault = null;

        //Iterate over files in the directory
        string[] FilePaths = Directory.GetFiles(_BaseDir + "\\\" + ClaimsSuffix,
                                                "CLAIM*", SearchOption.TopDirectoryOnly);
        foreach(string File in FilePaths)
        {
            try
            {
                Claim Claim = GetClaimFromFile(File);
                if(Claim.Status == ClaimState.Submitted)
                {
                    if (Fault == null)
                    {
                        OutstandingClaims.Add(Claim);
                    }
                    else
                    {
                        //if we are faulting add to the fault instead
                        Fault.AddOutstandingClaim(Claim);
                    }
                }
            }
            catch (Exception ei)
            {
                //Are we an exceptin type we can't handle
                if (ei is InvalidOperationException || ei is IOException)
                {
                    if (Fault == null)
                    {
                        //create a new fault
                        Fault = new
                            ViewOutstandingClaimsIncompleteResultsFault(request.MessageId);

                        //copy the results into the fault
                        foreach(Claim Claim in OutstandingClaims)
                        {
                            Fault.AddOutstandingClaim(Claim);
                        }
                    }
                    ClaimProcessingFault InnerFault = new
                        ClaimProcessingFault(request.MessageId);
                    InnerFault.CorrelationId = request.MessageId;
                    InnerFault.ErrorCode = "ERR006";
                    InnerFault.Message = "Error reading claim file " + File +
                        " Reason: " + ei.Message;
                    Fault.AddFault(InnerFault);
                }
            }
            else
            {

```

```

        throw;
    }
}
if (Fault != null)
{
    Fault.ErrorCode = "ERR007";
    Fault.Message = "There were errors reading claim files, results may" +
        "be incomplete. See remaining FaultList for details.";
    throw new FaultException<ViewOutstandingClaimsIncompleteResultsFault>(
        Fault, new FaultReason(new FaultReasonText(Fault.Message,
            CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
}
else
{
    Response = new ViewOutstandingClaimsResponse(OutstandingClaims);
}
}
catch (Exception e)
{
    //Are we an exceptin type we can't handle
    if (e is InvalidOperationException || e is IOException)
    {
        //Return the error to the caller
        ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
        Fault.ErrorCode = "ERR008";
        Fault.Message = "Could not generate new claim. Reason: " + e.Message;
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}
Response.CorrelationId = request.MessageId;
return Response;
}

```

This method has reasonably complex business logic. The purpose of the method is to return all Claim objects that exist on the WCF service's file system that are in the Submitted state. This is performed by using the `Directory.GetFiles()` method to obtain a list of all Claim objects and then iterating over these to build up a list of those that have the correct `Claim.Status`.

The complication is that reading any of the serialized Claim objects from the file system can potentially throw an exception. In the `ViewClaim` method, if an exception is encountered while reading the specific claim that is being requested, there is no way for the method to continue and generate a meaningful result and so a fault is returned to the client. In the case of the `ViewOutstandingClaims` method, however, an exception reading one file cannot prevent other serialized Claim objects from being successfully read and processed.

The service itself has no way of knowing if the client can tolerate an incomplete list of outstanding claims. An application using this interface on an agent's computer might tolerate the failure because the agent can work on claims other than the one that encountered a

problem. On the other hand, a central administration application using the same interface might need to know when there is a problem with the serialized data stored on the problem.

For this reason, the decision to tolerate an incomplete list of data is delegated to the client and a new fault type `ViewOutstandingClaimsIncompleteResultsFault` is introduced to return the list of `Claim` objects that were successfully read from the file system along with a list of the errors experienced during the operation. This fault type was defined in Example 6-40 on page 209.

If no exceptions are encountered while reading the claims on the file system, a `ViewOutstandingClaimsResponse` object is returned containing the complete list of outstanding claims.

Implementing the ViewPolicy method

The `ViewPolicy` method is almost identical to the `ViewClaim` method except that instead of operating on `Claim` objects this method operates on `Policy` objects. As such, this method is responsible for returning the full details of the policy referenced in the request.

Add the method definition shown in Example 6-58 to the class definition for `ClaimsProcessingWcfService`.

Example 6-58 The ViewPolicy method

```
public ViewPolicyResponse ViewPolicy(ViewPolicyRequest request)
{
    ViewPolicyResponse Response;
    try
    {
        //check if the file exists
        if (File.Exists(_BaseDir + "\\\" + PolicySuffix + "\\\" +
                                                                request.PolicyReference))
        {
            Response = new ViewPolicyResponse(GetPolicyFromFile(_BaseDir + "\\\" +
                                                                PolicySuffix + "\\\" + request.PolicyReference));
        }
        else
        {
            //Return the error to the caller
            ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR017";
            Fault.Message = "Could not view policy. Reason: Policy " +
                            request.PolicyReference + " does not exist.";
            throw new FaultException<ClaimProcessingFault>(Fault,
                new FaultReason(new FaultReasonText(Fault.Message,
                    CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
        }
    }
    catch (Exception e)
    {
        //Are we an exceptin type we can't handle
        if (e is InvalidOperationException || e is IOException)
        {
            //Return the error to the caller
            ClaimProcessingFault Fault = new ClaimProcessingFault(request.MessageId);
            Fault.ErrorCode = "ERR016";
            Fault.Message = "Could not view Policy. Reason: " + e.Message;
        }
    }
}
```

```
        throw new FaultException<ClaimProcessingFault>(Fault,
            new FaultReason(new FaultReasonText(Fault.Message,
                CultureInfo.InvariantCulture)), new FaultCode(Fault.ErrorCode));
    }
    else
    {
        throw;
    }
}
Response.CorrelationId = request.MessageId;
return Response;
}
```

Verifying the complete ClaimsProcessingWcfService class

The WCF Service is now complete. You can verify that all the required data members and methods are implemented using the Class View in Visual Studio:

1. Select **Class View** from the tab panel at the bottom of the Solution Explorer. If you do not see this tab, select **View** → **Class View**.
2. From the tree view, expand the Ibm.Broker.Example.ClaimsProcessing namespace node.
3. Select the **ClaimsProcessingWcfService** node.
4. Verify that all of the elements in the members list shown in Figure 6-8 on page 229 are present in your completed file. The order of these elements might be different in your solution; however, all of the methods and data members must be present.

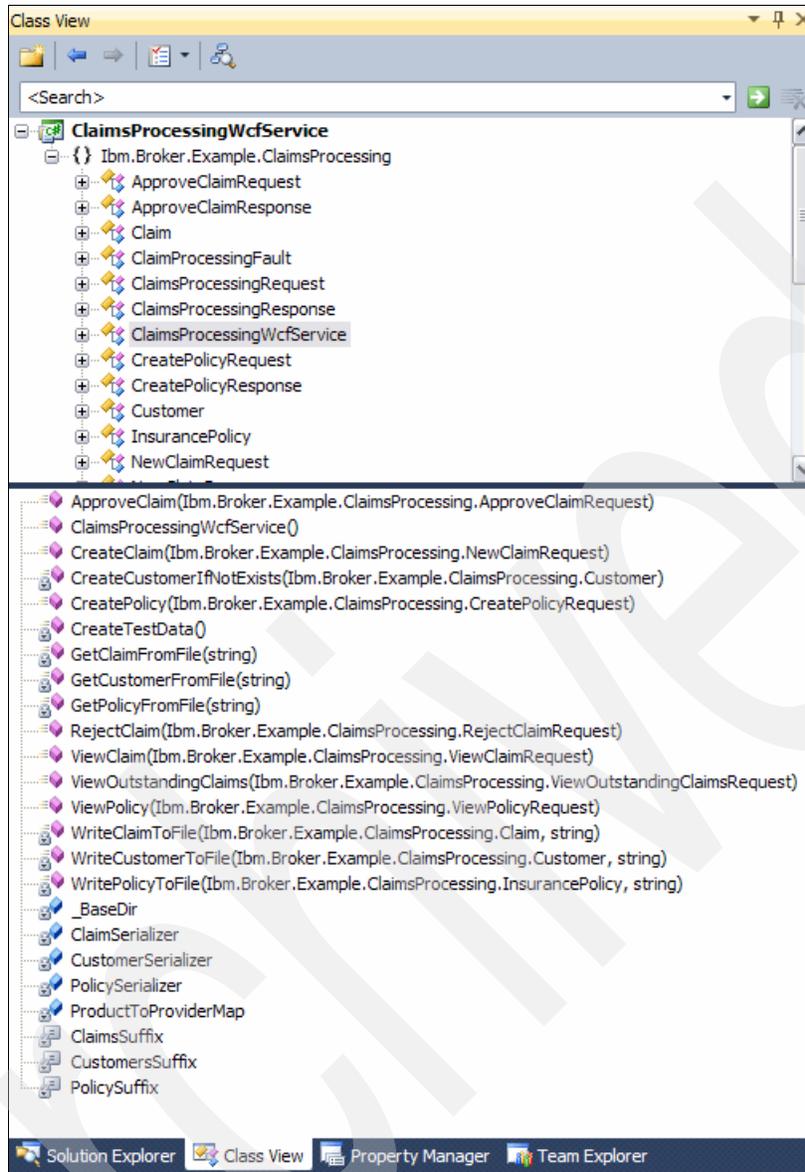


Figure 6-8 ClaimsProcessingWcfService class view

6.3.7 Building the WCF Service

To build the completed WCF service:

1. Save all the files in the solution by selecting the **File** → **SaveAll** menu command.
2. Select the **Build** → **Build Solution** menu item, or click the F7 key.
3. Check that the build succeeded by examining the build output in the Output panel. A successful build looks similar to the output shown in Figure 6-9 on page 230.

```

Output
Show output from: Build
----- Build started: Project: ClaimsProcessingWcfService, Configuration: Debug Any CPU -----
Build started 23/03/2012 15:24:19.
ResolveAssemblyReferences:
  A TargetFramework profile exclusion list will be generated.
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect to the input files.
_CopyAppConfigFile:
Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input files.
CopyFilesToOutputDirectory:
  ClaimsProcessingWcfService -> C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWcfService\ClaimsProcessingWcfService\bin\Debug\ClaimsProcessingWcfService.dll

Build succeeded.

Time Elapsed 00:00:00.01
***** Build: 1 succeeded or up-to-date, 0 failed, 0 skipped *****

```

Figure 6-9 A successful build

4. Check the output folder listed in the build output. The following files are generated:
 - ClaimsProcessingWcfService.dll
 - ClaimsProcessingWcfService.dll.config
 - ClaimsProcessingWcfService.pdb

6.3.8 Configuring the WCF service

In the previous section, you created a WCF service that provides a service contract that encapsulates a number of actions that can be invoked on the service by remote clients. You provided a file system-based implementation of this service contract in the `ClaimsProcessingWcfService` class.

When you implemented the service, you did not need to specify any of the specifics about where the service is located or what transport is being used. The WCF exposes these choices as configuration options that are only resolved at runtime. This effectively decouples the transport layer from the application layer, meaning that the application developer implementing a WCF service does not need to know how the service will be deployed in production. Similarly systems administrators can configure at runtime which endpoints and transports are used by the service without needing to rebuild the application.

The configuration information is held in the `App.config` file in the Solution. At build time this file is copied to a file named after its associated assembly with the `.config` extension. For example, when you built the service in section 6.3.8, “Configuring the WCF service” on page 230, the generated assembly produced was called `ClaimsProcessingWcfService.dll`, and the `App.config` file was copied to `ClaimsProcessingWcfService.dll.config`.

In this section, you modify the `App.Config` file directly because when the service is launched from within Visual Studio the file is automatically copied from `App.Config` to `ClaimsProcessingWcfService.dll.config`. If the WCF Service is hosted externally, modify the `ClaimsProcessingWcfService.dll.config` file directly.

The contents of the default App.config file

The New Project Wizard for a WCF Service library creates an `App.Config` file that already has some configuration completed. The `App.Config` created is displayed in Example 6-59.

Example 6-59 The automatically generated `App.Config` file

```

<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <system.web>
    <compilation debug="true" />

```

```

</system.web>
<!-- When deploying the service library project, the content of the config file
must be added to the host's
app.config file. System.Configuration does not support config files for
libraries. -->
<system.serviceModel>
  <services>
    <service
name="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService.ClaimsProces
singWcfService">
      <endpoint address="" binding="wsHttpBinding"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService.IClaimsP
rocessingWcfService">
        <identity>
          <dns value="localhost" />
        </identity>
      </endpoint>
      <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
      <host>
        <baseAddresses>
          <add
baseAddress="http://localhost:8732/Design_Time_Addresses/ClaimsProcessingWcfServic
e/Service1/" />
        </baseAddresses>
      </host>
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior>
        <!-- To avoid disclosing metadata information,
set the value below to false and remove the metadata endpoint above
before deployment -->
        <serviceMetadata httpGetEnabled="True"/>
        <!-- To receive exception details in faults for debugging purposes,
set the value below to true. Set to false before deployment
to avoid disclosing exception information -->
        <serviceDebug includeExceptionDetailInFaults="False" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
</configuration>

```

The configuration related to WCF is contained in the `<system.ServiceModel>` xml element. The `<services>` element contains specifications for all of the WCF services that are configured using this App.config file. It is possible for a single application to host more than one WCF service, for example.

Each service is defined by a `<service>` element within the `<services>` node, and each service consists of a name, an optional `behaviorConfiguration` attribute, and a set of endpoints. The `behaviorConfiguration` attribute can be used to refer to a behavior in the `<serviceBehaviors>` element. This can be useful when a WCF-provided behavior, such as Metadata Exchange (MEX), is desired on some services but not on others.

Each <endpoint> element specifies an address that the service exposes a service contract on, the name of the service contract to invoke when receiving requests at this address, and the binding to use. Addresses are relative to any defined-base addresses, so in Example 6-60, the MEX endpoint is set to:

```
http://localhost:8732/Design_Time_Addresses/ClaimsProcessingWcfService/Service1/mex
```

The main service is exposed on the following endpoint:

```
http://localhost:8732/Design_Time_Addresses/ClaimsProcessingWcfService/Service1/
```

Example 6-60 Service definition showing mex endpoint

```
<service
name="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService.ClaimsProcessingWcfService">
  <endpoint address="" binding="wsHttpBinding"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService.IClaimsProcessingWcfService">
    <identity>
      <dns value="localhost" />
    </identity>
  </endpoint>
  <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"
/>
</service>
<host>
  <baseAddresses>
    <add
baseAddress="http://localhost:8732/Design_Time_Addresses/ClaimsProcessingWcfService/Service1/" />
  </baseAddresses>
</host>
</service>
```

The MEX behavior is covered in more depth in section 6.4, “Obtaining information about the service using MEX” on page 246.

Setting an appropriate base address

The base address generated by default is long and unwieldy. In this example, we set the base address to `http://localhost:8732/ClaimsProcessingWcfService/` by replacing the content of the <host> xml element in App.Config with the address definition shown in Example 6-61.

Example 6-61 Setting the base address for this service

```
<baseAddresses>
  <add baseAddress="http://localhost:8732/ClaimsProcessingWcfService/" />
</baseAddresses>
```

Removing the WSHttpBinding endpoint

For this example, we use a NetTcpBinding to expose the service through TCP/IP rather than the default web service based binding. The endpoint with a wsHttpBinding must therefore be removed by deleting the endpoint definition shown in Example 6-62 on page 233 from your App.Config file.

Example 6-62 Removing the endpoint using a wsHttpBinding

```
<endpoint address="" binding="wsHttpBinding"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService.IClaimsP
rocessingWcfService">
  <identity>
    <dns value="localhost" />
  </identity>
</endpoint>
```

Adding a NetTcpBinding

In this example, you configure WCF to use a TCP/IP-based protocol for communication between the client and the service. The WCF framework provides a suitable binding called `NetTcpBinding` for this purpose. The `NetTcpBinding` is suitable for communication between different machines over an existing tcp network and has the advantage of having a lighter serialization and deserialization cost than the `WebServices`-based bindings. This makes `NetTcpBinding` a good choice when performance is a key consideration.

To configure the `NetTcp` binding several key pieces of information are required:

- ▶ The endpoint address that the service is configured to use. For the `NetTcpBinding`, this consists of the following key components:
 - The `net.tcp://` URI prefix, which indicates that this is a `NetTcp` address.
 - The host name of the service or IP address to bind to. In this instance, you will use `localhost`.
 - The service to use for listening to connections. In this case, you will use `8523`.
 - Optionally a subpath indicating that this endpoint definition is only to be used for requests to some subset of the URI space at this address. In this example, you are not setting a path because only one service contract will be exposed to the client.

Tip: One use for setting a subpath for an endpoint is so that multiple services can listen on the same port. This is possible by enabling the `Net.Tcp` port sharing feature. More details about port sharing is at:

<http://msdn.microsoft.com/en-us/library/ms734772.aspx>

- ▶ A name for the binding, which is used when the binding configuration must be referred to in other parts of the `App.Config` file.
- ▶ The fully-qualified name of the service contract exposed by this endpoint. In this example, the contract is `Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService`.

To add a `NetTcp` endpoint listening on port `8523` on `localhost`, add the endpoint definition shown in Example 6-63 to the `<services>` element under the `<system.serviceModel>` tag.

Example 6-63 A NetTcpBindings endpoint definition

```
<endpoint address="net.tcp://localhost:8523" binding="netTcpBinding"
bindingConfiguration="" name="TcpIpEndPoint"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService" />
```

For this example, the default configuration of the `NetTcpBindings` is used, so the `bindingConfiguration` attribute is blank. If however the default configuration is not sufficient, the binding can be configured by providing a value for the `bindingConfiguration` attribute. When set, this attribute refers to a `<binding>` element with the same name. This `<binding>`

element can be used to hold additional configuration options and must be placed under the `system.ServiceModel/bindings/netTcpBinding` xml node. For example, to configure the endpoint to only select a single connection at a time, the `<bindings>` element shown in Example 6-64 is created. To enable this on the `TcpIpEndPoint`, set the `bindingConfiguration` attribute to `singleConnectionBinding`.

Example 6-64 Configuring a binding for `maxConnections = 1`

```
<bindings>
  <netTcpBinding>
    <binding name="singleConnectionBinding" maxConnections="1" />
  </netTcpBinding>
</bindings>
```

Some common options supported by the `NetTcpBindings` are shown in Table 6-19. For the complete list of supported options and additional documentation, see:

http://msdn.microsoft.com/en-us/library/system.servicemodel.nettcpbinding_properties.aspx

Table 6-19 Common `NetTcpBinding` properties

Property	Description
MaxConnections	The maximum number of accepted connections that can be queued awaiting dispatch on the server.
SendTimeout	The length of time that the server will wait for a write operation to the socket to complete before raising an exception.
ReceiveTimeout	The length of time that the server will wait for a read operation to the socket to complete before raising an exception.
ListenBacklog	The maximum number of connection requests that can be queued by the listener.
MaxReceivedMessageSize	The maximum size of message that this binding will accept.

Using the WCF Service Configuration Editor to configure the WCF Service

Microsoft Visual Studio includes a tool for configuring WCF services. You can use this tool to modify the WCF settings in the `App.Config` file:

1. Select **Tools** → **WCF Service Configuration Editor** from the menu.
2. After the WCF Service Configuration Editor loads, select **File** → **Open** → **Config File** from the menu.
3. Use the Open dialog to navigate to the `App.Config` file that is associated with your project, and click **Open**.

The WCF Service Configuration editor will load your `App.Config` file and display it.

4. Navigate to **Services** → **Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService** → **Endpoints** → **TcpIpEndPoint**.

You will see the configuration associated with this endpoint displayed in the editor as shown in Figure 6-10 on page 235.

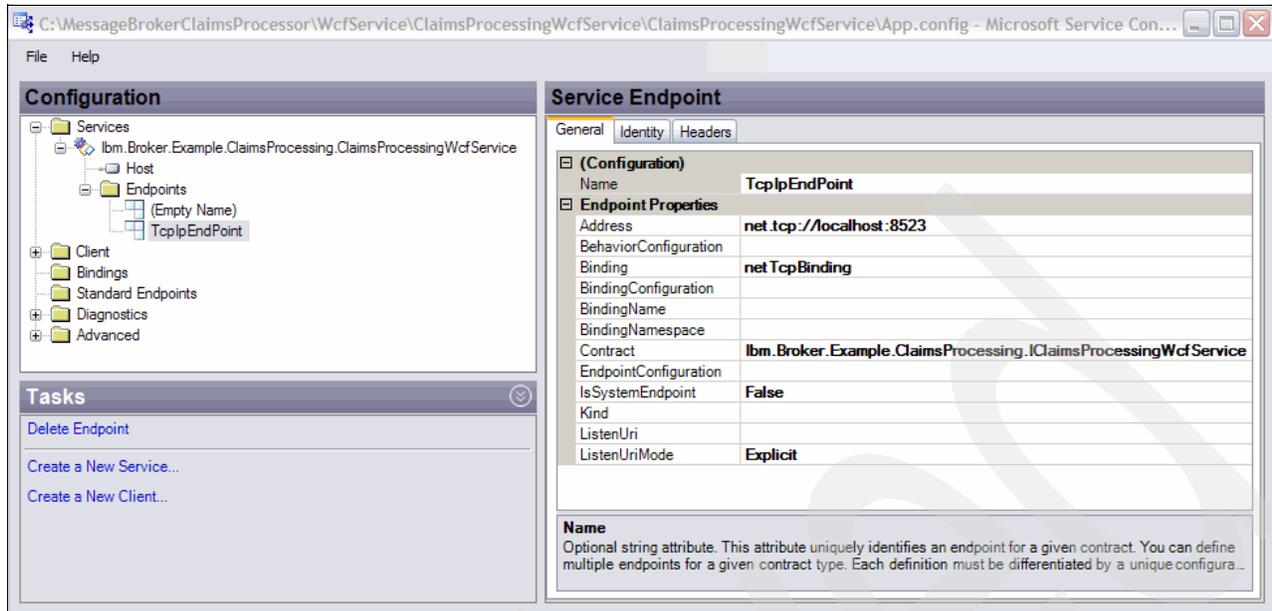


Figure 6-10 Viewing the `TcpIpEndPoint` in the WCF Service Configuration Editor

You can make changes to the configuration editor, which are reflected in the project's `App.Config` file when you save the file.

6.3.9 Testing the WCF Service using the WCF Test Client

Visual Studio provides a tool that can read the service contract of a WCF service and allow the user to invoke service operations by populating fields in a graphical tool. This tool is called the WCF Test Client and is launched automatically when you run the WCF Service from within Visual Studio.

To test the WCF Service:

1. Ensure that the `_BaseDir\customers` directory is empty (`_BaseDir` was defined in the example code as `c:\temp\testData`, so if you have not changed the path, ensure `c:\temp\testData\customers` is empty). This makes sure that the service will generate the example data when it starts. If you do not clear the directory, you can still use the WCF Test Client to test the service; however, the results from some of the operations can vary.
2. Select **Debug** → **Start without Debugging** from the Visual Studio menu.

Administrator privileges might be required: Depending on your version of windows and your user configuration, you might need to log on as Administrator or start Visual Studio with elevated privileges using the **Run as Admin** right-click context menu option.

Visual Studio automatically hosts the WCF Service for you. An information bubble is briefly displayed in the system tray to indicate this, as shown in Figure 6-11 on page 236.

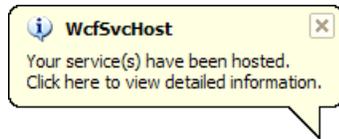


Figure 6-11 Visual Studio automatically hosts your WCF Service when you run the project

3. The WCF Test Client also automatically launches. The WCF Test Client generates the list of known operations using the MEX endpoint. You will see that there is a Service node corresponding to this services mex URI. Fully expanding this node of the tree shows all of the operations defined by this service, as shown in Figure 6-12.

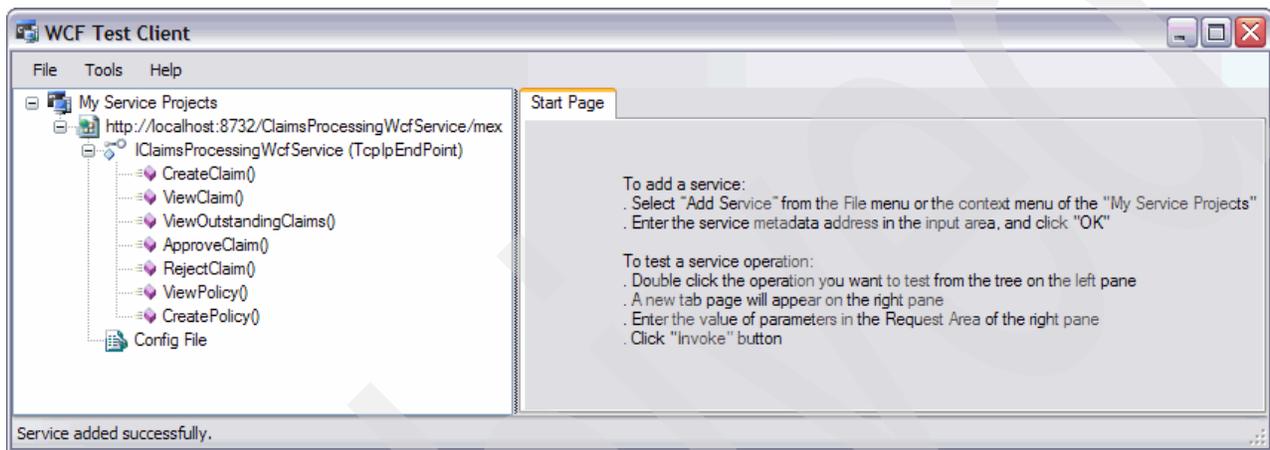


Figure 6-12 The WCF Test Client

4. To test the CreatePolicy operation:
 - a. Double-click the CreatePolicy operation.
 - b. Fill in the value of each field in the Request panel.

Selecting a field that corresponds to one of the defined datatypes allows you to select the appropriate type from a drop-down menu. For example, in the Value Column for the PrimaryCustomer field, clicking the box allows you to select the Customer datatype, which is then expanded to allow you to populate the fields appropriate to that datatype.

Fields that take a list or collection start off with the value length=0, which refers to the number of objects in the collection. Altering this to read length=1, for example, causes the WCF Client to automatically generate space to input a single object.

Your completed CreatePolicy Request panel now looks like the example in Figure 6-13 on page 237.

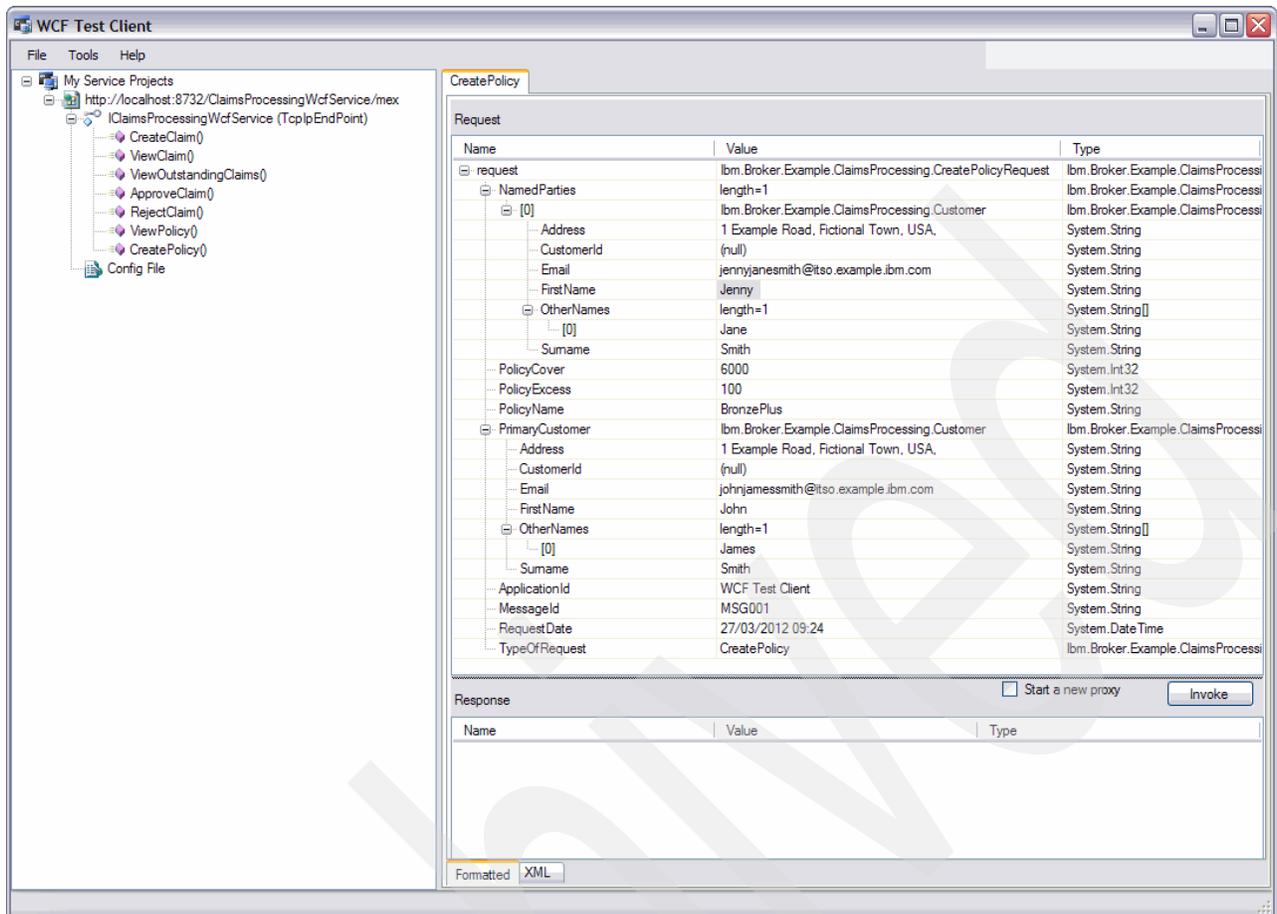


Figure 6-13 Testing the CreatePolicy Operation

- c. Click the **Invoke** button.

The request is sent to the WCF service, and the response is displayed in the Response panel, as shown in figure Figure 6-14 on page 238.

(return)		Ibm.Broker.Example.ClaimsProcessing.CreatePolic
Policy		Ibm.Broker.Example.ClaimsProcessing.InsuranceP
NamedParties	length=1	Ibm.Broker.Example.ClaimsProcessing.Customer[]
[0]		Ibm.Broker.Example.ClaimsProcessing.Customer
Address	"1 Example Road, Fictional Town, USA,"	System.String
CustomerId	"CUST265039c4dc934c7abf3e08bd11ffce07"	System.String
Email	"jennyjanesmith@itso.example.ibm.com"	System.String
FirstName	"Jenny"	System.String
OtherNames	length=1	System.String[]
[0]	"Jane"	System.String
Surname	"Smith"	System.String
PolicyCover	6000	System.Int32
PolicyExcess	100	System.Int32
PolicyExpiryDate	27/03/2013 09:24:00	System.DateTime
PolicyName	"BronzePlus"	System.String
PolicyNumber	"POLICY682422a48abc4e40a56c6aa1341fec"	System.String
PolicyStartDate	27/03/2012 09:24:00	System.DateTime
Policyholder		Ibm.Broker.Example.ClaimsProcessing.Customer
Address	"1 Example Road, Fictional Town, USA,"	System.String
CustomerId	"CUST02c78a60d86142f09eb406dcf998c7a8"	System.String
Email	"johnjamesmith@itso.example.ibm.com"	System.String
FirstName	"John"	System.String
OtherNames	length=1	System.String[]
[0]	"James"	System.String
Surname	"Smith"	System.String
Underwriter	"ItsoInsurancePartner1"	System.String
CorrelationId	"MSG001"	System.String
ResponseTime	27/03/2012 09:36:39	System.DateTime
TypeOfResponse	CreatePolicy	Ibm.Broker.Example.ClaimsProcessing.ResponseT

Figure 6-14 The response from the CreatePolicy Operation

Tip: The TypeOfRequest was initially set incorrectly in the dialog box because in WCF the constructors for objects are not run when serialization / deserialization takes place. Because the field does not access a member variable but rather returns a constant for each class of request it is actually possible to successfully make the request even if the TypeOfRequest field is set incorrectly in the WCF Test Client.

Because the CustomerIDs for the two customers created were left null, new IDs were generated for them by the system. Similarly a new ID was generated for the policy itself.

5. To test the CreateClaim:
 - d. In the WCF Test Client, double-click the CreateClaim operation. A new tab is created in the right panel for this operation.
 - e. Fill in the Request Panel, as shown in Figure 6-15 on page 239.

request	Ibm.Broker.Example.ClaimsProcessing.NewClaim	Ibm.Broker.Example.ClaimsProcessing.NewClaimR
NewClaim	Ibm.Broker.Example.ClaimsProcessing.Claim	Ibm.Broker.Example.ClaimsProcessing.Claim
ClaimDate	22/03/2012 12:21:00	System.DateTime
ClaimReceivedDate	27/03/2012 09:47	System.DateTime
ClaimReference	(null)	System.String
ClaimValue	200	System.Int32
Customer	Ibm.Broker.Example.ClaimsProcessing.Custome	Ibm.Broker.Example.ClaimsProcessing.Customer
Address	1 Generic Street, Any Town, USA	System.String
CustomerId	CUST00001	System.String
Email	john.doe@itso.example.ibm.com	System.String
FirstName	John	System.String
OtherNames	length=2	System.String[]
[0]	James	System.String
[1]	Jeffrey	System.String
Surname	Doe	System.String
Description	Windscreen chipped while driving on the highw	System.String
Notes	(null)	System.String
PolicyNumber	POLICY0001	System.String
Status	Submitted	Ibm.Broker.Example.ClaimsProcessing.ClaimState
ApplicationId	WCF Test Client	System.String
MessageId	MSG002	System.String
RequestDate	27/03/2012 09:46	System.DateTime
TypeOfRequest	NewClaim	Ibm.Broker.Example.ClaimsProcessing.RequestTyp

Figure 6-15 CreateClaim Request

- f. Click the **Invoke** button.

The response is displayed in the Response panel, as shown in Figure 6-16.

(return)	Ibm.Broker.Example.ClaimsProcessing.NewClaim	Ibm.Broker.Example.ClaimsProcessing.NewClaimR
CorrelationId	"MSG002"	System.String
ResponseTime	27/03/2012 09:52:11	System.DateTime
TypeOfResponse	NewClaim	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-16 Create Claim Response

Tip: If you make a mistake supplying the data, it is possible for the Proxy object to enter the Faulted state. If this happens, select the Start a new proxy check box before invoking the operation again.

6. The newly created claim refers to an already existing policy POLICY0001. To view this policy, test the ViewPolicy operation by following these steps:
 - a. Select the **ViewPolicy** operation.
 - b. Fill in the request panel, as shown in Figure 6-17.

request	Ibm.Broker.Example.ClaimsProcessing.ViewP	Ibm.Broker.Example.ClaimsProcessing.ViewPolicy
PolicyReference	POLICY0001	System.String
ApplicationId	WCF Test Client	System.String
MessageId	MSG003	System.String
RequestDate	27/03/2012 09:58	System.DateTime
TypeOfRequest	ViewPolicy	Ibm.Broker.Example.ClaimsProcessing.RequestTyp

Figure 6-17 ViewPolicy Request

- c. Click the **Invoke** button. The WCF Service will return the response shown in Figure 6-18 on page 240.

(return)		Ibm.Broker.Example.ClaimsProcessing.ViewPolicy
Policy		Ibm.Broker.Example.ClaimsProcessing.InsuranceP
NamedParties	length=2	Ibm.Broker.Example.ClaimsProcessing.Customer[
[0]		Ibm.Broker.Example.ClaimsProcessing.Customer
[1]		Ibm.Broker.Example.ClaimsProcessing.Customer
PolicyCover	1000	System.Int32
PolicyExcess	150	System.Int32
PolicyExpiryDate	27/03/2013 09:34:00	System.DateTime
PolicyName	(null)	NullObject
PolicyNumber	"POLICY0001"	System.String
PolicyStartDate	27/03/2012 09:34:00	System.DateTime
Policyholder		Ibm.Broker.Example.ClaimsProcessing.Customer
Address	"1 Generic Street, Any Town, USA"	System.String
CustomerId	"CUST00001"	System.String
Email	"johndoe@tso.example.ibm.com"	System.String
FirstName	"John"	System.String
OtherNames	length=2	System.String[]
Surname	"Doe"	System.String
Underwriter	"Partner Insurance Company A"	System.String
CorrelationId	"MSG003"	System.String
ResponseTime	27/03/2012 10:40:29	System.DateTime
TypeOfResponse	ViewPolicy	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-18 ViewPolicy Response

7. There are now two claims in the system. To view the predefined claim, CLAIM0001, follow these steps:
 - a. Double-click the ViewClaim Operation
 - b. Fill in the request parameters, as shown in Figure 6-19. Note that we are passing in the claim reference CLAIM0001.

request	Ibm.Broker.Example.ClaimsProcessing.ViewCl	Ibm.Broker.Example.ClaimsProcessing.ViewClaim
ClaimReference	CLAIM0001	System.String
ApplicationId	WCF Test Client	System.String
MessageId	MSG004	System.String
RequestDate	27/03/2012 10:47	System.DateTime
TypeOfRequest	ViewClaim	Ibm.Broker.Example.ClaimsProcessing.RequestTy

Figure 6-19 ViewClaim Request

- c. Click the **Invoke** button. The WCF Service will return the response shown in Figure 6-20 on page 241.

(return)		Ibm.Broker.Example.ClaimsProcessing.ViewClaim
Claim		Ibm.Broker.Example.ClaimsProcessing.Claim
ClaimDate	07/03/2012 09:34:00	System.DateTime
ClaimReceivedDate	27/03/2012 09:34:00	System.DateTime
ClaimReference	"CLAIM0001"	System.String
ClaimValue	400	System.Int32
Customer		Ibm.Broker.Example.ClaimsProcessing.Customer
Address	"1 Generic Street, Any Town, USA"	System.String
CustomerId	"CUST00003"	System.String
Email	"janedoe@tso.example.ibm.com"	System.String
FirstName	"Jane"	System.String
OtherNames	length=2	System.String[]
Surname	"Doe"	System.String
Description	"Accidental damage to car, no other parties in	System.String
Notes	(null)	NullObject
PolicyNumber	"POLICY0001"	System.String
Status	Submitted	Ibm.Broker.Example.ClaimsProcessing.ClaimState
CorrelationId	"MSG004"	System.String
ResponseTime	27/03/2012 10:47:25	System.DateTime
TypeOfResponse	ViewClaim	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-20 ViewClaim Response

8. All outstanding claims in the system can be viewed using the ViewOutstandingClaims operation. You can use this operation to determine the ClaimReference's of these claims so that we can use the ApproveClaim operation on one of the claims and the RejectClaim operation on the other. To run the ViewOutstandingClaims operation:
 - a. Double-click the ViewOutstandingClaims operation.
 - b. Fill in the Request panel, as shown in Figure 6-21.

request	Ibm.Broker.Example.ClaimsProcessing.ViewOutstandingClaims	Ibm.Broker.Example.ClaimsProcessing.ViewOutstandingClaimsRequest
ApplicationId	WCF Test Client	System.String
MessageId	MSG005	System.String
RequestDate	27/03/2012 10:53	System.DateTime
TypeOfRequest	ViewOutstandingClaims	Ibm.Broker.Example.ClaimsProcessing.RequestType

Figure 6-21 ViewOutstandingClaims Request

- c. Click the **Invoke** button. The WCF Service will return the response shown in Figure a on page 242.

(return)		IBM.Broker.Example.ClaimsProcessing.Vi
OutstandingClaims	length=2	IBM.Broker.Example.ClaimsProcessing.Cl
[0]		IBM.Broker.Example.ClaimsProcessing.Cl
ClaimDate	07/03/2012 09:34:00	System.DateTime
ClaimReceivedDate	27/03/2012 09:34:00	System.DateTime
ClaimReference	"CLAIM0001"	System.String
ClaimValue	400	System.Int32
Customer		IBM.Broker.Example.ClaimsProcessing.Cl
Description	"Accidental damage to car, no other parties involved"	System.String
Notes	(null)	NullObject
PolicyNumber	"POLICY0001"	System.String
Status	Submitted	IBM.Broker.Example.ClaimsProcessing.Cl
[1]		IBM.Broker.Example.ClaimsProcessing.Cl
ClaimDate	22/03/2012 12:21:00	System.DateTime
ClaimReceivedDate	27/03/2012 09:47:00	System.DateTime
ClaimReference	"CLAIMfc933957413b4594824e1f76d1b74fbb"	System.String
ClaimValue	200	System.Int32
Customer		IBM.Broker.Example.ClaimsProcessing.Cl
Description	"Windscreen chipped while driving on the highway."	System.String
Notes	(null)	NullObject
PolicyNumber	"POLICY0001"	System.String
Status	Submitted	IBM.Broker.Example.ClaimsProcessing.Cl
CorrelationId	"MSG005"	System.String
ResponseTime	27/03/2012 10:56:02	System.DateTime
TypeOfResponse	ViewOutstandingClaims	IBM.Broker.Example.ClaimsProcessing.R

Figure 6-22 ViewOutstandingClaims Response

- d. Take a note of the two ClaimReference values. One of these will be the predefined claim CLAIM0001, but the other one will have a uniquely generated claim reference number. In the example shown, this is CLAIMfc933957413b4594824e1f76d1b74fbb; however, this will be different on your system.
9. We will use the ApproveClaim method to approve the predefined claim CLAIM0001. To use the WCF Test Client to execute the ApproveClaim operation follow these steps:
 - a. Double-click the ApproveClaim operation in the WCF Test Client.
 - b. Fill in the Request panel, as shown in Figure 6-23.

request		IBM.Broker.Example.ClaimsProcessing.Approv	IBM.Broker.Example.ClaimsProcessing.ApproveCl
AgentReference	Agent001	System.String	
ClaimReference	CLAIM0001	System.String	
ValueApproved	400	System.Int32	
ApplicationId	WCF Test Client	System.String	
MessageId	MSG006	System.String	
RequestDate	27/03/2012 11:00	System.DateTime	
TypeOfRequest	ApproveClaim	IBM.Broker.Example.ClaimsProcessing.RequestTy	

Figure 6-23 ApproveClaim Request

- c. Click the **Invoke** button. The WCF Service generates the response shown in Figure 6-24 on page 243.

(return)		Ibm.Broker.Example.ClaimsProcessing.ApproveCl
Payment		Ibm.Broker.Example.ClaimsProcessing.Payment
Amount	250	System.Int32
Payee		Ibm.Broker.Example.ClaimsProcessing.Customer
Address	"1 Generic Street, Any Town, USA"	System.String
CustomerId	"CUST00003"	System.String
Email	"janedoe@itso.example.ibm.com"	System.String
FirstName	"Jane"	System.String
OtherNames	length=2	System.String[]
[0]	"Jill"	System.String
[1]	"Jenny"	System.String
Surname	"Doe"	System.String
Underwriter	"Partner Insurance Company A"	System.String
CorrelationId	"MSG006"	System.String
ResponseTime	27/03/2012 11:01:51	System.DateTime
TypeOfResponse	ApproveClaim	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-24 ApproveClaim Response

- d. The service generated the payment value of 250 by subtracting the value of the excess applicable to this policy (150) from the total value of the claim (400). As you can see, this means that the service had to refer to the policy referenced by the claim because we did not pass in the value for the excess as part of the request.
 - e. The service also generated the payment for the correct Underwriter Partner Insurance Company A.
10. Reject the second claim using the RejectClaim operation. Refer to the claim reference number. To use the WCF Test Client to invoke the RejectClaim operation:
- a. Double-click the RejectClaim operation.
 - b. Fill in the Request Panel, as shown in Figure 6-25. Substitute the claim reference value CLAIMfc933957413b4594824e1f76d1b74fbb with the unique value generated on your system.

request		Ibm.Broker.Example.ClaimsProcessing.RejectClaimRe
AgentReference	Agent001	System.String
ClaimReference	CLAIMfc933957413b4594824e1f76d1b74fbb	System.String
Reason	Insufficient description of incident.	System.String
ApplicationId	WCF Test Client	System.String
MessageId	MSG007	System.String
RequestDate	27/03/2012 11:09	System.DateTime
TypeOfRequest	RejectClaim	Ibm.Broker.Example.ClaimsProcessing.Re

Figure 6-25 RejectClaim Request

- c. Click the **Invoke** button. The WCF Service will return the response shown in Figure 6-26.

(return)		Ibm.Broker.Example.ClaimsProcessing.RejectClai
CorrelationId	"MSG007"	System.String
ResponseTime	27/03/2012 11:10:58	System.DateTime
TypeOfResponse	RejectClaim	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-26 RejectClaimResponse

11. To confirm that the state of the claims was updated, invoke the ViewOutstandingClaims operation again by repeating the instruction in step 8 on page 241. If you do so, you will see that the Claims no longer appear in the list of outstanding claims, as shown in Figure 6-27 on page 244.

return		
OutstandingClaims	length=0	Ibm.Broker.Example.ClaimsProcessing.ViewOutsta
CorrelationId	(null)	Ibm.Broker.Example.ClaimsProcessing.Claim[]
ResponseTime	27/03/2012 11:14:50	System.DateTime
TypeOfResponse	ViewOutstandingClaims	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-27 There are no outstanding claims in the system

12. You can use the ViewClaim operation on either of these claims by repeating the instructions in step 7 on page 240. For example you can see that the Status of CLAIM0001 is now approved, as shown in Figure 6-28.

return		
Claim		Ibm.Broker.Example.ClaimsProcessing.Claim
ClaimDate	07/03/2012 09:34:00	System.DateTime
ClaimReceivedDate	27/03/2012 09:34:00	System.DateTime
ClaimReference	"CLAIM0001"	System.String
ClaimValue	400	System.Int32
Customer		Ibm.Broker.Example.ClaimsProcessing.Customer
Description	"Accidental damage to car, no other parties in	System.String
Notes	"\n[27/03/2012 11:01:51] AgentRef: Agent00	System.String
PolicyNumber	"POLICY0001"	System.String
Status	Approved	Ibm.Broker.Example.ClaimsProcessing.ClaimState
CorrelationId	"MSG004"	System.String
ResponseTime	27/03/2012 11:17:38	System.DateTime
TypeOfResponse	ViewClaim	Ibm.Broker.Example.ClaimsProcessing.Response

Figure 6-28 CLAIM0001 is now Approved

13. So far you tested the service for operations that complete successfully. You can test the fault behavior of the system by supplying data that generates a ClaimsProcessingFault by following these steps:

- Double-click the ViewClaim operation.
- Fill out the Request panel so that it refers to a claim that does not exist in the system, as shown in Figure 6-29.

request		
ClaimReference	CLAIM_INVALID_01	Ibm.Broker.Example.ClaimsProcessing.ViewClaim
ApplicationId	WCF Test Client	Ibm.Broker.Example.ClaimsProcessing.ViewClaim
MessageId	MSG008	System.String
RequestDate	27/03/2012 11:20	System.DateTime
TypeOfRequest	ApproveClaim	Ibm.Broker.Example.ClaimsProcessing.RequestTy

Figure 6-29 Using the ViewClaim operation on an invalid Claim Reference

- Click the **Invoke** button.
- The WCF Client displays an error box to indicate that a Fault occurred. The Error Details indicate that the reason for the fault is that the claim does not exist in the system, as shown in Figure 6-30 on page 245.

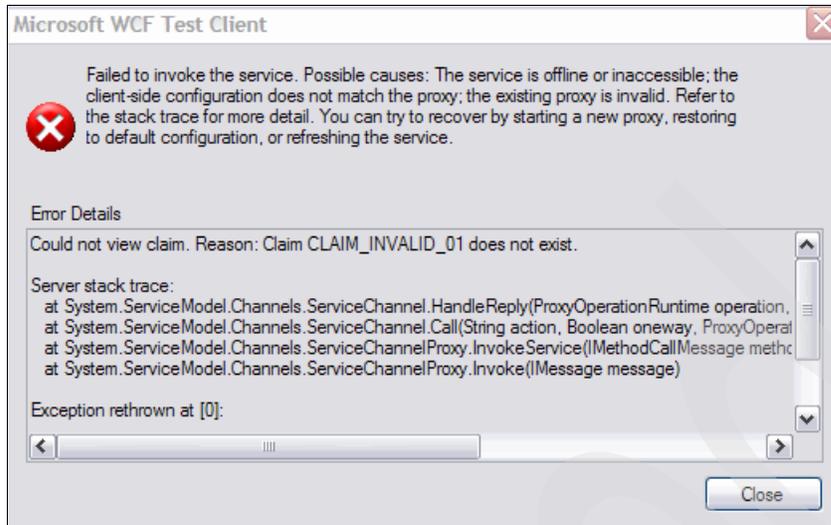


Figure 6-30 A Fault occurred during processing

The WCF Test Client can be a useful tool for developing a WCF service because it allows developers to test and troubleshoot the service without needing to develop a client. The WCF Test Client also displays the ABC specification that it is using to communicate with the service in the Config File node. You can view this configuration by double-clicking the Config File node from the tree on the left of the WCF Test Client, as shown in Figure 6-31.

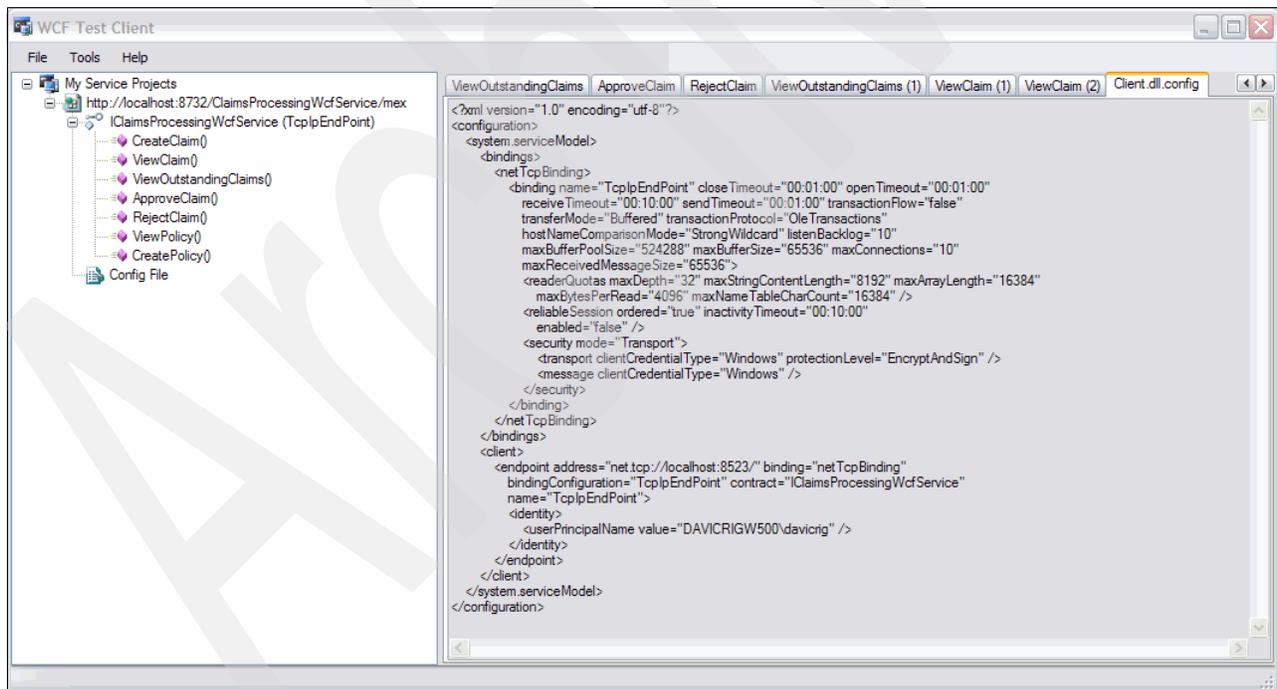


Figure 6-31 Viewing the configuration that was used by the WCF Test Client

Exit the debug mode by selecting **Debug** → **Stop Debugging** in the Visual Studio menu. For now, leave the service running.

6.4 Obtaining information about the service using MEX

WCF exposes a facility called MetaData Exchange (MEX) that allows ABC information to be obtained by querying the service itself. This facility allows clients to automatically generate the code required to interact with the WCF service.

MEX is automatically enabled by the New Project Wizard when the WCF service library is created. The endpoint definition shown in Example 6-65 specifies that MEX is active over the HTTP transport at the address mex.

Example 6-65 MEX endpoint definition in App.Config

```
<endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
```

The address is relative to the base address of the service, shown in the host definition in Example 6-66, so the full address of the MEX endpoint is:

```
http://localhost:8732/ClaimsProcessingWcfService/mex
```

Example 6-66 Base address definition in App.Config

```
<host>
  <baseAddresses>
    <add baseAddress="http://localhost:8732/ClaimsProcessingWcfService/" />
  </baseAddresses>
</host>
```

The `IMetadataExchange` contract is a service contract implemented by the .NET WCF classes. For the metadata exchange to work correctly the service may either implement the `MetadataReference` contract or add the `serviceMetadata` behavior to the `App.config` file. The New Project Wizard automatically added this behavior, as shown in Example 6-67.

Example 6-67 The serviceMetadata behavior in App.Config

```
<serviceMetadata httpGetEnabled="true"/>
```

Navigating to the base address of the application with a web browser returns a web page that provides instructions about how to automatically generate a client to call the service, as shown in Figure 6-32 on page 247.

<http://localhost:8732/ClaimsProcessingWcfService/>

ClaimsProcessingWcfService Service

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using the svcutil.exe tool from the command line with the following syntax:

```
svcutil.exe http://localhost:8732/ClaimsProcessingWcfService/?wsdl
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your client application and use the generated client class to call the Service. For example:

C#

```
class Test
{
    static void Main()
    {
        ClaimsProcessingWcfServiceClient client = new ClaimsProcessingWcfServiceClient();

        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

Visual Basic

```
Class Test
    Shared Sub Main()
        Dim client As ClaimsProcessingWcfServiceClient = New ClaimsProcessingWcfServiceClient()
        ' Use the 'client' variable to call operations on the service.

        ' Always close the client.
        client.Close()
    End Sub
End Class
```

Figure 6-32 A MEX enabled WCF service can display client creation instructions at the Base Address

When MEX is enabled for a WCF service and the `httpGetEnabled` is set to true, a WSDL document describing the endpoint is also published. The address that the WSDL is published on is by default equal to the base address plus the `?wsdl` suffix.

For this service the WSDL publication address is:

```
http://localhost:8732/ClaimsProcessingWcfService/?wsdl
```

The WSDL document is useful for both generating .NET service to consume this WCF service and in the case of web services-based bindings enabling other clients to consume the service. Figure 6-33 on page 248 shows the output of the WSDL displayed in a web browser.

```

<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions name="ClaimsProcessingWcfService" targetNamespace="http://tempuri.org/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" xmlns:tns="http://tempuri.org/"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
  xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
  xmlns:wsa10="http://www.w3.org/2005/08/addressing"
  xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata">
- <wsp:Policy wsu:Id="TcpIpEndPoint_policy">
- <wsp:ExactlyOne>
- <wsp:All>
  <msb:BinaryEncoding
    xmlns:msb="http://schemas.microsoft.com/ws/06/2004/mspolicy/netbinary1" />
- <sp:TransportBinding
  xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
- <wsp:Policy>
- <sp:TransportToken>
- <wsp:Policy>
- <msf:WindowsTransportSecurity
  xmlns:msf="http://schemas.microsoft.com/ws/2006/05/framing/policy">
  <msf:ProtectionLevel>EncryptAndSign</msf:ProtectionLevel>
</msf:WindowsTransportSecurity>
</wsp:Policy>
</sp:TransportToken>
- <sp:AlgorithmSuite>
- <wsp:Policy>
  <sp:Basic256 />
</wsp:Policy>
</sp:AlgorithmSuite>
- <sp:Layout>
- <wsp:Policy>
  <sp:Strict />
</wsp:Policy>

```

Figure 6-33 A MEX enabled WCF service will respond to ?WSDL requests over http

Enabling MEX is required to automatically generate WCF clients and to use the WCF test client although it can be viewed as a security exposure in a production environment. It is therefore common practice to remove the serviceMetadata behavior and mex endpoint definitions prior to deployment in a production environment.

6.5 Generating WCF client code from a MEX enabled WCF service

Visual Studio includes a tool called SVCUtil that can automatically generate the client side of a service contract based on the metadata provided by an MEX enabled endpoint. To generate client code:

1. Ensure that the WCF Service is running. In this case, the service was started during the debug exercise in 6.3.9, "Testing the WCF Service using the WCF Test Client" on page 235 and is running in Visual Studio.

2. Start a new Visual Studio Command prompt by selecting **Start → Microsoft Visual Studio 2010 → Visual Studio Tools → Visual Studio Command Prompt (2010)**.
3. Navigate to a location on disk where you want to store the output files using the `cd` command.
4. Run the `svcutil` command against the WSDL address of the service, as shown in Example 6-68.

Example 6-68 Running SVCUtil against the running WCF service

```
svcutil.exe http://localhost:8732/ClaimsProcessingWcfService/?wsdl
```

The `svcutil` command produces two files, `ClaimsProcessingWcfService.cs` and `output.config`.

The `ClaimsProcessingWcfService.cs` file can be added to .NET projects to allow a client to create a proxy object for the service. Opening this file in a text editor will show that the content is different from the `ClaimsProcessingWcfService.cs` file used to actually implement the service on the server, as shown in the extract in Example 6-69.

Example 6-69 ClaimsProcessingWcfService.cs output from svcutil

```
//-----
// <auto-generated>
//   This code was generated by a tool.
//   Runtime Version:4.0.30319.239
//
//   Changes to this file may cause incorrect behavior and will be lost if
//   the code is regenerated.
// </auto-generated>
//-----

namespace IBM.Broker.Example.ClaimsProcessing
{
    using System.Runtime.Serialization;

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Runtime.Serialization",
    "4.0.0.0")]
    [System.Runtime.Serialization.DataContractAttribute(Name="NewClaimRequest",
    Namespace="http://schemas.datacontract.org/2004/07/IBM.Broker.Example.ClaimsProcessing")]
    public partial class NewClaimRequest : object,
    System.Runtime.Serialization.IExtensibleDataObject
    {
        private System.Runtime.Serialization.ExtensionDataObject
    extensionDataField;

        private string ApplicationIdField;

        private string MessageIdField;

        private IBM.Broker.Example.ClaimsProcessing.Claim NewClaimField;
```

```

private System.DateTime RequestDateField;

public System.Runtime.Serialization.ExtensionDataObject ExtensionData
{
    get
    {
        return this.extensionDataField;
    }
    set
    {
        this.extensionDataField = value;
    }
}

```

The details contained in this file do not need to be understood by the consumer of the WCF service. Instead the consumer creates an object based on this generated class and can use the object exactly as though it was a local object with WCF automatically managing the required serialization and deserialization on the wire. Just like a real local object, a developer working with a WCF client can use content assist and automatic code completion to examine the service contract.

The svcutil utility also generates an output.config file that contains the endpoint, bindings and contract details required to consume the service. Because the svcutil utility ran against localhost, the address of the endpoints are relative to localhost. These can be easily edited when deploying the application to a remote system without rebuilding. As shown in Example 6-70, the svcutil tool also adds default values for transport specific parameters, such as TCP/IP based timeouts.

Example 6-70 Output.config

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="TcpIpEndPoint" closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00" sendTimeout="00:01:00" transactionFlow="false"
          transferMode="Buffered" transactionProtocol="OleTransactions"
          hostNameComparisonMode="StrongWildcard" listenBacklog="10"
          maxBufferPoolSize="524288" maxBufferSize="65536" maxConnections="10"
          maxReceivedMessageSize="65536">
          <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
          <security mode="Transport">
            <transport clientCredentialType="Windows" protectionLevel="EncryptAndSign" />
            <message clientCredentialType="Windows" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
    <client>
      <endpoint address="net.tcp://localhost:8523/" binding="netTcpBinding"
        bindingConfiguration="TcpIpEndPoint" contract="IClaimsProcessingWcfService"
        name="TcpIpEndPoint">

```

```
<identity>
  <userPrincipalName value="DAVICRIGW500\davicrig" />
</identity>
</endpoint>
</client>
</system.serviceModel>
</configuration>
```

We use these two files in Chapter 7, “Scenario: Integrating Windows Communication Foundation in message flows - Part 2” on page 273 to build a .NETCompute node that can consume the ClaimsProcessingWcfService.

6.6 WCF hosting options

One of the advantages of WCF is that the same service can be deployed or hosted in a number of different ways without modification. This allows the user to choose the most appropriate hosting option for a broad range of tasks. Some hosting options, such as Visual Studio's built in host or hosting from a command line application, suit development and test activities; whereas, in production environments the resiliency and the uniform administrative interfaces of IIS or Windows Services can be leveraged by hosting the service within one of these run time environments. In this section, we use the ClaimsProcessingWcfService to demonstrate the hosting flexibility provided by WCF.

6.6.1 Hosting using Visual Studio

When you use the Debug menu in Visual Studio to launch the ClaimsProcessingWcfService, Visual Studio automatically hosts your WCF service for you. Visual Studio continues to host your service until the WCF Test Client is closed or until you terminate the application.

This form of hosting is extremely useful when developing a service because it allows the rapid testing of any changes in the test client. It has the downside that Visual Studio must be running to host the service. This is clearly not suitable for production deployment but there are also instances where hosting with Visual Studio might not be desirable, such as when automating the testing of the service.

6.6.2 Hosting using a stand-alone application

It is possible to host a WcfService from any stand-alone .NET application, which is achieved by instantiating a ServiceHost object passing in the type of the Service Library you created, as shown in Example 6-71.

Example 6-71 Instantiating a new ServiceHost

```
ServiceHost serviceHost = new ServiceHost(typeof(ClaimsProcessingWcfService))
```

To create a console based host:

1. Start a new instance of Visual Studio.
2. Create a new project:
 - a. Select **File** → **New** → **Project**.
 - b. Select **Visual C#** → **Windows** from the Installed Templates.

- c. Select **Console Application** from the middle panel.
- d. Enter ClaimsProcessingConsoleHost in the Name field.
- e. Enter ClaimsProcessingConsoleHostSolution in the Solution name field.
- f. Select a directory on the file system to store your solution in the Location field.
- g. Ensure that the Create directory for solution option is selected.

The completed New Project Wizard now looks like the example in Figure 6-34.

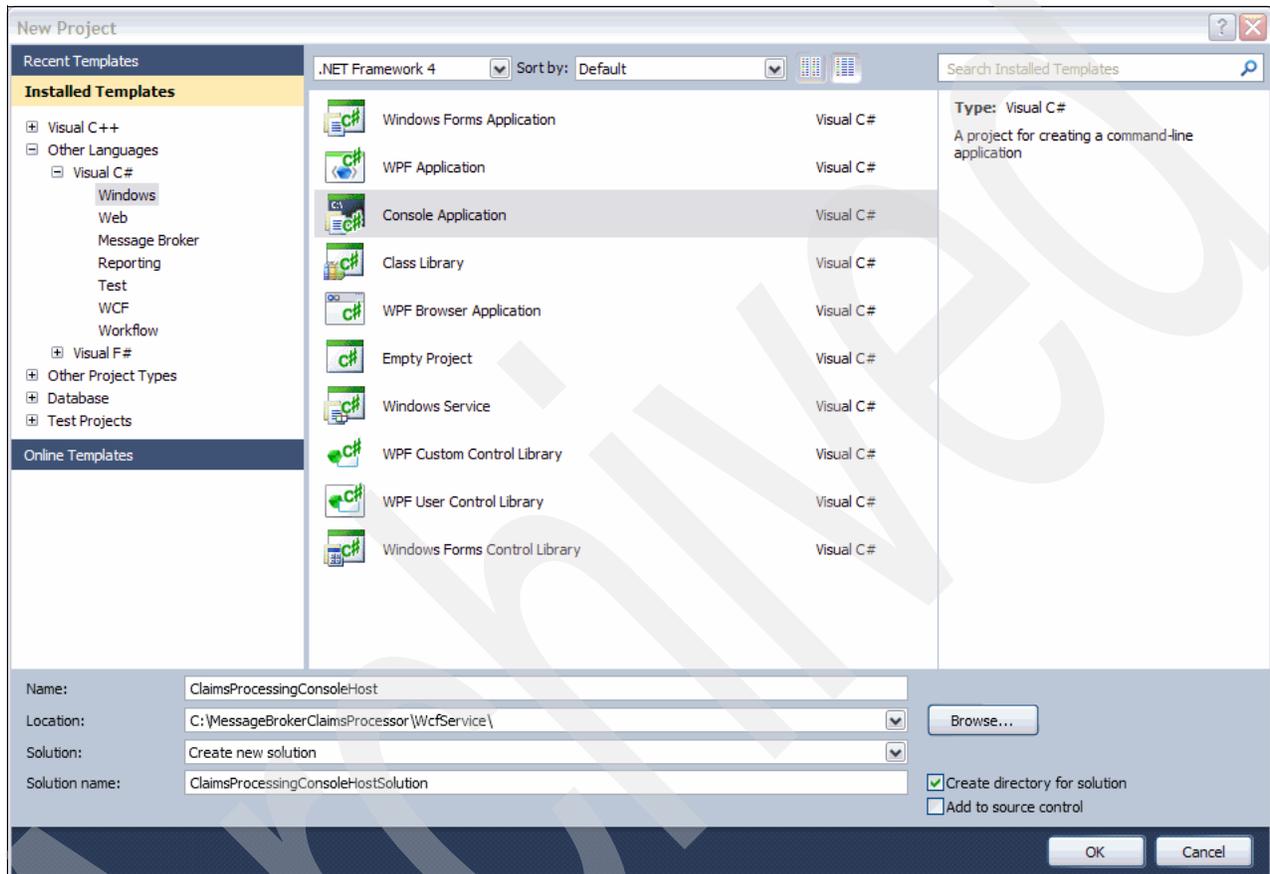


Figure 6-34 Creating a new Console Application to host the WCF service

- h. Click **OK**.

The New Project Wizard creates a new class called Program in the Program.cs file.

3. Rename the class to ClaimsProcessingConsoleHost and the Program.cs implementation file to ClaimsProcessingConsoleHost.cs.
4. Save ClaimsProcessingConsoleHost.cs.
5. To have access to the WCF framework and the ClaimsProcessingService, you need to add their references to the project:
 - a. Right-click the References folder in the Solution Explorer, and select **Add Reference**.
 - b. From the .NET tab select **System.ServiceModel**, and click the **OK** button.
 - c. Right-click the References folder, and select **Add Reference**.
 - d. From the Browse tab, navigate to the directory on the disk that holds the ClaimsProcessingWcfService.dll file from your completed WCF Service, as shown in Figure 6-35 on page 253.

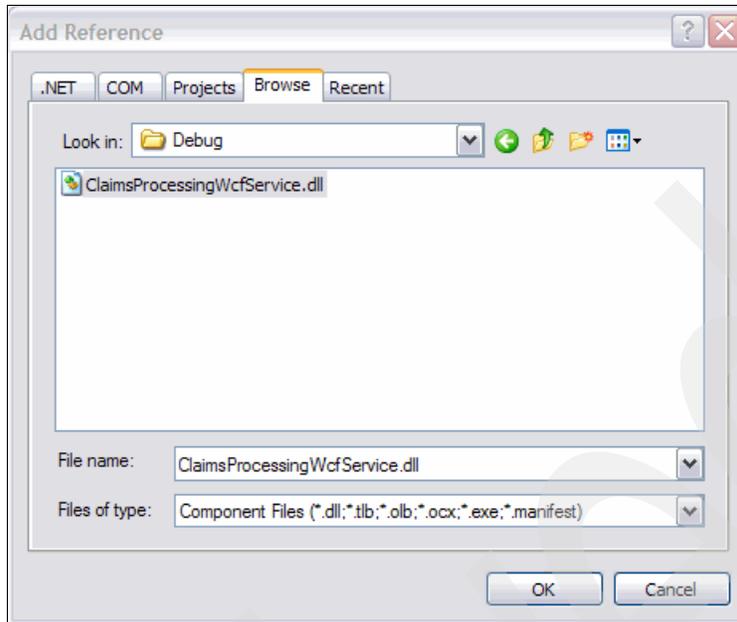


Figure 6-35 Adding a reference to the ClaimsProcessingWcfService assembly

- e. Click the **OK** button.
6. Modify the code for the ClaimsProcessingConsoleHost.cs file so that it matches the code shown in Example 6-72.

Example 6-72 ClaimsProcessingConsoleHost.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.ServiceModel;
using System.ServiceModel.Description;
using IBM.Broker.Example.ClaimsProcessing;

namespace ClaimsProcessingConsoleHost
{
    class ClaimsProcessingConsoleHost
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Starting console host for
ClaimsProcessingWcfService");

            //instantiate a ServiceHost
            using (ServiceHost serviceHost = new
ServiceHost(typeof(ClaimsProcessingWcfService)))
            {
                //Open the service so that it start listenign for request
                serviceHost.Open();

                //Readline() blocks for input so service will continue tor un until
                //the user presses Enter
            }
        }
    }
}
```



```

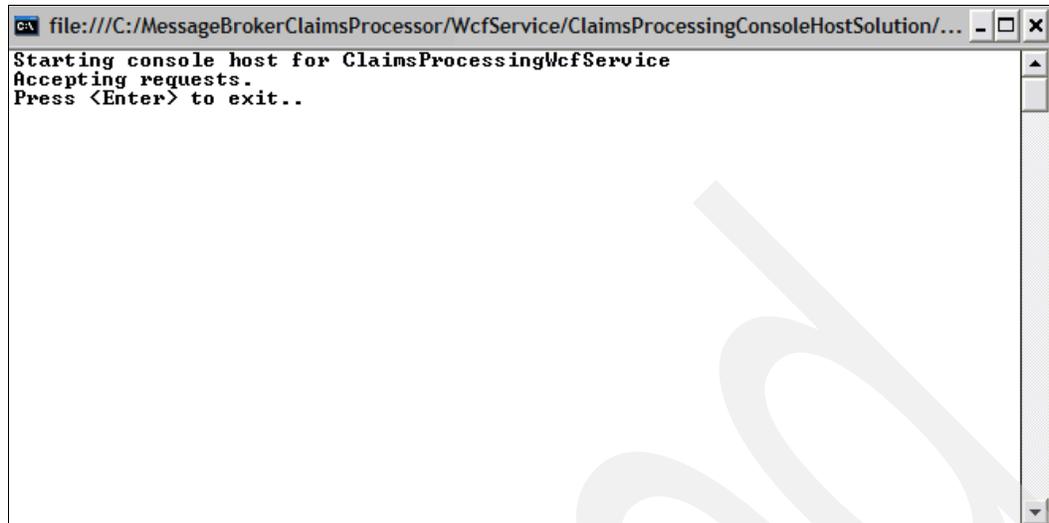
</system.web>
<!-- When deploying the service library project, the content of the config file
must be added to the host's
app.config file. System.Configuration does not support config files for libraries.
-->
<system.serviceModel>
  <bindings/>
  <services>
    <service
name="Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService">
      <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
      <endpoint address="net.tcp://localhost:8523" binding="netTcpBinding"
bindingConfiguration="" name="TcpIpEndPoint"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService" />
    <host>
      <baseAddresses>
        <add baseAddress="http://localhost:8732/ClaimsProcessingWcfService/" />
      </baseAddresses>
    </host>
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior>
      <!-- To avoid disclosing metadata information,
      set the value below to false and remove the metadata endpoint above
before deployment -->
      <serviceMetadata httpGetEnabled="true"/>
      <!-- To receive exception details in faults for debugging purposes,
      set the value below to true. Set to false before deployment
to avoid disclosing exception information -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>

```

e. Save the App.Config file.

8. Launch the Console Application by selecting **Debug** → **Start without debugging** from the Visual Studio menu.

A new console window opens, as shown in Figure 6-37 on page 256.



```
file:///C:/MessageBrokerClaimsProcessor/WcfService/ClaimsProcessingConsoleHostSolution/...
Starting console host for ClaimsProcessingWcfService
Accepting requests.
Press <Enter> to exit..
```

Figure 6-37 The console host running

After the console-based host starts, it will continue to run until you click the Enter key. You can verify that the service is working by using a browser to navigate the base endpoint address `http://localhost:8732/ClaimsProcessingWcfService/`. If the service is running, you can see the service description page, as shown in Figure 6-38.



ClaimsProcessingWcfService Service

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using the `svcutil.exe` tool from the command line with the following syntax:

```
svcutil.exe http://localhost:8732/ClaimsProcessingWcfService/?wsdl
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your client application and use the generated client class to call the Service. For example:

C#

```
class Test
{
    static void Main()
    {
        ClaimsProcessingWcfServiceClient client = new ClaimsProcessingWcfServiceClient();

        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

Figure 6-38 The service description page

If you implemented the WebSphere Message Broker flow that consumes the service, you can use the provided Message Broker Test Client to invoke operations on the WCF Service.

When you want to terminate the service, click Enter to close the console window.

Hosting the service in a stand-alone application is ideal for testing in an automated manner or deploying within a development team to prototype and refine the service. However when

deploying to a production environment, there are options that give a greater level of resiliency and are easier to administer in an enterprise environment.

6.6.3 Hosting as a service

You can host your WCF Service as a Windows service so that it can be managed in the same way that you manage your other Windows services. In this case the Windows Service will create an instance of the WCF Service that exists for the lifetime of the Windows Service.

Creating the project

To create a Solution and Project to host your application:

1. Create a new project:
 - a. Select **File** → **New** → **Project** from the Visual Studio menu.
 - b. From the Installed Templates, select **Visual C#** → **Windows**.
 - c. From the list in the center pane, select **Windows Service**.
 - d. In the Name field, enter `ClaimsProcessingWindowsServiceHost`.
 - e. In the Location field, enter a location on the file system to store your project.
 - f. In the Solution name field, enter `ClaimsProcessingWindowsServiceHostSolution`.
 - g. Ensure that the Create Directory for solution box is selected.
 - h. Click **OK**.

The completed New Project dialog now looks like Figure 6-39.

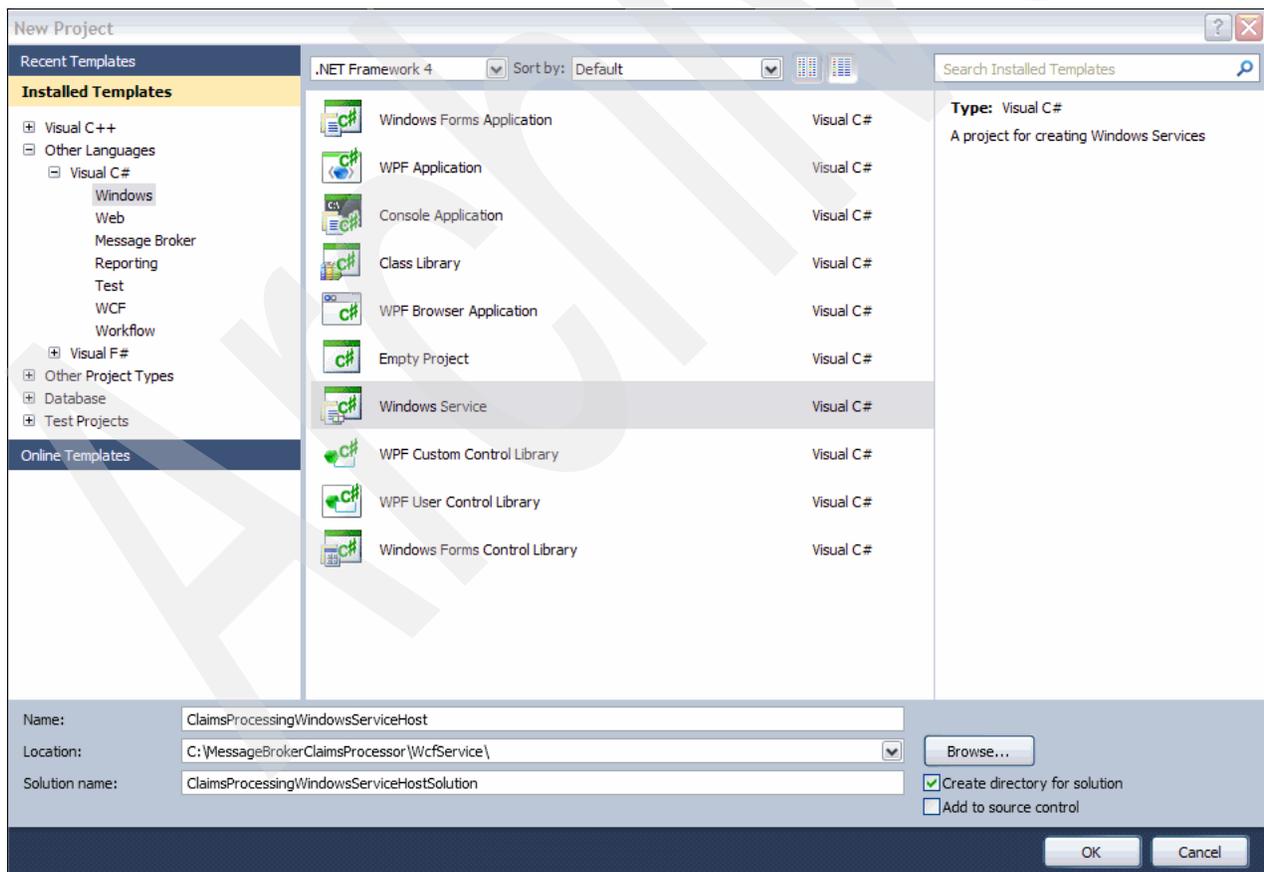


Figure 6-39 Creating a New Windows Service

2. The New Project Wizard creates a Service1.cs file and a Program.cs file. Follow these steps to rename these classes to more meaningful names:
 - a. Select the **Service1.cs** file in the Solution Explorer. Right-click and select **Rename**. Enter ClaimsProcessingHostService.cs as the name, and click Enter. A dialog asks you if you want to rename all project references to Service1.cs. Select **Yes**.
 - b. Select **Program.cs** from the Solution Explorer, right-click, and select **Rename**. Enter ClaimsProcessingHostServiceMain.cs, and click Enter. A message appears that asks if you want to rename all project references to Program. Select **Yes**.
3. To reference the ClaimsProcessingWcfService, add a reference to the Assembly. To do this, follow these steps:
 - a. Right-click the References folder in the Solution Explorer, and select **Add Reference**.
 - b. Click the **Browse** panel.
 - c. Navigate to the output folder where the ClaimsProcessingWcfService.dll assembly is located, and select it.
 - d. Click **OK**.
4. You also need to reference the System.ServiceModel assembly to gain access to the WCF APIs used and the System.Configuration.Install assembly to gain access to the API that allows you to create an installer for this Windows Service.

To add a reference to the System.ServiceModel.dll assembly:

 - a. Right-click the References folder in the Solution Explorer, and select **Add Reference**.
 - b. In the .NET panel, select **System.ServiceModel**.
 - c. Click **OK**.
5. To add a reference to the System.Configuration.Install assembly:
 - a. Right-click the References folder in the Solution Explorer, and select **Add Reference**.
 - b. In the .NET panel, select **System.Configuration.Install**.
 - c. Click **OK**.
6. To add the code to implement the Windows Service:
 - a. Open the ClaimsProcessingHostService.cs file by double-clicking it.
 - b. Right-click in the service Design pane, and select **Properties**. In the ServiceName field, enter ClaimsProcessingService as the value, and close the properties view.
 - c. In the service Design pane dialog, select **click here to switch to code view**, as shown in Figure 6-40.

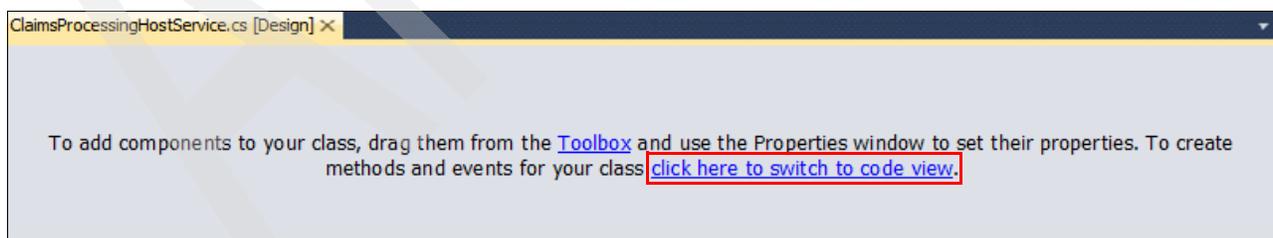


Figure 6-40 Switch to code view

- d. Modify the code for the ClaimsProcessingHostService.cs so that it matches the code shown in Example 6-74.

Example 6-74 ClaimsProcessingHostService.cs

```
using System;
```

```

using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.ServiceProcess;
using System.Text;
using System.ServiceModel;
using System.Configuration;
using IBM.Broker.Example.ClaimsProcessing;

namespace ClaimsProcessingWindowsServiceHost
{
    public partial class ClaimsProcessingHostService : ServiceBase
    {
        public ServiceHost claimsProcessingHost = null;
        public ClaimsProcessingHostService()
        {
            InitializeComponent();
        }
        protected override void OnStart(string[] args)
        {
            claimsProcessingHost = new
ServiceHost(typeof(ClaimsProcessingWcfService));
            claimsProcessingHost.Open();
        }
        protected override void OnStop()
        {
            if (claimsProcessingHost != null)
            {
                claimsProcessingHost.Close();
            }
        }
    }
}

```

7. Double-click the ClaimsProcessingHostService.cs file to open the design.
8. Right-click in the design pane, and select **Add Installer**.
9. Double-click the new ProjectInstaller.cs file to open the ProjectInstaller in the design view.
10. Select **serviceProcessInstaller1**. Right-click, and select **Properties**. Click the **Categorized** button in the Properties view ().
11. Set the Account value to LocalService, as shown in Figure 6-41, and close the Properties view.



Figure 6-41 Setting the service to run under the LocalService account

12. Save and close the ProjectInstaller.cs file.
13. Add an App.Config file to the project so that the ClaimsProcessingWcfService, created by the Windows Service host, knows what end points are required.

Add the App.Config file by following these steps:

- a. Select **Project** → **Add New Item** from the Visual Studio menu.
- b. In the Add New Item dialog, select **Application Configuration File**, and leave the default name of App.config set, as shown in Figure 6-36 on page 254.

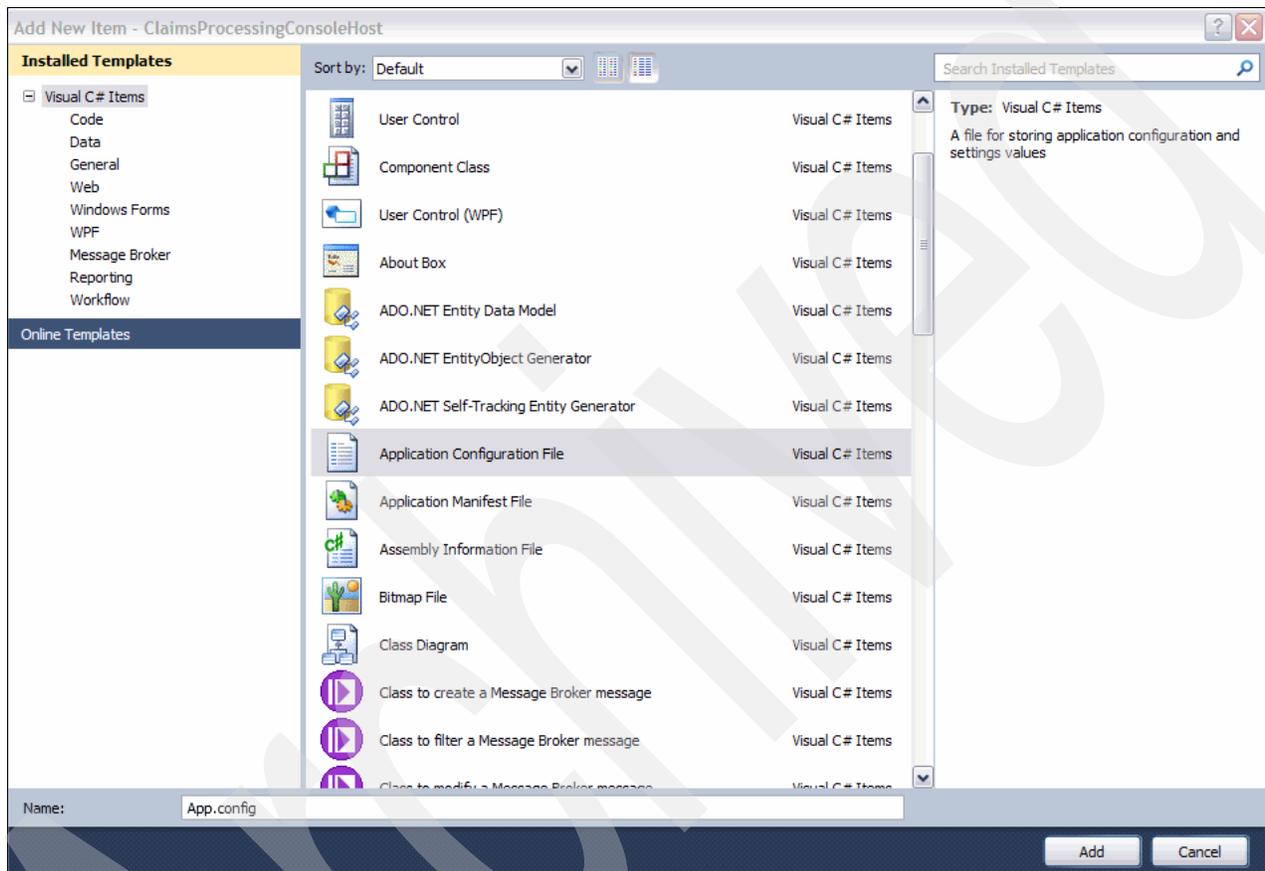


Figure 6-42 Adding the App.config file

- c. Click **Add**.
- d. Copy the configuration information from the ClaimsProcessingWcfServiceProject's App.Config file into the newly created App.Config file. The completed App.Config file now looks like Example 6-73 on page 254.

Example 6-75 An App.Config file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- When deploying the service library project, the content of the config file
  must be added to the host's
  app.config file. System.Configuration does not support config files for libraries.
  -->
  <system.serviceModel>
```

```

    <bindings/>
    <services>
      <service
name="Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService">
        <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
        <endpoint address="net.tcp://localhost:8523" binding="netTcpBinding"
bindingConfiguration="" name="TcpIpEndPoint"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8732/ClaimsProcessingWcfService/" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- To avoid disclosing metadata information,
set the value below to false and remove the metadata endpoint above before
deployment -->
          <serviceMetadata httpGetEnabled="true"/>
          <!-- To receive exception details in faults for debugging purposes,
set the value below to true. Set to false before deployment
to avoid disclosing exception information -->
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

```

e. Save the App.Config file.

14. Build the Windows Service using the **Build** → **Build Solution** menu item. Select **View** → **Output** to see the results of the build. Note the name of the directory where the executable is stored.

