IBM

Enabling SOA Using WebSphere Messaging

Service-oriented architecture and messaging

ESB implementation with WebSphere Message Broker

ESB implementation with WebSphere ESB

Carla Sadtler
Philipp Huber
SangMin Yi

Redbooks

International Technical Support Organization

Enabling SOA Using WebSphere Messaging

March 2006

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

First Edition (March 2006)

This edition applies to WebSphere Message Broker Version Version 6.0, WebSphere Application Server Version 6.0.2, WebSphere Enterprise Service Bus V6.0.1, and WebSphere MQ V6.

Contents

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates 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. 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 IBM's application programming interfaces.

vii

Trademarks

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

@server®	Everyplace®	RETAIN®
@server®	HACMP™	SupportPac™
Redbooks (logo) ™	IBM®	Tivoli®
developerWorks®	IMS™	WebSphere®
z/OS®	Parallel Sysplex®	Workplace™
CICS®	Rational®	
DB2®	Redbooks™	

The following terms are trademarks of other companies:

EJB, Java, Java Naming and Directory Interface, JDBC, JSP, J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

Successfully implementing an SOA requires applications and infrastructure that can support the SOA principles. Applications can be enabled by creating service interfaces to existing or new functions hosted by the applications. The service interfaces should be accessed using an infrastructure that can route and transport service requests to the correct service provider. As organizations expose more and more functions as services, it is vitally important that this infrastructure should support the management of SOA on an enterprise scale.

This IBM® Redbook looks at how the IBM messaging products support an SOA environment. In particular, it will look at WebSphere® Application Server, WebSphere Enterprise Service Bus, WebSphere MQ, and WebSphere Message Broker in an SOA environment. It discusses how they support SOA, compare the potential ESB product implementations, and show examples of building the infrastructure and creating mediations.

The team that wrote this redbook

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

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

Philipp Huber is an Advisory IT Specialist at IBM Switzerland. He has been with IBM for six years, working primarily on design and development of J2EE™-based e-business applications. Philipp holds a degree in Technical Computer Science from the University of Applied Science Aargau, Switzerland. His areas of expertise include object-oriented analysis and design, messaging and J2EE application design and development using Rational® Application Developer, WebSphere Application Server, and WebSphere MQ.

SangMin Yi is a an Advisory IT Specialist of WebSphere Business Integration solutions. He joined IBM seven years ago and works in the IBM Software Group. He is a specialist for IBM Web services technologies and service-oriented architecture and lectures on these topics in a major company in Korea. Before joining IBM, he worked as a producer in a broadcasting system.

ix

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or clients.

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

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

      ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

► Use the online **Contact us** review redbook form found at:

      ibm.com/redbooks

► Send your comments in an email to:

      redbook@us.ibm.com

- ► Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HZ8 Building 662
  P.O. Box 12195
  Research Triangle Park, NC 27709-2195

1

Introduction

This chapter provides an introduction to service-oriented architecture (SOA), the enterprise service bus (ESB), and the IBM SOA strategy. The purpose of this chapter is to solidify the concept of SOA as it pertains to messaging.

1

1.1 SOA overview

SOA defines integration architectures based on the concept of a *service*. Applications collaborate by invoking each others' services, and services are composed into larger sequences to implement business processes.

1.1.1 The driver for SOA

The main driver for SOA is to define an architectural approach that assists in the flexible integration of IT systems. Organizations spend a considerable amount of time and money trying to achieve rapid, flexible integration of IT systems across all elements of the business cycle. The drivers behind this objective include:

► Increasing the speed at which businesses can implement new products and processes, can change existing ones, or can recombine them in new ways

► Reducing implementation and ownership costs of IT systems and the integration between them

► Enabling flexible pricing models by outsourcing more fine-grained elements of the business than were previously possible or by moving from fixed to variable pricing, based on transaction volumes

► Simplifying the integration work that is required by mergers and acquisitions

► Achieving better IT utilization and return on investment

► Achieving implementation of business processes at a level that is independent from the applications and platforms that are used to support the processes

SOA prescribes a set of design principles and an architectural approach to achieve this rapid, flexible integration.

1.1.2 Architectural approach

SOA is an integration architecture approach based on the concept of a service. The business and infrastructure functions that are required to build distributed systems are provided as services that collectively, or individually, deliver application functionality to end-user applications or other services.

SOA specifies that within any given architecture, there should be a consistent mechanism for services to communicate. That mechanism should be loosely coupled and support the use of explicit interfaces.

SOA brings the benefits of loose coupling and encapsulation to integration at an enterprise level. It applies successful concepts proved by Object Oriented

development, Component Based Design, and Enterprise Application Integration technology to an architectural approach for IT system integration.

Services are the building blocks of SOA, providing interfaces to functions out of which distributed systems can be built. Services can be invoked independently by either external or internal service consumers to process simple functions, or can be chained together to form more complex functionality and so to quickly devise new functionality.

What a service is

A service can be defined as any discrete function that can be offered to an external consumer. This can be an individual business function or a collection of functions that together form a process. For example, the process of a car dealer selling a car to a new client involves process steps like:

1. Create client.

2. Correlate car and client.

Each of these steps can be designed as an independent service. A higher-level service could use both and bind the single process steps to one process.

A service consists of the following three parts, which together are able to deliver usable service functionality:

► Service contract

   The service contract provides the formal specification of a service and contains information like purpose, functionality, format of request and reply, pre-conditions and post-conditions, exception handling, quality of service, and so on. A formal interface definition provided by IDL or WSDL can be seen as an optional part of the service contract.

► Service interface

   The service interface exposes the implementation of the functionality. Although the description of the interface is part of the service contract, the physical implementation is contained in the service producer and consumer.

► Service implementation

   The service implementation provides the business logic that is exposed by the service interface.

1.1.3 Design principles

To meet the requirements of the drivers, a set of design principles is needed to refine the desired characteristics of the system. Because services are a crucial part of SOA, the principles mostly define their behavior and interaction.

Granularity

Services should be delivered at a level of granularity and abstraction that is meaningful to the service requestor. Many descriptions of SOA refer to the use of large-grained services; however, some powerful counter examples of successful, reusable, fine-grained services exist. For example, getBalance is a very useful service, but hardly large grained.

More realistically, there will be many useful levels of service granularity in most SOAs. For example, all of the following are services; however, they have different granularity. Some degree of choreography or aggregation is required between each granularity level for them to be integrated in an SOA:

► Technical Function Services

   For example, auditEvent, checkUserPassword, and checkUserAuthorization

► Business Function Services

   For example, calculateDollarValueFromYen and getStockPrice

► Business Transaction Services

   For example, checkOrderAvailability and createBillingRecord

► Business Process Services

   For example, openAccount, createStockOrder, reconcileAccount, and renewPolicy

A service can be any business or technical function; however, in an SOA it is preferable that the function is genuinely reusable. It is therefore inevitable that conventions will need to be defined for assigning services to corresponding abstraction layers. A criteria for this definition may be the level on which a service provides a reasonable amount of reuse.

Figure 1-1 on page 5 provides an overview of the different abstraction levels within an SOA.
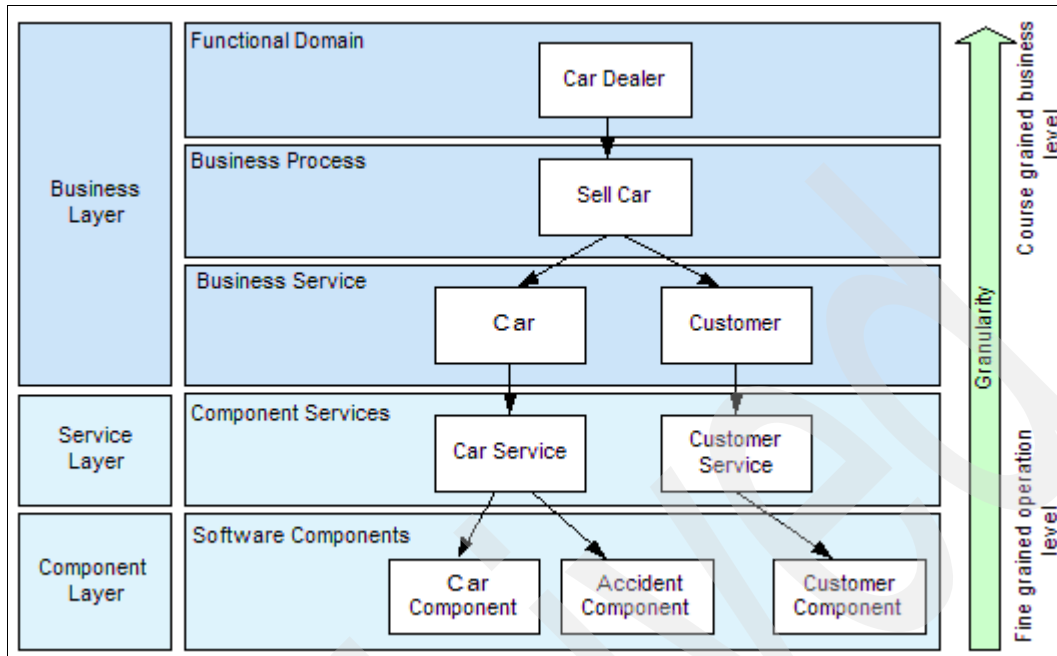
Figure 1-1   SOA abstraction layers

Figure 1-1 is described below:

▶ Functional domain

Acts as the owner of different business processes. For example, a car dealer needs to have a process for selling cars.

▶ Business process

A business process typically has a state attached to it and often acts as a controller or workflow engine by calling different lower-level services providing parameters that reflect the actual state of the process. For example, the selling of a car involves the credit check for a client as well.

▶ Business services

A business service is atomic in nature but orchestrates the invocation of lower-level component services into a business-level process. A business service is stateless from the view of the requestor but may hold internal state while calling lower-level services. Business services can be called synchronously or asynchronously. For example, a car business service provides the functionality of checking if a specific car is reserved for a client before it performs the selling.

▶ Component services

A component service is a simple atomic action on a simple entity that does not depend on another service to function. For example, database access to a single table like the car table can be thought of as a component service.

► Software components

This layer contains enterprise resources, existing systems, or applications including packaging application and historical application. For example, the car table within the database is a software component.

Fine-grained services are best consumed by more coarse-grained services instead of directly by the end applications. If an application is built using fine-grained services, the application will have to invoke multiple services over the network, each of which exchanges a small amount of data. Consumers of coarse-grained services are not exposed to the fine-grained services that they use.

Modularity

In an SOA, services are described as being modular and self-contained. A service supports a set of interfaces. These interfaces should be cohesive, which means they should all relate to each other in the context of a module. An appropriate service modularity in an SOA enables the aggregation of the services into an application with a few well-known dependencies.

There exists a set of criteria to help determine wether a service is sufficiently modular:

► Modular decomposability (top-down approach)

The modular decomposability of a service refers to the breaking of an application into many smaller modules. The goal is to identify the smallest unit of software that can be reused in different contexts.

► Modular composability (bottom-up approach)

The modular composability of a service refers to the implementation of services that can be reused as they are to implement a new system. In contrast to the bottom-up approach, which is more focused on the application functions, the top-down approach is focused on the business problem.

► Modular understandability

The modular understandability of a service refers to the ability of a person to understand the function of a service without knowing the others. For example, a service name called CustomerSellCar that mixes the semantics of the customer service and a car service limits the modular understandability. If a service is not understood from a functional perspective, it might never be reused.

► Modular continuity

The modular continuity of a service refers to the impact of a change in one service requiring a change in others or in the consumers of the service. An interface that does not sufficiently hide the implementation details of a service creates a domino effect when changes are needed.

► Modular protection

The modular protection of a service is sufficient if an abnormal condition in the service does not cascade to other services or consumers. For example, validating user input at its source prevents the propagation through the application.

Taking these criteria into consideration noticeably increases the modular design of services.

Loose coupling

Services are described as being loosely coupled. The systems must have some common understanding to conduct an interaction. Instead, to achieve the benefits of loose coupling, consideration should be given to how to couple or decouple various aspects of service interactions, such as the platform and language in which services are implemented, the communication protocols used to invoke services, and the data formats used to exchange input and output data between service consumers and providers.

► Language independence

The language independence of services refers to their ability to communicate with other services independent of the programming language that both are written with.

► Transport protocol transparency

The transport protocol transparency of services refers to their ability to communicate with other services independent of the protocol that both are connected with. From a requestor's perspective, the transport protocol that the provider is using has no impact on how a request is made.

► Location transparency

The location transparency of services refers to their ability to communicate with other services without knowledge of where they really reside. From a requestor's perspective, the location of the service has no impact on how a request is made.

► Data format independence

Data format independence of services refers to their ability to communicate with each other without using exactly the same schema (structure, element types) for the data they need to exchange. From a requestor's perspective, the schema that a provider expects has no impact on how a request is made.

► Platform independence

The platform independence of services refers to their ability to communicate with others independent of the kind of platform they are located on.

► Communication model transparency

The communication model transparency of services refers to their ability to communicate with each other independent of whether they are connected through a synchronous or asynchronous medium.

1.2 SOA and messaging

Successfully implementing an SOA requires applications and infrastructure that can support the SOA principles. Applications can be enabled by creating service interfaces to existing or new functions hosted by the applications. The service interfaces should be accessed using an infrastructure that can route and transport service requests to the correct service provider. As organizations expose more and more functions as services, it is vitally important that this infrastructure should support the management of SOA on an enterprise scale.

Benefits of messaging in an SOA environment

The design principles of granularity and modularity are resolved primarily at the application level. In contrast, the aspect of loose coupling can be greatly addressed by using messaging middleware. The use of such middleware supports the principles of an SOA implementation by:

► Decoupling the consumer's view of a service from the actual implementation of the service

► Decoupling technical aspects of service interactions

► Integrating and managing services in the enterprise

Decoupling the consumer's view of a service from the actual implementation greatly increases the flexibility of the architecture. It allows the substitution of one service provider for another (for example, because another provider offers the same services for lower cost or with higher standards) without the consumer being aware of the change, or needing to be altered to support it.

This decoupling is better achieved by having the consumers and providers interact via an intermediary. Intermediaries publish services to consumers. The consumer binds to the intermediary to access the service, with no direct coupling to the actual provider of the service. The intermediary maps the request to the location of the real service implementation.

Figure 1-2 shows the requestor and the provider connected using a messaging infrastructure as an intermediary.
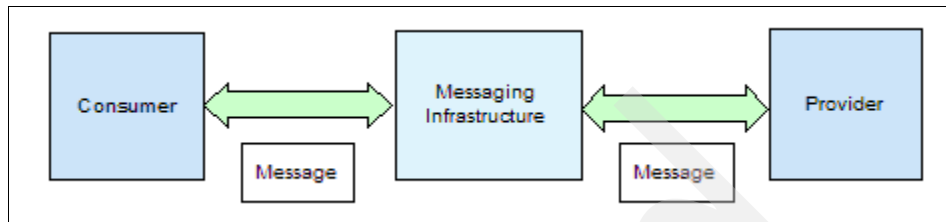
Figure 1-2   Decoupling requestor and provider using messaging as intermediary

Table 1-1 shows an overview of the coupling aspects related to the use of messaging middleware.

Table 1-1   Coupling aspects of messaging middleware

Coupling aspect	Justification
Language independence	The payload of a message can be passed in a language-independent manner. An appropriate interface to the messaging middleware needs to exist for requestor and provider.
Transport protocol transparency	The transport protocol used is encapsulated by the interface of the messaging infrastructure. A requestor does not need to know if a provider is connected using the same transport protocol.
Location transparency	For a service requestor or provider the messaging infrastructure is just a communication medium. A requestor does not need to know the route a message takes as long as it gets the result it expects.
Data format independence	The payload of a message is passed in a data format independent manner.
Platform independence	Messaging infrastructure supports the communication between different platforms and even provides mapping functionality between different data and encoding formats.
Communication model transparency	Messaging not only supports the synchronous communication model but also the asynchronous, thus providing enhanced flexibility in binding and orchestrating services.

1.3 Using an enterprise service bus

An enterprise service bus (ESB) provides an infrastructure that removes any direct connection between service consumers and providers. Consumers connect to the bus and not the provider that actually implements the service. This type of connection further decouples the consumer from the provider. A bus also implements further value add capabilities. For example, security and delivery assurance can be implemented centrally within the bus instead of having this buried within the applications.

Integrating and managing services in the enterprise outside of the actual implementation of the services in this way helps to increase the flexibility and manageability of SOA. The primary driver for an ESB, however, is that it increases decoupling between service consumers and providers. Protocols such as Web services define a standard way of describing the interface to a service provider that allow some level of decoupling, as the actual implementation details are hidden. However, the protocols imply a direct connection between the consumer and provider.

Although it is relatively straightforward to build a direct link between a consumer and provider, these links can lead to an interaction pattern that consists of building multiple point-to-point links (Figure 1-3) that perform specific interactions. With a large number of interfaces this quickly leads to the build up of a complex spaghetti of links with multiple security and transaction models. Routing control is distributed throughout the infrastructure, and probably no consistent approach to logging, monitoring, or systems management is implemented. This environment is difficult to manage or maintain and inhibits change.
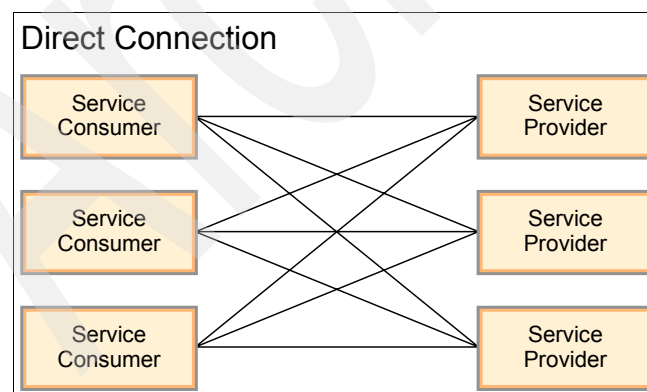
Figure 1-3   Direct connection integration style

A common approach to reduce this complexity is to introduce a centralized point through which interactions are routed, as shown in Figure 1-4.
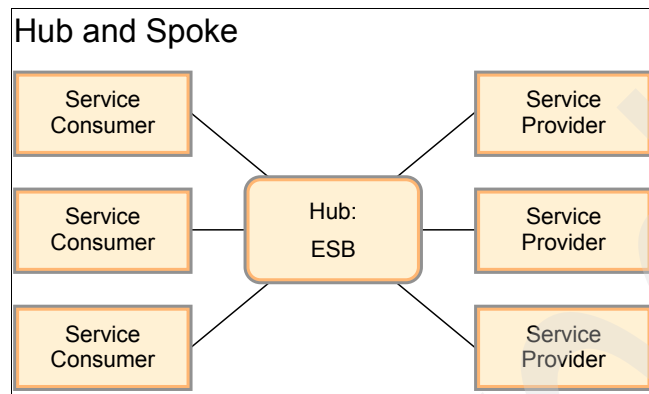
Hub and Spoke

Figure 1-4   Central hub integration style

This hub and spoke architecture is a common approach that is used in application integration architectures. In a hub, the distribution rules are separated from applications. The applications connect to the hub and not directly to any other application. This type of connection allows a single interaction from an application to be distributed to multiple target applications without the consumer being aware that multiple providers are involved in servicing the request. This connection can reduce the proliferation of point-to-point connections.

Note that the benefit of reducing the number of connections only truly emerges if the application interfaces and connections are genuinely reusable. For example, consider the case where one application needs to send data to three other applications. If this is implemented in a hub, the sending application must define a link to the hub, and the hub must have links that are defined to the three receiving applications, giving a total of four interfaces that need to be defined. If the same scenario was implemented using multiple point-to-point links, the sending application would need to define links to each of the three receiving applications, giving a total of just three links. A hub only offers the benefit of reduced links if another application also needs to send data to the receiving applications and can make use of the same links as those that are already defined for the first application. In this scenario, the new application only needs to define a connection between itself and the hub, which can then send the data correctly formatted to the receiving applications.

Hubs can be federated together to form what is logically a single entity that provides a single point of control but is actually a collection of physically distributed components. This is commonly termed a bus. A bus provides a

consistent management and administration approach to a distributed integration infrastructure.

ESB capabilities

This section discusses the capabilities an ESB must have to support the requirements of an SOA enabling infrastructure component. Understanding the capabilities allows you to assess the suitability of individual technologies or products for implementing an ESB by analyzing the functionality that they offer. In discussions on ESB, the most commonly agreed upon elements for defining an ESB are:

► The ESB is a logical architectural component that provides an integration infrastructure that is consistent with the principles of SOA.

► The ESB can be implemented as a distributed, heterogeneous infrastructure.

► The ESB provides the means to manage the service infrastructure and the capability to operate in a distributed, heterogeneous environment.

Table 1-2 summarizes the capabilities that an ESB should have in order to provide an infrastructure consistent with these elements, and thus consistent with the benefits of SOA.

Table 1-2   Capabilities of an ESB

Category	Capabilities	Reason
Communication	► Routing ► Addressing ► At least one messaging style (request/response, publish/subscribe) ► At least one transport protocol that is or can be made widely available	Provides location transparency and supports service substitution
Integration	► Several integration styles or adapters ► Protocol transformation	Supports integration in heterogeneous environments and supports service substitution
Service interaction	► Service interface definition ► Service messaging model ► Substitution of service implementation	Separates application code from specific service protocols and implementations
Management	► Administration capability	A point of control over service addressing and naming

Let us take a closer look at each of these categories:

► Communication

The ESB needs to supply a communication layer to support service interactions. It should support communication through a variety of protocols. It should provide underlying support for message and event-oriented middleware and integrate with existing HTTP infrastructure and other enterprise application integration (EAI) technologies. As a minimum capability, the ESB should support at least the protocols that make sense given the requirements of a specific situation. The ESB should be able to route between all these communication technologies through a consistent naming and administration model.

► Integration

The ESB should support linking to a variety of systems that do not directly support service-style interactions so that a variety of services can be offered in a heterogeneous environment. This includes existing systems, packaged applications, and other EAI technologies. Integration technologies might be protocols (for example, JDBC™, FTP, or EDI) or adapters such as the J2EE Connector Architecture resource adapters or WebSphere Business Integration Adapters. It also includes service client invocation through client APIs for various languages (Java™, C+, or C#) and platforms (J2EE or .Net), CORBA, and RMI.

► Service interaction

The ESB needs to support SOA concepts for the use of interfaces and support declaration service operations and quality of service requirements. The ESB should also support service messaging models consistent with those interfaces, and be capable of transmitting the required interaction context, such as security, transaction, or message correlation information.

► Management

As with any other infrastructure component, the ESB needs to have administration capabilities so that it can be managed and monitored to provide a point of control over service addressing and naming. In addition, it should be capable of integration into systems management software.

1.4 The IBM SOA Foundation

IBM SOA Foundation is an integrated, open-standards-based set of software, best practices, and patterns that can help you get started with your SOA. It is designed to help you extend the value of the applications and business processes that currently run your business.

You can read about the IBM SOA Foundation in *IBM SOA Foundation: providing what you need to get started with SOA* at:

ftp://ftp.software.ibm.com/software/soa/pdf/SOA_g224-7540-00_WP_final.pdf

The following is a quick summary of the highlights.

1.4.1 SOA life cycle

SOA can be thought of in terms of a life cycle consisting of the following stages:

- The *model* phase consists of gathering business requirements and designing business processes.

- In the *assemble* phase the processes are implemented by assembling new and existing services to form these business processes.

- In the *deploy* phase, these assets are deployed into a secure integrated services environment.

- And last, in the *manage* phase, these business processes are monitored and managed from both an IT and business perspective. Information gathered in this phase is fed back into the life cycle to enable improvement.

Governance and processes underpin these life-cycle stages to provide guidance and oversight for the SOA project.

Each life-cycle stage is supported by software that has been chosen to be a part of the IBM SOA Foundation.

1.4.2 SOA Reference Architecture

IBM SOA Foundation is based on the SOA Reference Architecture, which defines the comprehensive IT services required to support your SOA at each stage in the SOA life cycle. The SOA Reference Architecture includes development environment, services management, application integration, and runtime process services. The capabilities of the architecture can be implemented on a build-as-you-go basis as new requirements are addressed over time.

Figure 1-5 on page 15 shows the SOA Reference architecture and a sampling of the supporting software.
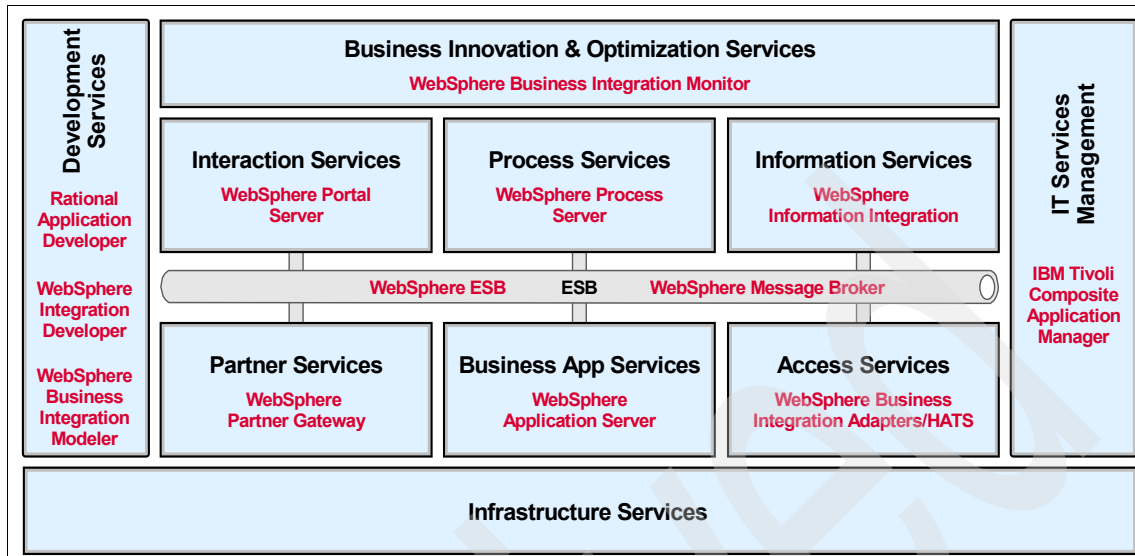
Figure 1-5   SOA Reference Architecture with product mapping

The following services are defined in the SOA Reference Architecture:

▶ *Infrastructure services* optimizes throughput availability and performance.

▶ *Partner services* connects with trading partners.

▶ *Business application services* provides a scalable, highly available, and secure application environment.

▶ *Access services* facilitates interaction with existing information and application assets using application adapters or host access services.

▶ *Interaction services* manages collaboration between people, processes, and information.

▶ *Process services* orchestrates and automates business processes.

▶ *Information services* manages diverse data in a unified manner.

▶ *Business innovation and optimization services* provide real-time business information monitoring to facilitate better decision making.

▶ *Enterprise service bus* facilitates communication between services.

▶ *Development services* provides an integrated environment to design and create solution assets.

▶ *IT service management* manages and secures services, applications, and resources.

Model and assemble phases

The Development Services provide services for the model and assemble phases of the SOA life cycle. The IBM development tools are based on a common platform, making the learning curve easy as you progress from tool to tool. These development tools are tailored to the task at hand and align with the runtime environments.

Rational Application Developer is the primary development tool for WebSphere Application Server applications. With this tool, you can create J2EE assets such as servlets and EJBs, test the code, and deploy to the runtime server. Among the many useful features are the Web services tools that provide wizards to assist you in creating Web services clients, Web services providers, and in wrapping existing applications as services.

WebSphere Integration Developer is the primary development tool for WebSphere Process Server and WebSphere ESB. With this tool you can create a business process with BPEL, configure adapters, create mediations, test the code, and deploy to the runtime server.