Installing the service

Before you can start the service, it must be installed using the `installutil` command provided with Visual Studio.

To install the service:

1. Open a new Visual Studio command prompt using **Start** → **All Programs** → **Microsoft Visual Studio 2010** → **Visual Studio Tools** → **Visual Studio Command Prompt (2010)**.
2. Navigate to the directory that the generated executable was placed in using the `cd` command.
3. Using the `dir` command you can see an executable generated for your project, as shown in Figure 6-43 on page 262.

```

Visual Studio Command Prompt (2010)
C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug>dir
Volume in drive C has no label.
Volume Serial Number is 6023-676D

Directory of C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug

29/03/2012  16:25    <DIR>          .
29/03/2012  16:25    <DIR>          ..
24/03/2012  11:45                31,232 ClaimsProcessingWcfService.dll
24/03/2012  11:45            134,656 ClaimsProcessingWcfService.pdb
29/03/2012  16:25                7,680 ClaimsProcessingWindowsServiceHost.exe
29/03/2012  16:25                1,645 ClaimsProcessingWindowsServiceHost.exe.config
29/03/2012  16:25            24,064 ClaimsProcessingWindowsServiceHost.pdb
29/03/2012  15:29            11,600 ClaimsProcessingWindowsServiceHost.vshost.exe
17/03/2010  17:39                490 ClaimsProcessingWindowsServiceHost.vshost.exe.manifest

             7 File(s)          211,367 bytes
             2 Dir(s)      4,652,748,800 bytes free

C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug>

```

Figure 6-43 Checking the output file exists

4. Enter the command `installutil ClaimsProcessingWindowsServiceHost.exe`.

When the installer finishes, you will see the output shown in Figure 6-44.

```

Visual Studio Command Prompt (2010)
ProcessingWindowsServiceHost.InstallLog.
Committing assembly 'C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.exe'.
Affected parameters are:
  logtoconsole =
  logfile = C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.InstallLog
  assemblypath = C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.exe

The Commit phase completed successfully.

The transacted install has completed.

C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug>

```

Figure 6-44 The Service is installed successfully

After the service is installed, it can be managed from the Computer Management console in the same way that you manage any other Windows Service.

Setting the access control list

Before the service can start, you must set an access control list to permit the service to listen on the URL listed in the MEX endpoint. This requires a tool called `httpcfg.exe`, which is shipped by default on server versions of Windows and is available separately for Windows XP. Visit the following web site:

<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=18546>

On Windows 7 and Windows Vista, the `httpcfg.exe` tool is superseded by the `netsh` tool.

To configure the HTTP registration permissions:

1. Open a new Command Prompt window using the **Start** → **All Programs** → **Accessories** → **Command Prompt** menu item.

2. Navigate to c:\Program Files\Support Tools.
3. Enter the following command that is appropriate for your system:

```
httpcfg.exe set urlacl /u http://+8732/ClaimsProcessingWcfService/ /a  
D:(A;;GX;;;WD)
```

Or

```
C:\>netsh.exe http add urlacl url=http://+8732/ClaimsProcessingWcfService/  
user=Everyone
```

Complete the command with a completion code of zero.

Note: The ACL definition given in the httpcfg.exe command allows all processes from all users to make URL registrations on the `http://localhost:8732/ClaimsProcessingWcfService` URL space. If your local security guidelines do not allow you to grant global access to this part of the URL space, check with your Windows System Administrator to obtain the correct ACL definition to use in the httpcfg.exe command.

Starting and stopping the service

Start the service by following these steps:

1. Open the Computer Management Console by selecting **Start** → **Run**, and entering `compmgmt.msc`.
2. Navigate to **Services and Applications** → **Services** in the explorer pane.
3. Select **ClaimsProcessingService** from the right panel.
4. Right-click and select **Start**.

You can verify that the service is running by navigating to the base address of the application, `http://localhost:8732/ClaimsProcessingWcfService/`, using a web browser. If the service is running successfully, you will see the service description page, as shown in Figure 6-45 on page 264.



Figure 6-45 The service description page

Note: Windows Services reports errors to the Windows Event Log, so if your service fails to start, you can look up the cause in the Applications section of the Event Viewer. For example, failing to enable the URL registration with httpcfg.exe results in a System.ServiceModel.AddressAccessDeniedException being reported to the Event Log.

The WCF service will continue to run for the lifetime of the Windows service. To stop the Windows service and therefore stop the WCF service follow these steps:

1. Open the Computer Management Console by selecting **Start** → **Run** and entering compmgmt.msc.
2. Navigate to **Services and Applications** → **Services** in the explorer pane.
3. Select **ClaimsProcessingService** from the right panel.
4. Right-click, and select **Stop**.

Uninstalling the service

When you make changes to your Windows Service, uninstall the old service before installing the new changed service. To uninstall the old service:

1. Open a new Visual Studio command prompt using **Start** → **All Programs** → **Microsoft Visual Studio 2010** → **Visual Studio Tools** → **Visual Studio Command Prompt (2010)**.
2. Navigate to the directory that the generated executable was placed in using the cd command.
3. Enter the command `installutil /u ClaimsProcessingWindowsServiceHost.exe`.
You will see output saying that the uninstall completed, as shown in Figure 6-46 on page 265.

```

Visual Studio Command Prompt (2010)
Uninstalling assembly 'C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.exe'.
Affected parameters are:
  logtoconsole =
  logfile = C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.InstallLog
  assemblypath = C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug\ClaimsProcessingWindowsServiceHost.exe
Removing EventLog source ClaimsProcessingService.
Service ClaimsProcessingService is being removed from the system...
Service ClaimsProcessingService was successfully removed from the system.
Attempt to stop service ClaimsProcessingService.

The uninstall has completed.
C:\MessageBrokerClaimsProcessor\WcfService\ClaimsProcessingWindowsServiceHostSolution\ClaimsProcessingWindowsServiceHost\bin\Debug>

```

Figure 6-46 Uninstalling the Windows Service

After the service is uninstalled, it no longer appears in the Computer Management Console, and you will no longer be able to navigate to the service description page.

6.6.4 Hosting in Internet Information Services (IIS)

Perhaps the most common way to host an enterprise WCF application is to use Microsoft's Internet Information Services (IIS) web server. To complete this section, you need to have IIS and ASP.NET installed on your machine.

If you are using IIS V6, you also need the IIS 6 Metabase and IIS 6 Configuration Compatibility features installed.

Visual Studio can automatically deploy a WCF service to the local IIS instance. To deploy the ClaimProcessingWcfService:

1. Open the ClaimsProcessingWcfService solution in Visual Studio.
2. Select **Build** → **Publish ClaimsProcessingWcfService** from the menu.
3. Select the **Browse** button next to the Target Location field, as shown in Figure 6-47.

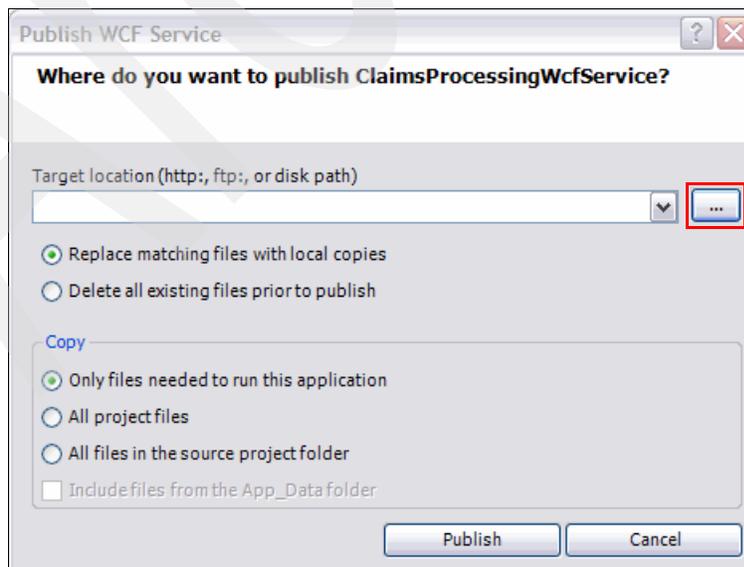


Figure 6-47 PublishWizard1

4. In the Open Web site pane, select **Local IIS**, and then highlight the **Default Web site Node**, as shown in Figure 6-48.

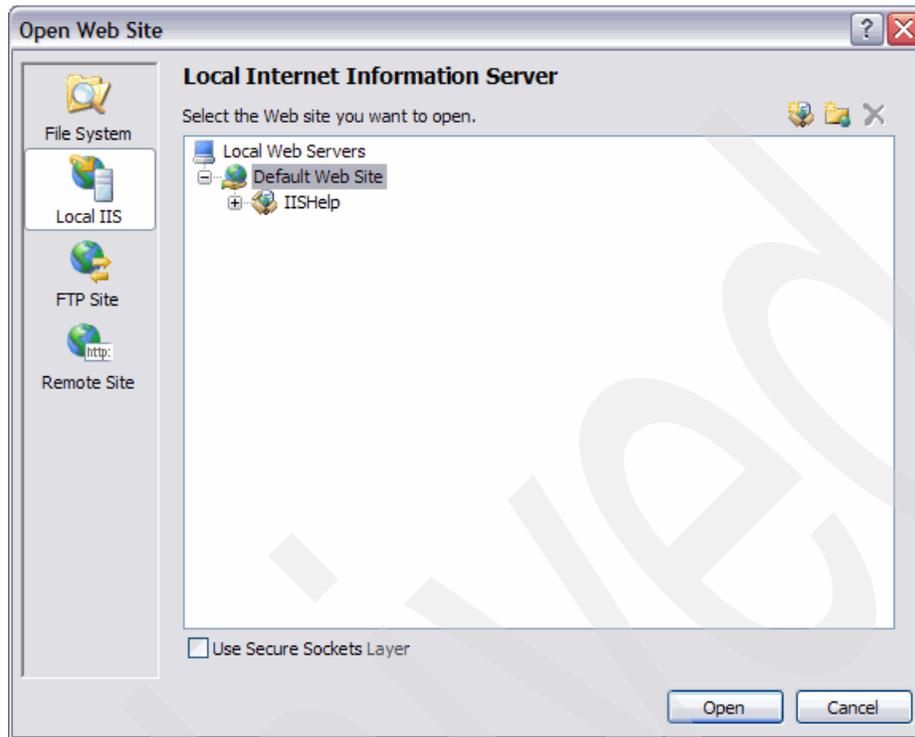


Figure 6-48 PublishWizard2

5. Click the **Create New Web Application** button () in the top-right corner.
6. In the newly created Web Application node, enter the name ClaimsProcessing, as shown in Figure 6-49 on page 267.

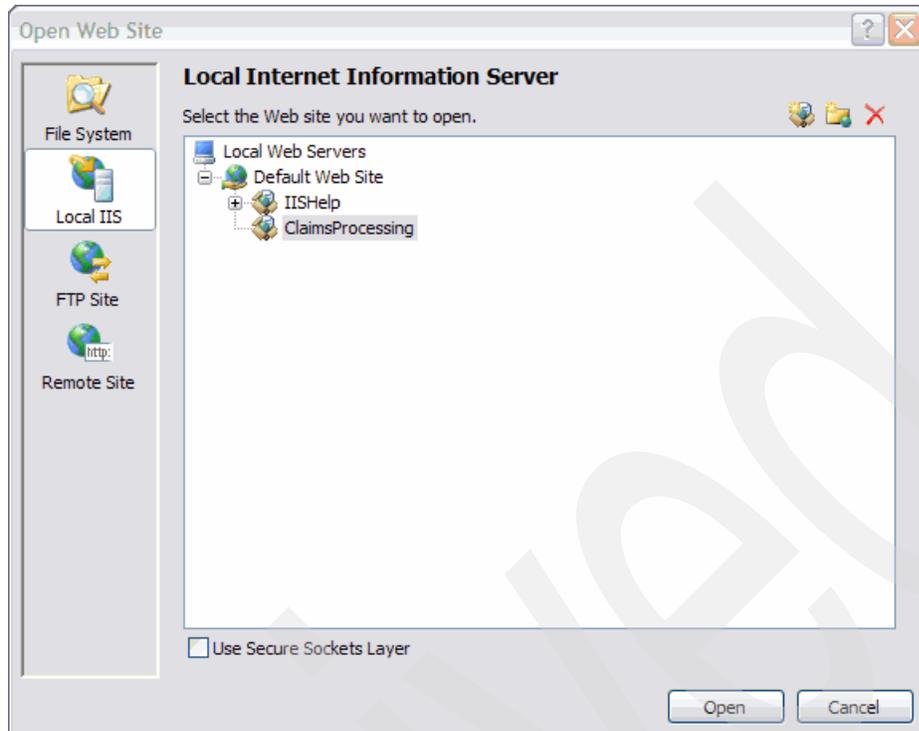


Figure 6-49 Publish Wizard 3

7. Click **Open**.
8. Click **Publish**. The project is rebuilt and deployed to IIS. Successful output is shown in Figure 6-50.

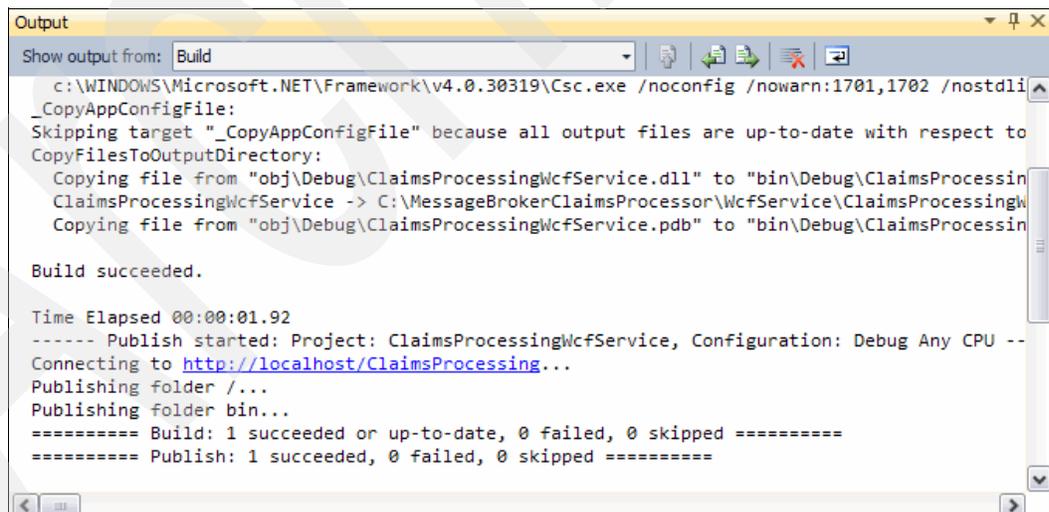


Figure 6-50 Successful publication output

The Publish Wizard created two new files in the ClaimsProcessing Web Application. You can view these in the Computer Management console by following these steps:

1. Open the Computer Management Console by selecting **Start** → **Run**, and entering `compmgmt.msc`.

2. Navigate to **Services and Applications** → **Internet Information Services** → **Web Sites** → **Default Web Site** → **ClaimsProcessing**.

You can see two new files, `Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService.svc` and `Web.Config`, as shown in Figure 6-51.

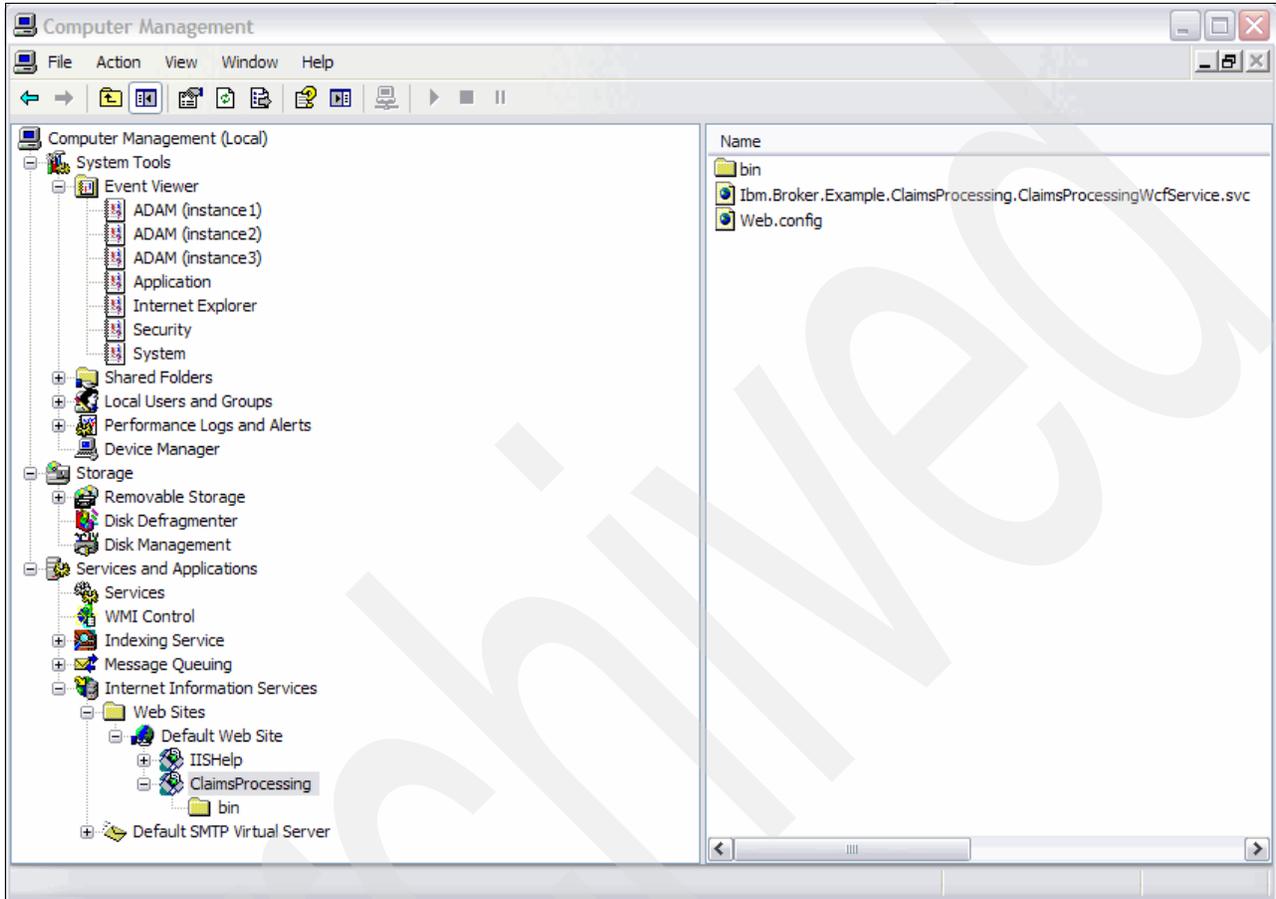


Figure 6-51 Files created by the publish wizard

Example 6-76 shows the contents of the `Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService.svc` file.

Example 6-76 `Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService.svc`

```
<%@ ServiceHost  
Service="Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService" %>
```

This file tells IIS the class that implements the service defined in the `Web.Config` file. Note that the assembly itself is placed in the `bin` directory of the Web Application, as shown in Figure 6-52 on page 269.

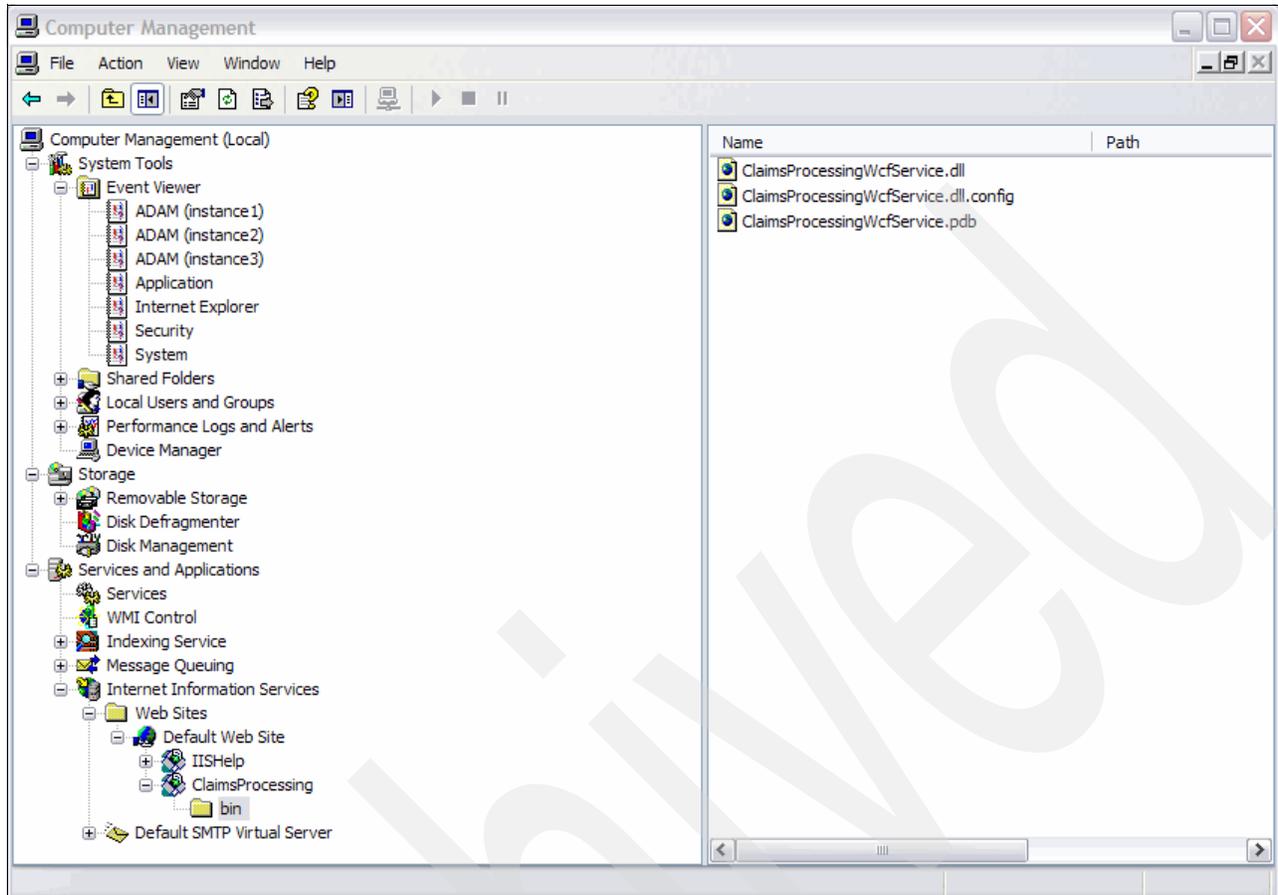


Figure 6-52 The assembly is located in the Web Application's bin directory

Version 7 of IIS supports non-http bindings; however, previous versions do not, so we must modify the Web.Config file to expose an http endpoint.

Open the Web.Config file and modify the contents to match Example 6-77.

Example 6-77 Web.Config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <!-- When deploying the service library project, the content of the config file
  must be added to the host's app.config file. System.Configuration does not support
  config files for libraries. -->
  <system.serviceModel>
    <bindings>
    </bindings>
    <services>
      <service
name="Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService">
        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
        <endpoint address="/WebServices" binding="basicHttpBinding"
bindingConfiguration=""
```

```
name="WebServicesEndpoint"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService" />
  </service>
</services>
<behaviors>
  <serviceBehaviors>
    <behavior>
      <!-- To avoid disclosing metadata information, set the value
below to false and remove the metadata endpoint above before deployment -->
      <serviceMetadata httpGetEnabled="true"/>
      <!-- To receive exception details in faults for debugging purposes,
set the value below to true. Set to false before deployment
to avoid disclosing exception information -->
      <serviceDebug includeExceptionDetailInFaults="False" />
    </behavior>
  </serviceBehaviors>
</behaviors>
</system.serviceModel>
</configuration>
```

The file is essentially the same as the services App.Config file; however, there is no base address for the service. This is because when hosted under IIS the base address from the App.Config file is not used; instead, the address is defined by the location of the .svc file created by the publish wizard.

In this example, the WCF service is hosted at:

```
http://localhost/ClaimsProcessing/Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService.svc
```

You can navigate to this address using a web browser. You can see the service description web page, as shown in Figure 6-53 on page 271.



Figure 6-53 The service description page

If instead of the service description page you see an error stating that the service Failed to access IIS metabase, run the ASP.NET IIS Registration tool, which configures the ASP.Net framework for use in IIS. To perform this step:

1. Open a new Visual Studio command prompt using **Start → All Programs → Microsoft Visual Studio 2010 → Visual Studio Tools → Visual Studio Command Prompt (2010)**.
2. Enter the command `aspnet_regiis -i`.

Now that the service is created, it can be invoked from a message flow. Proceed to Chapter 7, “Scenario: Integrating Windows Communication Foundation in message flows - Part 2” on page 273 to continue this scenario.

Archived

Scenario: Integrating Windows Communication Foundation in message flows - Part 2

In this chapter, we develop, deploy, and test a WebSphere Message Broker application that consumes the ClaimsProcessingWcfService developed in Chapter 6, “Scenario: Integration Windows Communication Foundation in message flows - Part 1” on page 163.

The WebSphere Message Broker application uses the .NETCompute node to embed a WCF client for the service using a TCP/IP-based transport. The scenario also highlights the potential of the .NETCompute node by showing examples for transformation and integration with Microsoft Word.

The scenario also showcases some of the new message modelling and mapping technology introduced in WebSphere Message Broker V8 in the form of the DFDL Parser and the new Mapping node.

Additional materials: You can download the WebSphere Message Broker project interchange file and the .NET class code for this scenario from the IBM Redbooks publication web site. See Appendix A, “Additional material” on page 485 for more information.

7.1 Scenario overview

Figure 7-1 shows an overview of the completed scenario.

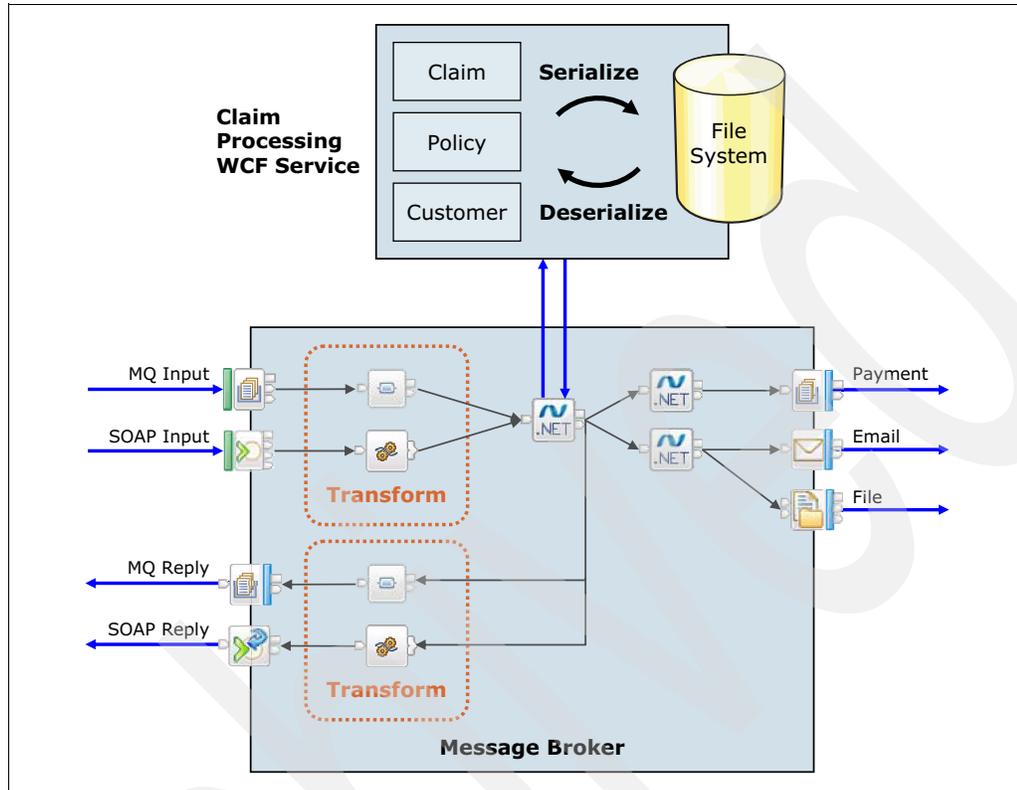


Figure 7-1 New message modeling and mapping technology

The scenario developed in this chapter represents an enterprise application for processing insurance claims. Recall from 6.1, “ClaimsProcessingWcfService overview” on page 164, that the ClaimsProcessingWcf service exposes seven operations for processing insurance claims through the system:

- ▶ ViewOutstandingClaims
- ▶ ViewClaim
- ▶ ApproveClaim
- ▶ RejectClaim
- ▶ CreateClaim
- ▶ CreatePolicy
- ▶ ViewPolicy

These operations are exposed as WCF services; however, in a heterogeneous environment, not all applications can interface directly with WCF. Some applications might be running on operating systems other than Microsoft Windows or might use heritage data exchange formats that cannot be modified when introducing new functionality to the system. WebSphere Message Broker can be used to provide an interface between these heritage applications and the WCF service.

In this scenario, we assume that our fictional insurance company has an existing Enterprise Service Bus (ESB) architecture and that components of this ESB exchange data in a Canonical Message Format. The ESB is implemented with WebSphere Message Broker.

The purpose of the message flow in this scenario is to provide an interface for use by existing applications that communicate using MQ messages in a text/binary message format and by SOAP-based web service clients. The MQ interface represents multiple internal applications used by employees at the insurance company to perform business functions, such as agents that must view claims and then approve or reject them. The SOAP interface represents a web service that can be exposed to customers, allowing them to check on the status of their claim.

In addition to providing an interface for the WCF Service, the message flow also provides the necessary routing and connectors for additional business logic associated with some of the operations in the WCF service.

To provide a robust and resilient solution, the scenario provides an error handling strategy so that the root cause of failures in processing can be communicated back to the originator of each request for remedial action.

The general execution flow for an operation is:

1. An input message is received on one of the input nodes for the flow.
2. The input message is transformed into a Canonical Message Format that is suitable for transmission across other ESB services.
3. A WCF request is made to the ClaimsProcessingWcfService using TCP/IP. The WCF service executes the operation.
4. Any additional processing required for this operation is performed.
5. A reply message is sent to the originator of the request with the results from the operation.

The CreatePolicy and ApproveClaim operations both require additional processing:

CreatePolicy When a new policy is created, a word document containing a policy confirmation letter is generated. This letter is emailed to the customer's email address and is also written to the local file system so that it can be printed and mailed to the customer's physical address.

ApproveClaim When an insurance claim is approved, a payment is issued by another interface on the ESB (message flow) that deals with sending the appropriate data to the financial partner that underwrites the particular policy that the customer bought. This interface is represented by an MQ queue and is assumed to exchange data in the Canonical Message Format.

7.1.1 Prerequisites

This scenario assumes that you have the following prerequisites:

- ▶ WebSphere Message Broker version 8.0.0.0 or higher installed.
- ▶ WebSphere Message Broker Toolkit version 8.0.0.0 or higher installed.
- ▶ A message broker defined on the local machine with a corresponding WebSphere MQ queue manager.
- ▶ Four queues are defined on the queue manager for the broker: IN, OUT, PAYMENT_QUEUE, and ERROR_QUEUE.
- ▶ An SMTP email server running on the local host or access to an SMTP server for sending outbound email.
- ▶ The ClaimsProcessingWcfService described in Chapter 6, "Scenario: Integration Windows Communication Foundation in message flows - Part 1" on page 163 is running on the local host.

- ▶ Visual Studio 2010 is installed.
- ▶ Open XML SDK V2 installed. Download Microsoft Open XML SDK V2 at: <http://www.microsoft.com/download/en/details.aspx?id=5124>
- ▶ Microsoft Word 2010 installed.

If you do not have a full copy of Microsoft Word, you can download and install the Microsoft Word Viewer and the Microsoft Office Compatibility Pack:

- Download Microsoft Word Viewer at: <http://www.microsoft.com/download/en/details.aspx?id=4>
- Download Microsoft Office Compatibility Pack at: <http://www.microsoft.com/download/en/details.aspx?id=3>

7.2 Creating the message flow

In this section, we create the message flow in the Message Broker Toolkit by dragging nodes from the palette onto the canvas and wiring their connections. This section defines the execution order of the flow and sets any node properties that are used by the flow, as illustrated in Figure 7-2.

The ESQL and C# code required for each node are introduced in the subsequent sections that outline small groups of nodes in the flow by function.

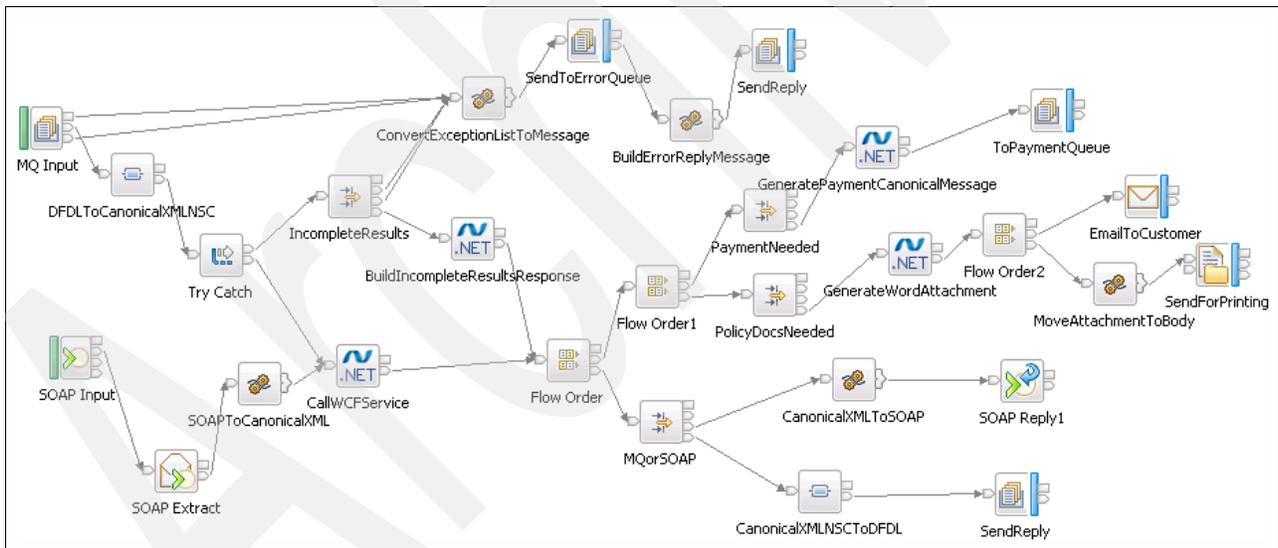


Figure 7-2 Message flow and connection wiring for nodes in WebSphere Message Broker

7.2.1 Preparing message models

This scenario uses several predefined message models. The creation of these models is beyond the scope of this book, so they are provided as WebSphere Message Broker libraries that can be referenced from the project you use to develop your message flow.

The message model libraries provided have the following functions:

ClaimsProcessingCanonicalMessageModel

An XML schema for the XMLNSC parser that models a Canonical message structure that is used for data exchange between services on the ESB. The Canonical format makes it easier for services to be composed into larger enterprise applications because it provides a common data integration point.

ClaimsProcessingDFDLMessageFormats

The MQ applications that use the service send data based on a combination text/binary format that used sequences of Initiators, Delimiters, and Terminators to define where parts of the data start and end. This project models the data for these applications using the Data Format Description Language (DFDL).

ClaimsProcessingSOAPImportedLibraries

The SOAP interface for the flow was built by dragging and dropping a WSDL onto the SOAPInput node. For this to work, the WSDL for this service must be imported into the Message Broker Toolkit where changes are made to ensure that the structure of referenced schemas, such as the SOAP envelope, are in the correct locations. This project already had this import step performed.

Importing message models

Additional materials: The message model libraries used in this section are in the additional materials as `WCFCClient\Project Interchange files\MessageModelPI.zip`.

In this section, the steps for importing Claims Processing Canonical Message Model, Claims Processing DFDL Message Formats, and Claims Processing SOAP Imported Libraries are covered in detail.

Use the following procedure to import message models:

1. Open the Message Broker Toolkit, and enter a new workspace name. You are now in the Broker Application Development perspective with an empty workspace that is ready for new projects to be added and created.
2. Right-click in the **Application Development** pane, and select **Import**. Select **Other** → **Product Interchange**, and click **Next**.
3. Click **Browse**, and locate the project interchange file, and then select the three message model projects, as shown in Figure 7-3 on page 278.

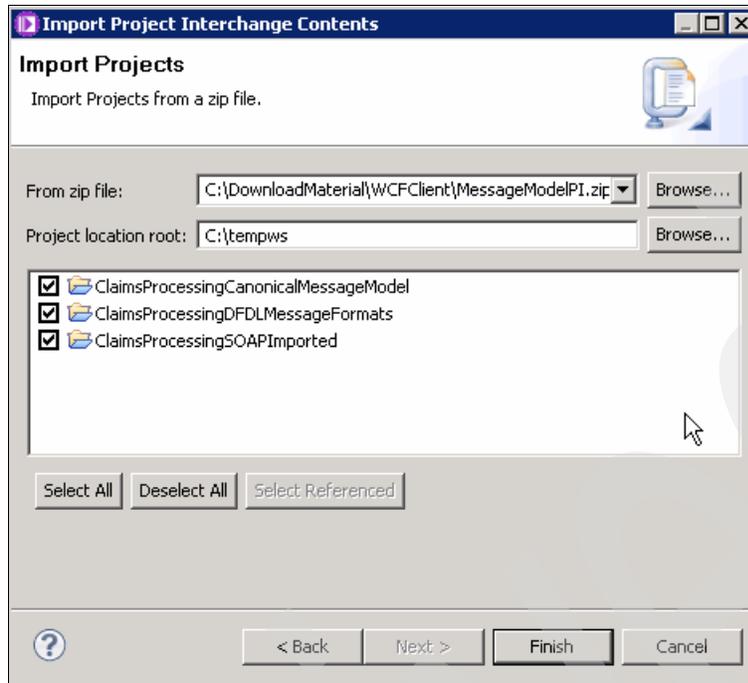


Figure 7-3 Browse for a root directory

4. Click **Finish**.

7.2.2 Creating the message flow

In this section, we create the message flows, create a new application, and import mapping models for our scenario.

Creating a new application called ClaimProcessing

Use the following procedure to quickly outline how to create a new application. These easy steps create efficiency, accuracy, and help standardize the process:

1. Select **File** → **New** → **Application**
2. Name the application `ClaimProcessing` as shown in Figure 7-4. Click **Finish**.

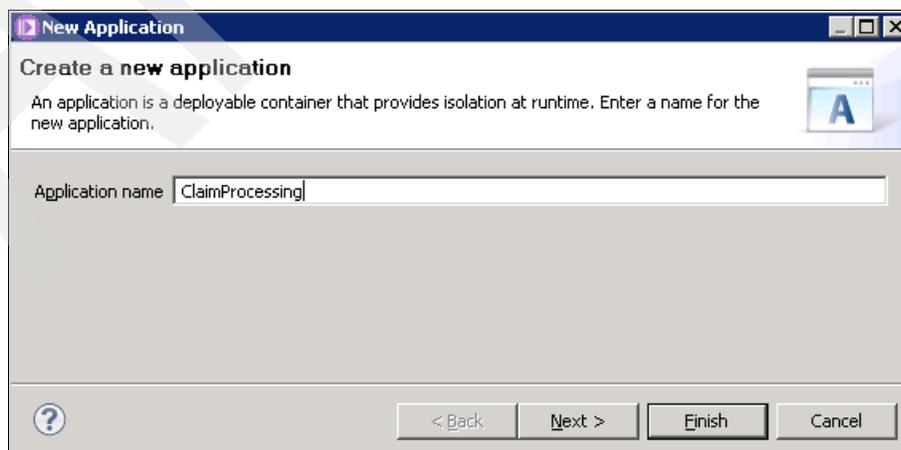


Figure 7-4 The new application panel in WebSphere Message Broker

Adding references to the libraries

The next step is to add references to the libraries:

1. Right-click the **ClaimProcessing** application in the **Application Development** pane, and select **Properties**.
2. Select **Project References** and then select **ClaimsProcessingDFDLMessageFormats**, **ClaimsProcessingCanonicalMessageModel**, and **ClaimsProcessingSOAPImported**, as shown in Figure 7-5. Click **OK**.

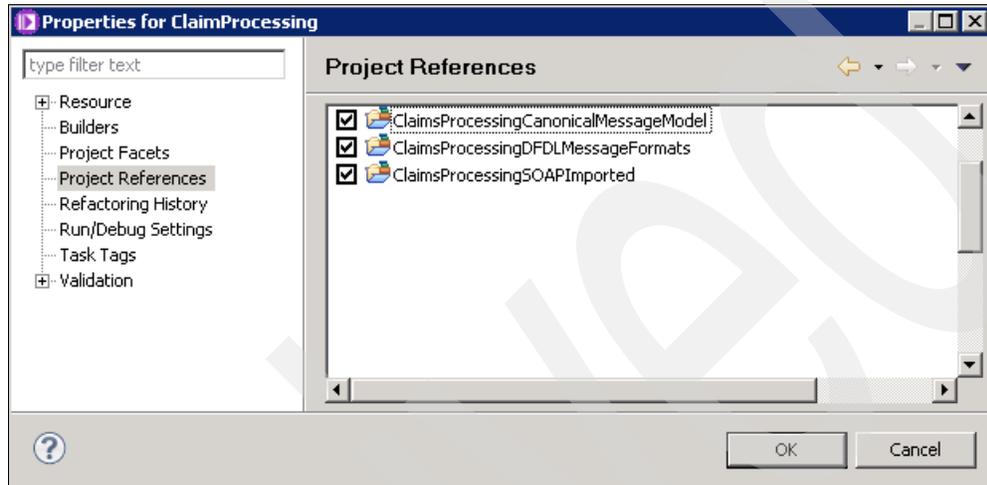


Figure 7-5 Assigning project references in WebSphere Message Broker

Importing mapping modules

After adding the references to the library, the next step is to import the mapping modules. Use the following procedure to import the modules `CanonicalXMLNSCToDFDL.map` and `DFDLToCanonicalXMLNSCMap.map`.

Additional materials: The maps are in the additional materials for this book in the `WCFCClient\Maps` folder.

1. Right-click **ClaimProcessing** and then select **Import**.
2. Select **General** → **File System**, and click **Next**.
3. Click **Browse** to locate and select the folder where the maps are located. Click **OK**.
4. Select `CanonicalXMLNSCToDFDL.map` and `DFDLToCanonicalXMLNSCMap.map`, as shown in Figure 7-6 on page 280. Click **Finish**.

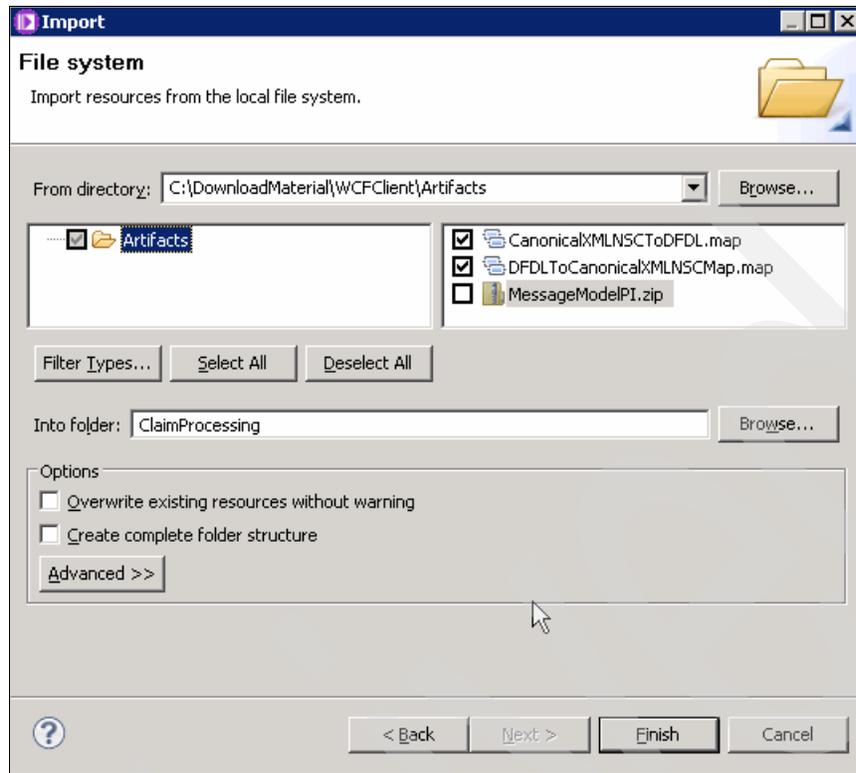


Figure 7-6 Importing the .map files

Note: If you see errors on the imported maps, make sure that you added the references to the libraries. Then, rebuild the workspace using the following steps.

1. Go to **Project** → **Clean**.
2. Select **Clean all projects** and click **OK**.

Creating a new message flow

Use the following procedure to create a message flow for the ClaimProcessing application:

1. Click **File** → **New** → **Message Flow**.
2. Enter the name `ClaimProcessFlow`, and click **Finish**.

7.2.3 Creating and connecting the nodes

The workspace is now ready for the message flow to be built. This section takes you through the process of adding and configuring the nodes in the flow.

Creating the SOAP Input node using a WSDL file

To create a SOAP input node:

1. Select the **ClaimsProcessingSOAP.wsdl** file from the `ClaimsProcessingSOAPImported` library, and then drag-and-drop it on to the canvas, as shown in Figure 7-7 on page 281.

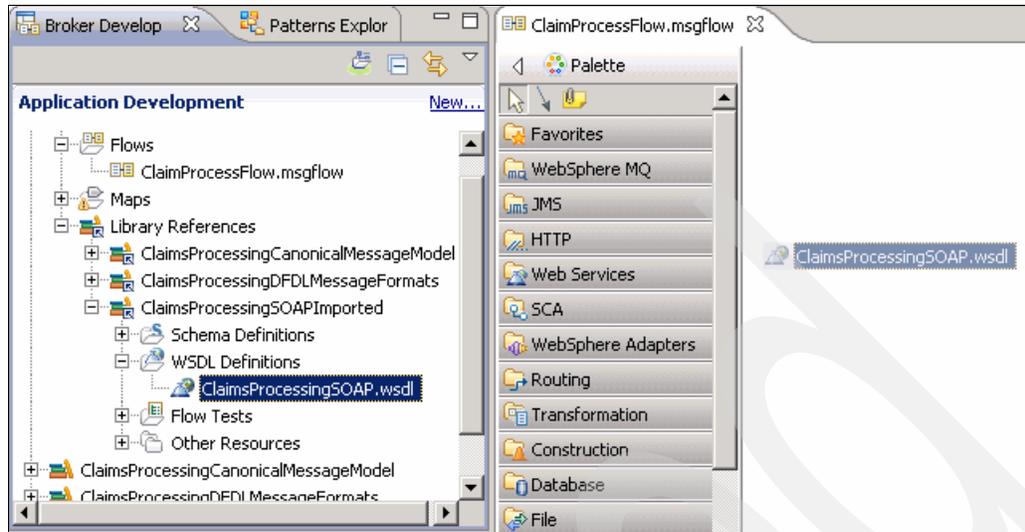


Figure 7-7 Dragging and dropping a file on to the canvas to create a SOAP input node

2. Accept the default configuration on the Configure New Web Service Usage dialog, and click **Finish**, as shown in Figure 7-8.

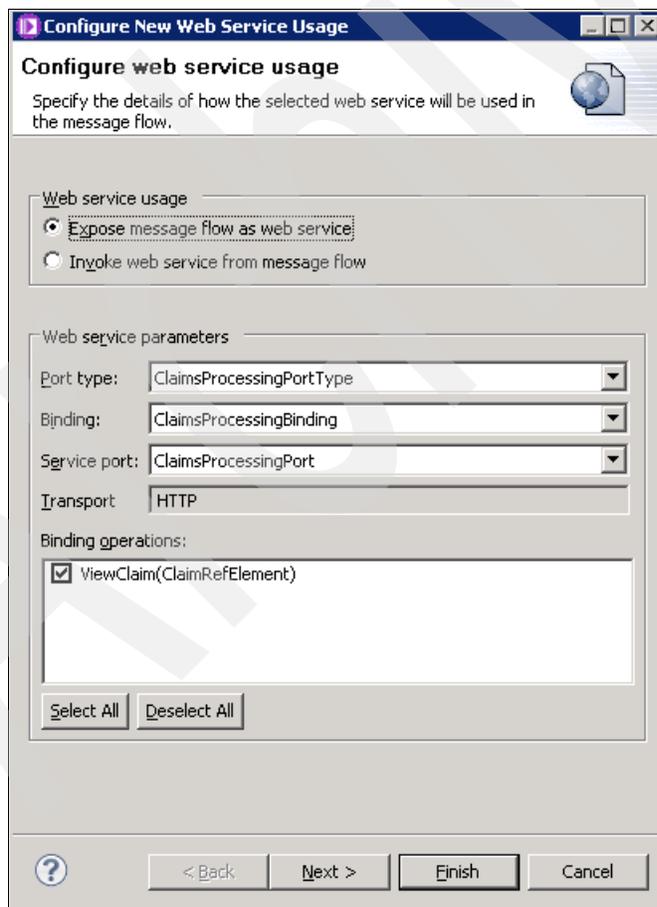


Figure 7-8 Configure a new web service

3. Right-click the subflow **ClaimsProcessingSOAP** node, and select **Delete**, as shown in Figure 7-9.

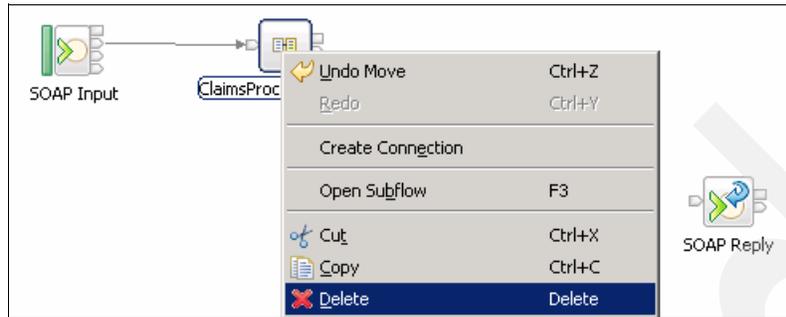


Figure 7-9 Delete a subflow

Creating the remaining nodes using the palette

To complete our message flow application, we must add, name, and connect (wire) the remaining nodes. In “Creating the SOAP Input node using a WSDL file” on page 280, we used the ClaimsProcessingSOAP.wsdl file to create the SOAP input and SOAP Reply nodes. In this section, we show you how to add the rest of the nodes for our application using nodes provided on the palette.

Table 7-1 provides the location, the icon, and type of the node, what you need to rename the nodes after you drag-and-drop them on to the canvas from the palette, and how you need to wire them to other nodes.

When building a message flow, keep in mind the following items:

- ▶ Use your cursor to drag-and-drop the nodes from the palette to the canvas.
- ▶ To rename a node, right-click the node name, and select **Rename**.
- ▶ To connect (wire) a node to another node, click the output terminal for the first node, and then click the input terminal on the next node in the flow.

As you build the message flow, use Figure 7-2 on page 276 as a reference for what you want the completed message flow to look like.

Table 7-1 Node naming and wiring information

Palette drawer / node type	Node name	Wiring information (terminal: to input terminal on node)
Web Services SOAP Input	SOAP Input	▶ Out: SOAP Extract
Web Services SOAP Extract	SOAP Extract	▶ Out: SOAPToCanonicalXML
Transformation Compute	SOAPToCanonicalXML	▶ Out: CallWCFService
Transformation .NETCompute	CallWCFService	▶ Out: Flow Order
WebSphere MQ MQInput	MQ Input	<ul style="list-style-type: none"> ▶ Failure: ConvertExceptionListToMessage ▶ Out: DFDLToCanonicalXMLNSC ▶ Catch: ConvertExceptionListToMessage

Palette drawer / node type	Node name	Wiring information (terminal: to input terminal on node)
Transformation Mapping	DFDLToCanonicalXMLNSC	▶ Out: Try Catch
Construction TryCatch	Try Catch	▶ Try: CallWCFService ▶ Catch: IncompleteResults
Routing Filter	IncompleteResults	▶ True: BuildIncompleteResultsResponse ▶ False: ConvertExceptionListToMessage ▶ Unknown: ConvertExceptionListToMessage
Transformation .NETCompute	BuildIncompleteResultsResponse	▶ Out: Flow Order
Transformation Compute	ConvertExceptionListToMessage	▶ Out: SendToErrorQueue
WebSphere MQ MQOutput	SendToErrorQueue	▶ Out: BuildErrorReplyMessage
Transformation Compute	BuildErrorReplyMessage	▶ Out: SendReply
WebSphere MQ MQOutput	SendReply	(none, end of flow)
Construction FlowOrder	Flow Order	▶ First: Flow Order1 ▶ Second: MQorSOAP
Construction FlowOrder	Flow Order1	▶ First: PaymentNeeded ▶ Second: PolicyDocsNeeded
Routing Filter	PaymentNeeded	▶ True: GeneratePaymentCanonicalMessage
Transformation .NETCompute	GeneratePaymentCanonicalMessage	▶ Out: ToPaymentQueue
WebSphere MQ MQOutput	ToPaymentQueue	(none, end of flow)
Routing Filter	PolicyDocsNeeded	▶ True: GenerateWordAttachment
Transformation .NETCompute	GenerateWordAttachment	▶ Out: Flow Order2
Construction FlowOrder	Flow Order2	▶ First: EmailToCustomer ▶ Second: MoveAttachmentToBody
Email EmailOutput	EmailToCustomer	(none, end of flow)
Transformation Compute	MoveAttachmentToBody	▶ Out: SendForPrinting (In)
File FileOutput	SendForPrinting	(none, end of flow)
Routing Filter	MQorSOAP	▶ True: CanonicalXMLNSCToDFDL ▶ False: CanonicalXMLToSOAP

Palette drawer / node type	Node name	Wiring information (terminal: to input terminal on node)
Transformation Compute	CanonicalXMLToSOAP	► Out: SOAP Reply
Web Services SOAPReply	SOAP Reply	(none, end of flow)
Transformation Mapping	CanonicalXMLNSCToDFDL	► Out: SendReply
WebSphere MQ MQOutput	SendReply	(none, end of flow)

7.2.4 Configuring node properties

In this section, we tell you how to configure the properties for the following nodes in the message flow:

- MQ Input
- DFDLToCanonicalXMLNSC
- EmailToCustomer
- SendForPrinting
- CanonicalXMLNSCToDFDL

To define node properties, display the Properties view by right-clicking a node and selecting **Properties**.

MQ Input node

To define properties for the MQ Input node:

1. In the **Input Message Parsing** tab, use the drop-down list on the Message domain field to select **DFDL: For binary or text messages with a Data Format Description Language schema model**, as shown in Figure 7-10.

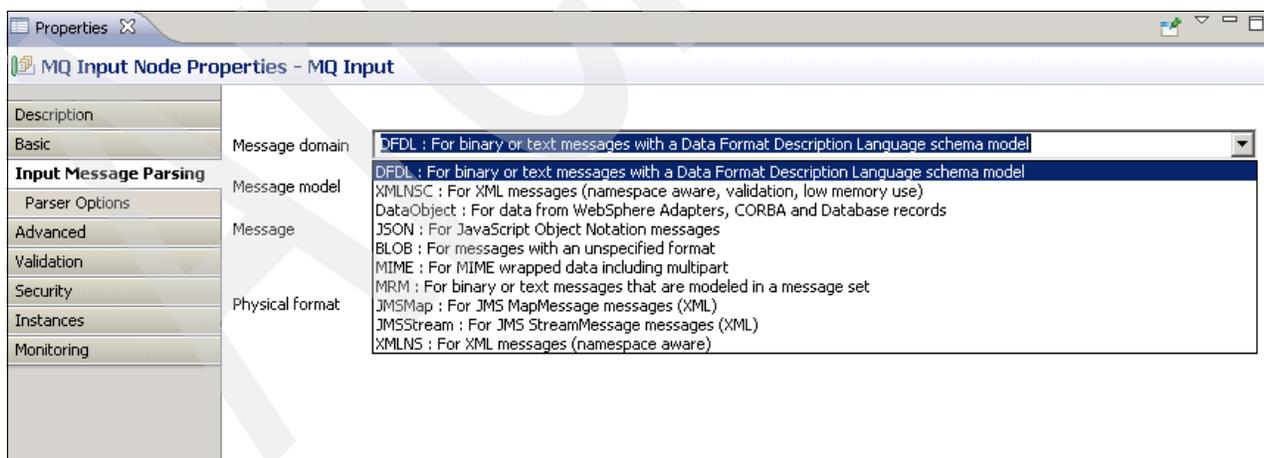


Figure 7-10 Select a message domain

2. Click the **Browse** button next to the Message field, and then select the DFDL schema for **REQUESTMESSAGE**, as shown in Figure 7-11 on page 285. Click **OK**.

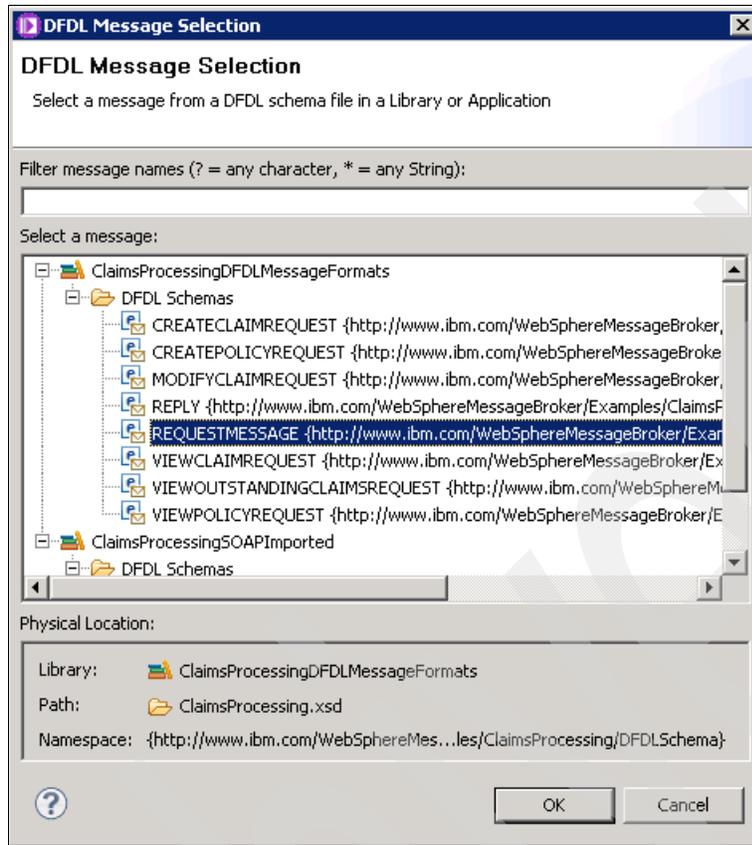


Figure 7-11 Configuring the MQ Input node

3. Press Ctrl+S to save your changes.

DFDLToCanonicalXMLNSC node

Use the following procedure to define properties for the DFDLToCanonicalXMLNSC node:

1. In the **Basic** tab, click the **Browse** button next to the Mapping routine field for the mapping routine property, as shown in Figure 7-12.

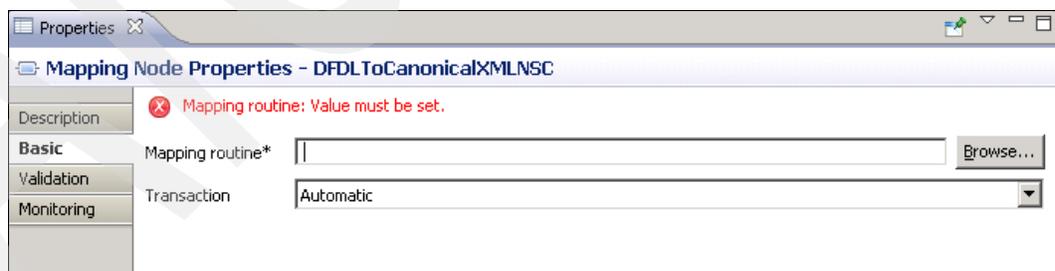


Figure 7-12 Browse for a mapping routine

2. Select **{default}:DFDLToCanonicalXMLNSCMap**, as shown in Figure 7-13 on page 286 and click **OK**.

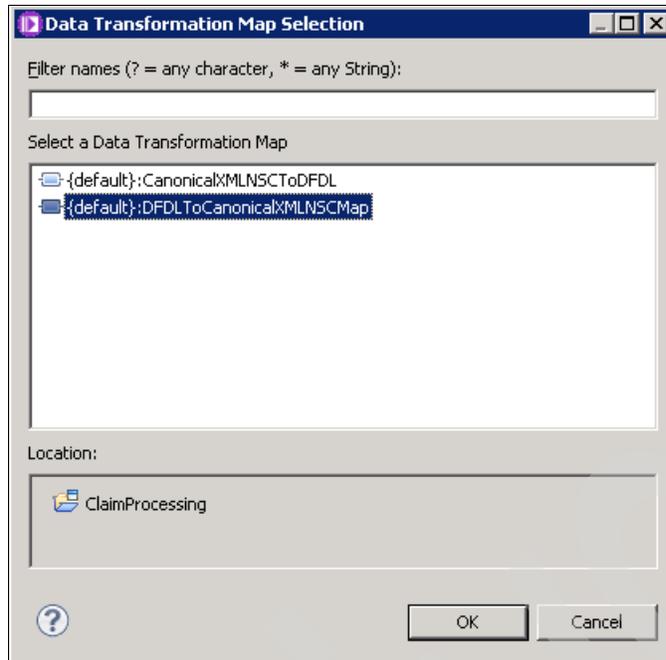


Figure 7-13 Select a mapping routine

3. Press Ctrl+S to save your changes.

EmailToCustomer node

Use the following procedure to define properties for the EmailtoCustomer node:

1. In the Basic tab, enter a location for the SMTP server name and port number. This property defines the SMTP server and port to which emails are sent to from this node. The value for this is in the format server:port (for example: my.smtp.server:25). In our scenario, we use my.smtp.localhost as a basic example, as shown in Figure 7-14. The port value is optional, but if you do not specify a port value, the default value is 25.



Figure 7-14 Specify an SMTP server and port

2. Press Ctrl+S to save your changes.

Note: You can also specify an alias value for this property. If the alias exists at runtime, the specified values are used. If the alias does not exist at runtime, the broker assumes the value to be a valid SMTP host.

You can configure the alias for SMTP server and port number as a broker external resource using WebSphere Message Broker Explorer or `mqsicreateconfigurableservice` command. More information is available at:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/ac66360_.htm

3. Define a user defined property for the node using the following steps:
 - a. Click the **User Defined Properties** tab on the bottom of the message flow editor.
 - b. Right-click **ClaimProcessFlow** → **Basic**, and select **Add Property**.
 - c. Rename the property `FromEmail`.
 - d. In the Default value field, enter the URL of the location you want to specify as the sending location for emails. In our example, we use `ITS0ExampleEmail@itso.example.ibm.com`, as shown in Figure 7-15.

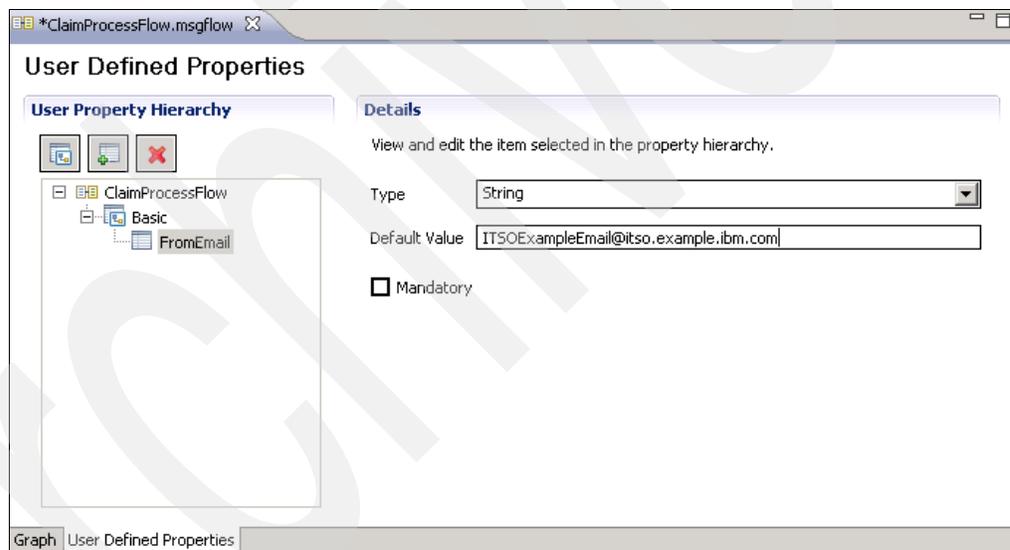


Figure 7-15 Define user defined properties

4. Press **Ctrl+S** to save your changes.
5. Switch back to the Graph view.

User-defined properties are defined at the flow level and do not need to be promoted. When you create a BAR file for deployment, you have the opportunity to configure the property.

SendForPrinting node

Use the following procedure to define properties for the `SendForPrinting` node. For this node, we want to configure the file action property so that it replaces the file, if there is a file that already exists, and promotes the Directory property so that it can be configured in the BAR file:

1. Right-click the `SendForPrinting` node, and select **Promote Property**.

- From the available node properties, select **Basic** → **Directory**, as shown in Figure 7-16. Click **Promote**.

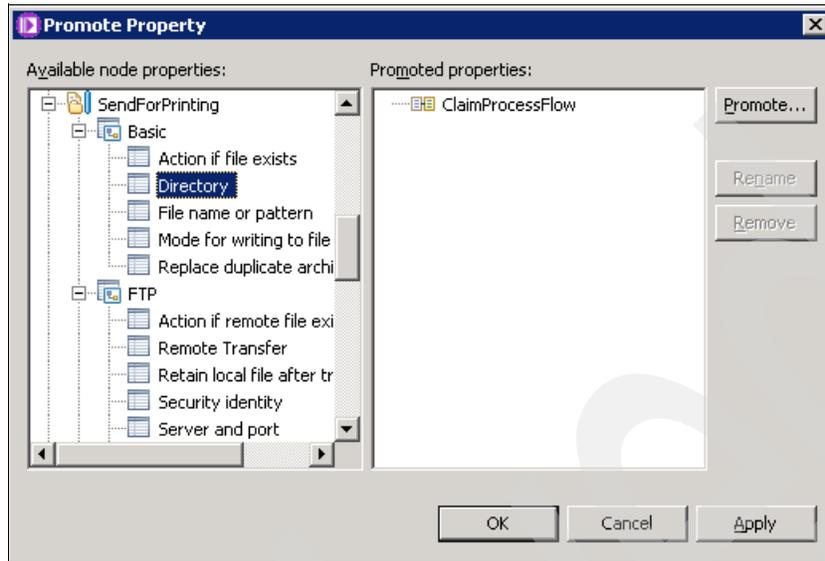


Figure 7-16 Promote a property

- In the Target Selection window, select the default group, **ClaimProcessFlow**, and click **OK**.
- The Directory property will now display in the promoted properties list, as shown in Figure 7-17. Click **OK**.

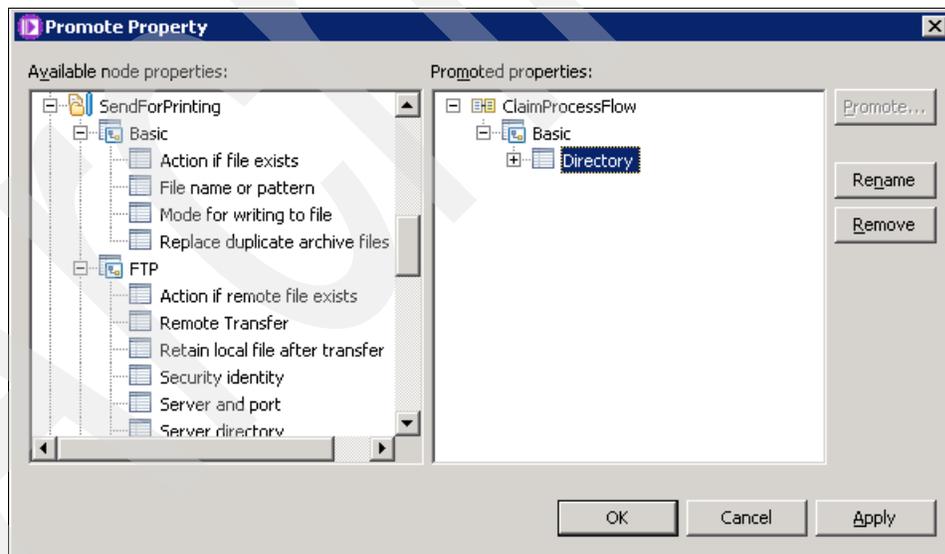


Figure 7-17 Promoting properties in file output nodes

- In the Basic tab, in the Properties view for the node, verify that the Directory property was promoted, as shown in Figure 7-18 on page 289.

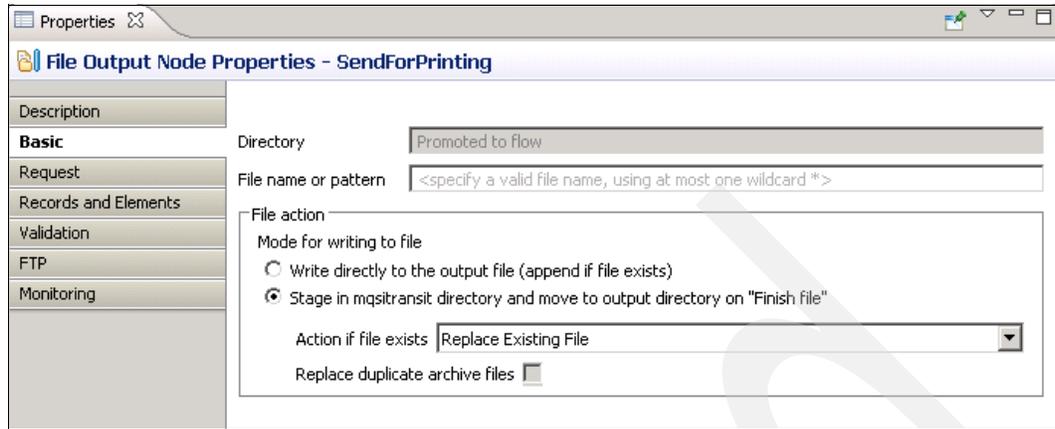


Figure 7-18 Verify the directory property was promoted

6. Select the **Request** tab, specify \$Body for the data location (this is the default), and specify \$LocalEnvironment/Destination/Email/Attachment/ContentName for the request file name property location, as shown in Figure 7-19.

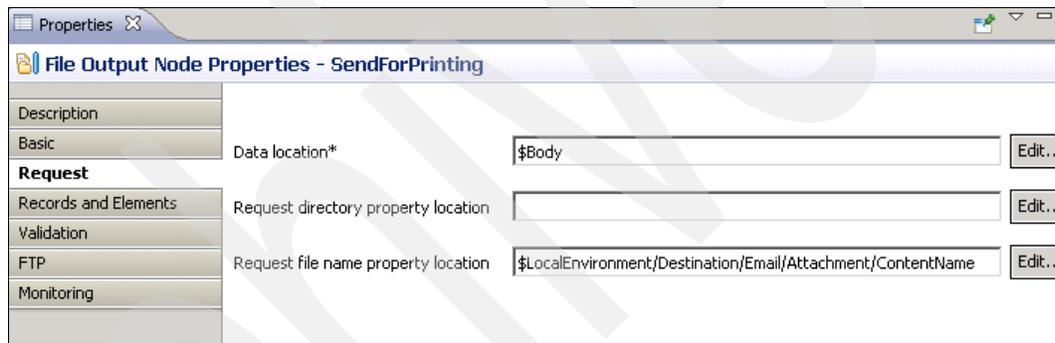


Figure 7-19 Defining location information for the file output node

7. Press Ctrl+S to save your changes.

CanonicalXMLNSCToDFDL node

Use the following procedure to define properties for the CanonicalXMLNSCToDFDL node:

1. In the **Basic** tab, click the **Browse** button for the Mapping routine property, as shown in Figure 7-20.

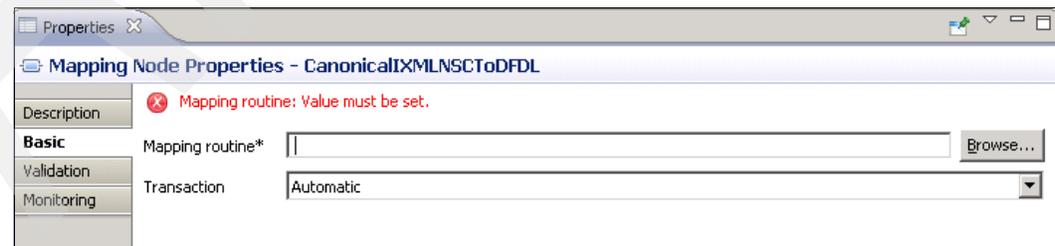


Figure 7-20 Browse for a mapping routine

2. Select **{default}:CanonicalXMLNSCToDFDL**, as shown in Figure 7-21 on page 290. Click **OK**.

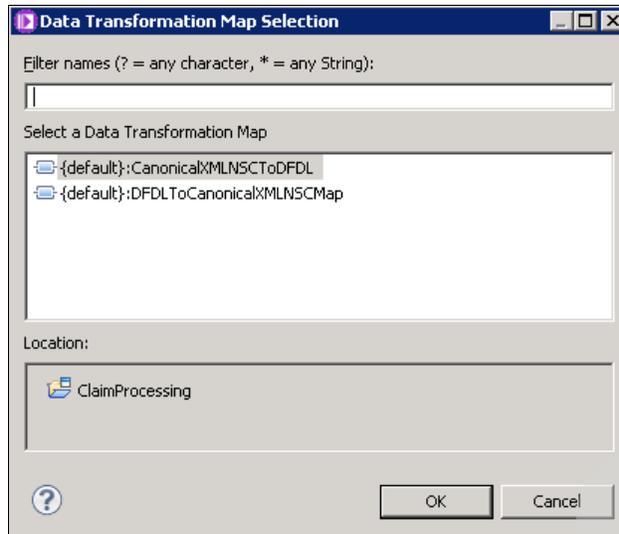


Figure 7-21 Select a transformation map

3. The selected value is added. Press Ctrl+S to save your changes.

7.2.5 Queue names

Some of the nodes in the message flow need to have Queue names configured. Use Table 7-2 to configure the properties for the nodes listed in the table. Accept the default values for all properties unless an alternative value is listed in the table.

Table 7-2 Queue naming for nodes in the flow

Node	Tab	Property	Value
MQ Input	Basic	Queue name	IN
ToPaymentQueue	Basic	Queue name	PAYMENT_QUEUE
SendToErrorQueue	Basic	Queue name	ERROR_QUEUE
SendReply (both nodes)	Advanced	Destination mode	Destination List

7.3 Using the Mapping node to transform the input to the Canonical Message Format

The input from the MQ transport is in a record-oriented text/binary format modeled in *Data Format Description Language* (DFDL). DFDL is an Open Grid standard for modelling text and binary formatted messages using XML. WebSphere Message Broker includes support for the DFDL domain with a new DFDL parser in the broker runtime and a new set of message modelling tools in the Message Broker Toolkit. The DFDL parser supersedes the MRM parsers and is the only supported way to create new message models in version 8 of WebSphere Message Broker. The placement of the DFDL node is highlighted in the message flow shown in Figure 7-22 on page 291.

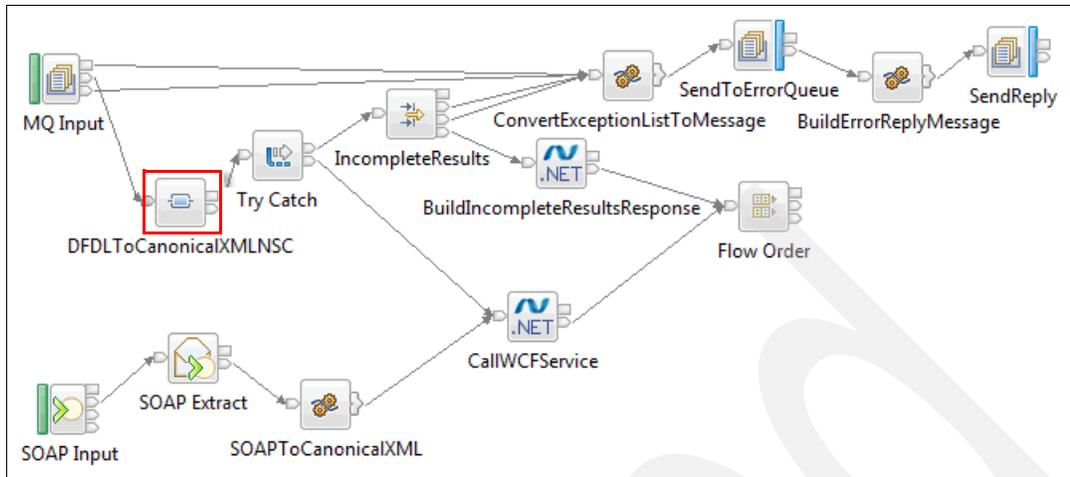


Figure 7-22 DFDL node

Note: The MRM parsers still exist in the broker runtime, so any existing projects or bar files from previous versions of WebSphere Message Broker will still continue to work.

The message model information is held in the ClaimsProcessing.xsd file in the ClaimsProcessingDFDLMessageFormats library, as shown Figure 7-24 on page 292.

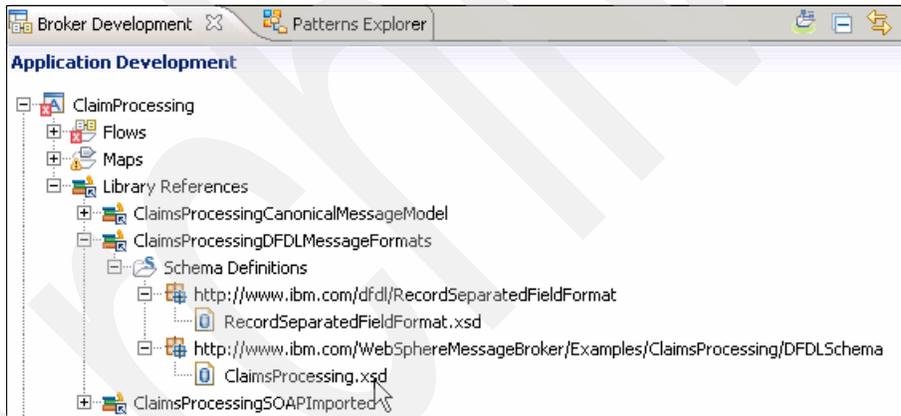


Figure 7-23 ClaimsProcessing.xsd location

Double-click **ClaimsProcessing.xsd** to open it.

In the REQUEST message, one of a number of types based on the operation can be called. Each operation has different children based on the information that needs to be supplied. In Figure 7-23, you can see a CREATECLAIMREQUEST that was expanded to show the detail.

Name	Type	Min Occurs	Max Occurs	Default Value	Sample Value
[-] REQUESTMESSAGE					
[-] choice		1	1		
[-] CREATECLAIMREQUEST		1	1		
[-] sequence		1	1		
[-] CLAIM	CLAIM	1	1		
[-] sequence		1	1		
[-] DATE	dateTime	1	1		2012-04-22T14:00:00
[-] DATERECVD	dateTime	0	1		2001-11-22T12:00:00
[-] ID	string	1	1	<empty string>	CLAIM001
[-] CUSTREF	string	1	1		CUST001
[-] POLICYREF	string	1	1		POLICY001
[-] STATUS	<string>	1	1		Submitted
[-] VALUE	int	1	1		2147483647
[-] DESCRIPTION	string	1	1		This is a claim for an accident that occurred while driving my car
[-] NOTES	string	1	1		The claims agent can add some notes into the claim data
[-] CUSTOMERLIST	CUSTOMER	1	1		
[-] sequence		1	1		
[-] ID	string	1	1		CUST001
[-] NAME	NAME	1	1		
[-] ADDRESS	string	1	1		1 Generic Street, Any town, USA
[-] EMAIL	string	1	1		john.doe@itso.example.ibm.com
[-] CREATEPOLICYREQUEST		1	1		
[-] MODIFYCLAIMREQUEST		1	1		
[-] VIEWCLAIMREQUEST		1	1		
[-] VIEWOUTSTANDINGCLAIMSREQUEST		1	1		
[-] VIEWPOLICYREQUEST		1	1		

Add a Local Element

Figure 7-24 The REQUESTMESSAGE displayed in the DFDL Editor

Covering the building of the DFDL message model is beyond the scope of this chapter. However, you can examine the properties of individual elements to see how they are represented on the wire. For example, Figure 7-25 shows that the CLAIM element has an initiator of CLAIMLIST.

The screenshot shows the DFDL Editor interface. On the left, a tree view shows the 'REQUESTMESSAGE' structure with 'CLAIM' selected. On the right, the 'Representation Properties' pane for the 'CLAIM' element is displayed. The 'Initiator' property is highlighted with a red box and set to 'CLAIMLIST'. Other properties include 'Content' (delimited), 'Occurrences' (Min Occurs: 1, Max Occurs: 1), 'Alignment' (no), 'Terminator' (<no terminator>), and 'Output New Line' (%CR;%LF;).

Figure 7-25 The Representation Properties of the CLAIM element displayed in the DFDL editor

This initiator is highlighted in Figure 7-26 on page 293, which shows a hexadecimal view of a sample CREATECLAIM message.