Manage phase

The IT Service Management provides services for the manage phase, including the management and security of services, applications, and resources. The following IBM products support the manage phase:

► IBM WebSphere Business Monitor
► IBM Tivoli® Composite Application Manager
► IBM Tivoli Identity Manager and IBM Tivoli Access Manager

Deploy phase

The remainder of the services are involved in the deploy phase, providing runtime services. The following IBM products support the deploy phase:

► IBM WebSphere Process Server
► IBM WebSphere ESB and IBM WebSphere Message Broker
► IBM WebSphere Partner Gateway and IBM WebSphere Adapters
► IBM WebSphere Portal
► IBM WebSphere Everyplace® Deployment
► IBM Workplace™ Collaboration Services
► IBM WebSphere Information Integrator
► IBM WebSphere Application Server
► IBM WebSphere Extended Deployment

1.5 For more information

For more information about the concepts covered in this chapter, see:

► IBM Service Oriented Architecture (SOA) Web page

   http://www-306.ibm.com/software/solutions/soa/

► *IBM SOA Foundation: providing what you need to get started with SOA*

   ftp://ftp.software.ibm.com/software/soa/pdf/SOA_g224-7540-00_WP_final.pdf

► *Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6*, SG24-6494

► *Patterns: Implementing Self-Service in an SOA Environment*, SG24-6680

2

Product selection

This chapter discusses the IBM products that form the foundation for messaging in the IBM SOA strategy. It will help you determine what products may be appropriate for your situation.

2.1 IBM SOA Foundation products for messaging

In this book we take a look at the set of products shown in Figure 2-1. These products form the heart of the IBM SOA strategy. In particular, we take these within the context of a messaging solution.
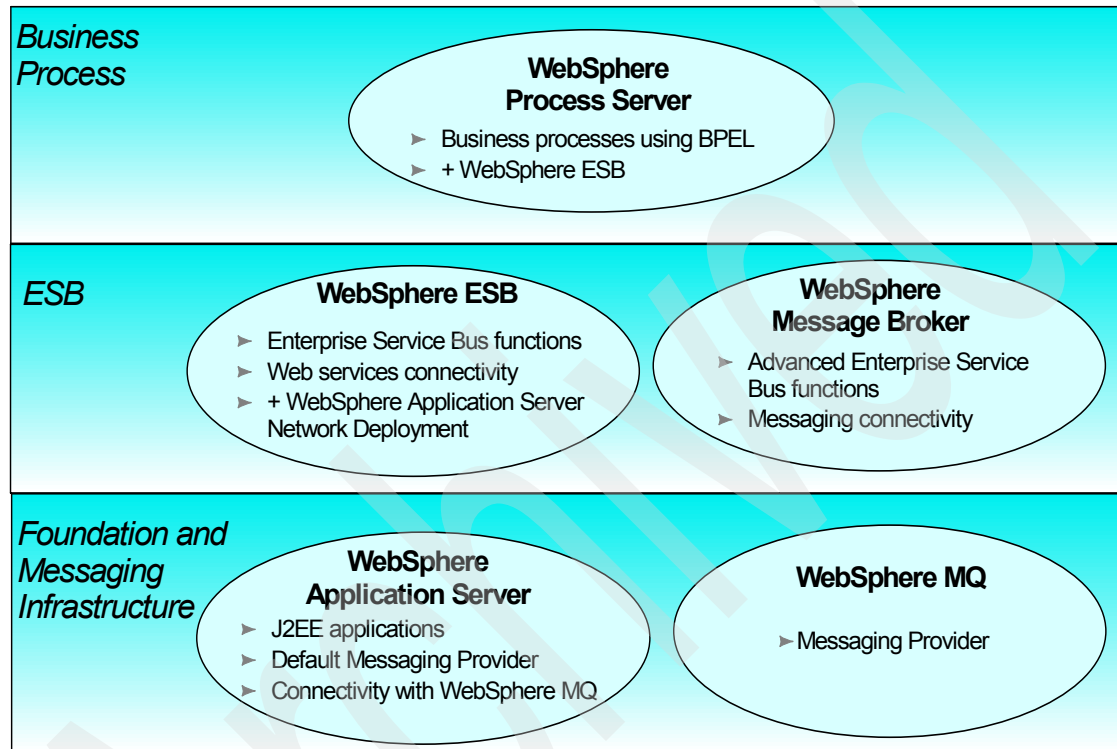
Figure 2-1   IBM SOA products for messaging

Figure 2-1 is explained below:

► Foundation and messaging infrastructure

WebSphere Application Server provides the runtime environment for J2EE applications. WebSphere Application Server provides the J2EE containers required to execute the applications, as well as the services that enable the execution of specific Java application components. WebSphere Application Server hosts a JMS-based messaging engine that provides enhanced messaging functionality and a Web services engine that is capable of hosting and invoking SOAP-based Web services.

WebSphere MQ provides the transport mechanism for messages. WebSphere MQ supports assured, asynchronous, once-only delivery of

messages across a broad range of hardware and software platforms like Windows®, z/OS®, .NET, and J2EE and is accessible using various programming languages and interfaces like Visual Basic®, C++, Java, COBOL, PL/I, as well as MQI and JMS.

► ESB

The enterprise service bus (ESB) layer provides flexible connectivity and integration for Web services and messaging-focused applications.

WebSphere Enterprise Service Bus (WebSphere ESB) provides ESB functionality for standards-based applications. Built on WebSphere Application Server, WebSphere ESB takes advantage of its many features, including high availability, scalability, and performance. Beyond the basic programming-based mediation functionality available in WebSphere Application Server, WebSphere ESB provides a mediation layer with pre-built mediations for XML transformation, content-based routing, and message logging. Like WebSphere Process Server, WebSphere ESB supports Service Component Architecture (SCA) and Service Data Objects (SDO) to provide a unified programming model.

WebSphere Message Broker provides advanced ESB functionality for universal support of messaging applications. Built on WebSphere MQ, WebSphere Message Broker takes advantage of the services provided by the messaging infrastructure and enhances them by adding a runtime environment that supports message processing like message transformation and routing.

► Business process

WebSphere Process Server provides services to enable the integration of composite applications. It addresses issues of system and technology heterogeneity by applying the IBM SOA programming model consisting of SCA and SDO. SCA and SDO allow the definition of service component interfaces, implementations, and references in a technology-neutral way that can be bound later to the technology chosen. On top of this technology-neutral service component layer is a Business Process Execution Language (BPEL)-based orchestration and runtime environment for the dynamic composition of the service components to processes.

> **IBM Message Service API (XMS):**
>
> This book focuses on JMS applications; however, you should be aware of the IBM Message Service API (XMS). XMS is a programming API that allows access from C, C++, and .NET applications to the following IBM messaging servers:
>
> - ► WebSphere MQ
> - ► WebSphere Message Broker real-time transport
> - ► WebSphere Application Server V6 default messaging provider
>
> Similar to JMS, it provides a mechanism for non-Java applications to participate in point-to-point and pub/sub messaging. It supports both synchronous and asynchronous messaging styles.
>
> - ► *Introducing XMS -- The IBM Message Service API*
>
>   `http://www-128.ibm.com/developerworks/websphere/library/techarticles/0509`
>   `_phillips/0509_phillips.html`

2.2 WebSphere Application Server

The foundation of the WebSphere brand is the application server, which provides the runtime environment and management tools for J2EE and Web services based applications. WebSphere Application Server provides qualities of service like clustering, failover, scalability, and security. It also includes a built-in messaging provider, which can be configured to connect to an existing WebSphere MQ network.

WebSphere Application Server is available in three packages:

- ► WebSphere Application Server Express

  The Express package is geared to those who need to get started quickly with e-business. It is specifically targeted at medium-sized businesses or departments of a large corporation, and is focused on providing ease of use and ease of application development. It contains full J2EE 1.4 support but is limited to a single-server environment. WebSphere Application Server - Express is bundled with the Rational Web Developer application development tool.

- ► WebSphere Application Server

  The WebSphere Application Server package provides the next level of server infrastructure. Though the server is functionally equivalent to the server shipped with Express, this package differs slightly in packaging and licensing.

The development tool included is a trial version of a Rational Application Developer, full J2EE 1.4 compliant development tool.

► WebSphere Application Server Network Deployment

WebSphere Application Server Network Deployment is an even higher level of server infrastructure in the WebSphere Application Server family. It extends the WebSphere Application Server base package to include clustering capabilities, Edge components, and high availability for distributed configurations. These features become more important at larger enterprises, where applications tend to service a larger client base, and more elaborate performance and availability requirements are in place.

The basic architectural model for WebSphere Application Server is shown in Figure 2-2 on page 24.
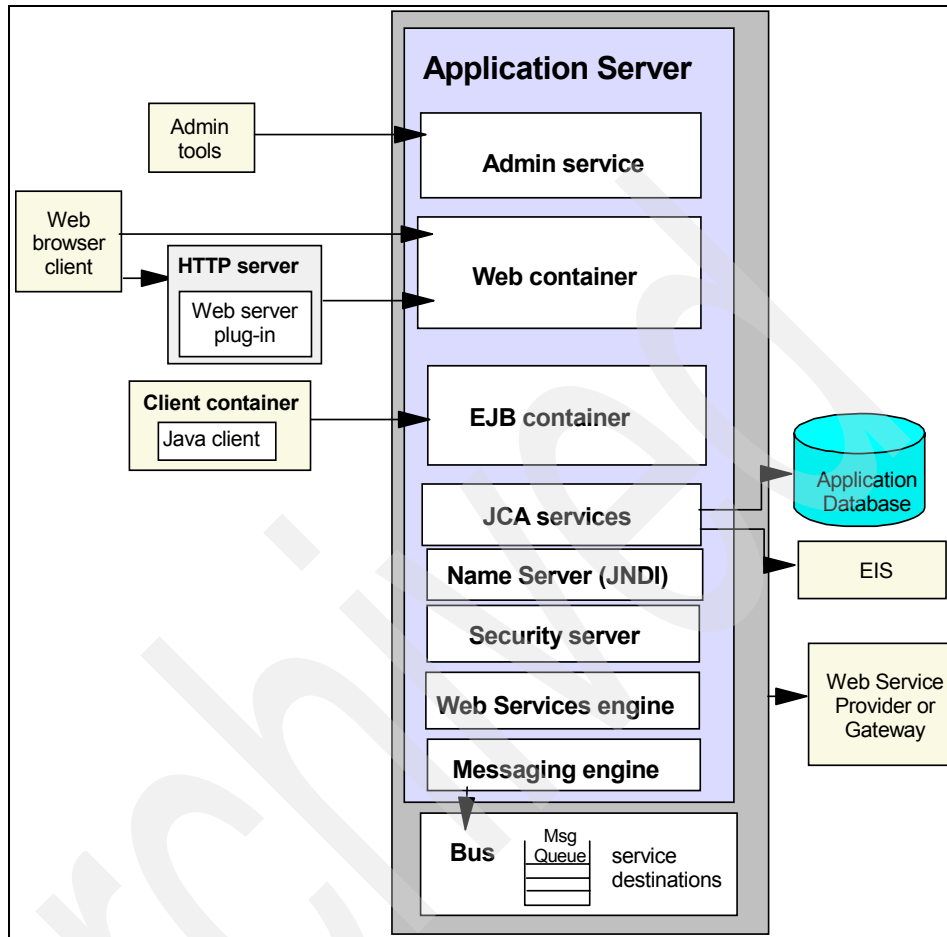
Figure 2-2   Architectural model of WebSphere Application Server

The architectural model for Network Deployment is similar to this, but adds a deployment manager for central administration and management for all application servers grouped together in a cell. In Network Deployment, the admin services resides in the deployment manager. Multiple servers are possible and application server clustering is used for workload management and high availability.

WebSphere Application Server V6 provides full support for the J2EE 1.4 specification. The J2EE specification defines the concept of containers to provide runtime support for applications. There are three types of containers in the application server implementation:

► Web container

The Web container processes HTML, servlets, JSP™ files, and other types of server-side includes. It provides infrastructure support like Web container transport chains, session management, and Web services engine.

► EJB™ container

The EJB container provides all the runtime services that are needed to deploy and manage enterprise beans. It is a server process that handles requests for both session and entity beans. The container provides low-level services including threading and transaction support.

► Client container

The client container is a separately installed component on the client's machine. It allows the client to run applications in an environment that is compatible with J2EE.

In addition to the definition of containers as a runtime environment for application components, J2EE prescribes the support of the J2EE Connection Architecture (JCA) that provides connection management for access to enterprise information systems (EIS). The connection between the enterprise application hosted by the application server and the EIS is done through the use of EIS-provided resource-adapters, which are plugged into the application server. The architecture specifies the connection management, transaction management, and security contracts that exist between the application server and the EIS.

Each application server hosts a name service that provides a Java Naming and Directory Interface™ (JNDI) name space. The service is used to register resources hosted by the application server. The JNDI implementation in WebSphere Application Server is built on top of a Common Object Request Broker Architecture (CORBA) naming service (CosNaming). The naming architecture is used by clients of WebSphere applications to obtain references to objects related to those applications. These objects are bound into a mostly hierarchical structure, referred to as a namespace. The namespace can be accessed and manipulated through a name server.

WebSphere Application Server security provides flexibility by providing pluggable modules that can be configured according to requirements and existing IT resources. The application server's security features sit on top of the operating system security and the security features provided by other components, including the Java language. The WebSphere Application Server supports J2EE security with the required role mapping as well as Java 2 security and JAAS support. The authentication features support LTPA and SWAM and the applied user registry can be based on the local operating system, LDAP, or a custom registry.

The WebSphere Application Server supplies messaging providers for its own messaging implementation (default messaging provider) and for WebSphere MQ (WebSphere MQ JMS provider). It also supports generic JMS providers.

The default messaging provider is WebSphere Application Server's JMS API implementation (connection factories, JMS destinations, and so on) for messaging. The concrete destinations (queues and topic spaces) behind the default messaging provider are implemented in the service integration bus. (Similarly, the WebSphere MQ JMS provider is the JMS API implementation with WebSphere MQ implementing the real destinations for the JMS interface.) Application server access to the service integration bus is managed by the messaging engine.

WebSphere messaging provides a consolidation of support for queuing, pub/sub, and Web services, thus providing a platform for several key messaging and interaction patterns. For example, both pseudo-synchronous request-reply messaging and asynchronous fire-and-forget messaging are supported. Messaging engines connect to a service integration bus that provides message transport. Message producers and consumers communicate by interacting with the bus, not directly with each other. A key feature of the bus is the ability to use a variety of protocols to send and receive messages to and from the bus. For example, a message can be sent into the bus as a Web services request message based on the SOAP protocol over HTTP. The bus can then forward this message to a JMS consumer. The messaging engine supports the concept of mediations. A mediation is a piece of code that is associated with a destination, the logical target for a message. Mediation code operates on a message as it traverses that destination and thus supports, for example, message transformation and routing.

The Web services engine of the WebSphere Application Server supports SOAP-based Web service hosting as well as invocation. It contains support for various Java and Web services standards including WS-I Basic Profile, WS-Security, JAX-RPC, JAXR, SAAJ, and UDDI.

Web server plug-ins enable the Web server to send requests for dynamic content, such as servlets, to the application server. A configuration file that contains information about the application server configuration is generated at the application server and copied to the Web server for use by the plug-in. The plug-in provides load balancing among application servers.

With Network Deployment, clustering application servers automatically enables plug-in workload management for the application servers and the servlets they host. The routing is based on weights associated with the cluster members. If all cluster members have identical weights, the plug-in sends equal requests to all members of the cluster. Workload management for EJB containers can be performed by configuring the Web container and EJB containers on separate

application servers. Multiple application servers with the EJB containers can be clustered, enabling the distribution of EJB requests between the EJB containers.

WebSphere Application Server Network Deployment also provides high availability features. The following is a quick overview of the failover capabilities:

► Web container failover

The Web server plug-in in the Web server is aware of the configuration of all Web containers and can route around a failed Web container in a cluster. Sessions can be persisted to a database or in-memory using data replication services.

► EJB container failover

Client code and the ORB plug-in can route to the next EJB container in the cluster.

► Critical services failover

Hot standby and peer failover for critical services (such as workload management routing, PMI aggregation, JMS messaging, transaction manager, and so on) is provided through the use of high availability domains. A high availability domain defines a set of WebSphere processes (core group) that provides high availability functions to each other.

One or more members of the core group can act as a high availability coordinator, managing the HA activities within the core group processes. If a high availability coordinator server fails, another server in the core group takes over the duties of that coordinator. High availability policies define how the failover occurs. Workload management information is shared between core members and failover of critical services is done among them in a peer-to-peer fashion. Little configuration is necessary, and in many cases, this function works with the defaults that are created automatically as you create the processes.

► JMS messaging failover

The messaging engine keeps messages in a remote database. When a server in a cluster fails, WebSphere selects an online server to run the Messaging Engine and the workload manager routes JMS connections to that server.

WebSphere Application Server provides a browser-based administrative console for administration. Command-line and scripting administration is also provided.

You can find more information about the WebSphere Application Server at the WebSphere Application Server home page:

http://www.ibm.com/software/webservers/appserv/was/

2.3 WebSphere MQ

IBM WebSphere MQ is an established and reliable message queuing middleware platform. A message queuing infrastructure built upon WebSphere MQ technology can provide an available, reliable, scalable, secure, and maintainable transport for messages with exactly once delivery assurance.

Figure 2-3 shows the architectural overview for WebSphere MQ, including the interface for Java Message Service (JMS) and the Message Queuing Interface (MQI).
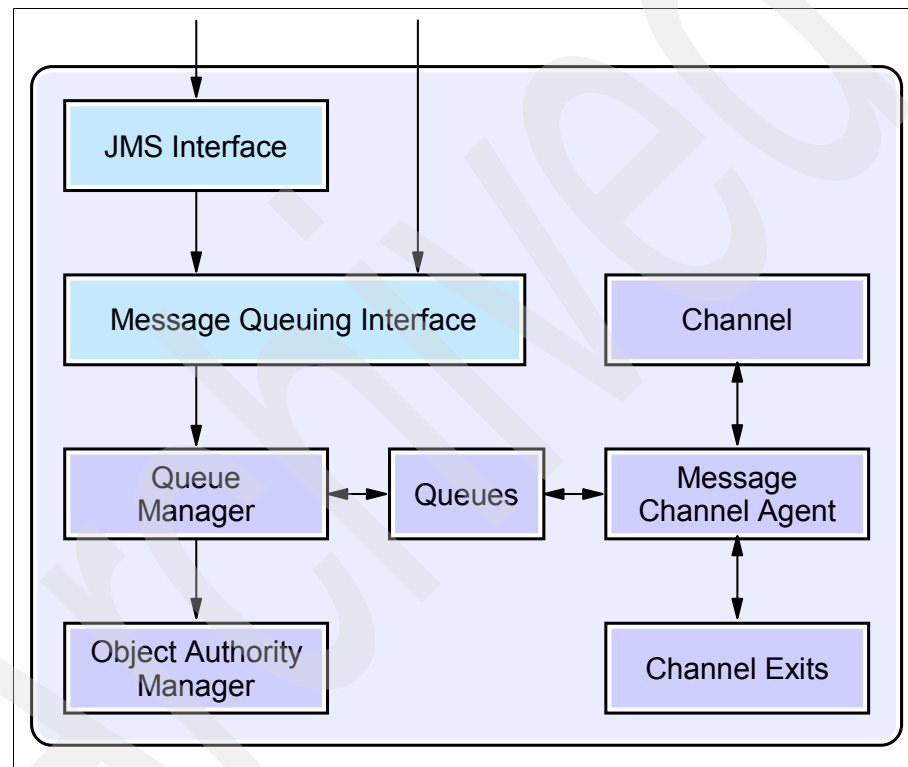
Figure 2-3   Architectural model of WebSphere MQ

MQI is the core API provided by WebSphere MQ. It is a procedural API suitable for applications developed within procedural programming languages. The MQI also defines structures, constants, and basic data types required to interact with WebSphere MQ. Procedural languages like C and COBOL most likely utilize MQI directly, while object-oriented languages like Java and C++ are supported with object-oriented APIs built upon MQI.

WebSphere MQ also supports XMS, a programming API that allows access from C, C++, and .NET applications.

Queue managers are the core element within a WebSphere MQ messaging infrastructure. They provide queuing services to applications and manage the queues that belong to them. Queue managers ensure that object attributes are changed according to the commands they receive. They trigger special events when certain conditions are met and they put messages onto the correct queue. Every application that wants to access a WebSphere MQ messaging infrastructure needs to be connected to a queue manager within this infrastructure. Applications can only retrieve messages from queues hosted by the queue manager to which they are connected, but they can send messages to queues hosted by other queue managers, as long as a network link exists between both queue managers. A machine with a WebSphere MQ installation can host multiple queue managers; the amount is limited only by the resources of the machine.

WebSphere MQ defines different kinds of queues:

► Local queues

 Local queues are the only type of queues that represent a real queue and hold messages.

► Transmission queues

 Transmission queues utilize message channels to transmit messages to remote queue managers. They provide a queue manager with knowledge of how to route messages to a single destination queue manager. Any messages sent with a queue manager name the same as the name of the transmission queue are placed upon that transmission queue.

► Alias queues

 An alias queue is a representation of another target queue, which has a different name. Alias queues can be used to enhance flexibility regarding naming.

► Model queues

 Model queues provide the attributes of a local queue that can be created dynamically by an application. Dynamically created queues are instances of local queues and can hold messages. They can, for example, be used as dynamic reply queues in request-reply scenarios.

► Remote queues

 Remote queues are used to define routes to other queue managers within the WebSphere MQ messaging infrastructure. This involves mapping queue manager names to transmission queues, and mapping queue names to different queue names on remote queue managers.

A channel is a communication object used by distributed queue managers. There are two types of channels:

► Message channels, which are unidirectional and transfer messages from one queue manager to another

► MQI channels, which are bidirectional and transfer MQI calls from a WebSphere MQ client to a queue manager and responses from a queue manager to the client

Every channel in WebSphere MQ is a network link between two Message Channel Agents (MCAs). A connection performed by an application connecting to a queue manager is also performed by an MCA, even though it is not performed from within a queue manager.

Channel exits are user-written libraries that are called at defined places in the processing sequence of an MCA. They can be used to do additional processing on messages, for example, encryption or data compression. Based on the functionality and place where they are called, there are different types of exits (message exit, message-retry exit, receive exit, security exit, send exit, and auto-definition exit).

WebSphere MQ provides features to assure security of access, authentication of identity, and security and integrity of communication. The Object Authority Manager (OAM) is the default authorization service for command and object management. All actions performed by an application connected to a queue manager are authenticated by the OAM.

WebSphere MQ provides high availability through workload balancing and failover capabilities. It supports the concept of queue manager clusters consisting of a set of queue managers. Applications requesting a particular service can connect to any queue manager within the cluster. The queue manager to which the application is connected automatically load balances the request with the others. Queue managers can dynamically join or leave clusters and can be hosted on different machines, which is especially useful in a distributed environment where capacity is scaled to accommodate the current load through multiple servers rather than one mainframe or high-capacity server.

WebSphere MQ provides reliability and data integrity by supporting units of work as well as persistent and non-persistent messages.

Units of work performed by applications accessing a WebSphere MQ infrastructure can include sending and receiving messages, as well as updates to databases. WebSphere MQ can coordinate all resources to ensure that a unit of work is only completed if all actions within that unit of work complete successfully. WebSphere MQ can also participate in units of work that are coordinated by other products. For example, actions against a WebSphere MQ

infrastructure can be included in units of work that are coordinated by WebSphere Application Server.

WebSphere MQ supports the concept of persistent and non-persistent messages. Persistent messages are retained during system failures but provide comparable low performance, while non-persistent messages provide high performance but may be lost during system failures.

WebSphere MQ contains monitoring and accounting functionality. It provides real-time performance information about flow of messages, it allows report generation for queue manager usage on application level base, and it provides facilities to identify the route that a messages took through a WebSphere MQ infrastructure or interconnected WebSphere MQ infrastructures.

Administration of WebSphere MQ is typically done using control commands or the WebSphere MQ Explorer administrative interface (Windows or Linux® only).

You can find more information about IBM WebSphere MQ at the WebSphere MQ home page:

    http://www.ibm.com/software/integration/wmq/

2.4 WebSphere ESB

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

The WebSphere ESB architectural model is shown in Figure 2-4 on page 32.
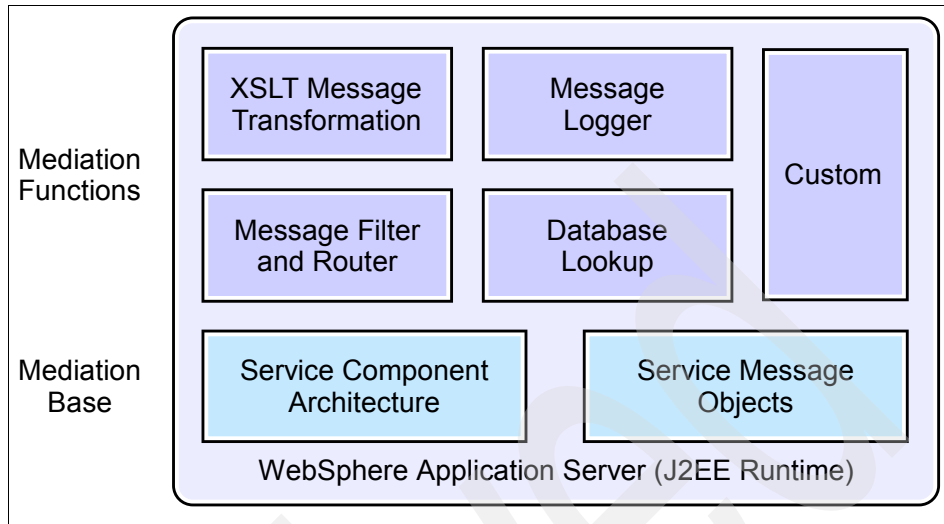
Figure 2-4   Architectural model of WebSphere ESB

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

On top of the infrastructure provided by WebSphere Application Server, WebSphere ESB implements a mediation layer consisting of a mediation base and mediation functions. The newly provided mediation framework is different from the one implemented by WebSphere Application Server, as it is based on the Service Component Architecture (SCA). It allows enhanced flexibility, encapsulation, and reuse. Mediations implemented for WebSphere Application Server can still be used together with WebSphere ESB, but the new tooling provided for WebSphere ESB does not support the modification of these mediations.

The mediation base is provided by SCA and Service Message Objects (SMO). SCA supports the description of every mediation module through a technology-neutral interface. SMO is based on SDO and supports the representation of a binding-specific data format in a common, neutral way. The application of this SCA/SMO-based programming model allows for the configurable assembly of different mediation modules to a mediation flow, thus enabling a very flexible and encapsulated solution.

Mediation functions are built upon the mediation base and consist of one or more mediation modules. An SCA/SMO-based mediation module is composed of different parts such as imports representing providers, exports representing service consumers, and mediation flow components representing integration and mediation functionality.

WebSphere ESB provides pre-built components called mediation primitives that can be used in mediation flows to perform XSLT message transformation, logging, routing, and database lookup. It also supports the implementation of custom mediation primitives.

WebSphere ESB supports different binding types for imports and exports, thus allowing the connection of different kinds of service consumers and providers. Supported binding types are JMS binding, Web services binding, WebSphere adapter binding, as well as a default binding used for module to module communication.

The mediation framework and its mediation modules separate the processing of requests from the processing of replies. They allow the mediation flow components to pass a potentially modified request from a service consumer to a service provider and to pass a potentially modified reply from a service provider to a service consumer. The request processing within a mediation flow component can send a reply back to the consumer without necessarily needing to contact a service provider.

2.4.1 Mediation functions in WebSphere ESB versus WebSphere Application Server

Before the announcement of WebSphere ESB, the service integration bus in WebSphere Application Server was often positioned as a basic ESB. Though this is still a useful strategy for development environments, WebSphere ESB is now the recommended solution for environments where the service integration bus was used.

WebSphere ESB adds the following functionality to the service integration bus:

► Easy-to-build mediation layer
► Simplified administration
► Pre-built mediation functions
► Broad connectivity

Mediation functions in WebSphere ESB are service intermediaries that:

► Operate on interactions between service endpoints (consumer and provider).

► Are administered as part of WebSphere ESB.

- Are created using visual tooling exploiting supplied and custom mediation functions.
- Have access to binding-specific header data like SOAP and JMS headers.

Mediation handlers in the WebSphere Application Server service integration bus are message handlers that:

- Operate on messages traversing the bus.
- Are administered as part of the bus.
- Are created by implementing Java programs.
- Allow access to the full WebSphere messaging header information.

You can find more information about the WebSphere ESB at the WebSphere ESB home page:

> http://www.ibm.com/software/integration/wsesb/

2.5 WebSphere Message Broker

WebSphere Message Broker enhances the flow and distribution of information by enabling the transformation and intelligent routing of messages without the need to change either the applications that are generating the messages or the applications that are consuming them.

Figure 2-5 on page 35 shows a high-level architectural view of WebSphere Message Broker.
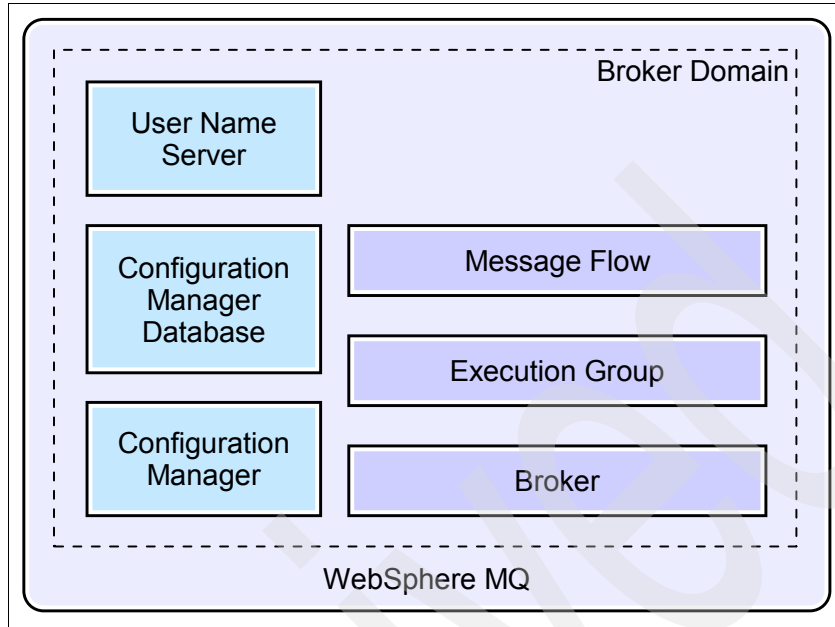
Figure 2-5   Architectural model of WebSphere Message Broker

The broker is a set of application processes that host and run message flows consisting of a graph of nodes that represent the processing needed for integrating applications.

When a message from a business application arrives at the broker, the broker processes the message before passing it on to one or more other business applications. The broker routes, transforms, and manipulates messages according to the logic that is defined in message flow applications. A broker uses WebSphere MQ as the transport mechanism both to communicate with the Configuration Manager, from which it receives configuration information, and to communicate with any other brokers to which it is associated. Each broker has a database in which it stores the information that it needs to process messages at run time.

Execution groups enable message flows within the broker to be grouped together. Each broker contains a default execution group. Additional execution groups can be created as long as they are given unique names within the broker. Each execution group is a separate operating system process and, therefore, the contents of an execution group remain separate from the contents of other execution groups within the same broker. This can be useful for isolating pieces of information for security because the message flows execute in separate address spaces or as unique processes. Message flow applications are

deployed to a specific execution group. To enhance performance, the same message flows and message sets can be running in different execution groups.

The Configuration Manager is the interface between the Message Brokers Toolkit and the brokers in the broker domain. The Configuration Manager stores configuration details for the broker domain in an internal repository, providing a central store for resources in the broker domain. The Configuration Manager is responsible for deploying message flow applications to the brokers and delivering reports on the progress of the deployment and on the status of the broker. When the Message Brokers Toolkit connects to the Configuration Manager, the status of the brokers in the domain is derived from the configuration information stored in the Configuration Manager's internal repository.

Brokers are grouped together in broker domains. The brokers in a single broker domain share a common configuration that is defined in the Configuration Manager. A broker domain can also contain a User Name Server. The components in a broker domain can exist on multiple machines and operating systems, and are connected together with WebSphere MQ channels. A broker belongs to only one broker domain.

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

WebSphere Message Broker together with WebSphere MQ provide high availability features. This is quite important since WebSphere Message Broker acts as a hub and therefore needs to be eliminated as a single point of failure.

Load balancing and high availability can be achieved by providing multiple broker instances serving the same logical hub with each instance mapped to its own WebSphere MQ queue manager. The different broker instances could reside on different machines. You can run multiple message broker instances with identical execution groups and message flows deployed so that each broker can process any message. It is also possible to run multiple brokers, each with specific message flows deployed to enable the distribution of specific processing to specific brokers (for prioritization purposes, for example). WebSphere MQ provides the base services for WebSphere Message Broker and must be set up for high availability.

WebSphere Message Broker provides the Message Broker Toolkit, a graphical environment for developing and deploying message flow applications.

For more information

You can find more information about IBM WebSphere Message Broker at the WebSphere Message Broker home page:

http://www.ibm.com/software/integration/wbimessagebroker/

2.6 ESB product comparison

The product you select to implement an ESB depends on the requirements of your solution. We have introduced two strategic products and described them. Now we provide a quick comparison of the two.

WebSphere ESB is designed to provide the core functionality of an enterprise service bus for a predominantly Web services based environment. It is built on WebSphere Application Server, which provides the foundation for the transport layer. WebSphere ESB adds a mediation layer based on the SCA programming model on top of this foundation to provide intelligent connectivity. If the client has a lot of Web services in their environment, WebSphere ESB is likely to be the better product to use.

WebSphere Message Broker provides a more advanced ESB solution with advanced integration capabilities such as universal connectivity and any-to-any transformation for data-centric deployments. It can handle services integration as well as integration with non-services applications. WebSphere MQ provides the transport backbone for messaging applications. Typically, clients who need a higher performance and throughput product in a message-centric environment would use Message Broker.

For a quick comparison of WebSphere MQ and WebSphere Message Broker see Table 2-1 on page 38.

Table 2-1   WebSphere ESB versus WebSphere Message Broker

	WebSphere ESB	WebSphere Message Broker
Connectivity	► MQ/JMS (via MQLINK configuration) JMS 1.1 (point-to-point, pub/sub) ► TCP/IP, SSL, HTTP(S), IIOP	► Native MQ, JMS 1.1 (point-to-point, pub/sub) ► Supports input handling for virtually all third-party JMS systems ► TCP/IP, SSL, HTTP(S), CICS®, VSAM, flat-files ► Supports MQ Enterprise Transport, MQ Mobile Transport, MQ Multicast Transport, MQ Realtime Transport, MQ Telemetry Transport, MQ Web Services Transport, JMS Transport
Web services support	► SOAP/HTTP(S), SOAP/JMS, WSDL 1.1 ► Supports WS-I Basic Profile 1.1 ► UDDI 3.0 Service Registry ► WS-Security, WS-Atomic Transactions ► Comprehensive client support by Message Service Client for C/C++ and .NET, Web Services Client, and J2EE Client	► SOAP/HTTP(S), SOAP/JMS, WSDL 1.1 ► Supports WS-I Basic Profile 1.0

	WebSphere ESB	WebSphere Message Broker
Adapter support	(JCA and WBI adapters	► WBI adapters ► WebLogic JMS ► Biztalk ► TIBCO Rendezvous ► MQe ► Multicast ► Tuxedo ► FTP ► TIBCO EMS JMS ► COBOL Copybook ► HIPAA ► EDI-FACT ► ACORD ► Real-time IP ► SonicMQ JMS ► SWIFT ► FIX ► ebXML ► EDI-X.12 ► MQTT ► AL3 ► Word/Excel/PDF ► Custom Formats ► HL7
Message logging	► Provides prebuilt mediations for message logging	► Supports message logging
Message transformation	► Protocol transformation between HTTP, JMS, and IIOP ► Supports transformation of XML, SOAP JMS message data format (many more if used with adapters) ► Provides prebuilt mediations for XML transformation	► Protocol transformation between HTTP and JMS ► Custom transformation logic can be implemented in Java, ESQL, or XSLT ► Supports transformation between any protocols available as input our output nodes (see Table 7-1 on page 218)
Message routing	► Content and transport/protocol-based routing ► Supports through custom-built mediations using Java and the IBM SOA programming model (SCA and SDO) ► Provides prebuilt mediations for message routing	► Content and transport/protocol based routing ► Custom routing logic can be implemented in Java or ESQL

	WebSphere ESB	WebSphere Message Broker
Event-driven processing	► Supports event-driven processing by leverage adapters for capture and dissemination of business events	► Supports complex event processing (processing of events formed by several earlier ones)

To summarize:

► WebSphere ESB

Building an ESB that is based entirely on WebSphere ESB is an option when Web services support is critical and the service provider and consumer environment is predominantly built on open standards. WebSphere ESB is most suitable for environments that are based on Web services standards and provides facilities to integrate services that are offered via enterprise application integration messaging and other sources. However, if integration with non-Web service standards-based services is a major requirement then WebSphere ESB may not be the right choice.

► WebSphere Message Broker

WebSphere Message Broker is considered to be suitable where advanced ESB functionality is required. WebSphere Message Broker is an option when Web services support is not critical and quality-of-service requirements demand the use of mature middleware. WebSphere Message Broker can support all the ESB capabilities that WebSphere ESB does but is not limited to open standards. However, in comparison with WebSphere ESB, it lacks the sophistication of Web services support that might be required in an ESB implementation, which makes extensive use of these standards.

2.7 WebSphere Process Server

WebSphere Process Server is built on WebSphere ESB, thus providing it with the mediation functionality of WebSphere ESB and the qualities of service that WebSphere Application Server provides (for example, clustering, failover, scalability, and security). To this, WebSphere Process Server adds the ability to build business processes that orchestrate multiple services to achieve a business goal.

The WebSphere Process Server architectural model consists of the three layers shown in Figure 2-6 on page 41.
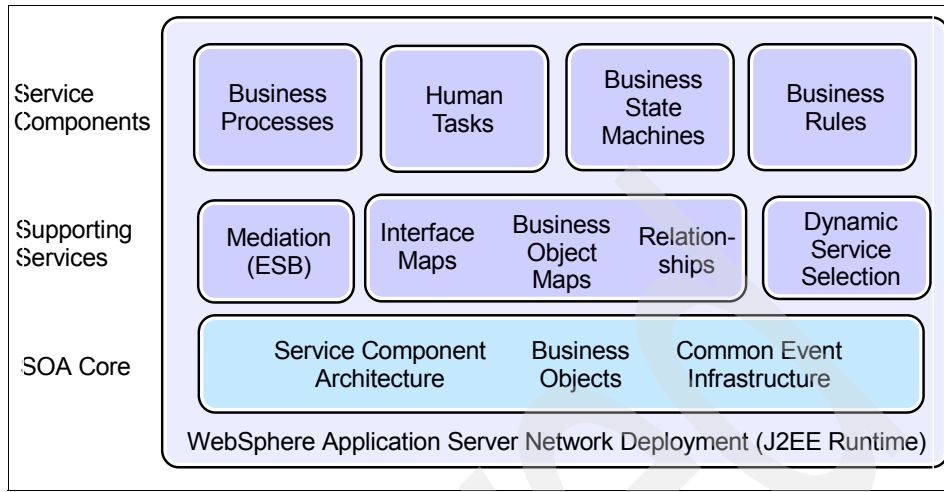
Figure 2-6   Architectural model of WebSphere Process Server

Above the infrastructure provided by WebSphere Application Server, WebSphere Process Server implements a layer called the SOA Core that includes the following:

▶ Service Component Architecture (SCA)

Using SCA, every integration component is described through an interface. These services can then be assembled in a Component Assembly editor, thus enabling a very flexible and encapsulated solution.

▶ Business objects

Business objects are the universal data description. They are used as data going in and out of services and are based on the Service Data Object (SDO) standard. SCA bindings contain the physical description of components. Services can be accessed as Java objects (POJOs), EJBs, Web services, JMS messages, and adapters.

▶ Common Event Infrastructure

The Common Event Infrastructure is the foundation for monitoring applications. IBM uses this infrastructure throughout its product portfolio, and monitoring products from Tivoli as well as WebSphere (WebSphere Business Monitor) exploit it. The event definition (Common Business Event, CBE) is being standardized through the OASIS standards body so that other companies as well as clients can use the same infrastructure to monitor their environment.

On top of this SOA Core layer lie the service components and supporting services layers. WebSphere Process Server implements a number of

components and services that can be used in an integration solution. In the service components layer you will find the following:

► Business processes

The business process component in WebSphere Process Server implements a WS-BPEL compliant process engine. Clients can develop and deploy business processes with support for long-running and short-running business processes and a robust compensation model in a highly scalable infrastructure. WS-BPEL models can be created in WebSphere Integration Developer or imported from a business model that has been created in WebSphere Business Modeler.

► Human tasks

Human tasks in the WebSphere Process Server are standalone components that can be used to assign work to employees or to invoke any other service. Additionally, the Human Task Manager supports the ad-hoc creation and tracking of tasks. Existing LDAP directories (as well as operating system repositories and the WebSphere user registry) can be used to access staff information. Of course, WebSphere Process Server supports multi-level escalation for human tasks including e-mail notification.

The WebSphere Process Server also includes an extensible Web client that can be used to work with tasks or processes. This Web client is built based on a set of reusable Java Server Faces (JSF) components that can also be used to create custom clients or embed human task functionality into other Web applications.

► Business state machines

A business state machine provides another way of modeling a business process. This enables businesses to represent their business processes based on states and events, which are sometimes easier to model than a graph-oriented business process model. One example would be an ordering process where the order can be cancelled or modified at any time during the order process.

► Business rules

Business rules are a means of implementing and enforcing business policy through externalization of business function. This enables dynamic changes of a business process for a more responsive businesses environment. Business rule authoring is supported within an Eclipse-based desktop tool. The WebSphere Process Server also includes a Web-based runtime tool for the business analyst so that business rules can be updated as business needs dictate without affecting other SCA services.

These components can use the features of a number of supporting services in the WebSphere Process Server. Most of these can be classified as some form of

transformation, which is not surprising. There are a number of transformation challenges when connecting components and external services, each of which is being addressed by a component of WebSphere Process Server:

► Interface maps

Very often interfaces of existing components match semantically but not syntactically (for example, updateCustomer versus updateCustomerInDB2). This is especially true for already existing components and services that need to be accessed. Interface maps allow the invocation of these components by translating these calls. Additionally, business object maps can be used to translate the actual business object parameters of a service invocation.

► Business object maps

A business object map is used to translate one type of business object into another type of business object. These maps can be used in a variety of ways, for example, in an interface map to convert one type of parameter data into another.

► Relationships

In business integration scenarios it is often necessary to access the same data (for example, client records) in various backend systems (for example, an ERP system and a CRM system). A common problem for keeping business objects in sync is that different backend systems use different keys to represent the same objects. The relationship service in the WebSphere Process Server can be used to establish relationship instances between objects in these disparate backend systems. These relationships are typically accessed from a business object map when translating one business object format into another.

► Dynamic service selection

A selector component allows dynamic selection and invocation of different services, which all share the same interface. For example, a customer support process could use different human tasks implementations during different times of day. This would enable routing of work to different support centers (Americas, Europe, Asia-Pacific) based on the time of day. Just like for the business rule component, WebSphere Process Server offers a Web-based interface to enable dynamic updates to the selection criteria and target services.

► Mediation

The mediation component can act on messages flowing between the requestor and the service. For example, it can transform messages from the format used by the requestor to the format required by the service. Other typical actions include routing based on message content and protocol transformation.

The primary development tool for the WebSphere Process Server is WebSphere Integration Developer. This is the same tool used for WebSphere ESB development tasks.

You can find more information about IBM WebSphere Process Server V6 at:

► WebSphere Process Server home page:

  http://www.ibm.com/software/integration/wps/

► *Technical Overview of WebSphere Process Server and WebSphere Integration Developer*, REDP-4041

► *Getting Started with WebSphere Process Integration V6*, SG24-7130

3

Runtime topology selection

This chapter shows how IBM messaging products can be introduced into an existing or new solution to enable SOA technology. It starts with the basics and proceeds to include high availability and scalability considerations.

45

3.1 Getting started

This section shows how WebSphere products can be applied to form topologies able to evolve systems based on direct application connections to decoupled solutions following the SOA principles.

The move to an SOA can be considered as stages of complexity and functionality. The first stage introduces messaging to directly connected applications, thus providing an initial move towards decoupled applications and flexibility. In environments where no messaging exists, this is a simple way to start with connections of similar messaging applications. The second stage further enhances decoupling by introducing an enterprise service bus providing mediation functionality. In the third stage, service integration and orchestration functionality are introduced to provide a configurable environment for reusable services.

3.1.1 Starting with simple messaging connections

In the most basic form of connectivity, two applications connect directly to each other. Each is aware of the other application's location and connectivity requirements. Special needs, such as data transformation, are handled within the applications themselves.

If this is your starting point, you can begin moving toward decoupling the service consumers and service providers by introducing middleware that provides connectivity logic for the applications. The applications only need to be aware of the location of the connectivity infrastructure, rather than the location of each application they connect to. In this instance, each application is still aware of any transformation or logic required to use the data from the other application.

In terms of a messaging environment, this first step would equate to two messaging applications putting and getting messages from a queue managed by WebSphere MQ, the WebSphere Application Server service integration bus, or a combination of the two. An application can place a message directly onto a WebSphere MQ queue or onto a queue hosted on the service integration bus.

When the service provider and service consumer are both WebSphere Application Server applications, the default messaging provider and service integration bus can provide this connectivity. Two applications within the same WebSphere Application Server cell can communicate directly through the bus. If there are multiple cells involved, each with a bus, the buses can be linked together. In the event that one application is a WebSphere Application Server application and the other is capable of connecting to WebSphere MQ, the bus can be linked to WebSphere MQ to provide connectivity. This connectivity is shown in Figure 3-1 on page 47.
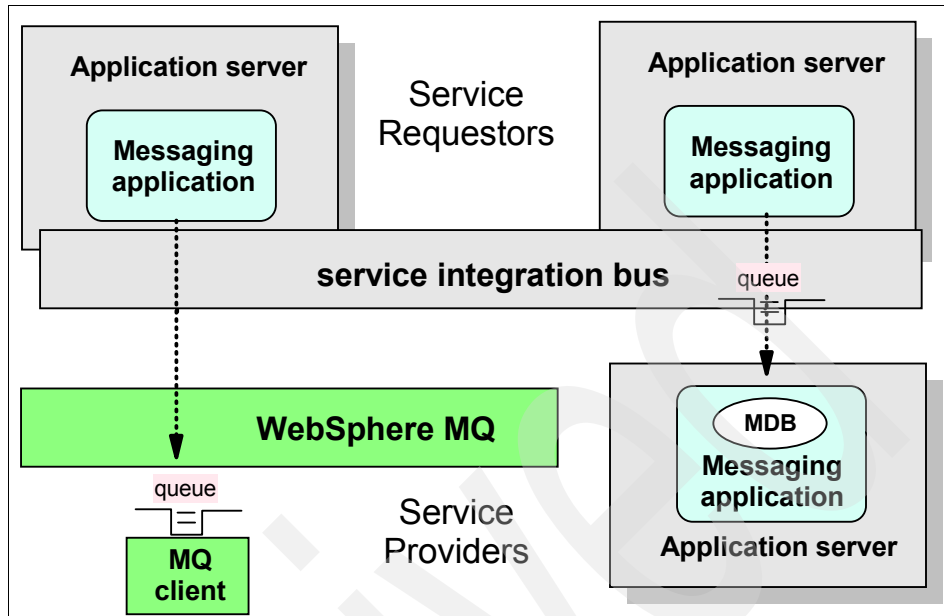
Figure 3-1   Direct connections

With the service integration bus or WebSphere MQ, you can integrate applications across a broad set of platforms and application environments. However, as your applications become more complex, more connections must be defined and maintained.

3.1.2 Adding an ESB for enhanced connectivity

The next step in moving toward an SOA environment is to start moving the mediation-type logic from the applications to a central location. Basic mediation-type logic would include things like:

► Routing based on the content of a message

► Augmenting a message

► Transforming a message from one format to another

► Converting transport protocols to allow communication between disparate messaging products

Of course, more complex actions are also possible, such as decomposing a single message into multiple messages, sending those messages to multiple service providers, and then recomposing the responses into a single response.

In an SOA environment, these mediation actions are handled by an enterprise service bus. Figure 3-2 shows a topology using an enterprise service bus to enhance flexibility and connectivity.
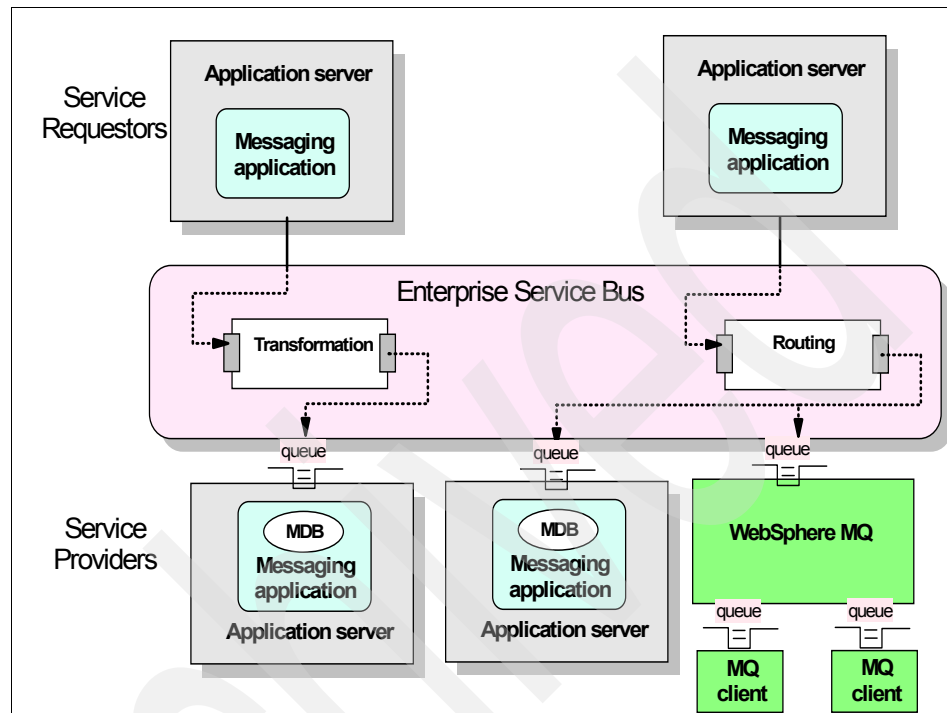
Figure 3-2   Application mediation

The choice of implementation depends on the current requirements for the interaction between the service consumers and service providers, and with consideration of future needs. For information that will help you determine the ESB product that is right for your situation, see "ESB product comparison" on page 37.

WebSphere ESB has the capability to participate in a WebSphere MQ network as a queue manager, making it possible for WebSphere ESB and WebSphere Message Broker to coexist in a solution. This combination can give you access to the best features of both products.

For example, an existing messaging environment using WebSphere MQ and WebSphere Message Broker that would like to extend their infrastructure to use Web services would benefit by adding WebSphere ESB to provide the enterprise service bus functionality for the Web services. A second example would be a situation where WebSphere Message Broker is being used as a central hub for

J2EE applications running in WebSphere Application Server. The use of WebSphere ESB gives you the ability to do mediations within the application server environment using a single technology stack.

3.1.3 Adding a business process engine for service orchestration

Once you have the enhanced connectivity of the ESB in place, you can move forward by composing services into processes that span people, workflows, applications, systems, platforms, and architectures. An SOA requires the ability to develop and modify applications dynamically, creating the need for infrastructure on the top of the SOA stack that provides this functionality.

The WebSphere Process Server is designed to simplify the integration of business services into processes by providing not only graphical tools for designing process flows but also by providing a technology-neutral way of describing interfaces using open standards for enabling the dynamic exchange of module implementations.

The WebSphere Process Server utilizes SCA and business objects based on SDO to provide an open-standards based approach for describing the interfaces for all integration artifacts. Integration artifacts can be processes, business rules, human tasks, and so on. This approach creates a very flexible environment where one module can be exchanged easily with another as long as the interface is the same.

The WebSphere Process Server implements a Web Services Business Process Execution Language (WS-BPEL) compliant process engine that supports long-running and short-running business processes as well as a compensation model for rollback operations. WS-BPEL models can be created in WebSphere Integration Developer or imported from a business model created in WebSphere Business Modeler.

Figure 3-3 on page 50 shows the orchestration of business services to process flows within the process server. The different process steps are mapped to service calls for providers.
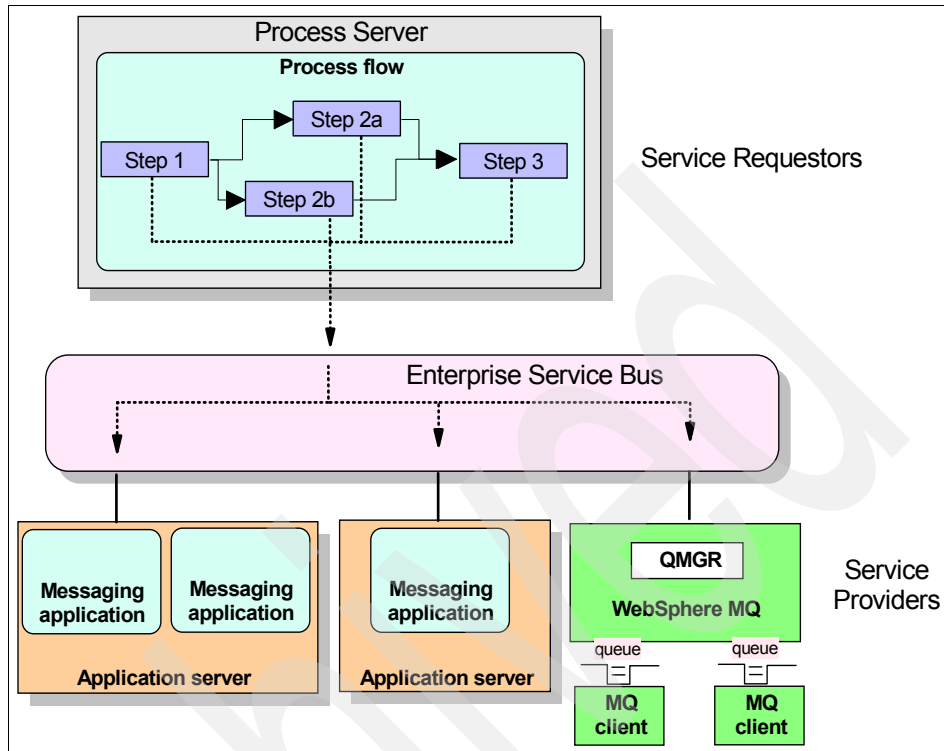
Figure 3-3   Service composition

3.2 Advanced topologies

A messaging infrastructure is suited for and often intended to act as the backbone of a service-oriented environment, not only for single applications but for whole enterprises. It is therefore inevitable that you need to consider and design for high availability, failover, and load balancing.

There are two basic methods for implementing high availability:

► Hardware clustering

Hardware clustering, also referred to as operating system clustering or shared disk, is a hardware-based approach of turning multiple servers into a cluster. Within the cluster there exists a controlling server that monitors the cluster and performs administrative tasks such as deciding when failover is necessary and assigning the load of a failed node to a functioning server. A hardware cluster may be set up in an active-passive mode or an active-active mode. Active-passive (cold-standby) means that a server is reserved for

failover duties and does not normally run applications. Active-active (mutual takeover) means that each server in the cluster runs its own applications but each has some resources left that could be used to perform failover for the others.

An example for this approach is High Availability Cluster Multi-Processing (HACMP™), an IBM clustering solution for IP-based cluster failover.

► Software clustering

Software clustering, also referred to as application clustering, is a software-based approach of turning multiple servers into a cluster. Clustering software is installed on each of the servers in the group, while each server maintains the same information and collectively they perform the same administrative tasks such as load balancing, determining node failures, and assigning failover duties.

This approach is more flexible and scalable than the hardware clustering approach in the sense that it does not require special hardware. Servers can be added and removed from clusters dynamically.

Both clustering methods can effectively be used to implement high availability for messaging systems.

3.2.1 WebSphere MQ

Both hardware and software clustering can be used for failover of WebSphere MQ. This section discusses both.