```

0000: 43 52 45 41 54 45 43 4c 41 49 4d 43 4c 41 49 4d CREATECLAIM CLAIM
0010: 4c 49 53 54 43 4c 41 49 4d 32 30 31 32 2d 30 34 LISTCLAIM2012.04
0020: 2d 32 32 20 30 32 3a 30 30 3a 30 30 2c 32 30 30 .22.02.00.00.200
0030: 31 2d 31 31 2d 32 32 20 31 32 3a 30 30 3a 30 30 1.11.22.12.00.00
0040: 2c 43 4c 41 49 4d 30 30 30 32 2c 43 55 53 54 30 .CLAIM0002.CUSTO
0050: 30 31 2c 50 4f 4c 49 43 59 30 30 31 2c 53 75 62 01.POLICY001.Sub
0060: 6d 69 74 74 65 64 2c 32 31 34 37 34 38 33 36 34 mitted.214748364
0070: 37 2c 44 45 53 43 54 68 69 73 20 69 73 20 61 20 7.DESCThis.is.a.
0080: 63 6c 61 69 6d 20 66 6f 72 20 61 6e 20 61 63 63 claim.for.an.acc
0090: 69 64 65 6e 74 20 74 68 61 74 20 6f 63 63 75 72 ident.that.occu
00a0: 65 64 20 77 68 69 6c 65 20 64 72 69 76 69 6e 67 ed.while.driving
00b0: 20 6d 79 20 63 61 72 0d 0a 0d 0a 2c 4e 4f 54 45 .my.car....NOTE
00c0: 53 54 68 65 20 63 6c 61 69 6d 73 20 61 67 65 6e SThe.claims agen
00d0: 74 20 63 61 6e 20 61 64 64 20 73 6f 6d 65 20 6e t.can.add.some.n
00e0: 6f 74 65 73 20 69 6e 74 6f 20 74 68 65 20 63 6c otes.into.the.cl
00f0: 61 69 6d 20 64 61 74 61 0d 0a 0d 0a 0d 0a 2c 43 aim.data.....C
0100: 55 53 54 4c 49 53 54 43 55 53 54 43 55 53 54 30 USTLISTCUSTCUSTO
0110: 30 31 2c 4e 41 4d 45 4a 6f 68 6e 2c 4a 61 6d 65 01.NAMEJohn.Jame
0120: 73 2a 2c 44 6f 65 0d 0a 2c 41 44 44 52 45 53 53 s..Doe...ADDRESS
0130: 22 31 20 47 65 6e 65 72 69 63 20 53 74 72 65 65 .1.Generic.Stree
0140: 74 2c 20 41 6e 79 20 74 6f 77 6e 2c 20 55 53 41 t..Any.town..USA
0150: 22 2c 6a 6f 68 6e 64 6f 65 40 69 74 73 6f 2e 65 ..johndoe.itso.e
0160: 78 61 6d 70 6c 65 2e 69 62 6d 2e 63 6f 6d 0d 0a xample.ibm.com..
0170: 0d 0a ..

```

Figure 7-26 The CreateClaim Initiator

To transform the input message from the DFDL domain into an XMLNSC CanonicalMessage, the Graphical Data Mapper is used. A map called DFDLToCanonicalXMLNSCMap is already defined for the DFDLToCanonicalXMLNSC node. You can double-click the node to explore the map.

As shown in Figure 7-27 on page 294, the main map has a set of local maps that are executed conditionally depending on the type of input message received. The condition can be any XPath expression. In this example, we use the count() function to test if the CREATECLAIMREQUEST element exists in the tree.

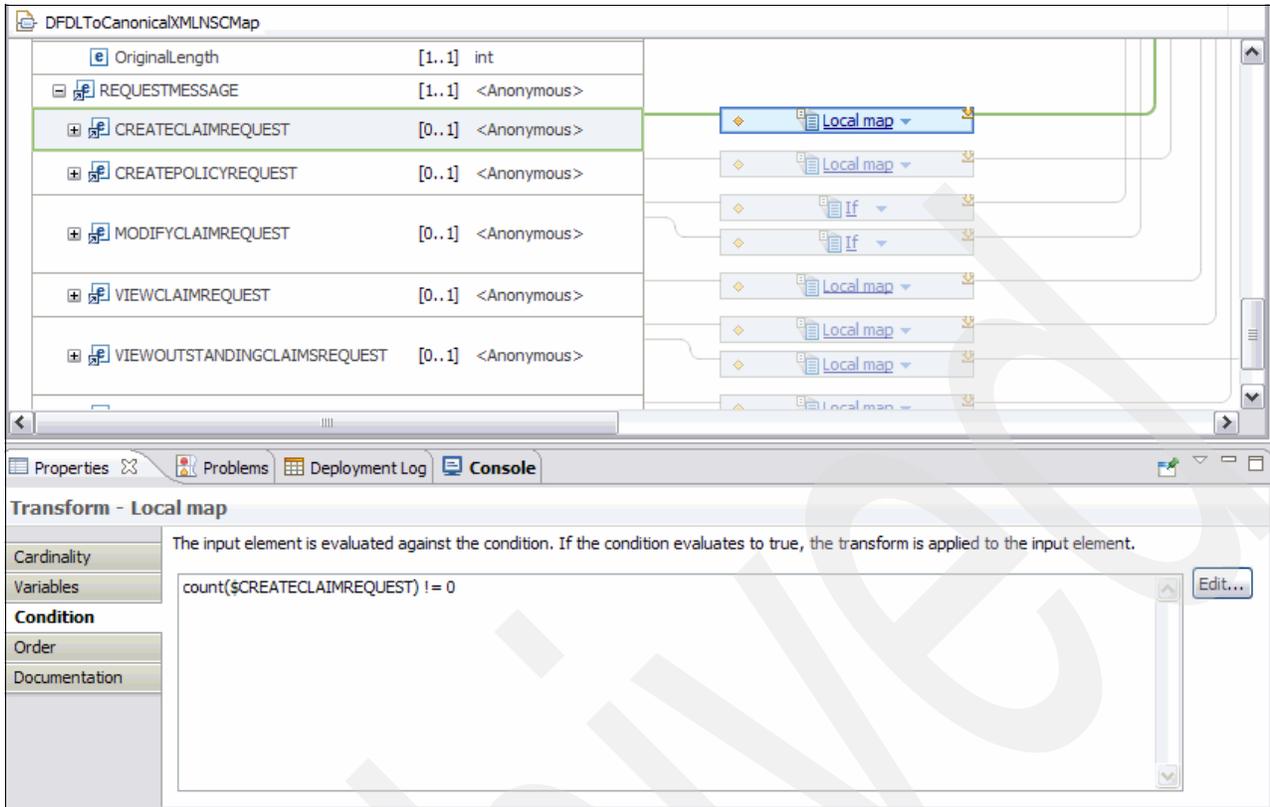


Figure 7-27 The top level map for the DFDLToCanonicalXMLNSC node

The Graphical Data Mapper allows a transformation to be built by dragging connections between data elements. This is ideal for rapid development scenarios since no code needs to be written to achieve the mapping. Figure 7-28 shows the local map that maps the CreatePolicy operation. This map uses a mixture of Move, Assign, and For-each constructs to convert the DFDL message into XMLNSC.

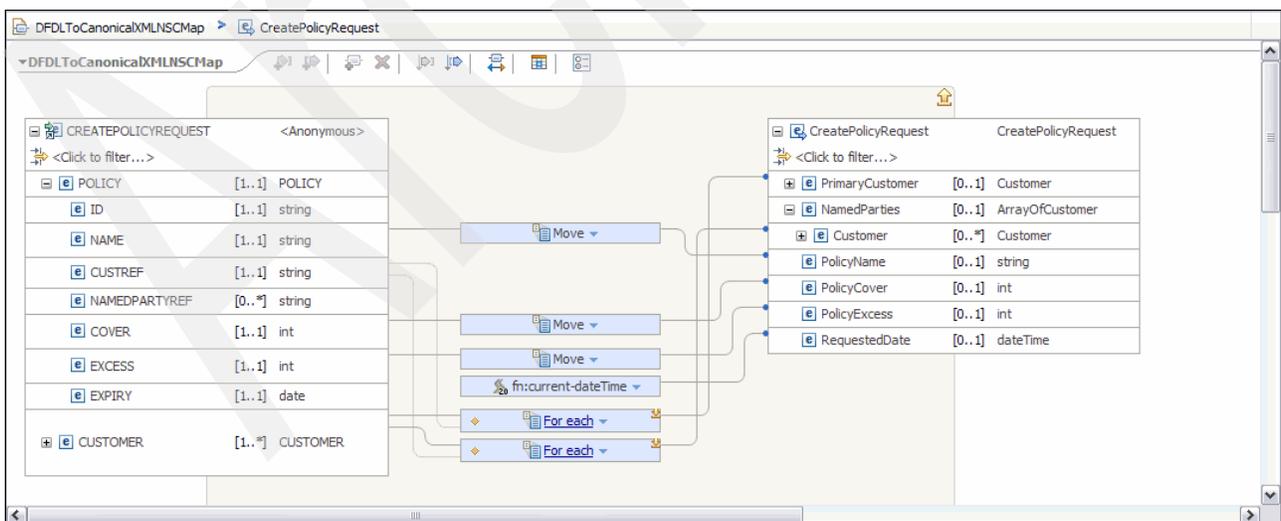


Figure 7-28 Mapping the CreateClaim operation

An example of the transformation performed by the CREATECLAIM message shown in Figure 7-26 on page 293 is mapped to XMLNSC.

Example 7-1 shows the output.

Example 7-1 A CreateClaim request in XMLNSC Canonical Message Format

```
<?xml version="1.0" encoding="UTF-8"?>
<cp:CanonicalMessage
xmlns:cp="http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/Cano
nicalMessage" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProce
ssing/CanonicalMessage ../Resource/differentmsgs/CannonicalMessageModel.xsd ">
  <cp:Header>
    <AppId>AppId</AppId>
    <MsgId>MsgId</MsgId>
    <Requester>Requester</Requester>
    <TransmissionDate>2001-12-31T12:00:00</TransmissionDate>
    <Command>Command</Command>
    <BusinessUnit>BusinessUnit</BusinessUnit>
    <BusinessUnitCode>BusinessUnitCode</BusinessUnitCode>
  </cp:Header>
  <ApplicationMessage>
  <ClaimsProcessingRequest>
  <NewClaimRequest>
  <NewClaim>
    <ClaimDate>2012-04-22T02:00:00</ClaimDate>
    <ClaimReceivedDate>2001-11-22T12:00:00</ClaimReceivedDate>
    <ClaimReference>CLAIM0002</ClaimReference>
    <ClaimValue>2147483647</ClaimValue>
    <Customer>
      <CustomerId>CUST001</CustomerId>
      <FirstName>John</FirstName>
      <OtherNames>
        <Name>James</Name>
      </OtherNames>
      <Surname>Doe</Surname>
      <Address>1 Generic Street, Any town, USA</Address>
      <Email>johndoe@itso.example.ibm.com</Email>
    </Customer>
    <Description/>
    <Notes>
      The claims agent can add some notes into the claim data
    <Note>
      <PolicyNumber>POLICY001</PolicyNumber>
      <Status>Submitted</Status>
    </NewClaim>
    <RequestedDate>2012-03-15T20:38:22</RequestedDate>
  </NewClaimRequest>
</ClaimsProcessingRequest>
</ApplicationMessage>
<cp:Trailer>
  <Created>2012-03-28T12:00:00</Created>
  <LastModified>2012-03-28T12:00:00</LastModified>
  <LastModifiedBy>LastModifiedBy</LastModifiedBy>
</cp:Trailer>
</cp:CanonicalMessage>
```

7.4 Transforming the SOAP input message

Messages that arrive at the SOAP Input node have their message extracted and converted to canonical XML format. This section shows the process that performs this function. The process is highlighted in Figure 7-29.

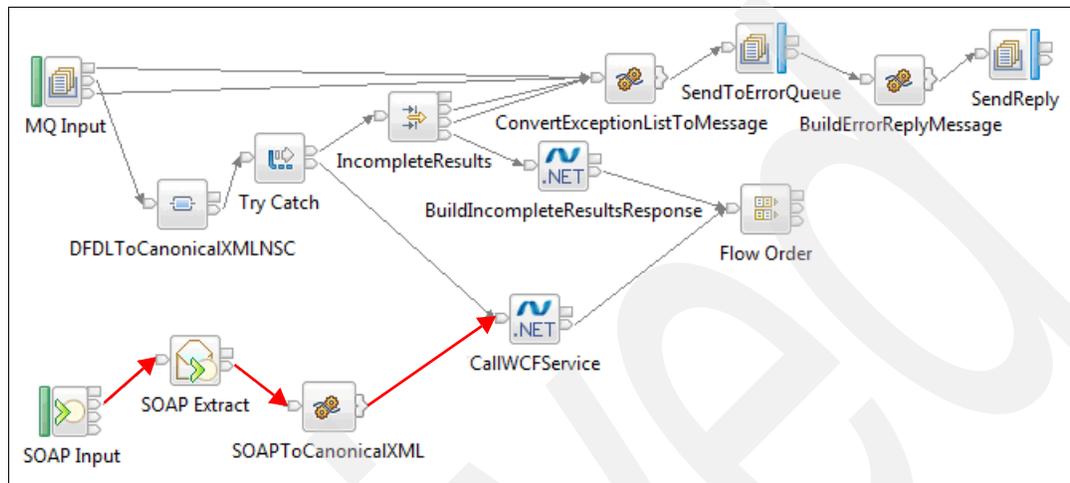


Figure 7-29 Extraction and conversion of SOAP input

The SOAP Input message for this flow contains only the ClaimReference information of the claim that needs to be returned. An example input message is shown in Example 7-2.

Example 7-2 SOAPInput Message example

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/SO
APMessage">
  <soapenv:Header/>
  <soapenv:Body>
    <soap:ClaimRefElement>CLAIM0001</soap:ClaimRefElement>
  </soapenv:Body>
</soapenv:Envelope>
```

To transform this message to the Canonical Message Format, the SOAP Envelope first needs to be removed. WebSphere Message Broker provides the SOAPEExtract node to perform this task. The default properties for this node are correct for the scenario.

Note: The SOAPEExtract node places the removed SOAP Envelope in the LocalEnvironment tree where it can be used by other parts of the flow. In this example, we allow the SOAP Reply node to generate a new output SOAP Envelope; however, one common pattern is to retrieve this SOAP Envelope from the Local Environment prior to making the reply.

After the SOAP Envelope is extracted, it is easy to construct some ESQL to move the ClaimRef element into an appropriate place in the CanonicalMessage model. However, there is a lot of information that is required by the Canonical Message Format that is not supplied by the input message. The SOAPToCanonicalXML ESQL compute node is used to assign default values to these fields.

To transform this message to the Canonical Message Format, by removing the SOAP Envelope:

1. Double-click the **SOAPToCanonicalXML** Compute node to open the ESQL for the node.
2. Add the following namespace declaration at the top of the ESQL file:

```
DECLARE sp NAMESPACE  
'http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/SOAPMessage';
```

3. Replace the generated code with the code shown in Example 7-3.

Example 7-3 The SOAPToCanonicalXML Compute node

```
CREATE COMPUTE MODULE ConvertToCanonicalXML  
  CREATE FUNCTION Main() RETURNS BOOLEAN  
  BEGIN  
    SET OutputRoot.Properties = InputRoot.Properties;  
  
    --Create an XMLNSC message  
    CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC');  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Header.AppId =  
      'WebCustomer';  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Header.BusinessUnit =  
      'OnlineOperations';  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Header.BusinessUnitCode =  
      'ONLINE1';  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Header.TransmissionDate =  
      CURRENT_TIMESTAMP;  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Header.MsgId = 'SOAP' ||  
      CAST(CURRENT_TIMESTAMP AS CHARACTER);  
  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ApplicationMessage  
      .ClaimsProcessingRequest.ViewClaimRequest.RequestedDate =  
      CURRENT_TIMESTAMP;  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ApplicationMessage  
      .ClaimsProcessingRequest.ViewClaimRequest.ClaimReference =  
      FIELDVALUE(InputRoot.XMLNSC.sp:ClaimRefElement);  
  
    --Create the trailer  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Trailer.Created =  
      CURRENT_TIMESTAMP;  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Trailer.LastModified =  
      CURRENT_TIMESTAMP;  
    SET OutputRoot.XMLNSC.ns:CanonicalMessage.ns:Trailer.LastModifiedBy =  
      'WebSphere Message Broker ' || BrokerVersion || ' - ' ||  
      BrokerName || '.' || MessageFlowLabel;  
    RETURN TRUE;  
  END;
```

Example 7-4 shows the XMLNSC message after transformation by this node.

Example 7-4 A ViewClaimRequest in XMLNSC Canonical Message Format

```
<?xml version="1.0" encoding="UTF-8"?>  
<cp:CanonicalMessage  
  xmlns:cp="http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/CanonicalMessage" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/CanonicalMessage ../Resource/differentmsgs/CannonicalMessageModel.xsd ">
  <cp:Header>
    <AppId>WebCustomer</AppId>
    <MsgId>SOAP2012-03-28T12:23:46</MsgId>
    <TransmissionDate>2012-03-28T12:23:46</TransmissionDate>
    <Command>Command</Command>
    <BusinessUnit>OnlineOperations</BusinessUnit>
    <BusinessUnitCode>ONLINE1</BusinessUnitCode>
  </cp:Header>
  <ApplicationMessage>
    <ClaimsProcessingRequest>
      <ViewClaimRequest>
        <ClaimReference>CLAIMf5f81da177294fe595512104c400ce93</ClaimReference>
        <RequestedDate>2012-03-15T20:38:22</RequestedDate>
      </ViewClaimRequest>
    </ClaimsProcessingRequest>
  </ApplicationMessage>
  <cp:Trailer>
    <Created>2012-03-28T12:23:46</Created>
    <LastModified>2012-03-28T12:23:46</LastModified>
    <LastModifiedBy>WebSphere Message Broker 8.0.0.0 - BRK8.ClaimsProcessingFlow</LastModifiedBy>
  </cp:Trailer>
</cp:CanonicalMessage>

```

4. Close the ESQL, and in the Basic tab of the Properties view for the SOAPToCanonicalXML node, enter ConvertToCanonicalXML in the ESQL module field, as shown in Figure 7-30.

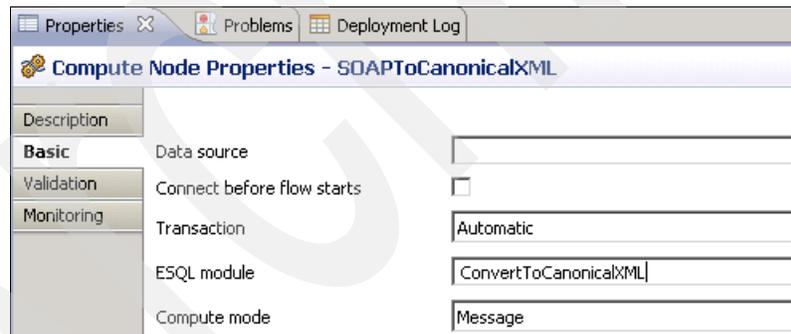


Figure 7-30 Enter the name of the ESQL module

7.5 Creating a .NETCompute node to consume the ClaimsProcessingWcfService

The main function of the ClaimProcessing application is to call the service developed in Chapter 6, “Scenario: Integration Windows Communication Foundation in message flows - Part 1” on page 163. The CallWCFService .NETCompute node uses a WCF client generated using MEX to invoke the service. Figure 7-31 on page 299 shows the location of the .NETCompute node in the message flow.

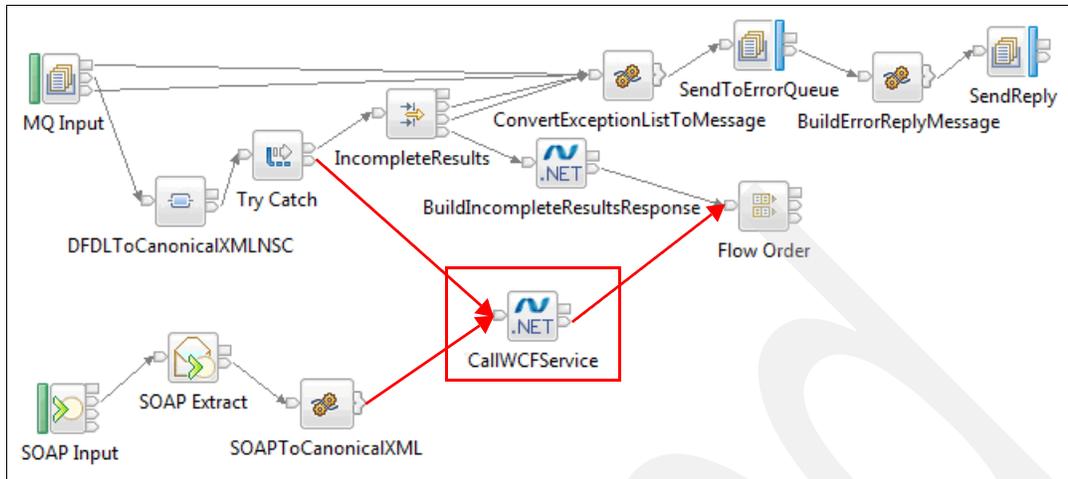


Figure 7-31 CallWCFService and associated input and output connections

To create a Visual Studio solution to contain all of the .NET code required by this scenario:

1. Double-click the **CallWCFService** node to open Visual Studio.
2. Select **File** → **New** → **Project**.
3. From the Installed Templates panel, select **Visual C#** → **Message Broker** → **Project to modify a Message Broker Message**.

Perform the following steps:

- a. In the Name field, enter `ClaimsProcessingBrokerProject`.
 - b. In the Location field, use the **Browse** button to choose a location for your project.
 - c. In the Solution name field, enter `ClaimsProcessingBrokerSolution`.
4. Click **OK** to save the project.

The New Project Wizard creates a skeleton compute node called `ModifyNode` in the project for you in a namespace that is named after the project. It is best practice to follow Microsoft naming guidelines when selecting a namespace for a project. Microsoft recommends using the format `CompanyName.TechnologyName.Qualifier` where `Qualifier` can be any unique dotted path. In this example, use the `Ibm.Broker.Example.ClaimsProcessingBroker` namespace. Note that this is different from the namespace used in Chapter 6, “Scenario: Integration Windows Communication Foundation in message flows - Part 1” on page 163 for the WCF Service. This naming differentiation is to avoid any potential name clashes with the Service.

To change the default namespace so that new files are created in the `Ibm.Broker.Example.ClaimsProcessingBroker` namespace:

1. In the Solution Explorer, right-click **ClaimsProcessingBrokerProject**, and select **Properties**.
2. In the Application tab, enter `Ibm.Broker.Example.ClaimsProcessingBroker` in the default namespace field, as shown in Figure 7-32 on page 300.

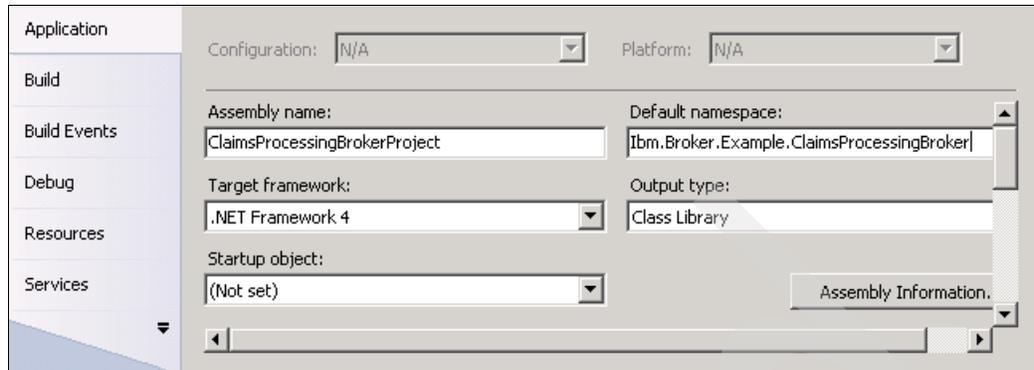


Figure 7-32 Setting the default namespace for the new project

3. Click **File** → **Save Selected Items** to save the project.

Setting the default namespace for the project ensures that new files created in this project are created in the correct namespace. However, existing files are not moved into the new namespace. Use the following procedure to move the `ModifyNode` skeleton class into the new namespace:

1. Double-click **ModifyNode.cs** in the Solution Explorer to open in the editor.
2. Change the namespace declaration to match Example 7-5.

Example 7-5 Updating the namespace for ModifyNode.cs

```
namespace Ibm.Broker.Example.ClaimsProcessingBroker
```

3. Press **Ctrl+S** to save `ModifyNode.cs`.

The New Project Wizard created the `ModifyNode.cs` skeleton file. However, rename the node class to something more descriptive. To rename the `ModifyNode.cs` node:

1. In the Solution Explorer, right-click **ModifyNode.cs**, and select **Rename**.
2. Enter `CallWCFNode.cs` as the new name, and press **Enter**. When asked if you want to perform a rename of all references, click **Yes**.
3. If `CallWCFNode` is not already open for editing, double-click it in the Solution Explorer to open it.
4. Make sure the class declaration has the new name:

```
public class CallWCFNode : NBComputeNode
```

If the name is not `CallWCFNode`:

- a. Highlight the class declaration for **ModifyNode**, right-click and select **Refactor** → **Rename**, as shown in Figure 7-33 on page 301.

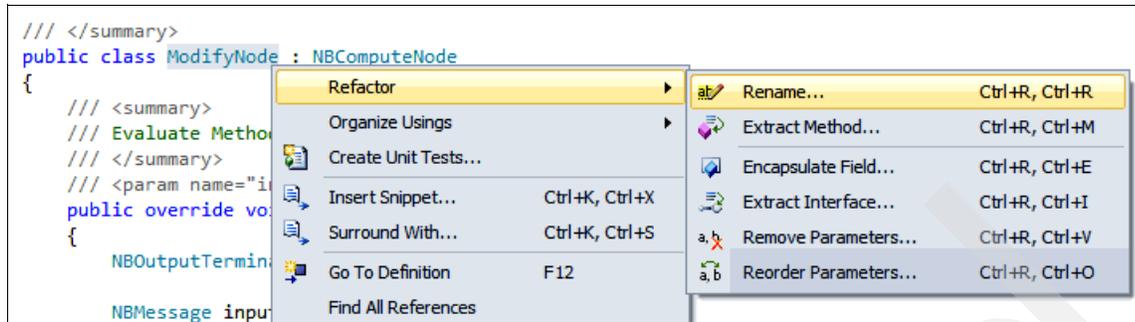


Figure 7-33 Renaming the ModifyNode class

- b. In the Rename dialog, enter CallWCFNode in the New name field. Click **OK**.
- c. In the Preview Changes dialog, click **Apply**.
5. Add references that are needed for the code in this class:
 - a. Right-click **References** in the Solution Explorer.
 - b. Select **Add Reference**.
 - c. In the .NET tab, select **System.Runtime.Serialization**, and click **OK**.
 - d. Repeat this process to add a reference to System.ServiceModel.
6. Add the following using statements to the top of CallWCFNode.cs:

```

using IBM.Broker.Example.ClaimsProcessing;
using System.Runtime.Serialization;
using System.ServiceModel;

```
7. Press Ctrl+S to save the class.

7.5.1 Creating a constant to hold the namespace prefix

Throughout this example, the Canonical Message model's namespace is referred to often when navigating the logical message tree. To avoid having to use this string inline in the code or having to define it multiple times, you can create a constant that can be referred to by any of the files in this project.

Because the constant is used for multiple diverse nodes, you can create a new Constants class to hold the namespace prefix constant.

To add the Constants class:

1. Select **Project** → **Add Class**.
2. In the Installed Templates menu, select **Visual C# Items**.
3. Select **Class**.
4. In the Name field, enter Constants.cs. Click **Add** to add the class.
5. Modify the code for Constants.cs so that it matches Example 7-6.

Example 7-6 Constants.cs

```

namespace IBM.Broker.Example.ClaimsProcessingBroker
{
    public class Constants
    {
    }
}

```

```

        public const string cp =
            "http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/CanonicalM
            essage";
        }
    }
}

```

6. Press Ctrl+S to save the class.

7.5.2 Adding the MEX-generated WCF client to the project

In 6.5, “Generating WCF client code from a MEX enabled WCF service” on page 248, you generated a WCF client using the `svcutil` tool. To allow the .NETCompute node code to invoke the service, this generated client needs to be imported into the project.

To import the client:

1. Select **Project** → **Add Existing Item**.
2. In the Add Existing Item dialog, change the Objects of Type drop-down list to read **All Files (*.*)**.
3. Navigate to the location holding the `svcutil` output.
4. Find the two files called `ClaimsProcessingWcfService.cs` and `output.config`. Select both of these files. Click **Add**.
5. Right-click the **output.config** file from the Solution Explorer, and select **Rename**.
6. Name the file `App.Config`, and press Enter.
7. Save the project.

You will now have two new files in the Solution Explorer: `App.config` and `ClaimsProcessingWcfService.cs`.

The `App.config` file contains a WCF ABC specification of the `ClaimsProcessingWcfService`. The generated code is shown in Example 7-7.

Example 7-7 An ABC specification allowing the client to invoke the `ClaimsProcessingWcfService`

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding name="TcpIpEndPoint" closeTimeout="00:01:00" openTimeout="00:01:00"
          receiveTimeout="00:10:00" sendTimeout="00:01:00" transactionFlow="false"
          transferMode="Buffered" transactionProtocol="OleTransactions"
          hostNameComparisonMode="StrongWildcard" listenBacklog="10"
          maxBufferPoolSize="524288" maxBufferSize="65536" maxConnections="10"
          maxReceivedMessageSize="65536">
          <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
            maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
          <security mode="Transport">
            <transport clientCredentialType="Windows" protectionLevel="EncryptAndSign" />
            <message clientCredentialType="Windows" />
          </security>
        </binding>
      </netTcpBinding>
    </bindings>
  </system.serviceModel>
</configuration>

```

```

</netTcpBinding>
</bindings>
<client>
  <endpoint address="net.tcp://localhost:8523/" binding="netTcpBinding"
    bindingConfiguration="TcpIpEndPoint" contract="IClaimsProcessingWcfService"
    name="TcpIpEndPoint">
    <identity>
      <userPrincipalName value="DAVICRIGW500\davicrig" />
    </identity>
  </endpoint>
</client>
</system.serviceModel>
</configuration>

```

Note: The svcutil utility generated default configuration options for the client based on the transport being used. In this instance, the `maxReceivedMessageSize` is set to 65536. The svcutil gave these configuration options inline rather than using a binding configuration. The configuration options are specific to the client and not necessarily the same as the service. For example, it is possible to permit the client to read larger messages than the service and vice versa. The configuration options do not form part of the Service Contract between the client and the service.

Notice that the code in the `ClaimsProcessingWcfService` file that you added looks substantially different from the code for the class you created in 6.3, “Developing the WCF Service” on page 168. This difference is because the file actually contains a proxy for the WCF service. If you examine the proxy class in the Class View, you will see that this proxy object consists only of the methods defined in the Service Contract, as shown in Figure 7-34.

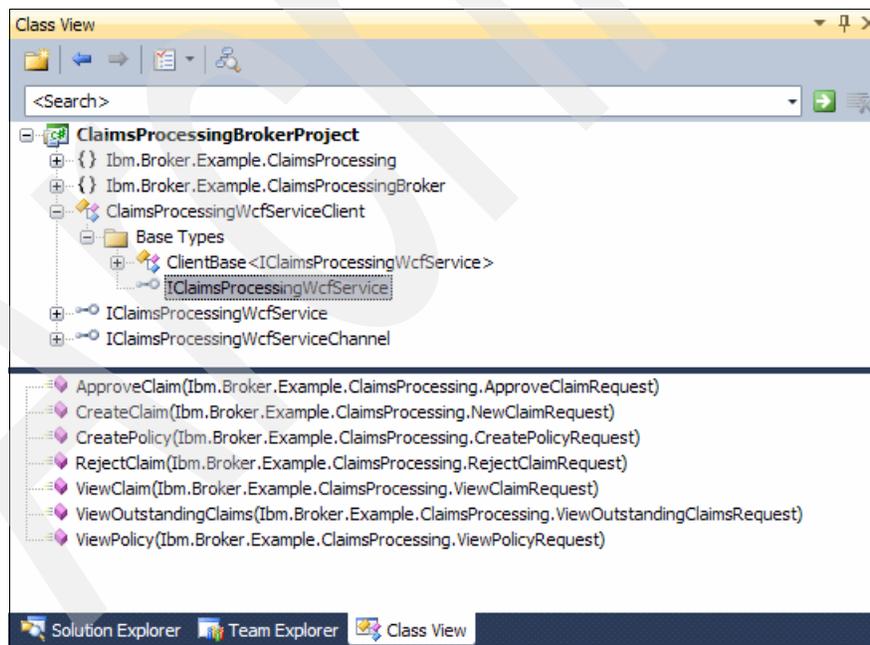


Figure 7-34 Examining the WCF proxy in the class view

After the WCF client is added to the project, any other class in the project can instantiate the proxy using code, as shown in Example 7-8 on page 304.

```
ClaimsProcessingWcfServiceClient serviceClient = new  
ClaimsProcessingWcfServiceClient();
```

7.5.3 Developing helper methods to interact with the WCF service

To separate concerns within the code, a single method in the .NETCompute node will handle all of the calls to the WCF client. This allows error handling associated with the client to be handled in one location.

For this example, a single instance of the client object is shared by all threads accessing the flow. The actual client constructor is called inline using a lazy loading pattern in the main line of code execution. To do so, the proxy object must be declared as a static data member of the node class.

Add the following code to the class definition for the CallWCFNode:

```
private static ClaimsProcessingWcfServiceClient serviceClient = null;
```

Not constructing the proxy when we construct the node means that construction of the proxy and reconstruction in an error scenario are handled by the same code block, which simplifies the application code needed. This also means that we need to take a lock on the object in fewer locations in the code.

Note: Constructing the static object in either the node constructor or during static initialization for this example is safe without taking a lock because Broker node framework will not allow a thread to drive the evaluate method until after the node constructor has completed and static initialization can only occur once for each class per Application Domain.

7.5.4 Concurrency

Although access to the proxy objects Service Contract methods are threadsafe, there are several complications to consider when using a WCF proxy in a multi-threaded environment, such as in a message flow with multiple additional instances.

Although the underlying communication is threadsafe and callers to the service do not receive messages intended for another thread, this is not the same as supporting concurrent access. If the underlying communication channel is not fully concurrent, requests might be queued by the channel.

Additionally, if one of the requests faults the channel, the application code has to cope with reinitializing the proxy and deal with the synchronization issues that result from this considerations.

To simplify the access to the WCF proxy object, the access is serialized so that only one message flow thread can make a WCF invocation at a time. In a production system where performance is critical, you might consider using a pooled approach to proxy objects or creating a proxy for each message flow thread, storing the proxy in thread local storage.

In this example, serialization is performed using the C# lock call to obtain a mutual exclusion lock on a static member of the node class:

1. Add the code shown in Example 7-9 to the class definition for CallWCFNode to create the static object to use for locking.

Example 7-9 Creating a static Object to use for serializing access to the WCF proxy

```
private static Object proxyLock = new Object();
```

The WCF service invocation is performed by a method called CallClaimsProcessingService that takes a ClaimsProcessingRequest as input and returns a ClaimsProcessingResponse. The message signature uses the base classes so that no matter what type of request is being performed all execution will pass through this method. This is important so that the lock operates on all service invocations.

2. Add the code shown in Example 7-10 to the class definition of CallWCFNode.

Example 7-10 The CallClaimsProcessingService method in CallWCFNode.cs

```
private ClaimsProcessingResponse CallClaimsProcessingService(ClaimsProcessingRequest request)
{
    //we must be the only thread accessing the shared client
    lock (proxyLock)
    {
        if (serviceClient == null ||
            serviceClient.InnerChannel.State.Equals(CommunicationState.Faulted))
        {
            serviceClient = new ClaimsProcessingWcfServiceClient();
        }
        try
        {
            ClaimsProcessingResponse response;
            switch(request.TypeOfRequest)
            {
                case RequestType.ApproveClaim:
                    response=serviceClient.ApproveClaim((ApproveClaimRequest)request);
                    break;
                case RequestType.CreatePolicy:
                    response=serviceClient.CreatePolicy((CreatePolicyRequest)request);
                    break;
                case RequestType.NewClaim:
                    response=serviceClient.CreateClaim((NewClaimRequest)request);
                    break;
                case RequestType.RejectClaim:
                    response=serviceClient.RejectClaim((RejectClaimRequest)request);
                    break;
                case RequestType.ViewClaim:
                    response=serviceClient.ViewClaim((ViewClaimRequest)request);
                    break;
                case RequestType.ViewOutstandingClaims:
                    response=serviceClient.ViewOutstandingClaims(
                        (ViewOutstandingClaimsRequest)request);
                    break;
                case RequestType.ViewPolicy:
                    response=serviceClient.ViewPolicy((ViewPolicyRequest)request);
                    break;
                default:
                    throw new NBRrecoverableException(
                        "Unrecognised ClaimsProcessing Command " + request.TypeOfRequest);
            }
        }
    }
}
```

```

    }
    return response;
}
catch(Exception e)
{
    /* WCF can cause the following exceptions:
    * System.TimeoutException
    * System.ServiceModel.CommunicationException
    * System.ServiceModel.AddressAlreadyInUseException
    * System.ServiceModel.AddressAccessDeniedException
    * System.ServiceModel.CommunicationObjectFaultedException
    * System.ServiceModel.CommunicationObjectAbortedException
    * System.ServiceModel.EndpointNotFoundException
    * System.ServiceModel.ProtocolException
    * System.ServiceModel.ServerTooBusyException
    *
    * Rather than handling all of these the approach taken
    * is to recreate the proxy
    * object for all exception except for ones that are
    * explicitly known to be safe
    */
    if(!(e is FaultException) || serviceClient.InnerChannel.State
        .Equals(CommunicationState.Faulted))
    {
        serviceClient.Abort();
        serviceClient = null;
    }
    throw;
}
}
}
}

```

The first thing the method does is take a lock on the proxyLock object to ensure that this thread is the only one accessing the static serviceClient object. If the serviceClient is null, a new serviceClient is constructed and assigned to the static variable, as shown in Example 7-11.

Example 7-11 Constructing the WCF proxy object

```

if (serviceClient == null)
{
    serviceClient = new ClaimsProcessingWcfServiceClient();
}

```

The serviceClient can be null for one of two reasons:

- ▶ This is the first time through the flow.
- ▶ There is an exception that left the service proxy in an unusable state, and it needs to be recreated.

After the service proxy is known to exist, the switch statement simply casts the request parameter based on the ClaimsProcessingRequest.TypeOfRequest field. This allows the code to know that the operation must be invoked on the serviceClient.

The code also checks that the underlying communication channel owned by the serviceClient did not enter the Faulted state. This can happen if the session times out or there is some other issue with connectivity while the flow is waiting for a message.

For example, if the request received has a `TypeOfRequest` value of `RequestType.ViewPolicy`, the request is cast to a `ViewPolicyRequest` object, and then the serviceClient's `ViewPolicy` method is invoked, as shown in the code in Example 7-12.

Example 7-12 Making the WCF invocation

```
case RequestType.ViewPolicy:
    response=serviceClient.ViewPolicy((ViewPolicyRequest)request);
    break;
```

The call `serviceClient.ViewPolicy()` causes the WCF framework to serialize the data held in the request parameter and send it using TCP/IP to the service. The service then executes its `ViewPolicy()` method that reads the data from the file system and then serializes the response message with the `DataContractSerializer` to send back over the wire to the serviceClient.

The result is that after the call to `serviceClient.ViewPolicy()` completes, the request variable holds a `ViewPolicyResponse` object containing the policy details returned from the WCF Service. Note that although response was declared to be of type `ClaimsProcessingResponse`, the `ViewPolicyResponse` object derives from this base class. To access any fields specific to the `ViewPolicyResponse` class, the calling code must cast the response back to the correct type.

In the default case where we are unable to identify the correct derived class based on the value of the `TypeOfRequest` field, an `NBRecoverableException` is thrown to return a meaningful error message to the caller.

7.5.5 Handling exceptions when invoking the WCF service

It is possible for the invocation of the service to fail, and the calling code must be prepared to deal with the failure gracefully. WCF failures can be due to either an exception encountered by the WCF service or exceptions caused by the underlying communications framework.

In all cases where the remote WCF service experiences an exception, the exception is wrapped in a `FaultException` when it is presented to the client. These `FaultException` objects can be grouped into two main types:

- ▶ `FaultExceptions` that are part of the declared Fault Contract
- ▶ Unexpected `FaultExceptions`

In the Fault Contract for the `ClaimsProcessingWcfService`, we defined two types of Fault in the Fault Contract, the `ClaimsProcessingFault` and the `ViewOutstandingClaimsIncompleteResultsFault`. By declaring these as part of the Fault Contract, we make the assertion that the client code knows about these Fault types and can take appropriate action.

If the Fault received is part of the fault contract, the underlying communication channel does not enter the faulted state, and it is possible to reuse the serviceClient. In general, take care when reusing WCF proxy objects when a Fault occurred because the state on the remote server might not be consistent. Some channels will, for example, close a session on receipt of a fault, which can lead to application code mistakenly trying to reuse the service, assuming the session is still valid.

For the example service, the service recovers from all `ClaimsProcessingFault`'s, and there is no stateful information stored in the service itself (because the state is stored on the file system). For this reason it is safe to reuse the serviceClient in any case where the underlying communications channel is not in the faulted state.

Unexpected FaultExceptions are the result of the remote WCF service encountering a managed exception that is not handled. In a production environment, a generic FaultException is returned; however, during development of an application, it can be useful to be able to see the details of these exceptions from the client. To enable this behavior, the App.Config of the WCF service can be updated with the following line:

```
<serviceDebug includeExceptionDetailInFaults="true"/>
```

It is also possible for the WCF framework itself to throw a number of exceptions derived from the System.ServiceModel.CommunicationException and System.TimeoutException base classes. These exceptions indicate some underlying problem with the network transport, such as for example, if the network cable is unplugged from the machine. In these cases, the serviceClient needs to be recreated.

The error handling in the CallClaimsProcessingService method has only the scope to handle the cleaning up of the serviceClient, if appropriate. The base type Exception is caught, and then the exception is tested to see if the serviceClient needs recreated. For the purposes of this example, we assume that any FaultException is safe unless the underlying communication channel is in the Faulted state. If the serviceClient needs to be recreated, the Abort() method is called to release local resources and transition the wcf proxy object to the closed state. As shown in Example 7-13, the serviceClient variable is then set to null so that the next invocation of the CallClaimsProcessingService will recreate the client.

Example 7-13 The catch block in CallClaimsProcessingService controls when the serviceClient object is recreated

```
catch(Exception e)
{
    /* WCF can cause the following exceptions:
    * System.TimeoutException
    * System.ServiceModel.CommunicationException
    * System.ServiceModel.AddressAlreadyInUseException
    * System.ServiceModel.AddressAccessDeniedException
    * System.ServiceModel.CommunicationObjectFaultedException
    * System.ServiceModel.CommunicationObjectAbortedException
    * System.ServiceModel.EndpointNotFoundException
    * System.ServiceModel.ProtocolException
    * System.ServiceModel.ServerTooBusyException
    *
    * Rather than handling all of these the approach
    * taken is to recreate the proxy
    * object for all exception except for ones that are
    * explicitly known to be safe
    */
    if(!(e is FaultException) ||
        serviceClient.InnerChannel.State.Equals(CommunicationState.Faulted))
    {
        serviceClient.Abort();
        serviceClient = null;
    }
    throw;
}
```

We will see in 7.5.13, “Handling FaultExceptions in the Evaluate() method” on page 321 that the known FaultException types that are part of the Fault Contract will cause different behavior of the CallWCFNode.evaluate() method.

7.5.6 Ensuring that the serviceClient is closed on flow deletion

A WCF proxy that is not closed might be holding resources open on the service, so it is essential to ensure that when an application is finished with the serviceClient, its Close() method is called. The generated proxy is derived from ClientBase that ultimately implements the IDisposable interface, ensuring that when an object is destroyed, the Close() method is called. The only problem is that the Close() method itself can cause an exception because it requires a network round trip to the service to inform it to release any resources held open by the client.

If there is a communications failure when the object is disposed of, this can lead to the finalizer for the node throwing an exception. To prevent this, the CallWCFNode should call the Close() method itself inside a try/catch block. If an exception occurs when trying to close the serviceClient, the Abort() method is called instead.

The Abort() method, in contrast to the Close() method, causes the underlying object to move immediately to a closed state and any local resources released without making a network call to the client.

The CallWCFNode overrides the onDelete() method that is called by the Broker node framework when a flow is not yet deployed to ensure that either the Close() or the Abort() method is called before the instance of the node is destroyed:

Add the code shown in Example 7-14 to the class definition for CallWCFNode.

Example 7-14 Overriding NBComputeNode.OnDelete() to ensure the serviceClient is closed

```
public override void OnDelete()
{
    //ensure we close the serviceClient
    lock (proxyLock)
    {
        if (serviceClient != null)
        {
            try
            {
                serviceClient.Close();
                serviceClient = null;
            }
            catch (Exception)
            {
                serviceClient.Abort();
                serviceClient = null;
            }
        }
    }
}
```

7.5.7 Creating methods to add datatypes to the message tree

To simplify the creation of the response messages in the message tree, the node has a number of helper methods for creating the main datatypes in the tree. These methods called AddClaimElementToParent(), AddPolicyElementToParent(), AddCustomerElementToParent() and AddPaymentElementToParent() all follow a similar implementation pattern.

1. Add the `AddCustomerElementToParent()` method shown in Example 7-15 to the class definition for `CallWCFNode`.

Example 7-15 The `AddCustomerElementToParent` method in `CallWCFNode.cs`

```
private void AddCustomerElementToParent(Customer customer, NBElement parent,
                                        string tagName = "Customer")
{
    NBElement customerElement = parent.CreateLastChild(tagName);
    customerElement.CreateLastChild("CustomerId").SetValue(customer.CustomerId);
    customerElement.CreateLastChild("FirstName").SetValue(customer.FirstName);
    NBElement otherNamesElement = customerElement.CreateLastChild("OtherNames");
    foreach(string name in customer.OtherNames)
    {
        otherNamesElement.CreateLastChild("Name").SetValue(name);
    }
    customerElement.CreateLastChild("Surname").SetValue(customer.Surname);
    customerElement.CreateLastChild("Address").SetValue(customer.Address);
    customerElement.CreateLastChild("Email").SetValue(customer.Email);
}
```

This method takes a `Customer` object, such as one that might have been returned by an invocation to the WCF service, and creates a new element as a child of the parent parameter with a tree-based representation of this customer. The final parameter allows the parent tag name to be set, which defaults to the name of the class.

The method uses the standard tree manipulation methods, `NBElement.CreateLastChild()` and `NBElement.SetValue()`.

Note: The `Customer`, `Claim`, `InsurancePolicy`, and `Payment` types are able to be serialized with an `XMLSerializer`, as you saw in 6.3.5, “Creating private methods to serialize and deserialize the Datatypes” on page 180. We can therefore also have used the `NBElement.CreateLastChildFromBitStream()` method. The XML created by the `XMLSerializer` is not quite in the right format; however, some transformation is still needed.

2. Implement the other three helper methods by adding the code shown in Example 7-16 to the `CallWCFNode` class definition.

Example 7-16 Add the other 3 helper methods to `CallWCFNode.cs`

```
private void AddPaymentElementToParent(Payment payment, NBElement parent,
                                        string tagName = "Payment")
{
    NBElement paymentElement = parent.CreateLastChild(tagName);
    paymentElement.CreateLastChild("Amount").SetValue(payment.Amount);
    AddCustomerElementToParent(payment.Payee, paymentElement);
    paymentElement.CreateLastChild("Underwriter").SetValue(payment.Underwriter);
}

private void AddPolicyElementToParent(InsurancePolicy policy,
                                        NBElement parent, string tagName = "Policy")
{
    NBElement policyElement = parent.CreateLastChild(tagName);
    policyElement.CreateLastChild("PolicyNumber").SetValue(policy.PolicyNumber);
    AddCustomerElementToParent(policy.Policyholder, policyElement,
                                "PolicyHolder");
}
```

```

NBElement namedPartiesElement = policyElement.CreateLastChild("NamedParties");
foreach (Customer customer in policy.NamedParties)
{
    AddCustomerElementToParent(customer, namedPartiesElement);
}
policyElement.CreateLastChild("PolicyCover").SetValue(policy.PolicyCover);
policyElement.CreateLastChild("PolicyExcess").SetValue(policy.PolicyExcess);
policyElement.CreateLastChild("PolicyExpiryDate").SetValue(
    policy.PolicyExpiryDate);
policyElement.CreateLastChild("PolicyName").SetValue(policy.PolicyName);
policyElement.CreateLastChild("PolicyStartDate").SetValue(
    policy.PolicyStartDate);
policyElement.CreateLastChild("Underwriter").SetValue(policy.Underwriter);
}

private void AddClaimElementToParent(Claim claim, NBElement parent,
    string tagName = "Claim")
{
    NBElement claimElement = parent.CreateLastChild(tagName);
    claimElement.CreateLastChild("ClaimDate").SetValue(claim.ClaimDate);
    claimElement.CreateLastChild("ClaimReceivedDate").SetValue(
        claim.ClaimReceivedDate);
    claimElement.CreateLastChild("ClaimReference").SetValue(claim.ClaimReference);
    claimElement.CreateLastChild("ClaimValue").SetValue(claim.ClaimValue);
    AddCustomerElementToParent(claim.Customer, claimElement);
    claimElement.CreateLastChild("Description").SetValue(claim.Description);
    claimElement.CreateLastChild("Notes").SetValue(claim.Notes);
    claimElement.CreateLastChild("PolicyNumber").SetValue(claim.PolicyNumber);
    claimElement.CreateLastChild("Status").SetValue(claim.Status);
}

```

7.5.8 Creating a helper method to build Customer object from the Logical Message Tree

The Customer datatype is used in the parameters for several of the request operations in this scenario. We therefore create a helper method that is responsible for taking an NBElement in the tree and converting this into a .NET Customer object that can be used to construct the data used making the WCF Service invocation.

1. Add the method shown Example 7-17 in to CallWCFNode.cs.

Example 7-17 The CreateCustomerFromElement method in CallWCFNode.cs

```

private Customer CreateCustomerFromElement(NBElement customerElement)
{
    Customer customer = new Customer();
    customer.CustomerId = customerElement["CustomerId"].ValueAsString;
    customer.FirstName = customerElement["FirstName"].ValueAsString;
    NBElement otherNamesElement = customerElement["OtherNames"];
    string[] otherNamesArray = new string[otherNamesElement.Children().Count()];
    for (int i = 0; i < otherNamesElement.Children().Count(); i++)
    {
        otherNamesArray[i] =
            otherNamesElement.Children().ElementAt(i).ValueAsString;
    }
    customer.OtherNames = otherNamesArray;
    customer.Surname = customerElement["Surname"].ValueAsString;
}

```

```

        customer.Email = customerElement["Email"].ValueAsString;
        return customer;
    }

```

Note: The `NBElement.Children()` method returns an `IEnumerable`, so we can use a method like `Count()` or the `for-each` construct. In this instance, a plain `for` loop is used because we need to index into the `otherNamesArray`.

7.5.9 Preparing the `ClaimsProcessingRequest` object

The pattern of execution that is used in the main body of the `Evaluate()` method is to first locate the appropriate fields in the message tree to be used to make the request, and then convert the data in the logical message tree into an appropriate derived class of `ClaimsProcessingRequest`.

This request object is then used to make the WCF invocation using the `CallClaimsProcessingService()` method. The returned response is then converted from a .NET object into a message tree for propagation to the of the flow.

To simplify the evaluate method, the preparation of the request object is managed by a single call to the `PrepareRequest()` method. This method uses a `switch` statement to delegate to the appropriately overloaded form of the method that corresponds to the type of request being processed:

1. Add the code shown in Example 7-18 to the `CallWCFNode` class definition.

Example 7-18 The `PrepareRequest` method in `CallWCFNode.cs`

```

private void PrepareRequest(ClaimsProcessingRequest request, NBElement requestElement)
{
    switch(request.TypeOfRequest)
    {
        case RequestType.ApproveClaim:
            PrepareRequest((ApproveClaimRequest) request, requestElement);
            break;
        case RequestType.CreatePolicy:
            PrepareRequest((CreatePolicyRequest)request, requestElement);
            break;
        case RequestType.NewClaim:
            PrepareRequest((NewClaimRequest)request, requestElement);
            break;
        case RequestType.RejectClaim:
            PrepareRequest((RejectClaimRequest)request, requestElement);
            break;
        case RequestType.ViewClaim:
            PrepareRequest((ViewClaimRequest)request, requestElement);
            break;
        case RequestType.ViewOutstandingClaims:
            PrepareRequest((ViewOutstandingClaimsRequest)request, requestElement);
            break;
        case RequestType.ViewPolicy:
            PrepareRequest((ViewPolicyRequest)request, requestElement);
            break;
        default:
            throw new InvalidOperationException("Generic request not possible!");
    }
}

```

The switch statement operates on the `TypeOfRequest` field of the request parameter. Note that if the type of request is not recognized, an `InvalidOperation` is thrown.

- Each of the specific `PrepareRequest` methods follows a similar pattern. Add the method shown in Example 7-19 to the `CallWCFNode` class definition.

Example 7-19 The `PrepareRequest()` method for `CreatePolicyRequest` message type in `CallWCFNode.cs`

```
private void PrepareRequest(CreatePolicyRequest request, NBElement requestElement)
{
    request.PrimaryCustomer =
        CreateCustomerFromElement(requestElement["PrimaryCustomer"]);
    IEnumerable<NBElement> namedPartiesElements =
        requestElement["NamedParties"].Children();
    Customer[] namedParties = new Customer[namedPartiesElements.Count()];
    for (int i = 0; i < namedParties.Count(); i++)
    {
        namedParties[i] =
            CreateCustomerFromElement(namedPartiesElements.ElementAt(i));
    }
    request.NamedParties = namedParties;
    request.PolicyName = requestElement["PolicyName"].ValueAsString;
    request.PolicyCover = requestElement["PolicyCover"].GetInt32();
    request.PolicyExcess = requestElement["PolicyExcess"].GetInt32();
}
```

The `requestElement` parameter passed in is expected to correspond to the `CanonicalMessage/ApplicationMessage/ClaimsProcessingRequest/CreatePolicyRequest` element in the logical tree.

This method uses array syntax to reference message tree elements in the `requestElement` argument and uses these values to set corresponding fields in the `CreatePolicyObject` .NET object. Note that for types that are not strings, the appropriate method is called on the referenced element to get the value as the correct datatype.

- To implement the remaining `PrepareRequest()` methods for the other types of requests, add the following code to the `CallWCFNode` class definition.

Example 7-20 The remaining `PrepareRequest()` methods in `CallWCFNode.cs`

```
private void PrepareRequest(ApproveClaimRequest request, NBElement requestElement)
{
    request.AgentReference = requestElement["AgentReference"].ValueAsString;
    request.ClaimReference = requestElement["ClaimReference"].ValueAsString;
    request.ValueApproved = requestElement["ValueApproved"].GetInt32();
}

private void PrepareRequest(NewClaimRequest request, NBElement requestElement)
{
    NBElement claimElement = requestElement["NewClaim"];
    Claim claim = new Claim();
    claim.ClaimDate = claimElement["ClaimDate"].GetDateTime();
    claim.ClaimReceivedDate = DateTime.Now;
    claim.ClaimReference = claimElement["ClaimReference"].ValueIsNull ?
        null : claimElement["ClaimReference"].ValueAsString;
    claim.ClaimValue = claimElement["ClaimValue"].GetInt32();
    claim.Customer = CreateCustomerFromElement(claimElement["Customer"]);
    claim.Description = claimElement["Description"].ValueAsString;
    claim.Notes = claimElement["Notes"].ValueAsString;
}
```

```

        claim.PolicyNumber = claimElement["PolicyNumber"].ValueAsString;
        claim.Status = ClaimState.Submitted;
        request.NewClaim = claim;
    }

    private void PrepareRequest(RejectClaimRequest request, NBElement requestElement)
    {
        request.AgentReference = requestElement["AgentReference"].ValueAsString;
        request.ClaimReference = requestElement["ClaimReference"].ValueAsString;
        request.Reason = requestElement["Reason"].ValueAsString;
    }

    private void PrepareRequest(ViewClaimRequest request, NBElement requestElement)
    {
        request.ClaimReference = requestElement["ClaimReference"].ValueAsString;
    }

    private void PrepareRequest(ViewOutstandingClaimsRequest request,
                                NBElement requestElement)
    {
        //This request type has no additional properties
        return;
    }

    private void PrepareRequest(ViewPolicyRequest request, NBElement requestElement)
    {
        request.PolicyReference = requestElement["PolicyReference"].ValueAsString;
    }

```

Note that the `ViewOutstandingClaimsRequest` does not provide any additional parameters over the base type of `ClaimsProcessingRequest`, so the `PrepareRequest()` message for this message type does not do anything.

The calling code is expected to set the fields defined by the base `ClaimsProcessingRequest` type, which is done in the `Evaluate` method with the code shown in Example 7-21.

Example 7-21 Setting the properties on the `ClaimsProcessingRequest` base type

```

//All requests share some common properties
request.ApplicationId = headerElement["AppId"].ValueAsString;
request.MessageId = headerElement["MsgId"].ValueAsString;
request.RequestDate = requestedDate;

```

7.5.10 Adding the information from the WCF response into the message tree

After a request is made to the WCF service, the data contained in the `ClaimsProcessingRequest` needs to be added to the tree. To keep message type-specific processing out of the main line of the `Evaluate()` method, this is handled by a single `AddResponseDataToTree` that takes a `ClaimsProcessingRequest` and adds the data to the provided `responseElement` in the tree. This general method delegates to a specific overridden method based on the `TypeOfResponse` field:

1. Add the code shown in Example 7-22 to the class declaration for `CallWCFNode`.

Example 7-22 The `AddResponseDataToTree` method delegates based on the `TypeOfRequest`

```

private void AddResponseDataToTree(ClaimsProcessingResponse response,
                                   NBElement responseElement)

```

```

{
    switch (response.TypeOfResponse)
    {
        case ResponseType.ApproveClaim:
            AddResponseDataToTree((ApproveClaimResponse)response,
                responseElement);
            break;
        case ResponseType.CreatePolicy:
            AddResponseDataToTree((CreatePolicyResponse)response,
                responseElement);
            break;
        case ResponseType.NewClaim:
            AddResponseDataToTree((NewClaimResponse)response,
                responseElement);
            break;
        case ResponseType.RejectClaim:
            AddResponseDataToTree((RejectClaimResponse)response,
                responseElement);
            break;
        case ResponseType.ViewClaim:
            AddResponseDataToTree((ViewClaimResponse)response,
                responseElement);
            break;
        case ResponseType.ViewOutstandingClaims:
            AddResponseDataToTree((ViewOutstandingClaimsResponse)response,
                responseElement);
            break;
        case ResponseType.ViewPolicy:
            AddResponseDataToTree((ViewPolicyResponse)response,
                responseElement);
            break;
        default:
            throw new InvalidOperationException(
                "Generic request not possible!");
    }
}

```

The specific methods use the methods defined earlier in this section to add various data types to the tree. In some cases, there is no response data that is specific to the type of response, so in this instance only, the root element of the response is created.

2. Add the specific `AddResponseDataToTree()` methods shown in Example 7-23 to the class declaration for `CallWCFNode`.

Example 7-23 Response type specific versions of `AddResponseDataToTree()`

```

private void AddResponseDataToTree(ApproveClaimResponse response,
    NBElement responseElement)
{
    NBElement approveClaimResponseElement = responseElement
        .CreateLastChild("ApproveClaimResponse");
    AddPaymentElementToParent(response.Payment, approveClaimResponseElement);
}

private void AddResponseDataToTree(CreatePolicyResponse response,
    NBElement responseElement)
{
    NBElement createPolicyResponseElement = responseElement
        .CreateLastChild("CreatePolicyResponse");
    AddPolicyElementToParent(response.Policy, createPolicyResponseElement);
}

```

```

    }

    private void AddResponseDataToTree(NewClaimResponse response,
        NBElement responseElement)
    {
        responseElement.CreateLastChild("NewClaimResponse");
        //No additional properties for this response type
    }

    private void AddResponseDataToTree(ViewClaimResponse response,
        NBElement responseElement)
    {
        NBElement viewClaimResponseElement = responseElement
            .CreateLastChild("ViewClaimResponse");
        AddClaimElementToParent(response.Claim, viewClaimResponseElement);
    }

    private void AddResponseDataToTree(ViewPolicyResponse response,
        NBElement responseElement)
    {
        NBElement viewPolicyResponseElement = responseElement
            .CreateLastChild("ViewPolicyResponse");
        AddPolicyElementToParent(response.Policy,
            viewPolicyResponseElement,
            "Policy");
    }

    private void AddResponseDataToTree(RejectClaimResponse response,
        NBElement responseElement)
    {
        responseElement.CreateLastChild("RejectClaimResponse");
        //No additional properties for this response type
    }

    private void AddResponseDataToTree(ViewOutstandingClaimsResponse response,
        NBElement responseElement)
    {
        NBElement outstandingClaimsElement = responseElement
            .CreateLastChild("ViewOutstandingClaimsResponse")
            .CreateLastChild("OutstandingClaims");
        foreach (Claim claim in response.OutstandingClaims)
        {
            AddClaimElementToParent(claim, outstandingClaimsElement);
        }
    }
}

```

7.5.11 The CallWCFNode.Evaluate() method

The Evaluate method of the CallWCFNode class is invoked by the WebSphere Message Broker node framework when execution reaches the input terminal of the corresponding .NETCompute node in the message flow.

This is the main entry point into the node and the main job of this method is to locate the information in the tree that is required to build the ClaimsProcessingRequest message, call the helper method CallClaimsProcessingService to make the WCF invocation, and finally put the details from the returned ClaimsProcessingResponse object into the message tree.

The Add Class Wizard generates skeleton code needed to access the input and output message tree, so the Evaluate method is implemented by filling in the UserCode #region block provided:

1. Add the code shown in Example 7-24 to the Evaluate() method inside the UserCode #region.

Example 7-24 Evaluate() method part 1 in CallWCFNode.cs

```
#region UserCode

//set these flags so that the error handler can tell where any tree navigations //failed
bool canonicalMessageMatched = false;
bool recognisedApplicationMessage = false;
bool requestParametersFound = false;
try
{
    NBElement canonicalMessageInputElement =
        inputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"];
    NBElement headerElement = canonicalMessageInputElement["Header"];

    canonicalMessageMatched = true;
}
```

The method starts by declaring some boolean values to use as flags so that we can detect when tree navigation failed to convert the resulting NullReferenceException into a meaningful error message. The first of these flag variables, canonicalMessageMatched, is set to true after we verify that the input message is a CanonicalMessage. Note the use of Constants.cp to reference the CanonicalMessage namespace.

Next, the specific request element is found in the logical message tree from the CanonicalMessage\ApplicationMessage\ClaimsProcessingRequest path. The name of the last child of the ClaimsProcessingRequest element is used in a switch statement to create an appropriately typed ClaimsProcessingRequest and the results assigned to the pre-declared base class variable ClaimsProcessingRequest.

Since in WCF constructors are not run on serialization and de-serialization of Data Contract types, the svcutil generated classes do not set the TypeOfRequest field in the constructor. Further the generated class actually treats this read only field as a normal field, as you can see by examining the generated code in ClaimsProcessingWcfService.cs shown in Example 7-25.

Example 7-25 Generated code for ClaimsProcessingRequest

```
[System.Runtime.Serialization.DataMemberAttribute()]
public IBM.Broker.Example.ClaimsProcessing.RequestType TypeOfRequest
{
    get
    {
        return this.TypeOfRequestField;
    }
    set
    {
        this.TypeOfRequestField = value;
    }
}
```

This means that when we create the derived class for ClaimsProcessingRequest, we also need to set the TypeOfRequest field so that subsequent code operating on the object can determine the request type without needing to use reflection.

2. Add the code shown in Example 7-26 to the UserCode region of the Evaluate() method (directly below the code in Example 7-24 on page 317) to implement this switch statement.

Example 7-26 Evaluate() method part 2 in CallWCFNode.cs

```
NBElement applicationMessageInputElement =  
  
canonicalMessageInputElement["ApplicationMessage"];  
NBElement requestElement =  
applicationMessageInputElement["ClaimsProcessingRequest"].LastChild;  
  
//Create the appropriate type of request  
ClaimsProcessingRequest request;  
switch (requestElement.Name)  
{  
    case "ApproveClaimRequest":  
        request = new ApproveClaimRequest();  
        request.TypeOfRequest = RequestType.ApproveClaim;  
        break;  
    case "CreatePolicyRequest":  
        request = new CreatePolicyRequest();  
        request.TypeOfRequest = RequestType.CreatePolicy;  
        break;  
    case "NewClaimRequest":  
        request = new NewClaimRequest();  
        request.TypeOfRequest = RequestType.NewClaim;  
        break;  
    case "RejectClaimRequest":  
        request = new RejectClaimRequest();  
        request.TypeOfRequest = RequestType.RejectClaim;  
        break;  
    case "ViewClaimRequest":  
        request = new ViewClaimRequest();  
        request.TypeOfRequest = RequestType.ViewClaim;  
        break;  
    case "ViewOutstandingClaimsRequest":  
        request = new ViewOutstandingClaimsRequest();  
        request.TypeOfRequest = RequestType.ViewOutstandingClaims;  
        break;  
    case "ViewPolicyRequest":  
        request = new ViewPolicyRequest();  
        request.TypeOfRequest = RequestType.ViewPolicy;  
        break;  
    default:  
        throw new NBRecoverableException("Unrecognised Request type " +  
                                         requestElement.Name);  
}  
  
//this is a message type the flow knows about  
recognisedApplicationMessage = true;
```

If the tree references complete successfully and the first child of the ClaimsProcessingRequest element is a type of request that the flow recognizes, the boolean flag is set to indicate that the ApplicationMessage was successfully recognized.

3. Add the code shown in Example 7-27 to the UserCode region in the Evaluate() method directly below the code in Example 7-26 on page 318.

Example 7-27 Evaluate() method part 3 in CallWCFNode.cst

```
//All requests share some common properties
request.ApplicationId = headerElement["AppId"].ValueAsString;
request.MessageId = headerElement["MsgId"].ValueAsString;
DateTime requestedDate = requestElement["RequestedDate"].GetDateTime();
request.RequestDate = requestedDate;
```

The ClaimsProcessingRequest base class defines some fields that are common across all requests. The code in Example 7-27 fills in these values from the request variable (that remember is typed to refer to the ClaimsProcessingRequest base class) by locating the AppId, MsgId, and RequestedDate from the logical message tree.

The next code segment is where most of the actual work in this node takes place. Recall that we defined three helper methods to populate the request object based on its type, make the WCF request, and the populate the message tree based on the response. These methods are now called from the Evaluate() method, as shown in Example 7-28. Implementing these methods means we avoided including a large amount of message-specific code in the Evaluate() method itself.

4. Add the following code to the UserCode region in the Evaluate() method.

Example 7-28 Evaluate() method part 4 in CallWCFNode.cs

```
//call overloaded method to fill in properties specific to each
//request type
PrepareRequest(request, requestElement);
requestParametersFound = true;

ClaimsProcessingResponse response = CallClaimsProcessingService(request);

NBElement applicationMessageOutputElement = outputRoot["XMLNSC"]
    [Constants.cp, "CanonicalMessage"]["ApplicationMessage"];
applicationMessageOutputElement.DeleteAllChildren();
NBElement claimsProcessingResponseOutputElement =
applicationMessageOutputElement.CreateLastChild("ClaimsProcessingResponse");

AddResponseDataToTree(response, claimsProcessingResponseOutputElement);
```

Note: The requestParameterFound flag is set to true if the PrepareRequest method completes successfully. The PrepareRequest method performs a substantial amount of tree navigation using the array syntax so that it can populate the request object. NullReferenceException, thrown within this method, is caught in the catch block for the Evaluate() method.

The AddResponseDataToTree needs an NBElement passed in as a parameter to which to attach the request data. This is created by deleting all children of the ApplicationMessage element and then using NBElement.CreateLastChild() to create the ClaimsProcessingResponse element.

After the response data is populated into the logical message tree, there are several fields that are common across all responses that are populated by the Evaluate() method. The

NBBroker class is used to obtain runtime information about the broker and the flow to fill in the LastModifiedBy field of the trailer.

Finally, the message is propagated to the Out terminal. Note that the propagate statement is moved inside the userCode #region block because all code paths from our catch blocks end up exiting using the throw.

5. Add the code shown in Example 7-29 to the UserCode region in the Evaluate() method.

Example 7-29 Evaluate() method part 5 in CallWCFNode.cs

```
NBElement innerResponseElement =
    claimsProcessingResponseOutputElement.FirstChild;
innerResponseElement.CreateFirstChild("CorrelId")
    .SetValue(response.CorrelationId);
innerResponseElement.CreateLastChild("ResponseTime")
    .SetValue(response.ResponseTime);

//Set the trailer information
outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
    ["LastModified"].SetValue(DateTime.Now);
outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
    ["LastModifiedBy"].SetValue("WebSphere Message Broker " +
    NBBroker.Broker.BrokerVersion + " - " +
    NBBroker.Broker.Name + "." + NBMessageFlow.MessageFlow.Name);

// Change the following if not propagating message to the 'Out' terminal
outTerminal.Propagate(outAssembly);
}
```

7.5.12 Handling failed tree navigations in the Evaluate() method

If any of the tree navigation fails in the Evaluate() method or any of the called methods, the result is a NullReference exception. The first catch block in the Evaluate() method checks the boolean flags that are set as execution proceeded through the Evaluate() method to convert the NRE in to a meaningful NBRecoverableException that explains that the content of the message did not match the expected structure:

1. Add the catch block shown in Example 7-30 to the UserCode region of the Evaluate() method.

Example 7-30 Evaluate() method part 6 in CallWCFNode.cs

```
catch (NullReferenceException)
{
    //Some NullReferenceException are generated due to a malformed input message,
    //test for cases where we want to raise an informative error for the user
    //throw other instances.
    if (!canonicalMessageMatched)
    {
        throw new NBRecoverableException("Input not recognized by canonical message
model. Root tag may be incorrect." +
        "Correct input and retry.");
    }
    else if (!recognisedApplicationMessage)
    {
        throw new NBRecoverableException("Input recognized as canonical " +
        "message but either the ApplicationMessage, " +
        "ClaimsProcessingRequest or RequestedDate fields were not " +
        "found. Message may not be intended for this" +

```

```

        "application, check routing and content and retry.");
    }
    else if (!requestParametersFound)
    {
        throw new NBRecoverableException("One or more parameters for this" +
            "request were missing. Correct input and retry");
    }
    //Any other NRE's are failure not due to navaigation of the tree
    throw;
}

```

7.5.13 Handling FaultExceptions in the Evaluate() method

The CallClaimsProcessing method throws FaultExceptions back to the calling code to handle. These FaultExceptions are caught in the Evaluate() method of the CallWCFNode. Recall from “Defining the Fault types” on page 208 that the Fault Contract of the WCF Service specifies that the calling code must be prepared to handle two types of Fault: ClaimsProcessingFaults and ViewOutstandingClaimsIncompleteResultsFault.

Because ViewOutstandingClaimsIncompleteResultsFault is derived from ClaimProcessingFault, it needs to be caught first to provide unique handling for this fault type. The catch block for ClaimsProcessingFaults is listed second to catch Faults that are instances of the base class. Recall that these exceptions are wrapped in a FaultException object:

1. Add the code shown in Example 7-31 to the UserCode region of the Evaluate() method to implement both catch blocks.

Example 7-31 Evaluate() method part 7 in CallWCFNode.cs

```

catch (FaultException<ViewOutstandingClaimsIncompleteResultsFault> e)
{
    //Some clients are happy with incomplete results so
    //copy the results to the environment tree
    //before rethrowing
    NBElement errorsElement =
inputAssembly.Environment.RootElement.CreateLastChild("Errors");
    NBElement claimsElement = errorsElement.CreateLastChild("OutstandingClaims");
    foreach (Claim claim in e.Detail.OutstandingResults)
    {
        AddClaimElementToParent(claim, claimsElement);
    }
    NBElement errorDetailsElement = errorsElement.CreateLastChild("ErrorList");
    foreach (ClaimProcessingFault fault in e.Detail.FaultList)
    {
        errorDetailsElement.CreateLastChild("Fault").SetValue(fault.ErrorCode + ": " +
fault.Message);
    }
    throw;
}
catch (FaultException<ClaimProcessingFault> e)
{
    throw new NBRecoverableException("CLAIMSPROCESSING FAULT. Code: " +
e.Detail.ErrorCode + " ErrorText: " + e.Detail.Message);
}

```

As discussed in 7.8, “Processing incomplete results for the ViewOutstandingClaims operation” on page 333, ViewOutstandingClaimsIncompleteResultsFaults are used to

construct a list of the incomplete claims in the Environment tree so that it can be accessed by upstream nodes in the flow.

ClaimsProcessingFaults are handled by extracting the ErrorCode and Message fields defined on the customer Fault type and then using these to construct a new NBRecoverableException that is thrown to be handled by the upstream flow.

7.5.14 Verifying the completed CallWCFNode

The Evaluate() method of the CallWCFNode is now complete, and the finished method is shown for reference in Example 7-32.

Example 7-32 The completed Evaluate() method in CallWCFNode.cs

```
public override void Evaluate(NBMessageAssembly inputAssembly)
{
    NBOutputTerminal outTerminal = OutputTerminal("Out");

    NBMessage inputMessage = inputAssembly.Message;

    // Create a new message from a copy of the inboundMessage, ensuring it is
    // disposed of after use
    using (NBMessage outputMessage = new NBMessage(inputMessage))
    {
        NBMessageAssembly outAssembly =
            new NBMessageAssembly(inputAssembly, outputMessage);
        NBElement inputRoot = inputMessage.RootElement;
        NBElement outputRoot = outputMessage.RootElement;

        #region UserCode

        //set these flags so that the error handler can tell where any tree
        navigations failed
        bool canonicalMessageMatched = false;
        bool recognisedApplicationMessage = false;
        bool requestParametersFound = false;
        try
        {
            NBElement canonicalMessageInputElement = inputRoot["XMLNSC"]
                [Constants.cp, "CanonicalMessage"];
            NBElement headerElement = canonicalMessageInputElement["Header"];

            canonicalMessageMatched = true;

            NBElement applicationMessageInputElement =
                canonicalMessageInputElement["ApplicationMessage"];
            NBElement requestElement = applicationMessageInputElement
                ["ClaimsProcessingRequest"].LastChild;

            //Create the appropriate type of request
            ClaimsProcessingRequest request;
            switch (requestElement.Name)
            {
                case "ApproveClaimRequest":
```

```

        request = new ApproveClaimRequest();
        request.TypeOfRequest = RequestType.ApproveClaim;
        break;
    case "CreatePolicyRequest":
        request = new CreatePolicyRequest();
        request.TypeOfRequest = RequestType.CreatePolicy;
        break;
    case "NewClaimRequest":
        request = new NewClaimRequest();
        request.TypeOfRequest = RequestType.NewClaim;
        break;
    case "RejectClaimRequest":
        request = new RejectClaimRequest();
        request.TypeOfRequest = RequestType.RejectClaim;
        break;
    case "ViewClaimRequest":
        request = new ViewClaimRequest();
        request.TypeOfRequest = RequestType.ViewClaim;
        break;
    case "ViewOutstandingClaimsRequest":
        request = new ViewOutstandingClaimsRequest();
        request.TypeOfRequest = RequestType.ViewOutstandingClaims;
        break;
    case "ViewPolicyRequest":
        request = new ViewPolicyRequest();
        request.TypeOfRequest = RequestType.ViewPolicy;
        break;
    default:
        throw new NBRecoverableException(
            "Unrecognised Request type " + requestElement.Name);
    }

    //this is a message type the flow knows about
    recognisedApplicationMessage = true;

    //All requests share some common properties
    request.ApplicationId = headerElement["AppId"].ValueAsString;
    request.MessageId = headerElement["MsgId"].ValueAsString;
    DateTime requestedDate = requestElement["RequestedDate"]
        .GetDateTime();
    request.RequestDate = requestedDate;

    //call overloaded method to fill in properties specific to each
    // request type
    PrepareRequest(request, requestElement);
    requestParametersFound = true;

    ClaimsProcessingResponse response =
        CallClaimsProcessingService(request);

    NBElem applicationMessageOutputElement = outputRoot["XMLNSC"]
[Constants.cp, "CanonicalMessage"] ["ApplicationMessage"];
    applicationMessageOutputElement.DeleteAllChildren();
    NBElem claimsProcessingResponseOutputElement =
    applicationMessageOutputElement

```

```

        .CreateLastChild("ClaimsProcessingResponse");

        AddResponseDataToTree(response,
            claimsProcessingResponseOutputElement);
        NBElement innerResponseElement =
            claimsProcessingResponseOutputElement.FirstChild();
        innerResponseElement.CreateFirstChild("CorrelId");
        .SetValue(response.CorrelationId);
        innerResponseElement.CreateLastChild("ResponseTime")
        .SetValue(response.ResponseTime);

        //Set the trailer information
        outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
["LastModified"].SetValue(DateTime.Now);
        outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
["LastModifiedBy"].SetValue("WebSphere Message Broker " +
            NBBroker.Broker.BrokerVersion + " - " + NBBroker.Broker.Name
+ "." + NBMessageFlow.MessageFlow.Name);

        // Change the following if not propagating message to the
// 'Out' terminal
        outTerminal.Propagate(outAssembly);
    }
    catch (NullReferenceException)
    {
        //Some NullReferenceException are generated due to a malformed
// input message,
        //test for cases where we want to raise an informative error
// for the user throw other instances.
        if (!canonicalMessageMatched)
        {
            throw new NBRecoverableException("Input not recognized by " +
                "canonical message model. Root tag may be " +
                "incorrect. Correct input and retry.");
        }
        else if (!recognisedApplicationMessage)
        {
            throw new NBRecoverableException("Input recognized as " +
                "canonical message but either the ApplicationMessage, "
+ "ClaimsProcessingRequest or RequestedDate fields " +
                "were not found. Message may not be intended for this"
+ "application, check routing/content and retry.");
        }
        else if (!requestParametersFound)
        {
            throw new NBRecoverableException("One or more parameters " +
                "for this request were missing. Correct input " +
                "and retry");
        }
        //Any other NRE's are failure not due to navigation of the tree
        throw;
    }
    catch (FaultException<ViewOutstandingClaimsIncompleteResultsFault> e)
    {
        //Some clients are happy with incomplete results so copy the results

```

```

// to the environment tree before rethrowing
NBElement errorsElement = inputAssembly.Environment.RootElement
    .CreateLastChild("Errors");
NBElement claimsElement = errorsElement
    .CreateLastChild("OutstandingClaims");
foreach (Claim claim in e.Detail.OutstandingResults)
{
    AddClaimElementToParent(claim, claimsElement);
}
NBElement errorDetailsElement = errorsElement
    .CreateLastChild("ErrorList");
foreach (ClaimProcessingFault fault in e.Detail.FaultList)
{
    errorDetailsElement.CreateLastChild("Fault")
        .SetValue(fault.ErrorCode + ": " + fault.Message);
}
throw;
}
catch (FaultException<ClaimProcessingFault> e)
{
    throw new NBRecoverableException("CLAIMSPROCESSING FAULT. Code: "
+ e.Detail.ErrorCode + " ErrorText: " + e.Detail.Message);
}

#endregion UserCode
}
}

```

The last step is to update the properties for the CallWCFService node to point to the assembly and class, which we complete later in 7.13.1, “Building a solution” on page 358.

This completes the implementation of the node itself. Now is a good opportunity to build the project using the **Build** → **Build Solution** menu item in Visual Studio. A successful build shows output like the example in Figure 7-35.

```

Output
Show output from: Build
----- Build started: Project: ClaimsProcessingBrokerProject, Configuration: Debug Any CPU -----
Build started 27/03/2012 14:13:30.
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date with respect to the input file
CoreCompile:
c:\WINDOWS\Microsoft.NET\Framework\v4.0.30319\Csc.exe /noconfig /nowarn:1701,1702 /nostdlib+ /errorreport:prompt /warn:4 /def
_CopyAppConfigFile:
Skipping target "_CopyAppConfigFile" because all output files are up-to-date with respect to the input files.
CopyFilesToOutputDirectory:
Copying file from "obj\Debug\ClaimsProcessingBrokerAssembly.dll" to "bin\Debug\ClaimsProcessingBrokerAssembly.dll".
ClaimsProcessingBrokerProject -> C:\MessageBrokerClaimsProcessor\WcfService\BrokerCallWcfNodeSolution\BrokerCallWcfNodeProjec
Copying file from "obj\Debug\ClaimsProcessingBrokerAssembly.pdb" to "bin\Debug\ClaimsProcessingBrokerAssembly.pdb".

Build succeeded.