Hardware clustering

Hardware clustering in a WebSphere MQ context means that a WebSphere MQ installation exists on a primary and a secondary server. Both servers have a queue manager configured to store data on a reliable storage medium. The storage has failover capabilities between the servers. This configuration is shown in figure Figure 3-4 on page 52.
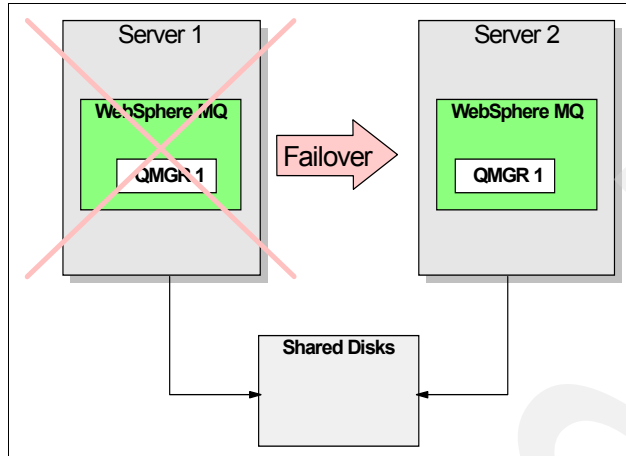
Figure 3-4   WebSphere MQ active-passive failover scenario

This approach is an active-passive scenario because the two queue managers cannot run concurrently and access the same data. The clustering software switches the reliable storage between the two servers if a failure is detected.

Queue manager clusters

WebSphere MQ supports the joining of single queue managers together in a queue manager cluster. Queue manager clusters allow multiple instances of the same service to be hosted through multiple queue managers.

Applications requesting a particular service can connect to any queue manager within the cluster. When applications make requests for the service, the queue manager to which they are connected automatically workload balances these requests across all available queue managers.

This allows a pool of machines to exist within the queue manager cluster, each hosting a queue manager and the application required to provide the service. This is especially useful in a distributed environment, where capacity is scaled to accommodate the current load through multiple servers, rather than one mainframe or high capacity server.

Queue managers can dynamically join or leave the queue manager cluster to cope with varying loads placed upon a particular service provided by a system. Configuration only needs to be performed on the queue manager joining or leaving the cluster, not the queue managers already within the cluster.

A queue manager can be a member of multiple queue manager clusters, allowing segregation of the components of a WebSphere MQ infrastructure based on the services provided or organizing into groups. These queue

managers can provide a bridge between queue manager clusters as well as bridging between a queue manager cluster and an existing WebSphere MQ infrastructure based on distributed channels.

If a queue manager fails, subsequent messages sent to the cluster are not routed to the failed queue manager. However, any persistent messages that have been already sent to the queue manager but not yet processed are marooned on this failed queue manager.

z/OS high availability options

WebSphere MQ for z/OS supports high availability features based on z/OS system architecture and capabilities. WebSphere MQ features specific to z/OS are described within this section.

Shared queues

WebSphere MQ shared queues depends on the coupling facilities provided by joining z/OS systems together in a sysplex and a shared DB2® database to allow queue managers to form a queue sharing group (QSG). Multiple queue managers that are members of a QSG have access to shared queues contained within that QSG. Each shared queue is available to all queue managers in the QSG, similar to hosting that queue locally.

Figure 3-5 shows the concept of a QSG. QMGR1 and QMGR2 both have access to the same shared queue Q1.
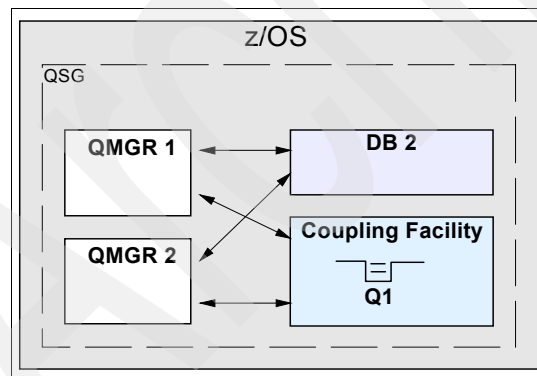

Figure 3-5   z/OS QSG

One application connected to a queue manager can put a message to a shared queue. A second application connected to a second queue manager within the QSG can get that message from the shared queue. Without utilizing the functionality of shared queues, this message would need to be transferred to a

queue hosted on the second queue manager by a distributed or cluster message channel before the second application could get the message.

Applying shared queues addresses the problem of message availability when a queue manager becomes unavailable while messages are on its queues. When a clustered queue manager fails, new requests are routed around that queue manager, but messages that exist on the failed queue manager cannot be accessed until the queue manager is made available again. However, if the queue manager is a member of a QSG, other queue managers within the QSG can access the messages in the queue, preventing those messages from being unavailable.

3.2.2 WebSphere Message Broker

WebSphere Message Broker benefits from WebSphere MQ high availability capabilities such as queue manager clustering and the z/OS shared queue functionality. To be able to lay out a high availability design for a WebSphere Message Broker application, the concepts of WebSphere MQ and the possibilities that they offer must be understood.

There are different levels in a WebSphere Message Broker environment that need to be considered in order to provide high availability.

First, WebSphere MQ provides the transport layer for WebSphere Message Broker, utilizing queue managers not only for administering services and resources but also for providing the connection point for the brokers. These queue managers should be configured for high availability through queue manager clustering, hardware clustering, and shared queues (for z/OS).

Also consider the case where a clustered queue manager does not recognize that a connected broker has failed and is still served messages to be processed. These messages are not processed until the failed broker is up and running again. One possible solution would be to provide more than one broker instance per queue manager. Another possible solution is to set up a monitor for the broker and input queue that triggers an alert in the event of a failure.

Second it must be guaranteed that the broker functionality is highly available. This can be achieved by providing multiple message broker instances serving the same logical hub. The different brokers could even reside on different machines. Each broker instance could either be mapped to a single queue manager or to a queue manager cluster.

Figure 3-6 on page 55 shows a topology using two broker instances together with z/OS shared queues and queue manager clustering. Note that Node C is not

included in the queue sharing group; therefore, it is not required to be a z/OS node.
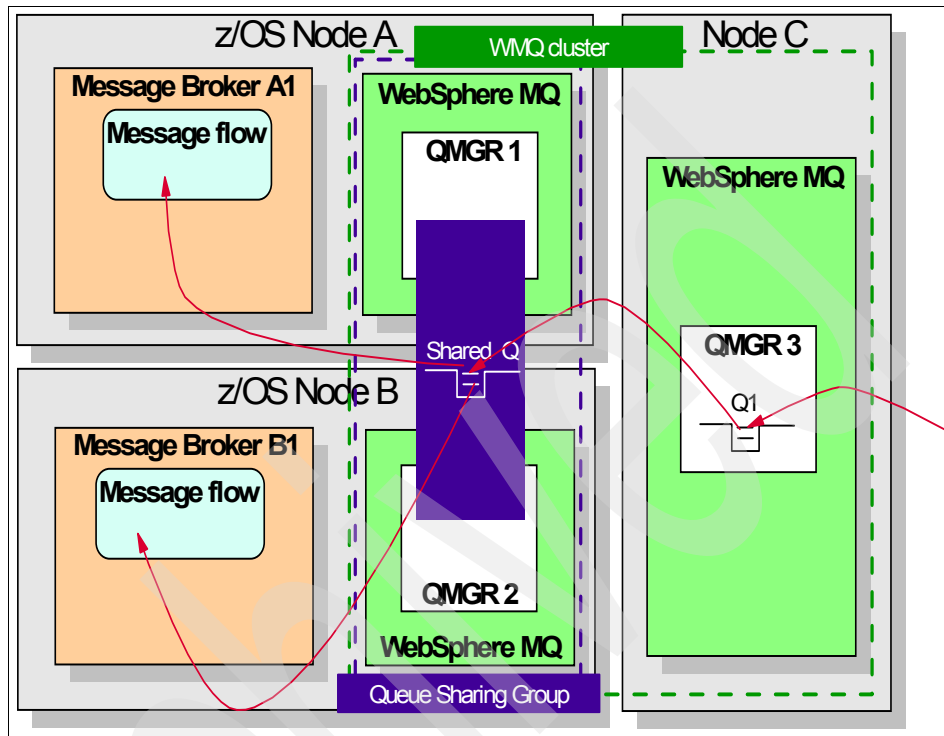
Figure 3-6   Multiple message brokers using MQ clustering and shared queue

WebSphere Message Broker supports the cloning as well as tailoring of execution groups over different broker instances. It is possible for all message broker instances to have identical execution groups and message flows deployed so that each broker can process any message. An alternative is to deploy specific message flows to each broker, which enables the distribution of specific processing to specific brokers.

3.2.3 WebSphere ESB and WebSphere Process Server

WebSphere ESB and WebSphere Process Server are built on WebSphere Application Server Network Deployment, and thus inherit its high availability capabilities.

Application server clusters

WebSphere Application Server Network Deployment provides the capability to cluster application server instances for load balancing and high availability.

Application changes such as installation, updates, or deletes are automatically distributed to all members in the cluster. The rollout update option allows you to update an application and then restart the servers on each node, one node at a time, providing continuous availability of the application.

The following failover capabilities are provided by application server clustering:

► Web container failover

   The HTTP server plug-in in the Web server is aware of the configuration of all Web containers and can route around a failed Web container in a cluster. Sessions can be persisted to a database or in-memory using data replication services.

► EJB container failover

   Client code and the ORB plug-in can route to the next EJB container in the cluster.

The members of a cluster can be located on a single machine (vertical cluster), across multiple machines (horizontal cluster), or on a combination of the two. Vertical scaling allows the machine's processing power to be more efficiently allocated. Horizontal scaling provides failover in case a machine becomes unavailable. You can combine the two to reap the benefits of both, as shown in Figure 3-7.
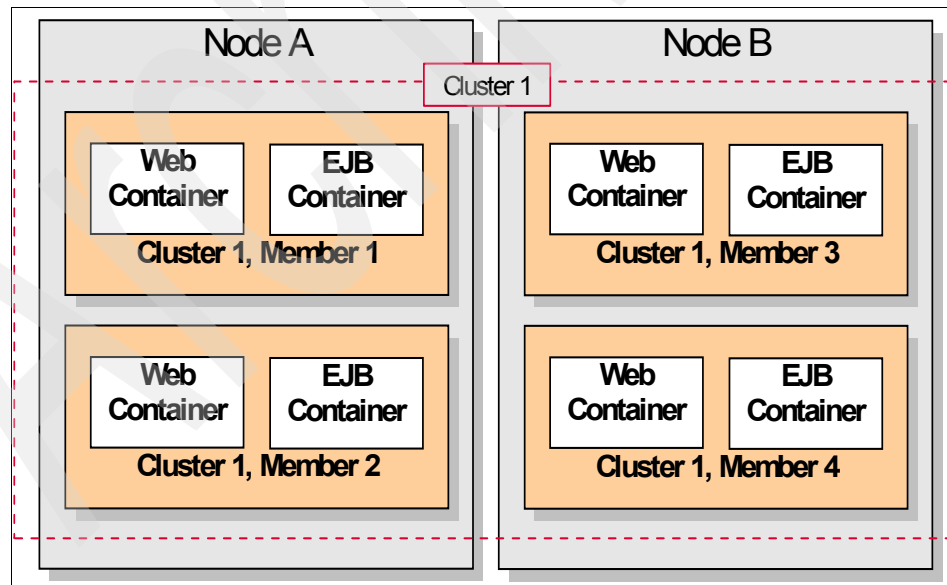
Figure 3-7   Combination of vertical and horizontal application server scaling

Normal workload management and failover would apply to each cluster. In the event that Node A becomes unavailable, the application servers on Node B would still be available to process incoming requests.

High Availability Manager

WebSphere Application Server V6 introduces a new feature for advanced failover called the High Availability Manager (HAManager). The concept is based on high availability domains called core groups consisting of a set of WebSphere processes that provide high availability functions to each other.

Core groups are not based on application server clustering but implement their own logical resource grouping that can contain stand-alone servers, cluster members, node agents, or the deployment manager. Grouping these components in core groups provides hot standby and peer failover for critical services such as workload management routing, PMI aggregation, JMS messaging, transaction manager, and so on.

As depicted in Figure 3-8 on page 58, each application server process runs an HAManager component and shares information through the underlying Distribution and Consistency Services (DCS) communication infrastructure such that no single point of failure will exist in the topology. Every member in a WebSphere cluster knows where singleton services are running.
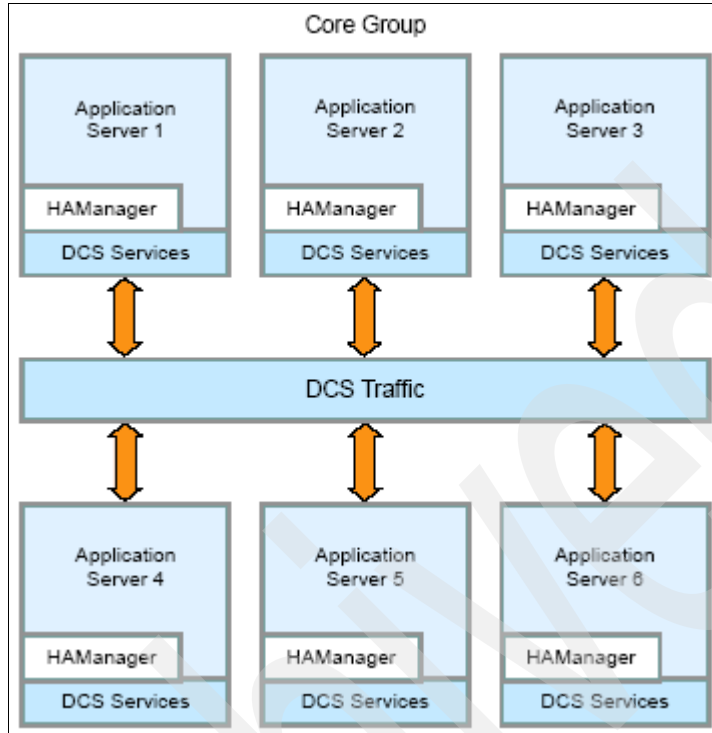
Figure 3-8   Core group

One or more members of the core group can act as a high availability coordinator, managing the high availability activities within the core group processes. If a high availability coordinator server fails, another server in the core group takes over the duties of that coordinator. High availability policies define how the failover occurs. Workload management information is shared between core members and failover of critical services is done among them in a peer-to-peer fashion. Little configuration is necessary and, in many cases, this function works with the defaults that are created automatically as you create the processes.

HAManager leverages the latest storage technologies, such as IBM Storage Area Network File System (SAN FS) to provide fast recovery time of two-phase transactions. In addition, it adds the possibility of hot standby support to high availability solutions using conventional failover middleware such as IBM High Availability Clustered Multi-Processing (HACMP) or Tivoli System Automation (TSA).

3.2.4 Application server and queue manager cluster

The clustering capabilities of WebSphere Application Server Network Deployment and WebSphere MQ can be used to design a solution that provides high availability across different product lines.

Figure 3-9 shows a sample topology that combines application server clustering with queue manager clustering. Both cluster types are vertically and horizontally scaled. Note that as long as one node is available the solution is able to deliver its functionality. Each queue manager serves a minimum of two application servers. In the event that an application or application server fails, the messages can be processed by another server in the cluster. Remember that a queue manager does not recognize whether a connected client is available and is served with messages according to the load balancing policy in the queue manager cluster.
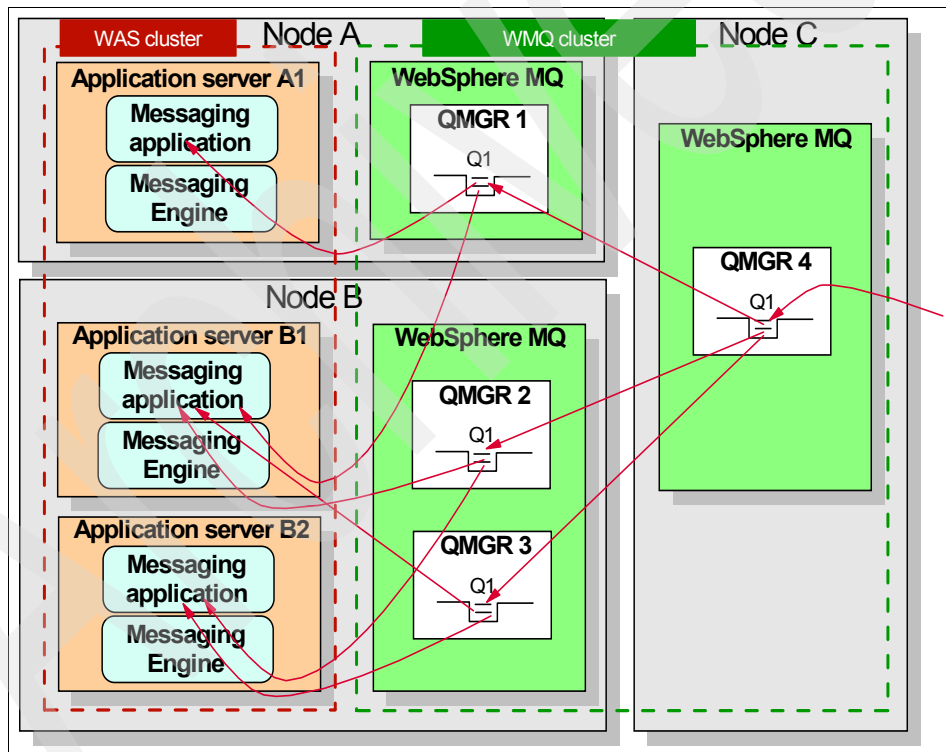
Figure 3-9   Application server clustering with queue manager clustering

3.3 End-to-end scenario

To illustrate an end-to-end scenario, consider the following travel bureau application. The travel bureau company services requests from clients wishing to book various travel services through their Web site. In addition to airline ticket purchases, these services include car rental, hotel reservations, and visa applications.

The travel bureau has a Web application that runs in WebSphere Application Server. Clients access this application through a Web site to book the various travel services. The scenarios in this book deal with the airline ticket booking service.

The travel bureau is supplied tickets by four airline companies. Each airline company has its own system and programming language:

- ► Airline A: COBOL
- ► Airline B: J2EE (WebSphere Application Server)
- ► Airline C: MQ (Pub/Sub)
- ► Airline D: SAP Application

To integrate the travel bureau systems with these suppliers, mediation services are required. We can add these mediation services using WebSphere ESB and/or WebSphere Message Broker as an ESB. In this scenario, both products are used.
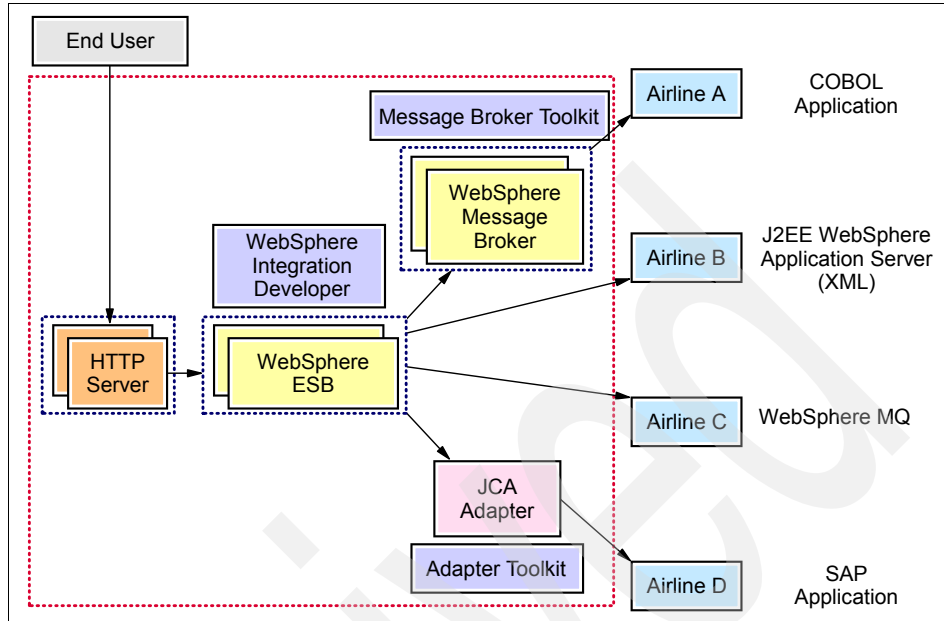
Figure 3-10   End-to-end scenario

The scenario is:

▶ When a client requests a flight reservation through the Web page, the WebSphere ESB converts the incoming BLOB data to XML format.

▶ If Airline A is selected, the request is put on a queue to be processed by a message flow in WebSphere Message Broker. The message flow converts the XML data to COBOL format using the defined COBOL message sets and message definitions.

▶ If Airline B is selected, the XML data is put on a queue in WebSphere ESB for delivery to the J2EE application running on a WebSphere Application Server.

▶ If Airline C is selected, the XML data is put on a JMS destination. The destination is an alias for a queue on WebSphere MQ.

▶ If Airline D is selected, the XML data is transformed to SAP format through a JCA adapter, then delivered to the target application through the existing network environment.

The Web servers, application servers, and broker servers are clustered for high availability.

4

Application design

This chapter provides an introduction to messaging and messaging design. It describes the different messaging models, styles, and patterns, and shows how they can effectively be applied to implement service consumers and service providers. Next, it describes an approach to develop service-oriented architectures and provides guidelines and best-practices to meet principles required for a service-oriented architecture.

The intent of this chapter is to show the capabilities as well as possible issues that may arise by applying messaging for a service-oriented architecture.

Attention: The term *consumer* is used both when discussing messaging (a message consumer) and services (a service consumer). Note that a service provider can in fact be a message consumer as well. The intent should be clear within the context of the discussion.

The following pairs relate to each other:

► Message producer <--> message consumer
► Service consumer <--> service provider

4.1 Introduction to messaging

A message is a collection of data sent by one program and intended for another one. A message consists of the message payload and the message header containing technical fields like message ID, correlation ID, and reply address. A message queue, known simply as a queue, is a named destination to which messages can be sent. Messages accumulate on queues until they are retrieved by programs that service those queues.

Messaging applications communicate by sending each other data in messages rather than calling each other directly. Applications can open a queue, put messages on it, get messages from it, and close the queue. They can also set and inquire about the attributes of queues.

To send a message, the application places the message on a queue. As the message traverses the messaging network to its destination it is stored at intermediate nodes until the system is ready to forward it to the next node. At the final destination, the message is stored until the receiver is ready to read it. Retrieval of the message by the receiver is done independently of the sender.

The use of messaging provides features that function well in an SOA environment:

► Communication between applications is done indirectly through the messaging middleware versus through direct connections. This reduces the complexity of the applications, adds flexibility in the location of the messaging partners, and helps accommodate differences in platforms.

► The message sender and receiver act independently of each other. Messages are held until the receiver is available to handle them.

► Messaging lends itself to the use of modularity in program design. Instead of a single large program performing all the parts of a job sequentially, you can spread the job over several smaller independent programs. The requesting program sends messages to each of the separate programs, asking them to perform their function and the results are sent back as one or more messages.

► Programs can be controlled according to the state of queues. For example, you can arrange for a program to start as soon as a message arrives on a queue, or you can specify that the program does not start until there are, for example, ten messages above a certain priority on the queue, or ten messages of any priority on the queue.

► A program can assign a priority to a message when it puts the message on a queue. This determines the position in the queue at which the new message is added. Programs can get messages from a queue either in the order in which the messages appear in the queue, or by getting a specific message.

- ► Data integrity is provided by units of work. The synchronization of the start and end of units of work is supported as an option on each get or put, allowing the results of the unit of work to be committed or rolled back.

4.2 Messaging models

There are two basic messaging models in common use, the point-to-point and the publish-subscribe (pub/sub). A point-to-point model is applied when a message producer needs to send a message to exactly one message consumer. A publish-subscribe model is applied when a producer (publisher) needs to send a message to one or more consumers (subscribers) that are interested in the message.

4.2.1 Point-to-point

In the point-to-point messaging model a message is intended to be consumed by one consumer at most. The messaging infrastructure makes sure that a message is retained until it is either consumed or expired.

Using a point-to-point messaging model, the messaging infrastructure ensures that only one message consumer gets any given message. If the queue has multiple consumers attached only one of them can successfully consume a particular message. Attaching many consumers to the same queue can make the consuming and the processing of messages highly scalable by providing failover and load balancing. For example, to increase the throughput of a system you would just need to attach another consumer to the queue. Or assuming one of the consumers crashes, there would still be others processing the incoming messages.

Figure 4-1 shows a conceptual overview of the point-to-point messaging model.
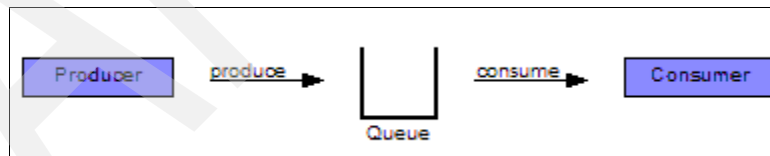

Figure 4-1   Point-to-point messaging

An example where a point-to-point messaging model would be appropriate is an online store. Each message represents one order; therefore, each order must be processed exactly once.

4.2.2 Publish-subscribe

In the publish-subscribe messaging model a message is intended for each consumer who is interested in it. Each consumer can consume a specific message not more than once. The messaging infrastructure makes sure that a message is retained until it is either consumed or expired.

Within the publish-subscribe model the message producer is called the publisher, the message consumer is called the subscriber, and the queue is called a topic.

Using a publish-subscribe messaging model, messages are broadcast from a publisher to multiple subscribers. The publisher and subscribers are related by a topic that represents the category of data in which the subscribers are interested. Each subscriber that subscribes to a topic receives a copy of every message that is published to that topic.

Figure 4-2 shows a conceptual overview of the publish-subscribe messaging model.
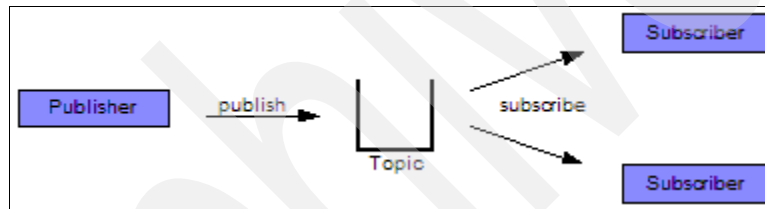
Figure 4-2   Publish-subscribe messaging

The functionality of delivering a message just once to each subscriber is implemented using a private queue for each subscriber. A message published to a topic is automatically placed as a copy on the private queue of each subscriber. Typically there is some latency coming from the fact that messages observed by subscribers depend on the capability of the underlying infrastructure to propagate them.

In the publish-subscribe model publishers and subscribers can be added dynamically, thus allowing the system to grow or shrink dynamically.

Durable and non-durable subscriptions

There are two types of subscriptions, durable and non-durable. The difference lies in whether a subscribing application wants to receive messages published while the application is not running.

While a durable subscriber is disconnected from the topic, the messages published for the topic are stored. When the subscriber reconnects, the messages that the subscriber otherwise would have missed are delivered. A durable subscriber has to explicitly unsubscribe from a topic to cancel its subscription.

While a non-durable subscriber is disconnected from the topic, the subscriber does not receive any messages published to the topic. A non-durable subscriber is unsubscribed when it disconnects and re-subscribed when it reconnects.

Topic hierarchies

When publishing information, the publisher specifies an identifier that defines the topic for which a message is destined. Topics may be defined within a hierarchy. Subscribing to a topic in the hierarchy that contains subtopics allows the subscriber to receive all messages published to the topic and its subtopics.

Figure 4-3 shows an example of a hierarchically structured topic.
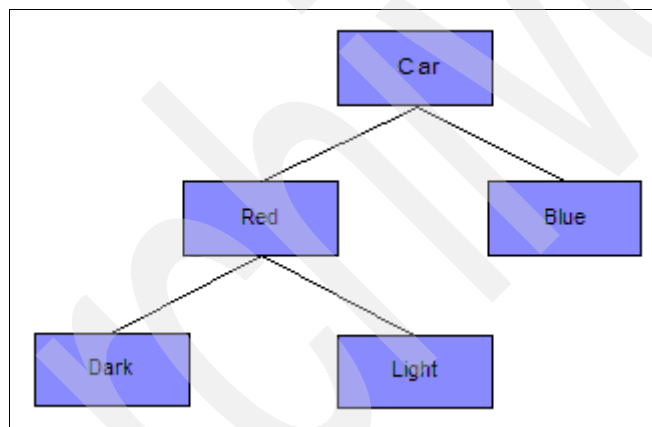

Figure 4-3   Example of topic hierarchy

If a subscriber is interested in dark and light red cars, it only needs to subscribe to the red topic.