Time Elapsed 00:00:05.98
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =====

```

Figure 7-35 Successful build output

7.6 Routing the output

Now we return to the Message Broker Toolkit to continue working on the message flow.

After the request to the WCF service is made, the output must be routed to execute conditional processing for some operations, and the response message must be returned to the client on the same transport that it was received on. For example, if the message was received from the SOAP input node, the reply must be made on the SOAPReply node. Similarly, if the message was received from the MQ Input node, the reply must be sent to the reply queue in the MQ Output node.

The Flow Order node is used to route the message through the branch of the flow that checks if any additional processing is required. The First terminal is wired to the Flow Order1 node, which is the root of this branch. The branch connected to the Second terminal handles sending the reply message, as seen in Figure 7-36.

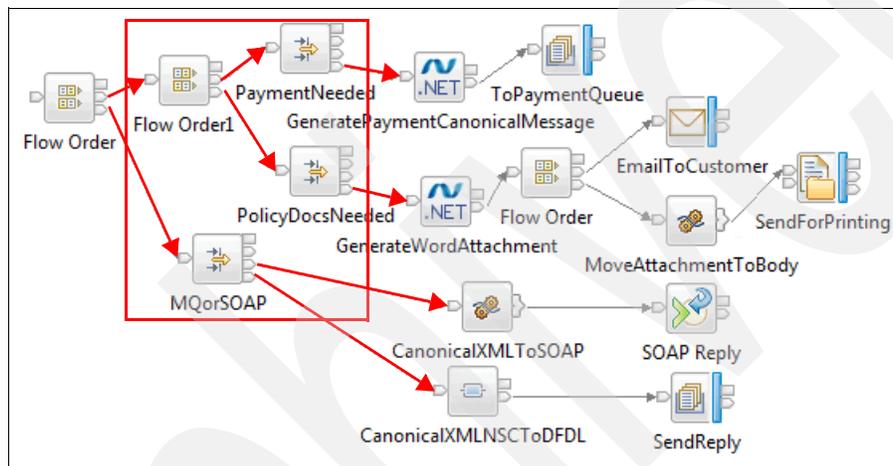


Figure 7-36 Flow order node routing messages

7.6.1 Additional processing

Recall from the scenario overview that there are two operations that require additional processing by the flow:

- ApproveClaim** A payment message is generated and sent to an MQ enterprise application for processing.
- Create Policy** A policy confirmation letter is emailed to the customer, and a file is sent to an external system for printing and eventually physically mailed to the customer as a hard copy.

This path in the message flow is shown in Figure 7-37 on page 327.

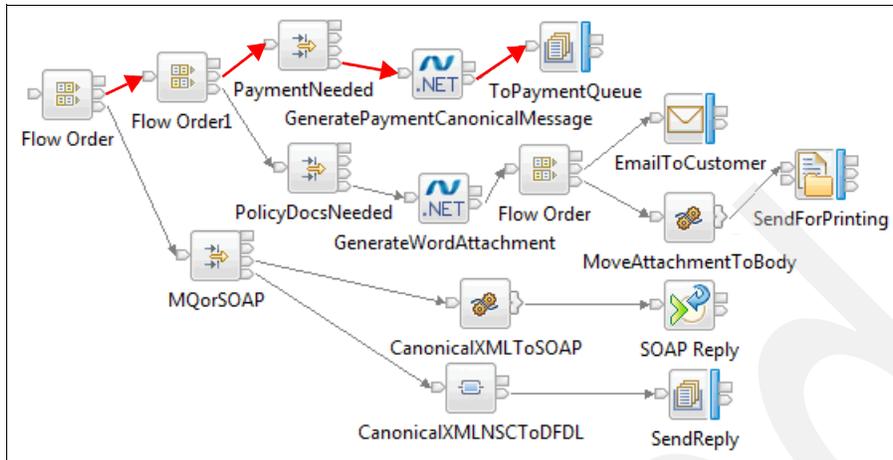


Figure 7-37 Additional processing flow

The Flow Order1 node propagates the message down a branch of the flow that implements this additional processing. Messages that do not require additional processing are filtered out of execution by the Filter Nodes PaymentNeeded and PolicyDocsNeeded:

1. Double-click the PaymentNeeded filter node.
2. Replace the generated code with the code in Example 7-33.

Example 7-33 Filter Expression for PaymentNeeded Filter node

```

DECLARE ns NAMESPACE
'http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/CanonicalMessage';

CREATE FILTER MODULE IsPaymentNeeded
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    DECLARE paymentRef REFERENCE TO
    Root.XMLNSC.ns:CanonicalMessage.ApplicationMessage.ClaimsProcessingResponse.ApproveClaim
    Response.Payment;
    IF LASTMOVE(paymentRef) THEN
      RETURN TRUE;
    END IF;
    RETURN FALSE;
  END;
END MODULE;

```

This filter expression declares a reference to a path in the message that only exists for ApproveClaimResponse messages. In particular, the reference is declared to the Payment element that is required to successfully complete the processing in the GeneratePaymentCanonicalMessage .NETCompute node. The LASTMOVE function is used to test if this element exists in the Logical Message Tree.

If the element exists, the expression returns true and execution proceeds to the GeneratePaymentCanonicalMessage node. If the expression returns false, ordinarily the message is propagated to the False terminal of the filter node. In this case, however, the False terminal is not wired, which means that execution on this branch of the flow completes. Execution will proceed from the Flow Order1 node's Second terminal.

Note that the namespace prefix ns is declared to the CanonicalMessage mode's namespace in a global scope. This namespace declaration can then be used by subsequent filter nodes and compute nodes developed in this chapter.

3. Close the ESQL. In the Basic tab of the Properties for the PaymentNeeded node, change the Filter expression field to IsPaymentNeeded.
4. The PolicyDocsNeeded Filter node performs a similar function; however, in this instance, rather than a Payment element, a reference is declared to the Policy element that is a child of the CreatePolicyResponse element, as shown in Figure 7-38.

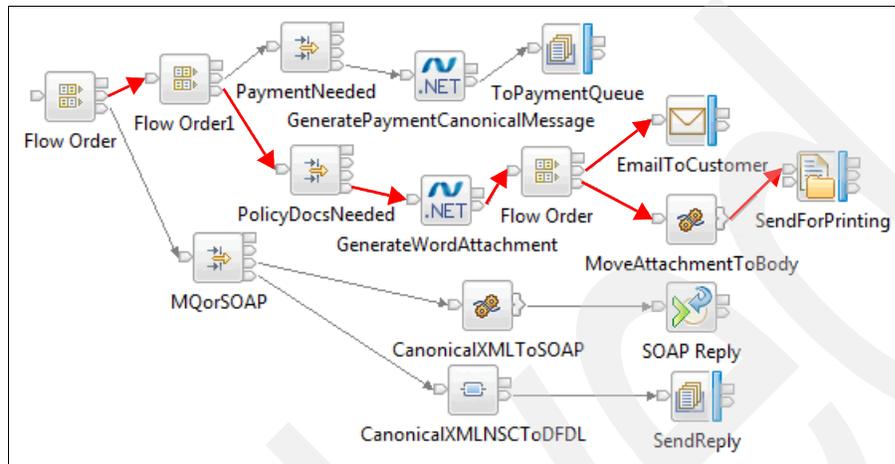


Figure 7-38 PolicyDocsNeeded Filter node determines if new policies need to be sent

5. Double-click the PolicyDocsNeeded node to open the filter expression.
6. Replace the generated code with the Filter expression in Example 7-34.

Example 7-34 Filter Expression for PolicyDocsNeeded Filter node

```
CREATE FILTER MODULE ArePolicyDocsNeeded
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    DECLARE policyRef REFERENCE TO
    Root.XMLNSC.ns:CanonicalMessage.ApplicationMessage.ClaimsProcessingResponse.CreatePolicy
    Response.Policy;
    IF LASTMOVE(policyRef) THEN
      RETURN TRUE;
    end IF;
    RETURN FALSE;
  END;
END MODULE;
```

7. Change the Filter expression field in the properties to ArePolicyDocsNeeded.

If the filter expression evaluates to true, execution proceeds to the GenerateWordAttachment .NETCompute node. Otherwise, because the False terminal is not wired, execution completes on this branch of the flow and resumes on the Second terminal of the Flow Order node.

7.6.2 Routing the reply message

The Second terminal of the Flow Order node is attached to a branch of the flow that handles the reply message. To ensure that the reply is made on the correct transport, the MQorSOAP Filter node is used to route the message to a branch that handles the transport specific output. The reply message branch flow is illustrated in Figure 7-39 on page 329.

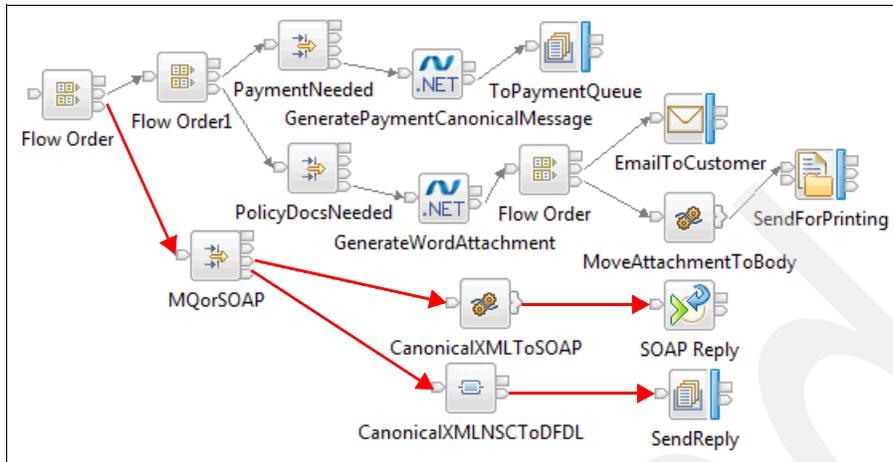


Figure 7-39 Reply message branch flow

The filter expression for the MQorSOAP node is shown in Example 7-35. To route the reply message:

1. Double-click the MQorSOAP node to open the filter expression.
2. Replace the generated code with the code in Example 7-35.

Example 7-35 The filter expression for the MQorSOAP Filter node

```
CREATE FILTER MODULE MQorSOAP
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    DECLARE soapEnvRef REFERENCE TO LocalEnvironment.SOAP.Envelope;
    IF LASTMOVE(soapEnvRef) THEN
        RETURN FALSE;
    END IF;
    RETURN TRUE;
END;
END MODULE;
```

3. Close the ESQL code.
4. Update the Filter expression in the properties to MQorSOAP.

Recall that when transforming the input message, the SOAPEExtract node is configured to place the SOAP Envelope in the LocalEnvironment under the SOAP/Envelope path, as shown in Figure 7-40.

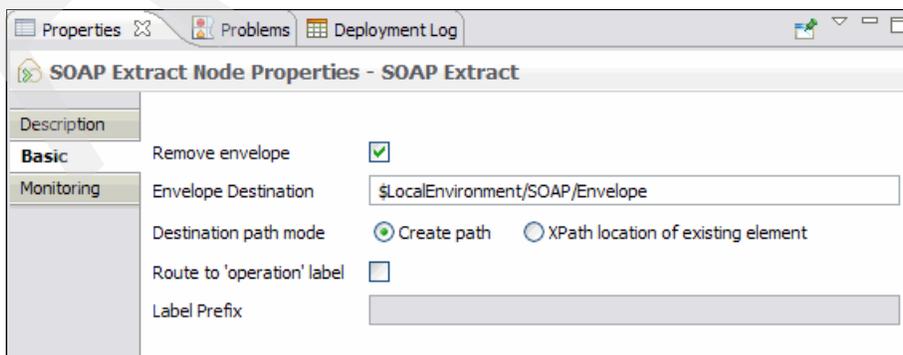


Figure 7-40 The Envelope Destination property of the SOAPEExtract node

The MQorSOAP filter expression declares a reference to the expected location of the SOAPEnvelope and uses the LASTMOVE function to determine if the element exists. If the SOAP Envelope exists in the LocalEnvironment tree, the expression returns False routing execution to the CanonicalXMLToSOAP node. Otherwise the input message is assumed to be an MQ message and execution is routed to the CanonicalXMLNSCToDFDL node.

7.7 Writing a .NETCompute node class to generate a payment message

When a Claim is approved, in addition to setting the state to Approved in the WCF service and sending a reply to the customer, the flow generates a message to send to a hypothetical payment processing system. This message causes a payment to be generated by the appropriate partner company that is underwriting the policy. In our example scenario, the application that processes the payments is a WebSphere MQ-based application that is part of the ESB.

Because the receiving application is part of the ESB, there is already a Canonical Message defined for sending payment defined in the ClaimsProcessingCanonicalMessageModel library.

Figure 7-41 shows the existing PaymentRequestType as a child of the ApplicationMessageType element.

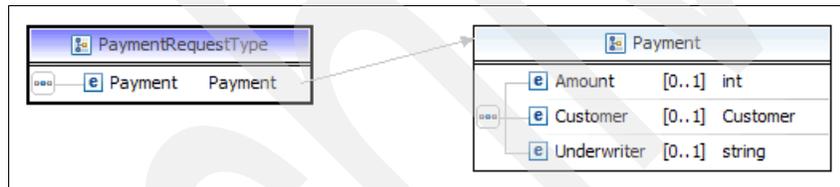


Figure 7-41 The PaymentRequest Type

As shown in Figure 7-42 on page 331, the PaymentRequestType itself is defined as a sequence of Payment elements.

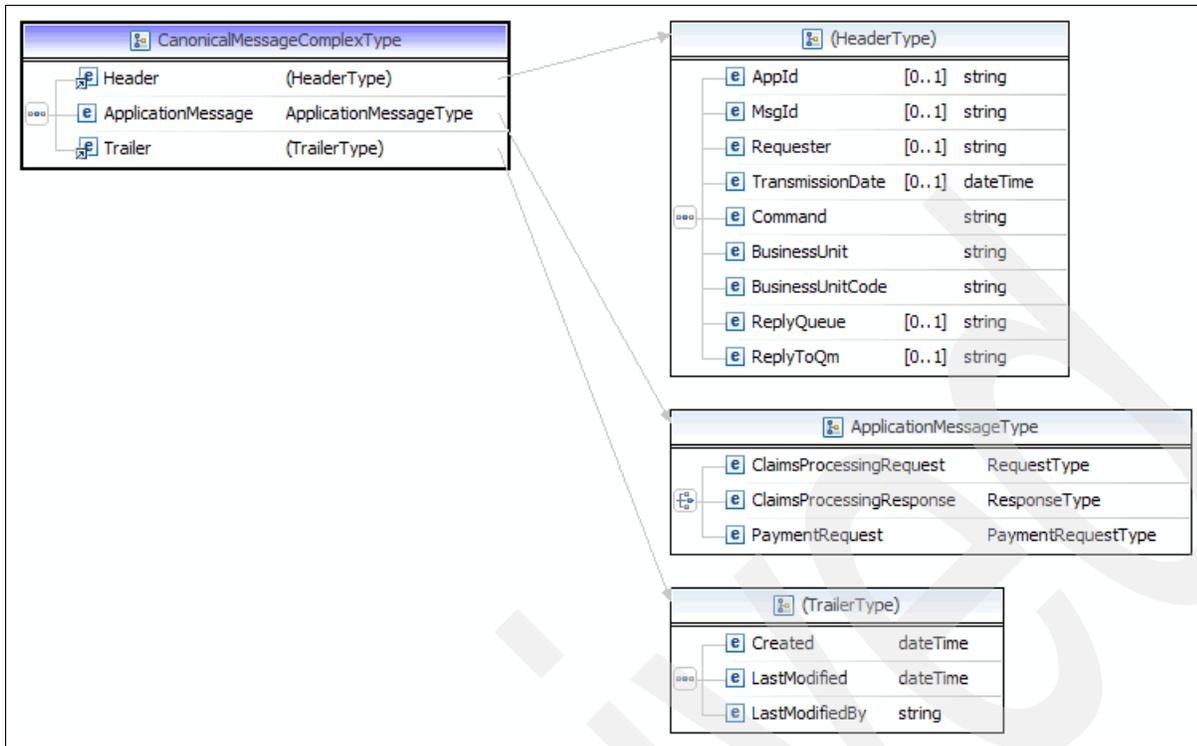


Figure 7-42 The Canonical Message Model already defines a message for PaymentRequests

The ApproveClaimResponse message returns a Payment element. The returned payment element can be used by a .NETCompute node to provide a simple transformation between the two message types, as shown in Figure 7-43.

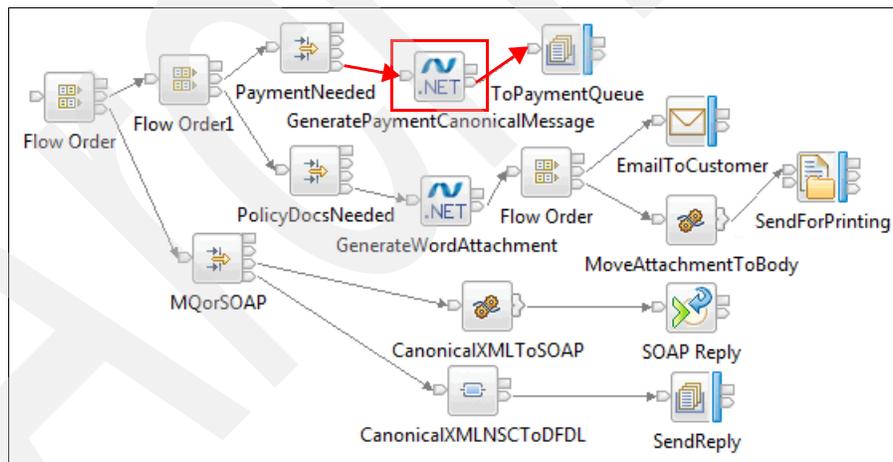


Figure 7-43 Transformation between two message types using a .NETCompute node

To implement the .NETCompute node, open ClaimsProcessingBrokerSolution in Visual Studio and use the following steps:

1. Select **Project** → **Add Class**.
2. From the Add New Item dialog, select **Message Broker** → **Class to Modify a Message Broker Message**.
3. Enter `GeneratePaymentNode.cs` in the Name field. Click **Add**.

4. Add the code shown in Example 7-36 to the UserCode region.

Example 7-36 GeneratePaymentNode.cs

```
//set these flags so that the error handler can tell where
//any tree navigations failed
bool canonicalMessageMatched = false;
bool foundPaymentElement = false;
try
{
    NBElement canonicalMessageInputElement =
        inputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"];
    canonicalMessageMatched = true;

    NBElement applicationMessageInputElement = canonicalMessageInputElement
        ["ApplicationMessage"];
    NBElement paymentInputElement = applicationMessageInputElement
        ["ClaimsProcessingResponse"]["ApproveClaimResponse"]["Payment"];
    foundPaymentElement = true;

    NBElement applicationMessageOutputElement = outputRoot["XMLNSC"]
        [Constants.cp, "CanonicalMessage"]["ApplicationMessage"];
    applicationMessageOutputElement.DeleteAllChildren();
    NBElement paymentRequestOutputElement = applicationMessageOutputElement
        .CreateFirstChild("PaymentRequest");
    paymentRequestOutputElement.AddLastChild(paymentInputElement);

    //Set the trailer information
    outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
        ["LastModified"].SetValue(DateTime.Now);
    outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
        ["LastModifiedBy"].SetValue("WebSphere Message Broker " +
            NBBroker.Broker.BrokerVersion + " - " +
            NBBroker.Broker.Name + "." +
            NBMessageFlow.MessageFlow.Name);
}
catch (NullReferenceException)
{
    //Check if the NRE is because the input tree was invalid
    if (!canonicalMessageMatched)
    {
        throw new NBRecoverableException("Input not recognized by canonical" +
            "message model. Root tag may be incorrect." +
            "Correct input and retry.");
    }
    else if (!foundPaymentElement)
    {
        throw new NBRecoverableException("Input recognized but could not" +
            "locate payment information. Correct input and retry.");
    }
    else
    {
        throw;
    }
}
}
```

5. Press Ctrl+S to save `GeneratePaymentNode.cs`.
6. The last step is to update the `GenericPaymentCanonicalMessage.NETCompute` node in the message flow to point to the .NET assembly file and class, which we will do later in 7.13.1, "Building a solution" on page 358.

Recall that when using the array syntax to reference the message tree, any failed navigation can result in a `NullReferenceException`. For this reason, boolean flags are set at the beginning of the method before navigating to the required portions of the input tree.

Any `NullReferenceExceptions` are caught and the flags tested to determine if the exception occurred during navigation on the input tree. If this is the case, a new `NBRecoverable` exception is thrown to give a more meaningful message to users of the flow.

The main code first copies the input message tree to the output tree using the code generated in Example 7-37.

Example 7-37 Copying the input tree in `GeneratePaymentNode.cs`

```
NBElement inputRoot = inputMessage.RootElement;
```

After copying the message tree, the `ApplicationMessage` element is cleared using a call to `DeleteAllChildren()` and a new child created named `PaymentRequest` and then the `Payment` element from the input message is added as a child to this new `Payment` message.

Finally, the trailer fields are updated to indicate that the message was modified by this node. The `NBBroker` class is used to extract runtime information, such as the name of the broker and the name of the message flow.

7.8 Processing incomplete results for the `ViewOutstandingClaims` operation

When the `ViewOutstandingClaims` operation encounters an error, it throws a custom `Fault` so that the consuming client can determine if it can continue with an incomplete set of results. This scenario is shown in Figure 7-44.

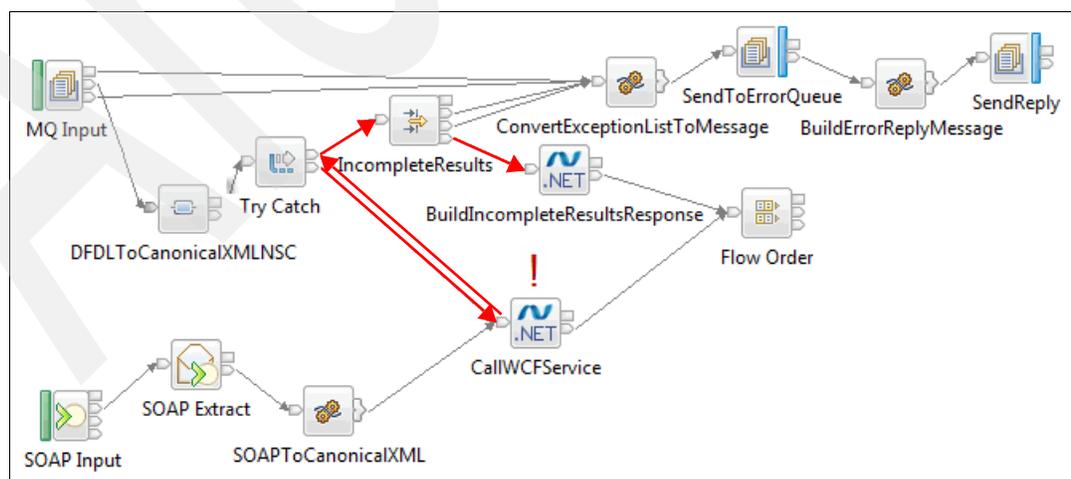


Figure 7-44 Consuming Client is determining if it can continue with incomplete results

When an incoming ViewOutstandingClaims request is mapped from the DFDL domain to the XMLNSC domain, a variable is placed in the LocalEnvironment under the Variables/VIEWOUTSTANDINGCLAIMSREQUEST/ACCEPTINCOMPLETE element, as shown in Figure 7-45.

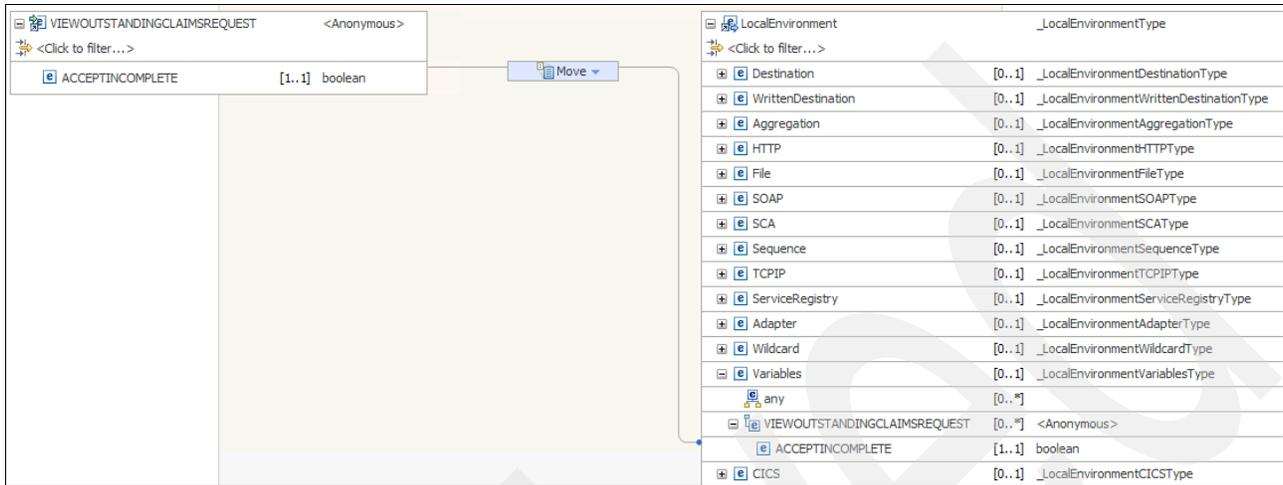


Figure 7-45 Mapping to the LocalEnvironment with the Graphical Mapper

Placing this value in the LocalEnvironment tree allows the flow to determine if the client indicated that it will accept incomplete results, and this value is tested for in the error handler.

Recall from 7.5, "Creating a .NETCompute node to consume the ClaimsProcessingWcfService" on page 298 that the service throws a custom fault of type ViewOutstandingClaimsIncompleteResultsFault. This exception type is handled separately in the CallWCFService .NETCompute node, as shown in Example 7-38.

Example 7-38 Catch block for FaultException<ViewOutstandingClaimsIncompleteResultsFault> in CallWCFNode.cs

```
catch (FaultException<ViewOutstandingClaimsIncompleteResultsFault> e)
{
    //Some clients are happy with incomplete
    //results so copy the results to the environment tree
    //before rethrowing
    NBElement errorsElement =
        inputAssembly.Environment.RootElement.CreateLastChild("Errors");
    NBElement claimsElement = errorsElement.CreateLastChild("OutstandingClaims");
    foreach (Claim claim in e.Detail.OutstandingResults)
    {
        AddClaimElementToParent(claim, claimsElement);
    }
    NBElement errorDetailsElement = errorsElement.CreateLastChild("ErrorList");
    foreach (ClaimProcessingFault fault in e.Detail.FaultList)
    {
        errorDetailsElement.CreateLastChild("Fault")
            .SetValue(fault.ErrorCode + ": " + fault.Message);
    }
    throw;
}
```

The handler catches `FaultExceptions` with a detail type of `ViewIncompleteResultsFault` and uses the `OutstandingResults` and `FaultList` collections defined on this custom type to place a copy of all the successfully retrieved `Claim` elements in the Environment tree under the `Errors/OutstandingClaims` path. Similarly, the faults that were experienced by the service are placed in the Environment tree under `Errors/ErrorMessageList` path.

After the details are added to the Environment tree, the exception is thrown again so that it can be handled upstream in the flow. Examining the flow structure, you can see that the `CallWCFService` node is preceded by a `TryCatch` node. This means that the `ExceptionList` is propagated down the `Catch` terminal to the `IncompleteResults` filter node, as shown in Figure 7-44 on page 333.

The expression for the `IncompleteResults` filter node evaluates to true if the input requested specified that incomplete results can be accepted. It does this by checking for the presence of the variable, placed in the `LocalEnvironment` by the DFDL mapping discussed at the beginning of this section, and the presence of the `Environment/Errors/OutstandingClaims` element that was placed in the in the Environment tree by the `CallWCFService` node. The filter expression is shown in Example 7-39.

1. Double-click the **IncompleteResults** node.
2. Replace the generated code for the filter with the code shown in Example 7-39.

Example 7-39 Filter expression for the IncompleteResults node

```
DECLARE ns1 NAMESPACE
'http://www.ibm.com/WebSphereMessageBroker/Examples/ClaimsProcessing/DFDLSchema';

CREATE FILTER MODULE IncompleteResults
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    DECLARE errorsRef REFERENCE TO Environment.Errors.OutstandingClaims;
    DECLARE acceptIncompleteRef REFERENCE TO
LocalEnvironment.Variables.ns1:VIEWOUTSTANDINGCLAIMSREQUEST.ACCEPTINCOMPLETE;
    IF LASTMOVE(errorsRef) AND LASTMOVE(acceptIncompleteRef) AND
FIELDVALUE(acceptIncompleteRef) = TRUE THEN
      RETURN TRUE;
    END IF;
    RETURN FALSE;
  END;
END MODULE;
```

3. Save and close the ESQL file.
4. Update the properties of the node so the Filter expression field contains `IncompleteResults`.

In this example, the existence of a tree element is tested by declaring a reference to the desired path and then using the `LASTMOVE` function to test if the parser can navigate to the path referred to by the reference successfully. Note that the `ACCEPTINCOMPLETE` variable was placed in the tree using the DFDL schema's namespace during the mapping, so the `ns1` prefix is defined to match the namespace in a global declaration.

If the filter expression evaluates to `False` or `Unknown`, the message is routed to the exception handler just as though the exception were caught by the input node. Otherwise, execution proceeds to the `BuildIncompleteResultResponse` `.NETCompute` node. The purpose of this node is to convert the outstanding claims list from the Environment tree into a `ViewOutstandingClaimsResponse` element in the Canonical Message XMLNSC model.

To create the .NET code for this node, switch to the ClaimsProcessingBrokerProject in Visual Studio, and follow these steps:

1. Select **Project** → **Add Class**.
 - a. From the Add New Item dialog, select **Message Broker** → **Class to Modify a Message Broker Message**.
 - b. Enter BuildIncompleteResultsResponseNode.cs in the Name field.
 - c. Click **Add**.
2. Add the code shown in Example 7-40 to the UserCode region.

Example 7-40 BuildIncompleteResultsResponseNode.cs

```
Boolean foundErrors = false;
Boolean foundCanonincalInputMessage = false;
try
{
    NBElement outstandingClaimsElement =
        inputAssembly.Environment.RootElement["Errors"]["OutstandingClaims"];
    foundErrors = true;

    NBElement canonicalMessageOutputElement = outputRoot["XMLNSC"]
        [Constants.cp, "CanonicalMessage"];
    NBElement applicationOutputElement =
        canonicalMessageOutputElement["ApplicationMessage"];
    foundCanonincalInputMessage = true;

    applicationOutputElement.DeleteAllChildren();
    NBElement viewOutstandingClaimsResponse = applicationOutputElement
        .CreateLastChild("ClaimsProcessingResponse")
        .CreateLastChild("ViewOutstandingClaimsResponse");
    viewOutstandingClaimsResponse.AddLastChild(outstandingClaimsElement);

    //Set the trailer information
    outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
        ["LastModified"].SetValue(DateTime.Now);
    outputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"]["Trailer"]
        ["LastModifiedBy"].SetValue("WebSphere Message Broker " +
        NBBroker.Broker.BrokerVersion + " - " +
        NBBroker.Broker.Name + "." +
        NBMessageFlow.MessageFlow.Name);
}
catch (NullReferenceException)
{
    if (!foundErrors)
    {
        throw new NBRecoverableException("No incomplete results were found" +
            "to process");
    }
    else if (!foundCanonincalInputMessage)
    {
        throw new NBRecoverableException("Input not recognized by canonical" +
            "message model. Root tag may be incorrect." +
            "Correct input and retry.");
    }
}
throw;
```

```
}
```

3. Press Ctrl+S to save BuildIncompleteResultsResponse.cs.

Failed tree navigations can result in `NullReferenceExceptions` being thrown from expressions, such as `inputAssembly.Environment.RootElement["Errors"]["OutstandingClaims"]`. If, for example, the `EnvironmentRoot/Errors` element did not exist, that is the equivalent of calling `null["OutstandingClaims"]`, which results in an exception. For this reason, `NullReferenceExceptions` are caught and some boolean flags tested to see if the NRE was the result of a failed tree navigation. If this is the case, an `NBRecoverable` exception is thrown instead with a more meaningful error message; otherwise, the NRE is thrown again with the `throw; statement`.

Correct syntax for throwing exceptions: Unlike in Java, it is not correct to throw the exception by name, for example, using syntax, such as `catch(Exception e) { throw e; }`. Using this syntax results in the stack trace for the exception being overridden and the `throw` statement in the catch block being the new origin point of the exception stack. This is because `throw e;` gets compiled into the MSIL `throw` instruction. Instead use the syntax `catch(Exception) { throw; }`, which compiles into the MSIL `re-throw` instruction that gives the desired behavior.

The main code path of the node deletes all children of the `CanonicalMessage/ApplicationMessage` element and creates a new child named `ClaimsProcessingResponse`. A new child of this `ClaimsProcessingResponse` element is created called `ViewOutstandingRequests`, and then the `OutstandingClaims` element from the `Environment` tree is added as a child to the `ViewOutstandingRequests` element. This structure was created by the `CallWCFService` node in the catch block for `FaultException<ViewOutstandingClaimsIncompleteResults>` exceptions, as shown in Example 7-38 on page 334.

Finally, the trailer is updated to indicate that the message flow modified the message. The `NBBroker` class is again used to access runtime information about the broker and the message flow.

If execution of the compute node completes successfully, the message is propagated back into the main line of the flow to the `Flow Order` node exactly as though the `CallWCFClaimsService` had completed successfully.

7.9 Creating a Word document for email and print

When the `CreatePolicy` operation is invoked, the flow needs to generate a Word document to send to the customer that contains the details of their newly purchased policy. This can be done using a `.NETCompute` node, as shown in Figure 7-46 on page 338.

The C# code in this chapter takes data from the broker logical tree and uses it to create a Microsoft Word document. The code uses the `Open XML Format SDK (Version 2)`, which is an API for the .NET Framework that enables developers in both client and server environments to interact with Microsoft Word.

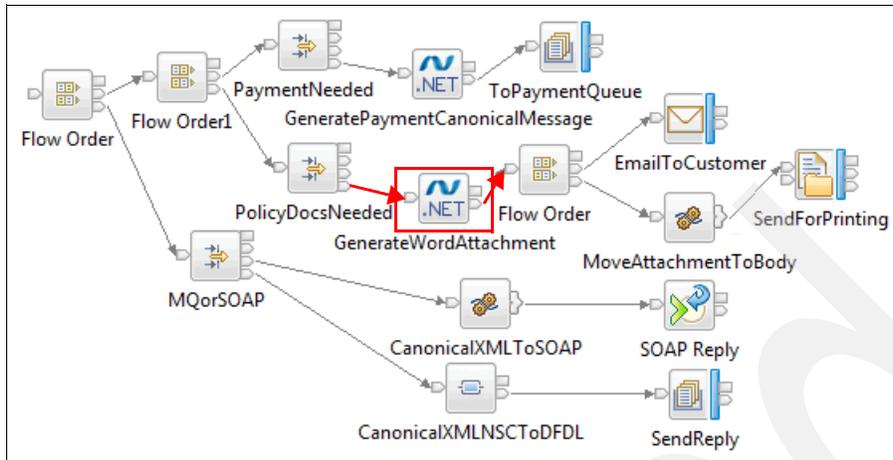


Figure 7-46 A customer document detailing policies is created in the .NET node

7.9.1 Preparing a letter template

Before implementing the .NETCompute node, prepare a template for the policy confirmation letter.

To create the template:

1. Open a new Word document with Microsoft Word 2010.
2. In the document, put your cursor where you want to add the content control, or select the text that you want the content control to replace.
3. Click the **Developer** tab.

If the **Developer** tab is not visible, you must first show the Developer tab for Microsoft Office 2010 applications:

- a. Click the **File** tab.
- b. Click **Options**.
- c. In the categories pane, click **Customize Ribbon**.
- d. In the list of main tabs, select **Developer**.
- e. Click **OK** to close the Options dialog box.

4. Click **Design Mode** in the Controls section, as shown in Figure 7-47.

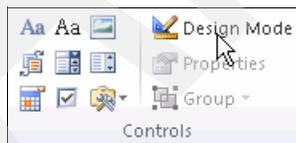


Figure 7-47 Switch to design mode

5. In the Controls section, click **Plain Text Content Control**, as shown in Figure 7-48 on page 339.

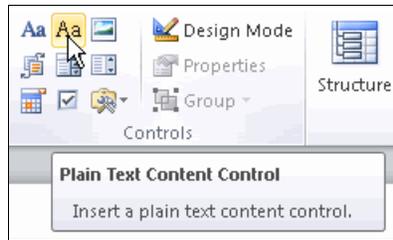


Figure 7-48 Plain Text Content Control selection

6. In the new content control field, type `Recipient`, as shown in Figure 7-49.



Figure 7-49 New content control field

7. Right-click the **content control** and select **Properties**, as shown in Figure 7-50.

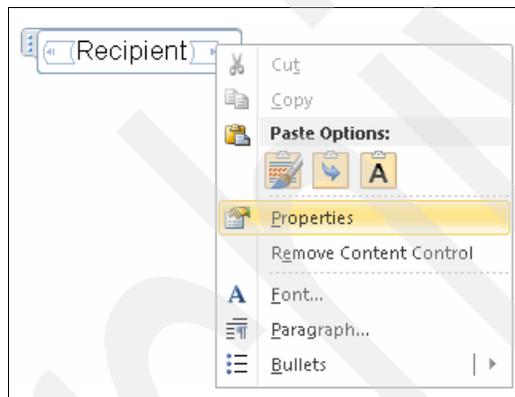


Figure 7-50 Select properties when preparing a letter template

8. Enter `Name` as the tag, as shown in Figure 7-51 on page 340. Click **OK**.
 Later, in “Creating a helper method” on page 345, you will code a .NET application to populate this template using these tags to identify the fields.

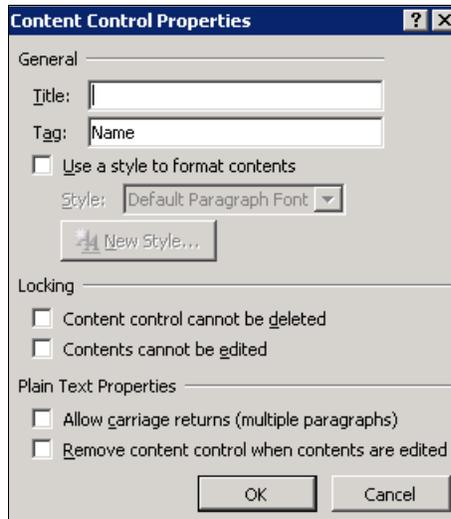


Figure 7-51 Properties for the content control field

9. Continue building the template using Figure 7-52 as a guide. The form is a mix of text and content control fields.

Figure 7-52 Content control example

10. Save and close the template.

7.9.2 Creating a user-defined property for the template location

To create a user-defined property for the node:

1. Click the **User Defined Properties** tab on the bottom of the message flow editor.
2. Right-click **ClaimProcessFlow** → **Basic** → **Add Property**:
 - a. Rename the property `TemplateLocation`.
 - b. In the Default value field, enter the location of the template.

Figure 7-53 on page 341 shows the new property settings.

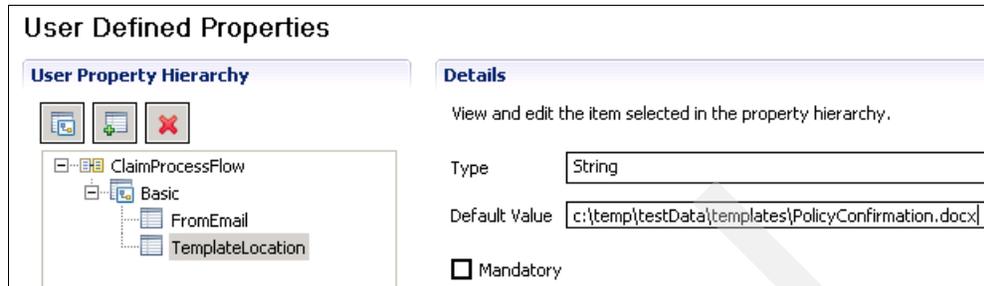


Figure 7-53 TemplateLocation user defined property

3. Press Ctrl+S to save your changes.
4. Switch back to the Graph view.

7.9.3 Creating .NET code to generate a Word document using a template

This process creates the .NET code for the GenerateWordAttachment .NETCompute node. With the ClaimsProcessingNodeProject open in Visual Studio, follow these steps:

1. Select **Project** → **Add Class**.
2. From the Add New Item dialog, select **Message Broker** → **Class to Modify a Message Broker Message**.
3. Enter BuildEmailAttachmentNode.cs in the Name field, and click **Add**.

Adding references to the project

Because you create and update the Word document with the Open XML SDK 2.0 API, you need to add it into reference. To add the Word document into reference:

1. In the hierarchy in Solution Explorer, right-click **References** → **Add Reference**.
2. In the Add Reference dialog, switch to the .NET tab, and select **DocumentFormat.OpenXml**.

If you cannot find the entry, first make sure that you sorted the components alphabetically. If the entry is definitely not in the list, follow the previous instructions to download and install the Open XML SDK 2.0 API.

3. Click **OK**.
4. Using the same process, add a reference to the WindowsBase reference.

Overriding the Evaluate method

The Evaluate() method builds a Microsoft Word document that contains the new policy document that is sent to the customer:

1. Replace the code for the Evaluate() method in BuildEmailAttachmentNode.cs with the code in Example 7-41.

Example 7-41 Add the Evaluate method to the project

```
public override void Evaluate(NBMessageAssembly inputAssembly)
{
    NBOutputTerminal outTerminal = OutputTerminal("Out");
    NBMessage inputMessage = inputAssembly.Message;

    // Create a new empty message, ensuring it is disposed after use
```

```

using (NBMessage outputMessage = new NBMessage())
{
    NBElement inputRoot = inputMessage.RootElement;
    NBElement outputRoot = outputMessage.RootElement;

    // Optionally copy message headers, remove if not needed
    CopyMessageHeaders(inputRoot, outputRoot);

    #region UserCode
    // Add user code in this region to modify the message
    // create attachment properties folders in the local environment
    NBMessage localEnvironment = inputAssembly.LocalEnvironment;
    NBMessage newEnvironment = new NBMessage(inputAssembly.LocalEnvironment);

    // add environment tree to output message assembly
    NBMessageAssembly outAssembly = new NBMessageAssembly(inputAssembly, outputMessage,
newEnvironment);

    // check response type
    NBElement canonicalMessageElement = inputRoot["XMLNSC"][Constants.cp, "CanonicalMessage"];
    NBElement claimsProcessingResponseElement =
canonicalMessageElement["ApplicationMessage"]["ClaimsProcessingResponse"];
    NBElement policyElement =
claimsProcessingResponseElement["CreatePolicyResponse"]["Policy"];
    NBElement emailOutputHeader = outputRoot.CreateLastChild("EmailOutputHeader");

    // read template into memory stream
    byte[] byteArray = File.ReadAllBytes(GetUserDefinedPropertyAsString("TemplateLocation"));
    using (MemoryStream mem = new MemoryStream())
    {
        mem.Write(byteArray, 0, (int)byteArray.Length);
        CreatePolicyWordDocumentMemoryStream(policyElement, mem);

    //add mem stream to output bitstream
    Byte[] myBytes = new Byte[(int)mem.Length];
    myBytes = mem.ToArray();
    outputRoot.CreateLastChildUsingNewParser(NBParsers.BLOB.ParserName);
    NBElement blob = outputRoot[NBParsers.BLOB.ParserName].CreateLastChild("BLOB");
    string samplemsg = "This is the confirmation letter of your policy";
    Byte[] emailbody = System.Text.Encoding.ASCII.GetBytes(samplemsg);
    blob.SetValue(emailbody);

    // create an email properties folders

    emailOutputHeader.CreateLastChild("To").SetValue(policyElement["PolicyHolder"]["Email"].ValueAsS
tring);

    emailOutputHeader.CreateLastChild("From").SetValue(GetUserDefinedPropertyAsString("FromEmail"));
        emailOutputHeader.CreateLastChild("Subject").SetValue("Policy Confirmation
from ITSO Insurance");

    //setting unique file name
    string filename = "PolicyConfirmation" + DateTime.Today.ToLongDateString() + ".docx";

```

```

//Set attachment details into the local env tree
NBElement destination = newEnvironment.RootElement.CreateLastChild("Destination");
NBElement attachmentElement =
destination.CreateLastChild("Email").CreateLastChild("Attachment");

attachmentElement.CreateLastChild("ContentName").SetValue(filename);
attachmentElement.CreateLastChild("ContentEncoding").SetValue("Base64");
attachmentElement.CreateLastChild("Content").SetValue(myBytes);

//propagate
outTerminal.Propagate(outAssembly);
}
#endregion UserCode
}
}

```

2. Add the following Using statements for the Open XML API classes so that we can code without using a fully-qualified name. See Example 7-42.

Example 7-42 Using directives

```

using DocumentFormat.OpenXml;
using DocumentFormat.OpenXml.Packaging;
using DocumentFormat.OpenXml.Wordprocessing;

```

3. Instead of opening a file from the disk or a file share, we work with a document in memory. Retrieve a document from the document library as a byte array, and then modify it as necessary. Put it back into the document library as a new document. When engaged, as shown in Example 7-43, this aspect opens to use System.IO.

Add the following using statement:

```
using System.IO;
```

The Template class does not access the LocalEnvironment Message. Looking at code for the Evaluate() method in Example 7-43, you can see that a new message assembly was included and other areas were commented out.

Example 7-43 New message assembly

```

// NBMessageAssembly outAssembly = new NBMessageAssembly(inputAssembly,
outputMessage);

#region UserCode
// Add user code in this region to modify the message

NBMessage localEnvironment = inputAssembly.LocalEnvironment;
NBMessage newEnvironment = new NBMessage(inputAssembly.LocalEnvironment);
NBMessageAssembly outAssembly = new NBMessageAssembly(inputAssembly,
outputMessage, newEnvironment);

```

In Example 7-44 on page 344, we created a message element reference that points to the response message payload. We also created emailOutputHeader that will hold the detail about the email.

Example 7-44 Message element reference and emailOutputHeader

```
NBElement canonicalMessageElement = inputRoot["XMLNSC"][Constants.cp,
"CanonicalMessage"];
NBElement claimsProcessingResponseElement =
canonicalMessageElement["ApplicationMessage"]["ClaimsProcessingResponse"];
NBElement policyElement =
claimsProcessingResponseElement["CreatePolicyResponse"]["Policy"];

NBElement emailOutputHeader = outputRoot.CreateLastChild("EmailOutputHeader");
```

We read the name and path to the letter template from user-defined properties, and we read it into In-Memory stream, as shown in Example 7-45.

Example 7-45 Name and path of the letter template

```
byte[] byteArray =
File.ReadAllBytes(GetUserDefinedPropertyAsString("TemplateLocation"));

using (MemoryStream mem = new MemoryStream())
{
    mem.Write(byteArray, 0, (int)byteArray.Length);
}
```

The following example calls our own method that updates the template based on the information that is available in the response message. We will create this helper method later.

```
CreatePolicyWordDocumentMemoryStream(policyElement, mem);
```

After updating the template, we copy it over to bitstream, as shown in Example 7-46.

Example 7-46 Copy the template to bitstream

```
Byte[] myBytes = new Byte[(int)mem.Length];
myBytes = mem.ToArray();
```

We then prepare the message that will be included in the email body, and put it in the message tree, as shown in Example 7-47.

Example 7-47 Create the body of the message

```
outputRoot.CreateLastChildUsingNewParser(NBParsers.BLOB.ParserName);
NBElement blob = outputRoot[NBParsers.BLOB.ParserName].CreateLastChild("BLOB");
string samplemsg = "This is the confirmation letter of your policy";
Byte[] emailbody = System.Text.Encoding.ASCII.GetBytes(samplemsg);

blob.SetValue(emailbody);
```

In the last section of code for the Evaluate method, we define the details about the email header and attachment and then propagate the results to the output terminal (Example 7-48).

Example 7-48 Define details for the message header and attachment

```
emailOutputHeader.CreateLastChild("To").SetValue(policyElement["PolicyHolder"]["Em
ail"].ValueAsString);

emailOutputHeader.CreateLastChild("From").SetValue(GetUserDefinedPropertyAsString(
"FromEmail"));
```

```

emailOutputHeader.CreateLastChild("Subject").SetValue("PolicyConfirmation from
ITSO Insurance");

string filename = "PolicyConfirmation" + DateTime.Today.ToLongDateString() +
".docx";

NBElement destination = newEnvironment.RootElement.CreateLastChild("Destination");
NBElement attachmentElement =
destination.CreateLastChild("Email").CreateLastChild("Attachment");

attachmentElement.CreateLastChild("ContentName").SetValue(filename);
attachmentElement.CreateLastChild("ContentEncoding").SetValue("Base64");
attachmentElement.CreateLastChild("Content").SetValue(myBytes);

outTerminal.Propagate(outAssembly);
}
#endregion UserCode

```

Creating a helper method

To create a helper method:

1. Add the code shown in Example 7-49 after the Evaluate method, in BuildEmailAttachmentNode.cs.

Example 7-49 Create a helper method

```

private static void CreatePolicyWordDocumentMemoryStream(NBElement policyElement, MemoryStream
mem)
{
using (WordprocessingDocument wordDoc = WordprocessingDocument.Open(mem, true))
{
MainDocumentPart DocPart = wordDoc.MainDocumentPart;
List<SdtBlock> sdtList = DocPart.Document.Descendants<SdtBlock>().ToList();
foreach (SdtBlock sdt in sdtList)
{
string aliasname = sdt.SdtProperties.GetFirstChild<Tag>().Val;
SdtContentBlock contentblock = sdt.Descendants<SdtContentBlock>().FirstOrDefault();
Text text = contentblock.Descendants<Text>().FirstOrDefault();

//update content block
switch (aliasname)
{
case "name":
//add customer name on header
text.Text = policyElement["PolicyHolder"]["Surname"].ValueAsString
+ ", " + policyElement["PolicyHolder"]["FirstName"].ValueAsString;
break;
case "address":
// add address
text.Text = policyElement["PolicyHolder"]["Address"].ValueAsString;
break;
case "policynumber":
// add policy number
text.Text = policyElement["PolicyNumber"].ValueAsString;
break;
case "policystart":

```

```

// add policy starting date
text.Text = policyElement["PolicyStartDate"].GetDateTime().ToLongDateString();
break;
case "policyexpire":
// add policy expiring date
text.Text = policyElement["PolicyExpiryDate"].GetDateTime().ToLongDateString();
break;
case "date":
// add today's date
text.Text = DateTime.Today.ToLongDateString();
break;
case "dear":
// add customer name in body
text.Text = "Dear " + policyElement["PolicyHolder"]["Surname"].ValueAsString
    + ", " + policyElement["PolicyHolder"]["FirstName"].ValueAsString;
break;
}
}
//complete the update
wordDoc.Close();
}
}

```

2. Save the code.

Looking at the code, you can see that it opens the letter template stored in In-Memory stream using Open XML API (Example 7-50).

Example 7-50 Open the letter template

```

using (WordprocessingDocument wordDoc = WordprocessingDocument.Open(mem, true))
{
    MainDocumentPart DocPart = wordDoc.MainDocumentPart;

```

Next, it locates all content controls in the template (Example 7-51).

Example 7-51 All content controls

```

List<SdtBlock> sdtList = DocPart.Document.Descendants<SdtBlock>().ToList();

```

It then loops through the list of content controls and updates the template based on the information about the response message (Example 7-52).

Example 7-52 Update the template

```

foreach (SdtBlock sdt in sdtList)
{
    string aliasname = sdt.SdtProperties.GetFirstChild<Tag>().Val;
    SdtContentBlock contentblock =
    sdt.Descendants<SdtContentBlock>().FirstOrDefault();
    Text text = contentblock.Descendants<Text>().FirstOrDefault();

    switch (aliasname)
    {
        case "name":
            text.Text = policyElement["PolicyHolder"]["Surname"].ValueAsString

```

```

        + ", " + policyElement["PolicyHolder"]["FirstName"].ValueAsString;
    break;
    case "address":
        text.Text = policyElement["PolicyHolder"]["Address"].ValueAsString;
        break;
    case "policynumber":
        text.Text = policyElement["PolicyNumber"].ValueAsString;
        break;
    case "policystart":
        text.Text =
policyElement["PolicyStartDate"].GetDateTime().ToLongDateString();
        break;
    case "policyexpire":
        text.Text =
policyElement["PolicyExpiryDate"].GetDateTime().ToLongDateString();
        break;
    case "date":
        text.Text = DateTime.Today.ToLongDateString();
        break;
    case "dear":
        text.Text = "Dear " +
policyElement["PolicyHolder"]["Surname"].ValueAsString
        + ", " + policyElement["PolicyHolder"]["FirstName"].ValueAsString;
        break;
    }
}

```

The code then saves the update and completes the method definition (Example 7-53).

Example 7-53 Update the method definition

```
wordDoc.Close();
```

7.9.4 Using the EmailOutput node to send an email to the customer

When a new policy is created, the message is routed to the BuildEmailAttachment node. It creates a Microsoft Word document using the template that contains the confirmation and sends it to the customer as an email attachment, as shown in Figure 7-55 on page 348.

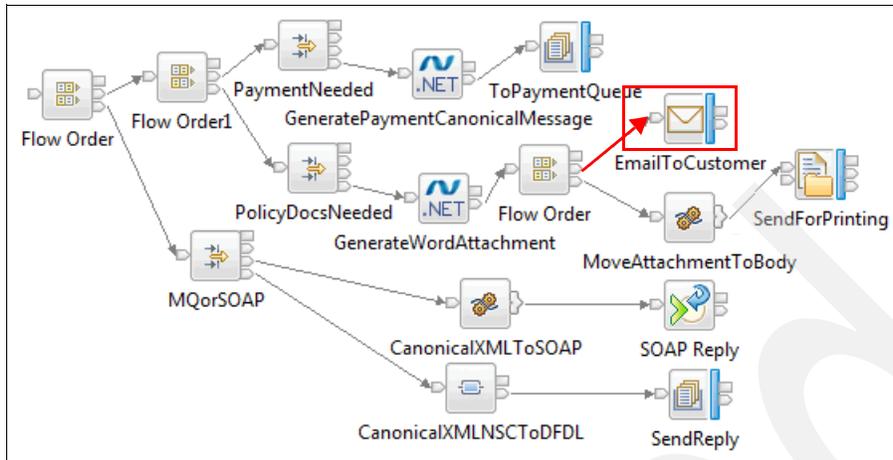


Figure 7-54 Flow order showing the point at that an email is sent to the customer

The `GenerateWordAttachment` `.NETCompute` node produces a bitstream that contains the customized word document and places it in the `LocalEnvironment` tree under the `Destination/Email/Attachment/Content` path. The `EmailToCustomer` email node automatically converts this content into an attachment when it sends the output message.

7.9.5 Using the FileOutput node to send the file for printing

After the new policy confirmation document is emailed to the customer, it also needs to be sent to a printer so that a hard copy can be mailed to the customer's address. The printing service requires the file to be placed in the file system using the `FileOutput` node, illustrated in Figure 7-55.

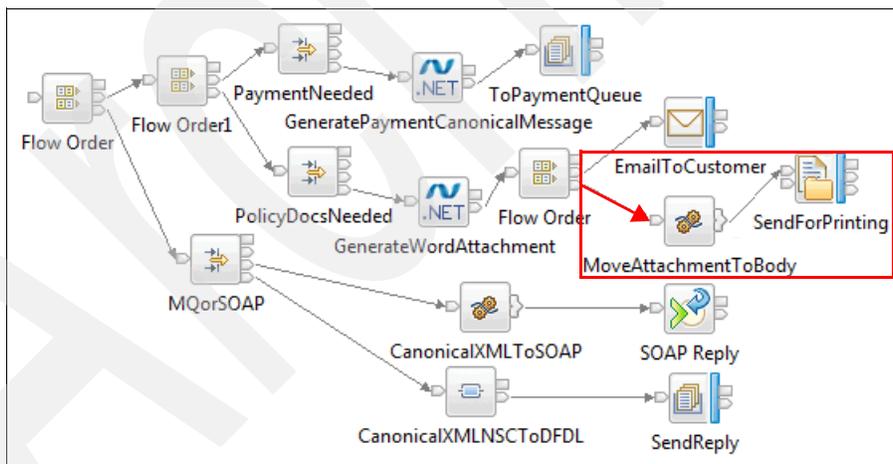


Figure 7-55 Sending the file to a printer

The attachment is currently located in the message tree in the `LocalEnvironment` so that the `EmailOutput` node can construct an email message. The `FileOutput` node requires the document to be in the message body, so we need to perform some simple transformation in an `ESQL` `Compute` node to move the element from the `LocalEnvironment` tree to the `Body`.

1. Double-click the **MoveAttachmentToBody** compute node to open the `ESQL`.
2. Change the name of the module to `MoveAttachmentToBody`.

3. Replace the BEGIN/END block in the Main() function with the content in Example 7-54 on page 349.

Example 7-54 Transform the ESQL Compute node

```

BEGIN
  CALL CopyMessageHeaders();
  SET OutputRoot.BLOB.BLOB =
InputLocalEnvironment.Destination.Email.Attachment.Content;
  RETURN TRUE;

END;

```

4. Close the ESQL.
5. In the Basic tab of the Properties view for the node, change the ESQL module to MoveAttachmentToBody.

This code moves the bitstream attached on the LocalEnvironment tree to the message body so that FileOutputNode can generate the same documentation attached on the email. The attachment itself is binary data, so we used the BLOB parser to ensure that no conversion happens on serialization by the FileOutput node. FileOutput node's Data location property is configured to extract the data from Message Body, as described in "SendForPrinting node" on page 287.

7.10 Using the Mapping node to transform the Canonical message to the output format

The flow must ensure that requests that originated from an MQ client receive a reply on the MQ transport. Requests that arrived on the MQInput node are routed to the branch of the flow that converts the XMLNSC Canonical message into the binary format used by the MQ clients before sending the reply. The binary format is modelled in the DFDL domain so a Graphical Mapping node can be used to perform the transformation, as show in Figure 7-56.

The CanonicalXMLNSCToDFDL node was configured in "CanonicalXMLNSCToDFDL node" on page 289. This section takes a closer look at this mapping function.

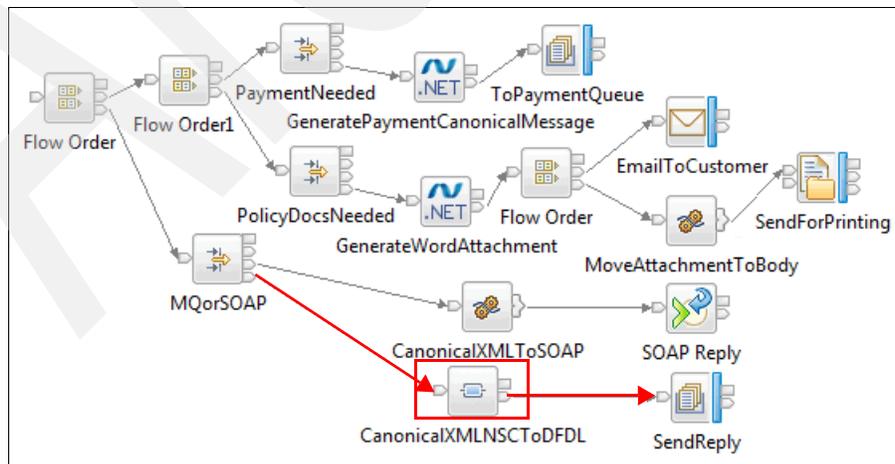


Figure 7-56 Mapping a Canonical Response message into a DFDL domain message

Additional material: The building of the map is beyond the scope of this chapter; however, you can explore the completed mapping in the CanonicalXMLNSCToDFDL.map files by downloading them from the additional materials and importing them to the Message Broker Toolkit. You can access the mapping editor by either double-clicking the map file in the Explorer pane or double-clicking the node itself.

Figure 7-59 shows the top level mapping.

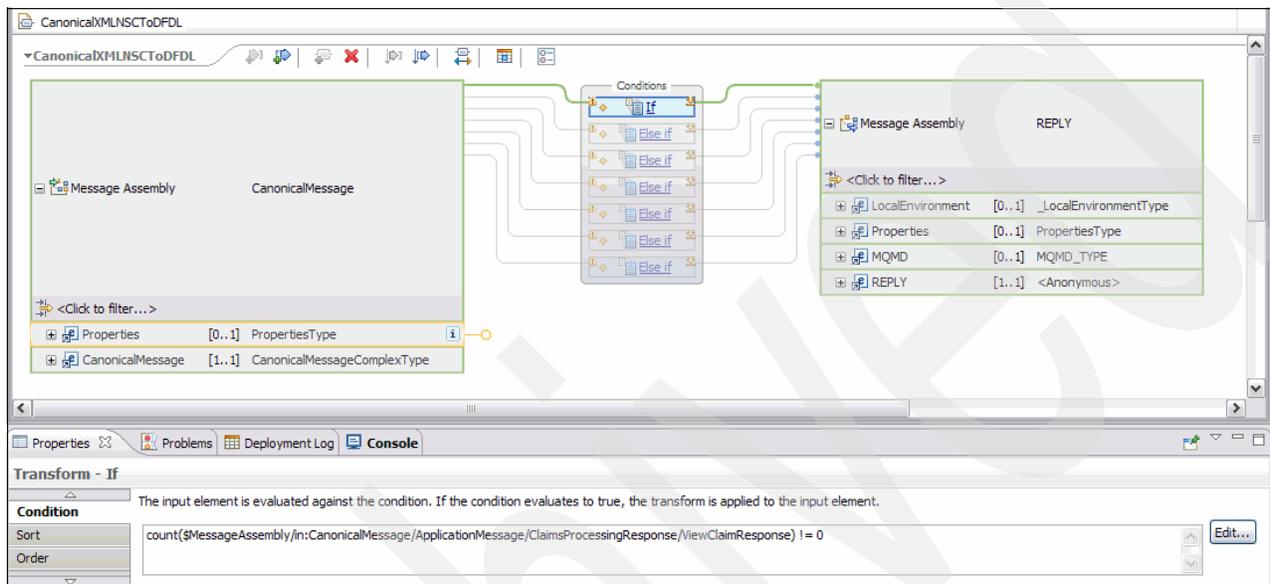


Figure 7-59 The top level map in CanonicalXMLNSCToDFDL.map

The map uses the Conditional mapping construct to run a specific submap, depending on the type of response that was received. In a Conditional mapping, it is possible to add individual conditions that are only executed when the XPath expression in the Condition property is satisfied by the input message. In this example, the XPath count() function is used to determine if there is a ViewClaimResponse element in the tree.

Figure 7-60 on page 352 highlights an inner section of the local map that maps an XMLNSC Claim element onto a DFDL CLAIMLIST element using Move operations. Each Move operation was created by simply dragging the source field on the left onto the target field on the right.

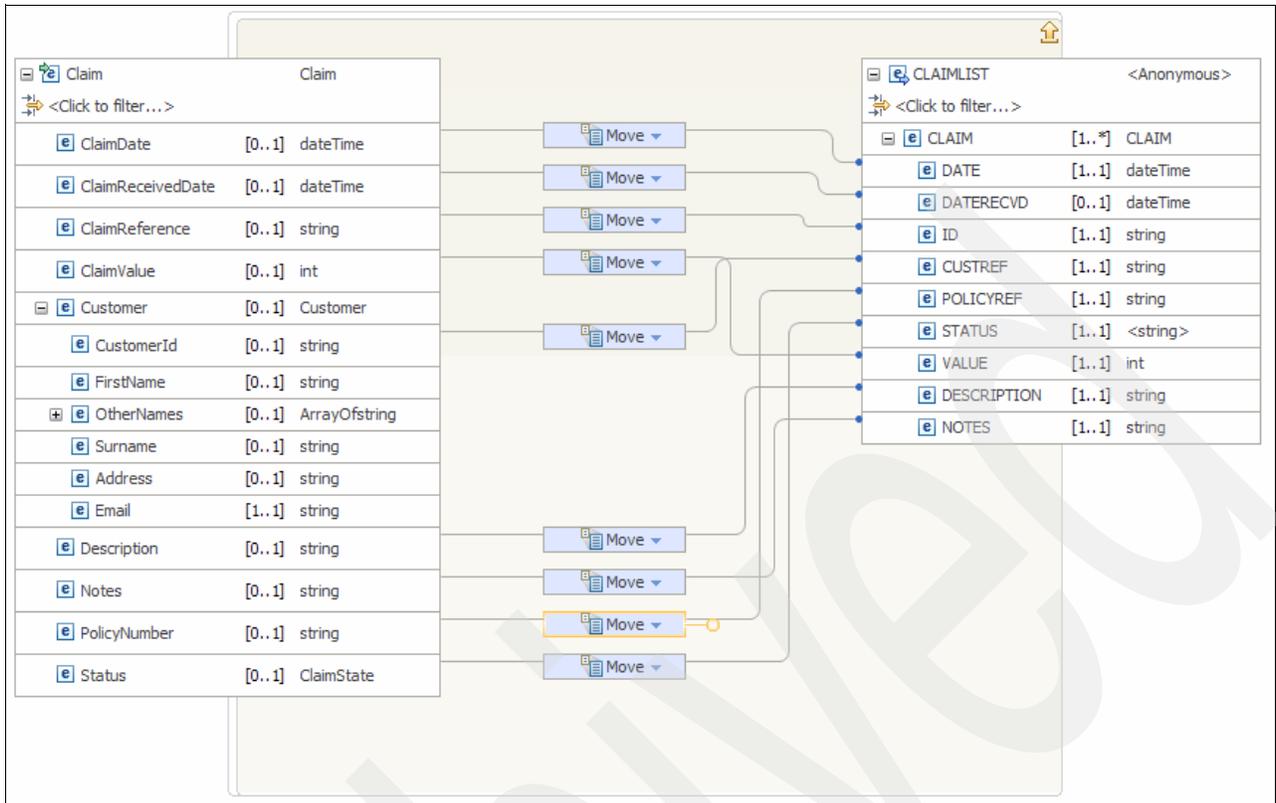


Figure 7-60 Mapping an XMLNSC Claim element onto a DFDL CLAIMLIST element

The mapping itself only needs to alter the logical structure of the message. Any physical representation, such as delimiters, terminators, and initiators, are added by the DFDL parser when the logical message tree is serialized by the MQOutput node.

An extract from serialized ViewClaim message is shown in Figure 7-61.

```

0000: 52 45 50 4c 59 43 43 30 25 32 43 52 45 53 50 4f REPLYCC0.2CRESPO
0010: 4e 53 45 44 41 54 41 43 4c 41 49 4d 4c 49 53 54 NSEDATACLAIMLIST
0020: 43 4c 41 49 4d 32 30 31 32 2d 30 33 2d 30 37 2b CLAIM2012.03.07.
0030: 30 34 25 33 41 32 35 25 33 41 34 36 25 32 43 32 04.3A25.3A46.2C2
0040: 30 31 32 2d 30 33 2d 32 37 2b 30 34 25 33 41 32 012.03.27.04.3A2
0050: 35 25 33 41 34 36 25 32 43 43 4c 41 49 4d 30 30 5.3A46.2CCLAIM00
0060: 30 31 25 32 43 43 55 53 54 30 30 30 30 33 25 32 01.2CCUST00003.2
0070: 43 50 4f 4c 49 43 59 30 30 30 31 25 32 43 41 70 CPOLICY0001.2CApp
0080: 70 72 6f 76 65 64 25 32 43 34 30 30 25 32 43 44 proved.2C400.2CD
0090: 45 53 43 25 32 32 41 63 63 69 64 65 6e 74 61 6c ESC.22Accidental
00a0: 2b 64 61 6d 61 67 65 2b 74 6f 2b 63 61 72 25 32 .damage.to.car.2
00b0: 43 2b 6e 6f 2b 6f 74 68 65 72 2b 70 61 72 74 69 C.no.other.parti
00c0: 65 73 2b 69 6e 76 6f 6c 76 65 64 25 32 32 25 30 es.involved.22.0
00d0: 44 25 30 41 25 30 44 25 30 41 25 32 43 4e 4f 54 D.OA.OA.2CNOT
00e0: 45 53 25 32 32 25 30 41 25 35 42 32 37 25 32 46 ES.22.OA.5B27.2F
00f0: 30 33 25 32 46 32 30 31 32 2b 32 33 25 33 41 30 03.2F2012.23.3A0
0100: 39 25 33 41 32 34 25 35 44 2b 41 67 65 6e 74 52 9.3A24.5D.AgentR
0110: 65 66 25 33 41 2b 41 47 45 4e 54 30 30 31 25 30 ef.3A.AGENT001.0
0120: 41 41 50 50 52 4f 56 45 44 2b 31 30 30 25 30 41 AAPPROVED.100.OA

```

Figure 7-61 A serialized ViewClaim message

7.11 Using a Compute node to transform the Canonical Message into a SOAP message

The expected response for the SOAP request is a single Claim element. The transformation from a CanonicalXML Message to a SOAP message, as shown in Figure 7-62, can therefore be done by copying the message from the input tree and placing it in the correct namespace of the SOAP request.

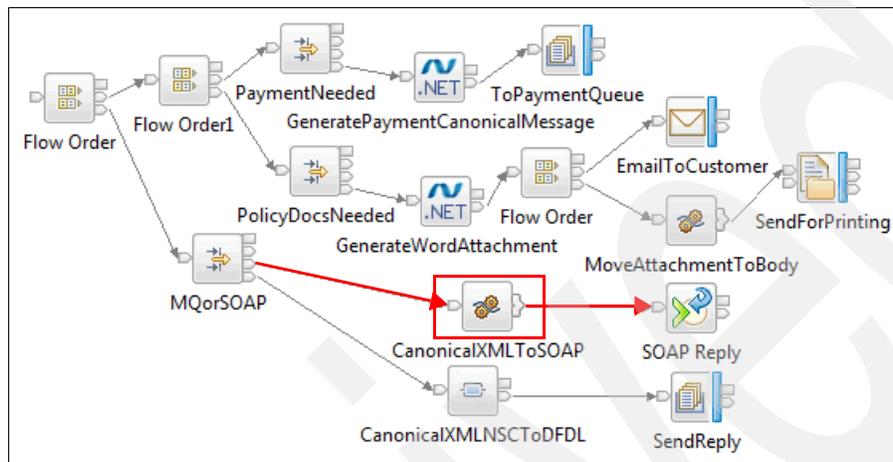


Figure 7-62 Transforming a CanonicalXML Message to a SOAP Message

1. Double-click the **CanonicalXMLToSOAP** node to open the ESQL.
2. Change the name of the new module to CanonicalXMLToSOAP.
3. Replace the Main() function with the code shown in Example 7-55.

Example 7-55 The CanonicalXMLToSOAP node.

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyMessageHeaders();
    CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC');
    SET OutputRoot.XMLNSC.sp:ClaimRefElement =
        InputRoot.XMLNSC.ns:CanonicalMessage.ApplicationMessage
            .ClaimsProcessingResponse.ViewClaimResponse.Claim;
    RETURN TRUE;
END;
```

4. Update the properties for the node, pointing the ESQL module field to CanonicalXMLToSOAP.

We do not need to build a SOAP envelope because the SOAP Reply node does it automatically when it receives the message.

7.12 Handling exceptions in the flow

The flow needs to handle exceptions that occur and convert these into meaningful messages that can be handled by the client. This process is shown in Figure 7-63 on page 354.

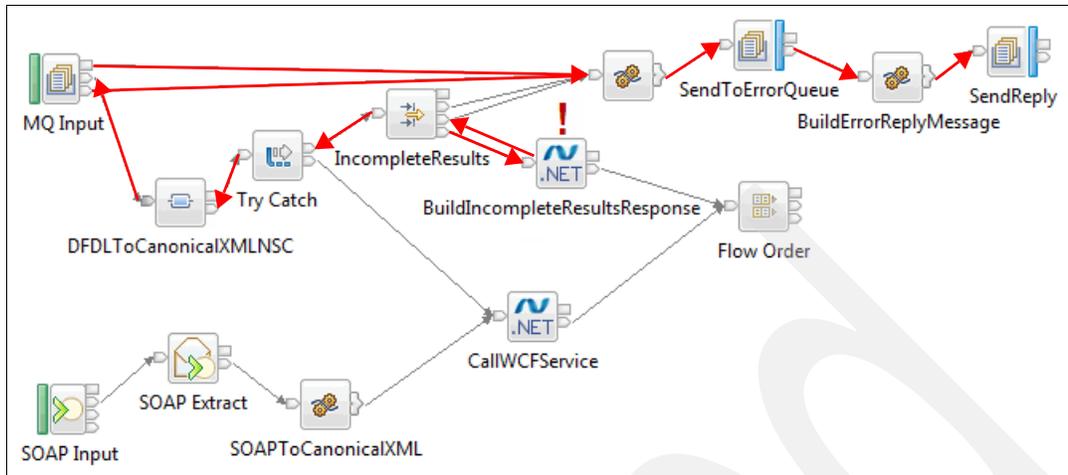


Figure 7-63 Errors and flow handling in Claim Processing

In the case of the SOAP transport, we leave exceptions unhandled and allow the message flow to automatically return a SOAP fault to the application.

Note: The SOAP fault thrown contains full details of the exception. In some environments, this is seen as a security risk. In fact, it is even possible to use the content of SOAP faults to decrypt intercepted messages from an HTTPS session. For this reason, WebSphere Message Broker provides the ability to make the faults that are returned to the client anonymous. This can be enabled by setting the Java property:

```
IBM_JAVA_OPTIONS=-Dwebservices.unify.fault=true
```

This property is set in the Broker's profile (%MQSI_FILEPATH%\bin\mqsiprofile.cmd on Windows or \$MQSI_FILEPATH/mqsiprofile on UNIX.

For the MQ transport, the error handling needs to be more sophisticated so that diverse heritage applications can act appropriately on errors. The strategy followed when an exception occurs in the messageflow is to log an informational message to an error queue. The informational message is in human-readable XML, and the content is based on the exception list structure in the message assembly. Message Broker automatically places details that relate to the error encountered in this exception list structure. A reply message is then sent to the application with a non-zero completion code to indicate failure. This message also contains the msgld of the message put to the error queue so that the application can examine the failure reason if required.

The ConvertExceptionListToMessage node is responsible for navigating through the ExceptionList tree and building the XML error message.

1. Double-click the **ConvertExceptionListToMessage** node to open the ESQL.
2. Change the name of the module to ConvertExceptionListToMessage.
3. Replace the Main() function with the code shown in Example 7-56 to the ConvertExceptionListToMessage node.

Example 7-56 Building the XML exception message

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    CALL CopyMessageHeaders();
    CREATE LASTCHILD OF OutputRoot DOMAIN('XMLNSC');
```

```

    SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)Version = '1.0';
    SET OutputRoot.XMLNSC.(XMLNSC.XmlDeclaration)*.(XMLNSC.Attribute)Encoding = 'UTF-8';
    CREATE LASTCHILD OF OutputRoot.XMLNSC NAME('ERRORMESSAGE');
    DECLARE outputRef REFERENCE TO OutputRoot.XMLNSC.ERRORMESSAGE;
    DECLARE exceptionRef REFERENCE TO InputExceptionList.*[1];
    WHILE LASTMOVE(exceptionRef) DO
        IF exceptionRef.Number IS NOT NULL THEN
            CREATE LASTCHILD OF outputRef NAME('ERROR') VALUE ('ERRORNUM: ' ||
                CAST(exceptionRef.Number AS CHARACTER) || ' TEXT:' ||
exceptionRef.Text);
            END IF;
            MOVE exceptionRef LASTCHILD;
        END WHILE;
    RETURN TRUE;
END;

```

4. Update the properties of the node so that the ESQL module field points to the ConvertExceptionListToMessage module.

The node starts out by taking a copy of the message headers from the input message, which provides the output message tree with a suitable properties folder and MQMD. We then create a new XMLNSC tree with an XML declaration.

The error message we are building has a root element of <ERRORMESSAGE> that is created using the CREATE LASTCHILD statement.

We then declare two references, the first outputRef is declared to the <ERRORMESSAGE> element that was just created and this will be used to add new elements into the output tree. The second is declared to the first child of the InputExceptionList tree. The * (asterisk) operator will select any element regardless of name and the [1] is an index to the first element matching.

ExceptionLists are nested structures with each exception containing exceptions relating the root cause. It is typical for an additional layer of exception wrapping to occur in each node that attempts to handle the exception and failed. The WHILE loop iteratively traverses the ExceptionList tree by moving to the last child setting, a new child of the <ERRORMESSAGE> root element, to a concatenation of the BIP error number and the exception text.

An example ExceptionList is shown in the WebSphere Message Broker Flow Debugger in Figure 7-64 on page 356.

[-] ◆ RecoverableException	
◆ File	F:\build\800_P\src\DataFlowEngine\ImbDataFlowNode.cpp
◆ Line	1129
◆ Function	ImbDataFlowNode::createExceptionList
◆ Type	ComIbmTryCatchNode
◆ Name	ClaimProcessingFlow #FCMComposite_1_22
◆ Label	ClaimProcessingFlow.Try Catch
◆ Catalog	BIPmsgs
◆ Severity	3
◆ Number	2230
◆ Text	Node throwing exception
[-] ◆ RecoverableException	
◆ File	F:\build\800_P\src\DataFlowEngine\DotNet\Library\ImbDotNetComputeNode.cpp
◆ Line	417
◆ Function	ImbDotNetComputeNode::evaluate
◆ Type	ComIbmDotNetComputeNode
◆ Name	ClaimProcessingFlow #FCMComposite_1_3
◆ Label	ClaimProcessingFlow.CallWCFService
◆ Catalog	BIPmsgs
◆ Severity	3
◆ Number	2230
◆ Text	Caught exception and rethrowing
[-] ◆ RecoverableException	
◆ File	
◆ Line	0
◆ Function	
◆ Type	
◆ Name	
◆ Label	
◆ Catalog	BIPmsgs
◆ Severity	3
◆ Number	7499
◆ Text	CLAIMSPROCESSING FAULT. Code: ERR004 ErrorText: Could not view claim. Reason: Claim CLAIM101 does not exist.
[+] ◆ Insert	

Figure 7-64 The nested ExceptionList structure

Output from the same exception after it is processed by the ConvertExceptionListToMessage node can be seen using the ERROR_QUEUE Dequeue file from the ClaimsProcessingMQTest Test Client. The XML structure is shown in Example 7-57.

Example 7-57 The XML Error Message

```
<?xml version="1.0" encoding="UTF-8"?>
<ERRORMESSAGE>
  <ERROR>
    ERRORNUM: 2230 TEXT:Caught exception and rethrowing
  </ERROR>
  <ERROR>
    ERRORNUM: 7499 TEXT:CLAIMSPROCESSING FAULT. Code: ERR004
    ErrorText: Could not view claim. Reason: Claim CLAIM101
    does not exist.
  </ERROR>
</ERRORMESSAGE>
```

As you can see, the exception structure was flattened from a nested parent-child relationship into a list for processing in connected MQ applications.

After the error message is sent to the ERROR_QUEUE, the flow needs to build an error reply to send back to the reply queue. This is performed by the BuildErrorReplyMessage node.

1. Double-click the **BuildErrorReplyMessage** node to open the ESQL.
2. Replace the Main() function with the code shown in Example 7-58.

Example 7-58 Building the error reply message

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyMessageHeaders();
  -- Create a new DFDL output message
  SET OutputRoot.MQMD.CorrelId = InputRoot.MQMD.MsgId;
  CREATE LASTCHILD OF OutputRoot DOMAIN('DFDL');
  SET OutputRoot.DFDL.ns1:REPLY.COMPLETIONCODE = 1;
  SET OutputRoot.DFDL.ns1:REPLY.ERRORMSGID = InputLocalEnvironment
    .WrittenDestination.MQ.DestinationData.msgId;
  SET OutputRoot.DFDL.ns1:REPLY.ERRORTEXT =
    'PROCESSING FAILED. ERROR MESSAGE WRITTEN TO ERROR QUEUE WITH '
    || 'SPECIFIED MSGID';

  --Build the destination list
  SET OutputLocalEnvironment.Destination.MQ.DestinationData.queueName =
    InputRoot.MQMD.ReplyToQ;
  SET OutputLocalEnvironment.Destination.MQ.DestinationData
    .queueManagerName = InputRoot.MQMD.ReplyToQMgr;

  RETURN TRUE;
END;
```

3. Update the properties for the BuildErrorReplyMessage node so that the ESQL module field points to BuildErrorReplyMessage.

The REPLY message in the DFDL domain for an error message consists of a sequence of COMPLETIONCODE, ERRORMSGID, and ERRORTEXT, as shown in the DFDL editor in Figure 7-65.

Name	Type	Min Occurs	Max Occurs	Default Value	Sample Value
[-] REPLY					
[-] sequence		1	1		
[+] COMPLETIONCODE	int	1	1		0
[+] ERRORMSGID	hexBinary	0	1	00000000000000000000000000000000	0F00
[+] ERRORTEXT	string	0	1		
[+] RESPONSEDATA		0	1		

Figure 7-65 The REPLY message shown in the DFDL editor

The code for the BuildErrorReplyMessage first copies the headers so that we have a suitable Properties tree and MQMD for propagation. The CorrelId is set to the input msgId so that the reply can be correlated with the request message by the MQ application that receives the reply. We then create a new message body using the DFDL parser and set the completion code to 1 to indicate a failure.

When the MQOutput node writes a message to a queue it places information about the MQ PUT operation in the LocalEnvironment under the WrittenDestination folder. This information is used to set the ERRORMSGID to the msgId of the MQ message sent to the ERROR_QUEUE.

Finally, the LocalEnvironment is updated with the DestinationData associated with the reply details so that the subsequent MQOutput node sends the message to the correct queue and queue manager.

7.13 Building and deploying the Visual Studio project

The code for the .NETCompute nodes was created in a Visual Studio solution called ClaimsProcessingBrokerProject. That code needs to be assembled in order for the .NETCompute nodes to reference the files.

7.13.1 Building a solution

To build a solution:

1. Open the **ClaimsProcessingBrokerProject** solution in Visual Studio.
1. From the Solution Explorer, right-click the project, and select **Build**.
2. Verify that the **Output window** shows you where the assembly file is saved on your file system, as shown in Figure 7-66. You must resolve any errors for the solution to build.

Note: If you do not see the Output tab, open it by selecting **Debug** → **Windows** → **Output** (Microsoft Visual Studio Professional Edition). If you are using Microsoft Visual Studio Express Edition, open it by selecting **View** → **Output**.

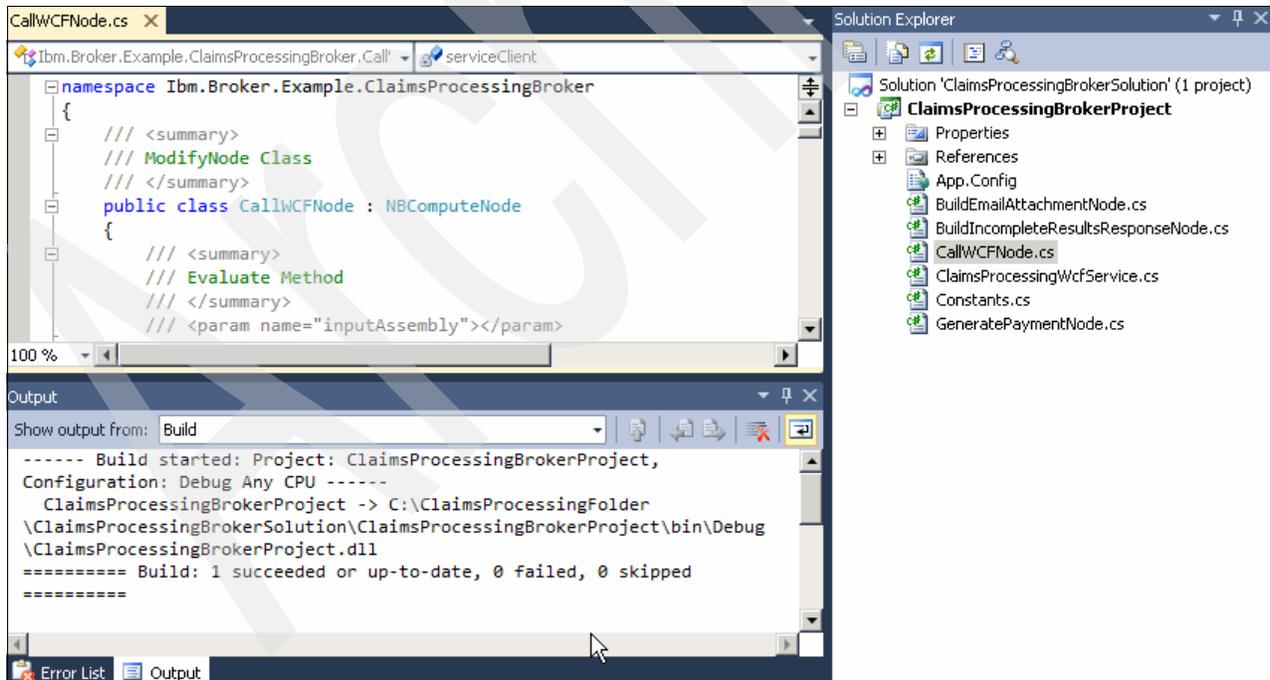


Figure 7-66 Verifying the built assembly file has been saved into the system

7.14 Updating the properties for the .NETCompute nodes

Each .NETCompute node must have a pointer to the assembly (dll) file and the name of the class to use. Before the .NET solution was built in Visual Studio, the dll file did not exist. Now that the build is performed, you can update the properties for all the Compute nodes.

1. In the Message Broker Toolkit, select the CallWCFNode node, and in the Properties view, verify that the Assembly name property is specified as follows:

C:\ClaimsProcessingFolder\ClaimsProcessingBrokerSolution\ClaimsProcessingBrokerProject\bin\Debug\ClaimsProcessingBrokerProject.dll

Specify CallWCFNode for the class name, as shown in Figure 7-67.

.NETCompute Node Properties - CallWCFService	
Description	
Basic	Assembly name* <input type="text" value="C:\ClaimsProcessingFolder\ClaimsProcessingBrokerSolution\ClaimsProcessingBrokerProject\bin\Debug\ClaimsProcessingBrokerProject.dll"/>
Visual Studio	Class name <input type="text" value="CallWCFNode"/>
Advanced	

Figure 7-67 Verifying Assembly name property and node class name

2. Use this same process to update each .NETCompute node using the values in Table 7-3. In our example, <Full Path> is:

C:\ClaimsProcessingFolder\ClaimsProcessingBrokerSolution\ClaimsProcessingBrokerProject\bin\Debug\

Table 7-3 Summary of .NETCompute nodes configuration

Name	Field	Path
BuildIncompleteResultResponse	Assembly name	<Full Path>/ ClaimsProcessingBrokerAssembly.dll.
	Class name	BuildIncompleteResultResponseNode
CallWCFService	Assembly name	<Full Path>/ ClaimsProcessingBrokerAssembly.dll
	Class name	CallWCFNode
GeneratePaymentCanonicalMessage	Assembly name	<Full Path>/ ClaimsProcessingBrokerAssembly.dll
	Class name	GeneratePaymentNode
GenerateWordAttachment	Assembly name	<Full Path>/ ClaimsProcessingBrokerAssembly.dll
	Class name	BuildEmailAttachmentNode

7.15 Building and deploying the Message Broker application

Before the Message Broker Application can be used, it needs to be deployed to be built and deployed to a broker. WebSphere Message Broker applications are contained in Broker Archive (BAR) files that can be deployed to an execution group by dragging and dropping from within the Message Broker Toolkit or the WebSphere Message Broker Explorer.

Additionally, because this scenario contains .NET code, the .NET assemblies must be built and referenced by the message flow.

7.15.1 Creating the BAR file

To create the BAR file for deployment:

1. Right-click the application in the Broker Development view, and select **New** → **BAR File**. Enter `ClaimProcessingBarFile` as the name of the BAR file, and click **Finish**.
2. On the Prepare tab, select **ClaimProcessing**, and then click the **Build and Save** button.
3. Select the **Manage** tab:
 - a. Expand the tree, and select **ClaimProcessFlow**.
 - b. Provide values for the properties:
 - **Directory**: A directory for file archiving
 - **FromEmail**: The sender email address
 - **TemplateLocation**: The location of letter template.

These steps are highlighted in Figure 7-68.

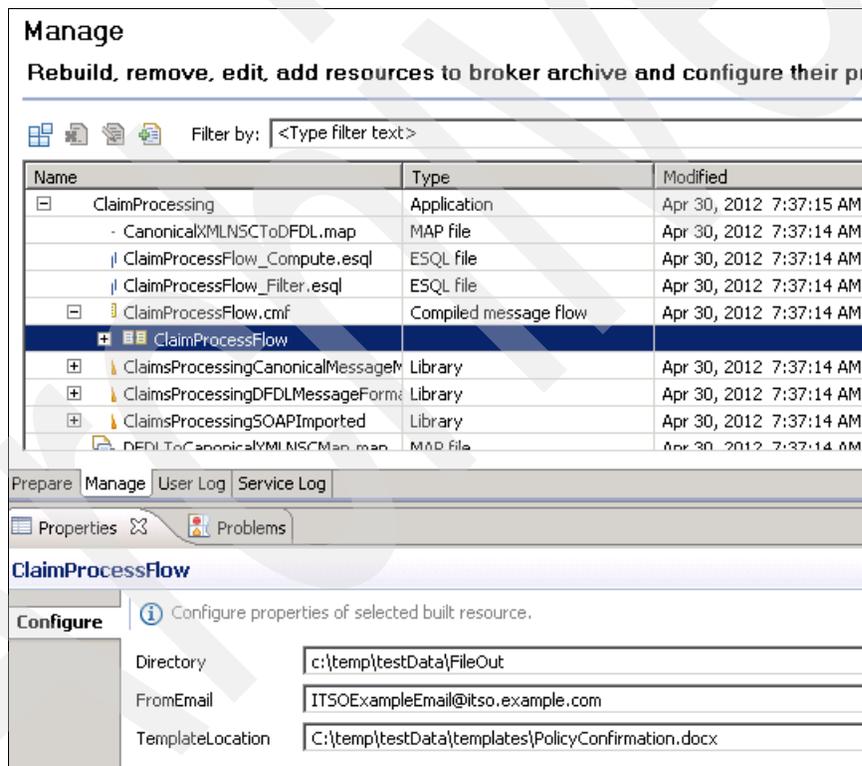


Figure 7-68 Completing the properties for the flow in the BAR file

4. Press **Ctrl+S** to save the BAR file.

7.15.2 Deploying the BAR file

The next phase is to deploy the BAR file. To deploy the BAR file:

1. Drag the BAR file and drop it into the target execution group, as shown in Figure 7-69 on page 361.

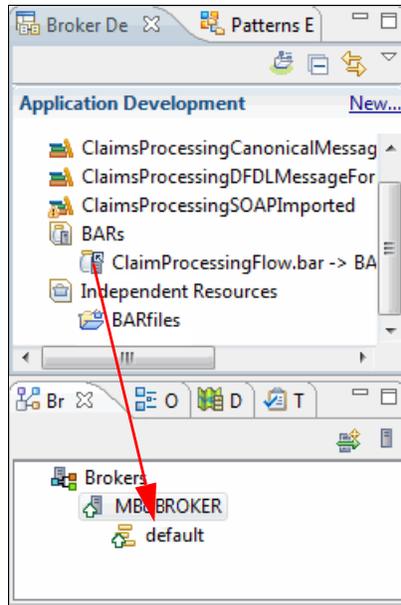


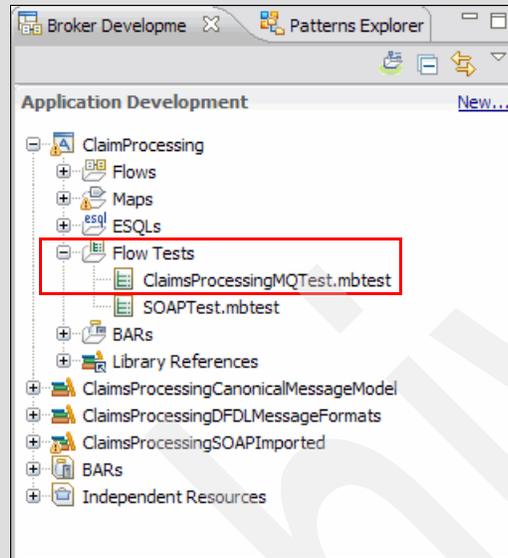
Figure 7-69 Drag and drop the BAR file to deploy

7.16 Testing the WebSphere Message Broker application

This section outlines testing options to ensure the success of the Message Broker application. First, we test the WebSphere MQ interface, and then we test the SOAP interface.

7.16.1 Testing the MQ interface

Additional materials: The completed message flow project includes a Message Broker Test Client that defines an Enqueue file for testing the operations exposed by the WCF Service. The project interchange file `WCFClient\Project Interchange files\ClaimProcessingPI.zip` is included in the additional materials for this book. You can open a new workspace, import the project interchange, and copy the `ClaimsProcessingMQTest.mbttest` file to the `ClaimProcessing` application in your own workspace. This process creates a `Flow Tests` folder with the new test client.



To test the message flow using the MQ transport:

1. Delete all files from the `WCF Services_BaseDir\customers` directory. The `_BaseDir` variable is defined in “Implementing the `IClaimsProcessingWcfService` interface” on page 210 (see Example 6-41 on page 210) and is set to `c:\temp\testData` by default. Clearing the `c:\temp\testData\customers` directory ensures that the example data is recreated when the WCF service starts.
2. If there are claims other than `CLAIM0001` in the claims subdirectory, such as claims created while testing the WCF service with the WCF Test Client in 6.3.9, “Testing the WCF Service using the WCF Test Client” on page 235, delete these too.
3. Stop and restart the **ClaimsProcessingWCFService**.
4. If you want to test the `CreatePolicy` operation, ensure that an smtp server is running and that the `EmailOutput` node is configured to connect to it.
5. Ensure that the `ClaimProcessing` application is deployed to your broker and the prerequisite queues are defined on the broker’s queue manager.
6. Double-click **ClaimsProcessingMQTest.mbttest** in the `Flow Tests` folder to open the Message Broker Test Client in the editor pane. The test client defines a number of Enqueue operations that will send a message to the input queue. Each of the Enqueue operations corresponds to a different operation on the WCF service.

The Dequeue events correspond to reading a message from a queue. There are three Dequeue events defined to read from the reply queue (`OUT`), the error queue (`ERROR_QUEUE`), and the payment queue (`PAYMENT_QUEUE`).

Figure 7-70 shows the ClaimsProcessingMQTest Test Client in its initial state.

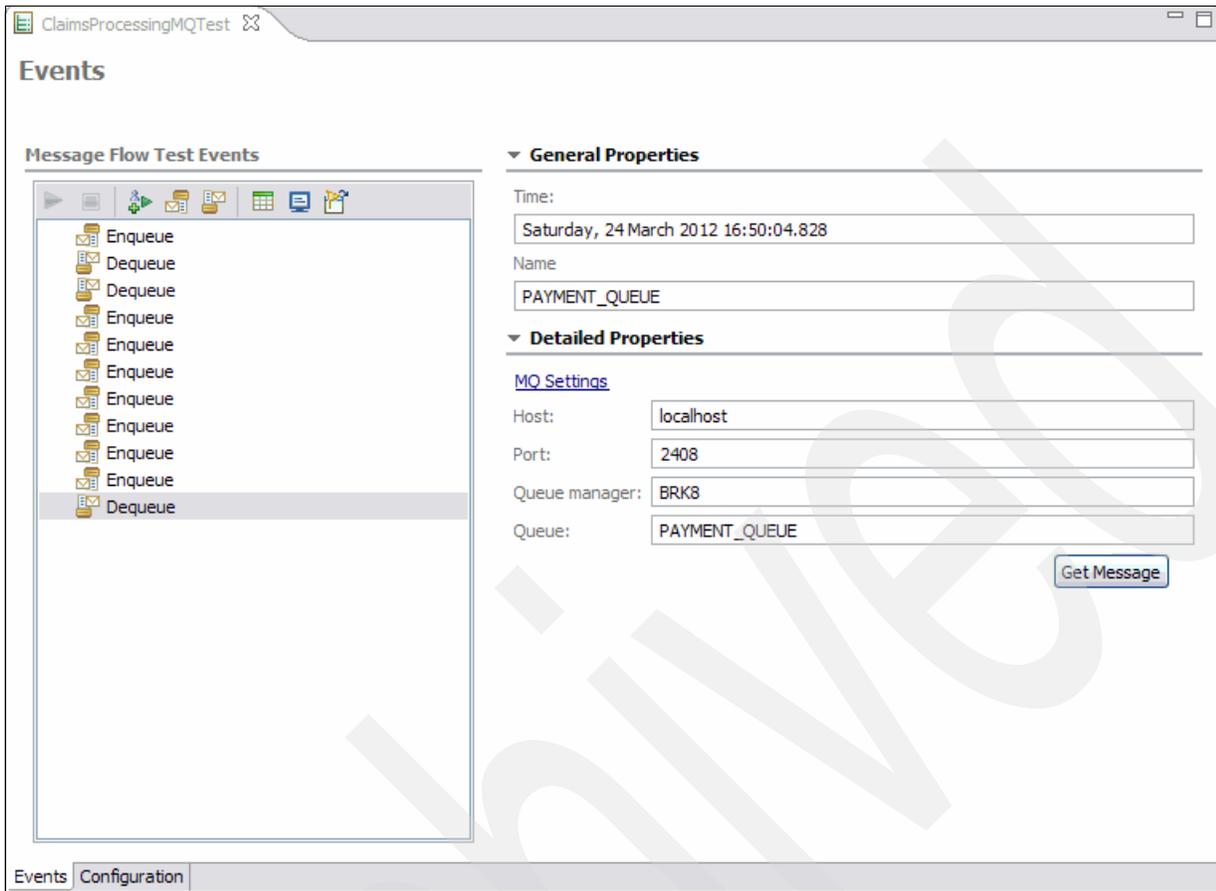


Figure 7-70 The ClaimsProcessingMQTest Message Broker Test Client in its initial state

7. The Message Broker Test Client allows some common configuration options to be applied across all Enqueue and Dequeue operations in the client. In this instance, all requests must have a valid ReplyTo queue set so that the flow knows where to send the response. As shown in Figure 7-71 on page 364, the OUT queue is specified as the ReplyTo queue name for all operations by setting it in the MQ Default Header. This causes the Message Broker Test Client to populate the MQMD with the ReplyTo queue.

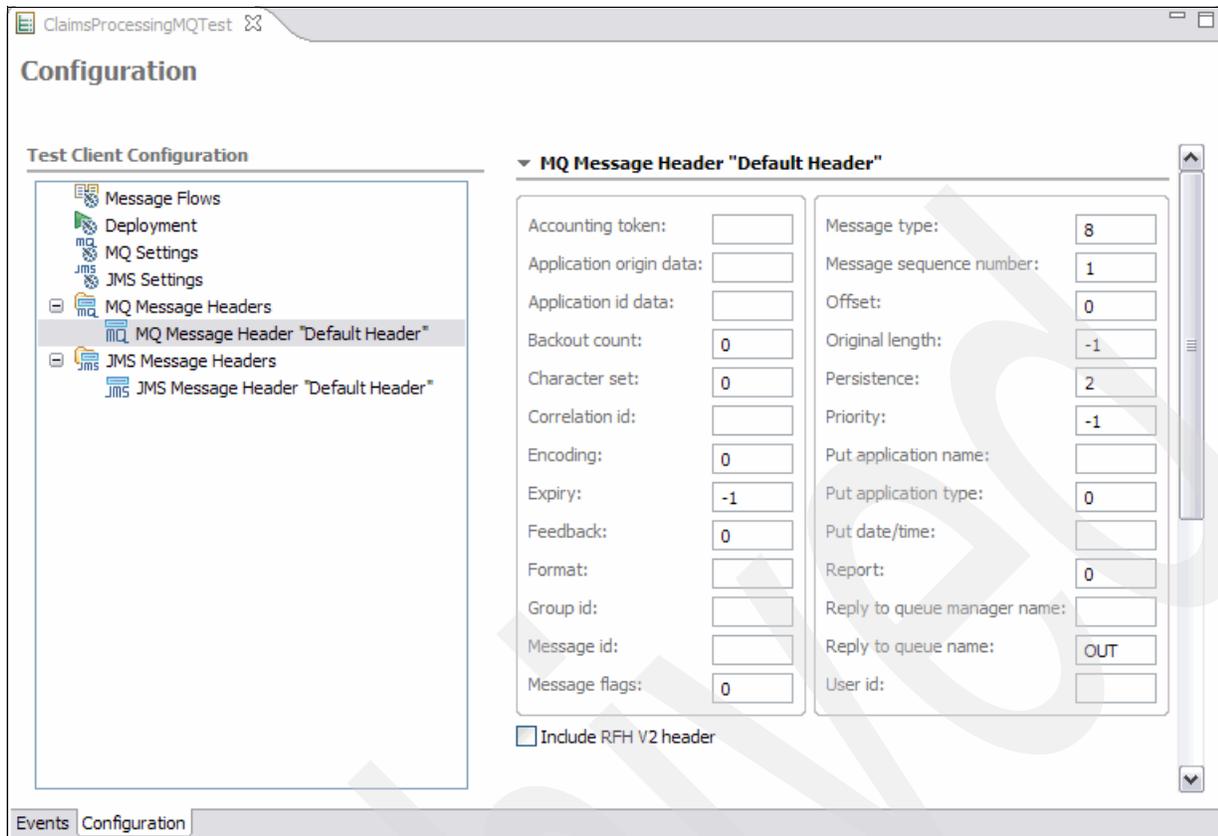


Figure 7-71 Configuring the default MQMD for all enqueue and dequeue operations

Testing the ViewOutstandingClaims operation

To test the ViewOutstandingClaims operation:

1. Locate the enqueue named ViewOutstandingClaims. The name is in the General Properties section.
2. Modify the connection properties to refer to your broker. Because the Message Broker Client uses the MQ Java Client to connect, the port must correspond to a running listener on the broker's queue manager.
3. The input message is displayed in the Message section. When working with binary message types, it can be useful to view the data as hexadecimal by checking the **Show in hexadecimal viewer** check box, as shown in Figure 7-72 on page 365.

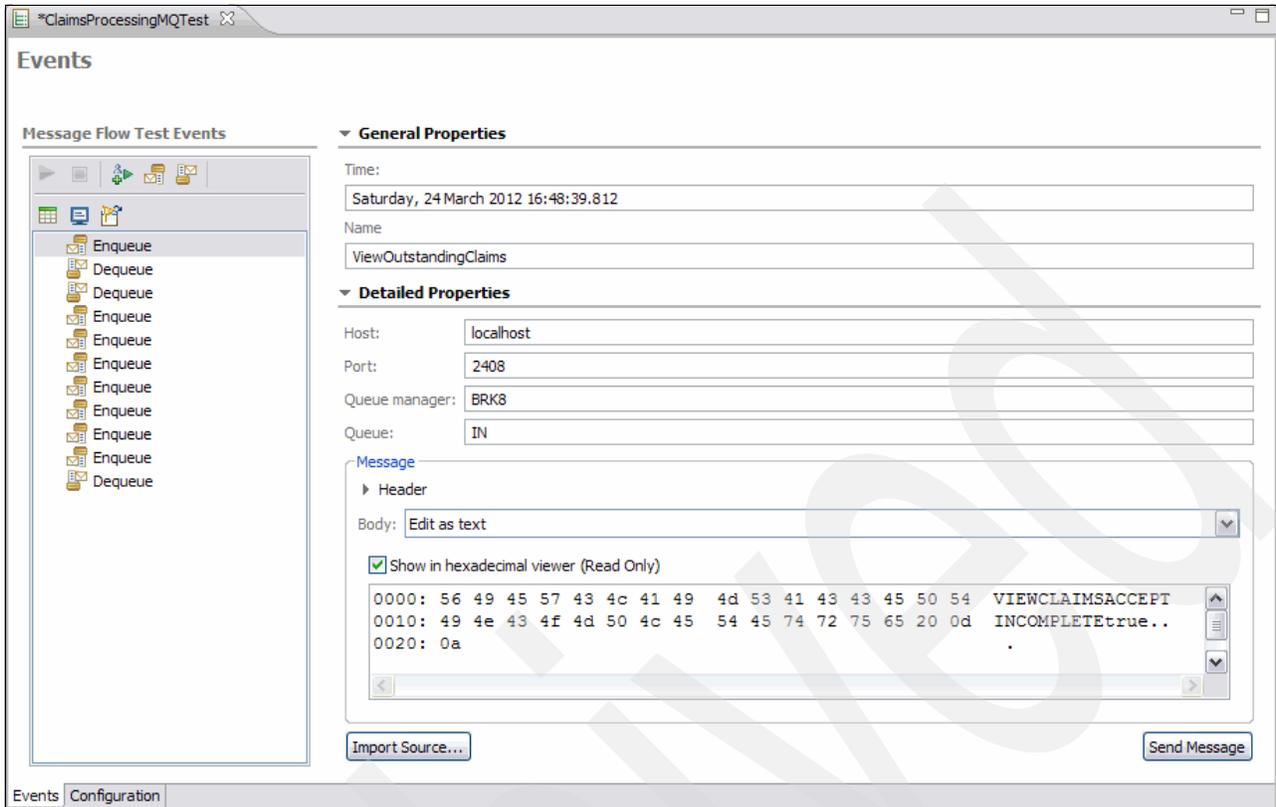


Figure 7-72 The ViewOutstandingClaims Enqueue

4. Click **Send Message** to send the message to the IN queue.

A message is displayed below the Enqueue file saying that a message was sent to MQ queue IN, as shown in Figure 7-73. If there was a problem putting the message on the queue, an error message appears here instead.

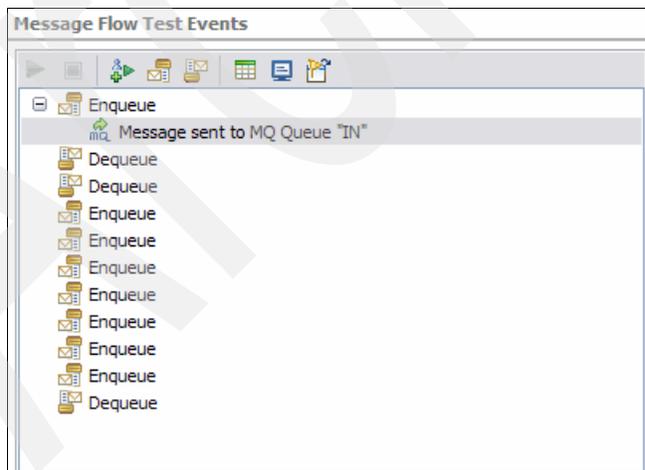


Figure 7-73 Confirmation that the message has been sent to MQ

5. The message is now processed by the ClaimsProcessing flow, and a message is put on the Reply queue. Locate the Dequeue corresponding to the OUT queue, as shown in Figure 7-74 on page 366.

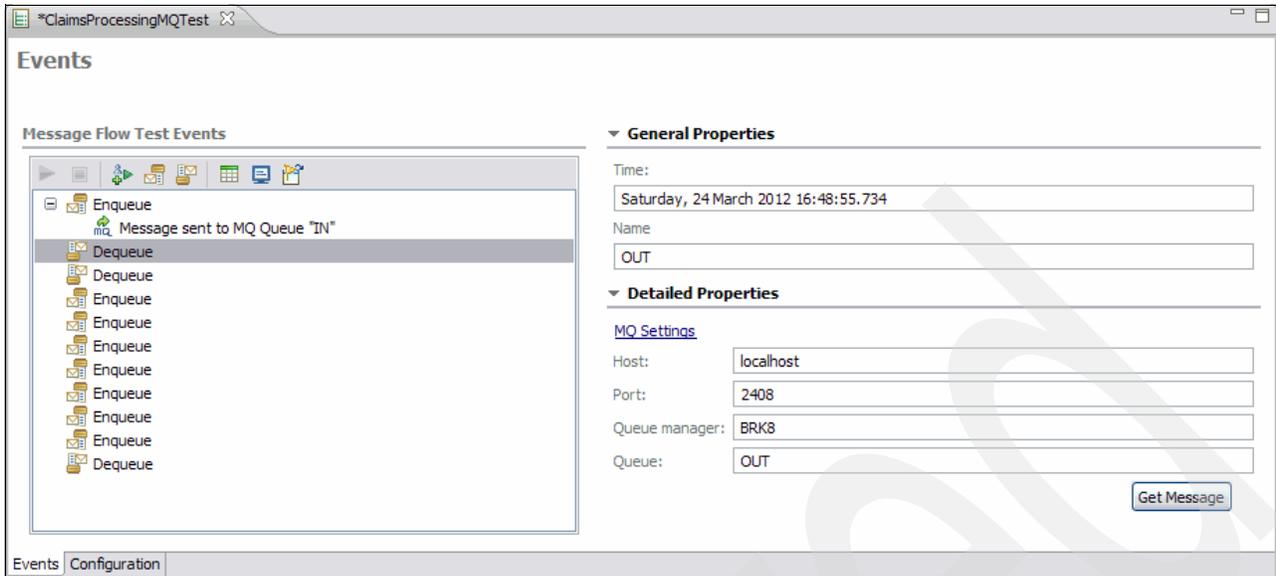


Figure 7-74 The Dequeue for the OUT queue

6. Click **Get Message**.
7. If you receive an Error message indicating that the OUT queue was empty (shown in Figure 7-75), check the Exception trace.

If you see the text `com.ibm.mq.MQException: MQJE001: Completion Code '2', Reason '2033'`, the message flow has not yet finished processing the message, so there is no message to get on the OUT queue. This is more likely to happen on the first invocation of the message flow that initializes the WCF client and the service instance. If this happens, simply wait for a few seconds, and click the **Get Message** button again.

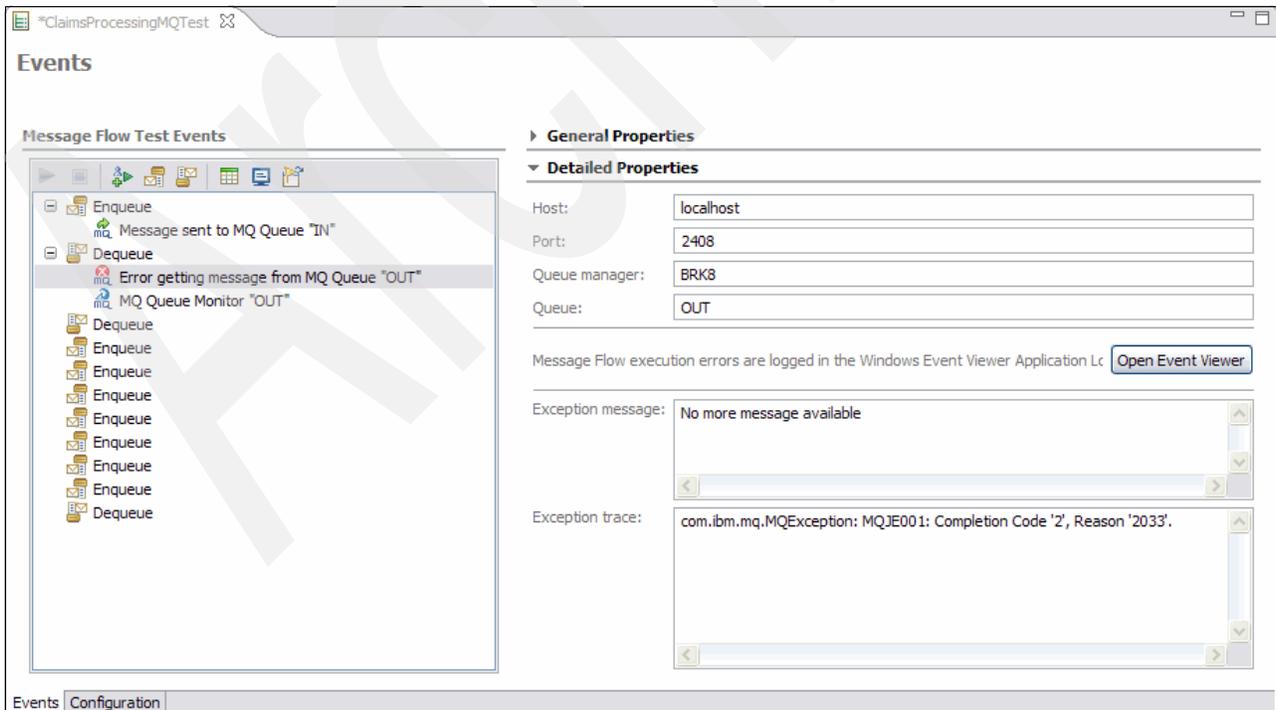


Figure 7-75 An MQRC 2033 error indicating that the OUT queue was empty

If a message is present, you will see an MQ Queue Monitor OUT element created under the Dequeue. The contents of the received message are displayed in the Message panel to the right, as shown in Figure 7-76. Here the REPLYCC completion code was 0, indicating that the operation was successful and the CLAIM details for CLAIM0001 are returned as the only element in the CLAIMLIST.

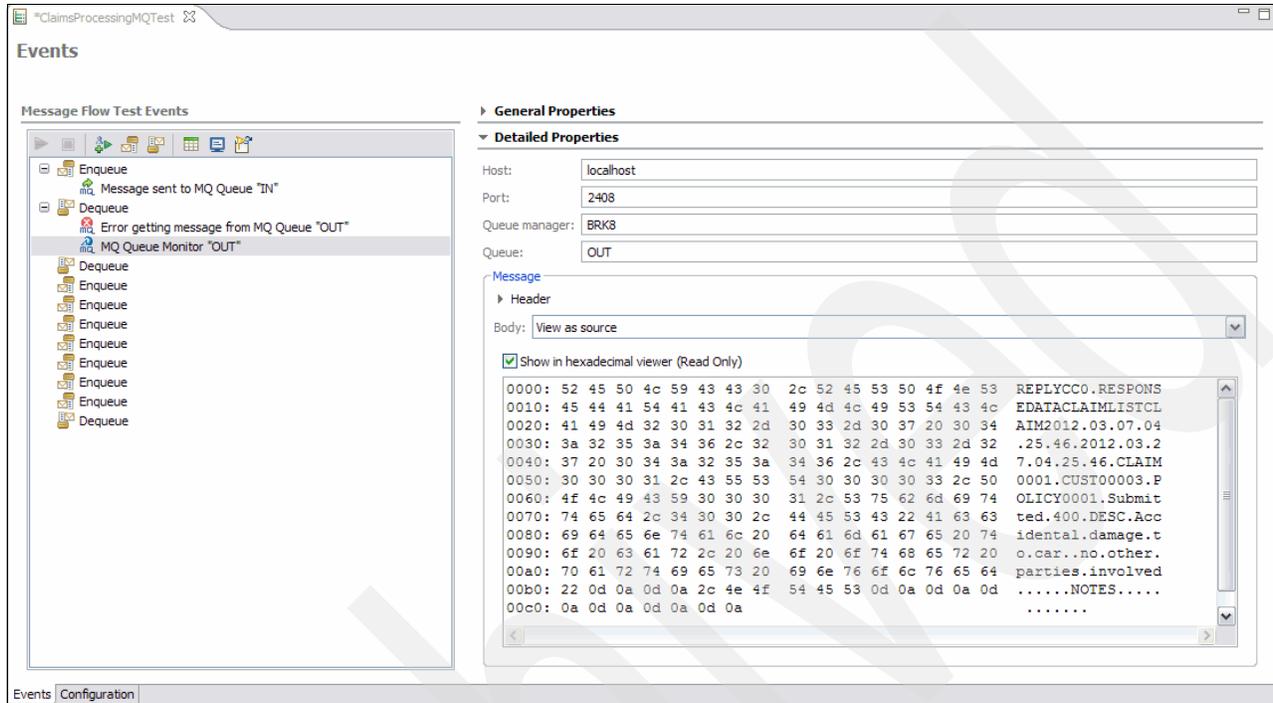


Figure 7-76 The ViewOutstandingClaims response message

Testing the ViewClaims operation

To test the ViewClaims operation:

1. Locate and select the Enqueue named **ViewClaim** to open it in the Message panel.
2. Click the **Send Message** button.

A confirmation that the message was sent to MQ appears underneath the Enqueue, as shown in Figure 7-77 on page 368.

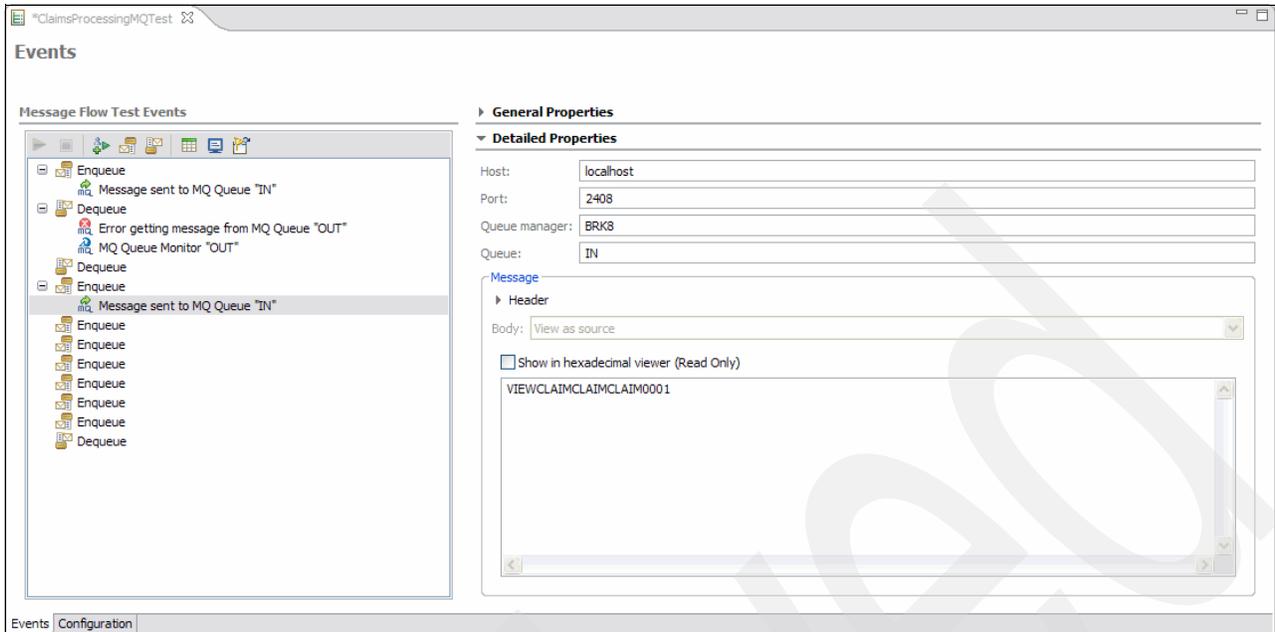


Figure 7-77 The ViewClaim Enqueue

- Return to the Dequeue for the OUT queue, and click the **Get Message** button.

The details of CLAIM0001 are displayed in the Message panel, as shown in Figure 7-78.

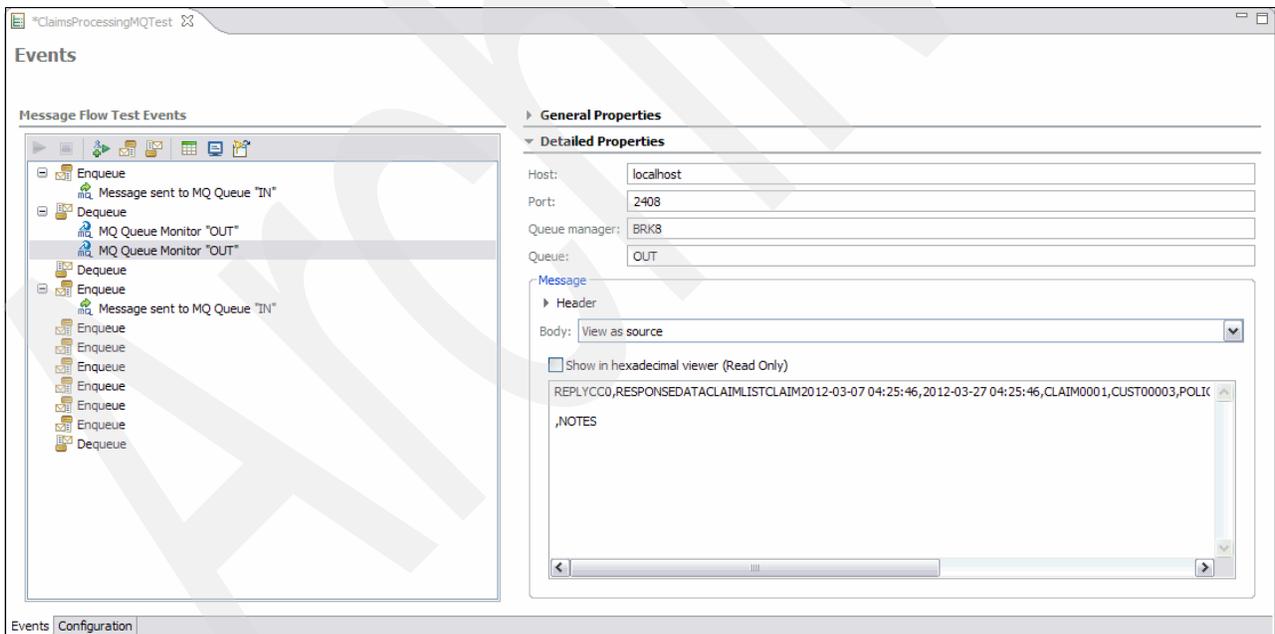


Figure 7-78 The ViewClaim response showing CLAIM0001

To test the ViewClaim method using an invalid input:

- Locate and select the Enqueue named **ViewInvalidClaim**.
- In this case, we make a request to view the claim CLAIM101 that does not exist in the system, so we expect an error.
- Click the **Send Message** button.

You will see a confirmation that the message was sent, as shown in Figure 7-79.

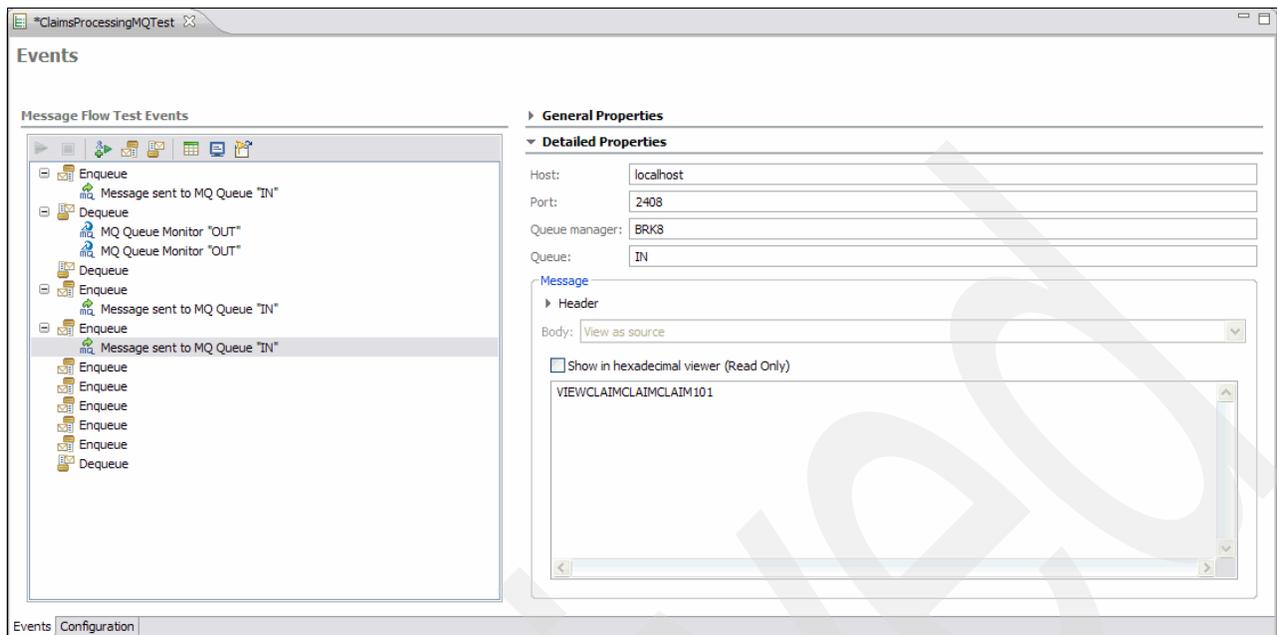


Figure 7-79 The ViewInvalidClaim Enqueue

4. Select the Dequeue associated with the OUT queue, and press the **Get Message** button. You will see a response with a REPLYCC completion code indicating that there was an error. The error text will note that an error was written to the error queue. The response to the OUT queue is shown in Figure 7-80.

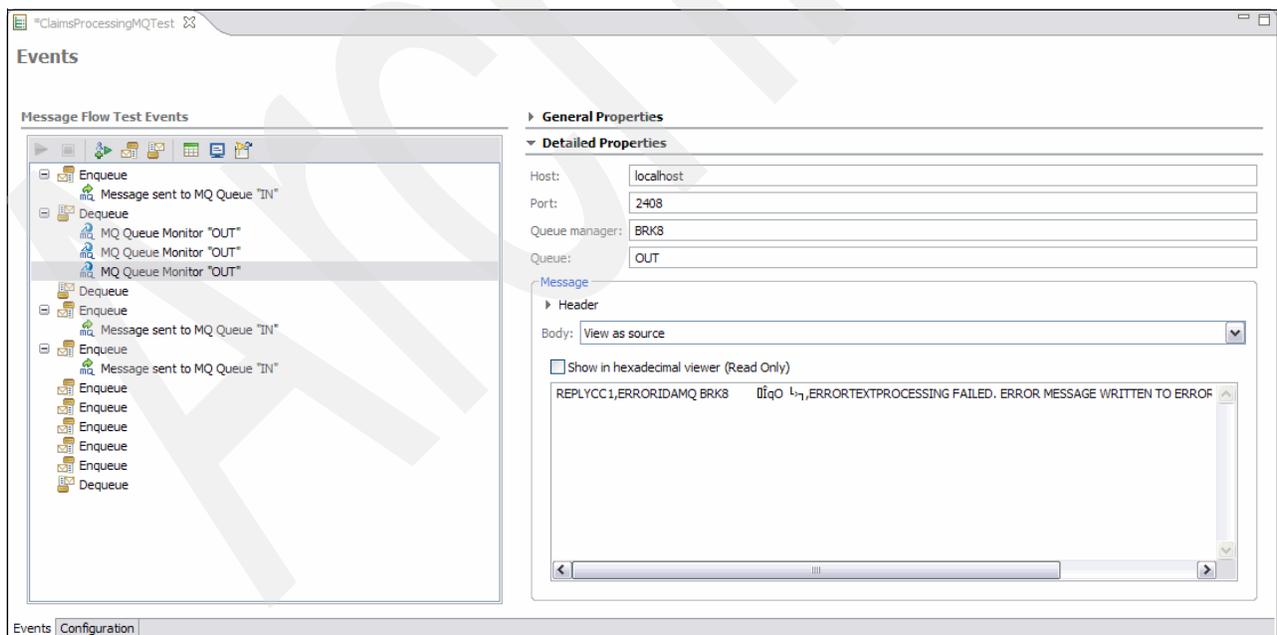


Figure 7-80 An error response

5. In the case of an error, the message flow writes an error message to the `ERROR_QUEUE`. Locate the Dequeue corresponding to the `ERROR_QUEUE`, and press the **Get Message** button.

A message is displayed with the error list encountered by the broker. You can view this as an XML structure and scroll to see that the error code and message set by the WCF service are included in the error message, as shown in Figure 7-81.

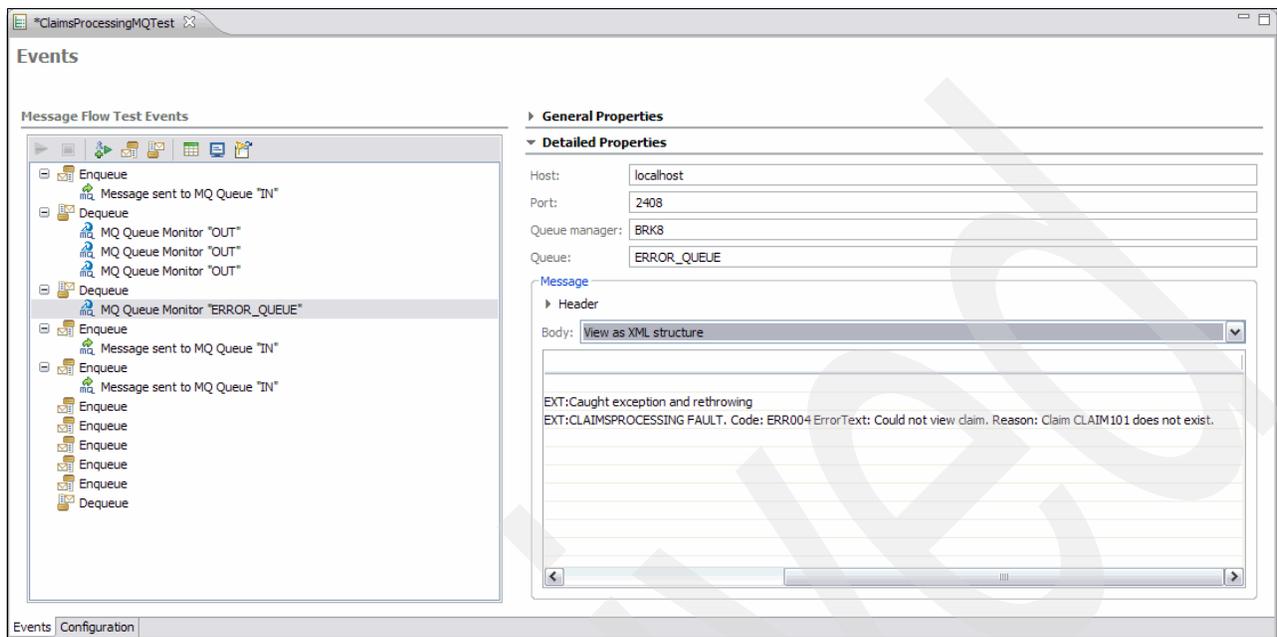


Figure 7-81 The error message contains the customer fault message and error code fields

Testing the ViewPolicy operation

To test the ViewPolicy operation:

1. Locate and select the Enqueue file named **ViewPolicy**.
2. Click **Send Message**.

You should see a confirmation that the message was sent appear underneath the Enqueue, as shown in Figure 7-82 on page 371.

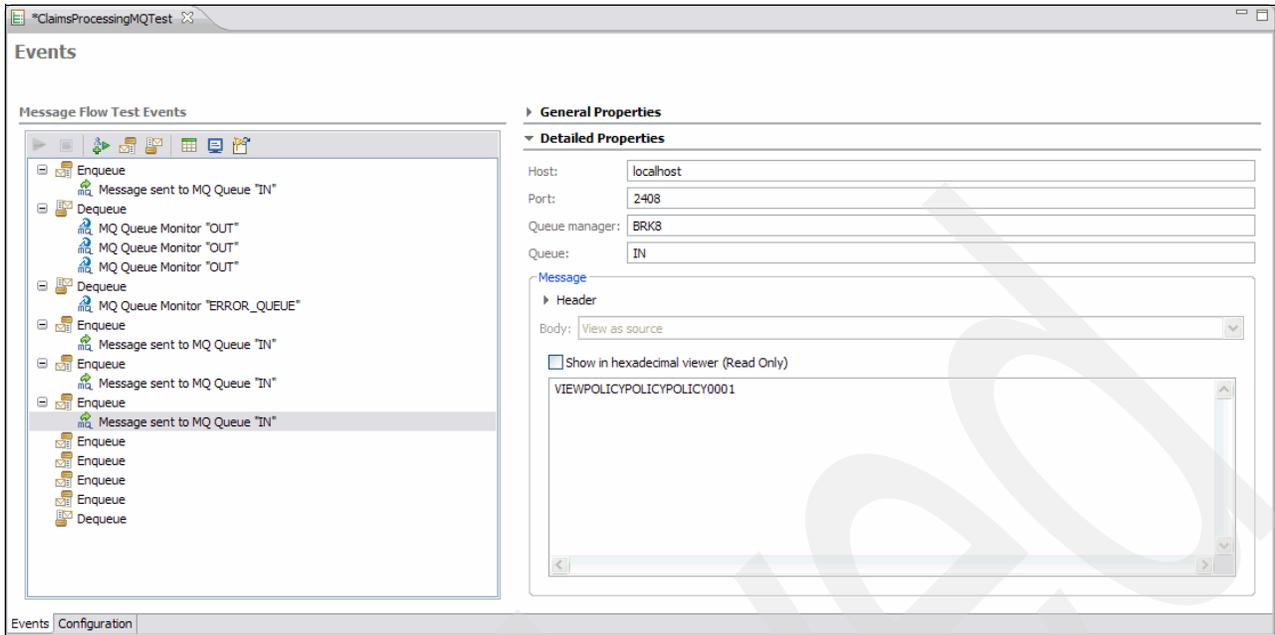


Figure 7-82 The ViewPolicy Enqueue

3. Select the Dequeue corresponding to the OUT queue, and press the **Get Message** button.

You will see a response on the OUT queue with a completion code of 0 indicating success and the details for POLICY0001 attached in the POLICYLIST, as shown in Figure 7-83.

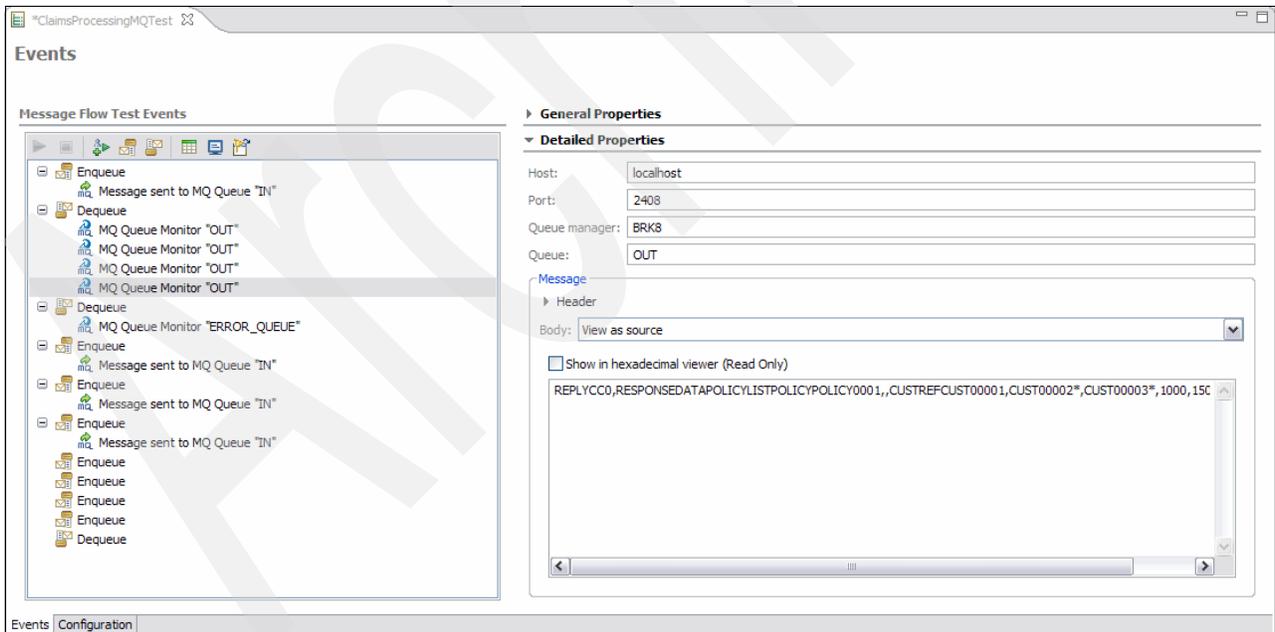


Figure 7-83 The response from the ViewPolicy operation

Testing the CreatePolicy operation

The CreatePolicy operation sends an email to the customer on successful completion of the operation. To run this step, you must ensure that you have a local smtp server running and that the EmailOutput node is configured to access it.

To test the CreatePolicy operation:

1. Select the Enqueue named **CreatePolicy**. Note the terminator character for the list of Named Parties and the list of Other Names in the binary message.
2. If you want to see the email document produced, replace the email address johndoe@itso.example.ibm.com with your own email address.
3. Press the **Send Message** button.

You will see a confirmation message indicating that the message was sent, as shown in Figure 7-84

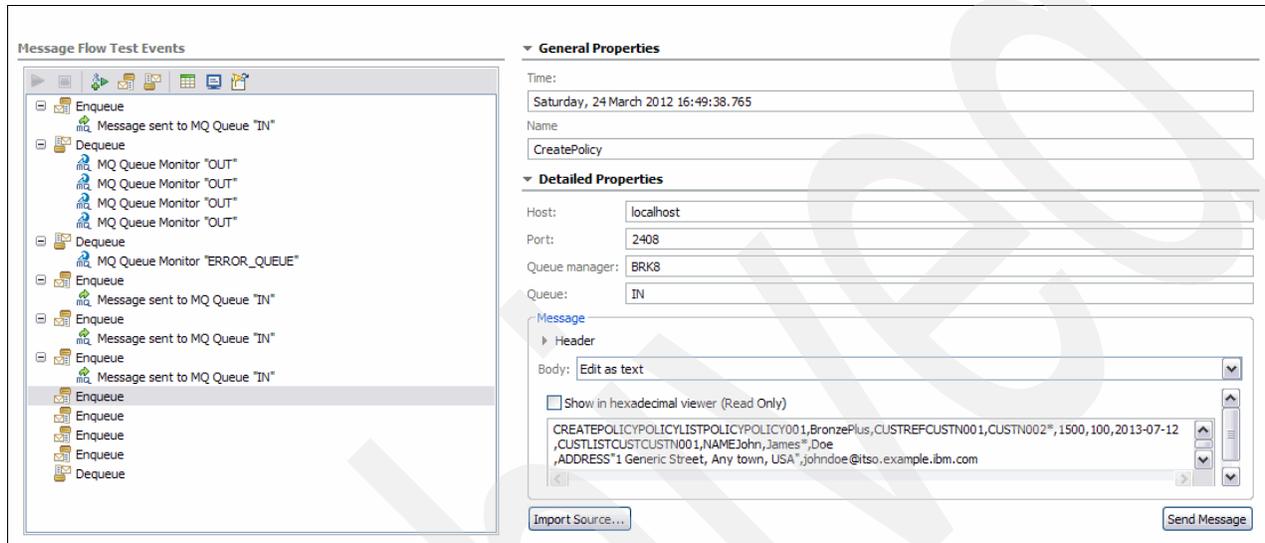


Figure 7-84 The CreatePolicy Enqueue

4. Select the Dequeue corresponding to the OUT queue.
5. Click **Get Message**.
6. The Message panel in the Dequeue will display a successful completion code of 0 and return the policy information including the new policy reference, as shown in Figure 7-85 on page 373.

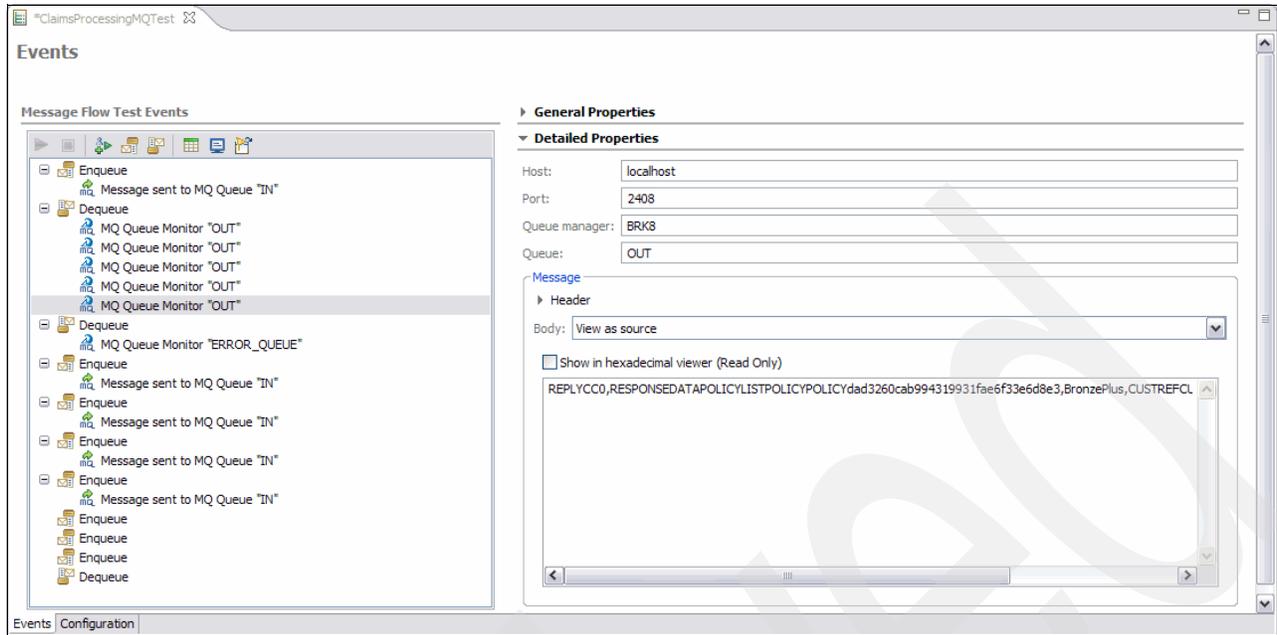


Figure 7-85 The response from the CreatePolicy operation

7. If you configured your own email, you will receive an email with the Policy confirmation document attached.
8. The CreatePolicy operation also causes a file to be written to the output directory that you configured in 7.15, "Building and deploying the Message Broker application" on page 359. By default, this directory is C:\temp\testdata\fileOut. If you check this location, you will see a .docx file named PolicyConfirmation<Date>.docx. A sample document is shown in Figure 7-86 on page 374.



Figure 7-86 Sample document sent by the CreatePolicy operation

Testing the CreateClaim operation

We use the CreateClaim operation to create a second claim in the system so that in the next two steps we can approve one claim and reject the other. To create the CLAIM0002 claims using the Message Broker Test Client, follow these steps:

1. Select the **CreateClaim** Enqueue. Note that the claim specifies the claim reference to be created as CLAIM0002.
2. Click **Send Message**.

You will see a confirmation that the message was sent to the MQ queue appear below the Enqueue, as shown in Figure 7-87 on page 375.

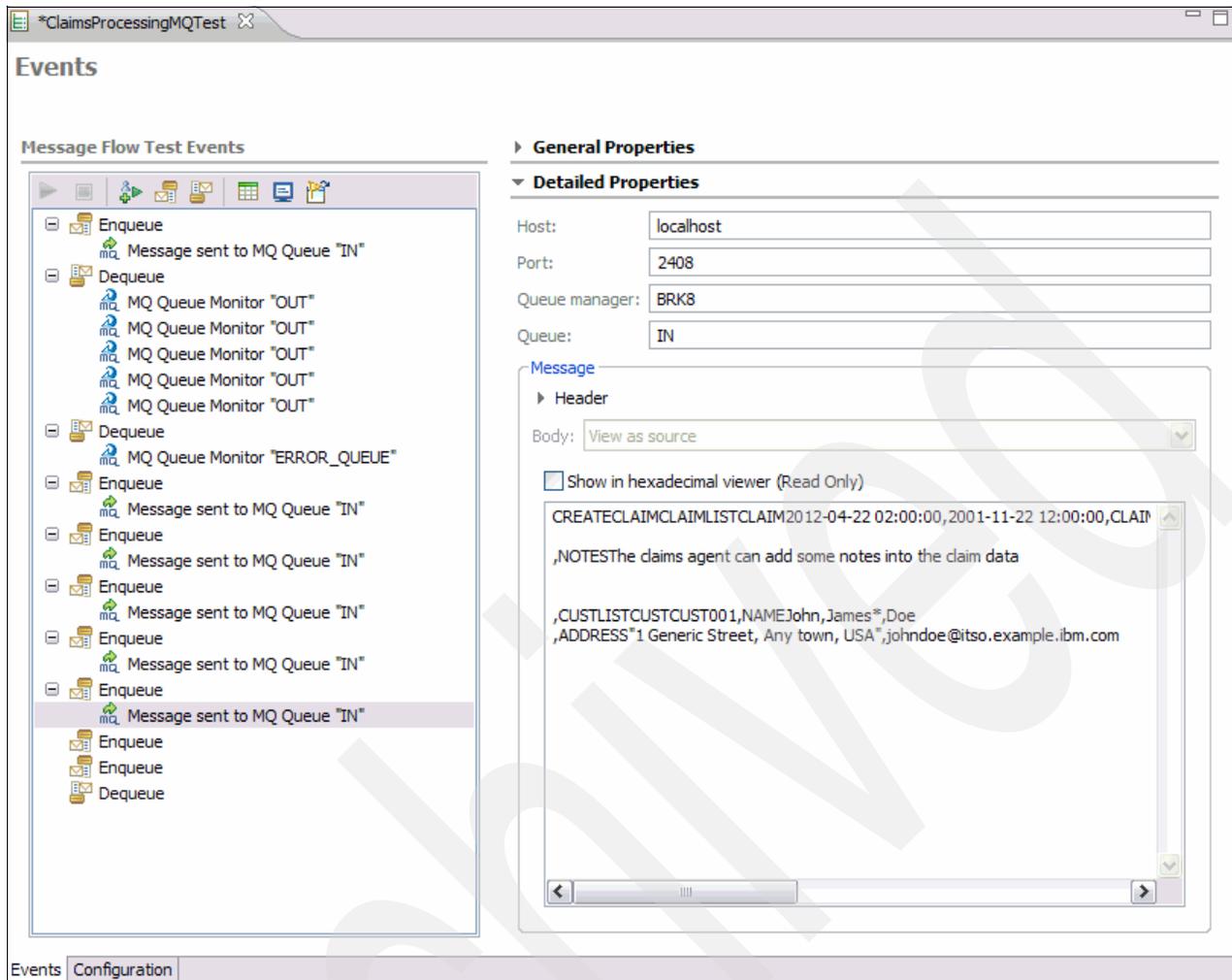


Figure 7-87 The CreateClaim Enqueue

3. Select the Dequeue for the OUT queue.
4. Click **Get Message**.

The response message shows a completion code of 0, indicating that the claim was created successfully, as shown in Figure 7-88 on page 376.

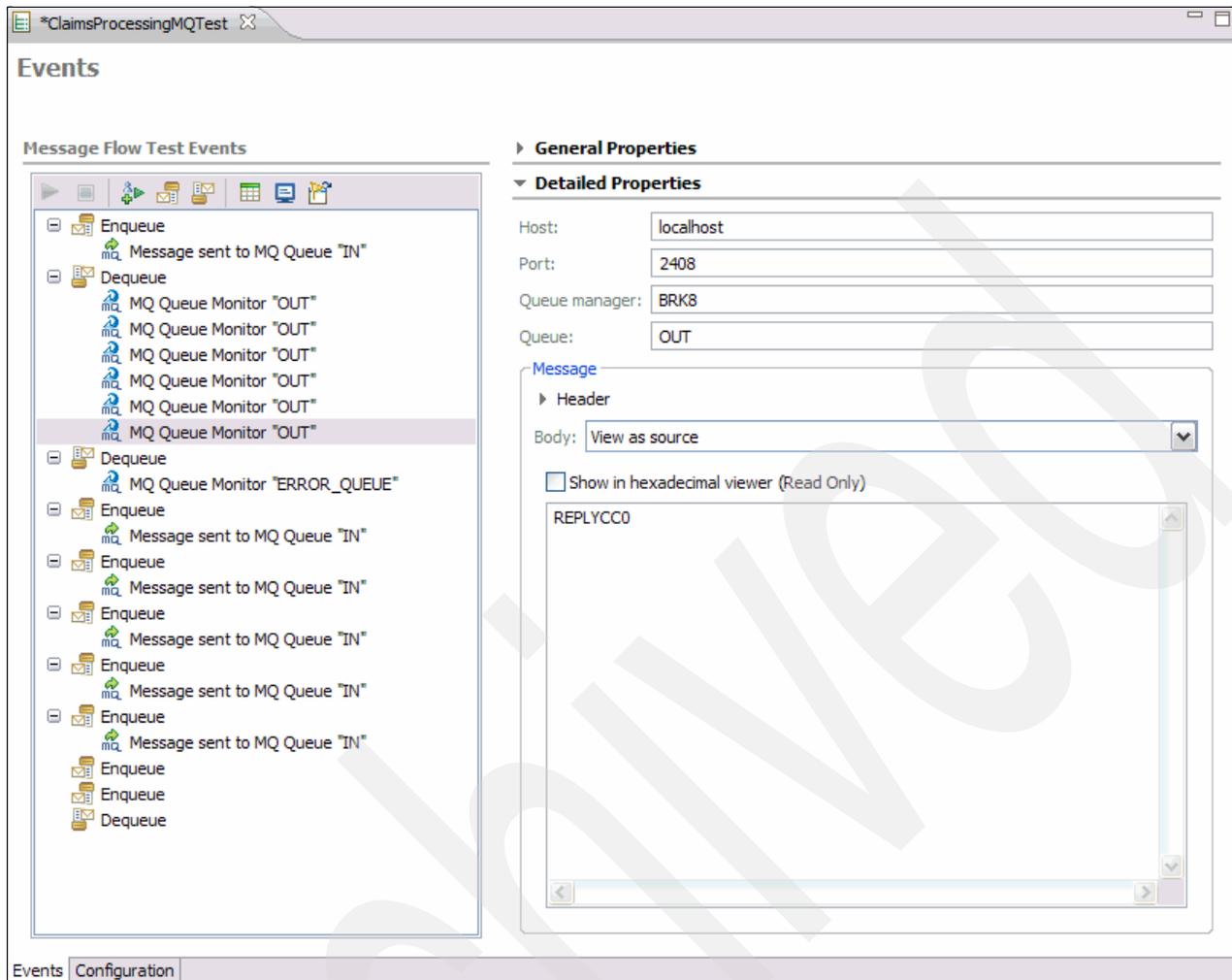


Figure 7-88 CreateClaim response

Test the ApproveClaim operation

Use the ApproveClaim operation to approve the CLAIM0001 claim. To test the ApproveClaim operation:

1. Select the **ApproveClaim** Enqueue.
2. Click **Send Message**.

A confirmation will appear below the Enqueue, showing that the message was put on the MQ queue, as shown in Figure 7-89 on page 377.

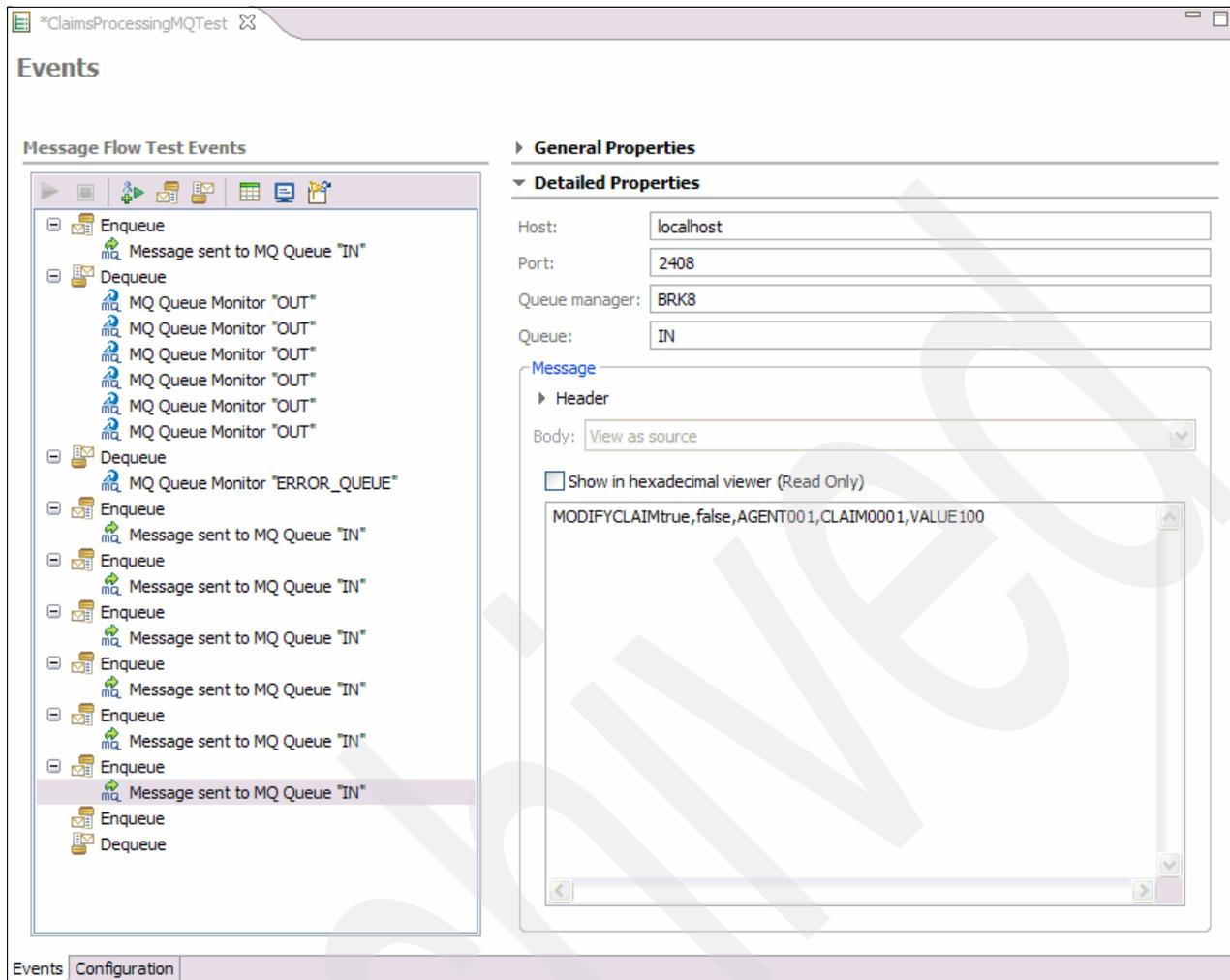


Figure 7-89 The ApproveClaim Enqueue

3. Select the Dequeue for the OUT queue.
4. Click **Get Message**.

The response will show a completion code of 0, indicating successful completion, as shown in Figure 7-90 on page 378.

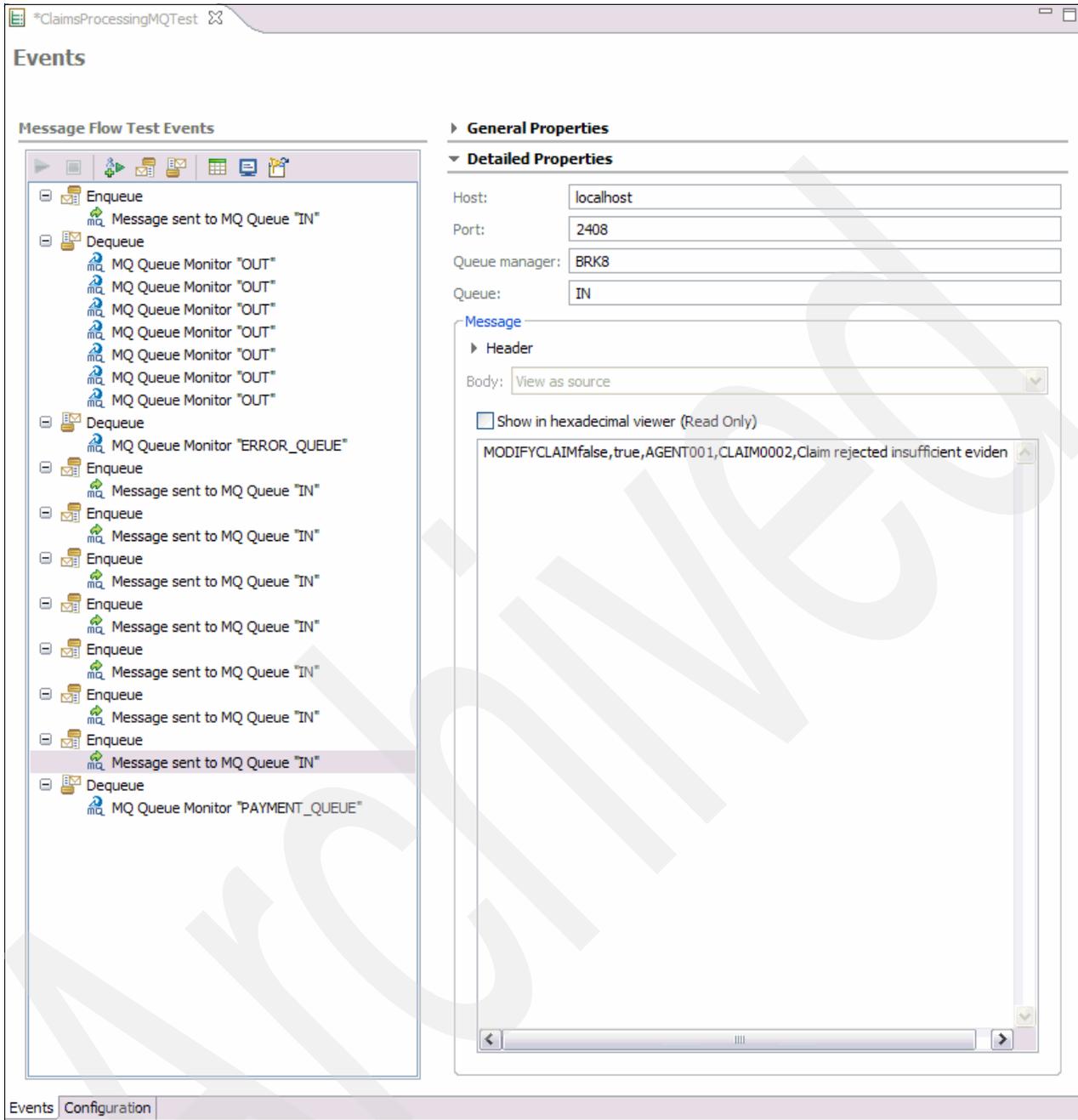


Figure 7-92 The RejectClaim Enqueue

3. Select the **Dequeue** for the OUT queue.
4. Click **Get Message**.

The response message shows a completion code of 0, as shown in Figure 7-93 on page 381.

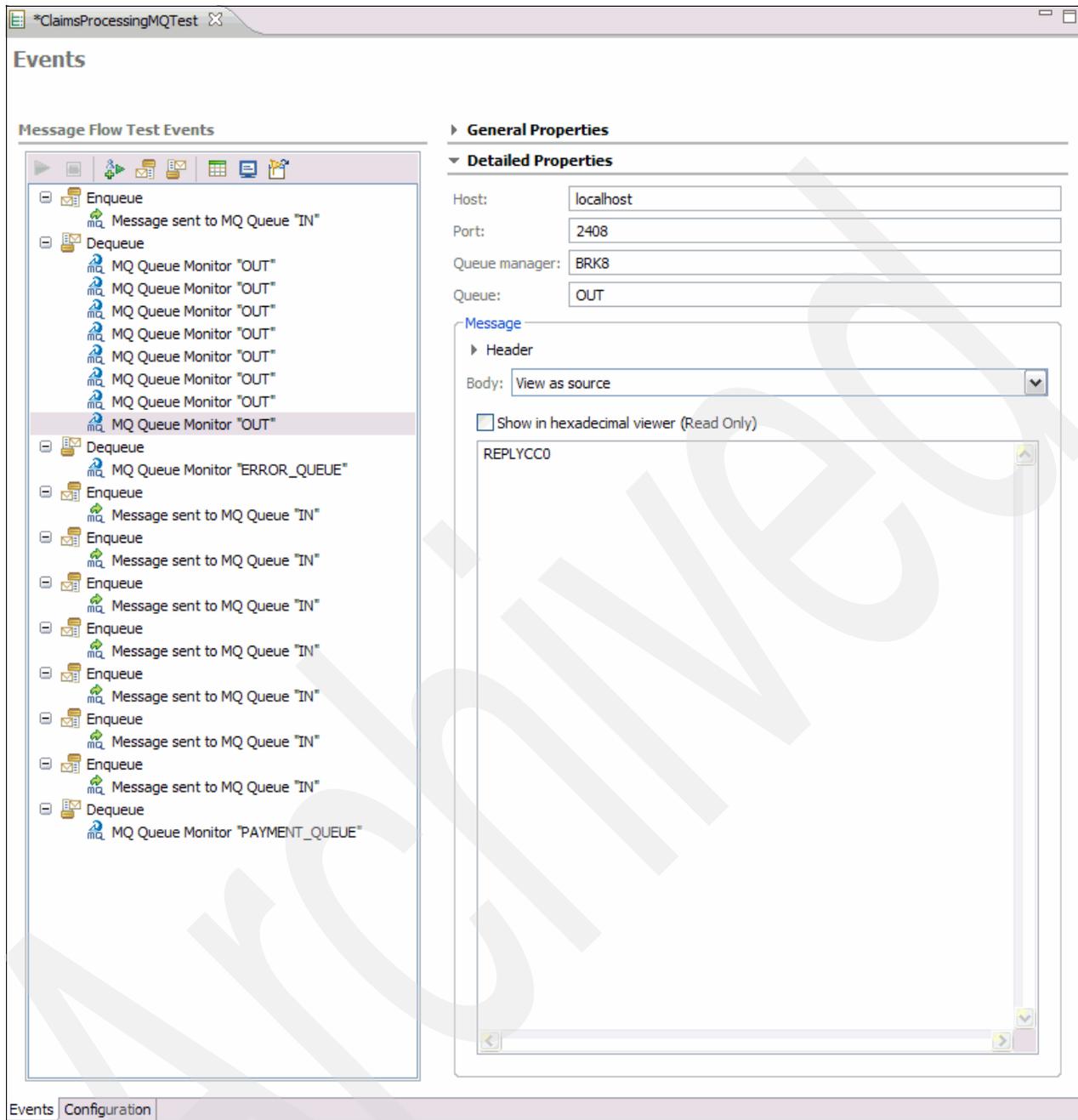


Figure 7-93 The RejectClaim response

7.16.2 Testing the SOAP Interface

Recall from the scenario overview that in addition to the MQ interface that connects to the insurance company's existing messaging infrastructure, the flow also hosts a SOAP request that forms the basis of a web service that allows customers to check the status of their claim online.

Additional materials: The ClaimsProcessingSOAPImported project in the additional materials for this book includes a Message Broker Test Client that can be used to test the SOAP interface.

To test the SOAP interface:

1. Double-click the **SOAPTest.mbttest** Message Broker Test Client to open it in the editor.
2. The SOAPTest Test Client already has a message flow Invoke operation defined that will send a SOAP message to the broker listener to view the status of the CLAIM0001 claim.
3. Select the **Configuration** tab, and set the deployment options to refer to your local broker, as shown in Figure 7-94.

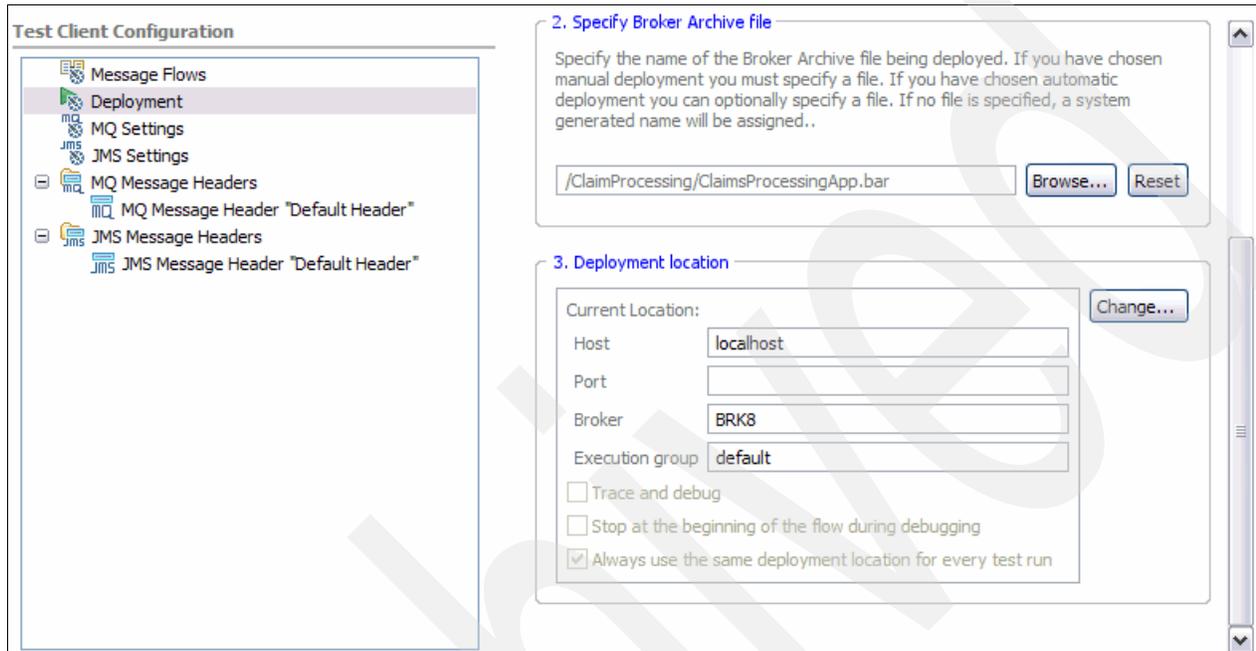


Figure 7-94 Configuring the SOAP Test Client

4. Return to the Events tab, as shown in Figure 7-95 on page 383. The message content is already filled in so that the ViewClaim operation looks up the claim details for CLAIM0001. Click **Send Message**.

The Test Client is set for manual deployment. However, you might see an information message saying that the deployment was metallized. Ignore this.

The Test Client sends the message to the `http://localhost:7800/ClaimsProcessing` end point and listens for a response.

When a response is received, you will see a SOAP message displayed in the right panel of the test client, as shown in Figure 7-95 on page 383.

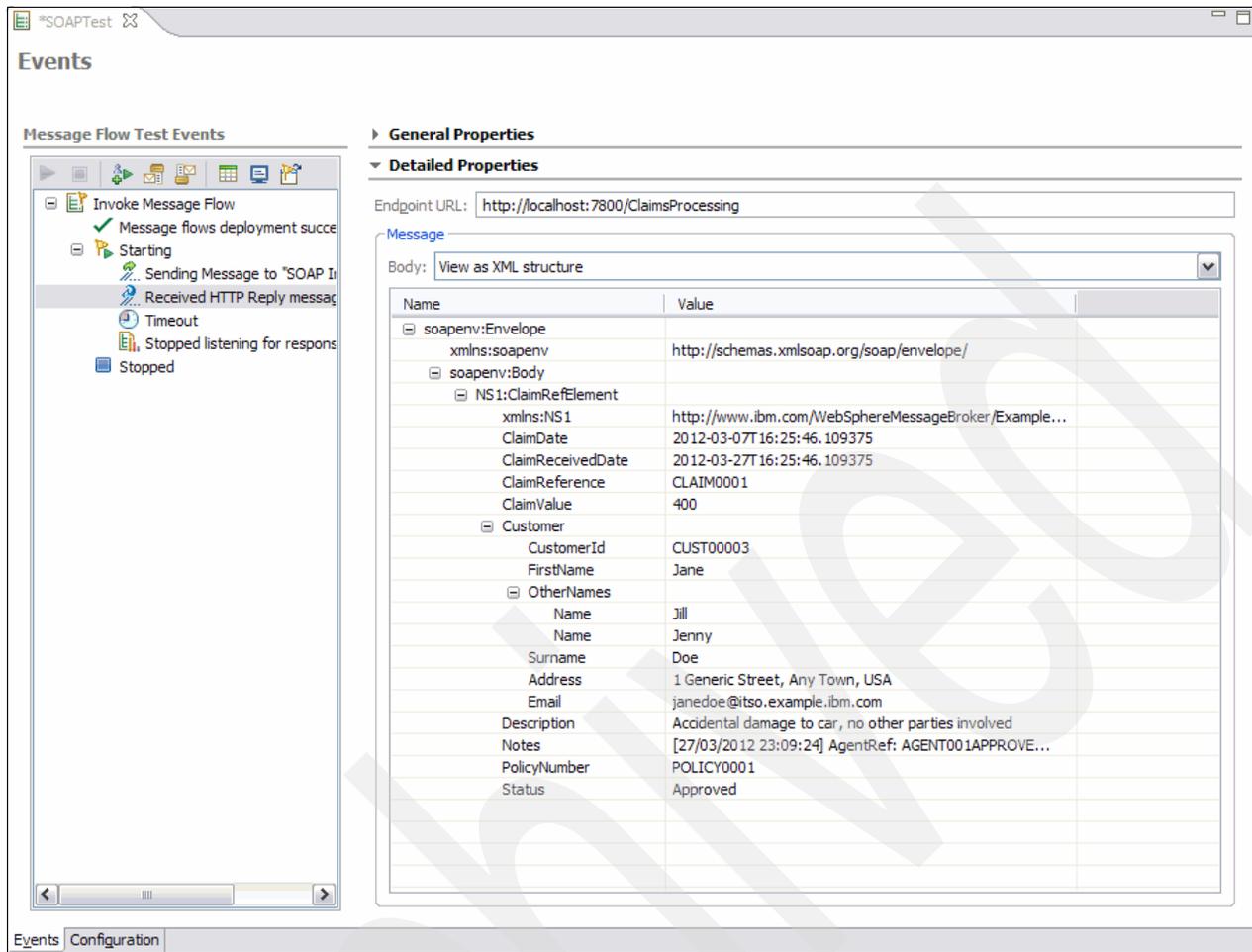


Figure 7-95 Output from the SOAP Test Client

7.17 Altering the scenario to use a SOAP over HTTP based binding

One of the main benefits of the WCF framework is that the choice of transport is decoupled from the application logic. It is therefore possible to update the service to use a web services based transport without having to change any of the code.

7.17.1 Altering the WCF Service to expose an http based binding

We use the WCF Service Configuration Editor to add a new service endpoint for the WCF service, which uses the basicHttpBinding to expose a new endpoint to the ClaimsProcessingWcfService.

To enable the basicHttpBinding endpoint:

1. In Visual Studio, launch the WCF Configuration Editor by selecting **Tools** → **WCF Service Configuration Editor**.
2. Select **File** → **Open** → **Config File**.

3. Navigate to the App.Config for the WCF service in the Open Dialog, and click the **OK** button.
 4. Expand the **Services** → **Ibm.Broker.Example.ClaimsProcessing.ClaimsProcessingWcfService** → **Endpoints** tree node.
 5. Select the Endpoint folder, right-click and select **New Service Endpoint**.
 6. In the Name field, enter `WebServicesEndpoint`.
 7. In the Address field, enter `/WebServices`.
 8. In the Binding field, enter `basicHttpBinding`.
 9. In the Contract field, enter `Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService`.
- The completed Service Endpoint definition will match the example in Figure 7-96.

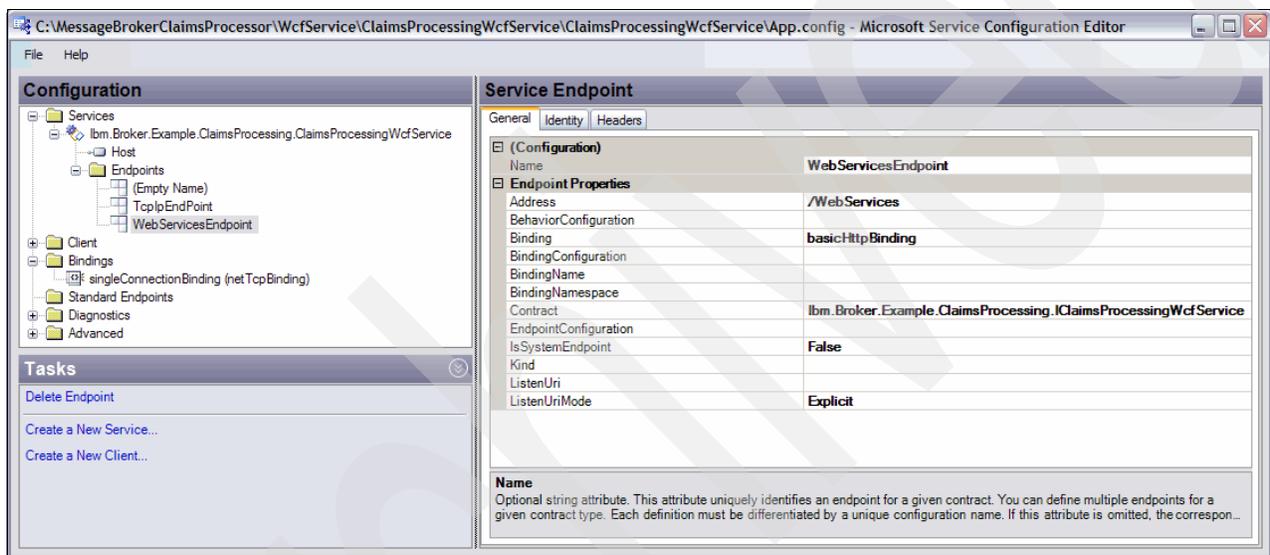


Figure 7-96 Adding the WebServices endpoint

10. Select **File** → **Save**.

After you save the file, open the App.Config in the code editor by double-clicking its icon in the Solution Explorer. You will see that the WCF Configuration Editor added the endpoint definition shown in Example 7-59 to your file.

Example 7-59 The WebServices endpoint definition

```
<endpoint address="/WebServices" binding="basicHttpBinding"
bindingConfiguration=""
name="WebServicesEndpoint"
contract="Ibm.Broker.Example.ClaimsProcessing.IClaimsProcessingWcfService" />
```

Recall that the endpoint address is relative to the base address of the application, so the new endpoint is available on the `http://localhost:8732/ClaimsProcessingWcfService/WebServices` URI.

To make the new endpoint available, relaunch the service using the **Debug** → **Start Without Debugging** menu item. Visual Studio automatically hosts the WCF and launches the WCF Test Client, as usual. Notice that there are two endpoints defined, the `TcpIpEndPoint` and the

WebServicesEndpoint. You can test the operations on the WebServicesEndpoint in exactly the same way as you did in 6.3.9, “Testing the WCF Service using the WCF Test Client” on page 235 for the TcpIpEndPoint. Figure 7-97 shows an example of invoking the ViewClaim operation from the WebServicesEndpoint.

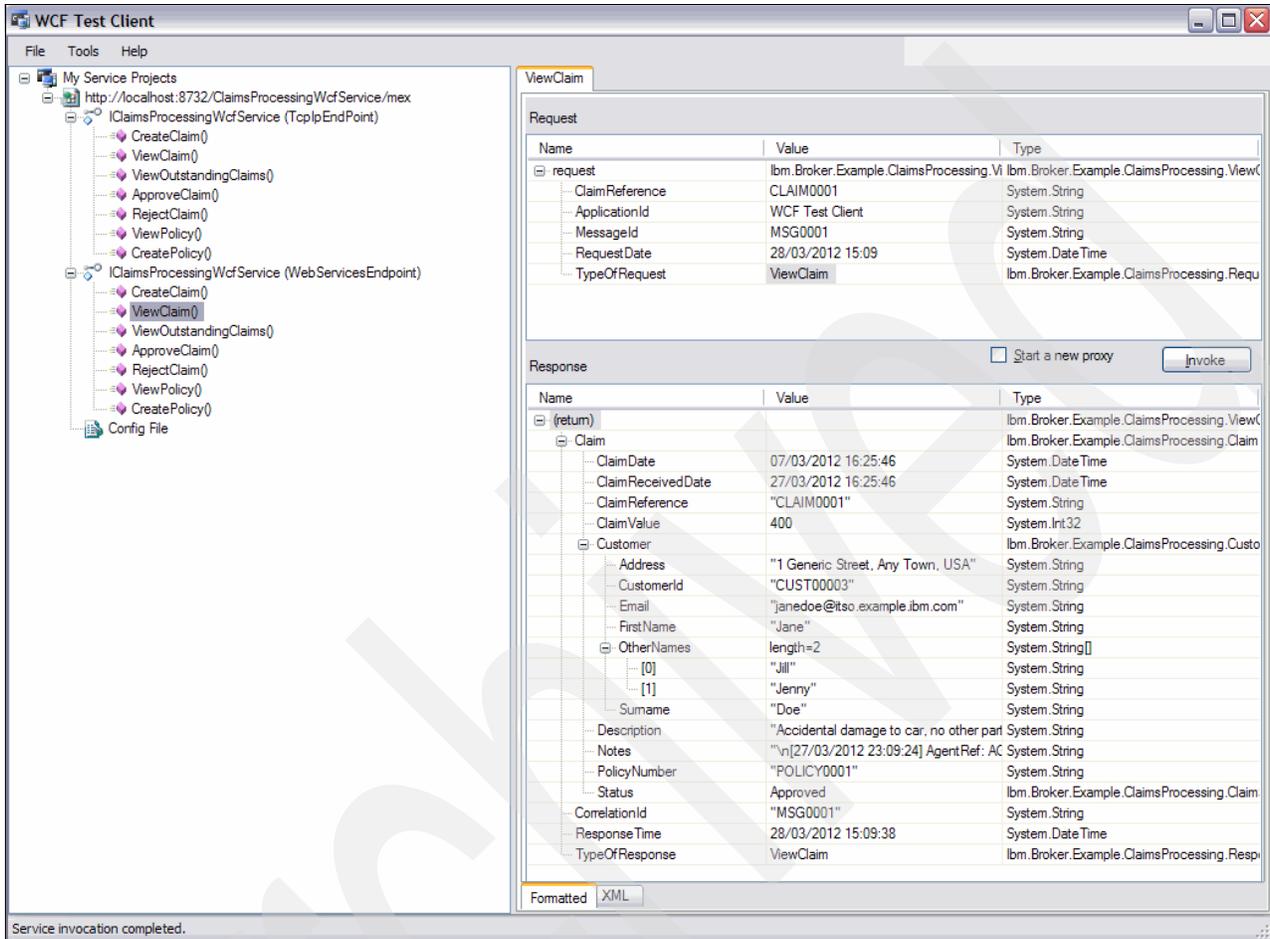


Figure 7-97 The WCF Test Client displays both endpoints

7.17.2 Altering a message flow to use an HTTP based binding

In this section, you alter a message flow in WebSphere Message Broker to access the service with an HTTP-based binding. To follow this section, you must have already modified the WCF service to use basicHttpBinding, as discussed in section 7.17.1, “Altering the WCF Service to expose an http based binding” on page 383.

To modify the message flow to use the basicHttpBinding, we must alter the configuration file for the assembly used by the broker. Locate the folder containing the ClaimsProcessingAssembly.dll, ClaimsProcessingBrokerAssembly.dll.config, and ClaimsProcessingBrokerAssembly.pdb files. Open the config file in a text editor. The content must match the example shown in Example 7-60.

Example 7-60 The ClaimsProcessingBrokerAssembly.dll.config file

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
```

```

    <netTcpBinding>
      <binding name="TcpIpEndPoint" closeTimeout="00:01:00"
openTimeout="00:01:00"
      receiveTimeout="00:10:00" sendTimeout="00:01:00" transactionFlow="false"
      transferMode="Buffered" transactionProtocol="OleTransactions"
      hostNameComparisonMode="StrongWildcard" listenBacklog="10"
      maxBufferPoolSize="524288" maxBufferSize="65536" maxConnections="10"
      maxReceivedMessageSize="65536">
    <readerQuotas maxDepth="32" maxStringContentLength="8192" maxArrayLength="16384"
      maxBytesPerRead="4096" maxNameTableCharCount="16384" />
      <reliableSession ordered="true" inactivityTimeout="00:10:00"
        enabled="false" />
      <security mode="Transport">
    <transport clientCredentialType="Windows" protectionLevel="EncryptAndSign" />
      <message clientCredentialType="Windows" />
      </security>
    </binding>
  </netTcpBinding>
</bindings>
<client>
  <endpoint address="net.tcp://localhost:8523/" binding="netTcpBinding"
bindingConfiguration="TcpIpEndPoint" contract="IClaimsProcessingWcfService"
name="TcpIpEndPoint">
    <identity>
      <userPrincipalName value="DAVICRIGW500\davicrig" />
    </identity>
  </endpoint>
</client>
</system.serviceModel>
</configuration>

```

Remove the <endpoint> element and all of its children from the <client> element, and replace it with the endpoint definition shown in Example 7-61, which refers to the WebServices endpoint.

Example 7-61 The client endpoint definition

```

<endpoint address="http://localhost:8732/ClaimsProcessingWcfService/WebServices"
  binding="basicHttpBinding"
bindingConfiguration="WebServicesEndpoint"
  contract="IClaimsProcessingWcfService" name="WebServicesEndpoint" />

```

The endpoint definition refers to the WebServicesEndpoint bindingConfiguration. This bindingConfiguration supplies configuration information for the binding. To add this element, delete the <netTcpBinding> element and all of its children and replace it with the bindingsConfiguration shown in Example 7-62.

Example 7-62 The configuration for the basicHttpBinding

```

<basicHttpBinding>
  <binding name="WebServicesEndpoint" closeTimeout="00:01:00"
openTimeout="00:01:00"
  receiveTimeout="00:10:00" sendTimeout="00:01:00" allowCookies="false"
  bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
  maxBufferSize="65536" maxBufferPoolSize="524288"
  maxReceivedMessageSize="65536"

```

```

    messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
    useDefaultWebProxy="true">
    <readerQuotas maxDepth="32" maxStringContentLength="8192"
        maxArrayLength="16384"
        maxBytesPerRead="4096" maxNameTableCharCount="16384" />
    <security mode="None">
    <transport clientCredentialType="None" proxyCredentialType="None"
    realm="" />
    <message clientCredentialType="UserName" algorithmSuite="Default" />
    </security>
    </binding>
</basicHttpBinding>

```

The completed ClaimsProcessingBrokerAssembly.dll.config file must match the example shown in Example 7-63.

Example 7-63 The completed ClaimsProcessingBrokerAssembly.dll.config file

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="WebServicesEndpoint" closeTimeout="00:01:00"
            openTimeout="00:01:00"
            receiveTimeout="00:10:00" sendTimeout="00:01:00" allowCookies="false"
            bypassProxyOnLocal="false" hostNameComparisonMode="StrongWildcard"
            maxBufferSize="65536" maxBufferPoolSize="524288"
            maxReceivedMessageSize="65536"
            messageEncoding="Text" textEncoding="utf-8" transferMode="Buffered"
            useDefaultWebProxy="true">
          <readerQuotas maxDepth="32" maxStringContentLength="8192"
              maxArrayLength="16384"
              maxBytesPerRead="4096" maxNameTableCharCount="16384" />
          <security mode="None">
            <transport clientCredentialType="None" proxyCredentialType="None"
            realm="" />
            <message clientCredentialType="UserName" algorithmSuite="Default" />
          </security>
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:8732/ClaimsProcessingWcfService/WebServices"
          binding="basicHttpBinding" bindingConfiguration="WebServicesEndpoint"
          contract="IClaimsProcessingWcfService" name="WebServicesEndpoint" />
    </client>
  </system.serviceModel>
</configuration>

```

When you save the file you can see a BIP7451 message in the Event Viewer indicating that a host swap deploy was successful, as shown in Figure 7-98 on page 388.

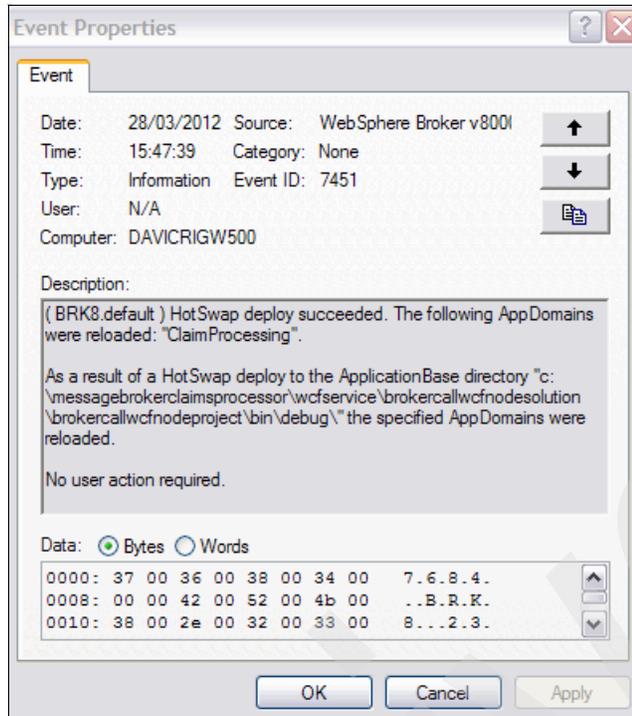


Figure 7-98 Changing the config file causes Broker to perform a hot swap deploy

The changes are picked up by WebSphere Message Broker immediately without needing to redeploy the flow or change any of the code. You can now test the message flow using the Message Broker Test Client using the same procedure that you used in section 7.16.1, “Testing the MQ interface” on page 362.