4.2.3 Point-to-point versus publish-subscribe

Table 4-1 on page 68 shows a comparison of the two messaging models, point-to-point and publish-subscribe.

Table 4-1   Point-to-point versus publish-subscribe

Point-to-point	Publish-subscribe
► Exactly one consumer consumes and processes a message. ► More natural fit with request-reply pattern. ► Less complex than publish-subscribe (configuration and administration). ► Faster than publish-subscribe.	► Natural fit where a message needs to be sent to multiple providers ► Works well with changing delivery requirements as new subscribers can be added easily ► Additional decoupling between producer and consumer as the publisher does not know about subscribers

Note: JMS 1.1 unifies the point-to-point and the publish-subscribe domain by providing one common interface for both messaging models. The differentiation is made with the destination settings of the JMS resource configuration.

4.3 Messaging styles

Messaging can be applied in either an asynchronous or in a pseudo-synchronous way. The distinction is made based on the whether the same consumer thread that issued a request needs to handle a possible reply.

4.3.1 Asynchronous communication

Using asynchronous communication, the consumer thread follows a fire-and-forget approach. After sending a request the thread is not blocked even if a reply is expected.

Figure 4-4 shows the behavior of the consumer and provider in case of an asynchronous communication style.
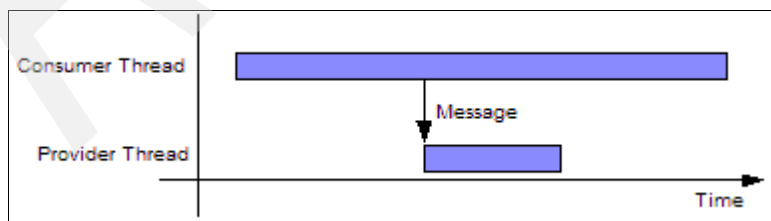
Figure 4-4   Thread behavior using asynchronous communication

There are two pattern variants for asynchronous communication:

► Fire-and-forget
► Request-reply

4.3.2 Pseudo-synchronous communication

Using pseudo-synchronous communication, the consumer thread follows a synchronous request-reply approach. After sending a request it is blocked until the expected reply from the provider arrives.

Figure 4-5 shows the behavior of consumer and provider in case of a pseudo-synchronous communication call.
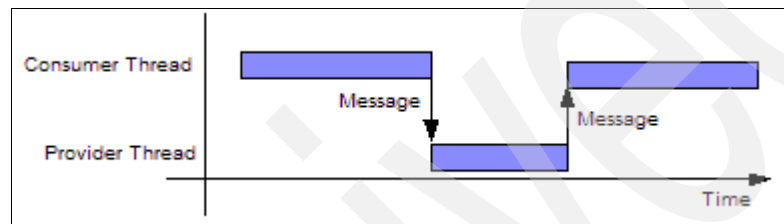
Figure 4-5   Thread behavior using pseudo-synchronous communication

The behavior of the pseudo-synchronous communication style is in fact synchronous, but because the underlying medium is asynchronous the communication style is called pseudo-synchronous.

Unlike asynchronous communication, the pseudo-synchronous style just supports one variant: Request-reply.

Asynchronous versus pseudo-synchronous communication

Table 4-2 on page 70 shows a comparison between the two communication styles.

Table 4-2   Asynchronous versus pseudo-synchronous communication

Asynchronous	Pseudo-synchronous
► Natural fit for purely fire-and-forget driven systems. ► Loose coupling because of the fully asynchronous style. ► Supports fire-and-forget as well as request-reply. ► If a consumer crashes while waiting for a reply, on restart it can continue waiting for the reply; thus the reply is not lost. ► Resource efficient because of the fully asynchronous nature.	► Natural fit where a synchronous call would be required. ► Tighter coupling because of the quasi synchronous nature. ► Supports just request-reply. ► If a consumer crashes while blocking until the reply arrives, on restart it has no way of reconnecting to the invocation in progress and the response is lost. The consumer must repeat the invocation. ► More resource inefficient as there is a blocked thread on consumer side.

4.4 Messaging patterns

Messaging products are inherently asynchronous in that no fundamental time dependency between the message production and the message consumption exists. Applying appropriate patterns can turn communication over an asynchronous infrastructure in a quasi synchronous manner.

There are two basic messaging patterns and variants of them. The fire-and-forget pattern supports one-way communication while the request-reply pattern provides functionality for a two-way communication between consumer and provider.

4.4.1 Fire-and-forget

Fire-and-forget is the simplest of the messaging patterns. It supports one-way communication from a service consumer to a provider. It can be thought of as a request for which a reply is neither expected nor needed. The fire-and-forget pattern exists just for asynchronous communication. The consumer does not expect any reply from the provider; thus, after sending the request it immediately continues with processing. The fire-and-forget pattern provides a high level of decoupling between consumer and provider.

Figure 4-6 on page 71 shows a sequence diagram of the fire-and-forget pattern. Thread A continues processing directly after sending the request.
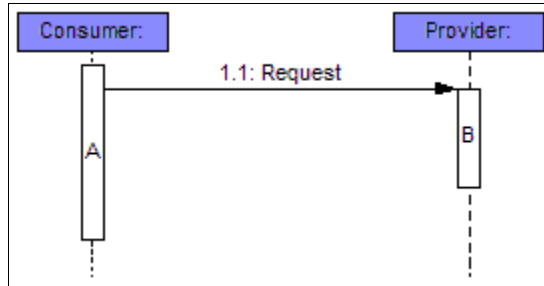
Figure 4-6 Sequence diagram fire-and-forget

4.4.2 Request-reply

Sometimes it is not reasonable to send a message without expecting a reply. Although there are advantages to decoupling consumer and provider, there are scenarios in which confirmation or results of the remote processing is needed in the form of a reply.

The request-reply pattern supports two-way communication between a consumer and a provider. Based on the request from the consumer, the provider sends back a reply. The reply may either contain an acknowledge for the arrival of the request or a result based on the provider's request processing.

The request-reply pattern exists for both asynchronous and pseudo-synchronous communication. The difference is based on the blocked or running consumer thread during the request processing on the provider side.

The request-reply pattern provides a tighter coupling between consumer and provider. Additional logic is needed to correlate request and reply, to handle reply delays or even failures in receiving replies.

Figure 4-7 on page 72 shows two sequence diagrams of the request-reply pattern. With asynchronous communication thread A continues processing after triggering thread B. The reply of thread B is handled on the consumer side by a third thread C. With pseudo-synchronous communication thread A blocks its processing after triggering thread B until it receives the reply.

Figure 4-7   Sequence diagram request-reply

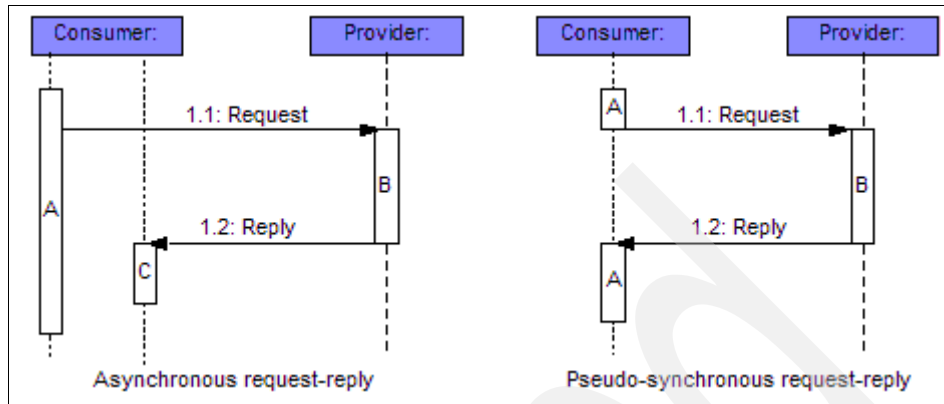Consider the case where a data store is not only used in an interactive manner from a Web interface and also in a batch mode for data replication purposes.

With the Web interface, where the user will be automatically forwarded to the inquiry result page when the service result is available, the pseudo-synchronous solution is suitable. Since polling from the Web interface is not an option, the technical implementation requires a blocked consumer thread until the reply arrives.

For batch mode the asynchronous request-reply scenario is suitable as there is no need to wait for the reply and therefore no reason to block a thread.

Request-reply design considerations

To support the asynchronous or pseudo-synchronous request-reply behavior, a couple of issues must be addressed that do not exist when replies are made synchronously.

Request and reply correlation

Because of the asynchronous nature of messaging there is no link between request and reply. A service consumer sends a request and sometime later the reply arrives. But what if the consumer made multiple requests and no longer knows which reply matches what request?

The consumer could ensure that only one reply is due at a time, thus establishing an implicit correlation between the request and the reply. Unfortunately, this scenario is rarely applicable, as it greatly slows down the message processing. Relying on the message order does not solve the problem either. Since the duration of message processing may vary, the order of the requests may not relate to the order of the replies.

There are two common approaches:

1. Correlation identifier

   A correlation identifier is a unique ID that indicates to which request message a reply is related. In addition to the payload, a message contains a field for the message ID and a field for the correlation ID that, when used together, relate request and reply.

   A consumer creates a request message and assigns it a unique message ID. A provider that receives the request processes it, creates the reply, and assigns the message ID from the request to the correlation ID field of the reply. The consumer now is able to correlate request and reply by matching the correlation ID of the reply to the message ID of a request.

   Figure 4-8 shows a request-reply round trip implementing the correlation ID scenario. Note that the correlation ID of the reply matches the message ID of the request, while the message ID of the reply is completely new.
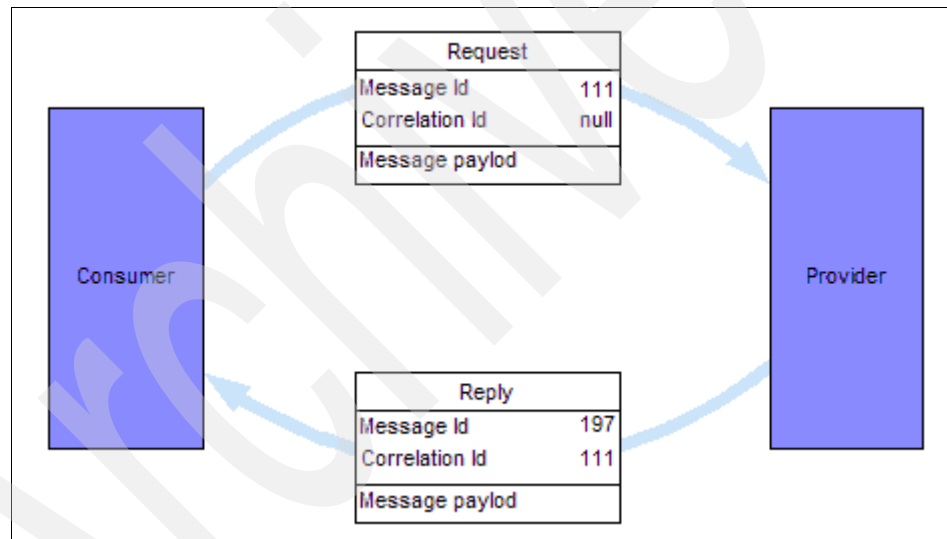

Figure 4-8   Sample of a correlation identifier scenario

   An extension of this scenario would be the chaining of multiple request-reply round trips. For example, the reply from the provider could at the same time be a request that the original service consumer would have to answer. The use of message ID and correlation ID could be applied in the same way as already described.

> **Note:** Some messaging systems do not allow users to set the message ID, but instead provide their own unique identifier. In this case, an alternative to the previous design would be to fill the correlation ID field already in the request and have the provider copy the correlation ID for the reply.

For the pseudo-synchronous approach we still have the issue that there may be more than one reply message within the same queue and we need to select a specific one. The messaging system addresses this by providing selection functionality for messages. The consumer can define specific selection criteria for messages to be consumed. This could in our case be the correlation ID in the reply message, thus making sure that the correct reply is read from the queue.

2. Dynamic point-to-point queue

   Some messaging systems support the dynamic creation of queues that, for example, can be used as reply queues.

   Before sending a request, a consumer creates a reply queue. Within the request message it supplies the name of the queue to the provider. The provider receives the request with the name of the reply queue, processes the message, and sends the reply back to the consumer using the reply queue created by the consumer. The consumer now reads the reply and removes the queue afterwards.

   The use of dynamic queues provides strong isolation between individual applications, as the reply queue is completely separate.

   There are two types of dynamic queues, temporary and permanent dynamic queues. Temporary dynamic queues are automatically removed by the infrastructure when the application that created the queue no longer accesses it. Temporary dynamic queues do not support persistent messages. Permanent dynamic queues are never automatically removed. Instead, the application that created the queue must remove it. In contrast to the temporary dynamic queue, the permanent dynamic queue supports persistent messages.

Reply address determination

There must be an agreed-upon mechanism to identify the return address to which to send a reply. Two common approaches are:

► Implicit addressing

   With implicit addressing, the name of the request queue leads to the name of the reply queue based on naming conventions. This approach is easy to implement but makes the software less flexible by hard coding the reply queue within the provider. It reduces location transparency and may not

always be appropriate. Consider, for example, a provider that needs to serve multiple consumers over the same request queue or a consumer that does not want the reply sent back to itself but instead wants to address another receiver selected based on some internal logic.

► Explicit addressing

With explicit addressing the request message contains a return address indicating where the reply should be sent to. This approach supports greater flexibility and enhanced location transparency. The knowledge of which reply queue to use is encapsulated in the message and does not have to be hard coded in the provider.

To support the transfer of the reply address, a message normally provides a field for the reply address in addition to the message payload.

Missed or delayed replies

Applications should not be designed without appropriate timeout or retry capability. Waiting indefinitely on a queue for the reply message to arrive can cause, at best, a poor user experience, and at worst can result in a consumption of system resources to the point where the entire application fails or stops responding. Applications should not be designed without a mechanism to purge queues of orphaned messages that may, for example, result from resubmitted requests. Orphaned messages in the queues can lead to poor performance and maintenance issues.

Decisions regarding how to behave in these situations is generally governed by the business requirements. For example, an online query of a data store has different functional and non-functional requirements than a batch job that updates the data store's data. In the case of the query, the same request may be submitted many times with the reply needed within a reasonable amount of time. In the case of the update batch job, the update must only occur once, but the response time is not that important.

Resubmitting a request when a reply does not arrive in a reasonable amount of time is usually an appropriate approach and leads to self-healing systems. A pseudo-synchronous call is driven by the consumer who puts a message on the request queue and immediately starts listening for the reply. After a couple of seconds a listener timeout occurs notifying the consumer that a reply did not arrive yet. The consumer now has the option to put the same request with the same message ID once more onto the request queue and wait for the reply. If the reply arrives now, the consumer may continue its processing, repeat the procedure, or take an exception path.

Figure 4-9 on page 76 shows the sequence diagram for resubmit and exception handling after a listener timeout of the consumer. The timeout occurs after the consumer invokes a get on the reply queue between steps 1.2 and 1.3.
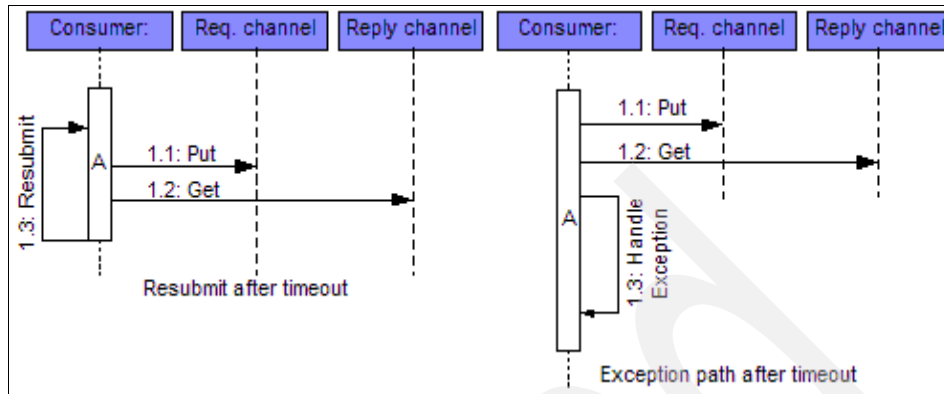
Figure 4-9   Resubmit and exception handling after a timeout

When resubmitting a request, consider the following:

▶ Ensure that the original request and the resubmits do not influence each other.

In cases where requests lead to persistent data manipulation, ensure that only one of the requests is executed, either the original request or one of the resubmits. A way of achieving this is by maintaining a request log containing message ID, request, and reply message. For further information see "Request-reply logging" on page 78.

▶ Ensure that neither request nor reply messages reside on the queue for an indefinite amount of time.

Consider a resubmit that leads to two reply messages with the same correlation ID because of a delayed reply. The pseudo-synchronous consumer expects just one, and is therefore never going to read the second message. For this reason messaging systems provide expiration functionality for messages, meaning that after a defined time a message is discarded and removed from the queue. For further information see "Message expiration" on page 78.

4.4.3 Selecting a messaging pattern

None of these options is incorrect if implemented correctly. The user's requirements and experience will dictate which decision is the correct one. A fire-and-forget communication pattern is applied if a consumer does not need to get a reply. A request-reply communication pattern needs to be applied if a consumer needs to get a reply based on the request.

You need to decide if the communication should be fully asynchronous or pseudo-synchronous. Either is valid in a Web environment. If using the asynchronous, provide a page where the user can poll for the reply. If using the pseudo-synchronous style, lead the user automatically to the next page when the reply arrives.

4.5 Messaging application design

After the adoption of the messaging model and pattern, the next step is to refine the design with regard to the message producer, the message consumer, and the messages to be exchanged.

4.5.1 Application design in general

This section contains general messaging guidelines related to message producers, consumers, and the underlying infrastructure.

Message types

Based on the intent, messages can be classified as one of the following:

► Command messages that enable procedures call semantics to be executed on another system

► Document messages enabling a messaging system to transport a document or information

A command message in fact is a simple regular message containing a command together with its parameters. Command messages are usually sent point-to-point so that each command will be consumed and executed once.

A message does not necessarily need to trigger some functionality on another system. Sometimes it just needs to pass information. For this reason there are document messages. A document message can be seen as one parameter of a command message or the result of a command message, with no intention of triggering a specific function on the other system. Document messages are not only used in point-to-point but also in publish-subscribe scenarios in case a number of recipients need to be addressed.

A request-reply scenario is a sample where both message types are used. The request represents a command message, triggering some remote functionality. The reply only transmits the result of the functionality and therefore tends to be a document message.

Message expiration

Sometimes guaranteed delivery without time constraint makes no sense. For that reason an expiration duration can be attached to each message to indicate after what amount of time a message becomes unusable and therefore should just be discarded.

Purging old messages from queues is an important consideration in the design of a messaging-based application. Not doing so will lead to poor performance and maintenance issues as more and more messages are jamming the queues.

In the event that discarding messages by expiration is not possible, another approach needs to be applied, like adding application logic to a client for monitoring the queues and handling old messages. For this reason messaging systems provide browse functionality for queues in their API to support the scanning of messages without reading them out of the queue.

Message persistency

Another design consideration is whether messages need to be persistent or non-persistent. Non-persistent messages do not survive process failure, but because of their nature they can be processed much faster by the messaging infrastructure and are less resource consuming. The decision to use persistent or non-persistent messages is generally governed by the business requirements.

Request-reply logging

Both the service consumer and service provider should maintain a request-reply log containing the message ID, the request and reply message, as well as related key information like the request and reply timestamps and the reply address. In addition, the system should be designed such that each request belongs to a unique message ID and can thus be identified using the message ID. In other words, two messages containing the same message ID reflect the same request.

> **Note:** Some messaging systems do not allow users to set the message ID, but provide their own unique identifier. In such cases the correlation ID should be used for the request identification.

Maintaining a request-reply log is considered a best practice and shows advantages in different areas:

► Eases system maintenance

By maintaining a request-reply log we always know the requests a service consumer issued, the requests a provider got, which requests have been

processed, and what the outcome of the processing was. This provides a solid background for debugging purposes and performance measurement.

► Prevents the execution of the same request more than once

In cases where requests lead to persistent data manipulation we need to make sure that each request is processed only once. However, we need to support resubmit functionality so application logic is required to implement this constraint.

► Supports the concept of self-healing systems

The concept of self-healing systems leads to the ability of a system to recover itself from certain failures. By maintaining a request-reply log the service consumer always knows which requests are due and is therefore able to decide whether a specific request should be resubmitted.

Recovery can be needed if, for example, a service consumer crashes during the processing of the reply and the reply message may be lost. To recover and successfully process the reply, the service consumer would need to force the provider to send the reply again. This can be achieved by resubmitting a request using the same message ID.

Figure 4-10 on page 80 shows the activity diagram of a provider's request handling using an applied request-reply log. A request only gets processed if there is no log entry with the same message ID. If an entry exists, the reply is taken from the log and put on the reply queue. Further processing is skipped, thus guaranteeing consistency between request and reply.
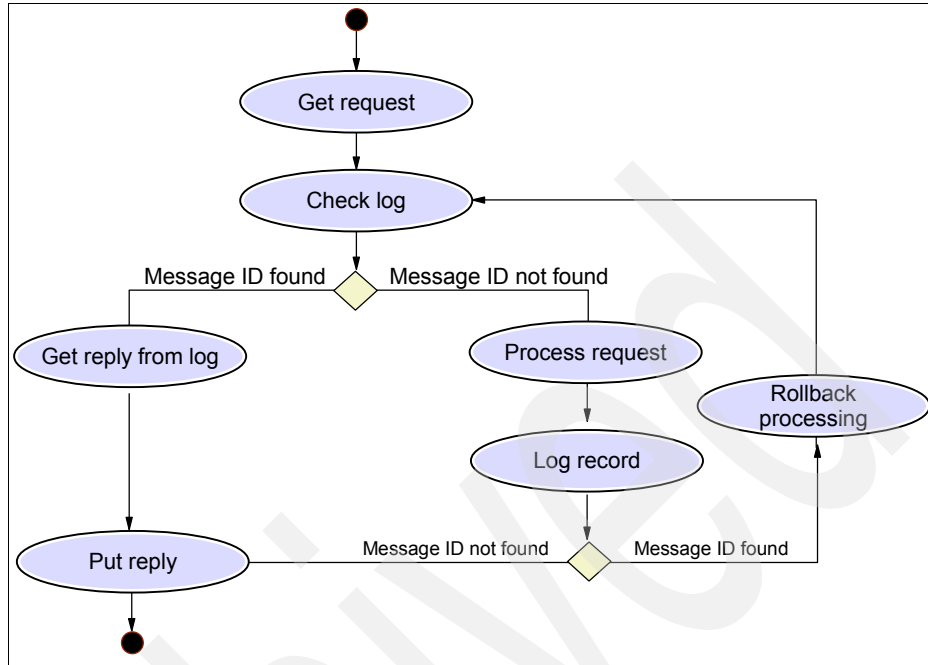
Figure 4-10   Activity diagram of the request handling using a request log

4.5.2 Message consumers

This section contains design guidelines and best practices applied to message consumers. It discusses the synchronous as well as the asynchronous implementation approach. A behavior that needs to be supported by both types is the ability to select specific messages from a queue, for example, to get just the corresponding reply for a request.

Selective consumer

The selective consumer enables selective reading of specific messages in a queue. This ability is required because it is often not possible to create a specific queue for each message type, either because this would involve too many queues or because dynamic queues are preferred and not known in advance.

Messaging addresses this issue by providing a mechanism that allows message producers to set message header properties that can be used from message consumers as selection criteria for specific messages.

Selective consumers can be used to implement filtering, dispatching, and ordering functionality.

Consider, for example, a car purchase process. An employee may buy cars without telling someone until a specific price limit; above that limit there is notification to his manager needed. The notification process can be implemented using messaging by copying the purchase price as a property into the message header, thus allowing two different consumers to select messages based on the price. One consumer would have to consume the messages below the critical price, the other one above this price, providing notification functionality.

Example 4-1 shows a code sample of a producer that sets the price as a header property in the message.

Example 4-1   Message producer

```
public class JmsSelectionSupportedProducer {
    public void sendMessage(String msg, String price, String conFactoryName,
                       String destName) throws NamingException, JMSException {
        ...
        //Create the message, set the property and send the message
        TextMessage message = session.createTextMessage(msg);
        message.setIntProperty("price", price);
        producer.send(message);
        ...
    }
}
```

Example 4-2 shows a code sample of a selective consumer. The consumer only reads messages whose value of the price attribute in the message header is higher than 10000.

Example 4-2   Selective consumer

```
public class JmsSelectiveConsumer {
    public String receiveMessage(String conFactoryName, String destName)
                                    throws NamingException, JMSException {

        String messageSelector = "price > 10000";

        //Use the session, destination and selector to create the consumer
        consumer = session.createConsumer(destination, messageSelector);

        //Reveive the message
        TextMessage message = consumer.receive();
        ...
    }
}
```

The selection criteria should be chosen in a way that guarantees all possible variants are served. If this is not the case messages that do not expire may stay in the queue without ever being read by a consumer.

Polling consumer

The polling consumer acts in a synchronous manner since the receiver thread is blocked until a message is available. Normally, it is the application or service that controls polling consumers by telling them when to start polling.

A pseudo-synchronous request-reply scenario is a sample where polling consumers are applied. The service consumer triggers the receiver functionality immediately after sending the request. The receiver thread is then blocked until either the reply message arrives or a receiver timeout occurs.

Design considerations that should be made during the design of polling consumers involve the definition of receiver timeouts as well as definitions about exception paths to take if expected messages do not arrive. Additional design is needed for consumers that may just read specific messages, for example, messages with a specific correlation ID.

Example 4-3 shows a code sample of a simple polling consumer. The consumer could be used as a receiver on the service consumer side to support a pseudo-synchronous request-reply scenario. Note the selector, which enables the consumer to listen for a message with a specific correlation ID.

Example 4-3   Polling consumer

```
public class JmsPollingConsumer {
    public String receiveMessage(String conFactoryName, String correlationId,
            String destName, int timeout) throws NamingException, JMSException {

        Session session = null;
        Connection connection = null;
        MessageConsumer consumer = null;
        String msg = null;
        String messageSelector = "JMSCorrelationID='ID:" + correlationId + "'";

        try {
            //Get the specified connection factory and queue
            Context jndiConext = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory)
                                            jndiContext.lookup(conFactoryName);
            Destination destination = (Destination)jndiContext.lookup(destName);

            //Create the connection and session
            Connection conenction = factory.createConnection();
            Session session = connection.createSession(false,
```

```
                                                        Session.AUTO_ACKNOWLEDGE);

            //Use the session, destination and selector to create the consumer
            consumer = session.createConsumer(destination, messageSelector);

            //Reveive the message
            TextMessage message = consumer.receive(timeout);
            msg = (String)msg.getText();
        }
        finally {
            if (consumer != null) {
                consumer.close();
            }
            if (session != null) {
                session.close();
            }
            if (connection != null) {
                connection.close();
            }
        }

        return msg;
    }
}
```

Event-driven consumer

An event-driven consumer acts in an asynchronous manner since it
automatically consumes messages as they become available. Unlike the polling
consumer, the event-driven consumer does not have a blocked receiver thread.
Instead, a new thread is started as soon as a message is available.

Using event-driven consumers instead of polling consumers allows systems to
behave more responsively because messages are processed as they arrive and
not as the application decides.

Event-driven design provides the following benefits:

► Eases implementation of applications and services involving unpredictable
  and asynchronous occurrences

► Eases assembling of existing applications and services

► Supports component and service reuse

► Supports loose coupling between message producer and consumer

The design of event-driven consumers should include message throttling. Because event-driven consumers are triggered by the arrival of messages, the design needs to make sure that the system is not overloaded. A simple way to achieve throttling is to limit the number of threads that can be created dynamically to process incoming messages.

Example 4-4 shows a code sample of a simple event-driven consumer. Event-driven consumers are supported by message driven beans (MDBs). MDBs provide an onMessage method, which is called by the infrastructure for each incoming message.

Example 4-4   Event-driven consumer

```
public class JmsEventDrivenConsumer implements MessageDrivenBean,
                                                MessageListener {
    ...
    public void onMessage(Message msg) {
        //This method is triggered by each incomming message
    }
}
```

Handling of poison messages

A poison message is a message that an event-driven consumer is not able to process. It might be corrupt or just in an unexpected format. If an event-driven consumer discovers a poison message it has several options for dealing with it:

► Roll back the message.

   By triggering rollback on an event-driven consumer the message is put back on the queue where it came from. This approach only works if the consumer runs within a transaction.

► Move the message to another queue.

   Moving the message to another queue enables special processing for that message. This approach is useful if the consumer does not run within a transaction.

► Discard the message.

   The message is deleted, and thus lost without being processed.

The rollback approach needs special consideration, as it can lead to possible loops. Consider the case of the rollback of a corrupt message back into the queue where it came from. The same message stays corrupt and will therefore be rolled back again and again and again. A useful approach for resolving this issue (also provided by MDBs) is the definition of a *maximum retries* property that defines how many times a consumer tries to read the same message before

the listener port automatically stops listening. Unfortunately, this stops all processing. In fact, there are two more properties needed, and provided by JMS:

► The *redelivery count* message property

This property defines how many times a message has already been read from the queue.

► The *backout threshold* destination property

When the same message has been read a number of times equal to the backout threshold, the infrastructure moves the message to a default queue for messages that could not be delivered. The queue is called the dead letter queue.

A rollback behavior that supports multiple (but not unlimited) processing retries without stopping the listener can be achieved by setting the maximum retries higher than the backout threshold. This allows the consumer to try to process a message for the number of times specified by the backout threshold without reaching the maximum retries number. Each time the redelivery fails the redelivery count is incremented. When the redelivery count equals the backout threshold, the message is moved to the dead letter queue. Maximum retries is never reached and the listener port is not stopped.

4.5.3 Message producers

Message producers are triggered by programs to deliver messages. Producers wrap messages they get from the application as payload in messaging-specific messages; make sure the header attributes like expiry, reply address, message ID, and maybe correlation ID are set correctly; and put the messages onto a queue.

Example 4-5 shows a code sample of a simple message producer that does not set any header attributes on the message.

Example 4-5   Simple message producer

```
public class JmsProducer {
    public void sendMessage(String msg, String conFactoryName, String destName)
                                         throws NamingException, JMSException {

        Session session = null;
        Connection connection = null;
        MessageProducer producer = null;

        try {
            //Get the specified connection factory and queue
            Context jndiConext = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory)
```

```
                                            jndiContext.lookup(conFactoryName);
            Destination destination = (Destination)jndiContext.lookup(destName);

            //Create the connection and session
            Connection conenction = factory.createConnection();
            Session session = connection.createSession(false,
                                                    Session.AUTO_ACKNOWLEDGE);

            //Use the session and destination to create the producer
            producer = session.createProducer(destination);

            //Create and send the message
            TextMessage message = session.createTextMessage(msg);
            message.setJMSMessageID(messageId);
            producer.send(message);

        }
        finally {
            if (producer != null) {
                producer.close();
            }
            if (session != null) {
                session.close();
            }
            if (connection != null) {
                connection.close();
            }
        }
    }
}
```

Example 4-6 shows an extension of the previous example by setting some
header attributes on the message. This producer implementation could be used
as a sender on the service consumer side to support a pseudo-synchronous
request-reply scenario. Note the reply queue and the message ID are set.

Example 4-6   Message producer setting message header attributes

```
public class JmsProducer {
    public void sendMessage(String msg, String conFactoryName, String destName,
            String replyDestName, long messageTimeout, String messageId)
                                     throws NamingException, JMSException {
        ...
        //Set header properties
        Destination replyDestination = (Destination)
                                         jndiContext.lookup(replyDestName);
        message.setJMSReplyTo(replyDestination)
```

```
                message.setJMSExpiration(messageTimeout);
                message.setJMSMessageID(messageId);
                ...
        }
}
```

Special design consideration is needed for the generation of the message ID.
The algorithm calculating it needs to support globally unique ID generation
across multiple systems and environments.

> **Note:** Some messaging systems provide their own unique identifier for
> message IDs. If this is the case, a generator can be used for the correlation
> ID.

4.5.4 Message producer and consumer in combination

This section shows the combination of message consumers and providers to
implement an end-to-end sample of a pseudo-synchronous request-reply
scenario.

Service consumer

Example 4-7 shows a code sample of a service consumer implementing a
pseudo-synchronous request-reply approach. Note that the consumer triggers
the receive functionality directly after the message is sent, thus blocking its
thread until the reply with the specific correlation ID arrives or a receiver timeout
occurs.

The ServiceConsumer class makes use of the previous message producer and
consumer samples by incorporating the JmsProducer as well as the
JmsPollingConsumer code.

Example 4-7   Service consumer

```
public class ServiceConsumer {
    ...
    public String pseudoSyncReqRpl(String msg) {
        String messageId = generateMessageId();
        String correlationId = messageId();
        String conFactoryName = "aConnectionFactory";
        String destName = "aDestination";
        String replyDestName = "aReplyDestination";
        long messageTimeout = 6000;
        long receiverTimeout = 5000;

        JmsProducer producer = new JmsProducer();
```

```
                JmsPollingConsumer consumer = new JmsPollingConsumer();

                try {
                    producer.sendMessage(msg, conFactoryName, destName, replyDestName,
                                                    messageTimeout, messageId);
                    msg = consumer.receiveMessage(conFactoryName, correlationId,
                                                    replyDestName, receiverTimeout)
                    return msg;
                }
                catch(...) {
                    ...
                }
        }
}
```

Service provider

Example 4-8 shows a code sample of a service provider implementing a
request-reply approach. The service provider does not know the communication
actually is pseudo-synchronous, as the messaging style is influenced by the
behavior of the service consumer.

Example 4-8   Service provider

```
public class JmsEventDrivenConsumer implements MessageDrivenBean,
                                                            MessageListener {
    ...
    public void onMessage(Message msg) {
        String conFactoryName = "aConnectionFactory";

        //Get needed information from the message
        String correlationId = msg.getJMSMessageID();
        String replyDestName = msg.getJMSReplyTo().getQueueName();
        String msg = (String)((TextMessage)msg).getText();

        //Process the message
        msg = processMessage(msg);

        //Send the reply back
        JmsProducer producer = new JmsProducer();
        producer.sendMessage(msg, correlationId, conFactoryName, replyDestName)
    }
}
```

4.6 Designing a messaging-based SOA

SOA has implications not just at the technology level but at the junction point between business and technology. It helps bridge the gap by bringing a business-driven focus to how we expose and orchestrate services. The principles of an SOA must therefore not just be addressed on a technology design level, but need to be considered from the beginning. This section goes through the different steps of the SOA approach at a high level, and then looks at how to meet the following SOA principles when designing a messaging solution:

- ► Granularity

  Services should be delivered at a level of granularity and abstraction that is meaningful to the service requestor.

- ► Modularity

  Services should be modular to enable the aggregation of the services into an application with a few well known dependencies.

- ► Loose coupling

  Services should be loosely coupled to minimize the impact of changes across dependencies.

4.6.1 SOA approach

Figure 4-11 on page 90 shows the different steps of the SOA approach, leading from service identification to service specification and then to service realization.
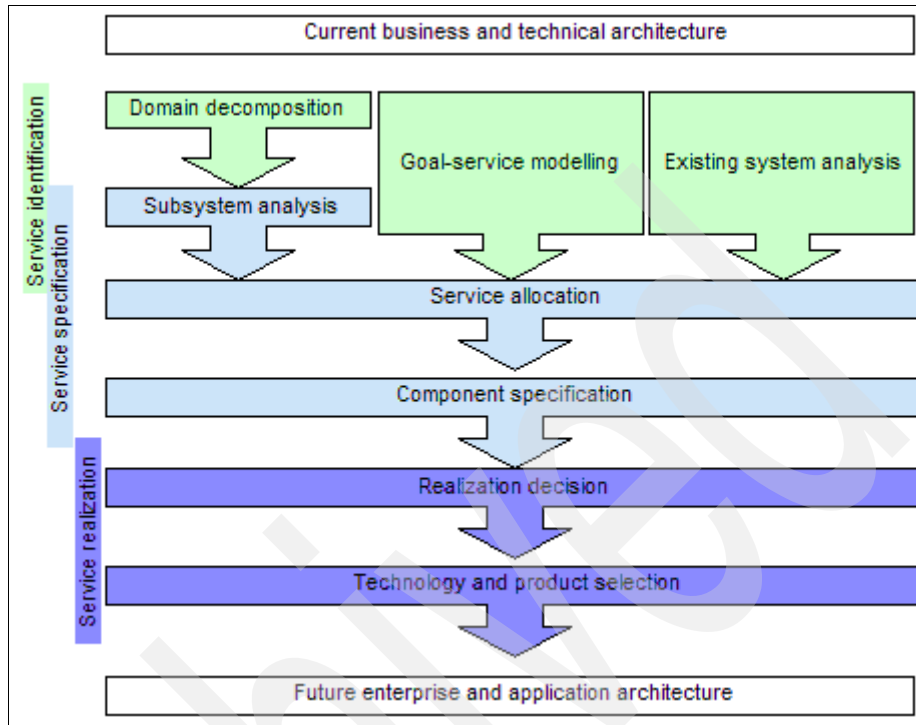
Figure 4-11   Steps of the SOA approach

Depending on the stage of the SOA approach, the different principles have more or less weight. The service granularity can be seen as crucial during the service identification and specification phase, while the modularity is mostly addressed during service specification. The loose coupling is quite technology dependent and needs special consideration during service realization. Of course, there is no strict assignment of a principle to a specific phase of the approach, but a decision taken within a specific step influences a particular principle more or less.

4.6.2 Service identification

The service identification process combines the three techniques of the top-down, bottom-up, and middle-out approach.

During the top-down approach the business domain is decomposed into subsystems and high-level business use cases that provide a specification for business services. The bottom-up approach focuses on asset harvesting through the analysis of existing systems and historical applications to determine those that could be turned into reusable services. The middle-out approach applies the

goal-service modeling technique where business services are identified based on the service requirements of a goal in scope.

Top-down: Domain decomposition

Decomposition describes an iterative approach of taking a piece of something and breaking it down into smaller pieces until the level reaches a desired granularity. In the case of service identification we talk about a business domain that gets decomposed into its functional areas and subsystems, processes, subprocesses, and high-level business use cases.

Business use cases are intended to be exposed as business services and should be kept on a granularity where they still offer a reusable amount of business functionality.

Middle-out: Goal-service modelling

Goal-service modelling is similar to the domain decomposition technique described above. Instead of decomposing a specific business domain, the goal-service modelling technique decomposes a business goal within the scope of work.

A goal can normally be identified by interviewing a business owner. The goal is then broken down into sub-goals. Again this is an incremental approach that is continued until the sub-goals reflect a desired granularity. The sub-goals are then mapped to services that can fulfill the sub-goals.

Note that the technique between domain decomposition and goal-service modeling is very similar but starts at different points. Choosing another starting point changes the question, and therefore the answer may be different.

Bottom up: Existing system analysis

The analysis of existing systems should always be considered, not just in cases where an integration of the whole or a part of it is planned. It must be seen as the complement to the domain decomposition and goal-service modelling approach, which is primarily driven by the business. System analysis is normally based on information gathered from documentation (a good starting point), from interviews with the maintenance team, and of course from code reviews.

As with the other service identification approaches, decomposition works here as well. In addition, it helps deal with the complexity of the systems. Existing systems can be broken into applications, modules, and transactions, as well as into business rules. The information gathered during this process either provides input for some system componentization decisions leading to services, or in case of a replacement it provides input for new services.

4.6.3 Service specification

The service specification step transforms the conceptual business-related view into a more design and architecture-driven view. Service specification means the definition and specification of business and technical components as well as the mapping of the use cases and services onto these components.

Subsystem analysis

Subsystem analysis involves the exposition of the identified business use cases as services on the subsystem interface, the refinement of the business use cases into system use cases, and the structuring of the subsystem into business and technical components.

Each business use case relies on a set of system use cases encapsulated in the subsystem. The subsystem leverages the business and technical components to realize the use case and support the exposed business service.

During subsystem analysis business and technical components can be discovered by:

► Analyzing the process flow within the subsystem.

► Using non-functional requirements to find technical components.

► Identifying the required functionality for each business component. In fact, these are the system-level use cases that each component must support.

Service allocation

During service allocation we ensure that each service is assigned to a component and leads to a business goal. This not only gives us certainty that all services have been identified, but also that they contain business value and are therefore justified.

We identified a set of justified services, but as we have already seen in "Granularity" on page 4, within an SOA there exist different levels of abstraction. Large-grained services are composed of finer-grained services that live on a lower level of abstraction. Justifying the existence of a service is not enough. The layer on which the service should reside must be defined.

An important part of service allocation is the categorization of the services and components into hierarchies and layers. This not only prevents the design and implementation of too many fine-grained services, which can lead to performance issues and limited reuse, but also gives an indication about their composition and dependencies.

Component specification

The component specification step involves capturing and developing the properties of the business and technical components:

► Rules that need to be implemented
► Services offered
► Used data elements
► Component dependencies
► Configurable profile settings like pluggable rules and strategies

4.6.4 Service realization

The service realization step needs to answer the question of how a specific service or component is realized and which technologies and products should be used.

Realization decision

Once the needed components are specified in detail there is enough of a decision base to decide how they should be implemented. This is done during the realization decision step. At the extremes, everything can be built from scratch or can be completely outsourced. In between there are various other approaches:

► Build new component functionality.
► Transform previous components to enable reuse of functionality exposed as services.
► Integrate by wrapping previous systems.
► Buy and integrate with third-party products.
► Outsource parts of the functionality.

Technology and product selection

After the realization decision is made, the next step involves the selection of the technologies and products that should be used for the realization of the service. Criteria affecting this decision are:

► Feature compliance

Features need to be supplied to meet the functional and non-functional requirements of the system.

► Specific technology and product standards

Already defined and established technology and product standards need to be considered and met if possible.

► Existing systems and platform investments

If there are already systems and platforms that fulfill a specific need there should be good reasons for not using them. This consideration normally has crucial side effects on the development skills.

► Existing skills

Already existing team skills and experiences are key for the success of each project and therefore need special considerations.

Often a proof of concept for the use of technologies and products is a good approach, as it addresses possible issues and shortcomings already at an early stage within a project. Issues and shortcomings may not only be technology or product-related, but may also be based on some skill gaps within the team.

4.6.5 Design considerations

This section contains more information about service specification and realization and discusses approaches to key issues related to SOA and with guidance for achieving design principles.

> **WebSphere ESB users:** The patterns discussed in this section (Business Delegate, Service activator, Service facade, Service adaptor) are provided for you by SCA. They are not so much patterns to be used in the identification and creation of services themselves, but rather implicit capability that the SCA programming model gives you.

Design for interface and implementation separation

An important aspect of SOA is the separation of the service interface from its implementation, thus providing flexibility, reusability, and loose coupling. The interface should encapsulate only those aspects of process and behavior that are used in the interaction between service consumer and service provider. Everything else, for example, the implementation of the service, needs to be hidden. By explicitly defining the interaction in this way, those aspects of either system that are not part of the interaction are free to change without affecting the other system.

Patterns providing separation

Table 4-3 on page 95 shows an overview of application patterns that can be applied to support the separation of service interface and implementation.

Table 4-3   Application patterns that support separation of interface and implementation

Pattern	Description	Supported coupling aspect
Business delegate/service proxy	The business delegate hides service invocation and service implementation details from clients.	► Transport protocol transparency ► Location transparency ► Communication model transparency
Service activator	The service activator enables asynchronous event-driven processing.	► Transport protocol transparency ► Communication model transparency
Service facade	The service facade encapsulates and orchestrates components within a service implementation, and thus can be used to control granularity and provide a unified logical interface to clients.	► Reduces coupling by enlarging interface granularity
Service adapter	The service adapter provides interface conversions by mapping signatures and message formats.	► Data format transparency

Figure 4-12 shows how patterns can be applied to support looser coupling. The base infrastructure can be messaging or some other infrastructure support.
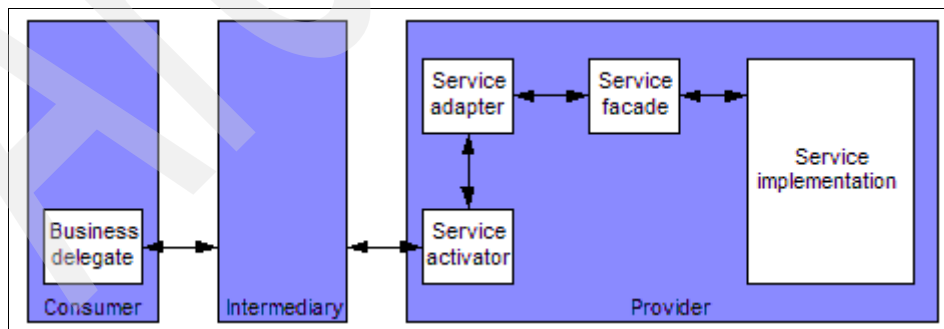
Figure 4-12   Overview of the pattern arrangement

Business delegate

The business delegate represents the interface to the consumer's service access layer. It decouples the service consumer logic from the service access logic and provides proxy functionality to the service provider. The business delegate takes care of lower-level details such as service localization and invocation.

The enhancement of the model with a factory for the creation of the business delegate supports flexibility and plugability by providing a mechanism of dynamically exchanging the service access logic without having to change any client code. The client does not know or care what kind of business delegate it gets back, whether it is one supporting JMS as the transport protocol or maybe RMI/IIOP is completely transparent. All the business delegates implement a common interface.

Figure 4-13 shows the component diagram representing the business delegate pattern together with a delegate factory.
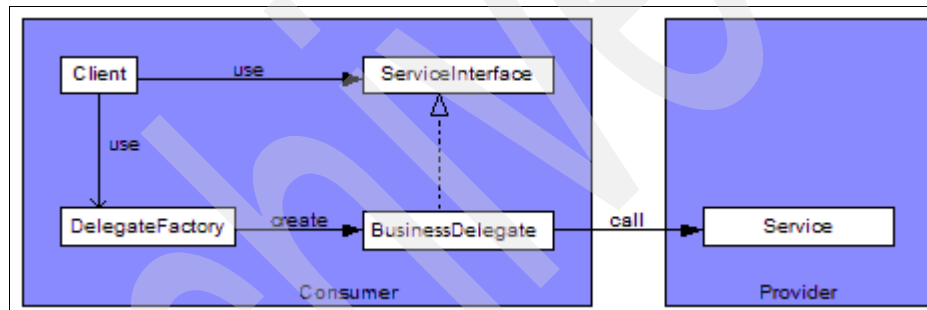
Figure 4-13   Business delegate pattern

Often business delegates can be found together with implementations of the service adapter pattern to provide interface conversion for signature as well as message format mappings. Consider, for example, that the message exchange format is XML. The right place to implement the marshalling and unmarshalling between consumer-specific data format and XML would be in the business delegate that, in this case, also acts as an adapter.

> **Tip:** The delegate pattern as well as the adapter pattern can provide a useful base for test purposes. Instead of calling a remote service they can be implemented in a way so they reply with predefined static test data.

Example 4-9 on page 97 shows the implementation of a class that contains functionality to determine whether a car identified by an ID is available. The client retrieves a car service delegate from the DelegateFactory and calls the

isCarAvailable method. For the client it appears to be a local service call, when in fact the service could reside anywhere.

Example 4-9   SellCarClient.class

```
public class CarServiceClient {
   ...
   private boolean isCarAvailable(String carId) {
      try {
         CarServiceInterface carService = DelegateFactory.getCarDelegate();
         boolean carAvailable = carService.isCarAvailable(carId);

         return carAvailable;
      }
      catch(...) {
         ....
      }
   }
}
```

Example 4-10 shows a simplified implementation of the DelegateFactory. The factory contains an algorithm that determines what kind of delegate to create. A client who uses the factory does not know what type of delegate it gets. It just accesses a generic interface whose concrete implementation was determined by some logic encapsulated in the factory.

Example 4-10   DelegateFactory.class

```
public class DelegateFactory {
   ...
   public static CarServiceInterface getCarDelegate() {
      String carDelegateType = retrieveDelegateType("carDelegate");

      if ("Jms".equals(carDelegateType)) {
         return new CarJmsDelegate();
      }
      else if ("Rmi".equals(carDelegateType)) {
         return new RmiCarDelegate();
      }
      else {
         ...
      }
   }
}
```

Example 4-11 shows a concrete delegate implementation supporting JMS as transport protocol. The delegate contains the logic for the transport binding as well as some functionality for service and parameter mappings.

Note that the receive method takes the request object as a parameter. This is needed to support the correlation between request and reply as well as to pass some context information like the name of the service that has been called.

Example 4-11   Delegate implementation

```
public class CarJmsDelegate implements CarServiceInterface {
    ...
    public boolean isCarAvailable(String carId) {
        try {
            Message request = JmsMsgHandler.marshal("isCarAvailable", carId);
            send(request);
            Message reply = receive(request);

            boolean carReserved = JmsMsgHandler.unmarshal(reply);
            return carReserved;
        }
        catch(...) {
            ...
        }
    }
}
```

Service activator

The service activator enables and encapsulates asynchronous event-driven processing by providing functionality that consumes messages and forwards them to be processed. The service that has been triggered by the service activator does not recognize the transport protocol used nor that it has been executed asynchronously. The service activator provides transport protocol and communication model transparency.

The service activator may optionally send an acknowledge to the service consumer to indicate that the request arrived or it may wait until the processing of the request has been finished and send the reply back to the consumer.

Figure 4-14 on page 99 shows the component diagram representing the service activator pattern.
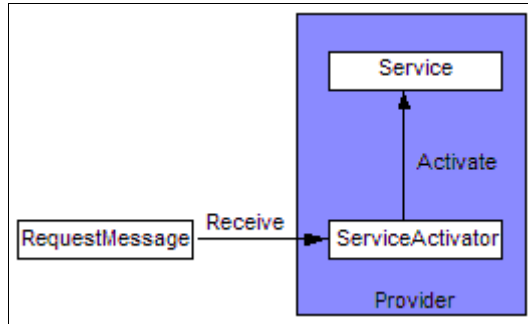
Figure 4-14   Service activator pattern

Often service activators can be found together with implementations of the service adapter pattern to provide interface conversion for signature as well as message format mappings. Consider, for example, the message exchange format is XML. The right place to implement the marshalling and unmarshalling between XML and some provider-specific data format would be directly after consuming the message, after the service activator.

Using J2EE as the selected technology, the service activator would usually be implemented by a message driven bean (MDB).

Service adapter

The service adapter provides interface conversion functionality by mapping signature and message formats of one service to the requirements of a client. The client in our case is either a service consumer or service provider functionality that wants to access a concrete service implementation.

The service adapter wraps the implementation of a service and propagates converted calls to that service to be processed. In the other direction the adapter takes the reply from the processing and converts it into the desired message exchange format.

The enhancement of the model with a factory for the creation of the service adapter supports flexibility and plugability by providing a dynamic way for adding additional format support. Consider, for example, a scenario where a service needs to be accessed with XML as well as with some key-value messages. The only thing that a service provider would have to change to support both formats is the adapter, which converts between the exchanged message format and the provider's language-specific format.

Figure 4-16 on page 103 shows the component diagram representing the service adapter pattern together with an adapter factory.
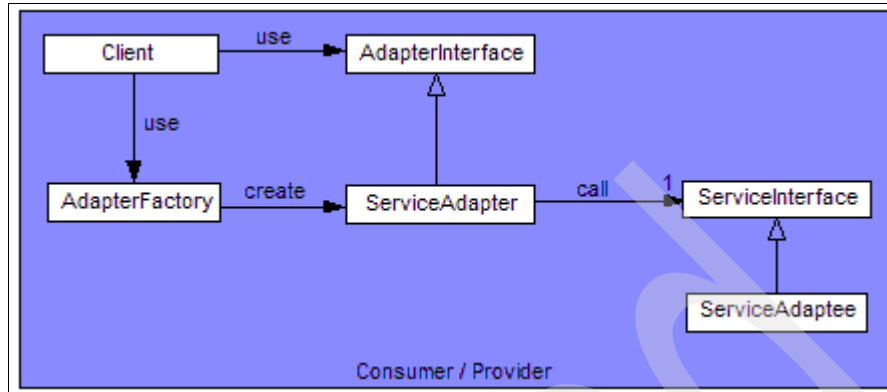
Figure 4-15   Service adapter pattern

Example 4-12 shows a client implemented with the service activator pattern using an MDB. The JmsMessageHandler class converts a JMS-specific message into a more generic ServiceCall object, thus hiding the transport protocol for the subsequent adapter call. To support plug-in functionality all adapters would have to provide just one generic method, the execute method.

Example 4-12   CarServiceActivatorBean.class

```
public class CarServiceActivatorBean implements MessageDrivenBean,
                                                MessageListener {
    ...
    public void onMessage(Message msg) {
        try {
            ServiceCall srvCall = JmsMsgHandler.unmarshal(msg);
            CarAdapterInterface carAdapter =
                                    AdapterFactory.getCarAdapter(srvCall);
            srvCall = carAdapter.execute(serviceCall);

            Message reply = JmsMsgHandler.marshal(srvCall);
            sendReply(reply);
        }
        catch() {
            ...
        }
    }
}
```

Example 4-13 on page 101 shows a simplified implementation of the adapter factory. The functionality is similar to the one used for the business delegate. The adapter factory contains the algorithm that determines what kind of adapter to create, thus hiding further invocation and processing details. Note the

ServiceCall object that is passed as a parameter is needed to determine the format of the message. As with the example before, it could be XML as well as some key-value format.

Example 4-13   AdapterFactory.class

```
public class AdapterFactory {
    ...
    public static CarAdapterInterface getCarAdapter(ServiceCall srvCall)
                                                throws AdapterException {
        String carAdapterType = retrieveAdapterType(srvCall);

        if ("Ejb".equals(carAdapterType)) {
            return new CarEjbAdapter();
        }
        else if ("JavaLocal".equals(carAdapterType)) {
            return new CarLocalAdapter();
        }
        else {
            ...
        }
    }
}
```

Example 4-14 shows the implementation of the CarAdapter class. The adapter converts the thin, generic execute method used by the client to a broader service-specific method provided by the adaptee. The adaptee hosts the business logic to be executed.

Example 4-14   CarAdapter.class

```
public class CarAdapter implements CarAdapterInterface {
    ...
    private CarServiceInterface carAdaptee = null

    public CarAdapter() {
        initialize();
    }

    public ServiceCall execute(serviceCall) throws AdapterException {
        if(serviceCall != null) {
            String serviceName = serviceCall.getServiceName();
            if ("isCarAvailable".equals(serviceName)) {
                serviceCall = isCarAvailable(serviceCall);
            }
            else if (...) {
                ...
```

```
        }
    }
    else {
        ...
    }

    return serviceCall;
}

private ServiceCall isCarAvailable(ServiceCall serviceCall) throws
                                                    AdapterException{
    String carId = serviceCall.getParam("carId");
    try {
        boolean carAvailable = adaptee.isCarAvailable(carId);
        serviceCall.addReplyParam("carAvailable", carAvailable);
        return serviceCall;
    }
    catch(...) {
        ...
    }
}
}
```

Service facade

A service facade encapsulates and orchestrates the various components within a service implementation and can be used to control granularity and provide a unified logical interface to producers.

By applying the service facade pattern service, any component dependencies can be controlled and reduced by allowing service access only through the interface exposed by the service facade. Components shielded by the facade can never be accessed directly by clients external to the service implementation, but just by the service facade and internal components. Everything behind the facade could be changed without affecting the clients accessing the service.

Figure 4-16 on page 103 shows the component diagram representing the service facade pattern. Note that clients could either be internal to the provider (for example, some service activators) or external to the provider (for example, a consumer's business delegate could bind an EJB session facade directly using RMI/IIOP).
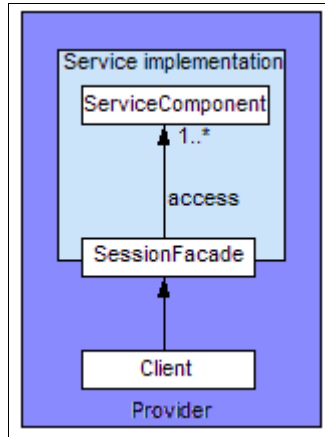
Figure 4-16   Service facade pattern

Example 4-15 shows the implementation of the CarFacade class. The facade provides a coarse-grained interface by orchestrating ReservationComponent and AccidentComponent, which provide data access functionality to database tables.