In Figure 7-99 on page 389, we used the ViewClaim enqueue to send a ViewClaimRequest message to the flow.

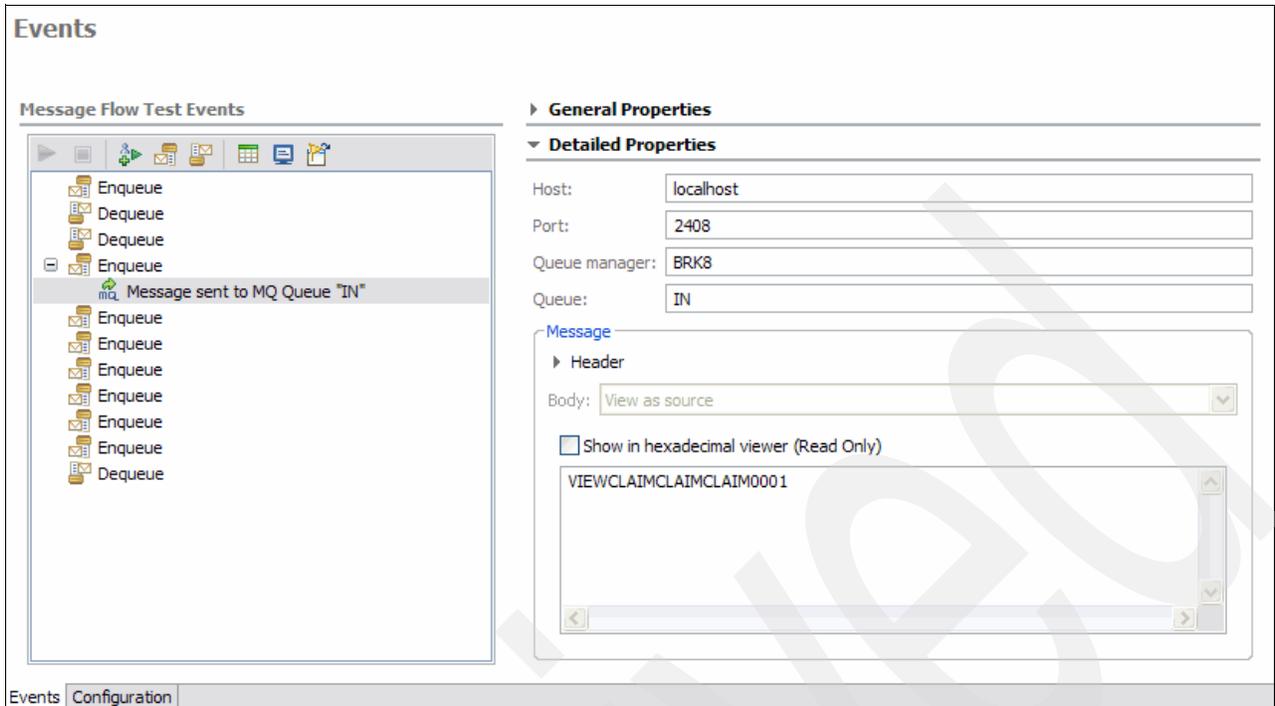


Figure 7-99 Sending a ViewClaimRequest to the message flow

The response from this message can be viewed using the Dequeue for the OUT message, as shown in Figure 7-100.

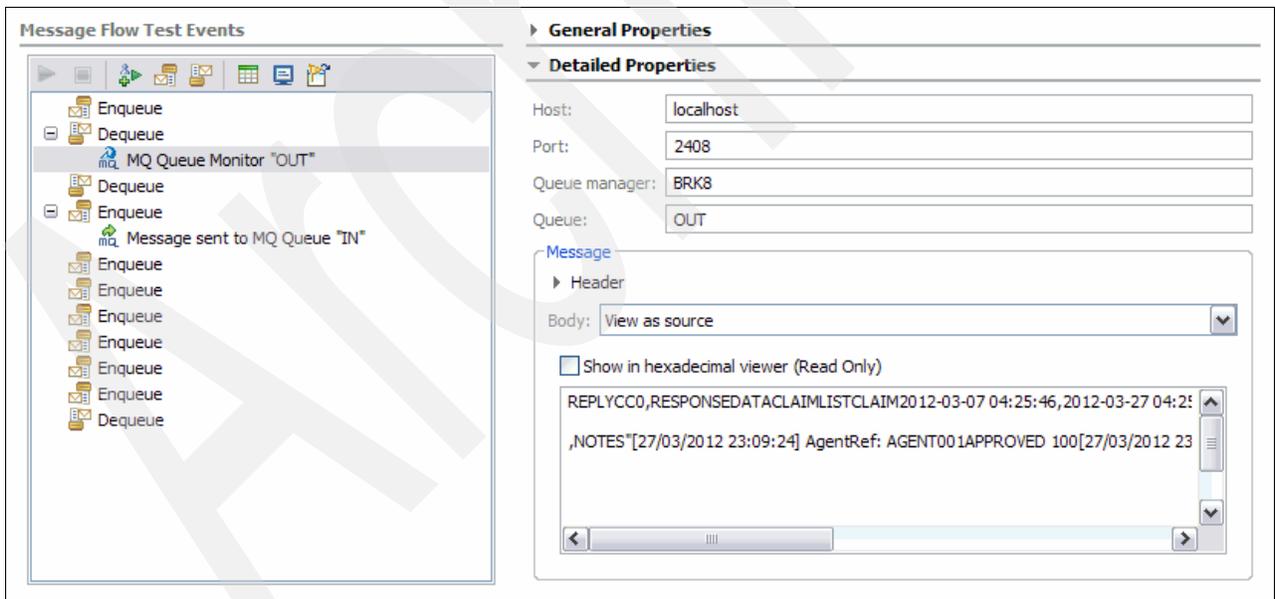


Figure 7-100 Using the Dequeue for the OUT queue to confirm the message was processed successfully

The message flow is now successfully using the basicHttpBinding binding to make WCF calls to the service.

Archived

Integrating file transfer with WebSphere MQ FTE into the message flow

The scenario in this chapter illustrates the use of WebSphere Message Broker to intercept files routed by WebSphere MQ File Transfer Edition, analyze and split that file into multiple files, and route those files to their appropriate destination. It also illustrates processing replies from the processing systems and collecting those replies into one message to send back to the originating system.

This scenario illustrates receiving files from WebSphere MQ File Transfer Edition into a message flow, inspecting and splitting those files, and then routing the files to multiple destinations and aggregating the replies. It also illustrates how to access .NET code directly from ESQL and how to use the DFDL editor to model message data format.

This chapter contains the following topics:

- ▶ Scenario overview
- ▶ Overview of the WebSphere MQ File Transfer edition
- ▶ Preparing the broker environment for this scenario
- ▶ Applications emulated in this scenario
- ▶ Creating the main message flow
- ▶ Running the scenario
- ▶ Extending the scenario

Additional materials: You can download the WebSphere Message Broker project interchange file and the .NET class code for this scenario from the IBM Redbooks publication web site. See Appendix A, “Additional material” on page 485 for more information.

8.1 Scenario overview

An investment firm has a number of branches that handle the day-to-day stock transactions of their customers. As each transaction is executed, a record of the transaction is created. At the end of the day, a summary report of the transactions is created and the branch stores the transaction records (referred to as operations) and the summary report into one consolidated file. The consolidated file is then transferred from the branch over WebSphere MQ FTE to a message flow for processing at the firm's head office.

The message flow receives the consolidated file and parses the entries in the file. The summary report record in the file is sent as a message to the Report Data Warehouse (RDW) over WebSphere MQ. The transaction records in the file are handled by a .NET assembly and sent to the Volume Analysis System (VAS) application for processing. As the RDW and VAS process the records sent to them, they send a reply back to the message flow. These replies are aggregated into one reply for the branch so that it can verify that each transaction was handled.

In addition to routing the files, the message flow tracks the status of the delivery and processing by recording the following events:

- ▶ A file is received by the message flow
- ▶ The messages are sent to their destination
- ▶ Each destination application receives and processes a message
- ▶ All messages from the original file have been received and processed by their destination

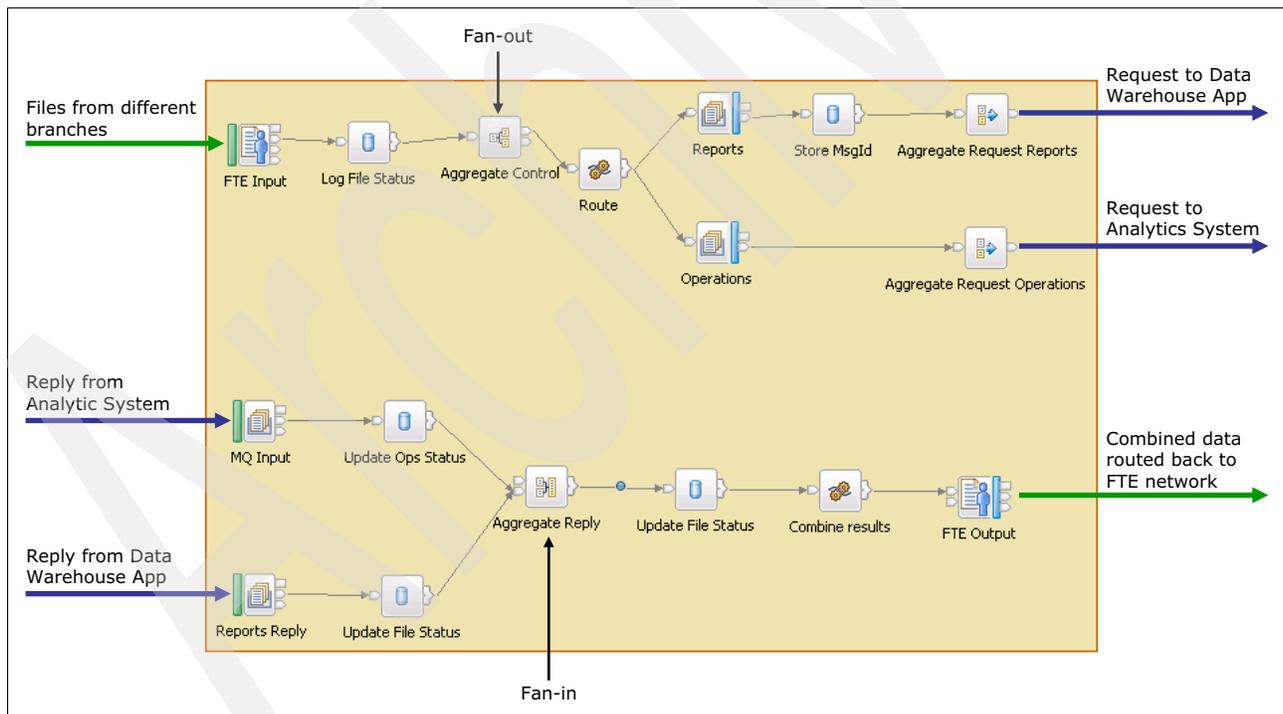


Figure 8-1 WebSphere MQ File Transfer Edition scenario overview

This scenario assumes that you have the following prerequisites:

- ▶ WebSphere MQ File Transfer Edition V7.0
- ▶ WebSphere Message Broker version 8.0.0.0 or higher installed
- ▶ WebSphere Message Broker Toolkit version 8.0.0.0 or higher installed

- ▶ A message broker defined on the local machine with a corresponding WebSphere MQ queue manager
- ▶ An SMTP email server running on the local host or access to an SMTP server for sending outbound email
- ▶ Microsoft Visual Studio 2010 installed

8.2 Overview of the WebSphere MQ File Transfer edition

WebSphere MQ File Transfer Edition V7.0 (FTE) delivers a reliable, managed file transfer solution for moving files between IT systems without the need for programming. Files that are larger than the maximum individual WebSphere MQ message size can be moved. A log of file movements demonstrates that business data in files is transferred with integrity from a source file system to a destination file system.

Using the WebSphere MQ File Transfer Edition nodes offers seamless integration with your existing WebSphere MQ File Transfer Edition network. Figure 8-2 shows a typical WebSphere MQ File Transfer Edition network.

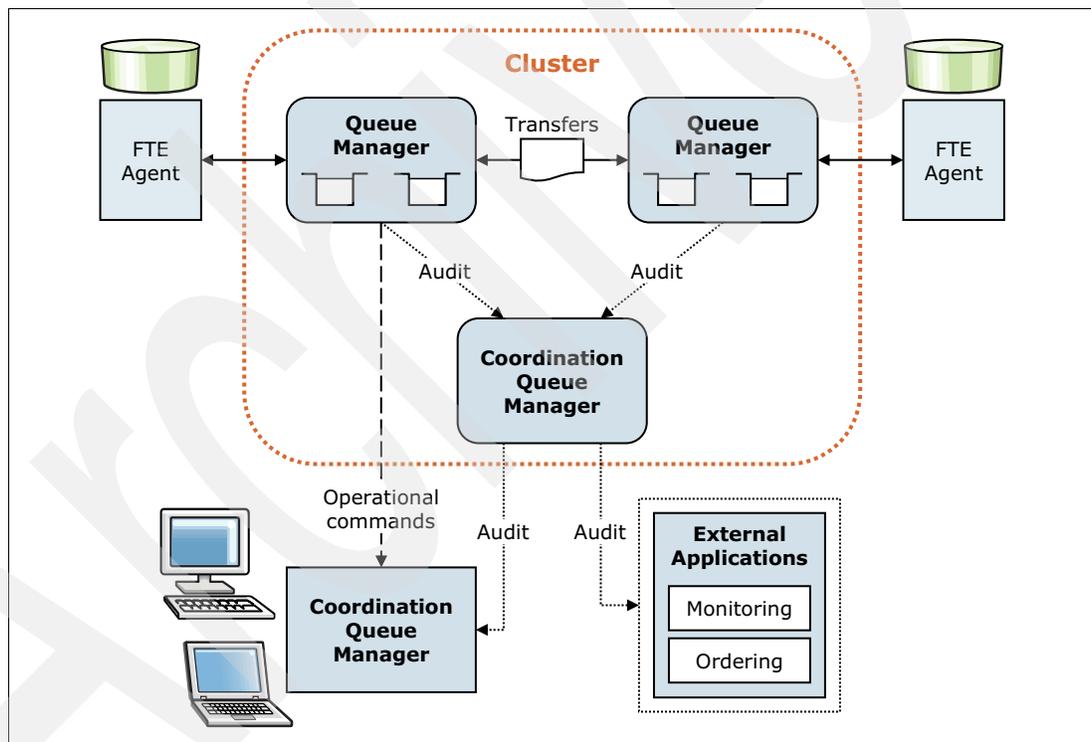


Figure 8-2 WebSphere MQ File Transfer Edition network

The main components and concepts of WebSphere MQ File Transfer Edition are:

- ▶ **Agent:** A process that forms the endpoint of a transfer (source or destination). An agent is a WebSphere MQ application and is connected to one queue manager. Many agents can be connected to the same queue manager. Operational commands can be sent to an agent (for example, to request a transfer) by putting an XML message on a particular queue on the agent's queue manager.
- ▶ **Coordination queue manager:** One queue manager in the topology takes on the responsibility of the coordination queue manager. All agents register with the coordination

queue manager and also send audit information about each transfer. The coordination queue manager is responsible for publishing that audit information to external monitoring applications.

- ▶ **Transfer:** A transfer is a movement of one or more files from one agent to another. The transfer goes directly from one agent's queue manager to the other agent's queue manager, not through the coordination queue manager. The transfer takes place even if the coordination queue manager is not running.
- ▶ **Transfer log:** The WebSphere MQ Explorer plug-in tool includes a Transfer Log view that subscribes to the coordination queue manager for the audit information. The view displays information about every transfer that occurs in a given topology.

Most of the file transfers occurred in a peer-to-peer way, although there is a feature to use agents in one-to-many ways, as shown in this website:

http://www.ibm.com/developerworks/websphere/library/techarticles/1103_cullen/1103_cullen.html

But to make complex tasks in the middle of file transfers, it is better to use WebSphere Message Broker.

8.2.1 Using WebSphere Message Broker as a bridge for FTE networks

One of the possible approaches when constructing an FTE network when using the broker is to have the broker as a central agent on the coordination queue manager and have many stand-alone FTE agents on other machines. See Figure 8-3.

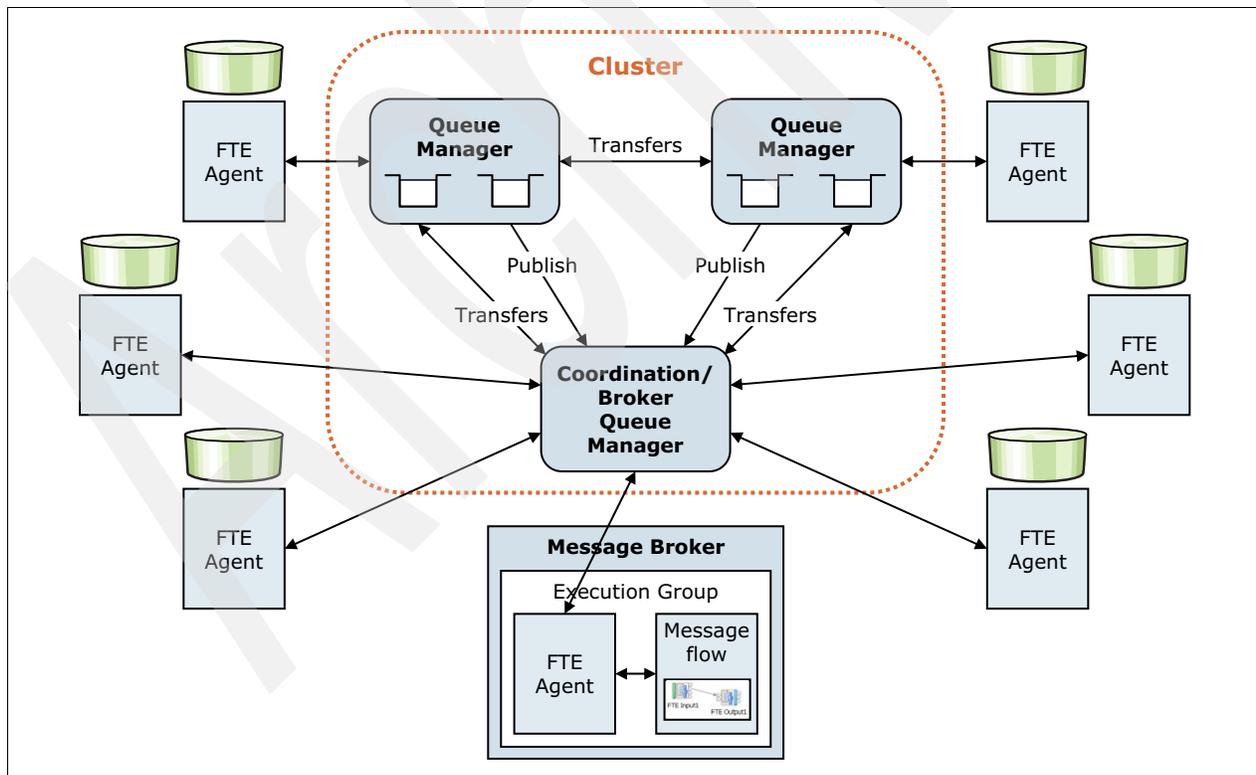


Figure 8-3 Using the broker as a bridge for FTE networks

With this design the stand-alone agents can either directly transfer their files to other agents or transfer them to the central broker agent, which can do any required processing before transferring it on to another agent or, if required, sending it to another system using a different transport protocol, such as HTTP, FTP or WCF.NET. It also allows for transformation of data before being sent to an end destination.

In this scenario, we demonstrate how you can use the broker with a file transfer to execute essential processing steps and route message between separate FTE agents.

8.3 Preparing the broker environment for this scenario

Before any other resources are configured for this scenario, ensure that there is an MQ Manager and a broker created at the local machine.

In this chapter, we use following naming conventions:

- ▶ QM_ITSO is the name of the queue manager used both for the broker and a stand-alone MQ FTE agent from the Branch application emulator. See 8.4.1, “Branch application (FTE)” on page 403.
- ▶ BRK_ITSO is the name of the broker.
- ▶ ITSO is the name of the default execution group.
- ▶ MOCKS is the name of the execution group for mock flows of the Report Data Warehouse and Volume Analysis System applications. See 8.4.2, “Report Data Warehouse” on page 409 and 8.4.3, “Volume Analysis System” on page 414 respectively.

Use the following procedure to create the WebSphere Message Broker resources:

1. In the Message Broker Toolkit, right-click **Brokers** in the Brokers tab. Select **New** → **Local Broker**.
2. In the New Local Broker dialog, enter:
 - BRK_ITSO as the name of the new broker
 - QM_ITSO as the queue manager name for the broker
 - LocalSystem as the user name
 - ITSO as the name of the default execution group, as shown in Figure 8-4 on page 396

Click **Finish**.

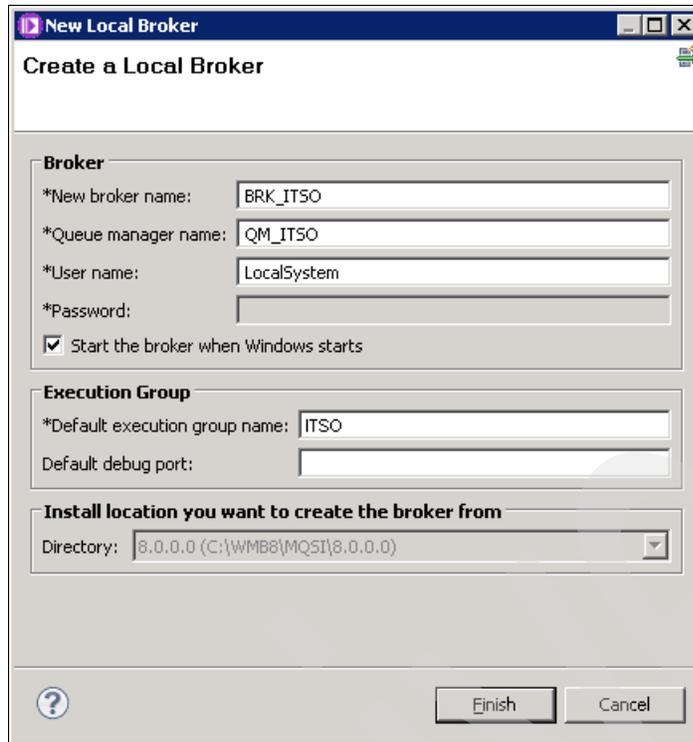


Figure 8-4 Preparing the broker for the FTE scenario

The Message Broker Toolkit creates a WebSphere MQ manager, broker, and the execution group.

3. In the Brokers view, right-click the **BRK_ITSO** broker, and select **New Execution Group**. Enter **MOCKS** as a name for new execution group, as shown in Figure 8-5, and then click **OK**.



Figure 8-5 Creating a new execution group

Other resources: There are a set of other resources that are used within this scenario. These resources are explained and created later on, together with respective parts of the scenario.

8.3.1 Creating the database

This scenario uses a Microsoft SQL Server 2008 database. This section provides information about creating the database.

Configuring SQL Server network connectivity services

Use the following procedure to configure the SQL Server network connectivity services:

1. Open the SQL Configuration Manager by clicking **Start** → **All Programs** → **Microsoft SQL Server 2008** → **Configuration Tools** → **SQL Server Configuration Manager**.
2. Select **SQL Server Network Configuration** → **Protocols for SQLEXPRESS** in the left column, and ensure that the Named Pipes and TCP/IP protocols are both enabled (Figure 8-6). If they are not, enable them by right-clicking the protocol and selecting **Enable**.

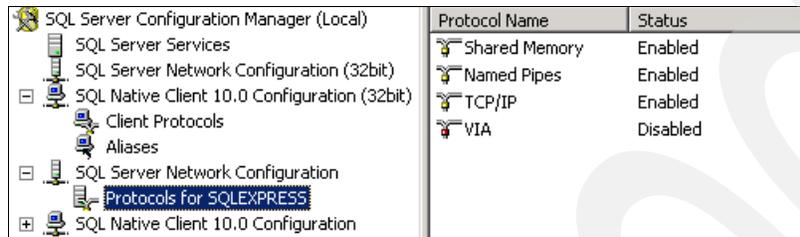


Figure 8-6 SQL Server Network Configuration window

3. Select **SQL Server Services** in the same window, and ensure that the browser is running (Figure 8-7).

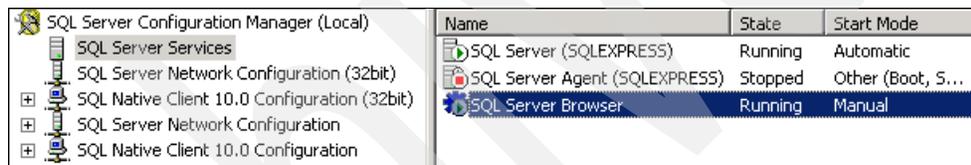


Figure 8-7 Run the SQL Server Browser service

If the browser is not running, start it by right-clicking it and selecting **Start**. If you cannot start it in this way, use the following steps:

- a. Right-click the browser, and select **Properties**.
- b. Select the **Service** tab in the SQL Server Browser Properties dialog, and select **Start Mode**. See Figure 8-8 on page 398.
- c. Open the drop-down box, and select **Manual**. Click **OK**.
- d. Right-click the browser, and select **Start**.

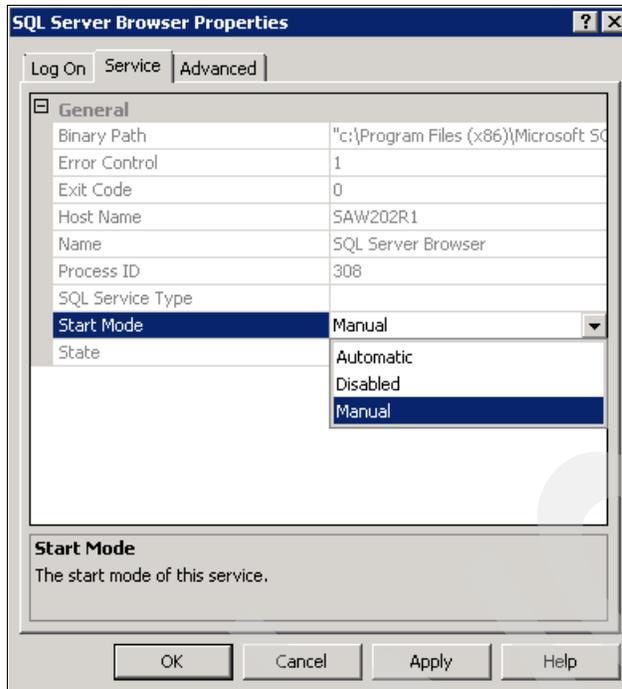


Figure 8-8 Change the start mode of SQL Server Browser service

Creating an ITSO database server instance

Tooling: In this and the next section, we use Visual Studio to administer SQL Server.

Use the following procedure to create a database server instance in Visual Studio:

1. Open Visual Studio by clicking **Start** → **All Programs** → **Microsoft Visual Studio 2010** → **Microsoft Visual Studio 2010**.
2. Click **View** → **Server Explorer** to open the Server Explorer view.
3. Right-click **Data Connection**, and select **Add Connection**.
4. In the Choose Data Source window, select **Microsoft SQL Server**, and click **Continue**.
5. In the Add Connection window:
 - a. Select **Microsoft SQL Server (SqlClient)** in the Data source field.
 - b. Select the server in the Server name drop down.
 - c. Select the log on method, and enter your credential if required.
 - d. In the Connect to a database section, enter the name for the new database in the **Select or enter a database name** field. In this case, enter ITSO as the database name.
 - e. Click **OK**.

Creating a FILE_STATUS table and indexes

After creating the database, you create a FILE_STATUS table there. This table is used within the scenario for logging purposes. You can use the script in Example 8-1.

Example 8-1 Sample script to create a FILE_STATUS table

```
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[FILE_STATUS]') AND type in (N'U'))
```

```

BEGIN
CREATE TABLE [dbo].[FILE_STATUS](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [file_name] [nvarchar](64) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [parts_in_total] [int] NOT NULL,
    [parts_processed] [int] NOT NULL,
    [completed] [bit] NOT NULL,
    [record_time] [datetime] NOT NULL,
    [last_modified] [datetime] NOT NULL,
    [report_msg_id] [nchar](51) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    CONSTRAINT [PK_FILE_STATUS] PRIMARY KEY CLUSTERED
(
    [id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
)
END
GO
IF NOT EXISTS (SELECT * FROM sys.indexes WHERE object_id =
OBJECT_ID(N'[dbo].[FILE_STATUS]') AND name = N'IX_FILE_STATUS')
CREATE UNIQUE NONCLUSTERED INDEX [IX_FILE_STATUS] ON [dbo].[FILE_STATUS]
(
    [report_msg_id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF,
IGNORE_DUP_KEY = OFF, DROP_EXISTING = OFF, ONLINE = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON)
GO

```

Note: In our scenario, we use SQL Server as an RBMS, so all scripts are tested for that particular product. For other RDBMS, the syntax varies.

The description of columns is as follows:

- ▶ **Id:** The primary key, auto generated.
- ▶ **File_name:** The name of the transferred file containing the report.
- ▶ **Parts_in_total:** The number of operations contained plus one summary report within the file.
- ▶ **Parts_processed:** The number of responses received from integrated systems, such as RDW and VAS.
- ▶ **Completed:** The flag that became true when all of the parts of the file are processed by RDW and VAS.
- ▶ **Record_time:** The time when the broker started to process this report.
- ▶ **Last_modified:** The time of the last modification of this row.
- ▶ **Report_msg_id:** Used as a temporary storage to correlate responses from RDW with rows in this table.

FILE_STATUS script: In addition to creating a table, this script also creates a primary key and index to speed up access to this table through the `id` and `report_id` columns. This is a good approach to think about indexes for every new table from the beginning, even if you only create this table for sample purposes.

Use the following procedure to create and execute a script:

1. To execute a script, open a new SQL Script Editor window by selecting **Data** → **Transact SQL-Editor** → **New Query connection**.

ITSO database instance: Make sure you are connected to the correct database instance. In our case, it is ITSO.

2. Enter the parameters in the Connect to Server dialog to connect to your database. Click the **Options** button at the Connect to Server window, and select ITSO as the value for Connect to the database. See Figure 8-9.

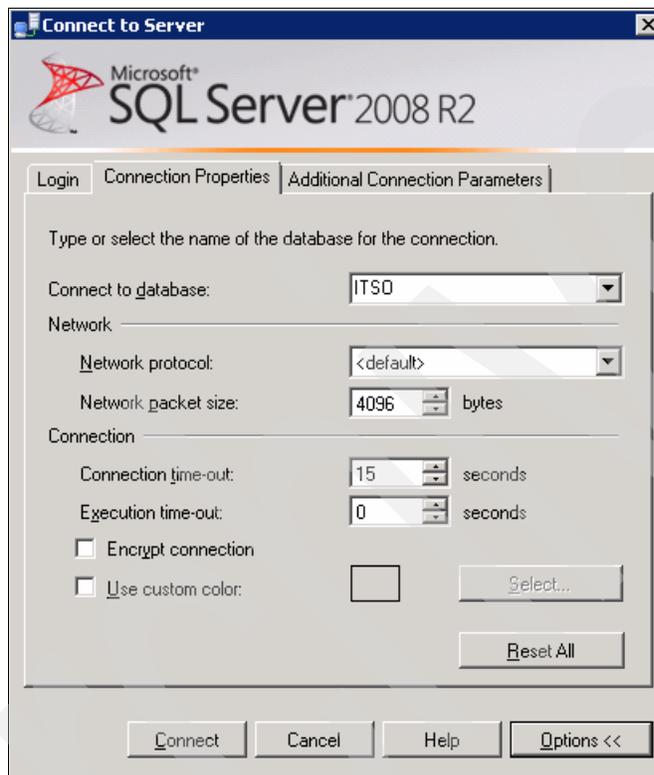


Figure 8-9 Connect to database ITSO

3. Insert the text of the script from Example 8-1 on page 398 into the script editor, and execute it by selecting **Data** → **Transact-SQL Editor** → **Execute SQL**. Verify the command output in the Messages tab to make sure that the table was created successfully. See Figure 8-10 on page 401.

```

IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id = OBJECT_ID(N'[dbo].[FILE_STATUS]') AND
BEGIN
CREATE TABLE [dbo].[FILE_STATUS] (
[id] [int] IDENTITY(1,1) NOT NULL,
[file_name] [nvarchar] (64) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
[parts_in_total] [int] NOT NULL,
[parts_processed] [int] NOT NULL,
[completed] [bit] NOT NULL,
[record_time] [datetime] NOT NULL,
[last_modified] [datetime] NOT NULL,
[report_msg_id] [nchar] (51) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
CONSTRAINT [PK_FILE_STATUS] PRIMARY KEY CLUSTERED
(
[id] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS =
)
END
GO
IF NOT EXISTS (SELECT * FROM sys.indexes WHERE object_id = OBJECT_ID(N'[dbo].[FILE_STATUS]') AND
CREATE UNIQUE NONCLUSTERED INDEX [IX_FILE_STATUS] ON [dbo].[FILE_STATUS]
(
[report_msg_id] ASC
) WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, SORT_IN_TEMPDB = OFF, IGNORE_DUP_KEY = OFF
GO

```

100 %

Messages

Command(s) completed successfully.

100 %

Figure 8-10 Script for FILE_STATUS table after executing

8.3.2 Configuring the ODBC data source

To use a database from Message Broker, you should create a System Data Source (DS) using Administrative tools managing snapshot. Use the following procedure to configure the ODBC data source.

1. Select **Start** → **Administrative Tools** → **Data Sources (ODBC)**.
2. Select the **System DSN** tab, and click **Add**.
3. In the ODBC DataSource Administrator dialog, select **SQL Server Native Client 10.0** as a database driver, and then click **Finish**.
4. Enter FTE_DS as the name, provide a description, and select the server, as shown in Figure 8-11 on page 402. Click **Next**.

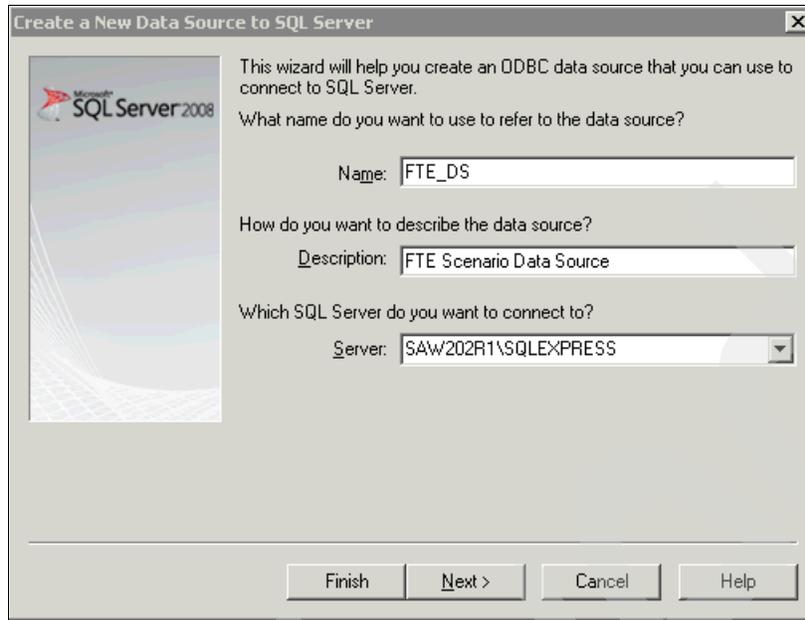


Figure 8-11 Creating new ODBC Data source

5. Enter ITS0 for SPN (optional), as shown in Figure 8-12. Click **Next**.

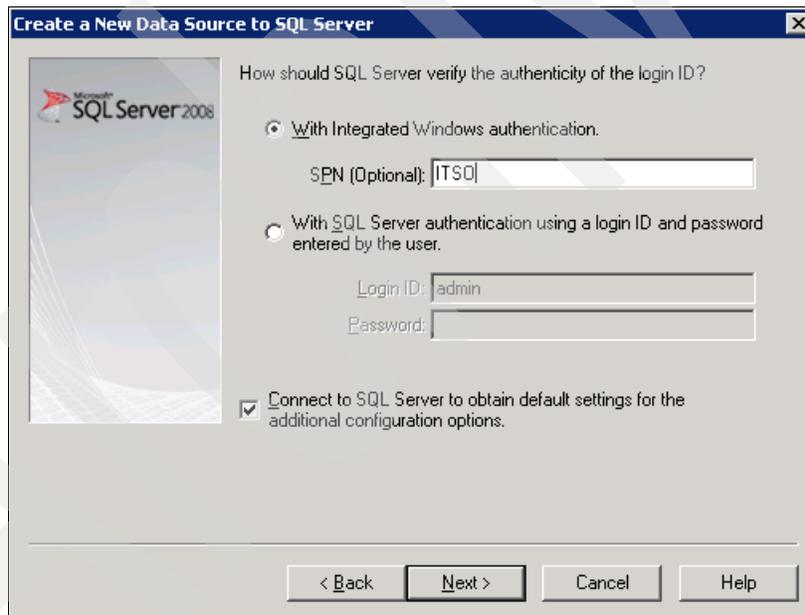


Figure 8-12 Creating new ODBC Data source (continued)

6. Click **Next** in the next two screens, and then click **Finish** to complete the ODBC DS creation process.

8.3.3 Preparing the file system structure

Create a C:\ITS0 folder on the root of your local file system for the purpose of creating transfers and testing main message flows.

Create the following subdirectories in the folder:

- ▶ \agent_a_in: This is a folder for files to be polled by the ITSO_A FTE agent.
- ▶ \agent_brk_out: This is where the broker agent BRK_ITSO.ITSO puts received files before processing them through FTE Input node.
- ▶ \agent_b_out: This folder is used by ITSO_B agent as a destination for transfers from the broker.

8.4 Applications emulated in this scenario

This section describes how the business applications listed in section 8.1, “Scenario overview” on page 392 are emulated from the technology perspective. You configure two FTE Server Agents and create two message flows to emulate Branch, Report Data Warehouse, and Volume Analysis System applications. You also create a simple .NET library to demonstrate how to call .NET directly from ESQL code.

8.4.1 Branch application (FTE)

This application emulator works outside of the WebSphere Message Broker environment. It is emulated by a stand-alone FTE server agent, as shown in Figure 8-13.

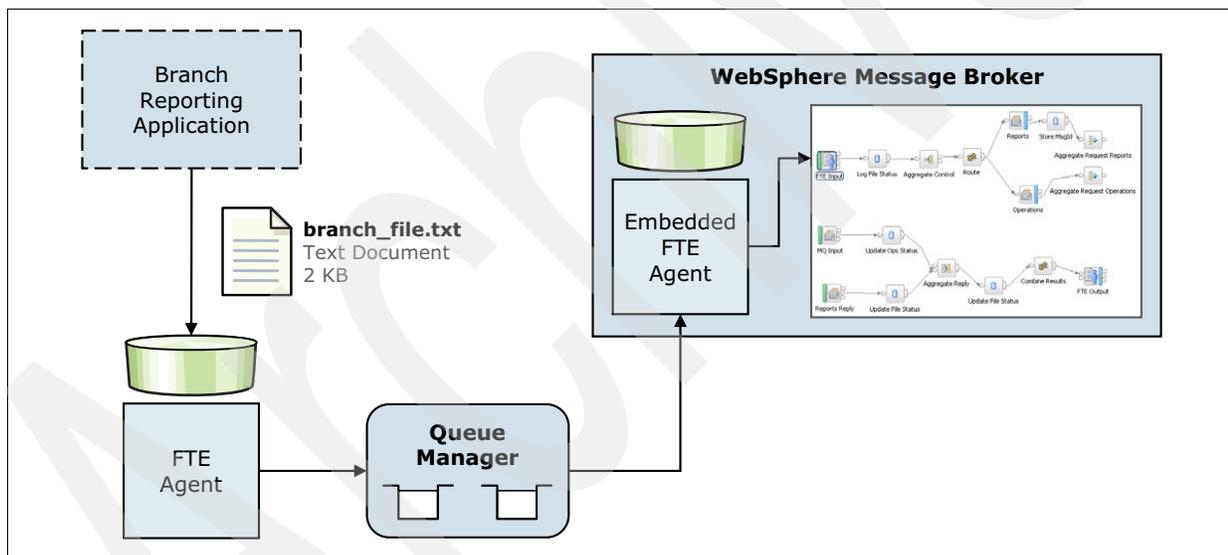


Figure 8-13 Branch application emulator schema

The branch application has no access to the broker; instead, it just produces files with reports and puts them into a Network-Attached Storage (NAS). The FTE application takes these files and forwards them to the another FTE agent that is embedded in the broker and runs under control of the execution group process.

To emulate the Branch Application, we must put an appropriate file into a directory polled by the FTE agent.

To create this agent, install the FTE server code binaries to the machine where it will run. For our scenario, we assume that the FTE agent will run on the same machine as the broker.

Note: The FTE server agent code binaries are shipped with WebSphere Message Broker, so you do not need to install them if you only plan to use FTE nodes inside the broker. This code is managed by the license of the WebSphere Message Broker. But, to run FTE as a stand-alone agent, install WebSphere MQ File Transfer Edition first. In this case, you need a separate license either for FTE client or for FTE server.

Installing the FTE server agent

It is beyond the scope of this chapter to provide the details for installation. Full installation details are provided in the Information Center and in following article at the IBM developerWorks® website:

http://www.ibm.com/developerworks/websphere/library/techarticles/1003_phillips/1003_phillips.html

Important note: Make sure the username under which you install and run the FTE configuration scripts is short enough to fit into the MQCHAR12 UserIdentifier field of the MQMD structure. If your username is longer than 12 characters, you will have problems configuring and registering FTE agents. For example your agent will not appear under the configured agents list in the FTE Configuration Manager.

This scenario uses the following names and settings during the installation:

- ▶ Coordination Queue Manager name is QM_ITS0
- ▶ FTE agent parameters:
 - Agent name is ITS0_A
 - Agent queue manager is QM_ITS0
- ▶ Command queue manager name is QM_ITS0
- ▶ Transport mode to connect to QM_ITS0 was selected to Bindings

Installation path defaults depend on the operating system. In our installation, the Product install folder was installed at:

C:\Program Files (x86)\IBM\WMQFTE

The Configuration folder was installed at:

C:\ProgramData\IBM\WMQFTE\config\

After installing WebSphere MQ FTE, you must run an MQSC script against the QM_ITS0 queue manager to create the resources required to run the FTE agent. The script you need to run is created during installation and is stored in the following path:

C:\ProgramData\IBM\WMQFTE\config\QM_ITS0\agents\ITS0_A\ITS0_A_create.mqsc

Use the following command to run the script:

```
runmqsc QM_ITS0 <  
C:\ProgramData\IBM\WMQFTE\config\QM_ITS0\agents\ITS0_A\ITS0_A_create.mqsc
```

If the command runs successfully, you can see the three lines in Example 8-2 at the end of the output.

Example 8-2 Sample output of ITS0_A_create.mqsc script

```
11 MQSC commands read.  
No commands have a syntax error.
```

All valid MQSC commands were processed.

Installing the FTE plug-in for WebSphere MQ Explorer

To monitor and submit your file transfers from a GUI application, you need to install binaries to augment the WebSphere MQ Explorer with FTE management components. You only need to install them at the machine where FTE coordination queue manager resides.

The installation procedure for the FTE plug-in for WebSphere MQ Explorer is well explained in this developerWorks article:

http://www.ibm.com/developerworks/websphere/library/techarticles/1003_phillips/1003_phillips.html#N101B2

Creating and configuring FTE agent ITS0_B

Create a second FTE agent named ITS0_B as a destination for aggregated replies from Message Broker. The agent will use another MQ Queue manager named QM_FTE. This will demonstrate the concept of distributed FTE network.

Note: In our scenario, we use a single MQ and FTE installation on one machine. But you can easily adopt the technique described in this section for the distributed FTE network. For example, you can use this technique if you need to connect a remote FTE Server agent with your FTE Configuration Manager. Just install the product binaries and replace localhost within the scripts with the actual IP addresses.

Set up a new queue manager and the communication channels between QM_FTE and QM_ITS0. Use the following procedure:

1. Create the QM_FTE using the following command:
`crtmqm QM_FTE`
2. Start it using the following command:
`strmqm QM_FTE`
3. Connect to it using the runmqsc command processor:
`runmqsc QM_FTE`
4. Copy and paste the script in Example 8-3 to set up channels to and from the Configuration Queue Manager and to create an alias for the QM_ITS0 queue manager.

Example 8-3 Configuring QM_FTE queue manager

```
DEFINE LISTENER (LISTENER.TCP) TRPTYPE(TCP) PORT(1415) CONTROL(QMGR)
START LISTENER (LISTENER.TCP)
DEFINE QREMOTE (QM_ITS0) RQMNAME(QM_ITS0) XMITQ(TO.QM_ITS0)
DEFINE QLOCAL(TO.QM_ITS0) USAGE(XMITQ) TRIGGER +
    TRIGTYPE(FIRST) +
    TRIGDPH(1) +
    TRIGMPRI(0) +
    TRIGDATA('QM_FTE.TO.QM_ITS0') +
    PROCESS(' ') +
    INITQ('SYSTEM.CHANNEL.INITQ')
DEFINE CHANNEL (QM_FTE.TO.QM_ITS0) CHLTYPE (SDR) +
    CONNAME('localhost(1414)') +
    XMITQ('TO.QM_ITS0')
```

```

DEFINE CHANNEL (QM_ITSO.TO.QM_FTE) CHLTYPE (RCVR)
START CHANNEL (QM_ITSO.TO.QM_FTE)
END

```

5. Connect to QM_ITSO using the following command:
runmqsc QM_ITSO
6. Copy and paste the script from Example 8-4 to create corresponded objects at QM_ITSO.

Example 8-4 Configuring QM_ITSO queue manager

```

DEFINE QREMOTE (QM_FTE) RQMNAME(QM_FTE) XMITQ(TO.QM_FTE)
DEFINE QLOCAL(TO.QM_FTE) USAGE(XMITQ) TRIGGER +
    TRIGTYPE(FIRST) +
    TRIGDPH(1) +
    TRIGPRI(0) +
    TRIGDATA('QM_ITSO.TO.QM_FTE') +
    PROCESS(' ') +
    INITQ('SYSTEM.CHANNEL.INITQ')
DEFINE CHANNEL (QM_ITSO.TO.QM_FTE) CHLTYPE (SDR) +
    CONNAME('localhost(1415)') +
    XMITQ('TO.QM_FTE')
DEFINE CHANNEL (QM_FTE.TO.QM_ITSO) CHLTYPE (RCVR)
START CHANNEL (QM_FTE.TO.QM_ITSO)
END

```

As a result, you created two pairs of channels between the Queue Managers and other complimentary components to enable communication between Agent ITSO_B with the Configuration Manager and back.

7. Configure an ITSO_B agent using the following command line:
fteCreateAgent.cmd -agentName ITSO_B -agentQMgr QM_FTE -f

If the command completes successfully, you will see the line in Example 8-5 at the end of the output.

Example 8-5 Configuring ITSO_B agent

```
BFGCL0053I: Agent configured and registered successfully.
```

This means that ITSO_B agent successfully updated (registered) the Configuration Manager's information, and it now appears in the list of available agents, as shown in Figure 8-14.

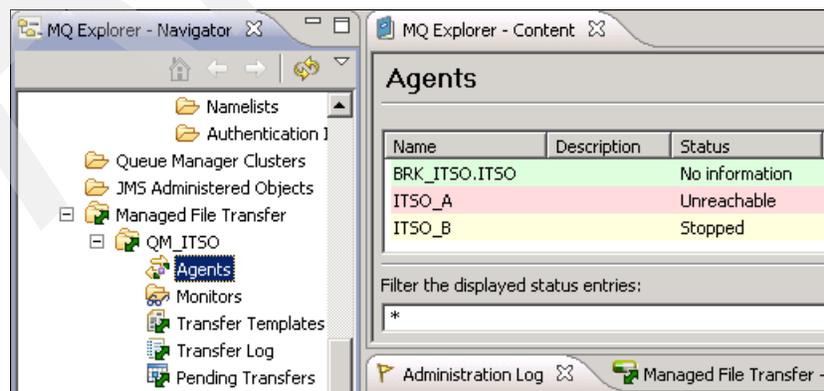


Figure 8-14 List of the registered agents

- To complete the ITSO_B configuration, execute a configuration script against QM_FTE manager:

```
runmqsc QM_ITSO <  
C:\ProgramData\IBM\WMQFTE\config\QM_ITSO\agents\ITSO_B\ITSO_B_create.mqsc
```

- Start both agents using the following command lines:

```
fteStartAgent.cmd ITSO_A  
fteStartAgent.cmd ITSO_B
```

- Verify that both agents started by using MQ Explorer. The result is similar to Figure 8-15.

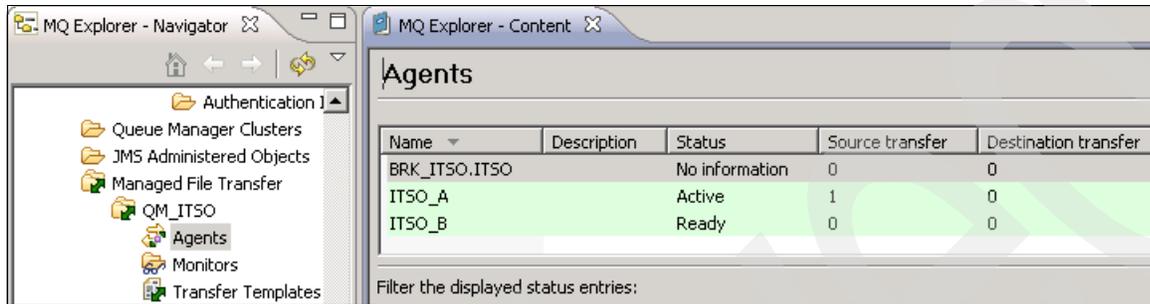


Figure 8-15 FTE agents status

FTE agent status: *Ready* status of the FTE agent means it is started. *Active* status means that the agent is ready and is processing a transfer.

Setting up a file transfer

To set up a file transfer:

- Open WebSphere MQ Explorer by clicking **Start** → **All Programs** → **IBM WebSphere MQ** → **WebSphere MQ Explorer**.
- In the MQ Explorer-Navigator view, navigate to **Managed File Transfer** → **QM_ITSO** → **Transfer Templates**.
- Right-click this element, and select **New Transfer**. In the new dialog, enter the parameters, as shown in Figure 8-16 on page 408. Click **Next**.

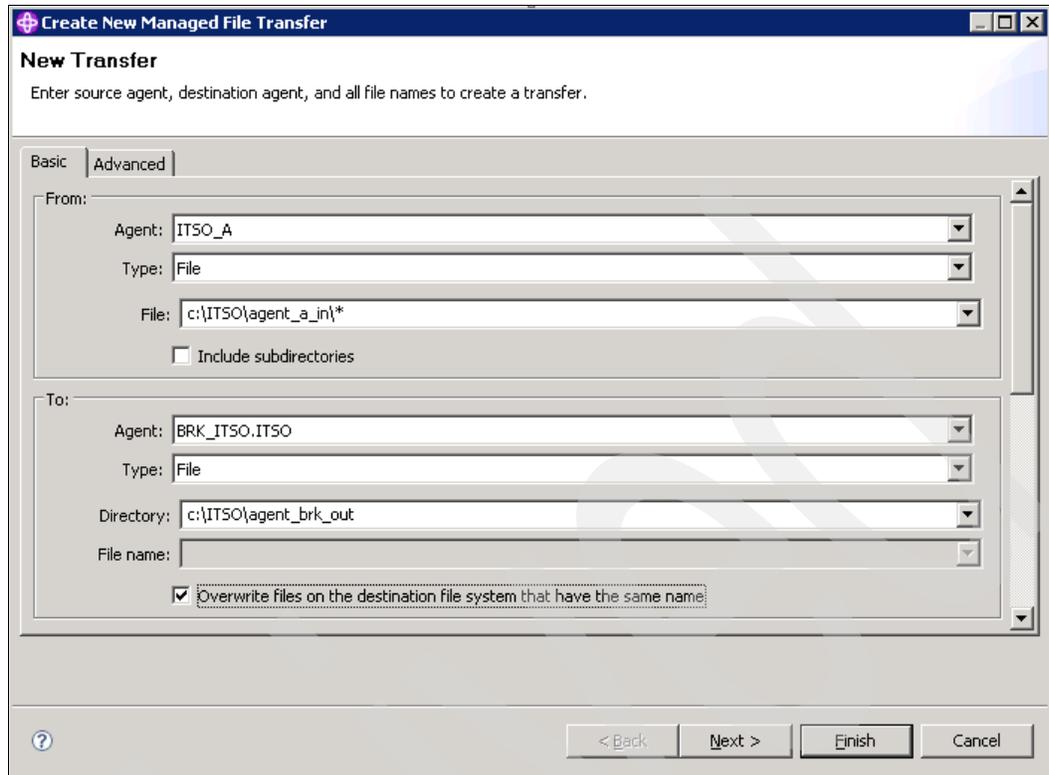


Figure 8-16 Creating a file transfer

4. In the new dialog, set up a schedule for this transfer for it to occur every minute and forever, as shown in Figure 8-17 on page 409.

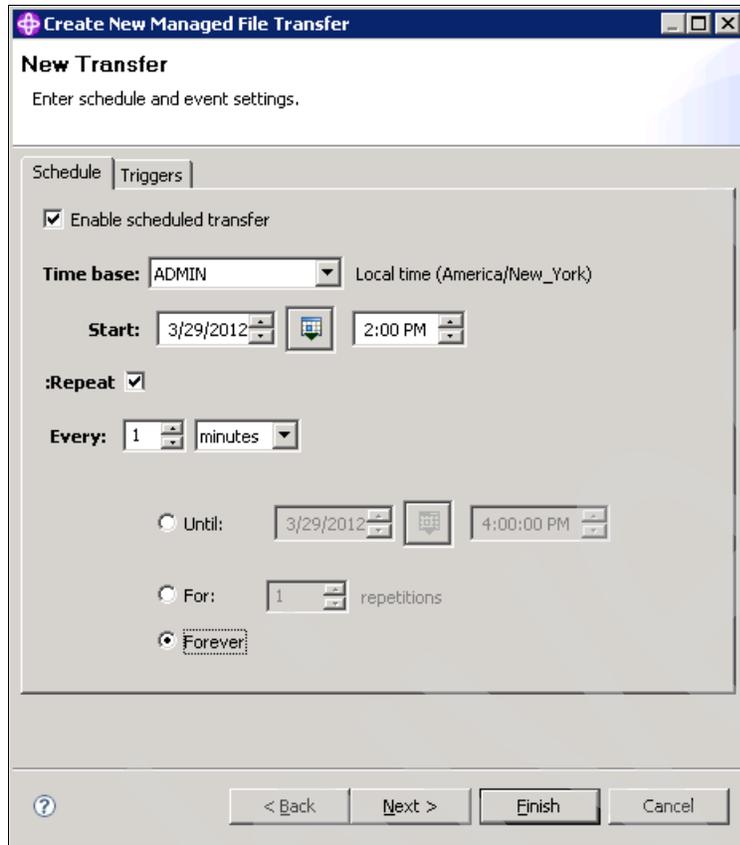


Figure 8-17 Scheduling file transfer

5. Set the start time to a time in the past (for example, an hour) to ensure that transfers start immediately after creation. Click **Finish** to create a file transfer.
6. Verify that the transfer created successfully by navigating to **Managed File Transfer** → **QM_ITSO** → **Transfer Templates**.

8.4.2 Report Data Warehouse

The Report Data Warehouse (RDW) application is MQ-enabled. This means that this application can receive WebSphere MQ messages, process them, and send a reply back to WebSphere MQ directly without using any adaptors, as shown in Figure 8-18.

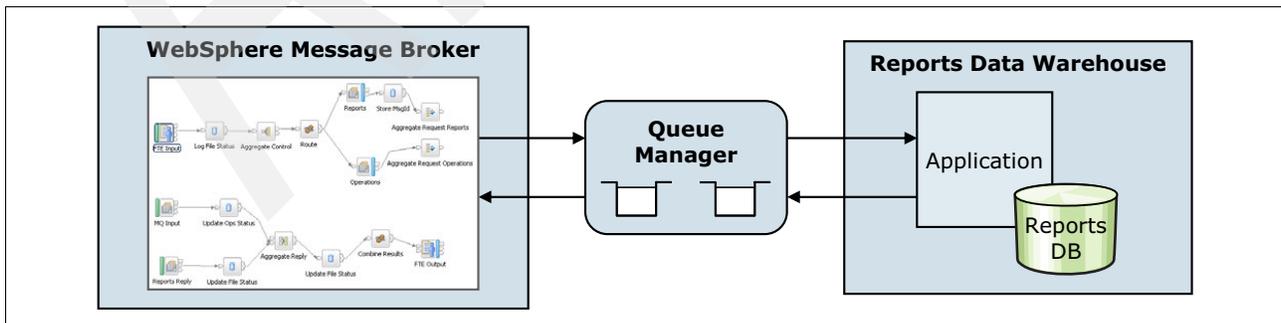


Figure 8-18 RDW application schema

Thus, it can be easily emulated as a message flow within the broker.

The emulator for an RDW application is straightforward. It consists only of two nodes: MQ Input and MQ Reply, as shown in Figure 8-19.



Figure 8-19 Report Data Warehouse Emulator

In our scenario, this RDW emulator technically does not do anything else than sending back an original request without any modification.

MQReply node behavior: The default setting for the MQReply node is to copy MsgId from the original message into CorrelationId MQMD field and send a message to the queue that is specified in the ReplyToQ field of the message. Although you can change this behavior to a different correlation schema, consider that Aggregate nodes use this particular default algorithm to correlate requests with replies.

Creating a new flow

To create a new flow:

1. In the Message Broker Toolkit, select **File** → **New** → **Application**.
2. Enter ReportsApplication as the application name, as shown in Figure 8-20. Click **Finish**.

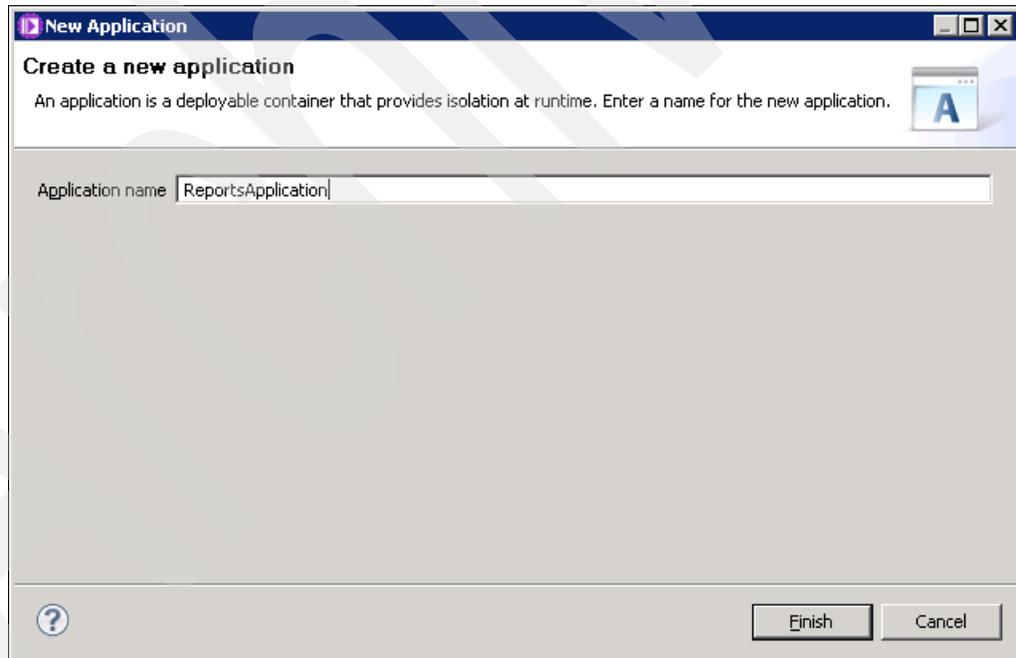


Figure 8-20 Creating the ReportsApplication

3. Right-click under the new application, and select **New** → **Message Flow** to create a message flow with the same name.
4. In the new dialog, enter ReportsApplication as the message flow name, as shown in Figure 8-21 on page 411. Click **Finish**.

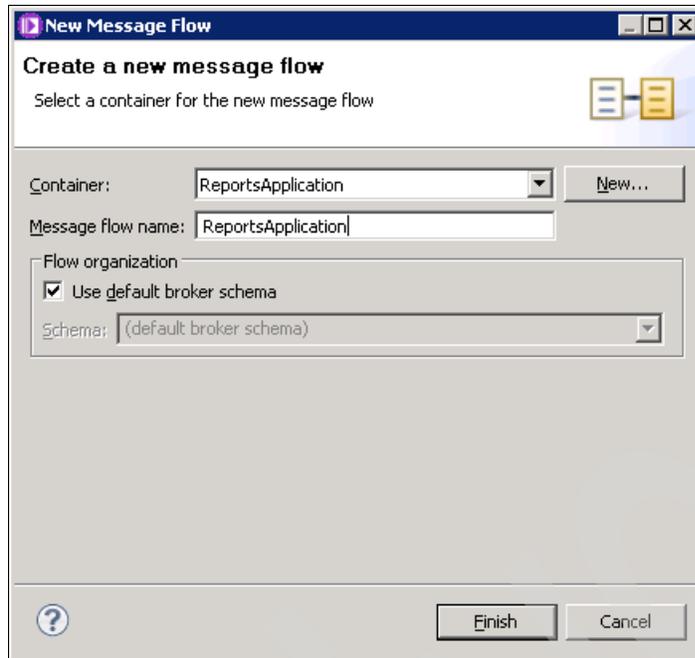


Figure 8-21 Creating ReportApplication flow

5. Drag-and-drop the MQInput and MQReply nodes from the Palette to the canvas. Connect the out terminal of the MQInput node to the input terminal of the MQReply node, as shown in Figure 8-22.

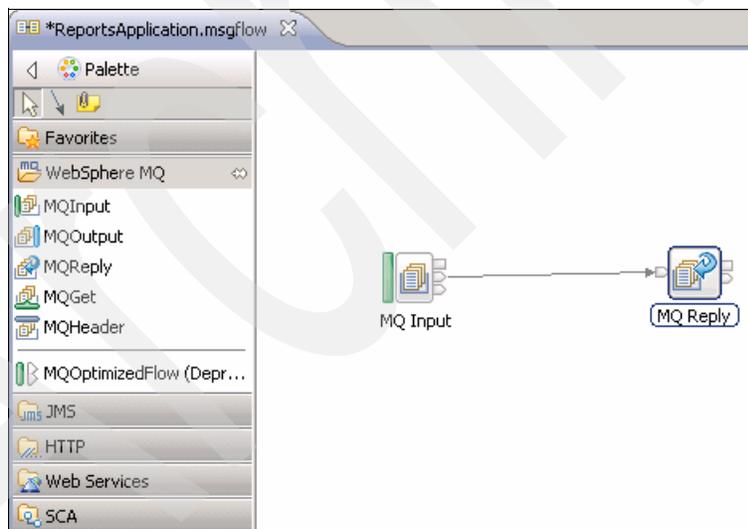


Figure 8-22 Composing ReportApplication flow

6. Select the MQ Input node and, in the Properties view, enter REPT.OUT as the queue name, as shown in Figure 8-23 on page 412.

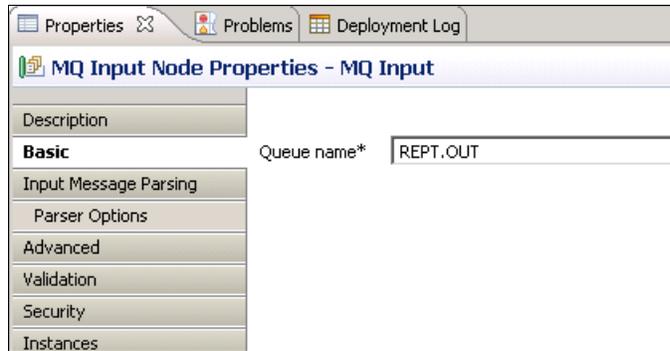


Figure 8-23 Setting Basic properties of MQ Input Node

7. Switch to the Input Message Parsing tab of the same window, and select XMLNSC as the message domain, as shown in Figure 8-24.

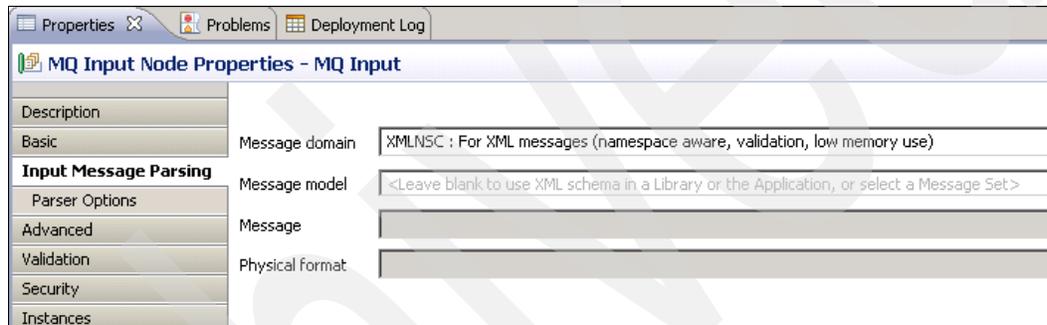


Figure 8-24 Setting Input Message Parsing properties of MQ Input Node

Note: The XML message format used here does not require a specific message model definition because it is a self-describing message format. But, the main FTE scenario uses a special tagged-delimited message, so it does include a DFDL message model, which we describe later in Section 8.5.3, “Creating a message definition” on page 422.

8. Click **File** → **Save** from the main menu. The RDW emulator is now complete. Before it can be used in runtime, it must be deployed and MQ queue resources created for it.

Configuring WebSphere MQ for the RDW application emulator

To create an input queue for the RDW emulator:

1. In WebSphere MQ Explorer, select **Queues** in the path under Queue Manager. Right-click **Queues**, and select **New** → **Local Queue**, as shown in Figure 8-25 on page 413.

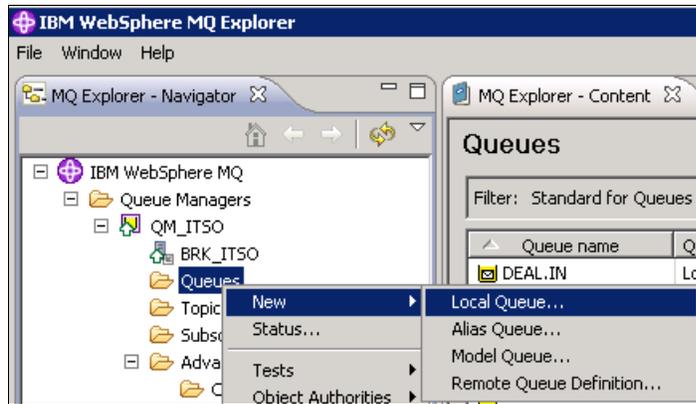


Figure 8-25 Create new local WebSphere MQ Queue

2. In the new dialog, enter REPT.OUT as the queue name, and click **Finish**. The REPT.OUT queue appears in the list of queues of the BRK_ITSO manager, as shown in Figure 8-26.

Queues						
Filter: Standard for Queues						
Queue name	Queue type	Open input count	Open output count	Current queue depth	Max queue depth	Put message
REPT.OUT	Local	0	0	0	5000	Allowed

Figure 8-26 New REPT.OUT queue

Note: The reply queue for this message flow is created during set up of the main flow of this scenario.

Deploying the RDW application emulator

To deploy the RDW application emulator:

1. In the Application Development view, right-click **ReportsApplication**, and select **Deploy**.
2. In the new dialog, select **MOCKS** from the path shown in Figure 8-27 on page 414. Click **Finish**.

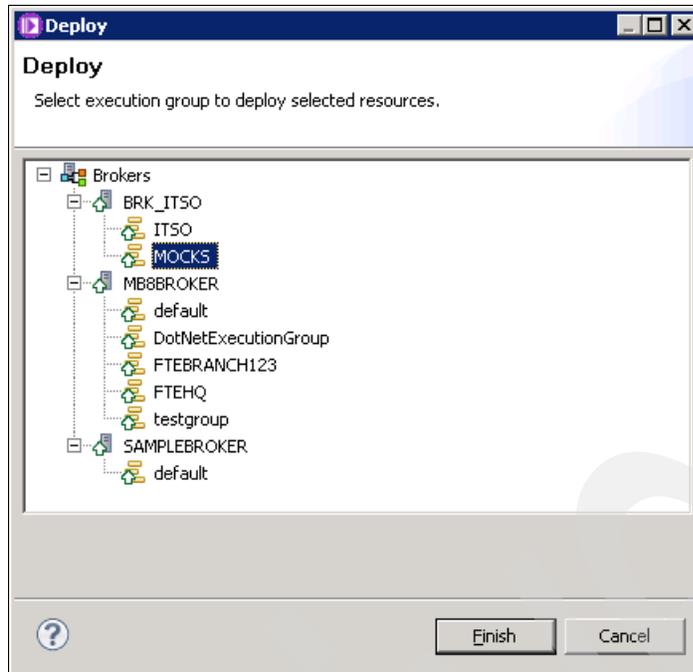


Figure 8-27

3. Ensure that the flow deployed without errors and appears in the list of flows within the execution group MOCKS. If the flow does not appear, check the Deployment log view for the errors.

8.4.3 Volume Analysis System

The Volume Analysis System (VAS) business application is technically a .NET application, which is remotely accessible through Windows Common Framework (WCF). The .NET client part of the VAS application is a library that is executed within the Broker's owned .NET runtime. See Figure 8-28.

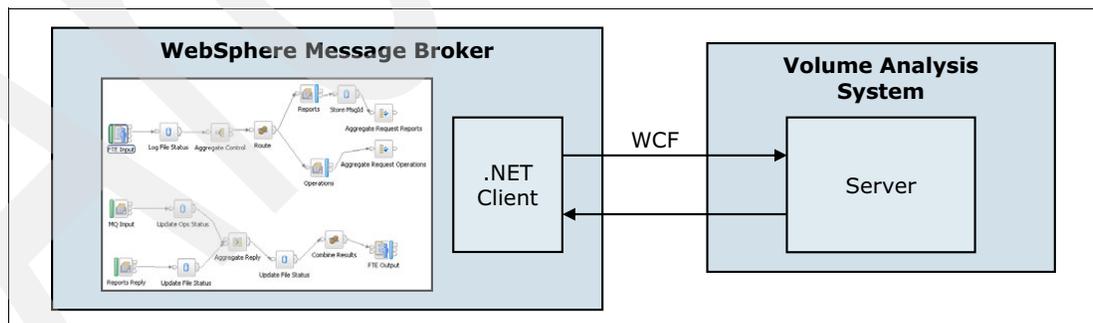


Figure 8-28 VAS application scheme

Accessing .NET code from ESQL

There are two different styles of using of .NET code from message flows. The first is to use a .NETCompute node, and the second is to invoke .NET procedures from an ESQL Compute node using a CREATE PROCEDURE statement.

Using a .NETCompute node for work with .NET in message flow has the following advantages:

- ▶ It can create instances of the classes and pass parameters of complex types to their methods.
- ▶ It can be used as a client for calling remote .NET application through WCF.
- ▶ It can be used for invoking non-static methods.
- ▶ It is more familiar to .NET programmers.

Using .NET from ESQL compute node has its own advantages:

- ▶ You can combine in one node ESQL to manipulate the data from messages and database tables and, at the same time, use the .NET API to access .NET and WCF applications.
- ▶ It is easier to manipulate and browse complex message data from ESQL.
- ▶ You do not have to create new Microsoft Visual Studio 2010 projects as long as you only need to call existing public static methods from .NET assembly.
- ▶ It is more familiar to broker developers.

Chapter 7, “Scenario: Integrating Windows Communication Foundation in message flows - Part 2” on page 273 is dedicated to a discussion about integration of WebSphere Message Broker with WCF applications. It calls .NET code from a .NETCompute node. In our scenario, we demonstrate how to invoke .NET code directly from ESQL.

The technical details about how to invoke .NET directly from the ESQL and the advantages of this approach are discussed in “Creating the broker for the VAS application emulator” on page 416.

Creating the VAS application .NET client emulator

To create a VAS application .NET client emulator:

1. Start Visual Studio, and click **File** → **New** → **Project**.
2. Select **Visual C#** from the left side of the window, and select **Class Library** as the project template. Perform the following steps:
 - a. Enter `OperationsAppClient` as the name.
 - b. Enter `c:\ITS0` as the Location.
 - c. Enter `OperationsAppClient` as the solution name.Click **OK**.
3. In the Solutions Explorer, right-click the `Class1.cs` file, and select **Rename**. Rename that class to `OperationsAppClient.cs`.
4. Open the `OperationsAppClient.cs`, and replace the generated code with the code in Example 8-6.

Example 8-6 Content of OperationsAppClient.cs class

```
using System;

namespace OperationsAppClient
{
    public class OperationsAppClient
    {
        public static string registerOperation(string branchId, string transactionId,
string symbol, double price, int amount, double commission1, double commission2)
        {
            return "00000000-0000-0000-0000-00000" + amount + "0" + transactionId;
```

```

    }
  }
}

```

In this sample method, we return nothing more than a string of numbers formatted as GUID, for example, {00000000-0000-0000-0000-000012345678}.

5. Press Ctrl+S to save the file.
6. Click **Build** → **Build the solution** from the main menu. The text in Example 8-7 is displayed in the Output view.

Example 8-7 Building the OperationsAppClient.cs file

```

----- Build started: Project: OperationsAppClient, Configuration: Debug Any CPU -----
OperationsAppClient ->
c:\ITS0\OperationsAppClient\OperationsAppClient\bin\Debug\OperationsAppClient.dll
===== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped =====

```

Limited functionality: The OperationsAppClient here is used as only an emulator of the real application. Because the main purpose of this scenario is to demonstrate the technique of calling .NET from ESQL, we are neither going into the details of communications between the .NET client and server nor creating a full fledged WCF client.

7. Close Visual Studio, and switch back to the Message Broker Toolkit.

Creating the broker for the VAS application emulator

The Aggregate nodes used in the scenario are designed to work with WebSphere MQ messages (see 8.5.5, “Using aggregation in WebSphere Message Broker” on page 431 for more information). To use a broker-supplied aggregation mechanism, you must wrap a synchronous .NET call into an asynchronous MQ invocation and put it into separate a message flow. In this scenario, we use the OperationsApplicationClient flow.

To develop the message flow for the VAS application emulator:

1. Create a new application and message flow, both named OperationsApplicationClient. See Figure 8-29.

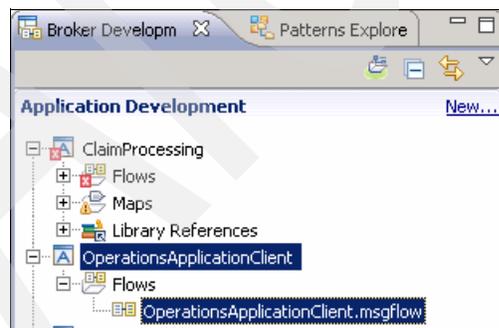


Figure 8-29 Create a new application and message flow

2. From the palette, perform the following steps:
 - a. Drag-and-drop the MQ Input and MQ Reply nodes on to the canvas.
 - b. Drag-and-drop a Compute node from the Transformation tab of the palette to the canvas. Rename the Compute node WCF Client.

- c. Wire the three nodes, as shown in Figure 8-30.

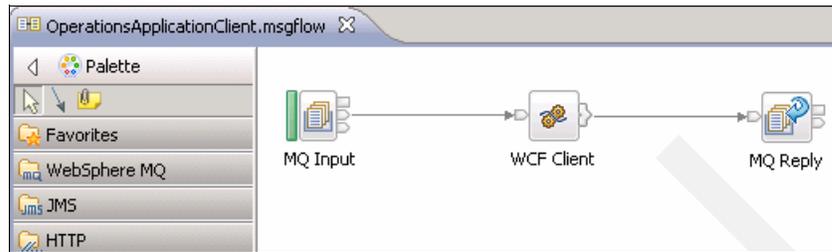


Figure 8-30 Creating a VAS application emulator

3. Select the MQ Input node and enter the following parameters in the Properties view:
 - a. In the Basic tab, enter OPTN.OUT as the queue name.
 - b. In the Input Message Parsing tab, select XMLNSC : For XML messages (namespace aware, validation, low memory use) as the message domain.
 - c. Press Ctrl+S to save your changes.

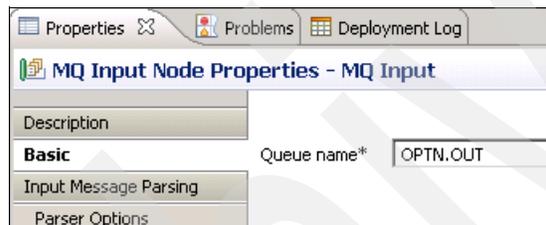


Figure 8-31 MQ Input node properties for VAS emulator

4. Select the WCF Client node. In the Basic tab of the Properties view, select **OperationsApplication_WCF_Client** as the value for ESQL Module.
5. Click Ctrl+S to save your changes.
6. Double-click the WCF Client node. In the ESQL window, copy and paste the declaration in Example 8-8 on the next line after the CREATE COMPUTE MODULE statement.

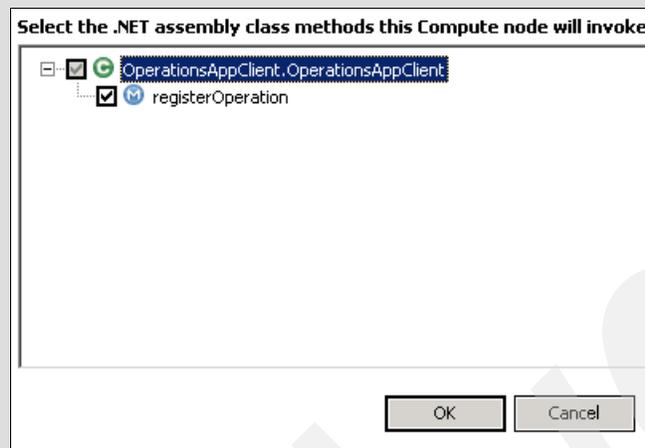
Example 8-8 Declaring the .NET procedure inside the ESQL module

```

CREATE PROCEDURE WCF_ProcessOperation(
  IN branchId CHARACTER,
  IN transactionId CHARACTER,
  IN symbol CHARACTER,
  IN price FLOAT NOT NULL,
  IN amount INT NOT NULL,
  IN commission1 FLOAT NOT NULL,
  IN commission2 FLOAT NOT NULL)
RETURNS CHARACTER
LANGUAGE .NET
EXTERNAL NAME "OperationsAppClient.OperationsAppClient.registerOperation"
ASSEMBLY
"c:\ITS0\OperationsAppClient\OperationsAppClient\bin\Debug\OperationsAppClient.dll";

```

Declare .NET procedure using drag-and-drop: The Message Broker Toolkit provides a plug-in that dramatically simplifies a process of declaring .NET methods inside ESQL modules. All you need to do to use this plug-in is to drag-and-drop the .NET library .dll file in to the ESQL compute module. As a result, a window where you can select a desired method to call from ESQL is displayed. The plug-in will generate an ESQL declaration.



7. Replace the Function Main content block with the code shown in Example 8-9.

Example 8-9 Calling .NET procedure from the ESQL module

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyEntireMessage();
  Declare b reference to OutputRoot.XMLNSC.branch;
  Declare op reference to b.operation;

  Set op.result = WCF_ProcessOperation(
    b.branch_number, op.transactionID, op.symbol,
    CAST(op.price as FLOAT),
    CAST(op.amount as INT),
    CAST(op.commission1 as FLOAT),
    CAST(op.commission2 as FLOAT));

  RETURN TRUE;
END;
```

8. Save and close the file.

8.5 Creating the main message flow

In this section, we create the primary message flow for the scenario. We illustrate integration with WebSphere MQ File Transfer Edition. We also show a process of modelling messages using the DFDL editor, (new in Message Broker V8.0). We guide you through creating *fan-out* and *fan-in* flows, which are common tasks, for example, when you integrate one front-end application with several back-end systems.

For those .NET developers who are interested in learning WebSphere Message Broker techniques, we also show a method of *splitting* a message into pieces and *routing* these different pieces to different end-systems using ESQL.

Following the Design → Build → Deploy → Test paradigm, we start our main development from the design of a message flow and our data model, then we fill our message flow with code, build it, and then run an overall test through it. We also discuss possible extension points to this scenario.

8.5.1 Message flow overview

Figure 8-32 shows the completed message flow. This flow actually consists of two flows: the first processes a message from FTE Agent ITS0_A and sends an asynchronous request to subsequent systems and the second processes an asynchronous response from these systems and routes a combined message to the FTE Agent ITS0_B.

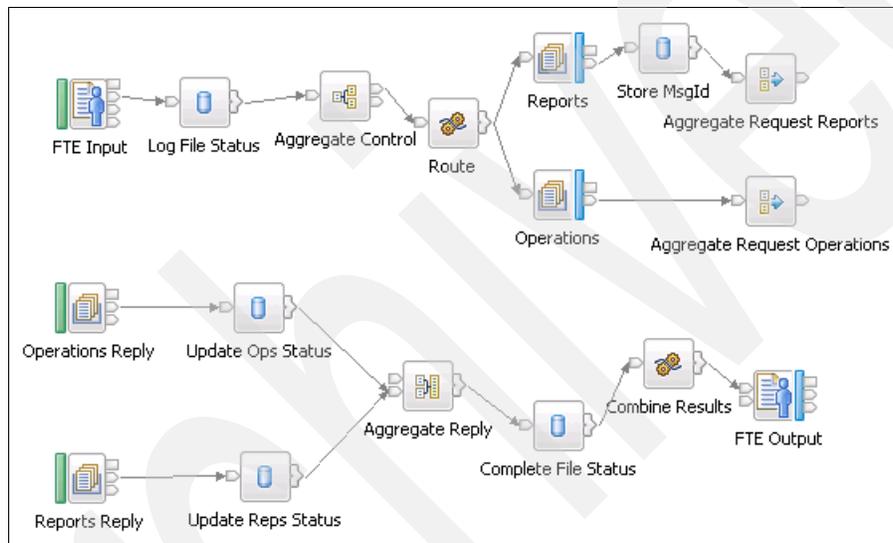


Figure 8-32 Completed message flow

The following list describes the behavior of each node:

1. The FTE Input node receives files from the FTE network and parses them into a logical tree.
2. The Log File Status node writes general information about the received file into the database.
3. The Aggregate Control node marks the beginning of the fan-out flow part.
4. The Route node splits the message into report and operation parts and routes them to the RDW and VAS applications.
5. The Reports node sends the message to the destination MQ queue for the RDW application.
6. The Store MsgId node stores the generated messageId into the database.
7. The Aggregate Request Reports node marks the completion of the request part of the Fan-out flow.
8. The Operations node sends the message to the destination MQ queue for the VAS application.

9. The Aggregate Request Operations node marks the completion of the second request part of the fan-out flow.
10. The Operations Reply node receives replies from the VAS application and parses them into a logical tree.
11. The Update Ops Status node records that replies were received from the VAS application.
12. The Aggregate Reply node combines the replies and sends them as one message to the subsequent part of Fan-in flow.
13. The Reports Reply node receives replies from the RDW application and parses them into the logical tree.
14. The Update Reps Status node records that replies were received from the RDW application.
15. The Complete File Status node updates the database with the completion status for the processed file and updates the last_modified information.
16. The Combine Results node extracts information from the OPERATION and REPORT folders of the combined replies and creates a new message using the FTERRecordSample DFDL data format definition.
17. The FTE Output node sends files back to the WebSphere MQ File Transfer Edition.

8.5.2 Creating and connecting the nodes

The first step in building the message flow is to add and wire the nodes together:

1. Create a new application named FTESample with an associated message flow named FTESampleFlow.
2. Add the nodes in Table 8-1 and wire them together to form the flow. As you build the message flow, use Figure 8-32 on page 419 as a reference.

Table 8-1 Node naming and wiring information

Palette drawer / node type	Node name and description	Wiring information (terminal to node)
File FTEInput	FTE Input	► Out: Log file status
Database Database	Log File Status	► Out: Aggregate Control
Routing AggregateControl	Aggregate Control	► Out: Route
Transformation Compute	Route	► Out: Reports ► Out1: Operations
WebSphere MQ MQOutput	Reports	► Out: Store Msgld
Database Database	Store Msgld	► Out: Aggregate Request Reports
Routing AggregateRequest	Aggregate Request Reports	(none, end of flow)
WebSphere MQ MQOutput	Operations	► Out: Aggregate Request Operations

Palette drawer / node type	Node name and description	Wiring information (terminal to node)
Routing AggregateRequest	Aggregate Request Operations	(none, end of flow)
WebSphere MQ MQInput	Operations Reply	▶ Out: Update Ops Status
Database Database	Update Ops Status	▶ Out: Aggregate Reply
Routing AggregateReply	Aggregate Reply	▶ Out: Complete File Status
WebSphere MQ MQInput	Reports Reply	▶ Out: Update Repls Status
Database Database	Update Repls Status	▶ Out: Aggregate Reply
Database Database	Complete File Status	▶ Out: Combine Results
Transformation Compute	Combine Results	▶ Out: FTE Output (In)
File FTEOutput	FTE Output	(none, end of flow)

- Table 8-2 describes the configuration information for the node properties. Select each node in the canvas and enter or select the values for each of the properties. When you are finished, save the message flow.

Table 8-2 Node properties

Node	Tabs, Fields, and Values
FTEInput	Do not configure this node yet. We configure this node later in “Configuring the FTE Input node for the scenario” on page 430, because we need to create a message definition first.
Log File Status	Basic: ▶ Data Source: FTE_DS ▶ Statement: FTESampleFlow_Log_File_Status
Aggregate Control	Basic: ▶ Aggregate Name: FTE_AGGR
Route	Basic: ▶ ESQL module: FTESampleFlow_Route
Reports	Basic: ▶ Queue name: REPT.OUT
Store MsgId	Basic: ▶ Data source: FTE_DS ▶ Statement: FTESampleFlow_Store_MsgId
Aggregate Request Reports	Basic: ▶ Folder name: REPORT

Node	Tabs, Fields, and Values
Operations	Basic: <ul style="list-style-type: none"> ▶ Queue name: OPTN.OUT
Aggregate Request Operations	Basic: <ul style="list-style-type: none"> ▶ Folder name: OPERATION
Operations Reply	Basic: <ul style="list-style-type: none"> ▶ Queue name: OPTN.IN Input Message Parsing: <ul style="list-style-type: none"> ▶ Message domain: XMLNSC
Update Ops Status	Basic: <ul style="list-style-type: none"> ▶ Data source: FTE_DS ▶ Statement: FTESampleFlow_Update_Ops_Status
Aggregate Reply	Basic: <ul style="list-style-type: none"> ▶ Aggregate name: FTE_AGGR
Reports Reply	Basic: <ul style="list-style-type: none"> ▶ Queue name: REPT.IN Input Message Parsing: <ul style="list-style-type: none"> ▶ Message domain:DFDL ▶ Message: {}:FTERecordSample
Update Reps Status	Basic: <ul style="list-style-type: none"> ▶ Data source: FTE_DS ▶ Statement: FTESampleFlow_Update_Reps_Status
Complete File Status	Basic: <ul style="list-style-type: none"> ▶ Data source: FTE_DS ▶ Statement: FTESampleFlow_Complete_File_Status
Combine Results	Basic: <ul style="list-style-type: none"> ▶ Data source: FTE_DS ▶ ESQL module: FTESampleFlow_Combine_Results ▶ Compute mode: LocalEnvironment and Message
FTE Output	Do not configure this node yet. We configure this node later on in “Configuring the FTE Output node” on page 430, because we need to create a message definition first.

Configuration is not complete yet: We configure the FTE nodes later in section 8.5.4, “Using FTE nodes” on page 429, because we need to create a message definition first.

The code for the database and ESQL nodes is also provided in the following sections. See 8.5.7, “Accessing databases from the database nodes” on page 435 and 8.5.6, “Producing multiple messages from the compute node” on page 434

8.5.3 Creating a message definition

In this section, we use the Message Broker Toolkit to create a DFDL message definition to parse a report file inside the FTESampleFlow message flow.

1. Right-click the FTESample application, and select **New** → **Message Model**.

2. Select **Record Oriented Text**, and click **Next**.
3. On the next screen, select **Create an empty DFDL schema file, I will model my data using the DFDL schema editor**, and click **Next**.
4. Enter `FTERecordSample` as the DFDL schema file name. Click **Finish**.
5. In the schema that displays, select the **Add a message to the schema** icon. In the Add Message dialog, enter `FTERecordSample` as the message name, as shown in Figure 8-33. click **OK**.

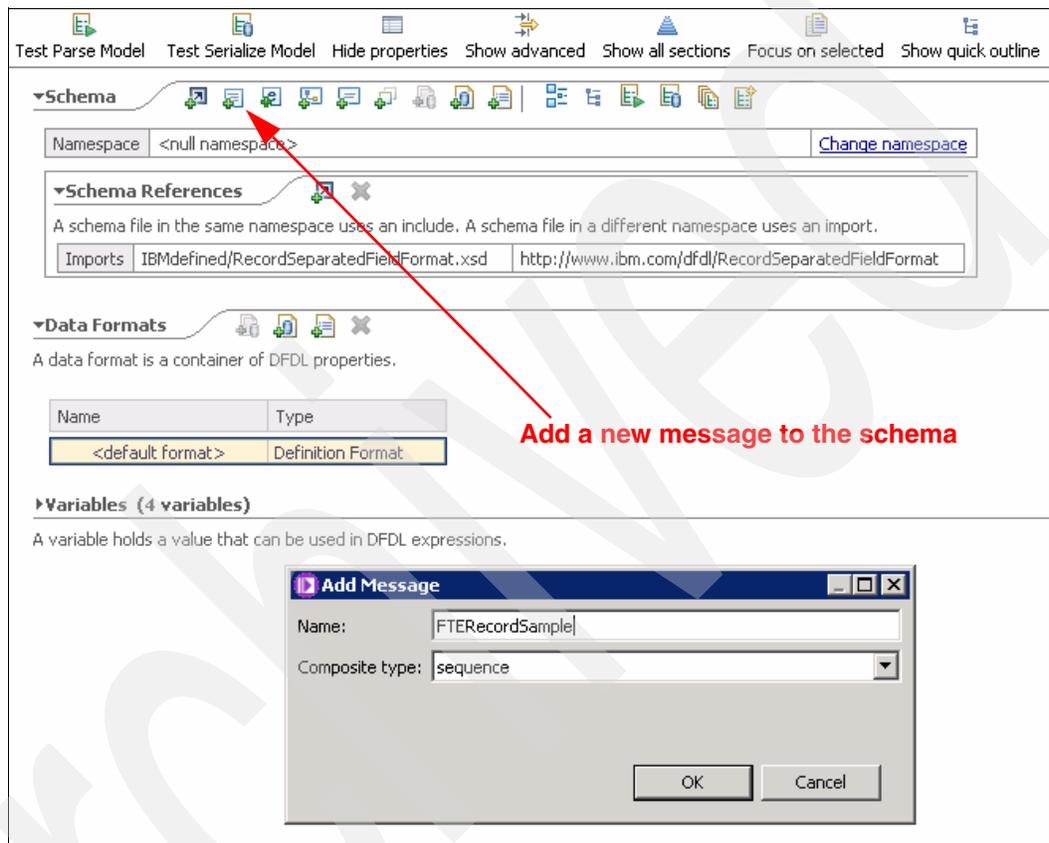


Figure 8-33 Add a new message to the schema

6. Right-click the **sequence** element, and select **Add Complex Local Element**, as shown in Figure 8-34 on page 424.

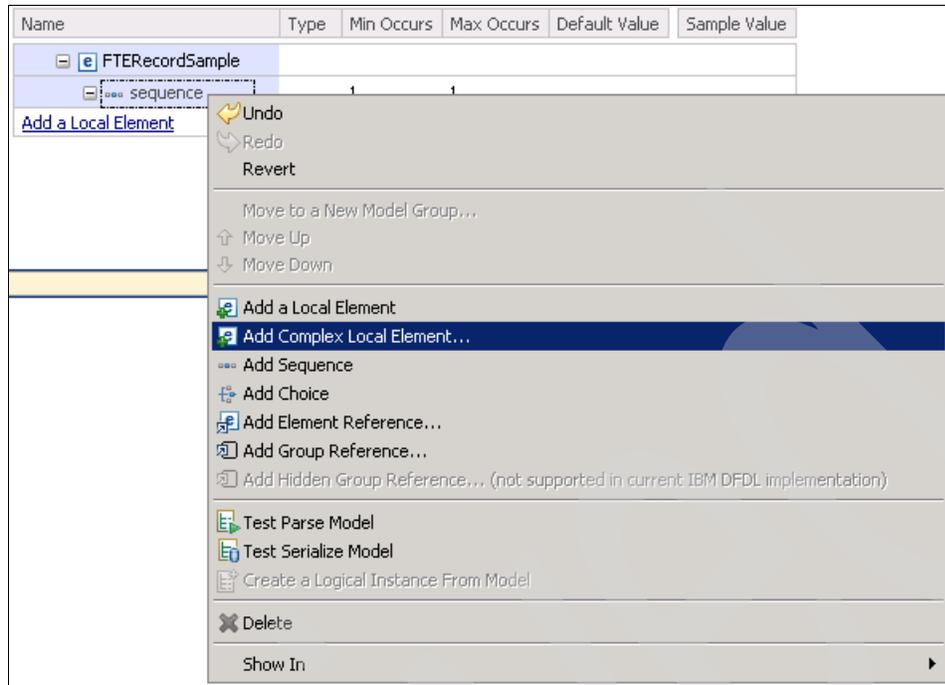


Figure 8-34 Adding a complex local element

- In the Add Complex Local Element dialog, enter `branch` as the Name, and choose `sequence` as the Composite Type. Click **OK**.

The new element is added to the sequence, as shown in Figure 8-35.

Name	Type	Min Occurs	Max Occurs	Default Value	Sample Value
FTERecordSample					
sequence		1	1		
branch		1	1		
sequence		1	1		

Figure 8-35 Adding a new branch element

- Repeat steps 6 and 7 for the same global sequence under `FTERecordSample`, and create additional child elements named:
 - customer
 - operations
 - report
- Right-click **sequence** under the branch element, and select **Create Local Element**. Enter `branch_number` as its name. Leave the Type of this element as `string`.
- Repeat step 9 to create the date element under branch (see Figure 8-36 on page 425).
- Using step 9 as an example, create a child element named `id` under customer.
- Create a complex child element named `operation` under operations sequence (see Figure 8-36 on page 425).
- Populate the operation sequence with child elements, as shown on Figure 8-36 on page 425. You can leave the type of elements as `string` for now. We will explain how to change them later.

14. Using Figure 8-36 as an example, create the remaining complex elements and their respective child elements.

Name	Type	Min Occurs	Max Occurs	Default Value	Sample Value
[-] FTERRecordSample					
[-] sequence		1	1		
[-] branch		1	1		
[-] sequence		1	1		
[e] branch_number	string	1	1	1	1
[e] date	string	1	1	1.3.2012	1.3.2012
[-] customer		1	1		
[-] sequence		1	1		
[e] id	string	1	1	R90306	R90306
[-] operations		0	1		
[-] sequence		1	1		
[-] operation		1	unbounded		
[-] sequence		1	1		
[e] transactionID	string	1	1	1026	1026
[e] symbol	string	1	1	IBM	IBM
[e] price	double	1	1	190.1	190.1
[e] amount	int	1	1	10	10
[e] commission1	double	1	1	0.1	0.1
[e] commission2	double	1	1	0.01	0.01
[e] registeredID	string	0	1		
[-] report		0	1		
[-] sequence		1	1		
[-] stocks		1	1		
[-] sequence		1	1		
[-] stock		1	unbounded		
[-] sequence		1	1		
[e] name	string	1	1	IBM	IBM
[e] open_amount	double	1	1	0	0
[e] close_amount	double	1	1	10	10
[e] delta	double	1	1	0	0
[-] total_commission		1	1		
[-] sequence		1	1		
[e] commission1	double	1	1	1.2	1.2
[e] commission2	double	1	1	0.01	0.01
[e] total	double	1	1	1.21	1.21
[-] cash		1	1		
[-] sequence		1	1		
[e] open_amount	double	1	1	1000000	1000000
[e] close_amount	double	1	1	500000	500000
[e] delta	double	1	1	-500000	-500000

Figure 8-36 FTERRecordSample data format definition

15. For each element of Type double in Figure 8-36:

- Click the respective cell under the Type column, and select **double** from the drop-down menu.
- Select the element, and in the Representation Properties panel on the right side of the editor window, find the Text Connect properties section.
- Click to expand **Text Number Representation**, and enter #####.## as a value for Number Pattern property, as shown in Figure 8-37 on page 426.

Text Content	
Text Number Representation	standard
Number Pattern	#####.##
Grouping Separator	,
Decimal Separator	.
Escape Scheme Reference	recSepFieldsFmt:RecordEscape

Figure 8-37 Number format properties for element of type double

16. Select the Type `int` for the amount element under the operation sequence.
17. Select the **operations** element:
 - a. In the Occurrences section in the Representation Properties at the right side of the editor window, set Occurs Count Kind to the `implicit` value.
 - b. Set the Min occurs property, and enter `0` as the value.
18. Repeat step 17 for the `registeredID` element under the operation sequence.
19. Repeat step 17 for the `report` element.
20. Select the unbounded value for the Max occurs column for the operation and stock elements.
21. (Optional). Change the default and sample values of each element to the values shown in Figure 8-36 on page 425.
22. Select the sequence element directly under the `FTERecordSample` element. In the Delimiters section in the Representation Properties panel, change the values of the properties Separator, Initiator, Terminator to `<no separator>`, `<no initiator>`, `<no terminator>` respectively by clearing the values of these fields.
23. Using step 22 as an example, change the Separator, Initiator and Terminator properties for each element, as shown in Table 8-3.

Note: There should be no spaces and new lines in the values for the Separator, Initiator and Terminator properties. Any new line and space that you see in Table 8-3 is present only because of limitations of the page size.

Table 8-3 Delimiters for the `FTERecordSample` format

Relative path	Element	Separator	Initiator	Terminator
/	sequence	<no separator>	<no initiator>	<no terminator>
/	branch		BRANCH%WSP*;	%CR;%LF;
/branch	sequence	,	<no initiator>	<no terminator>
/branch	branch_number		%WSP*#	<no terminator>
/branch	date		%SP;DATE%SP;	<no terminator>
/	customer		CUSTOMER	%CR;%LF;
/customer	sequence	,	<no initiator>	<no terminator>
/customer	id		%SP;ID:%SP;	<no terminator>

/	operations		OPERATIONS%CR;%LF;	%CR;%LF;END%SP;OF%SP;OPERATIONS%CR;%LF;
/operations	sequence	%CR;%LF;	<no initiator>	<no terminator>
/operations	operation		%SP;	<no terminator>
operations/operation	sequence	,	<no initiator>	<no terminator>
operations/operation	transactionID		<no initiator>	<no terminator>
operations/operation	symbol		%SP;	<no terminator>
operations/operation	price		%SP;	<no terminator>
operations/operation	amount		%SP;	<no terminator>
operations/operation	commission1		%SP;	<no terminator>
operations/operation	commission2		%SP;	<no terminator>
operations/operation	registeredID		%SP;	<no terminator>
/	report		REPORT%CR;%LF;	<no terminator>
report	/sequence	<no separator>	<no initiator>	<no terminator>
report	/stocks		STOCKS%CR;%LF;	%CR;%LF;END%SP;OF%SP;STOCKS%CR;%LF;
report/stocks	sequence	%CR;%LF;	<no initiator>	<no terminator>
report/stocks	stock		%SP;	<no terminator>
report/stocks/stock	sequence	,	<no initiator>	<no terminator>
report/stocks/stock	name		<no initiator>	<no terminator>
report/stocks/stock	open_amount		%SP;	<no terminator>
report/stocks/stock	close_amount		%SP;	<no terminator>

report/stocks/stock	delta		%SP;	<no terminator>
report	total_commission		COMMISSION%CR;%LF;	%CR;%LF;
report/total_commission	sequence	,	<no initiator>	<no terminator>
report/total_commission	comission1		%SP;	<no terminator>
report/total_commission	comission2		%SP;	<no terminator>
report/total_commission	total		%SP;	
report	cash		CASH%CR;%LF;	%CR;%LF;
report/cash	sequence	,	<no initiator>	<no terminator>
report/cash	open_amount		%SP;	
report/cash	close_amount		%SP;	
report/cash	delta		%SP;	

Special characters: Because DFDL is an extension of WSDL, it uses an XML-like style for defining non-printable or special characters. You can find some of these characters in Table 8-3 on page 426. %SP stands for space character, %WSP stands for any whitespace character, %CR is carriage return and %LF is line feed. You can also use %WSP*, which means optional whitespace or %WSP+, which means one or more whitespaces.

24. Save your work. You will verify that the message format definition is correct in the next sections.

Sample file used in scenario

We used the text file in Example 8-10 to verify that the scenario is working.

Example 8-10 Content of the branch_file.txt

```
BRANCH #1, DATE 1.3.2012
CUSTOMER ID: R90306
OPERATIONS
  1917, IBM, 190.10, 20, 0.38, 0.03
  1945, IBM, 190.50, 20, 0.38, 0.03
  1976, IBM, 190.99, 10, 0.38, 0.03
END OF OPERATIONS
REPORT
STOCKS
  IBM, 100, 150, 50
END OF STOCKS
COMMISSION
```

1.14, 0.09, 1.23
CASH
1000000, 990476.87, -9523.13

You can verify that your definition of the FTESampleRecord message is correct by executing a *test parsing* of that message.

Test parsing the DFDL model

To test parse the DFDL model:

1. Create a new text file named `branch_file.txt`. Copy and paste the content of Example 8-10 on page 428 to this file, and store it in the `c:\ITS0` folder of the file system.

Note: `branch_file.txt` should have an empty last line and empty space in front of each operation, stock, and commission elements, as defined in the FTESampleRecord format and shown in Example 8-10 on page 428.

2. Click the **Test Parse Model** icon on the top-left corner of the main DFDL editor window in the Message Broker Toolkit. Select **FTERecordSample** as a Message Name, and browse for the `branch_file.txt` using the Content from a data file option.
3. Click **OK**, and then switch to the DFDL Test perspective in the Message Broker Toolkit. After a while, you should see a hint stating 'Parsing completed successfully'. If it is not, you should go back and check the message elements properties with Table 8-3 on page 426.

8.5.4 Using FTE nodes

The broker interacts with the FTE network by means of the FTEInput and FTEOutput node types. These nodes read and write files from and to the FTE network, respectively. FTE nodes can parse records in the file and have the ability to process large files without reading the entire contents into memory at the same time. To configure FTE nodes, specify a file mask for incoming files and the remote FTE agent parameters for outgoing files. You do not have to create the agent that handles the incoming files because the broker handles this automatically.

FTE Input node overview

The FTEInput node receives files from remote FTE agents. On deploy, the broker automatically creates an FTE agent and its underlying queues, creating one agent per each execution group that contains a message flow with FTEInput nodes. The FTE agent name is derived from `Broker.ExecutionGroup` and is not configurable. The name must be a valid format for generating the queue name. In our scenario, the name of the FTE agent that the broker will generate is `BRK_ITS0.ITS0`.

The FTE Input node does not use a polling mechanism to scan directories because it is triggered by the embedded agent when a file arrives and the transfer is complete. It processes each file in a transfer separately and can process each file in parallel.

In addition to receiving the data from the transfer, it also receives all the metadata associated with the transfer. This includes information from FTE, but can also include user-defined data.

Each FTE Input node can specify a filter of which files it wants to process. By default, it processes all files. If two nodes both have a filter that matches a file, only one will get it. By default, if no filter expression is given, the node accepts any transferred file.

The file name filter accepts wild cards, blank (accept any files) or a string that must match exactly. Relative paths are allowed and are relative to the default transfer directory.

Configuring the FTE Input node for the scenario

To configure the FTEInput node:

1. Select the FTEInput node on the canvas to open the Properties view.
2. Select **DFDL** as the message domain in the Input Message Parsing tab, and choose FTERecordSample as the message, as shown in Figure 8-38.

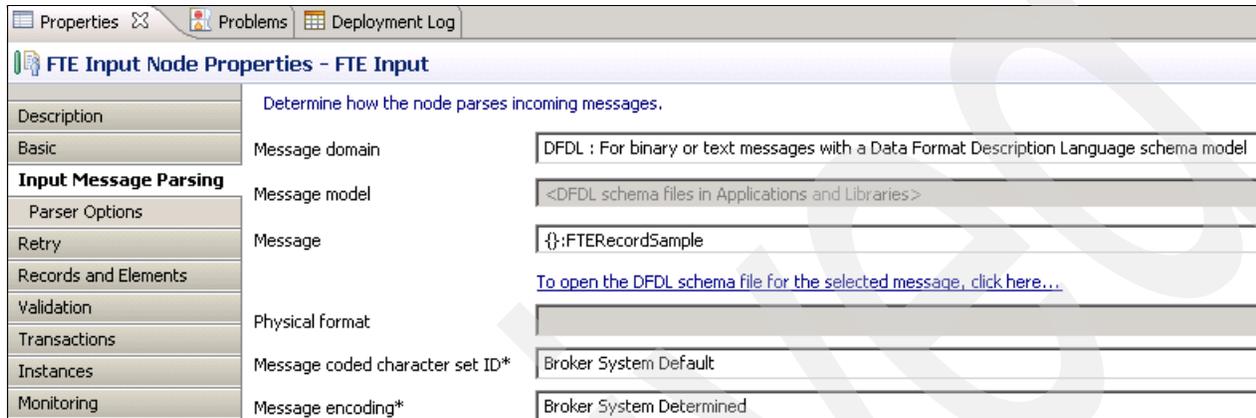


Figure 8-38 Setting FTEInput node properties

3. Select the **Record and Elements** tab, and select **Parsed Record Sequence** as the record detection mechanism.
4. Select the **Transactions** tab, and select **YES** as the transaction mode value. This action allows all requests for the RDS and VAS applications to be committed to WebSphere MQ under a single unit of work.
5. Click Ctrl+S to save your changes.

FTE Output node overview

The main properties of the FTEOutput node are the details of where to transfer the files. The properties can be overridden using the local environment. The destination the file is to be written to is a file system on a remote agent. The node first writes it to a staging directory on the local file system and then sends a request to the embedded agent to transfer it.

Files are built up in a mqsi transmit directory before being moved to the final file name after the file is complete and the transfer request is sent to the FTE agent. The name in the staging area is the same as the name the file will be transferred to, unless a file with that name already exists on the local file system. If it exists, a number is appended to the file. The file name transferred to the remote system is not affected by this and will not have the number appended.

Configuring the FTE Output node

The only mandatory property you have to specify for the FTEOutput node is the destination file name in the Basic tab.

Enter *.txt as the value, as shown in Figure 8-39 on page 431. This value is just a stub for the required field. The name is set later using ESQL.

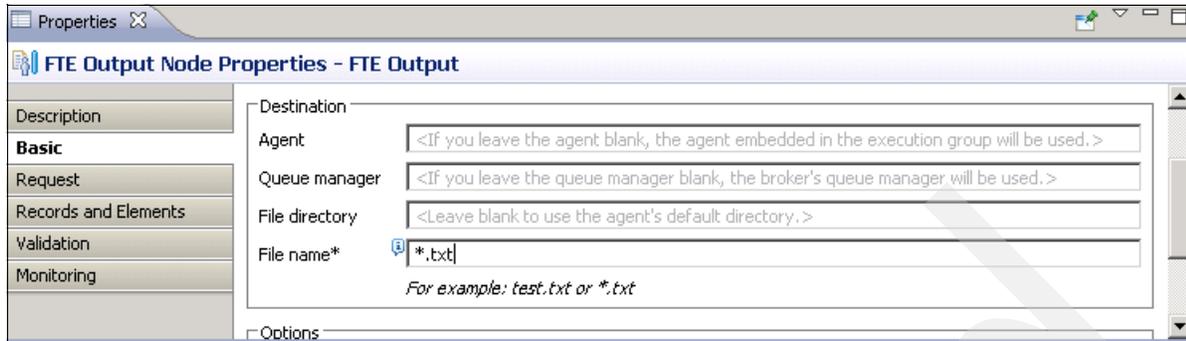


Figure 8-39 Setting FTEOutput node properties

Other FTE transfer parameters (for example, the destination agent name) are filled in dynamically at runtime using the Compute node. See section “Composing an FTE reply message from aggregated replies” on page 432 for more information.

8.5.5 Using aggregation in WebSphere Message Broker

Some business scenarios require request/reply processing with multiple replies (fan-out and fan-in). This design also provides an opportunity for an improvement in response time because slow requests can be performed in parallel and do not have to follow each other sequentially. If the subtasks can be processed independently, and do not have to be handled as part of a single unit of work, they can be processed by separate applications.

Aggregation nodes automatically perform the task of synchronizing multiple requests and their incoming replies. This saves a lot of work compared to coding multiple MQGet nodes in the flow for storing the state of the processing.

During the fan-out process (the upper part of the message flow in Figure 8-32 on page 419), some control information is stored automatically in the LocalEnvironment by the AggregateControl, AggregateRequest, and MQOutput nodes. A special folder named ComIbmAggregateControlNode is used for this information. Additionally, the AggregateRequest nodes store information about awaited requests in the broker database.

You can either create the fan-out and fan-in flows in the same message flow or in two different message flows. In either case, the two parts of the aggregation are associated by setting the Aggregate Name property. A single flow is easier to implement for a simple case, but there are some limitations to this approach, and, in most cases, you will find that the flexibility offered by two message flows is preferable. The two flows can be modified independently of each other. The two flows can also be stopped and started independently of each other. The two flows can also be deployed to separate execution groups to take advantage of multiprocessor systems, or to provide data segregation for security or integrity purposes.

AggregateControl node

After all request messages are sent, the control information is propagated from the AggregateControl node to the AggregateReply node in the fan-in process.

AggregateReply node

Any incoming reply from the MQInput node is recorded by the AggregateReply node and stored until all outstanding replies have arrived or a timeout occurs. Next, the AggregateReply node composes a logical message tree with multiple bodies. Each reply is contained in separate body folder, which is identified by the request name.

Handling transactions in aggregation nodes

As shown in Figure 8-40, an aggregation flow is a complex application that involves multiple transactions (units of work):

1. In a fan-out flow, the first logical unit of work ends when all requests are sent and the automatic control message is put onto the `SYSTEM.BROKER.AGGR.CONTROL` queue.
2. In a fan-in flow, there is a logical unit of work for each incoming reply while aggregating from the MQInput node to the AggregateReply node.
3. This is similar to step 2, where there is a logical unit of work for each incoming reply while aggregating from the MQInput node to the AggregateReply node.
4. For the aggregated result message, a new logical unit of work starts in the AggregateReply node (which has a `TransactionMode` property) when all replies have arrived or the timeout interval is expired.

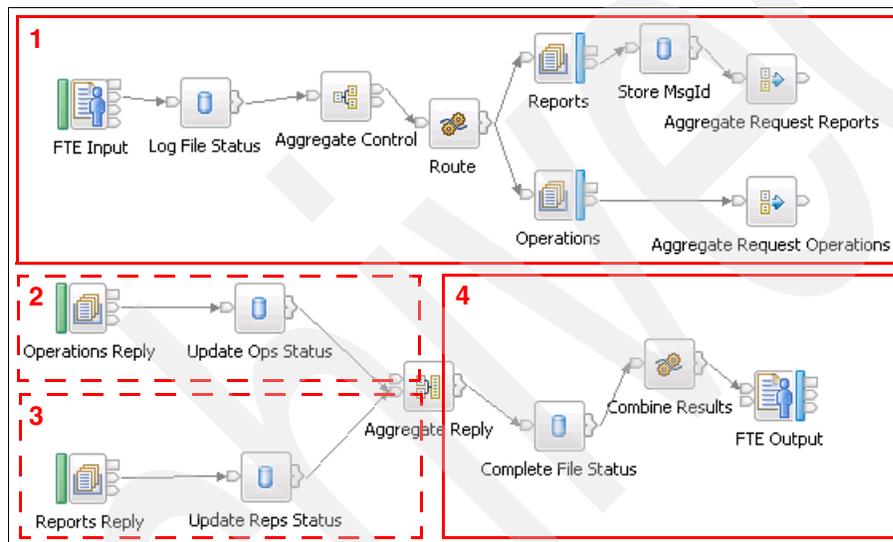


Figure 8-40 Handling transaction in Aggregation nodes

These are only the logical units of work in a message flow. There are additional logical units of work during MQ transport, and in those applications that consume the requests and send replies to the fan-in flow.

Composing an FTE reply message from aggregated replies

The aggregation message arrives on the output terminal of `AggregateReply` node. It consists of multiple messages under the `ComblmAggregateReplyBody` root element. Notice that all messages can be in different parser domains, and they appear in the same order as they arrive at the input terminal of the `AggregateReply` node, which might not be the order you expected. Note that we have a `REPORT` folder for replies from the `RDW` application and several `OPERATION` folders, one for each reply from the `VAS` application, as shown in Figure 8-41 on page 433. (Note that Figure 8-41 on page 433 was captured from the debugger in Message Broker Toolkit after a breakpoint was set right after the `AggregateReply` node).

Name	Value
Message	
Properties	
ComIbmAggregateReplyBody	
OPERATION	
Properties	
MQMD	
XMLNSC	
branch	
branch_number	1
date	1.3.2012
operation	
transactionID	1917
symbol	IBM
price	1.901E+2
amount	20
commission1	3.8E-1
commission2	3E-2
DB	
Id	754
result	00000000-0000-0000-0000-000000000000
OPERATION	
OPERATION	
REPORT	
Properties	
MQMD	
DFDL	
FTERRecordSample	
branch	
customer	
report	
LocalEnvironment	
Environment	
ExceptionList	

Figure 8-41 AggregateReplyBody tree

To construct a DFDL message from all of the replies, we use ESQL to deal with complex message structures.

Double-click the Combine Results node to view the code and replace the original content of the FTESampleFlow_Combine_Results module with the text in Example 8-11.

Example 8-11 ESQL module FTESampleFlow_Combine_Results

```

CREATE COMPUTE MODULE FTESampleFlow_Combine_Results
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
  CALL CopyEntireMessage();

  declare report_folder reference to
InputRoot.ComIbmAggregateReplyBody.REPORT;
  set OutputRoot.ComIbmAggregateReplyBody = null;
  create lastchild of OutputRoot domain 'DFDL' name 'DFDL';
  set OutputRoot.Properties.MessageType = 'FTERRecordSample';
  set OutputRoot.DFDL.FTERRecordSample = report_folder.DFDL.FTERRecordSample;
  create previoussibling of OutputRoot.DFDL.FTERRecordSample.report name
'operations';
  set OutputRoot.DFDL.FTERRecordSample.operations.operation[] = select
    0.XMLNSC.branch.operation.transactionID as transactionID,
    0.XMLNSC.branch.operation.symbol as symbol,

```

```

        cast(0.XMLNSC.branch.operation.price as float) as price,
        cast(0.XMLNSC.branch.operation.amount as float) as amount,
        cast(0.XMLNSC.branch.operation.commission1 as float) as commission1,
        cast(0.XMLNSC.branch.operation.commission2 as float) as commission2,
        0.XMLNSC.branch.operation.result as registeredID
    from
        InputRoot.ComIbmAggregateReplyBody.OPERATION[] AS O;

    set OutputLocalEnvironment.Destination.FTE.DestinationAgent = 'ITSO_B';

    declare FTE_Params reference to OutputLocalEnvironment.Destination.FTE;
    set FTE_Params.DestinationQmgr = 'QM_FTE';

    set FTE_Params.Directory = 'c:\ITSO\agent_b_out';

    set FTE_Params.Name = THE(
        SELECT ITEM F.file_name
        FROM Database.FILE_STATUS as F
        WHERE F.report_msg_id = cast(report_folder.MQMD.CorrelId as CHAR));

    set FTE_Params.Overwrite = true;

    RETURN TRUE;
END;

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

```

1. In this module, we first copy content from the input to the output message and cut a ComIbmAggregateReplyBody subtree from it.
2. We then create the FTERecordSample root element and copy to it all elements from the REPORT folder message body.
3. An interesting part of this code is done by the select ESQL statement. This select takes all OPERATION folders and extracts from them every child of the operation element. The folders are then taken into the destination operation subtree with the same names except the result element, which is renamed to registeredID to comply with FTERecordSample message definition.
4. The remaining code dynamically overrides the parameters for the FTEOutput node in order to send messages to the ITSO_B FTE agent.
5. The names of the destination files are the same as the source names and are taken from the database.

Note: You have to specify a remote queue manager name for your transfer, even if your source agent's queue manager and coordination queue manager are the same. This is why we specified QM_FTE as the DestinationQmgr.

8.5.6 Producing multiple messages from the compute node

Double-click the **Route node** and replace the original content of the FTESampleFlow_Route module with the text in Example 8-12 on page 435.

Example 8-12 ESQL Module FTESampleFlow_Route

```
CREATE COMPUTE MODULE FTESampleFlow_Route
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    CALL CopyEntireMessage();

    set OutputRoot.DFDL.FTERecordSample.operations = null;
    propagate to terminal 'out' delete none;

    set OutputRoot.DFDL.FTERecordSample.report = null;
    declare operation reference to
InputRoot.DFDL.FTERecordSample.operations.operation[1];
    set OutputRoot.DFDL = null;
    set OutputRoot.XMLNSC.branch = InputRoot.DFDL.FTERecordSample.branch;

    while lastmove(operation) do
      set OutputRoot.XMLNSC.branch.operation = operation;
      set OutputRoot.XMLNSC.branch.operation.DB = Environment.DB;
      propagate to terminal 'out1' delete none;
      move operation nextsibling repeat type name;
    end while;

    RETURN FALSE;
  END;

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

The code in this module first copies the entire content from the input message to the output message, and then it deletes the operations subtree (see the logical model in Figure 8-36 on page 425). Next, the module sends the message with the report part to the out terminal using the propagate to terminal statement.

After that, we used an operation cursor (reference) in a while cycle with a move operator to build a portion of source message for each operation and route these parts to the terminal out1.

At the end, the code returns false which tells the broker that the message has been handled properly and it does not need to be propagated to the default out terminal again.

8.5.7 Accessing databases from the database nodes

The database node is a general purpose place to execute operations against a database. You cannot modify content of the message in this node like you can in a compute node, for example, but you can execute selects, inserts, updates, and other SQL statements.

In our scenario, we use a database node for tracing, for storing the state of the flow, and for correlating responses with original information. For instance, we assume that the RDW application has a predefined XML format that does not allow us to put a file name information in the message. So upon receiving a reply from RDW, we pick up the file name information from the database using the message correlation information as a foreign key.

Log File status node

Double-click the Log File status node and replace the original content of the FTESampleFlow_Log_File_Status module with the text in Example 8-13.

Example 8-13 ESQL Module FTESampleFlow_Log_File_Status

```
CREATE DATABASE MODULE FTESampleFlow_Log_File_Status
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    Declare FTE_Params reference to LocalEnvironment.FTE.Transfer;
    Declare FTE_Rec reference to Root.DFDL.FTERecordSample;
    INSERT INTO Database.FILE_STATUS (file_name, parts_in_total)
    VALUES (FTE_Params.Name, cardinality(FTE_Rec.operations.operation[]) + 1);

    Declare ids ROW;
    set ids = PASSTHRU('Select id FROM FILE_STATUS WHERE id = @@IDENTITY');
    Set Environment.DB.Id = ids.id;
    RETURN TRUE;
  END;
END MODULE;
```

This code inserts general information about the received file into the FILE_STATUS table and stores an auto-generated value from the id column in the Environment.DB.Id subtree for later use.

To get file information, such as the file name, the code uses metadata stored by the FTEInput node at the LocalEnvironment.FTE.Transfer subtree. The code also counts the number of operations contained within the message using the cardinality function.

Store MsgId node

Double-click the Store MsgId node, and replace the original content of the FTESampleFlow_Store_MsgId module with the text in Example 8-14.

Example 8-14 ESQL Module FTESampleFlow_Store_MsgId

```
CREATE DATABASE MODULE FTESampleFlow_Store_MsgId
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    Declare MQ_Id BLOB
    LocalEnvironment.WrittenDestination.MQ.DestinationData.msgId;
    Update Database.FILE_STATUS as F
      Set report_msg_id = cast(MQ_Id as CHAR)
      Where F.id = Environment.DB.Id;
    RETURN TRUE;
  END;
END MODULE;
```

This code stores the information about the message that was sent to the RDW application into the database. The msgId of the message is generated by the MQOutput node and put into the LocalEnvironment.WrittenDestination.MQ subtree. The primary key for this operation is taken from the Environment.DB subtree where it is stored by the Database node, as described in “Log File status node” on page 436.

Update Reps Status node

Double-click the Update Reps Status node, and replace the original content of the FTESampleFlow_Update_Reps_Status module with the text in Example 8-15.

Example 8-15 ESQL Module FTESampleFlow_Update_Reps_Status

```
CREATE DATABASE MODULE FTESampleFlow_Update_Reps_Status
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    Declare corrID CHAR Root.MQMD.CorrId;
    UPDATE Database.FILE_STATUS as F
      SET parts_processed = F.parts_processed + 1, last_modified =
CURRENT_TIMESTAMP
    WHERE
      F.report_msg_id = corrID;
    RETURN TRUE;
  END;

END MODULE;
```

The MQReply node in the emulated applications returns the original msgId information in the CorrelId field of the message. This is the default behavior of an MQReply node. The CorrelId is used in the AggregateReply node. In the Update Reps Status database node, we also use the CorrelId information to update the number of received replies in FILE_STATUS table.

Update statement peculiarities: Note that you cannot use F. alias of the FILE_STATUS table at the left side of the assignment in the SET operator. But at the same time, you have to use F. alias at the right side of the assignment. This is a syntax rule of the ESQL update statement.

Update Ops status node

Double-click the Update Ops status node, and replace the original content of the FTESampleFlow_Update_Ops_Status module with the text in Example 8-16.

Example 8-16 ESQL Module FTESampleFlow_Update_Ops_Status

```
CREATE DATABASE MODULE FTESampleFlow_Update_Ops_Status
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    UPDATE Database.FILE_STATUS as F
      SET parts_processed = F.parts_processed + 1, last_modified =
CURRENT_TIMESTAMP
    WHERE
      F.id = Root.XMLNSC.branch.operation.DB.Id;
    RETURN TRUE;
  END;

END MODULE;
```

This node performs the same actions that the Update Reps Status node performs; however, in this case, the original primary key ID information is returned by the VAS emulator OperationApplicationClient message flow. Because OperationApplicationClient is emulating a broker-managed wrapper for a real .NET client, we can use any format to interact with it, so we put information about the database ID into the message payload of the request and send it back in response.

Complete File status node

Double-click the Complete File Status node and replace the original content of the FTESampleFlow_Complete_File_Status module with the text in Example 8-17.

Example 8-17 ESQL Module FTESampleFlow_Complete_File_Status

```
CREATE DATABASE MODULE FTESampleFlow_Complete_File_Status
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
    Declare corrID CHAR Root.ComIbmAggregateReplyBody.REPORT.MQMD.CorrId;
    UPDATE Database.FILE_STATUS as F
        SET completed = 1, last_modified = CURRENT_TIMESTAMP
    WHERE
        F.report_msg_id = corrID;
    RETURN TRUE;
END;

END MODULE;
```

In the Complete File Status node, we pick up CorrelId information from the REPORT folder of the aggregated reply message. We then use CorrelId to update the number of received replies in FILE_STATUS table.

8.5.8 Creating the queues for the FTESample application

Execute the two commands in Example 8-18 from the command line to create the queues for replies from the RDW and VAS application emulators.

Example 8-18 DEFINE QLOCAL commands

```
echo DEFINE QLOCAL(OPTN.IN) | runmqsc QM_ITSO
echo DEFINE QLOCAL(REPT.IN) | runmqsc QM_ITSO
```

8.6 Running the scenario

Use the following procedure to run the scenario:

1. Deploy all three applications to BKR_ITSO, as shown in Figure 8-42.

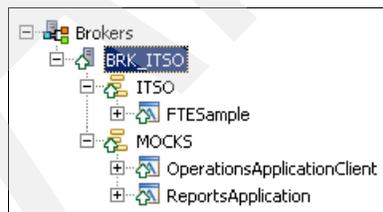


Figure 8-42 Brokers view

2. Ensure that a file transfer exists in the Pending Transfers category of the Managed File Transfer subtree in WebSphere MQ Explorer.
3. Create a file named branch_file.txt from Example 8-10 on page 428, and save it in the c:\ITSO\ folder, if it is not already created.

- Copy this file into to the c:\ITSI\agent_a_out directory and wait about one minute for the file to process. If everything goes right, you will see the two entries shown in Figure 8-43 in the Transfer log category of Managed File Transfer.

Job Name	Source	Destination	Completion State	Started (America/New_York)
ITSO	ITSO_A	BRK_ITSO.ITSO	Successful	3/29/12 4:37:36 PM EDT
ITSO	BRK_ITSO.ITSO	ITSO_B	Successful	3/29/12 4:37:37 PM EDT

Figure 8-43 File transfers succeeded

The first line tells you that file was successfully transferred from ITSO_A agent to the brokers embedded agent BRK_ITSO.ITSO. The second log entry tells you that the file was delivered from BRK_ITSO.ITSO to the ITSO_B FTE agent, so it successfully passed through the FTESample message flow. Notice the ITSO job name is the result of the work of the ESQLE module described in “Composing an FTE reply message from aggregated replies” on page 432.

Note: The set of columns in Figure 8-43 differs from the default setup. You can customize the columns using the Configure Agent Status Columns icon in the toolbar at the right-top corner of WebSphere MQ Explorer window.

- Look at the file branch_file in the c:\ITSO\agent_b_out directory. It must be similar to Example 8-19.

Example 8-19 Transformed version of branch_file.txt

```

BRANCH#1, DATE 1.3.2012
CUSTOMER ID: R90306
OPERATIONS
  1917, IBM, 190.1, 20, 0.38, 0.03, 00000000-0000-0000-0000-000002001917
  1945, IBM, 190.5, 20, 0.38, 0.03, 00000000-0000-0000-0000-000002001945
  1976, IBM, 190.99, 10, 0.38, 0.03, 00000000-0000-0000-0000-000001001976
END OF OPERATIONS
REPORT
STOCKS
  IBM, 100, 150, 50
END OF STOCKS
COMMISSION
  1.14, 0.09, 1.23
CASH
  1000000, 990476.87, -9523.13

```

Notice the recordedID added at the end of each operation in bold, which means that the VAS emulator successfully executed the .Net operation within OperationsAppClient.dll.

8.6.1 Troubleshooting tips

If something goes wrong, we recommend that you first check the Windows Application events in the Events Viewer of the Administrative Tools for any specific errors from the broker. Starting with the Event Viewer is a useful practice because the viewer collects all different types of errors, such as database connection failures, WebSphere MQ communication problems, messages rolled back to queues, and so on.

Another place to look is the WebSphere MQ Explorer. Because we have not implemented any error handling mechanisms, messages can become stuck in the message queues in the

event of an error. If this happens, stop all of the flows, and clean up the queues. Next, analyze the root cause of the error again by looking for the events in the Event Viewer.

8.6.2 Exploring the work of the scenario

In this section, we use the debugger in the Message Broker Toolkit to explore how the message moves through the flow.

Setting up debug

Right-click the **ITSO execution group**, and select **Launch Debugger**. Message Broker Toolkit will ask you to enter the value for the debug port. You can enter, for example, 14111 or another value you think is suitable, and click **OK**.

Examining the Route node

You can use breakpoints to examine how messages flow through the Route node:

1. Add break-points to your flow by right-clicking the **Route** node and choosing **Add Breakpoints After Node**. Wait a minute for repeating transfers to occur again. You will see that this time the flow execution stops between the Route and Reports nodes.
2. Examine the content of the message using the Variables view. You can see that there is only a report section there. Press **F8** or the **Resume** button in the Debug view, and notice that the next flow stops between the Route and Operations nodes.
3. Examine the message structure again and notice the difference. You can then continue using **F8** to continue through the flow, ensuring that it acts as you expect.
4. Remove the breakpoints by right-clicking them and choosing the **Remove breakpoint** option.

Examining the Aggregate Reply node

You can use breakpoints to examine how messages flow through the Aggregate Reply node:

1. Add breakpoints before and after the Aggregate Reply node and wait until replies from RDW or VAS arrive. Delete the `branch_file.txt` from the `agent_a_in` folder to stop the repeatable transfers.
2. Examine the message structure in the Message Broker Toolkit's Debug perspective, and press **F8** to resume the message flow execution. Notice that there are three replies from the VAS application and one from RDW in total.
3. After processing all of the replies, the message flow proceeds to the output terminal of the Aggregate Reply node. Examine the message structure at this point. It should contain all four replies.
4. Step over the Complete File Status and Combine Results nodes by pressing **F6 (Step over)** twice. Examine the message structure at the output terminal of the Combine Results node and notice the logical structure of the result message. You should also see under the LocalEnvironment subtree an overrides values for the FTE Output node.

Examining the logging process

You can put breakpoints after each database node to trace the flow from a database perspective. When the flow stops, use a database utility to execute a `SELECT FROM` statement with the `Id` field. This value is stored in `Environment.DB.Id` field in the Fan-out part of the flow, and can be seen from the Variables view. See Example 8-20 on page 441.

Example 8-20 Select for FILE_INFO table

```
select * from FILE_STATUS WHERE id = **YOUR_VALUE_HERE**
```

8.7 Extending the scenario

For simplification, we did not implement error handling in this scenario. For production-ready systems, error handling is a required functionality, so add error handling logic in your FTE integration message flows.

Another topic that was not covered in this chapter is a new function in this version of the broker to produce monitoring events from the flow without coding the events. You can use a configuration of the node properties to select information to be stored in the database. This functionality can be used instead of coding the multiple database nodes we have in our flow. Before WebSphere Message Broker Version 8, that kind of functionality was typically reinvented for each integration project. Now, this functionality is built into the broker.

Archived



Integrating file transfer using Sterling Connect:Direct with your message flow

This scenario illustrates the use of WebSphere Message broker to fetch a file transferred by a remote Sterling Connect:Direct server, process multiple transactions in the file one-by-one, validate each transaction, log the request into a file, and process them to its destination.

Specifically, we cover the following topics in this chapter:

- ▶ Scenario overview
- ▶ Overview of the Sterling Connect:Direct
- ▶ Using WebSphere Message broker with Connect:Direct
- ▶ Preparing the Connect:Direct environment
- ▶ Configuring WebSphere Message Broker

Additional materials: The WebSphere Message Broker project interchange file and the .NET class code for this scenario can be downloaded from the IBM Redbooks publication web site. See Appendix A, “Additional material” on page 485 for more information.

9.1 Scenario overview

A banking firm has a number of branches that handle the day-to-day banking transactions of their customers. These transactions can be a utility bill payment, an order of fixed deposits, a withdrawal request, a deposit request, and so on. At the end of the business day, the bank branch generates a report file containing a consolidated list of transactions for their clearance. Sometimes these reports are generated immediately based on the business need and sent to the core banking system for clearance. The consolidated report file is sent to the Sterling control center system.

The Sterling control center collects and manages files arriving from different branches. A WebSphere Message Broker flow at the head branch office receives each file that arrives at the Sterling control center. In this solution, the Sterling Control center is an intermediate system between different branches and the head branch office.

The message flow receives the file and parses each transaction record in the file one-by-one. Each transaction record is logged into a data warehouse using WebSphere MQ and validated in the message flow. If the validation is successful, the transaction record is sent to the core banking system for clearance. The message broker receives the response from the core banking system for each transaction request and sends a reply back to the requested branch as a report file. See Figure 9-1.

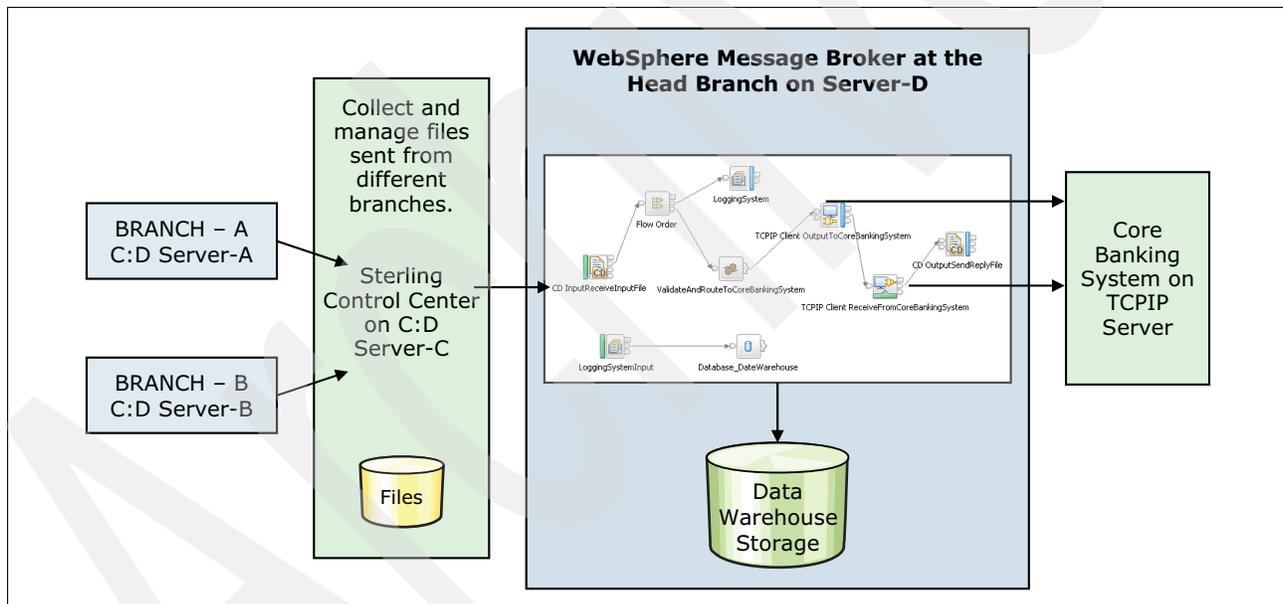


Figure 9-1 WebSphere Message Broker integration with Sterling Connect:Direct scenario

This scenario assumes that the following components are installed and configured:

- ▶ Server-A:
 - IBM Sterling Connect:Direct V4.5.0.1 is installed and configured a node called BRANCH-A.
- ▶ Server-B
 - IBM Sterling Connect:Direct is installed and configured a node called BRANCH-A.
- ▶ Server-C
 - IBM Sterling Connect:Direct has been installed and configured a node called CENPROC.

- ▶ Server-D:
 - WebSphere MQ is installed. A WebSphere MQ queue manager is defined and a local queue named LOGQ is created.
 - WebSphere Message Broker V8 is installed. A broker named ITSOBROKER is created.
 - IBM DB2 is installed.

9.2 Overview of the Sterling Connect:Direct

Sterling Connect:Direct is the point-to-point file transfer software optimized for high-volume, secure, and assured delivery of files within and among enterprises. Sterling Connect:Direct can deliver your files with:

- ▶ **Predictability:** Assures delivery through automated scheduling, checkpoint restart, and automatic recovery/retry.
- ▶ **Security:** Ensures that your customer information stays private and that your file transfers are auditable for regulatory compliance through a proprietary protocol, authorization, and encryption (FIPS 140-2, and Common Criteria certified).
- ▶ **Performance:** Handles your most demanding loads, from high volumes of small files to multi-gigabyte files.

Figure 9-2 on page 446 shows how Connect:Direct Nodes can communicate with each other using Requester or the Connect:Direct client.

Peer-to-Peer and Client/Server Concepts in Connect:Direct

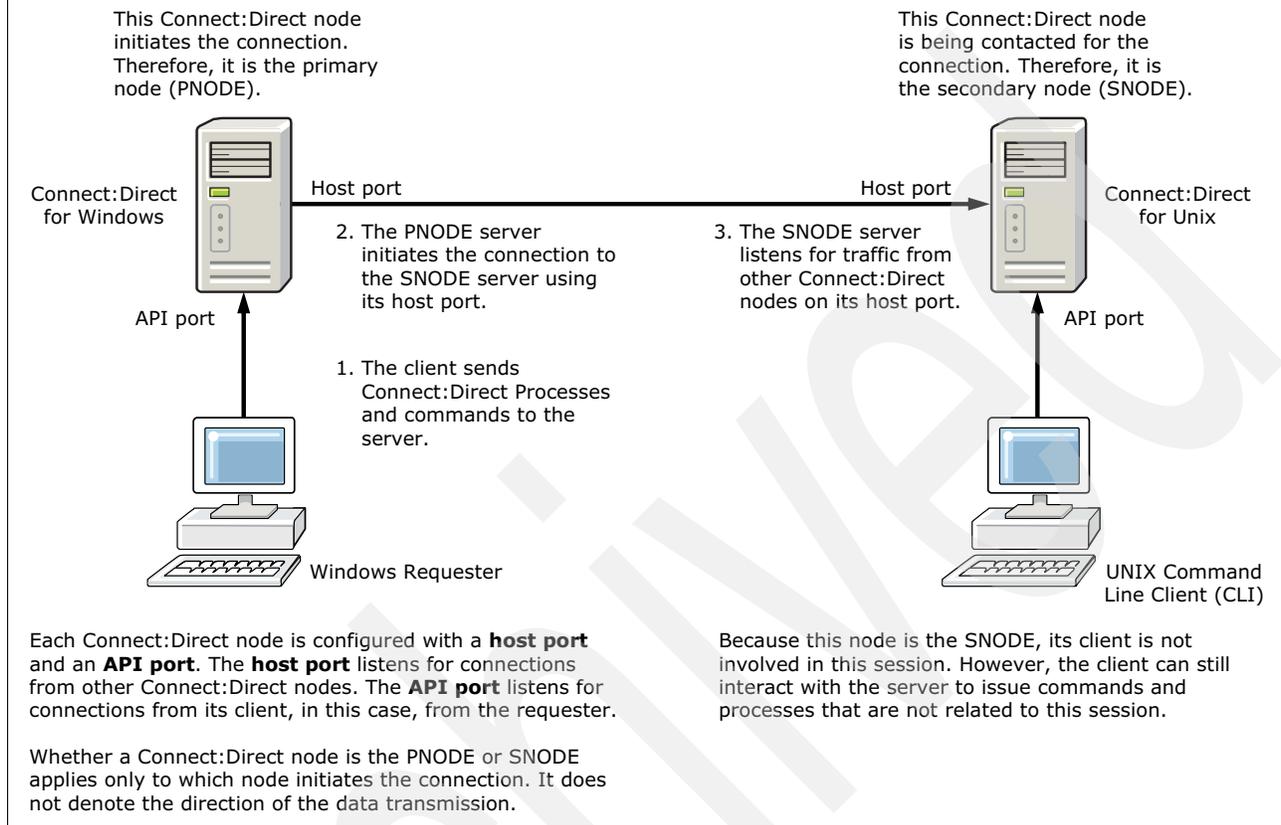


Figure 9-2 Connect:Direct nodes

Each Connect:Direct node is configured with a host port having a default value of 1364 and an API port having the default value of 1363. The host port listens for a connection from other Connect:Direct nodes. The API port listens for connections from its client, called a Requester. Whether a Connect:Direct node is a primary node (PNode) or secondary node (SNode) is determined by which node initiates the connection. In Figure 9-2, steps 1, 2, 3 show how a client sends a command to transfer a file from one Connect:Direct node to another.

9.3 Using WebSphere Message broker with Connect:Direct

WebSphere Message Broker can receive a file when any local or remote Connect:Direct file transfer is performed. Similarly WebSphere Message broker can send a file to a local or remote Connect:Direct server. Figure 9-3 on page 447 shows how a broker sends a file from a Windows Connect:Direct server to a UNIX Connect:Direct server using the CDOOutput Node in a message flow.

In Figure 9-3 on page 447, the WebSphere Message Broker CDOOutput node is a requester that sends a file transfer command to API port on Windows Connect:Direct. The Connect:Direct server on Windows acts as the PNode and does the file transfer to the UNIX Connect:Direct server (the SNode in this case). The message broker CDOOutput node needs

the SNODE name, host address, and API port to connect to the SNODE server and the destination file path for the transferred file.

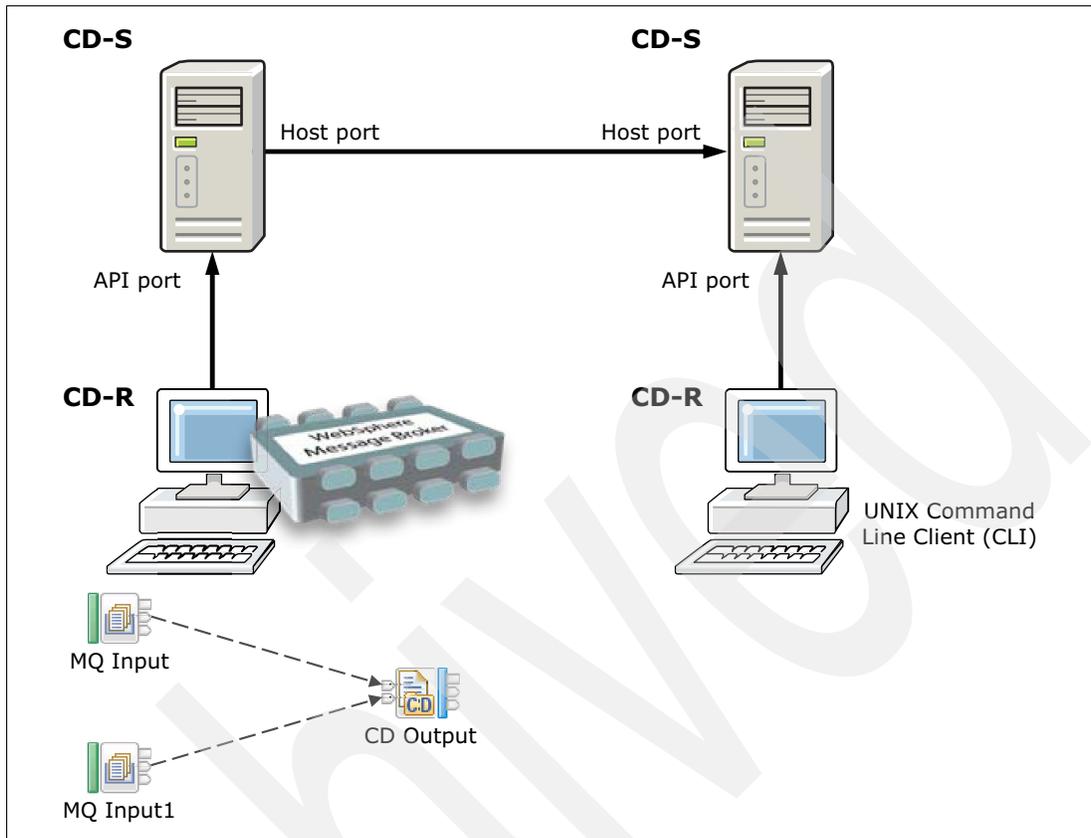


Figure 9-3 Message broker file send from Windows to UNIX

9.4 Preparing the Connect:Direct environment

The scenario in this section has two Connect:Direct server nodes at the branches, referred to as BRANCH-A and BRANCH-B. A third Connect:Direct server node that acts as the Sterling control center, referred to as CENPROC, resides on Server-C. All are Windows servers. An assumption is made that the Connect:Direct servers are installed and configured for local file transfers.

The following steps show the configuration required to enable a remote file transfer from BRANCH-A node to CENPROC node and from BRANCH-B node to CENPROC node:

1. Add a netmap entry for the CENPROC node in BRANCH-A's netmap:
 - a. In the CD Requester, right-click **BRANCH-A**, and select **Admin** → **Netmap**.
 - b. In the window that opens for NetMap, right-click and select **Insert**.
 - c. In the Netmap Node Properties window, select the **Main** tab, enter the node name, and choose the operating system from the drop-down for the remote node. In this scenario, the name is CENPROC, and the operating system is Windows, as shown in Figure 9-4 on page 448.

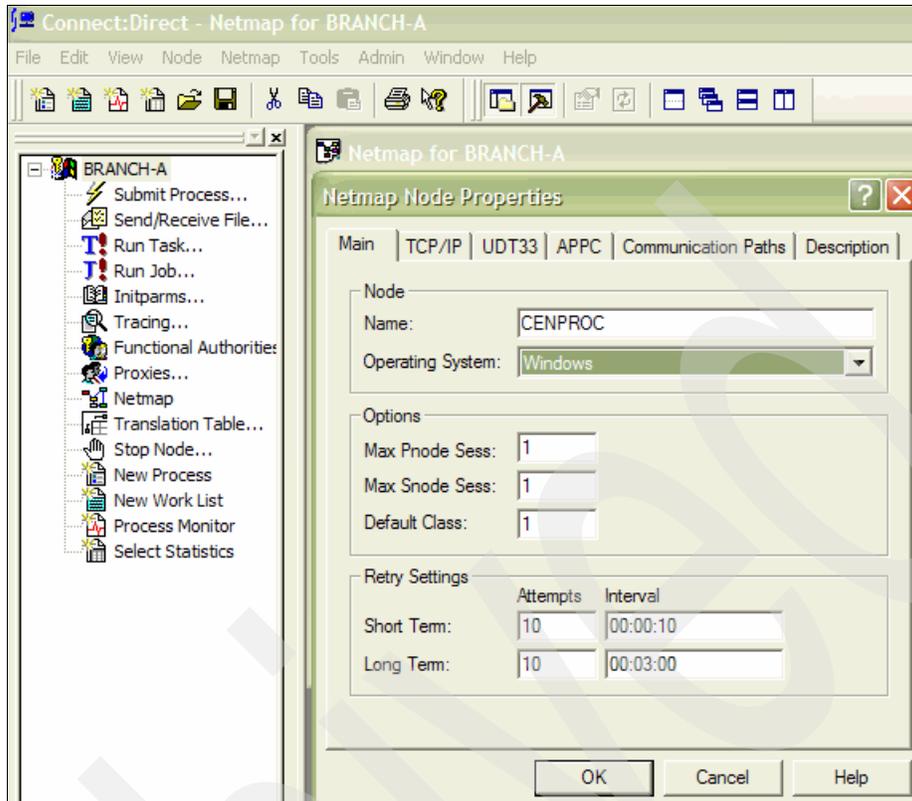


Figure 9-4 Netmap Main settings for Branch-A

- d. Select the **TCP/IP** tab, and enter the Host/IP Address. We used xxx.yyy.zzz.aaa, which is the IP address of Server-C, as shown in Figure 9-5 on page 449.

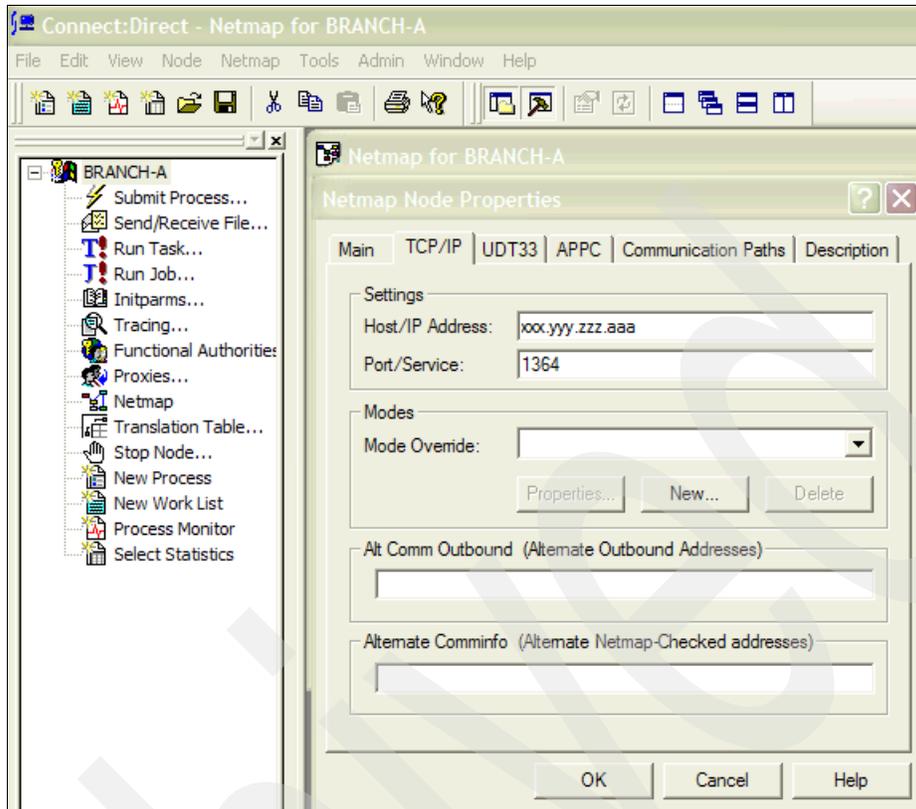


Figure 9-5 Netmap TCP/IP settings for Branch-A

- e. Select the **Communication Paths** tab, and select the existing TCPCommPath, as shown in Figure 9-6 on page 450.

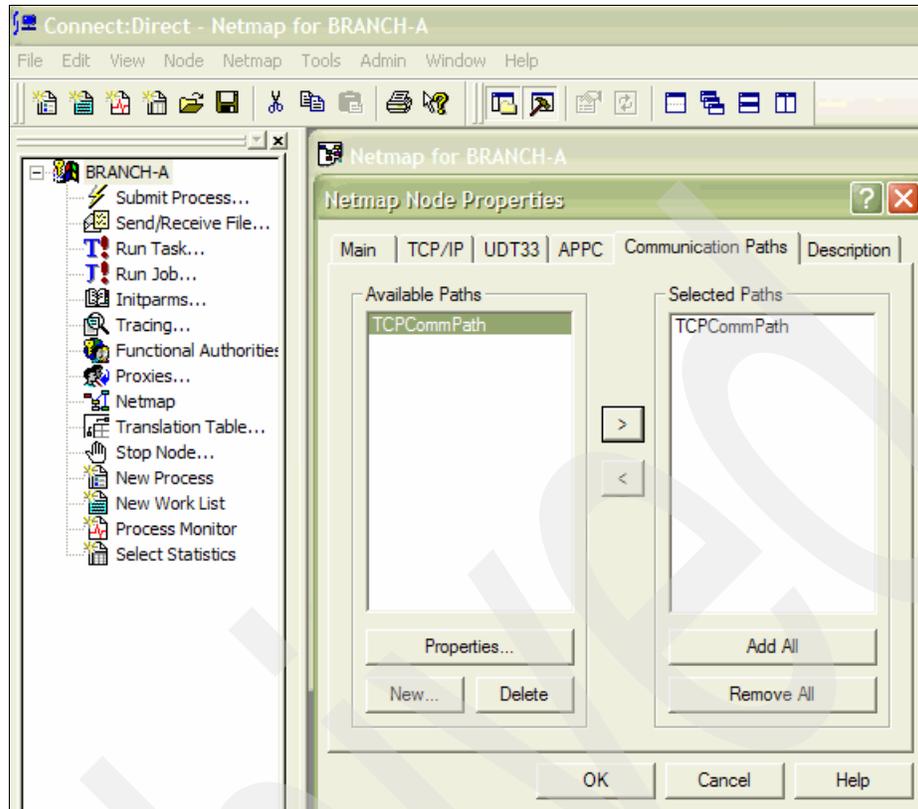


Figure 9-6 Netmap Communication Paths settings for Branch-A

- f. Click **OK**.
 - g. Right-click CENPROC, and click **Apply**.
2. Repeat step 1 to add a netmap entry for the CENPROC node in BRANCH-B's netmap.
 3. Add a netmap entry for BRANCH-A in CENPROC's netmap:
 - a. In the CD Requester, right-click **CENPROC**, and select **Admin** → **Netmap**.
 - b. In the window that opens for NetMap, right-click and select **Insert**.
 - c. Perform the following actions in the Netmap Node Properties window:
 - i. Under the Main tab, enter BRANCH-A as the node name, and select **Windows** as the operating system from the drop-down for the remote node (see Figure 9-7 on page 451).

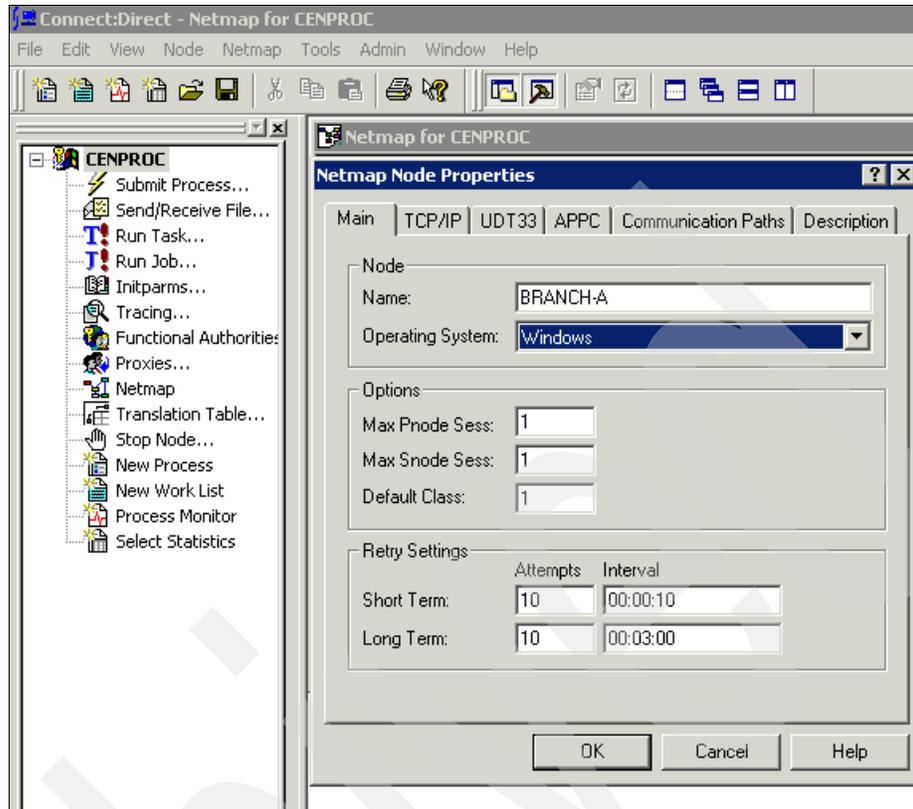


Figure 9-7 Netmap Main settings for Server-C

- ii. Select the TCP/IP tab, and enter the IP address for the BRANCH-A machine and the port number for BRANCH-A (1364), as shown in Figure 9-8 on page 452.

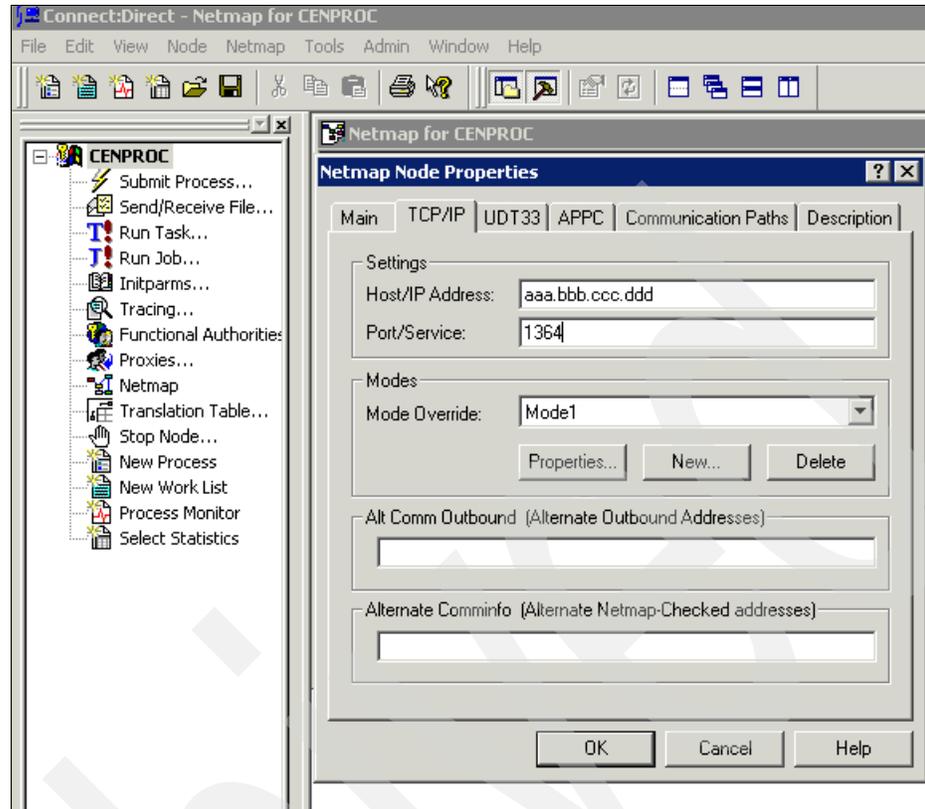


Figure 9-8 Netmap TCP/IP settings for Server-C

- iii. Select the **Communication Paths** tab, and select the existing TCPCommPath, as shown in Figure 9-9 on page 453.

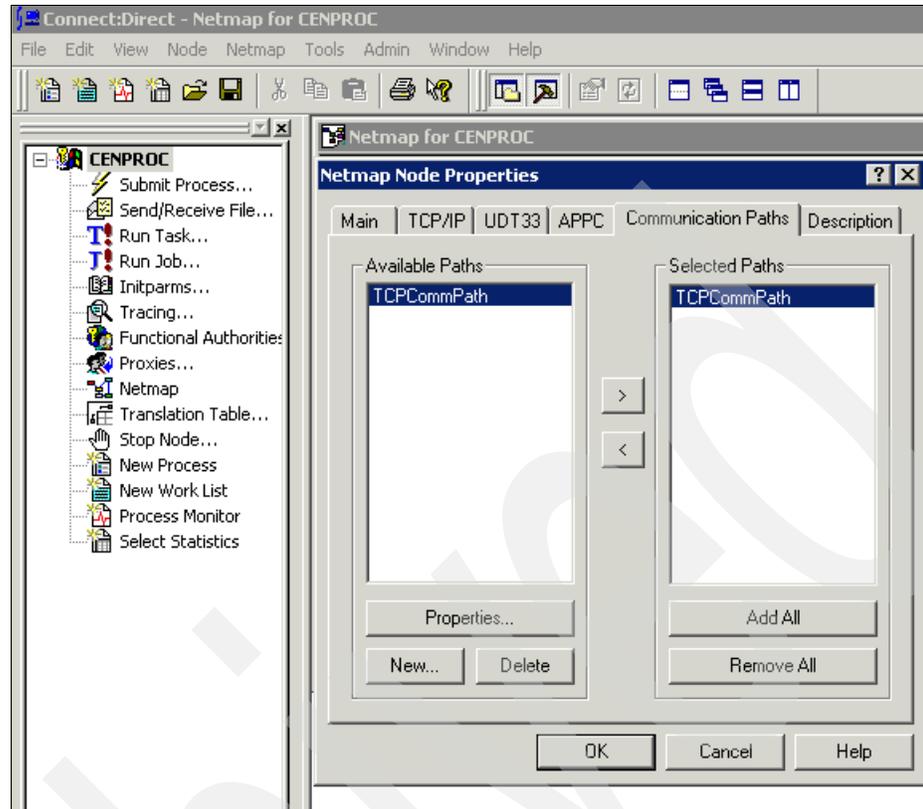


Figure 9-9 Netmap Communication Paths settings for Server-C

- d. Click **OK**.
 - e. Right-click CENPROC, and click **Apply**.
4. Repeat step 3 to add a netmap entry for BRANCH-B in CENPROC's netmap.

Now BRANCH-A and BRANCH-B are ready for remote file transfers with the Sterling control center node CENPROC.

Authorization: In this scenario, the same user ID and password used for the transfers is defined at all three systems. If the user ID and password are different on the systems, you either need to add functional authorities or specify the system user ID and password of the primary and secondary nodes in the CONNECT:DIRECT process definition.

9.5 Preparing the logging database

The message flow logs transactions to a DB2 database. Use these instructions to prepare the database:

1. Create the database ITSODB in DB2 using the Control Center. See Figure 9-10 on page 454.

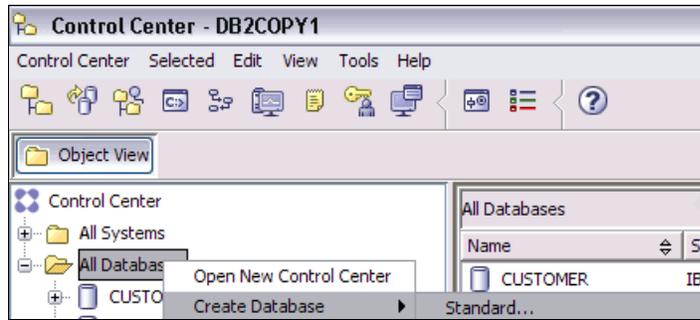


Figure 9-10 Creating the database

Alternatively, you can use the following DB2 command to create the database:

```
C:\>db2 create database ITSODB
```

2. Make sure the DSN name is created using ODBC Data Source Administrator. See Figure 9-11.

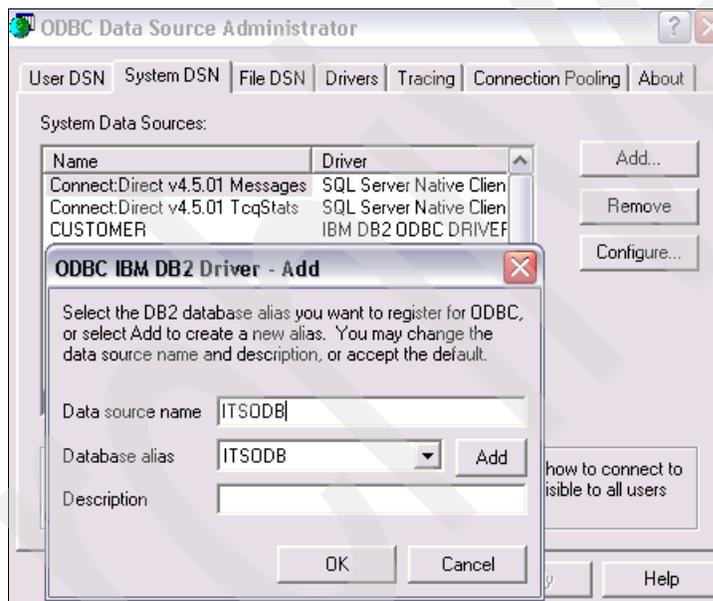


Figure 9-11 Creating the ODBC driver

3. Create the TRANSLOG table. Select **Tables** → **Create**, as shown in Figure 9-12 on page 455.

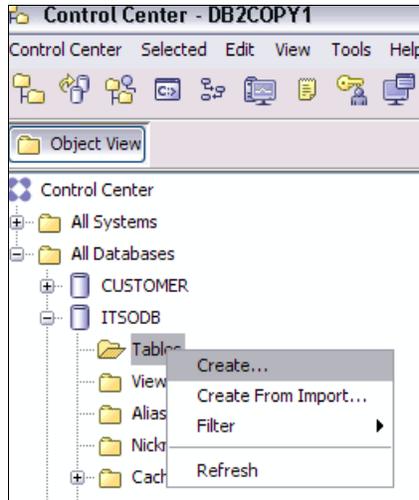


Figure 9-12 Creating a table

4. Select a Table schema and provide a Table name, as shown in Figure 9-13.

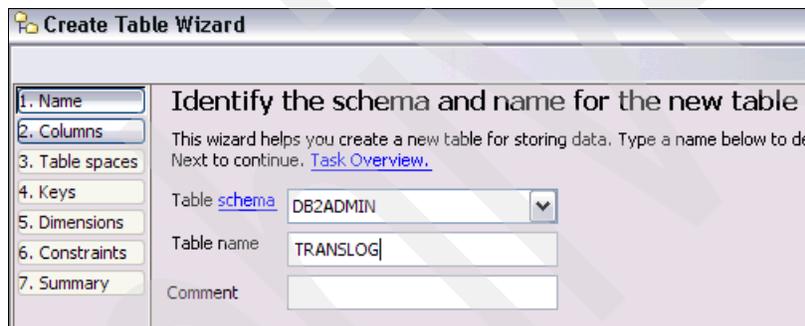


Figure 9-13 Defining a table schema and name

5. Add the columns shown in Figure 9-14 to the table.

Table - TRANSLOG						
Schema	: DB2ADMIN	Columns				
Creator	: DB2ADMIN	Key	Name	Data type	Length	Nullable
Columns	: 8					
Actions:						
Open						
Query						
Show Related Objects						
Create New Table						
			NAME	CHARACTER	10	No
			ACCOUNT_NO	BIGINT	8	No
			TO_ACCOUNT_NO	BIGINT	8	No
			TO_BRANCH_NAME	CHARACTER	10	No
			TO_IFSC_CODE	INTEGER	4	No
			AMOUNT	BIGINT	8	No
			DATE	DATE	4	No
			TRANS_ID	BIGINT	8	Yes

Figure 9-14 Adding columns to a table

Alternatively, you can use an equivalent SQL command to create the table, as shown in Example 9-1. In this example, the system user is db2admin. Replace it with your own system user.

Example 9-1 SQL command for creating a table

```
CONNECT TO ITSODB USER "db2admin";
```

```
CREATE TABLE DB2ADMIN.TRANSLOG ( NAME CHARACTER (10) NOT NULL , ACCOUNT_NO
BIGINT NOT NULL , TO_ACCOUNT_NO BIGINT NOT NULL , TO_BRANCH_NAME CHARACTER
(10) NOT NULL , TO_IFSC_CODE INTEGER NOT NULL , AMOUNT BIGINT NOT NULL ,
DATE DATE NOT NULL , TRANS_ID BIGINT NOT NULL , CONSTRAINT CC1332485129716
PRIMARY KEY ( TRANS_ID) ) ;
CONNECT RESET;
```

9.6 Configuring WebSphere Message Broker

The next step is to configure the WebSphere Message Broker on Server-D to process the files coming into the CENPROC node. This includes creating a configurable service and creating the message flow.

9.6.1 Creating a configurable service on WebSphere Message Broker

In this scenario, WebSphere Message Broker and the CENPROC node are on different machines, so a configurable service must be created on WebSphere Message Broker to process remote file transfers. Before starting the configuration, ensure that the directory to which file transfers occur on Server-C is mounted and is accessible on the machine where WebSphere Message Broker is installed. In this scenario, the file transfers go to the C:\EnterpriseDataCenter\TransactionFiles directory on Server-C. So, the TransactionFiles directory is shared.

The `brokerPathToInputDir` key of the CDServer configurable service then takes the value:

```
//IP Address of Server-C>/TransactionFiles
```

On UNIX systems, this value is the mounted path, and the IP address is not required, for example: `/mnt/TransactionFiles`, where `mnt` is the mount point on the broker system.

The `cdPathToInputDir` key takes the value `C:\EnterpriseDataCenter\TransactionFiles`.

Use WebSphere Message Broker Explorer to create the configurable service, as described in this procedure:

1. In the IBM WebSphere MQ Explorer, right-click **broker_name** → **Configurable Services**, and select **New** → **Configurable Service**.
2. In the window that opens, enter a Name, for example `CDConfigurableService`.
3. Select **CDServer** for the Type from the drop-down.
4. Enter the appropriate values for the keys in the table, as shown in Figure 9-15 on page 457. In this example:
 - `brokerPathToInputDir`: *IP Address of Server-C*\TransactionFiles
 - `cdPathToInputDir`: `C:\EnterpriseDataCenter\TransactionFiles`
 - `hostname`: *IP Address of Server-C*
 - `port`: 1363
 - `securityIdentity`: `cenproc`

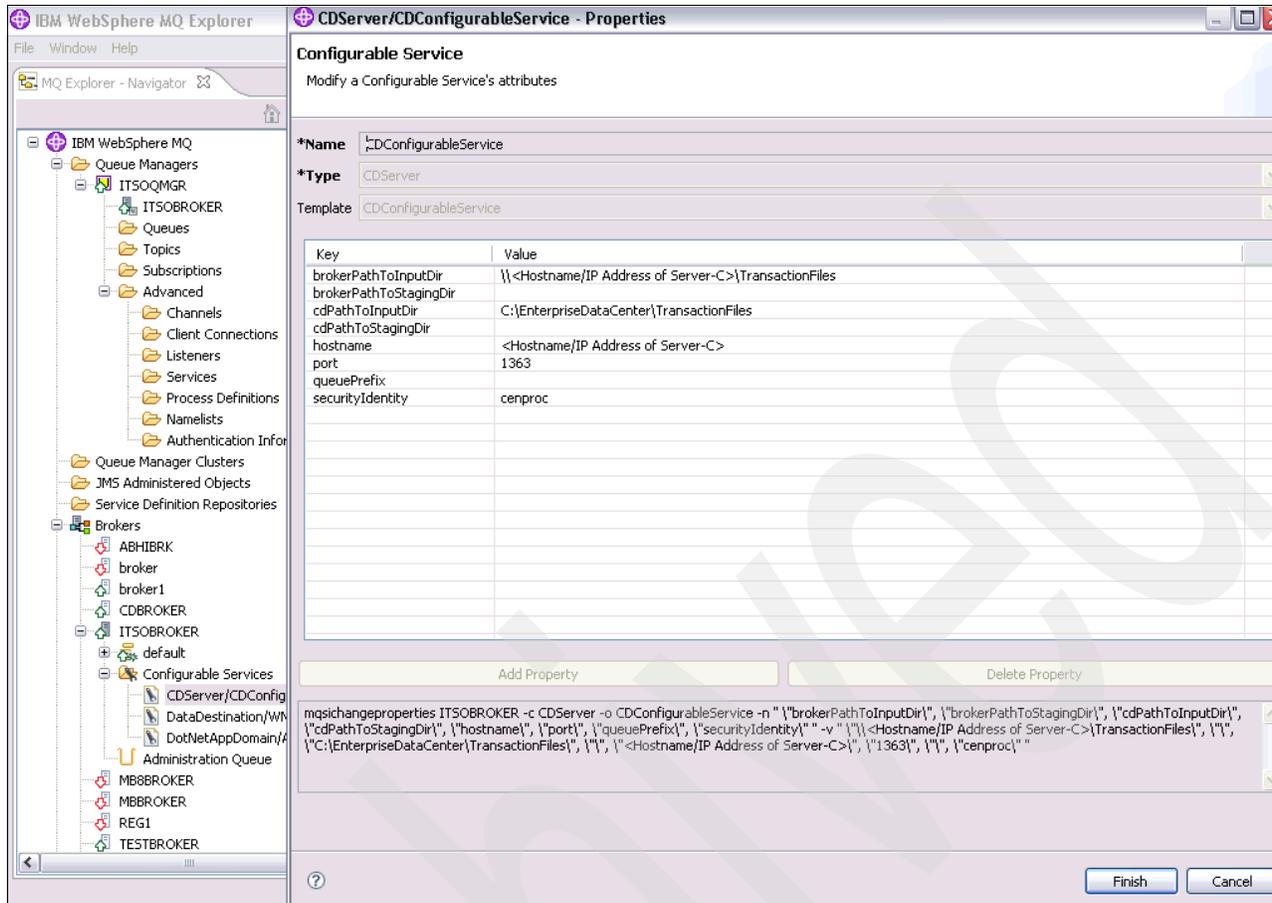


Figure 9-15 Creating Configurable Services for Broker on Server-D

Creating the service with a command: An alternative to using the WebSphere Message Broker Explorer to create the service is to use the following command:

```

mqsicreateconfigurableservice broker name -c CDServer -o
CDConfigurableService -n brokerPathToInputDir, brokerPathToStagingDir,
cdPathToInputDir, cdPathToStagingDir, hostname, port, queuePrefix,
securityIdentity -v IP Address of Server-C\TransactionFiles, ,
C:\EnterpriseDataCenter\TransactionFiles, , IP Address of Server-C, 1363, ,
cenproc

```

5. Configure the securityIdentity CENPROC used in the CDConfigurableService. Open a command window, and enter the following command with the necessary values:

```

mqsisetdbparms BrokerName -n CD::cenproc -u username-of-Server-C-machine -p
password-of-Server-C-machine

```

The value for *username-of-Server-C-machine* is the system user name of Server-C-machine where the CD Server is started.

Note: The default configurable service is used and default securityIdentity setup is needed as the local file transfers are set up. For default securityIdentity, use the following command with the proper values:

```
mqsisetdbparm <BROKERNAME> -n CD::default -u <username-of-broker-machine> -p <password-of-broker-machine>
```

9.6.2 Creating the message flow

The next step is to create and configure the message flow to process the files coming to the CENPROC Connect:Direct node. The message flow is shown in Figure 9-16.