Example 4-15   CarFacade.class

```
public class CarFacade implements CarServiceInterface {
    ...
    public boolean isCarAvailable(String carId) throws FacadeException {
        boolean carDamaged = isCarDamaged(carId);
        boolean carReserved = isCarReserved(carId);

        return !carDamaged && !carReserved;
    }

    private boolean isCarDamaged(String carId) {
        try {
            AccidentComponent accComponent = getAccidentComponent();
            boolean damagedCar = accComponent.isCarDamaged(carId);
            return damagedCar;
        }
        catch(...) {
            ...
        }
    }

    private boolean isCarReserved(String carId) throws FacadeException {
        try {
            ReservationComponent resComponent = getReservationComponent();
            boolean reservedCar = resComponent.isCarReserved(carId);
            return reservedCar;
```

```
        }
        catch(...) {
            ...
        }
    }
}
```

Design for stateless services

Service calls should be independent, self-contained requests. Service providers should not hold states from earlier processing, but they should receive state as part of the service parameters. Note that the fact that a service provider can also act as a service consumer does not break this rule, as its lifetime as a service consumer is limited to the duration of dependent service calls and so is their ability of holding state.

Stateless services do not require a service consumer and a specific, executable instance of the service provider to maintain a relationship between service interactions. The successful design of stateless services depends primarily on the design of the service interface, which needs to specify all the data that is required to perform the service.

Not all technologies and especially not those who support loose coupling are capable of dealing with retained handles to specific executable service instances. Consider, for example, asynchronous messaging. There is no infrastructure support for such functionality. In fact, the implementation of statefull services would lead to much more dependency, complexity, and additional implementation effort that finally leads to the following negative impacts:

► Load balancing and failover capabilities are lost.

   The service consumer depends on one provider; if this one crashes the transaction cannot be finished.

► Coupling is tighter as a consumer depends on one provider.

   Some flexibility is lost because of the tight dependency between consumer and provider.

► Resource consumption increases.

   Resources need to be preserved for further processing; therefore, they cannot be used by other consumers.

► Service management gets more complex.

   Special consideration is required for resource pooling and service lease times that define the maximum duration a provider is assigned to a consumer.

Design for upgradeable services

SOA intends to increase reuse by providing services to multiple service consumers. Issues that arise out of this intention are related to the management of changes within service providers that are, because of their nature, propagated to the service consumers.

In cases where service consumers are affected by provider changes it is advantageous to provide a mechanism that allows steady consumer migration instead of forcing all of them to migrate on a specific date. The mechanism that supports this advantageous compatibility is versioning.

In today's industry nomenclature there exist two variations for compatibility, both of which can be addressed using versioning:

► Backward compatibility

A service is considered to be backward compatible if it is compatible with earlier versions of itself and therefore does not break existing consumers.

► Forward compatibility

A service is considered to be forward compatible if planned future versions can be deployed without affecting existing consumers.

Note: Versioning is the approach of loosening service consumer and provider dependencies in order to enhance modification flexibility. An interface or contract should be designed such that any impact propagated between a service provider to its consumers is minimized. A good approach to designing an interface is to use the assumption that once an interface has been deployed it can no longer be changed.

A methodology focusing on designing systems with well-described, stable contracts is called design by contract. This methodology not only provides guidance about how to design but also how to approach.

Table 4-4 on page 106 shows what kind of changes will break consumers and those that will not. Changes that do not break consumers can be seen as being backward-compatible as well as forward-compatible.

Table 4-4   Effects of service provider changes to service consumers

Changes that break consumers	Changes that do not break consumers
► Removing operations Existing consumers need to be modified so that they no longer use a removed operation. ► Adding mandatory attributes into existing data structure Existing consumers need to be modified to enable them to send the required attribute. ► Removing mandatory attribute from existing data structure Existing consumers need to be modified to make them aware that they no longer get an expected attribute within the reply. ► Change attribute-type Existing consumers need to be modified that they are able to deal with the changed attribute-type.	► Adding new operations Existing consumers are not aware of new operations; therefore, it does not break them. New consumers nevertheless can use new functionality. ► Adding new data-structures and data-types Existing consumers are not aware of new data-structures and data-types; therefore, they do not break. ► Adding optional attribute into existing data structure Existing consumers are not aware of new attributes; therefore, they do not break. ► Removing optional attribute from existing data structure Existing consumers are not aware that the value of the removed optional attribute is just skipped on the provider side; therefore, they do not break.

Versioning approach

The following approach describes just one way of implementing versioning. The approach is based on data structure versioning where each data structure within an exchanged message reflects a deployed version. Each data structure carries a version number that consists of a major and a minor version. The version numbers are treated based on the following guidelines:

► Minor and major version are numbers that both start with 0.

► The minor version gets incremented if a change is backward compatible. A minor version change is always based on the last minor version; therefore, backward compatibility must not be expected over more than one minor version change.

► The major version is incremented if a change is not backward compatible.

 If the major version is incremented the minor version must be set to 0.

Using these guidelines, service consumers, providers, as well as possible intermediaries are able to determine the compatibility based on the message exchanged and are able to trigger required message processing like routing and transformation.

Service enablement

Service enablement is the migration of existing monolithic applications into the building blocks of SOA. It is the process that creates a service to encapsulate the functionality provided by an existing application.

There are three approaches for service enablement whose main differences are characterized by the amount of modifications that must be made on the exiting functionality:

► Re-engineering

The re-engineering approach limits the reuse of the existing system to analysis purposes. The design and implementation of the new system is done independent of the old one.

► Wrapping

The wrapping approach does not change the existing functionality of the historical application but applies service enablement by wrapping it with well-defined, accessible interfaces.

► Componentization

The componentization approach supports service enablement by refactoring the existing application structure into components accessible by well-defined, accessible interfaces.

Table 4-5 shows an overview of these approaches together with their characteristics.

Table 4-5   Comparison of service enablement approaches

Approach	Advantage	Disadvantage
Service enablement by re-engineering	► Optimal granularity and reuse ► High design flexibility ► Replacement with actual technology ► Good performance	► No production proven functionality ► High risk ► High cost

Approach	Advantage	Disadvantage
Service enablement by wrapping	► Production proven functionality ► Low risk ► Low cost	► Suboptimal granularity and reuse ► Suboptimal performance ► Often temporary solution ► Difficult to maintain
Service enablement by componentization	► Production proven functionality ► Easy to maintain ► Good performance	► High cost ► Suboptimal granularity

There is no silver bullet for dealing with older system modernization. The approach taken always depends on factors like cost and time budget, team skills, quality of the older system, and required flexibility.

4.7 For more information

For more information about this topic you may find the following useful:

► *WebSphere MQ Application Programming Guide*, SC23-6595

  http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pb
  i.cgi?CTY=US&FNC=SRX&PBL=SC34659500

► *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303

► Enterprise Integration Patterns Web site

  http://www.eaipatterns.com/index.html

► Pattern Solutions Web page on IBM developerWorks®

  http://www-128.ibm.com/developerworks/rational/products/patternsolutions/

5

Point-to-point runtime configuration

The simplest scenario in which to begin building an SOA environment is to introduce connection middleware that can be used to control the connectivity between applications. This lacks most of the features that make an SOA architecture, but is a simple way to begin introducing the elements into an existing environment. Since no mediation is used at this stage, this chapter focuses on connectivity between messaging transport software. These techniques form the basis for building more advanced SOA runtime infrastructures.

We first give an overview of the process required to configure WebSphere MQ queue managers, queues, and connectivity between queue managers. This will give you an idea of how the structural elements of WebSphere MQ are created and connected.

Next, we show how to connect a WebSphere Application Server service integration bus (or just the bus) to WebSphere MQ. WebSphere ESB is built on WebSphere Application Server and the bus is used for message transport. Thus, this process is the same whether you are using WebSphere Application Server or WebSphere ESB.

Last, we show how to set up shared queues using WebSphere MQ for z/OS.

5.1 WebSphere MQ configuration

This section gives an overview of the elements required in WebSphere MQ to receive and deliver messages. It covers the basics of creating queue managers and queues, and connecting one queue manager to another. If you are not familiar with WebSphere MQ concepts, review "WebSphere MQ" on page 28.

To illustrate, let us assume that an application wants to input data to node 1 and another application wants to get the data from node 2. Assured data delivery is a requirement.

First we need a queue manager on each node. QM1 will be created on node 1 and QM2 on node 2. In QM1, we have to define a remote queue definition, sender channel, and transmission queue. In QM2, we have to define target queue, receiver channel, and listener. This configuration is shown in Figure 5-1.
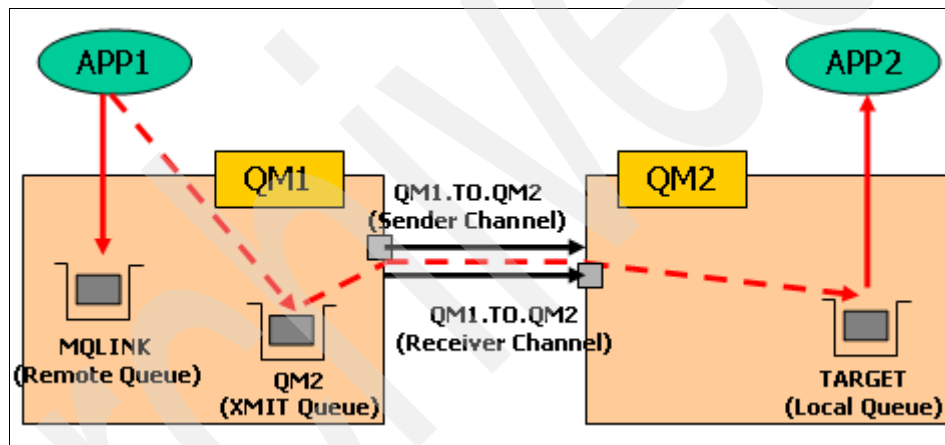
Figure 5-1   WebSphere MQ - WebSphere MQ integration

Note that in this configuration the following is true:

► The MQLINK queue in QM1 is a remote queue definition pointing to the TARGET queue in QM2.

► QM2 queue in QM1 is a transmission queue.

► There is a pair of sender and receiver channels between the two queue managers.

When application APP1 puts XML data on the MQLINK queue, the data is delivered through the channel. Application APP2 gets the XML data from the TARGET queue.

The steps required to create and test the connection are:

1. Create the queue managers.
2. Create a remote queue definition.
3. Create a transmission queue.
4. Create a sender channel.
5. Create a local queue.
6. Create a receiver channel.
7. Start the sender channel.
8. Test the connection.

The next sections illustrate how this is done for our example. These steps are performed using the WebSphere MQ Explorer tool. For testing purposes, we used one machine and one installation of WebSphere MQ, and thus one instance of WebSphere MQ Explorer. As you go through this example, keep in mind that under normal circumstances you would be creating a connection between two machines and two instances of WebSphere MQ.

5.1.1 Create the queue managers

Let us start our configuration by creating the new queue managers using the WebSphere MQ Explorer. In the Explorer window:

1. Right-click **Queue Managers** and select **New** → **Queue Manager**.

2. In Step 1, type QM1 in the Queue Manager Name field and click **Next**.

3. In Step 2, take the default for the log values and select **Next**.

4. In Step 3, deselect **Auto Start Queue Manager**, and select **Next**.

5. In Step 4, enter the listener port number. The default is 1414. If you already have a queue manager running on the system you will need to select a different port number.

6. Click **Finish**.

Create QM2 using the same process. Since we used one machine we needed to use a different listener port number (1415) in step 4.

IBM WebSphere MQ	Queue Managers				
Queue Managers	Queue...	Comman...	Queue m...	Platform	Que
QM1	QM1	600	Running	Windows	
QM2	QM2	600	Running	Windows	
Queue Manager Clus					
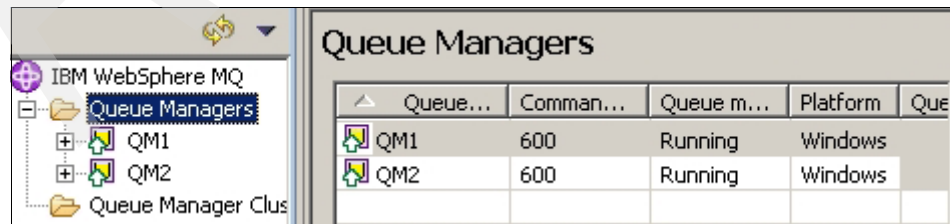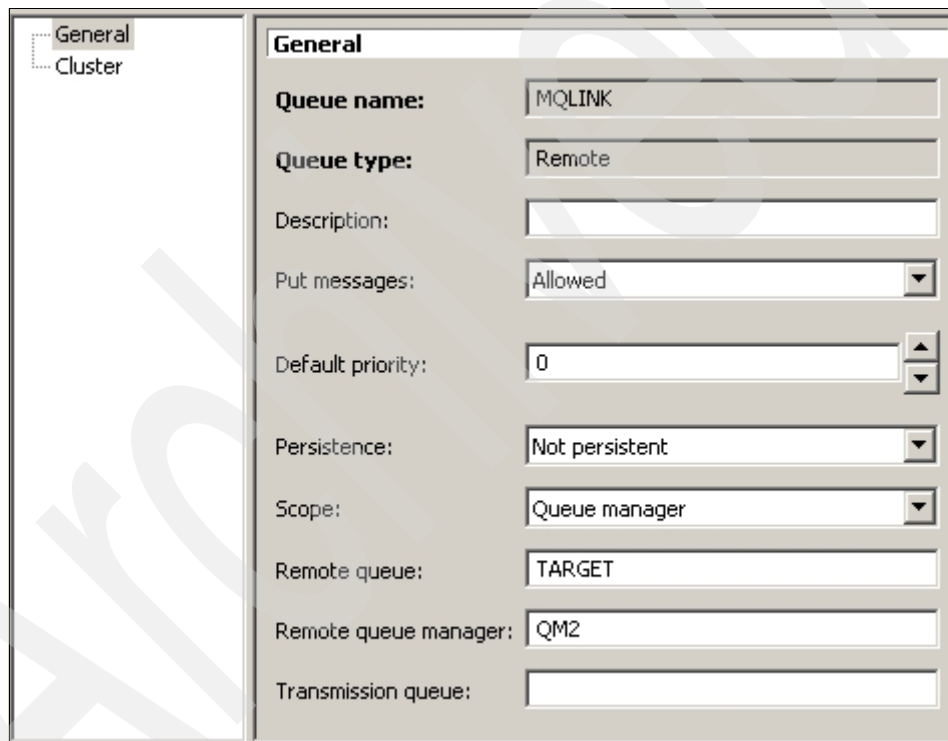
Figure 5-2   Create two queue managers

5.1.2 Create a remote queue definition

Next, we need a remote queue definition in QM1 to point to the TARGET queue that resides (or will reside) in QM2.

1. Navigate to **Queue Managers** → **QM1** → **Queues**.

2. Right-click **Queues**.

3. Select **New** → **Remote Queue Definition**.

4. Type `MQLINK` in the Name field, and click **Next**.

5. Type `TARGET` in Remote Queue field and `QM2` in the Remote Queue Manager field (Figure 5-3).

Figure 5-3   Create a remote queue definition

6. Click **Finish**.

5.1.3 Create a transmission queue

A transmission queue is a local queue on which prepared messages destined for a remote queue manager are temporarily stored. To create the transmission queue:

1. Navigate to **Queue Managers** → **QM1** → **Queues**.

2. Right-click **Queues**.

3. Select **New** → **Local Queue**.

4. Type `QM2` in the Name field and click **Next**.

5. Change the Usage field to **Transmission** and click **Finish**.

Figure 5-4   Create a transmission queue

When you are done, you will see the that there are two queues in QM1. The remote queue definition, MQLINK, and the transmission queue, QM2.
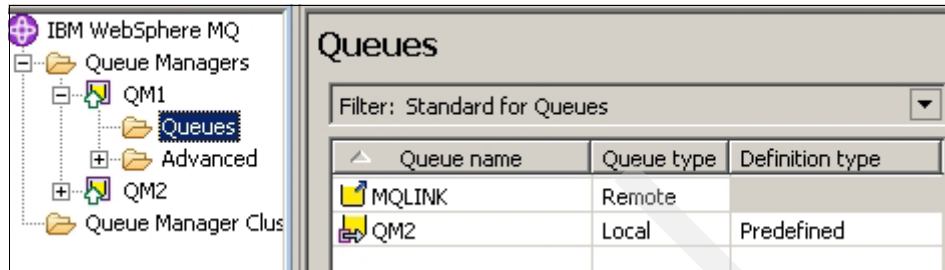
Figure 5-5   Queue definitions in QM1

5.1.4 Create a sender channel

A sender channel is a channel that initiates transfers, removes messages from a transmission queue, and moves them over a communication link to a receiver or requester channel. To create the sender channel:

1. Navigate to **Queue Managers** → **QM**1 → **Advanced** → **Channels**.

2. Right-click **Channels** and select **New** → **Sender Channel**.

3. Type `QM1.TO.QM2` in the Name field, and click **Next**.

4. In the Connection Name field, type the IP address or host name of the system hosting the QM2 queue manager concatenated with its listener port number in parentheses. In this example, because QM1 and QM2 are both on the same system, and the listener port for QM2 is 1415, we can use localhost(1415) or 127.0.0.1(1415).

5. Type `QM2` in the Transmission Queue field. This is the queue defined earlier in "Create a transmission queue" on page 113.

6. Click **Finish**.

Figure 5-6   Create a sender channel

This completes the configuration of the objects on QM1.

The next series of steps configures the objects on QM2 objects. This includes a target queue and receiver channel.

5.1.5 Create a local queue

The local queue on QM2 is the target queue of the remote queue definition on QM1. To create the queue:

1. Navigate to **Queue Managers** → **QM2** → **Queues**.

2. Right-click **Queues**.

3. Select **New** → **Local Queue**.

4. Type `TARGET` in the Name field and click **Finish**. Note that TARGET matches the name defined in the remote queue field of MQLINK in QM1 (see "Create a remote queue definition" on page 112).
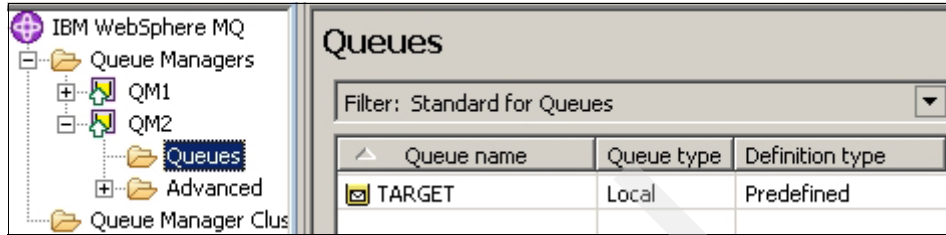
Figure 5-7   Create a local queue of the target

5.1.6 Create a receiver channel

The receiver channel is a channel that responds to a sender channel, taking messages from a communication link. To create the receiver channel:

1. Navigate to **Queue Managers** → **QM2** → **Advanced** → **Channels**.

2. Right-click **Channels** and select **New** → **Receiver Channel**.

3. Type `QM1.TO.QM2` in the Name field. Note that the sender and receiver channels must have the same name. This matches the sender channel defined in "Create a sender channel" on page 114). Unlike the sender channel, the receiver channel does not need the connection name defined.

4. Click **Finish**.
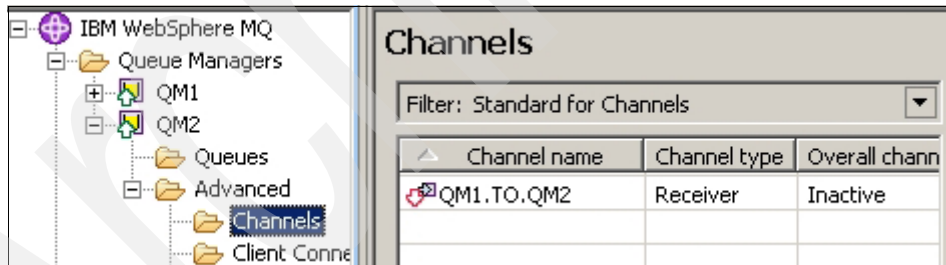
Figure 5-8   Create a receiver channel

5.1.7 Start the sender channel

Now that the definitions are complete, start the sender channel by right-clicking the **QM1.TO.QM2** sender channel under QM1 and selecting **Start**.

When the channel starts successfully, the status will change to `Running` and the icon will turn green. Note that the receiver channel icon also turns green.
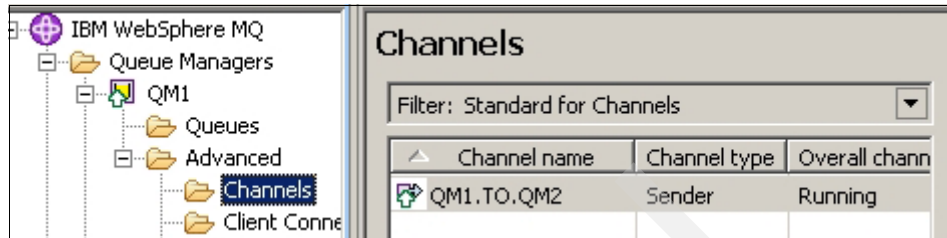
Figure 5-9   Start the sender channel

5.1.8 Test the connection

The easiest way to test the connection is using RFHUtil. This utility is provided as SupportPac™ IH03 and can be downloaded from the following location:

```
http://www-1.ibm.com/support/docview.wss?rs=203&uid=swg24000637&loc=en_US&cs=ut
f-8&lang=en
```

An XML file can be used for testing. We used Airline1.xml from "Sample XML files" on page 270.

To test the connection:

1. Run rfhutil.exe.
2. Select **QM1** in the Queue Manager name field.
3. Select **MQLINK** in Queue Name field.
4. Select **Read File** to open the XML file.
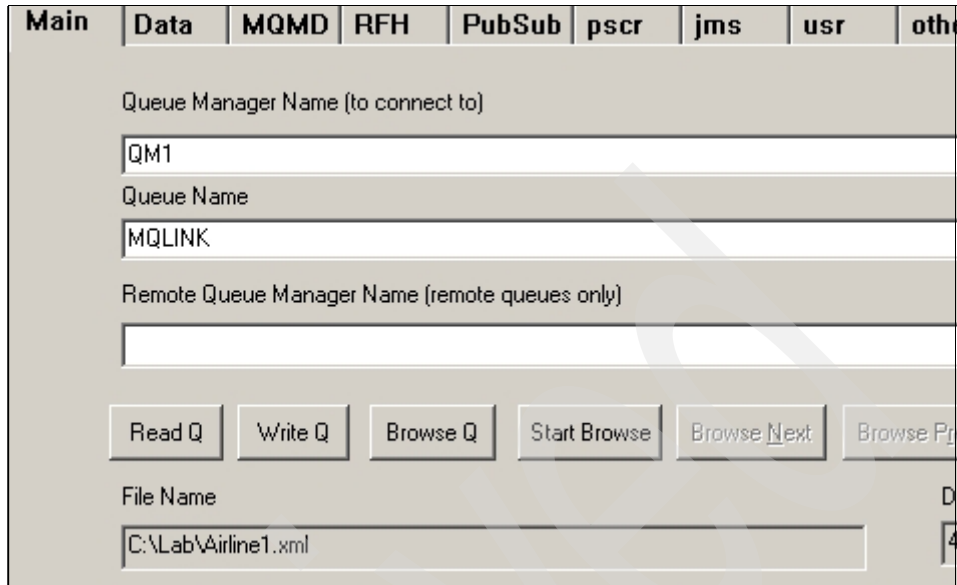
Figure 5-10   RFHUTIL Main panel

5. Click the **Write Queue** button.

   The XML data will be put on the MQLINK and sent to the TARGET queue in QM2.

6. You can see the message on the TARGET queue using WebSphere MQ Explorer. Click **Queues** under QM2. Select **TARGET** from the list displayed, right-click, and select **Browse Messages**.
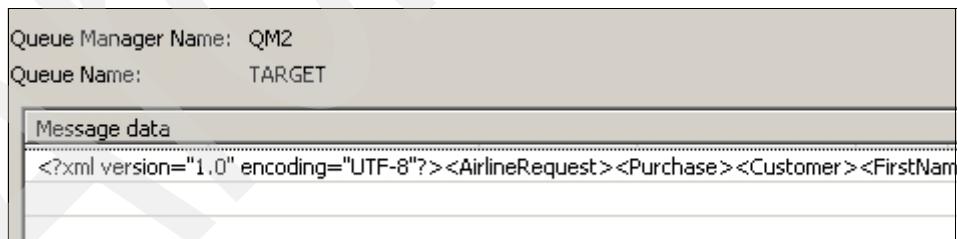
Figure 5-11   Delivered message in TARGET queue

5.2 Connect WebSphere ESB to WebSphere MQ

In this section we show how to connect WebSphere ESB to WebSphere MQ to allow the flow of messages from one network to the other.

Consider the travel bureau introduced in "End-to-end scenario" on page 60. Airline C uses an application to issue tickets that uses the same XML format as the travel bureau. The message can be used as is with no mediation. The connection between the applications becomes a simple point-to-point connection, with the primary concern being the assured data delivery between the two applications. Airline C has a WebSphere MQ network.

The travel bureau has chosen to use WebSphere ESB as its ESB with the intent of expanding its integration solution to include airline companies that require mediation to communicate. However, their immediate need is to simply connect the application server environment on WebSphere ESB with Airline C's WebSphere MQ networks.
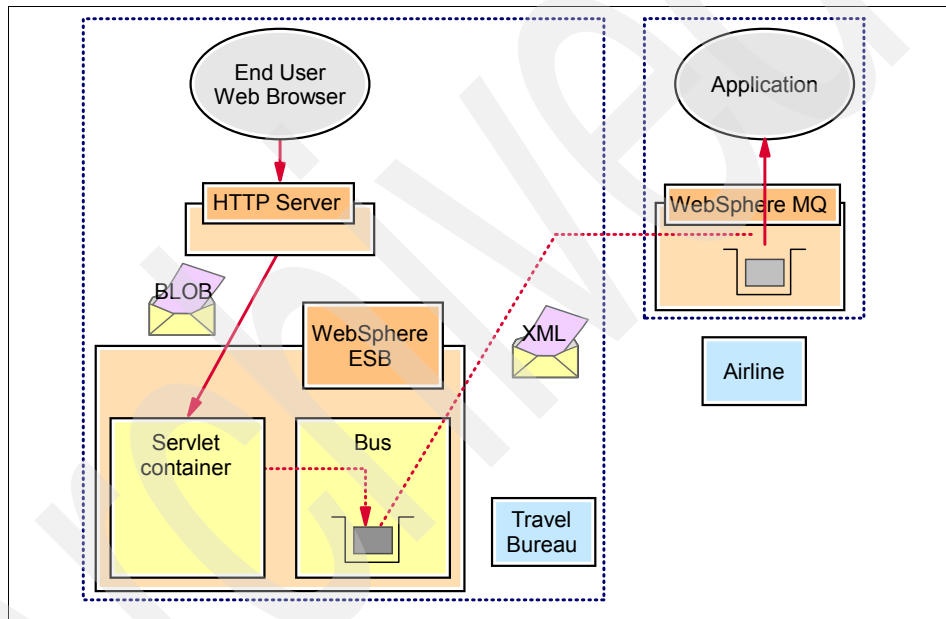
Figure 5-12   Integration scenario - Point-to-point with no mediation

> **WebSphere ESB, WebSphere Application Server, and the service**
> **integration bus:** The messaging infrastructure of WebSphere Application
> Server V6 is implemented in the service integration bus (referred to as the
> bus). WebSphere ESB, built on WebSphere Application Server, also uses the
> service integration bus as its messaging infrastructure.
>
> For this reason, the process used to connect WebSphere ESB to WebSphere
> MQ and WebSphere Application Server to WebSphere MQ are the same. In
> this discussion we use WebSphere ESB.
>
> The sample we show here is simply to illustrate the mechanics of connecting a
> bus to a WebSphere MQ environment. Before making any decisions, you
> should refer to the product documentation for planning assistance in designing
> a service integration topology.