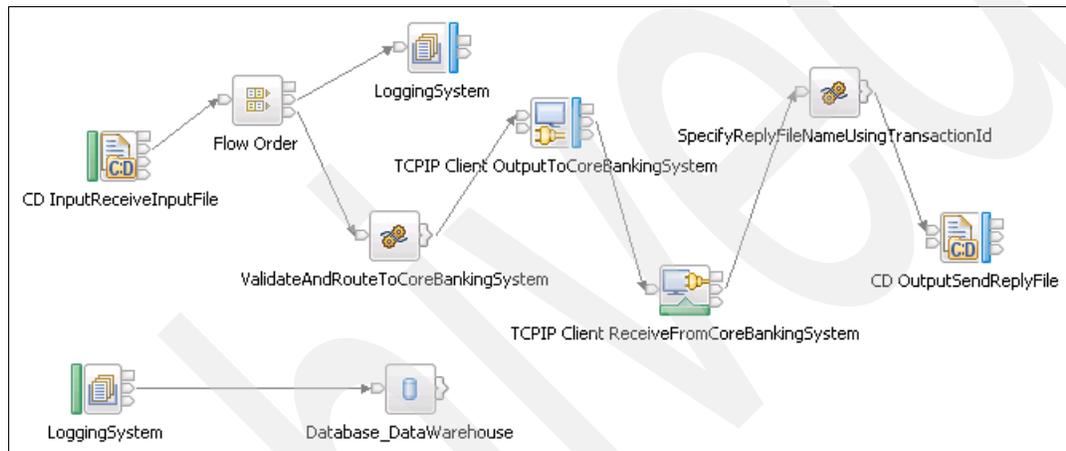


Figure 9-16 MessageFlow

1. A file containing one or more transaction records is transferred to a file system on Server-C. Here each transaction record in the file is in XML format and is delimited by end of line character.
2. The CD InputReceiveInputFile node receives the file from that location and parses the file, using the end of line delimiter to determine where each record ends.
3. Each transaction record is sent to the Flow Order node:
 - a. The flow order node first sends the transaction to the LoggingSystem node, which puts the transaction on a message queue. The transaction is then picked by another flow to insert it into the Data warehouse.
 - b. The flow order node then sends the transaction to the next node ValidateAndRouteToCoreBankingSystem for validation.
4. The ValidateAndRouteToCoreBankingSystem node validates the transaction and sends the transaction to the next node.
5. The TCPIPClientOutput node sends the transaction out of the flow to the core banking system, which is simulated by a separate message flow (not shown in Figure 9-16).
6. When the core banking system completes its processing, it sends back a successful response to the TCP/IP Client ReceiveFromCoreBankingSystem node. The transaction is then sent to the next node.

7. The SpecifyReplyFileNameUsingTransactionId Compute node creates a name for the transaction file based on the transaction ID.
8. The CDOutput node transfers each successful transaction message into a report file with to the Connect Direct Server on the WebSphere Message Broker system.

9.6.3 Creating the main message flow

Develop the message flow shown in Figure 9-16 on page 458 for the scenario, and configure it using the following steps:

1. Open the Message Broker Toolkit.
2. Create a new application called CDdemo.
3. Create a new message flow called FileTransferCDToMsgFlow.
4. Add the following nodes to the message flow, rename them, and wire them as indicated in Table 9-1.

Table 9-1 Message flow nodes

Palette section	Node type	Node name	Terminal x Wired to node y:
File	CDInput	CD InputReceiveInputFile	Out to Flow Order
Construction	FlowOrder	Flow Order	First to LoggingSystem Second to ValidateAndRouteToCoreBankingSystem
WebSphere MQ	MQOutput	LoggingSystem	None - message is put on the LOGQ queue
Transformation	Compute	ValidateAndRouteToCoreBankingSystem	Out to In terminal of TCP/IP Client OutputToCoreBankingSystem
TCP/IP	TCPIPClientOutput	TCP/IP Client OutputToCoreBankingSystem	Out to TCP/IP Client ReceiveFromCoreBankingSystem
TCP/IP	TCPIPClientReceive	TCP/IP Client ReceiveFromCoreBankingSystem	Out to In terminal of SpecifyReplyFileNameUsingTransactionId
Transformation	Compute	SpecifyReplyFileNameUsingTransactionId	Out to In terminal of CD OutputSendReplyFile
File	CDOutput	CD OutputSendReplyFile	None - end of flow
WebSphere MQ	MQInput	LoggingSystemInput	Out to Database_Warehouse
Database	Database	Database_DataWarehouse	None - end of flow

The following steps show how to configure the message flow nodes:

1. Each transaction in the file BranchA.txt is parsed by the CDInput node CD InputReceiveInputFile using the end of line as a delimiter:
 - a. Select the CD Input node and in the Properties view select the Basic tab and enter the following values:
 - Directory filter: The directory where the file transfers will take place
 \\<IPAddress of CENPROC node Server-C system>\TransactionFiles
 - Filter name filter:
 *

An asterisk acts as a wild card and means that the CDInput node will pick up any input file. If the filter was set to *.txt, the node picks up only the files with extension .txt.

 - Configurable service: CDConfigurableService
 This is the service created in 9.6.1, "Creating a configurable service on WebSphere Message Broker" on page 456

Figure 9-17 shows the Filters and Configurable service specified for this node.

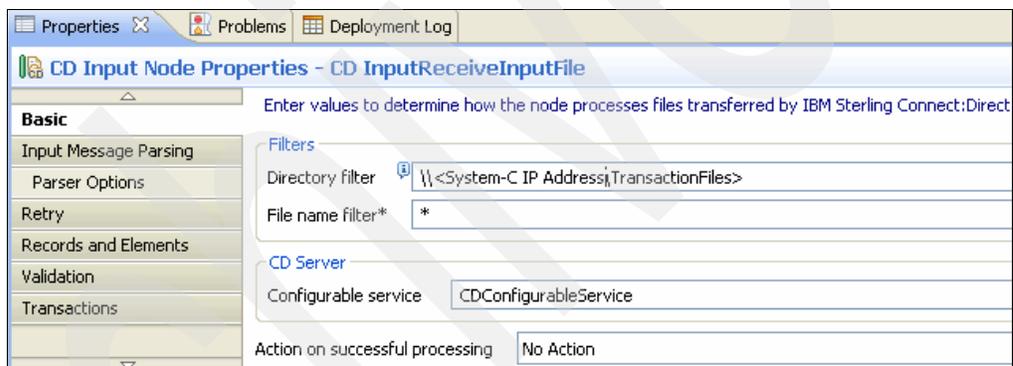


Figure 9-17 CD Input Node (1)

- b. Select the Input Message Parsing tab, and select the XMLNSC message domain, as shown in Figure 9-18.

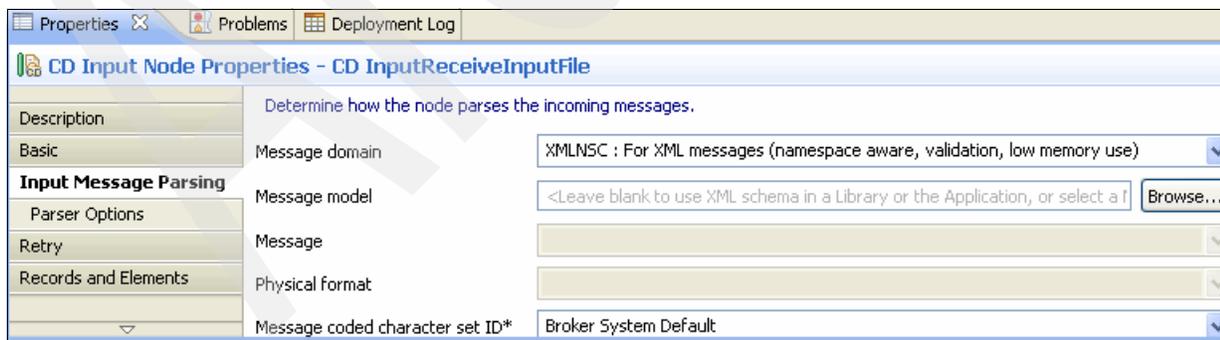


Figure 9-18 CD Input Node (2)

- c. Set the Records and Elements delimiters, as shown in Figure 9-19 on page 461:
 - Select **Delimited** for record detection.
 - Select **DOS** or **UNIX Line End** as the delimiter.

- Select **Postfix** as the delimiter type.

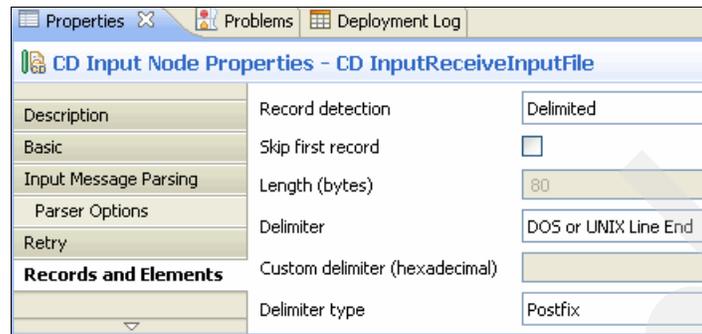


Figure 9-19 CD Input Node (3)

2. Each transaction is sent to the Flow Order node and first routed to the LogQ using the LoggingSystem MQ Output node (Figure 9-20). The transaction is then picked by another flow to insert it into the Data warehouse using the Database node Database DateWarehouse.

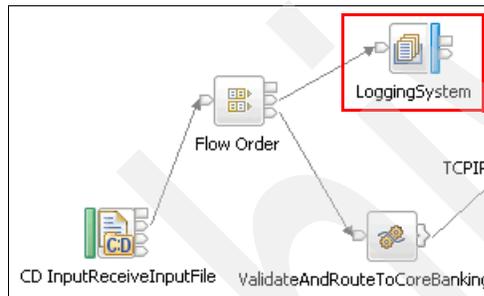


Figure 9-20

3. Select the LoggingSystem MQ Output node. In the Basic tab of the Properties, enter LOGQ as the queue manager name (Figure 9-21).

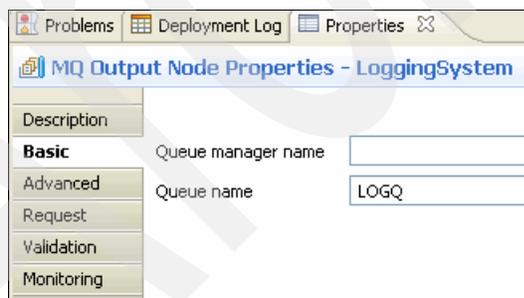


Figure 9-21 MQ Output properties

4. Messages put on the LOGQ are then picked up by the LoggingSystem MQ Input node and sent to the Database_DataWarehouse node (Figure 9-22 on page 462). Select the node, and on the Basics tab enter LOGQ as the queue name.

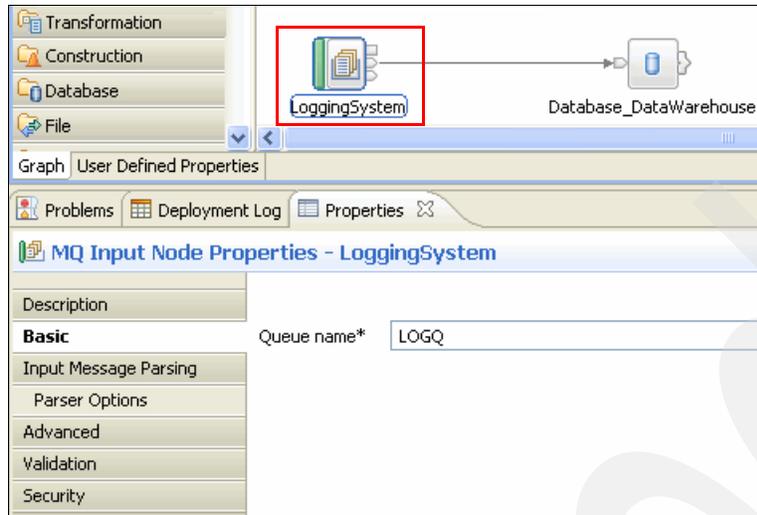


Figure 9-22 Database_DataWarehouse flow

5. Select the **Database_DataWarehouse** node.
 - a. In the Basic tab, enter ITSODB as the data source name, as shown in Figure 9-23.

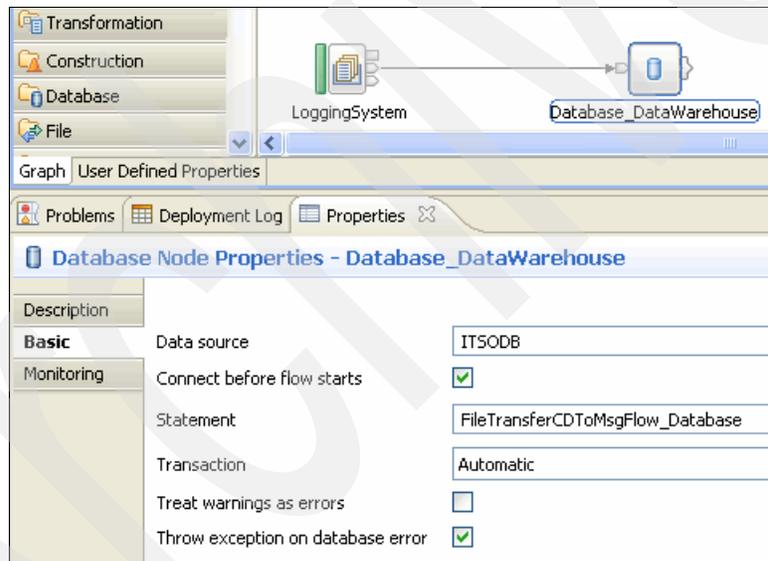


Figure 9-23 Setting the data source

- b. Double-click the Database_DataWarehouse, and enter the ESQL code shown in Example 9-2. This ESQL code inserts the transaction record into database.

Example 9-2 ESQL code

```
CREATE DATABASE MODULE FileTransferCDToMsgFlow_Database
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN

    DECLARE TRANS_ID INT Body.Customer.TransactionID;
    DECLARE ACCOUNT INT Body.Customer.AccountNo;
    DECLARE TO_ACCOUNT INT Body.Customer.ToAccountNo;
    DECLARE NAME CHAR Body.Customer.Name;
```

```

DECLARE TO_IFSC INT Body.Customer.IFSCCode;
DECLARE TO_BRANCH CHAR Body.Customer.Branch;
DECLARE AMOUNT INT Body.Customer.Amount;
DECLARE DATE1 DATE Body.Customer.Date;

INSERT INTO Database.TRANSLOG( TRANS_ID, NAME, ACCOUNT_NO,
TO_ACCOUNT_NO,TO_BRANCH_NAME, TO_IFSC_CODE, AMOUNT, DATE)
VALUES (TRANS_ID, NAME, ACCOUNT, TO_ACCOUNT, TO_BRANCH, TO_IFSC,AMOUNT,
DATE1 );
RETURN TRUE;
END;

END MODULE;

```

6. Each transaction is validated when the account number is validated, which should be 10-digits. The validation scans are done at the compute node `ValidateAndRouteToCoreBankingSystem`.
 - a. Double-click the compute node, and enter the ESQL code that validates the account number shown in Example 9-3.

Example 9-3 ESQL code

```

CREATE COMPUTE MODULE
FileTransferCDToMsgFlow_ValidateAndRouteToCoreBankingSystem
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
-- CALL CopyMessageHeaders();

CALL CopyEntireMessage();
-- validate account number
if(InputRoot.XMLNSC.Customer.AccountNo > 999999999 OR
InputRoot.XMLNSC.Customer.AccountNo < 999999999)
THEN
THROW USER EXCEPTION MESSAGE 2440
VALUES ('The customer account number is not valid',
InputRoot.XMLNSC.Customer.AccountNo);
END IF;
if(InputRoot.XMLNSC.Customer.ToAccountNo > 999999999 OR
InputRoot.XMLNSC.Customer.ToAccountNo < 999999999)
THEN
THROW USER EXCEPTION MESSAGE 2440
VALUES ('The To account number is not valid',
InputRoot.XMLNSC.Customer.ToAccountNo);
END IF;

RETURN TRUE;
END;

```

- b. In the Properties view on the Basics tab, select **FileTransferCDToMsgFlow_ValidateAndRouteToCoreBankingSystem** as the ESQL module.
7. Next, the transaction is routed to the core banking system using the `TCPIPClientOutput` node. To test this scenario, a core banking simulation flow is developed and deployed on

the same broker system. More details about simulation flow is provided in the 9.6.4, “Creating the simulation flow” on page 469.

- a. Select the **TCP/IP Client OutputToCoreBankingSystem** node, and open the Basics tab in the Properties view.

If you are using the core banking simulation flow, set the Connection details* value as localhost:2225. Otherwise, set the IPAddress:Port number of the core banking system, as shown in Figure 2-35.

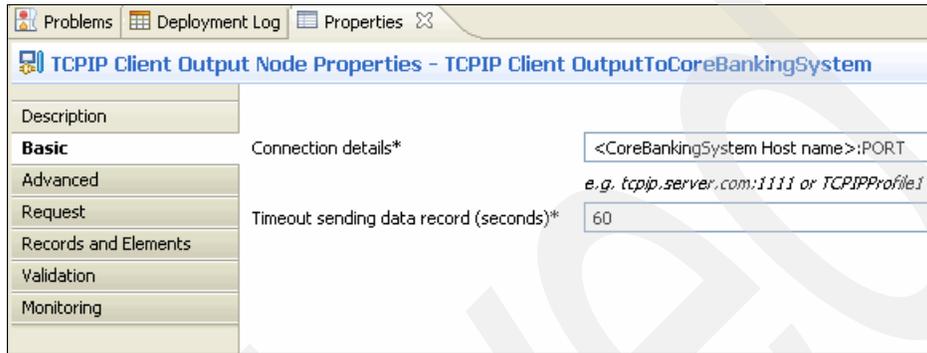


Figure 9-24 TCP/IP client output node properties (1)

- b. In the Advanced tab, set the Close connection option and Stream Control Options, as shown in Figure 9-25. These options are the default.

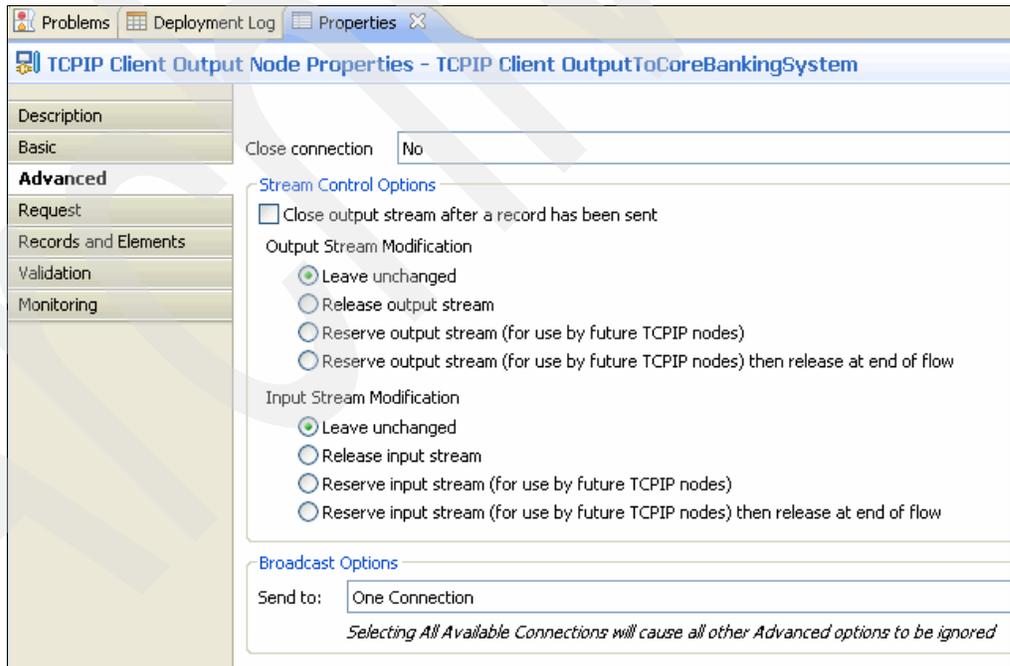


Figure 9-25 TCP/IP client output node properties (2)

- c. Set the Records and Elements delimiters, as shown in Figure 9-26 on page 465. Here, the record delimiter used is Broker System Line End.

Delimiters: The DOS and UNIX Line End delimiter type used on the CDInput node and the Broker System Line End used here are essentially the same delimiter type. The DOS and UNIX Line End delimiter is used with input nodes where the broker is receiving the message from an external transport that is setting the delimiter. The Broker System Line End is used when the broker sets the end of line character based on the operating system it is running on.

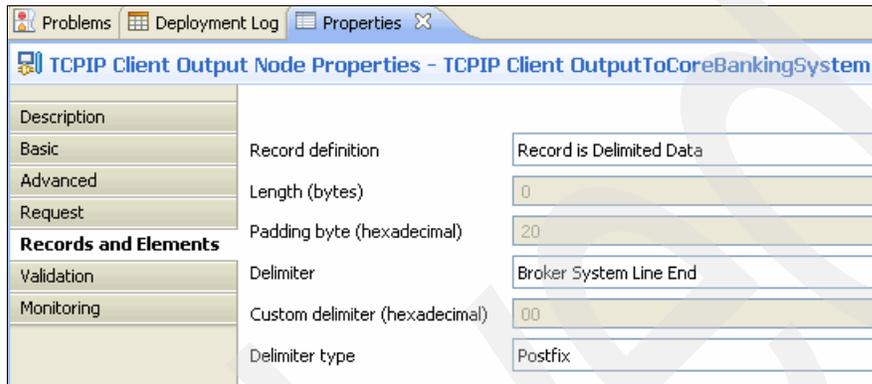


Figure 9-26 TCP/IP client output node properties (3)

8. When the core banking system simulation flow sends back a successful response within the timeout limit, the TCP/IP Client ReceiveFromCoreBankingSystem node receives it.
 - a. Set the basic properties in the TCP/IP Client ReceiveFromCoreBankingSystem, as shown in Figure 9-27. Replace <CoreBankingSystem Host Name>:PORT with the host name and port number of the core banking system. In this case, the simulation flow at localhost:2225.

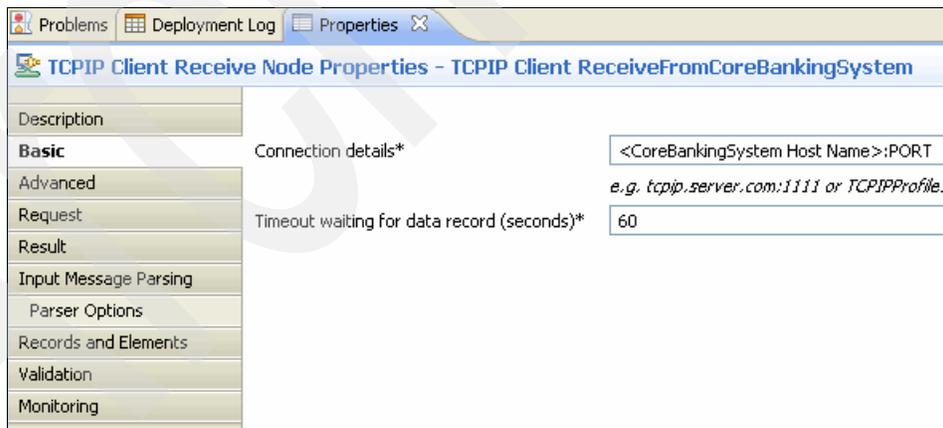


Figure 9-27 TCP/IP client receive properties (1)

- b. Figure 9-28 on page 466 shows the Advanced tab settings for the Close Connection and Stream Control options. Set the values as shown.

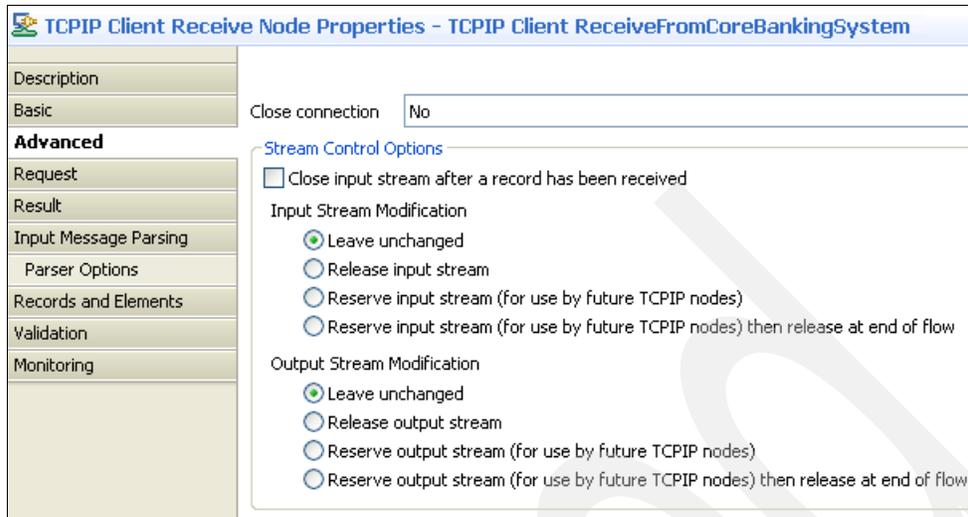


Figure 9-28 TCP/IP client receive properties (2)

Figure 9-29 shows Input Message Parsing options. The Message domain is set to XMLNSC to parse the XML message sent by the TCPIPServerOutput node.

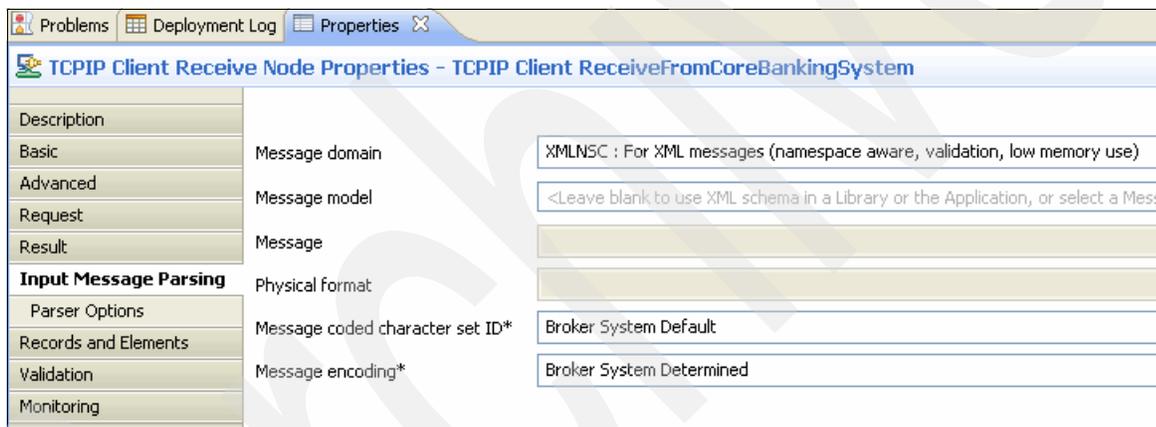


Figure 9-29 TCP/IP client receive properties (3)

- c. Figure 9-30 on page 467 shows the Records and Elements parsing options. Each TCP/IP stream record is delimited by the DOS or UNIX line end. Make sure the delimiter is the end of line type across TCPIPClient Output Node, TCPIPServerInput node, TCPIPServerOutput node, and TCPIPClientReceive node.

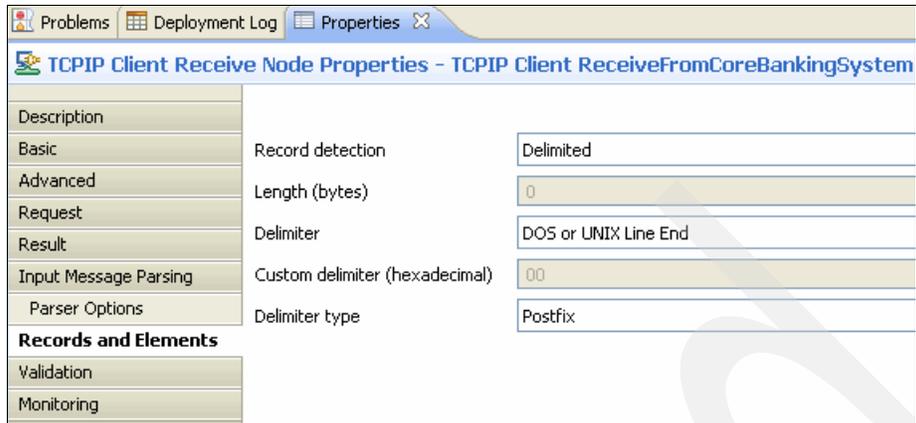


Figure 9-30 TCP/IP client receive properties (4)

9. After the TCP/IPClientReceive node receives the response, a transaction report file name is generated by the Compute node SpecifyReplyFileNameUsingTransactionId. The report file name is created using the unique TransactionId of each message with the file name as TransactionId.txt.

The ESQL code shown in Example 9-4 helps to generate the file name by populating the necessary OutputLocalEnvironment for the CDOOutput node that comes next in the flow. This ESQL code helps to override the file name set in the CDOOutput node File Name property.

- a. Double-click the Compute node, and replace the generated code with the code in Example 9-4.

Example 9-4 ESQL code

```

CREATE COMPUTE MODULE
FileTransferCDToMsgFlow_GenerateReplyFileNameUsingTransactionId
  CREATE FUNCTION Main() RETURNS BOOLEAN
  BEGIN
    -- CALL CopyMessageHeaders();
    --CALL CopyEntireMessage();
    SET OutputLocalEnvironment.Destination.CD.Name =
InputRoot.XMLNSC.Customer.TransactionID||'.txt';
    RETURN TRUE;
  END;

```

- b. Close the ESQL code.
- c. In the Basic tab of the Properties view, set the Compute node field to LocalEnvironment.
- d. Set the ESQL module field to the new module name FileTransferCDToMsgFlow_GenerateReplyFileNameUsingTransactionId.

See Figure 9-31 on page 468.

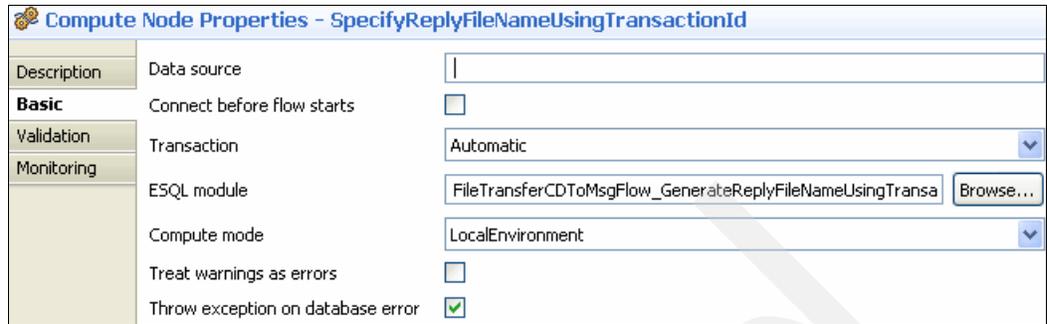


Figure 9-31 Setting the compute mode

10. The TransactionId.txt file is generated at the location C:\TransctionReply folder on the broker system (System-D) using the CDOOutput node CD OutputSendReplyFile. Set the following properties in the CDOOutput Node CD OutputSendReplyFile:
- In the Basics tab, replace SNODE with the host name of the Server-D. The Process Name is Proc1, which the CDOOutput node will use to copy the message into a file using CD Server. See Figure 9-32.

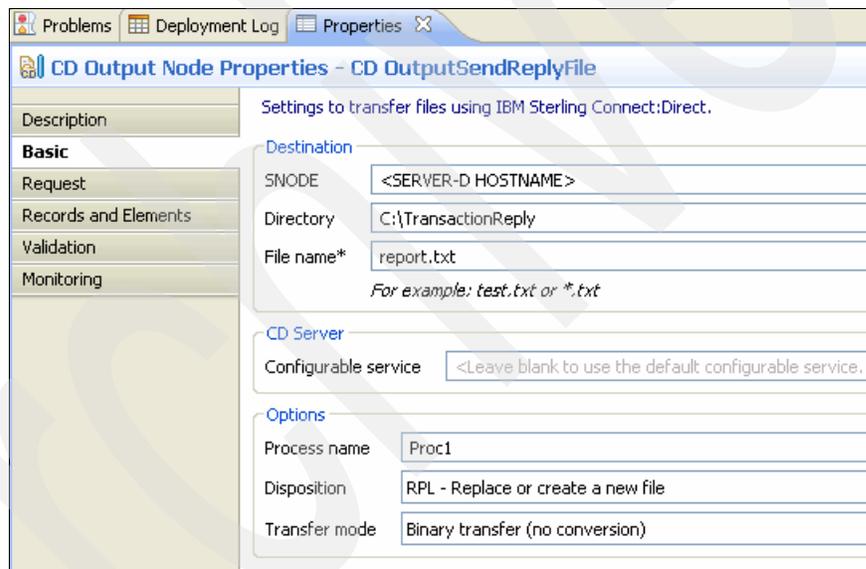


Figure 9-32 CDOOutput node properties (1)

- Set the values in Records and Elements, as shown in Figure 9-33.

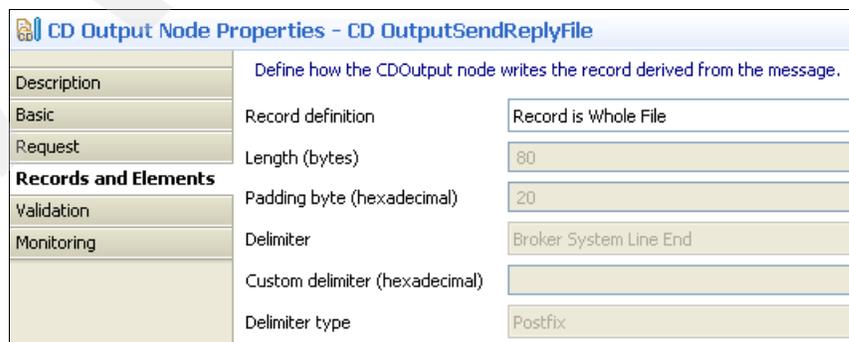


Figure 9-33 CDOOutput node properties (2)

9.6.4 Creating the simulation flow

A core banking simulation flow, `CoreBankingServerSimulation.msgflow`, is developed to test this scenario, Connect `TCIPServerInput` node to `TCIPServerOutput` Node. See Figure 9-34.

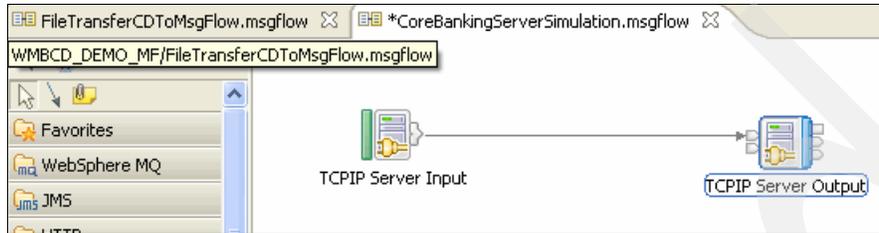


Figure 9-34 Core banking simulation flow

To create the simulation flow:

1. Create a new message flow called `CoreBankingServerSimulation` in the same application.
2. Drop a `TCIPServerInput` node and `TCIPServerOutput` node to the canvas.
3. Set the properties in `TCIPServer Input`:
 - a. Figure 9-35 shows the basic properties. In the `Connection details` field, specify the port to use as the TCP/IP server. In this example, 2225. Leave the `Timeout waiting for data record` field as 60 seconds (its default value).

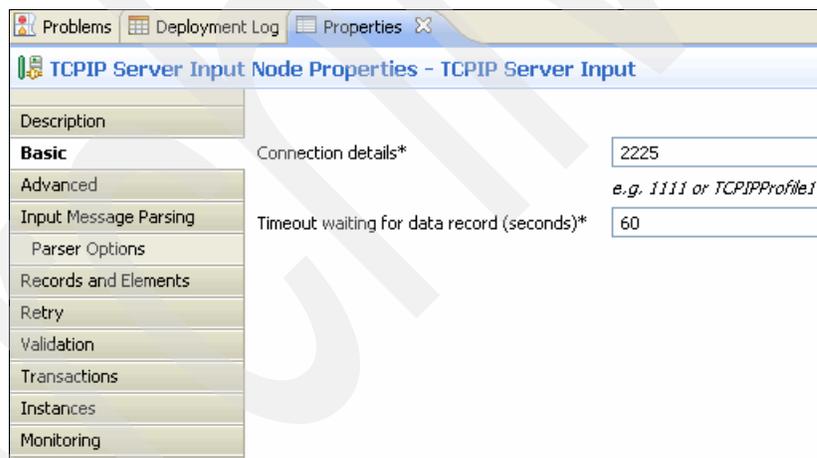


Figure 9-35 TCP/IP server input properties (1)

Figure 9-36 on page 470 shows the values in the `Advanced` tab for the `Close Connection` and `Stream control` options.

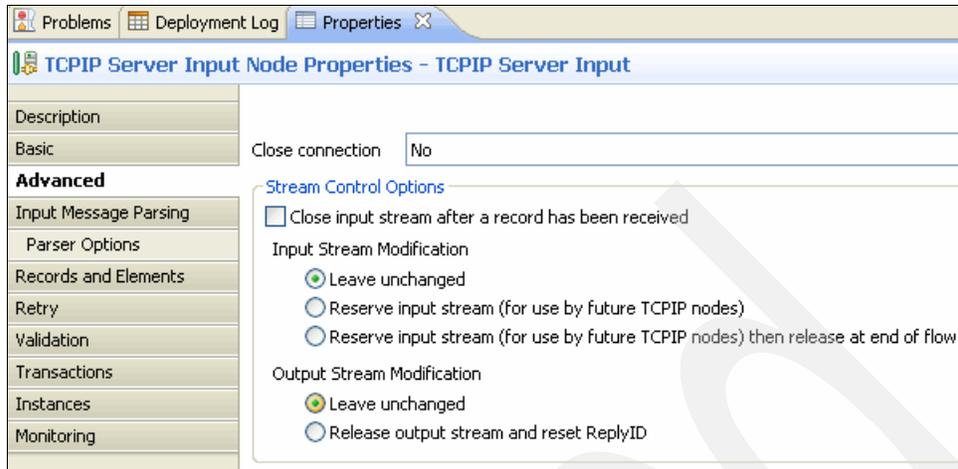


Figure 9-36 TCP/IP server input properties (2)

- b. Figure 9-37 shows the Input Message Parsing options. Set the message domain to XMLNSC because the TCP/IP Server gets messages in XML format.

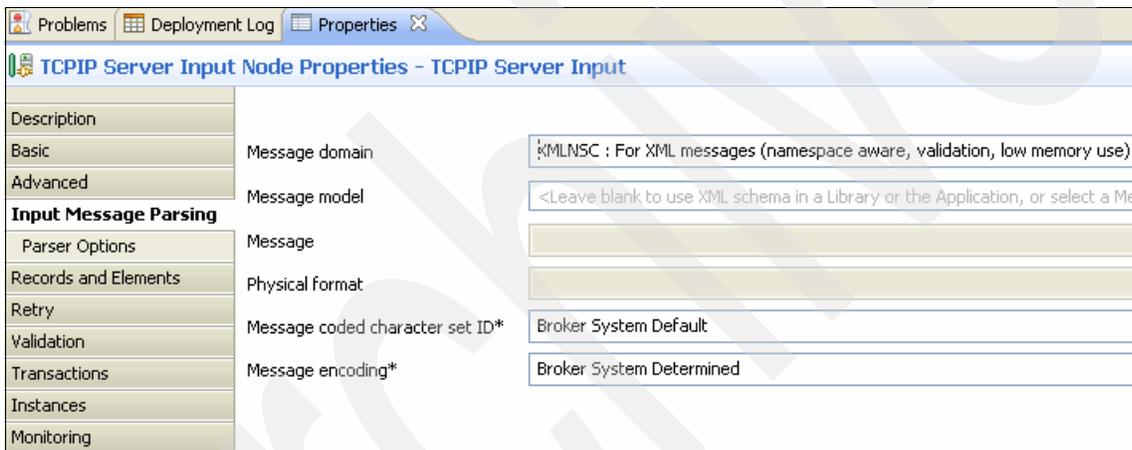


Figure 9-37 TCP/IP server input properties (3)

- c. Figure 9-38 on page 471 shows the Records and Elements properties. The end of record delimiter is the DOS or UNIX Line End. This delimiter type (end of line) matches the TCP/IPClientOutput node's delimiter (end of line), as shown in Figure Figure 9-26 on page 465.

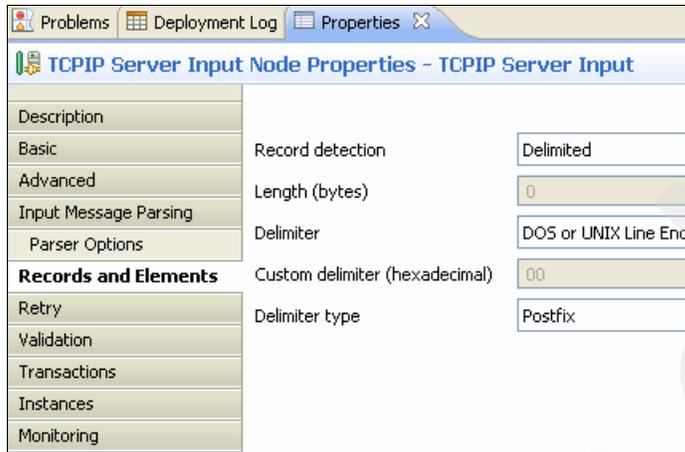


Figure 9-38 TCP/IP server input properties (4)

4. Set the properties for the TCP/IPServerOutput node:
 - a. Figure 9-39 shows the basic properties. Set the Connection details field to localhost:2225, and the Timeout sending the data record field to 60 seconds (the default value).

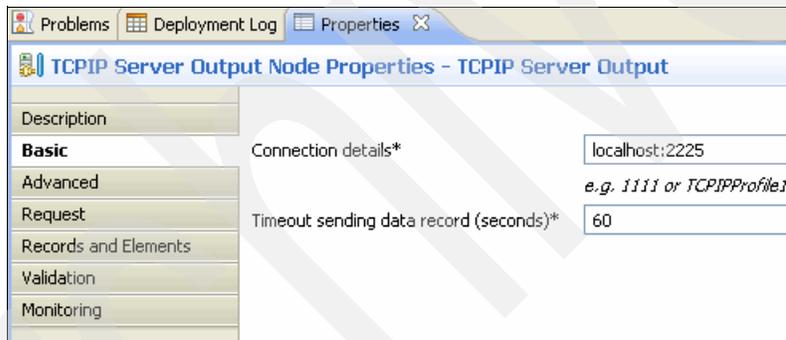


Figure 9-39 TCP/IP server output properties (1)

- b. Figure 9-40 on page 472 shows the Close Connection option and Stream Control Options. Set the values as shown in the figure.

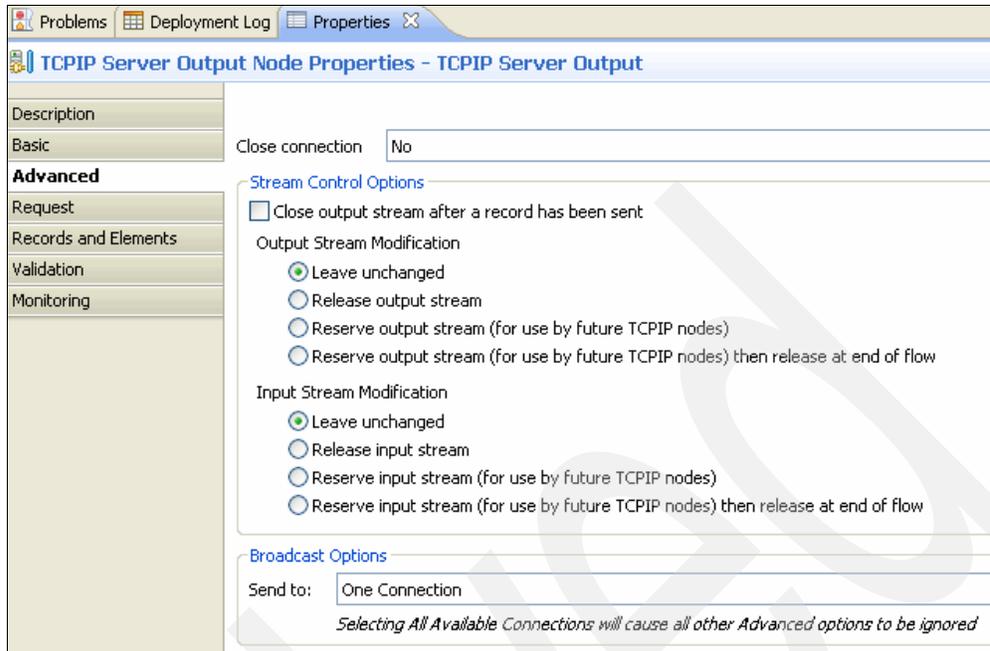


Figure 9-40 TCP/IP server output properties (2)

Figure 9-41 shows Records and Elements delimiter options. The TCPIPServerOutput node sets the specified delimiter at the end of the message before sending the reply. The delimiter used is Broker System Line End.

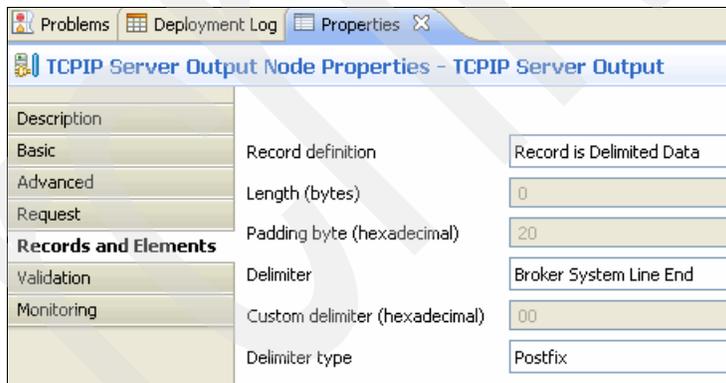


Figure 9-41 TCP/IP server output properties (3)

9.6.5 Deploying the message flows and preparing the environment

To test the message flow:

1. Deploy the FileTransferCDToMsgFlow and the CoreBankingServerSimulation flows into the message broker execution group. See Figure 9-42 on page 473.

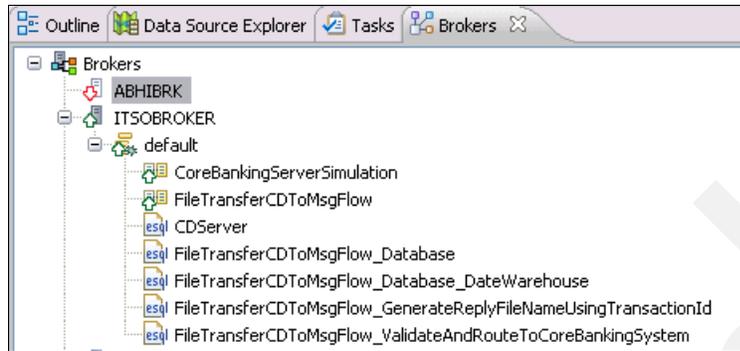


Figure 9-42 Deploying flows into the message broker execution group

2. Map the C:\EnterpriseDataCenter of System-C to the broker system (System-D). On broker System-D, you can see the TransactionFiles folder of System-C mapped.

9.6.6 Sample file transfer data

Example 9-5 shows sample data for the file (BranchA.txt) to be transferred. Each transaction record is delimited by End of line.

Example 9-5 BranchA.txt

```
<Customer><TransactionID>9129011</TransactionID><AccountNo>3882919201</AccountNo><ToAccountNo>2810380987</ToAccountNo><Name>John</Name><IFSCCode>1239</IFSCCode><Branch>Bangalore</Branch><Amount>10000</Amount><Date>2012-03-22</Date></Customer>
<Customer><TransactionID>9129012</TransactionID><AccountNo>1238289128</AccountNo><ToAccountNo>2452991991</ToAccountNo><Name>James</Name><IFSCCode>1234</IFSCCode><Branch>Bangalore</Branch><Amount>20000</Amount><Date>2012-03-22</Date></Customer>
<Customer><TransactionID>9129013</TransactionID><AccountNo>1238283456</AccountNo><ToAccountNo>2452923456</ToAccountNo><Name>James</Name><IFSCCode>2230</IFSCCode><Branch>Bangalore</Branch><Amount>50000</Amount><Date>2012-03-22</Date></Customer>
```

9.6.7 Testing the message flow

After the flow is deployed on the broker, the scenario can be tested.

Transferring the file from BRANCH-A to CENPROC

Using the CD Requester tool on BRANCH-A, the transaction file C:\CDFileTransfer\BranchA.txt is transferred from BRANCH-A to CENPROC. The file will be placed in C:\EnterpriseDataCenter\TransactionFiles\BranchA.txt on Server-C u.

This section provides two methods to start the file transfer.

Method 1: Using a process file

This method uses a process file to initiate the file transfer of a file from BRANCH-A:

1. Create a file called BRANCH2CENPROC.cdp with the content shown in Example 9-6.

Example 9-6 BRANCH2CENPROC.cdp

```
/*BEGIN_REQUESTER_COMMENTS
  $PNODE$="BRANCH-A" $PNODE_OS$="Windows"
  $SNODE$="CENPROC" $SNODE_OS$="Windows"
```

```

$OPTIONS$="WDOS"
END_REQUESTER_COMMENTS*/

PROC PROCESS
  SNODE=CENPROC
  PNODEID=(administrator,password-of-childbranch-machine)

SNODEID=(username-of-Server-C-machine,password-of-Server-C-machine,password-of-
Server-C-machine)

CPY COPY
  FROM (
    FILE="C:\CDFileTransfer\BranchA.txt"
  )
  TO (
    FILE=C:\EnterpriseDataCenter\TransactionFiles\BranchA.txt
    DISP=NEW
  )

PEND

```

2. Double-click the file to open it in the CD Requester editor.
3. In the CDRequester tool, use the process file BRANCH2CENPROC.cdp to initiate the transfer.
4. Change the values of password-of-childbranch-machine, username-of-Server-C-machine, and password-of-Server-C-machine appropriately in the BRANCH2CENPROC.cdp file.

Method 2: Creating a process

Alternatively, you can create a process using the CD Requester tool.

1. Go to **Start** → **All Programs** → **Sterling Commerce™ Connect Direct v4.5.01** → **CD Requester**.
2. Select **File** → **New** → **Process** to create a new process, as shown in Figure 9-43.

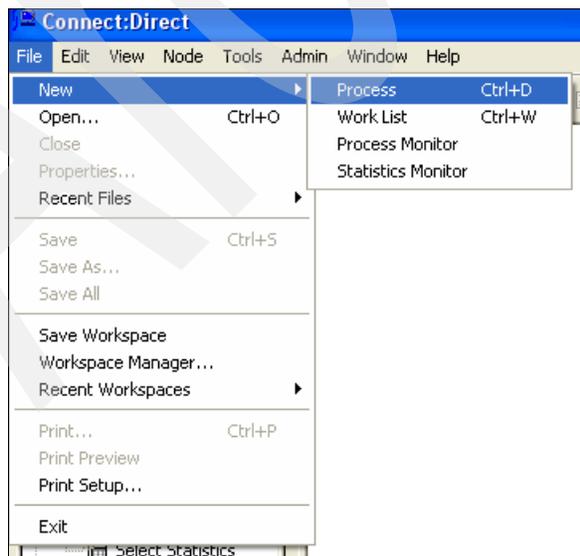


Figure 9-43 Create a new process

3. This will launch a new process window as shown in Figure 9-44. Select the Pnode and Snode name from the Name drop-downs.

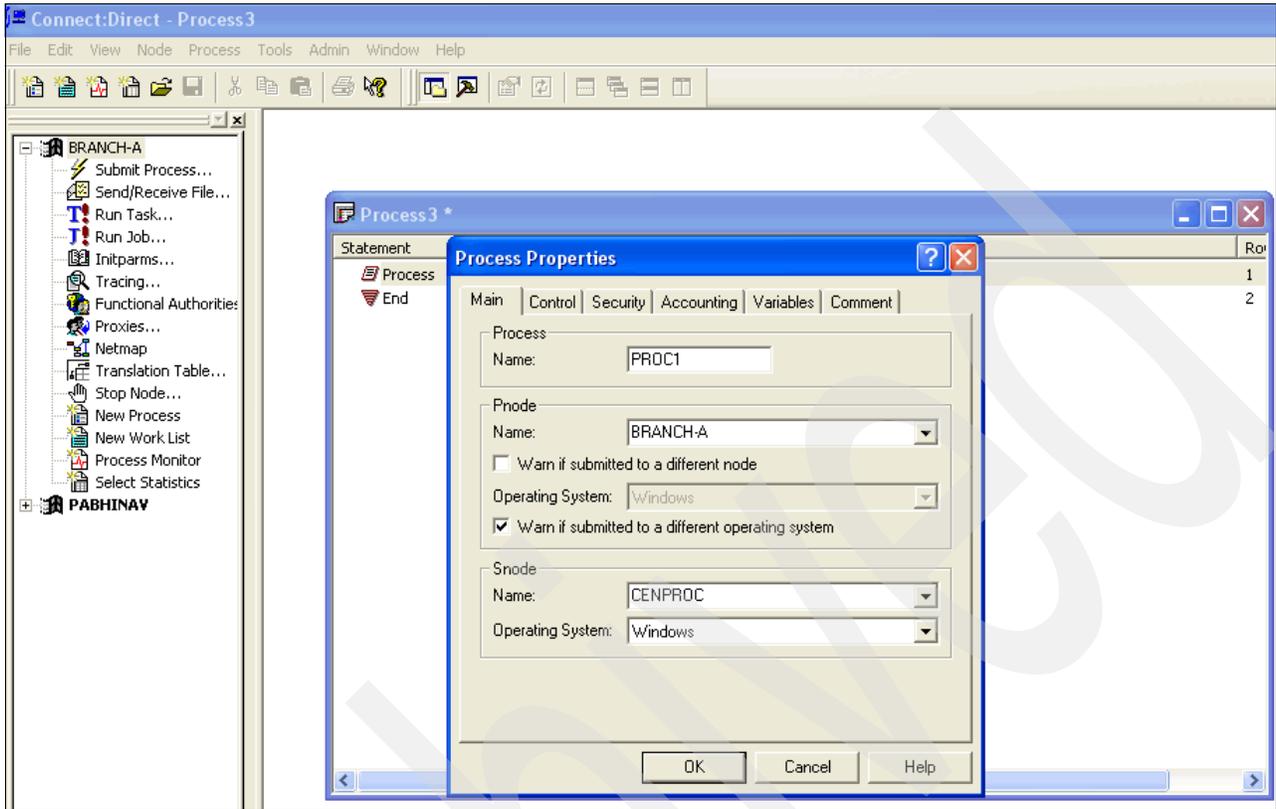


Figure 9-44 Set process properties

4. Set the security credentials to connect to the Pnode and Snode, as shown in Figure 9-45 on page 476.

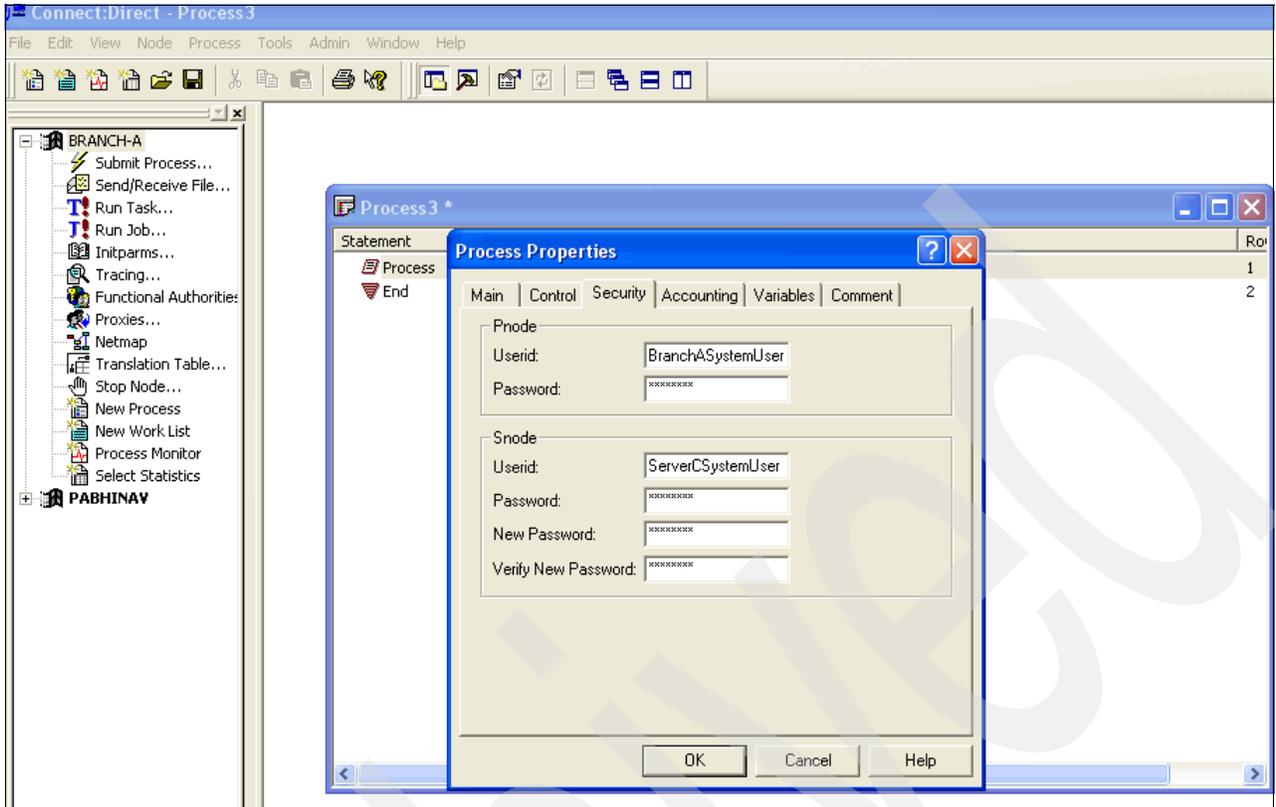


Figure 9-45 Set security credentials

5. Insert a copy statement to copy the file C:\CDFileTransfer\BranchA.txt from the Pnode on Branch-A To C:\EnterpriseDataCenter\TransactionFiles\BranchA.txt to the Snode CENPROC. See Figure 9-46 on page 477.

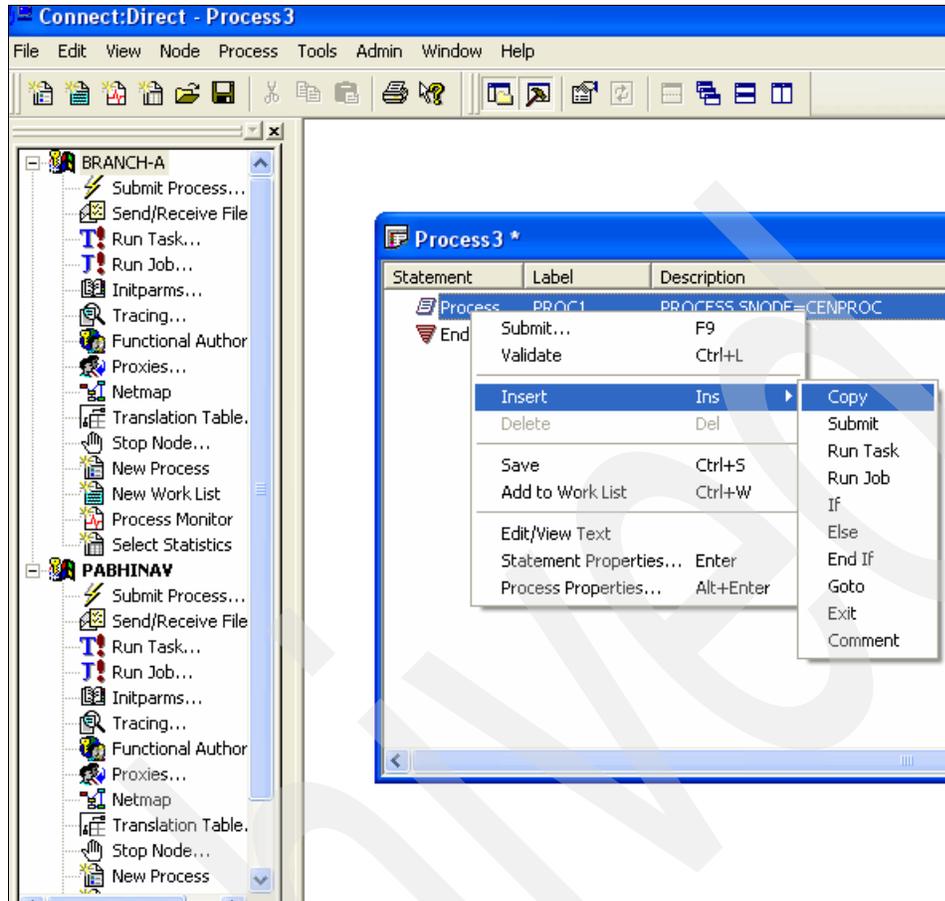


Figure 9-46 Set Copy statement

6. Insert the copy command information, as shown in Figure 9-47 on page 478.

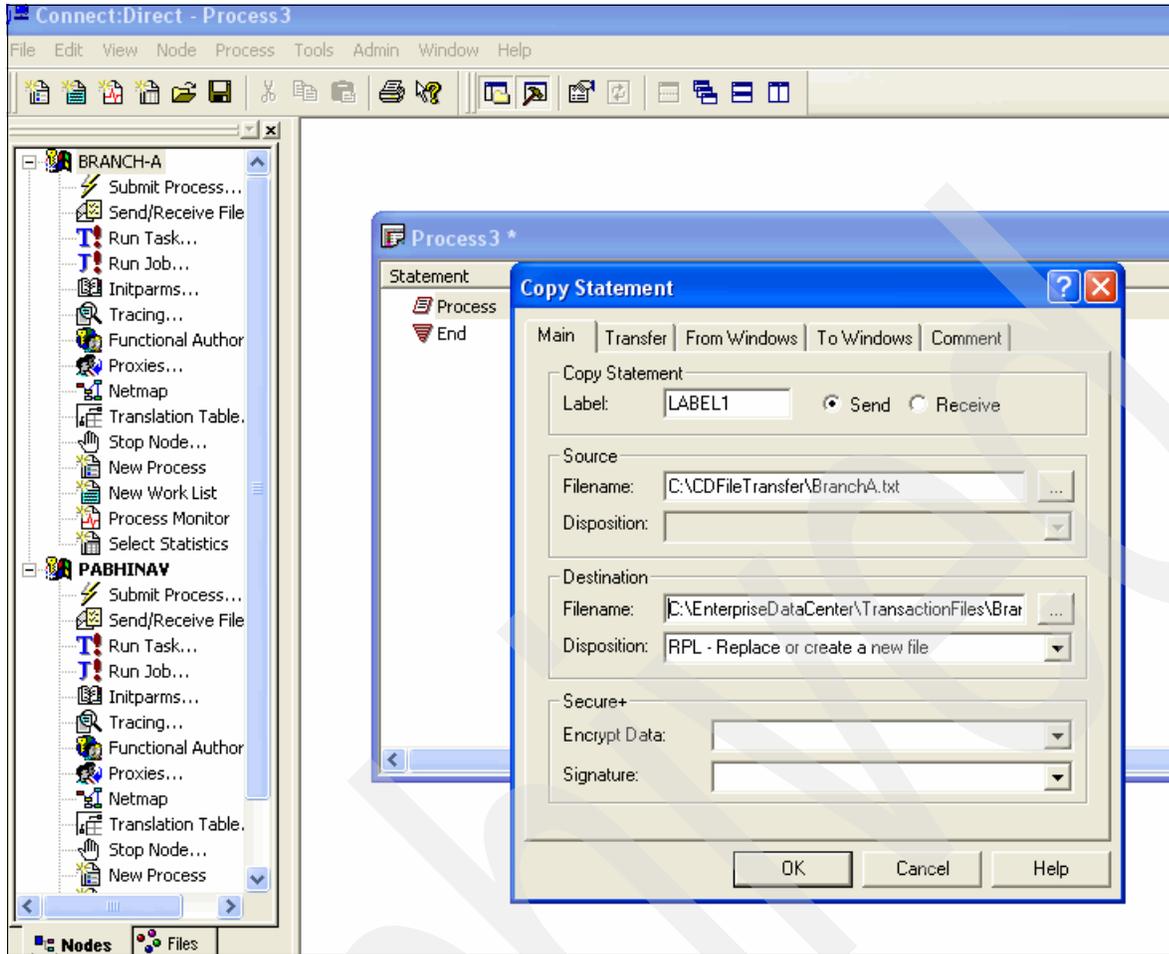


Figure 9-47 Set Copy statement parameters

7. Submit the process to copy the file from source to destination system folders, as shown in Figure 9-48 on page 479.

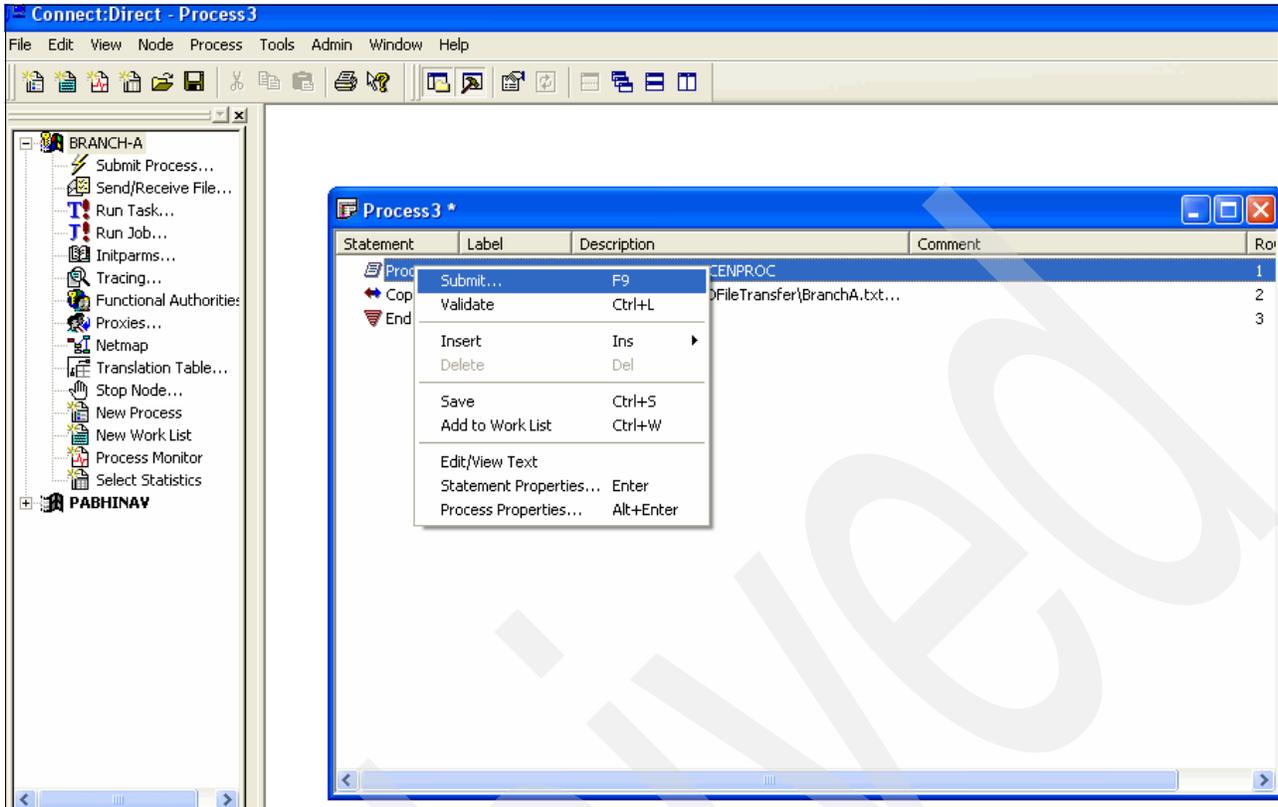


Figure 9-48 Submit a process

- Click **OK** to submit the process.

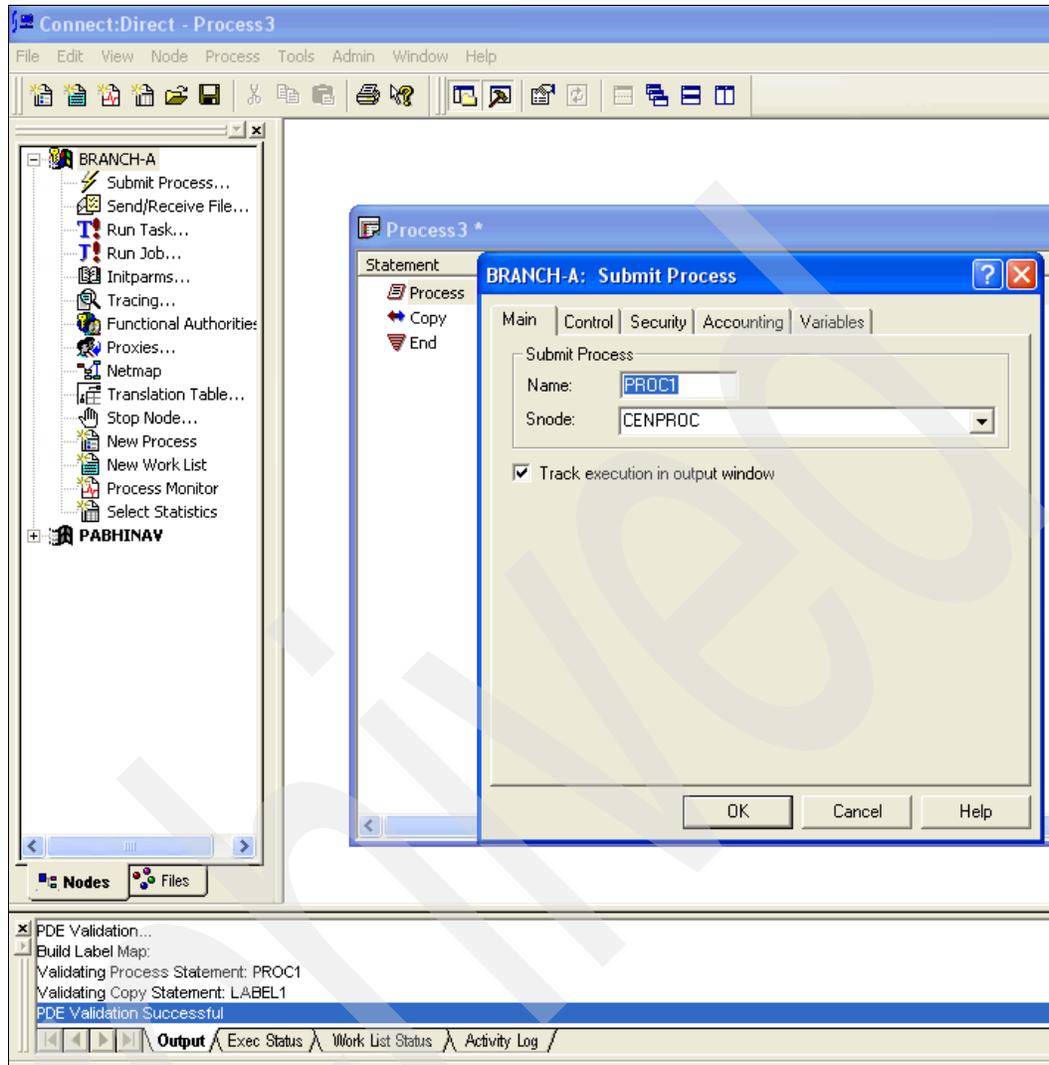


Figure 9-49 Submit a process verification

Verifying that the file was transferred to CENPROC

The following steps will help you determine if the file transfer was successful:

1. Check the statistics in the Process Execution Statistics window to verify whether the copy operation was successful:
 - a. Select the submit process number executed after submitting the process Proc1.
 - b. Right-click **View Details**, as shown in Figure 9-50 on page 481, to open the Process Execution Statistics window.

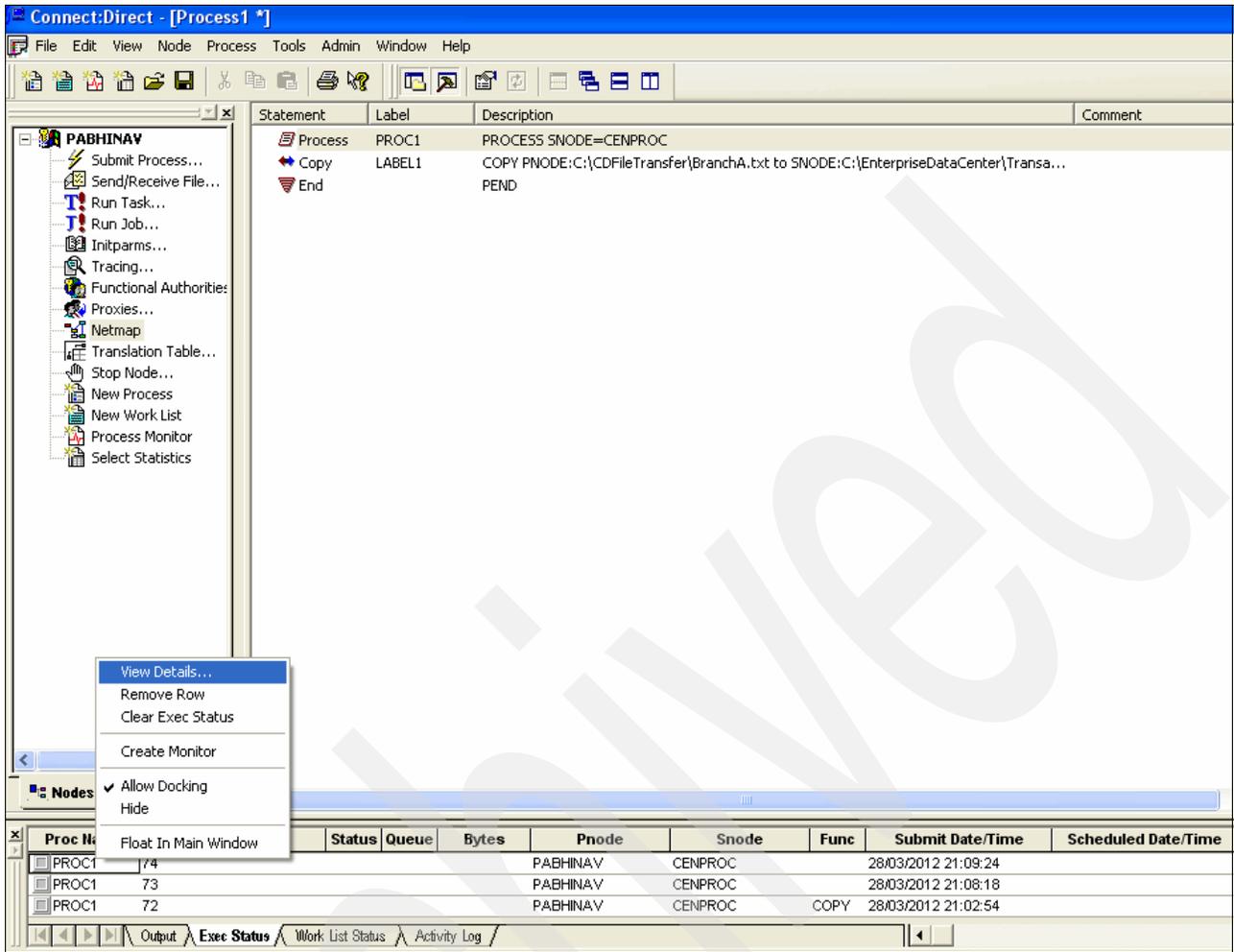


Figure 9-50 View Details

On successful copy, the following statement will be logged, as shown in Figure 9-51 on page 482:

Copy operation successful

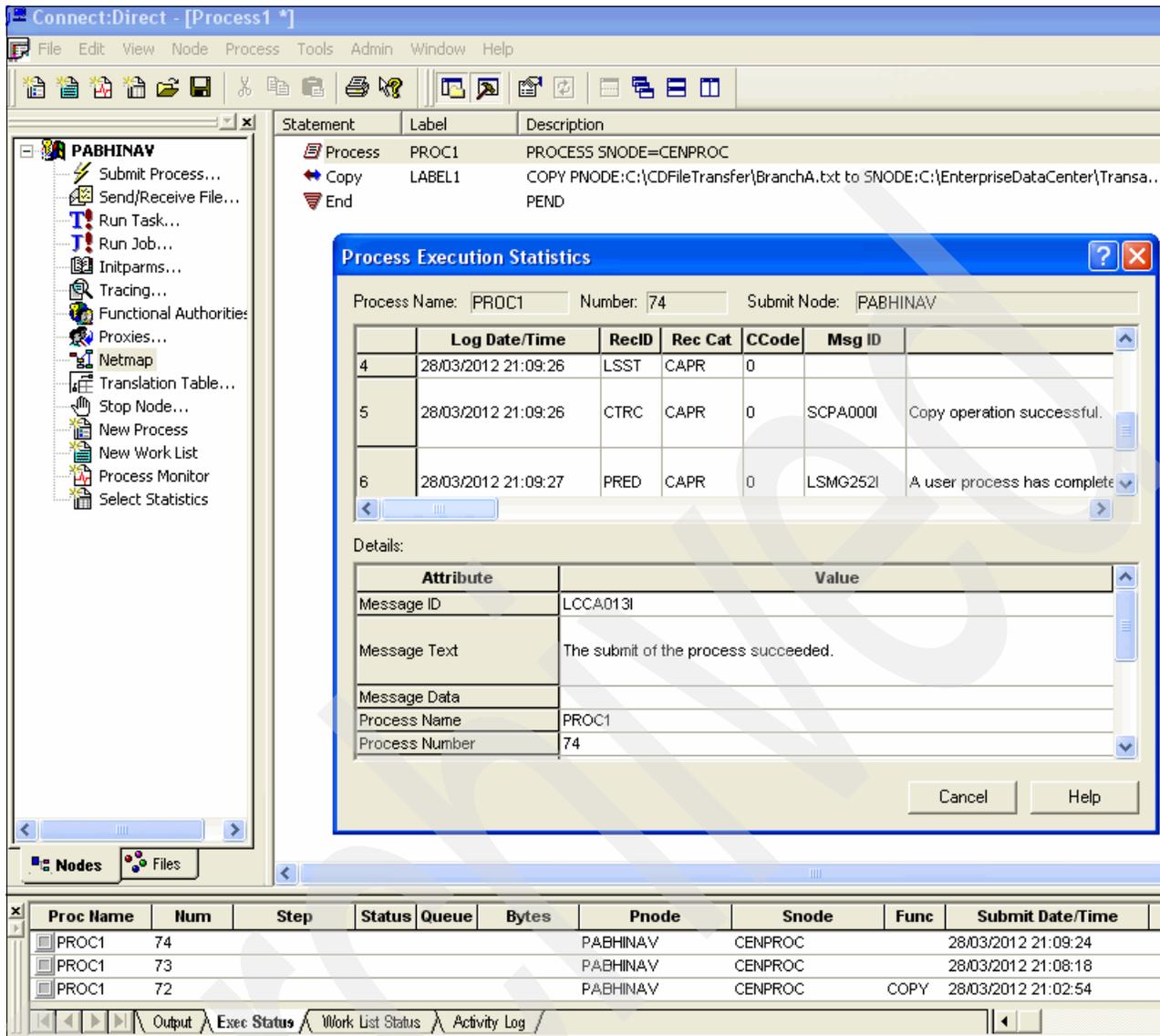


Figure 9-51 Process Execution Statistics window

Verifying that the message flow processed the file

After the file is successfully transferred by the CD node from Branch-A system to Server-C system, the message broker flow FileTransferCDToMsgFlow on Server-D picks up the file and processes each transaction record one by one. It updates the database and generates the report files for successful response from core banking system:

1. Use the DB2 utilities to verify that the Transaction records are stored in the data warehouse table TRANSLOG, as shown in Figure 9-52 on page 483.

IBM-60E4C41D5F8 - DB2 - ITSODB - DB2ADMIN.TRANSLOG

Edits to these results are performed as positioned UPDATES and DELETES. Use the Tools Settings notebook to change the form of editing.

NAME	ACCOUNT_NO	TO_ACCOUNT_NO	TO_BRANCH_NAME
John	3882919201	2810380987	Bangalore
James	1238289128	2452991991	Bangalore
James	1238283456	2452923456	Bangalore

Figure 9-52 Storing transaction records

- Verify that the transaction processing report for each transaction in the file was generated in the output folder on the broker system. See Figure 9-53.

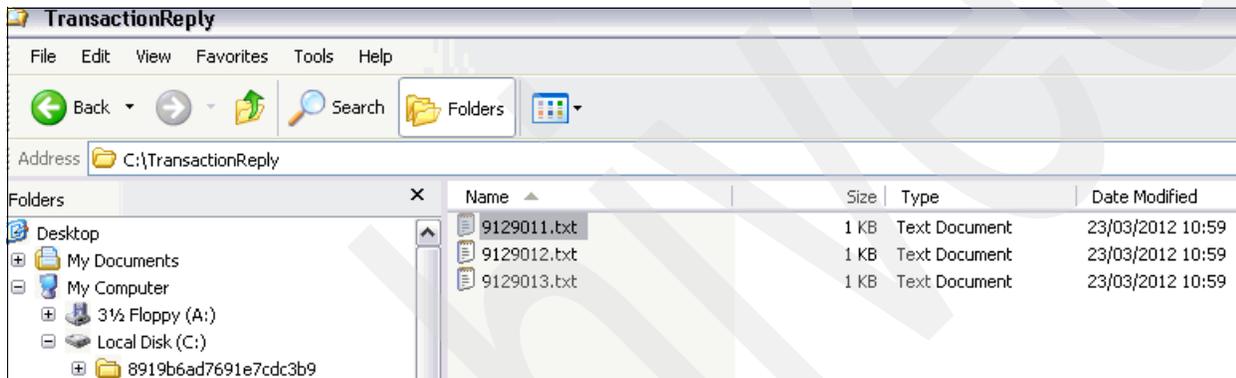


Figure 9-53 Transaction reports

Archived

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG248020>

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select **Additional materials**, and open the directory that corresponds with the IBM Redbooks form number, SG248020.

Using the Web material

The web material is useful as a reference as you follow along with the instructions in this book. The files included are not intended to be ready-to-use, but are simply artifacts from the projects.

The additional Web material contained in the archive file provides the following folders:

- ▶ **MSMQ folder:** Contains the WebSphere Message Broker project interchange file and the .NET class code for the scenario in Chapter 4, “Scenario: Bridging WebSphere MQ and Microsoft Message Queuing” on page 67.
- ▶ **CRM folder:** Contains the WebSphere Message Broker project interchange file and the .NET class code for the scenario in Chapter 5, “Scenario: Calling Microsoft Dynamics CRM from a message flow” on page 109.

- ▶ WCFService folder: Contains the .NET class code for the scenario in Chapter 6, “Scenario: Integration Windows Communication Foundation in message flows - Part 1” on page 163.
- ▶ WCFClient folder: Contains the WebSphere Message Broker project interchange file and .NET class code for the scenario in Chapter 7, “Scenario: Integrating Windows Communication Foundation in message flows - Part 2” on page 273
- ▶ FTE folder: Contains the WebSphere Message Broker project interchange file and the .NET class code for the scenario in Chapter 8, “Integrating file transfer with WebSphere MQ FTE into the message flow” on page 391.
- ▶ CD folder: Contains the WebSphere Message Broker project interchange file for the scenario in Chapter 9, “Integrating file transfer using Sterling Connect:Direct with your message flow” on page 443.

Downloading and extracting the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the Web material .zip file into this folder.

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

The following IBM Redbooks publications provide additional information about the topics in this document. Note that some publications referenced in this list might be available in softcopy only:

- ▶ *Connecting Your Business Using IBM WebSphere Message Broker V7 as an ESB*, SG24-7826
- ▶ *Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB*, SG24-7360
- ▶ *WebSphere MQ V7.0 Features and Enhancements*, SG24-7583

You can search for, view, download, or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- ▶ IBM WebSphere Message Broker Information Center
<http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp>
- ▶ Express Edition message flow nodes
http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/topic/com.ibm.etools.mft.doc/an67720_.htm
- ▶ Using Microsoft .NET in WebSphere Message Broker V8 - DeveloperWorks tutorials
http://www.ibm.com/developerworks/views/websphere/libraryview.jsp?search_by=Using+Microsoft+.NET+in+WebSphere+Message+Broker+V8
- ▶ Business Integration - WebSphere MQ SupportPacs
<http://www-1.ibm.com/support/docview.wss?rs=849&uid=swg27007205>
- ▶ IP04: WebSphere Message Broker SupportPacs
http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24006518&loc=en_US&cs=utf-8&lang=en
- ▶ developerWorks articles about WebSphere Message Broker Patterns
<http://www.ibm.com/developerworks/wikis/display/esbpatterns/>
<http://www.ibm.com/developerworks/library/ws-enterpriseconnectivitypatterns/index.html>

http://www.ibm.com/developerworks/websphere/library/techarticles/0910_phillips/0910_phillips.html

- ▶ Programming Models for Microsoft Dynamics CRM
<http://msdn.microsoft.com/en-us/library/gg327971.aspx>
- ▶ Open XML SDK 2.0 for Microsoft Office Download Center
<http://www.microsoft.com/download/en/details.aspx?id=5124>

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services



Redbooks

Using WebSphere Message Broker V8 in Mid-Market Environments

(1.0" spine)

0.875" x 1.498"

460 <-> 788 pages



Using WebSphere Message Broker V8 in Mid-Market Environments



**Develop and access
Windows
Communication
Foundation services**

**Integrate .NET
applications**

**Integrate managed
file transfer**

IBM WebSphere Message Broker is a lightweight, advanced enterprise service bus (ESB) that provides a broad range of integration capabilities that enable companies to rapidly integrate internal applications and connect to partner applications. Messages from business applications can be transformed, augmented and routed to other business applications. The types and complexity of the integration required will vary by company, application types, and a number of other factors.

Processing logic in WebSphere Message Broker is implemented using message flows. Through message flows, messages from business applications can be transformed, augmented, 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 perform tasks that are associated with message routing, transformation, and enrichment. Message flows are created and tested using the Message Broker Toolkit, a sophisticated, easy-to-use programming tool that provides a full range of programming aids.

This IBM Redbooks publication focuses on two specific integration requirements that apply to many midmarket companies. The first is the ability to use WebSphere Message Broker to integrate Microsoft.NET applications into a broader connectivity solution.

The second is the ability to integrate WebSphere Message Broker with file transfer networks, specifically with WebSphere MQ File Transfer Edition and IBM Sterling Connect Direct.

**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**

SG24-8020-00

ISBN 073843700X