To illustrate how to connect a service integration bus to WebSphere MQ, we use
the configuration in Figure 5-13.
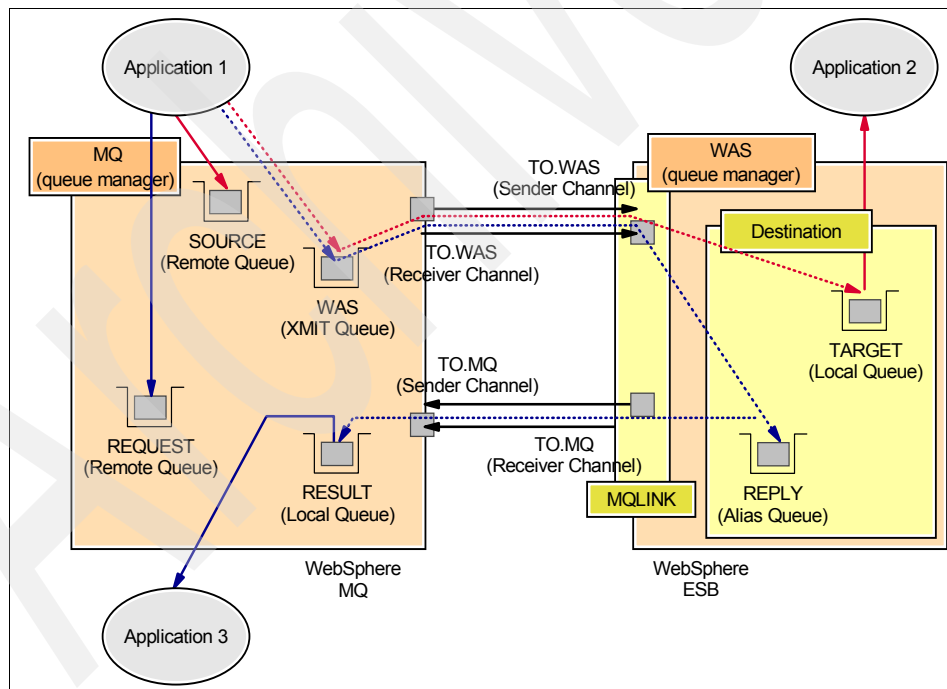
Figure 5-13   WebSphere MQ to WebSphere ESB connection

Note that in this configuration the following is required to connect the two systems:

► In WebSphere ESB

  – A service integration bus.

  – A foreign bus definition for WebSphere MQ.

  – A WebSphere MQ link that defines the specific queue manager, the listener port, and the sender channel name (TO.MQ) for the link to WebSphere MQ. The queue manager name for the bus (WAS) is also specified here.

► In WebSphere MQ

  – A queue manager in WebSphere MQ, called MQ.

  – A sender channel called TO.WAS. This defines the host and listener port for the bus. It also defines WAS as the name of the transmission queue.

  – A receiver channel called TO.MQ.

  – A transmission queue called WAS.

The following queues are defined to support the two message flow scenarios we use to illustrate the connection between the applications and how the various queue types function:

► The following queues are defined on the service integration bus:

  – A local queue called TARGET.

  – An alias queue called REPLY. This queue is an alias of the RESULT queue in the MQ queue manager and is used to route messages to that queue.

► The following queues are defined on the WebSphere MQ queue manager to support the message flow scenarios:

  – A local queue called RESULT

  – A remote queue called SOURCE that points to the TARGET queue in WAS

  – A remote queue called REQUEST that points to REPLY in WAS

In the first message flow scenario, application APP1 puts a message on the SOURCE queue in MQ. Because the SOURCE queue is a remote queue definition pointing to the TARGET queue in WAS, the data is delivered to the TARGET queue where application APP2 can read it.

In the second message flow scenario, application APP1 puts a message on the REQUEST queue. Because the REQUEST queue is a remote queue definition pointing to the REPLY queue in WAS, the data is delivered to the REPLY queue.

However, the REPLY queue is an alias for the RESULT queue in WebSphere MQ, so the message is sent back to MQ.

> **Note:** Before implementing a connection between the service integration bus and WebSphere MQ, see the following:
>
> ► PK15976; 6.0.2.3: Handling of message headers by the WebSphere default provider
>
> ```
> http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&dc=D400&dc
> =D410&dc=D420&dc=D430&q1=JMS&uid=swg24011220&loc=en_US&cs=utf-8&lang=en
> ```

5.2.1 Configure the service integration bus

The following steps are needed to define WebSphere MQ to the service integration bus:

1. Create a bus.
2. Add the application server as a member of the bus.
3. Define WebSphere MQ as a foreign bus.
4. Define a WebSphere MQ link.
5. Create the local and alias queues.

This configuration is done using the WebSphere administrative console.

Create a bus

In WebSphere ESB, there are predefind buses for use with mediation modules. Since we are not using mediation at this time, we create a new bus. In WebSphere Application Server, there are no predefind buses.

The following steps can be used to create a new bus:

1. In the console, navigate to **Service Integration** → **Buses**.

2. Click **New** and type in the name of the bus in the Name field. In this case, we use BUS as the name.

3. Click **OK** to create the new bus.
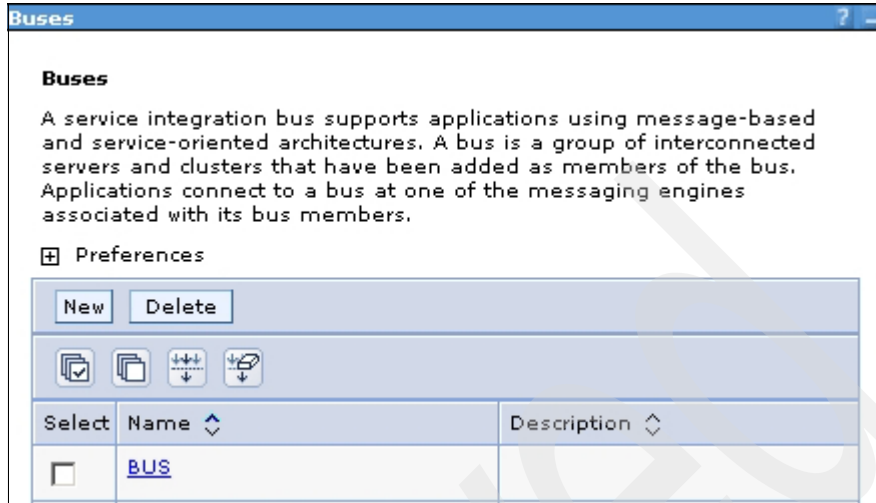
Figure 5-14   Add a new bus

Add the application server as a member of the bus

A bus can have multiple application servers or clusters within the same cell as members of the bus. Adding a member creates the messaging engine for the server or cluster that manages the message flow to and from the bus.

1. Click the bus name to open the details page.

2. Click **Bus Members**.

3. Click **Add** to add a bus member.

4. In Step 1 select the server or cluster to add as the bus member and click **Next**.

5. In Step 2 click the **Finish** button. The application server will be added as a member to the bus.
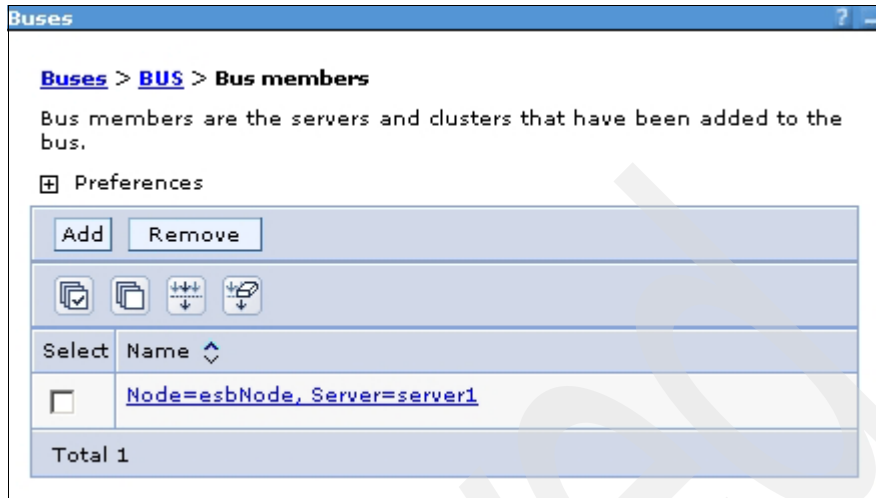
Figure 5-15   Add the application server as a new bus member

6. Save the changes.

Define WebSphere MQ as a foreign bus

The next step is to define WebSphere MQ to the service integration bus. WebSphere MQ is represented as a foreign bus.

1. Select **Service Integration** → **Buses**. Click the bus name to open the detail page.

2. Click **Foreign Buses**.

3. Click **New**.

4. In Step 1, type WAS in the Name field, and click **Next**.

5. In Step 2, select **Direct WebSphere MQ Link** and click **Next**.

6. In Step 3, just click **Next.**

7. And in Step 4, click **Finish** to add the new foreign bus definition.

Figure 5-16   Add a foreign bus

8. Save the changes.

Define a WebSphere MQ link

A WebSphere MQ link enables the exchange of messages with a WebSphere MQ network. Defining an MQ link defines the sender and receiver channels used to transmit messages to and from WebSphere MQ.

WebSphere MQ links are defined at the messaging engine:

1. Select **Service Integration** → **Buses**. Click the bus name to open it.

2. Click **Messaging Engines**. The messaging engine created for the application server (when you added the server to the bus) should be in Started state. If not, click **Start**.
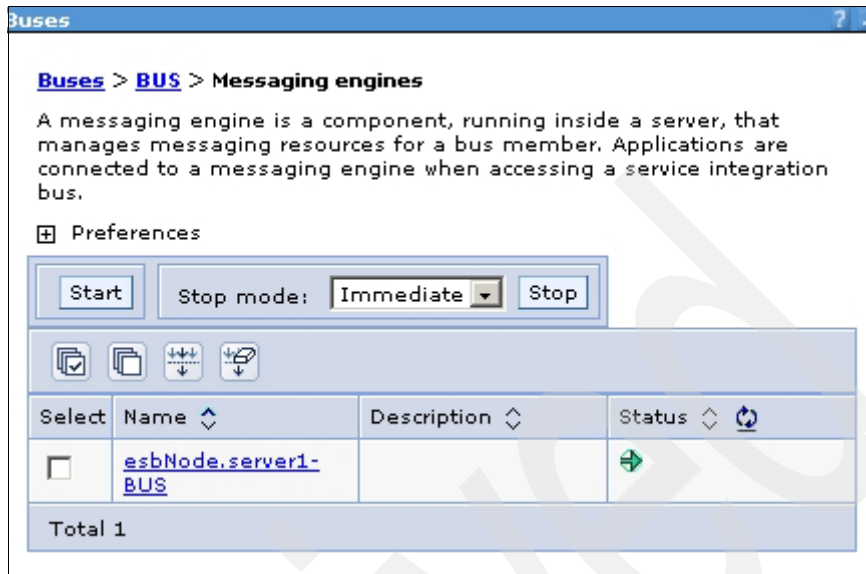
Figure 5-17   Messaging engine for server1

3. Click the messaging engine for your server to open the details page.

4. Click **WebSphere MQ Links**.

5. Click the **New** button.

6. In Step 1:

    –   Type `LINK` in the Name field.

    –   Select **WAS** in the Foreign Bus field.

    –   Type `WAS` in the Queue Manager Name field. This becomes the queue
        manager name that WebSphere MQ uses for WebSphere ESB.

    Click **Next**.

Figure 5-18   Create a WebSphere MQ link - Step 1

7. In Step 2:

   – Type `TO.MQ` in the Sender MQ Channel Name field. This defines the
     sender channel. The name used here must match the name you use for
     the partner receiver channel in WebSphere MQ.

   – Type the IP address of the WebSphere MQ host in the Host Name field.

   – Type the listener port number for the WebSphere MQ queue manager in
     the Port field. In this example, the WebSphere MQ queue manager has
     not been created yet, but assume that when it is, port 1414 will be used.
     This is the port number you enter here.

   – Select **OutboundBasicMQLink** in the Transport Chain field.

   Then click **Next**.

Figure 5-19   Create a WebSphere MQ link - Step 2

8. In Step 3, type `TO.WAS` in the Receiver MQ Channel Name field and click **Next**.

9. In Step 4, click **Finish**. You can see the newly created WebSphere MQ Link named LINK.

Figure 5-20   New WebSphere MQ link

10. Save the changes.

Create the local and alias queues

The next step prepares the bus for the application-specific queue requirements. In this example, we create a local queue to hold the messages and an alias queue to represent the WebSphere MQ queue.

1. Select **Service Integration** → **Buses**.

2. Click the bus name.

3. Click **Destinations**.

4. Click **New**.

5. Select **Queue** and click **Next**.

6. In Step 1, type TARGET in the Identifier field and click **Next**.

7. In Step 2, select the bus member to host the queue and click **Next**. In our example, we use a single server, so the default, server1, is correct.

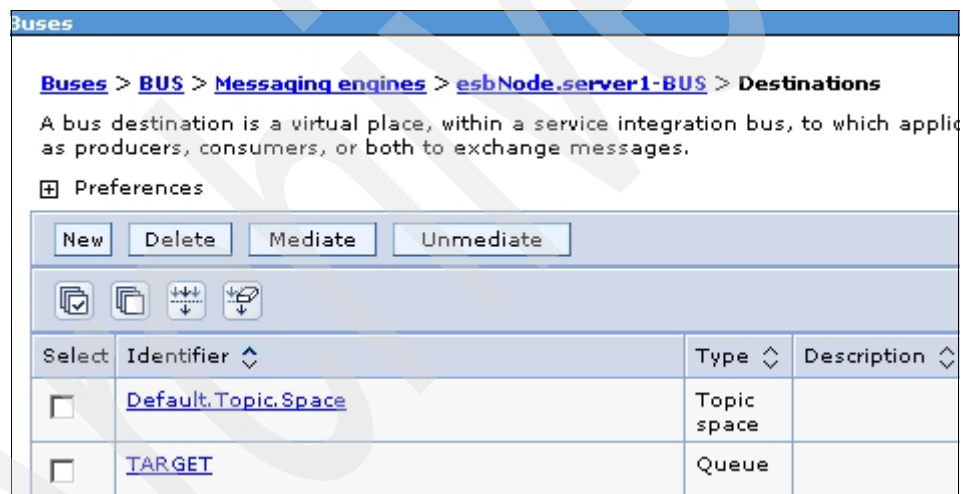8. In Step 3, click **Finish** to create the queue.

Figure 5-21   Add a new local queue in destinations

9. You should be back at the Destinations list. Click **New** to create the alias queue.

10. Select **Alias** and click **Next**.

11. In Step 1:

   a. Type REPLY in the Identifier field.
   b. Select **BUS** in the Bus field.
   c. Type RESULT@MQ in the Target Identifier field.
   d. Select the foreign bus, **WAS**, in the Target Bus field.

Click **Next**.

Figure 5-22   Alias destination attributes

12. In Step 2, click **Finish**. You will be able to see the two destinations, TARGET
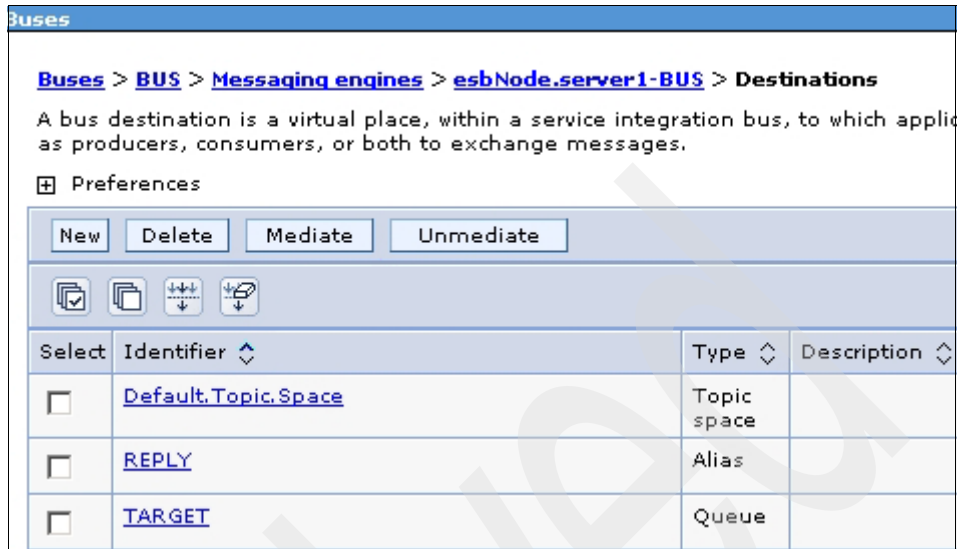   and REPLY, that you just created.

Figure 5-23   Create a new alias destination

13.Save all changes, log off, and restart the server.

5.2.2 Configure WebSphere MQ

Let us configure the objects in WebSphere MQ using the same techniques shown in "WebSphere MQ configuration" on page 110:

1. Create a new queue manager named MQ with 1414 as the listener port.

2. Create the following queues.

   – RESULT: Local queue
   – SOURCE: Remote queue definition targeting TARGET in WAS
   – REQUEST: Remote queue definition targeting REPLY in WAS
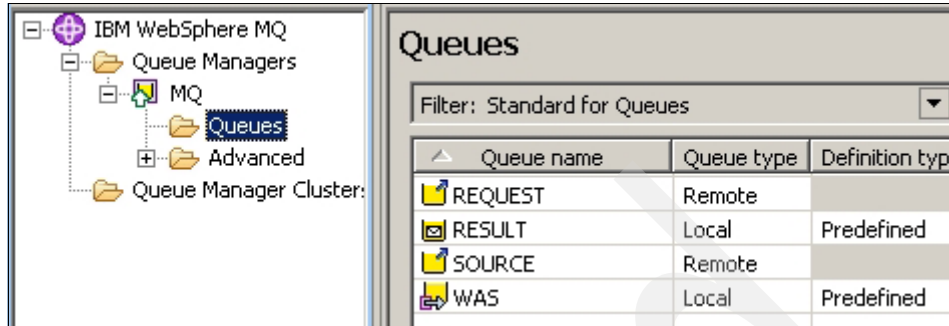   – WAS: Transmission queue

Figure 5-24   Create the WebSphere MQ queues

3. Create the following channels:

   – TO.WAS sender channel

     The connection name for this example is localhost(5558). This connects the sender channel to the service integration bus.

     The default listener port for the service integration bus is 5558 for the first application server on a node. You can check the port for your application server by going to **Servers → Application servers**. Click the server name to open the details page. Expand the **Ports** category under the Communications section. The port number used by the server is the SIB_MQ_ENDPOINT_ADDRESS.

     Enter WAS as the transmission queue.
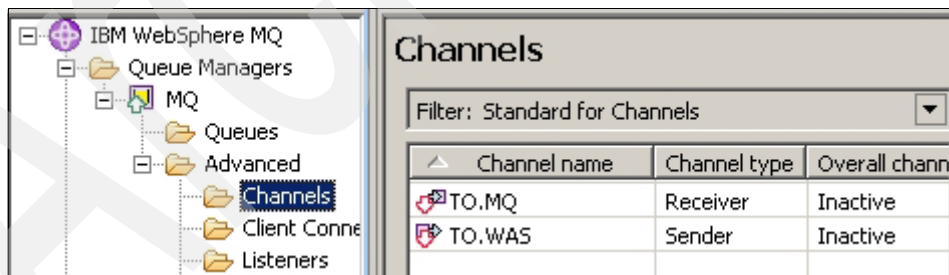
   – TO.MQ: receiver channel

Figure 5-25   Create a sender channel and a receiver channel in WebSphere MQ

5.2.3 Start the connection

After creating the sender channel on the service integration bus, it should have gone into standby status, waiting for the other side to become active. Now that

the corresponding channel definitions are defined in WebSphere MQ, you will be able to start it the sender channels on both sides.

In the WebSphere administrative console:

1. Select **Services Integration** → **Buses**. Then click the bus name to open its configuration.
2. Click **Messaging Engines**. Then click the messaging engine to open it.
3. Click **WebSphere MQ Links**. Then click the WebSphere MQ link name, LINK, to open it.
4. Click **Sender Channel**. Note the status of the channel is standby.
5. Check the box to the left of the TO.MQ channel and click **Start**. The channel will start and you can see the status change to Started.
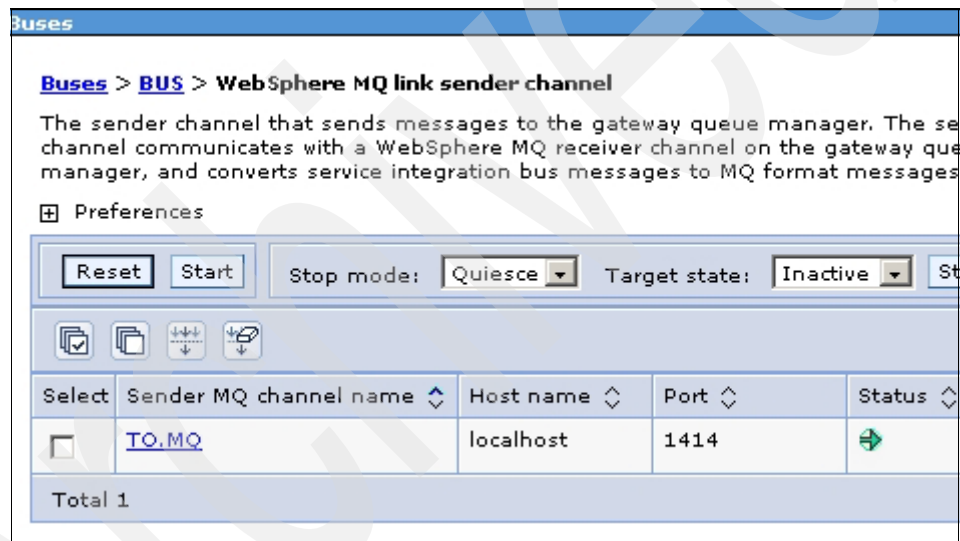

Figure 5-26   Start the sender channel in the bus

6. Start the TO.WAS sender channel in WebSphere MQ Explorer. If all channels are active on both sides, you will see the following status in WebSphere MQ Explorer.
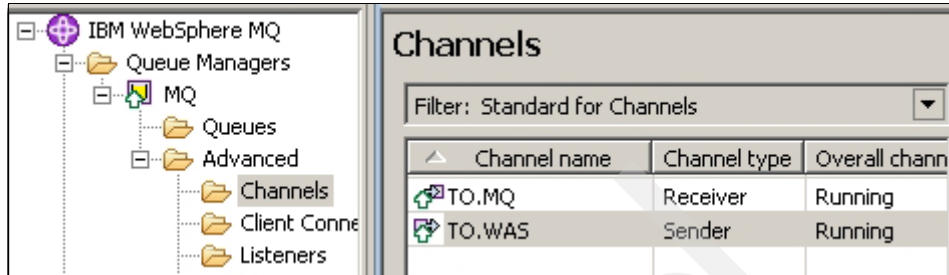
Figure 5-27   Channel status in WebSphere MQ Explorer

5.2.4 Test the connection

You can test the connection by sending a test message from WebSphere MQ to the WebSphere Application Server using the sample program RFHUtil:

1. Start RFHUtil.

2. Select **MQ** in the Queue Manager field.

3. Select **SOURCE** in the Queue Name field.

4. Click the **Read File** button to select the XML file Airline1.xml (see "Sample XML files" on page 270).

5. Click the **Write Q** button to send the message.
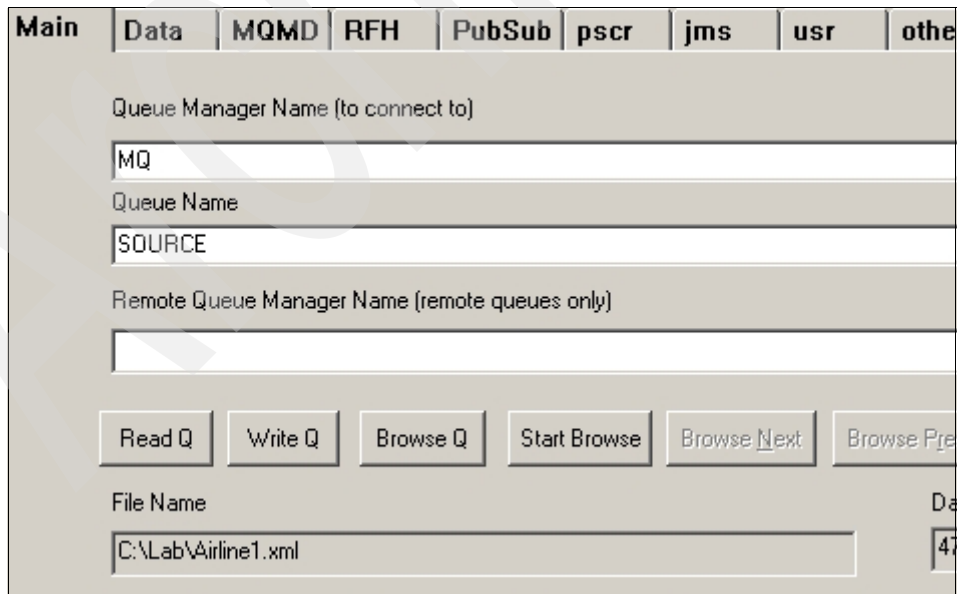
Figure 5-28   Put a test message on the queue using RFHUtil

Check that the message is delivered to WebSphere Application Server:

1. In the WebSphere administrative console, select **Services Integration** → **Buses**.

2. Click the bus name.

3. Click **Destinations**.

4. Click the destination name **TARGET**.

5. Click **Queue Points**.

6. Click the queue point name.

7. Select the **Runtime** tab. You should see that the current message depth is 1.
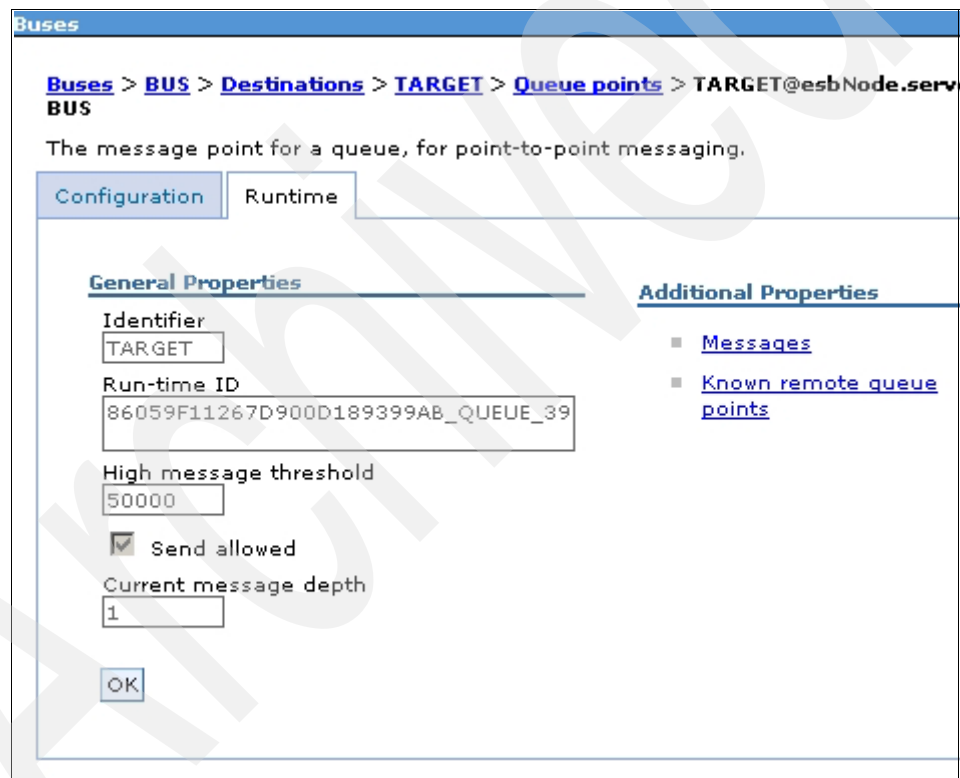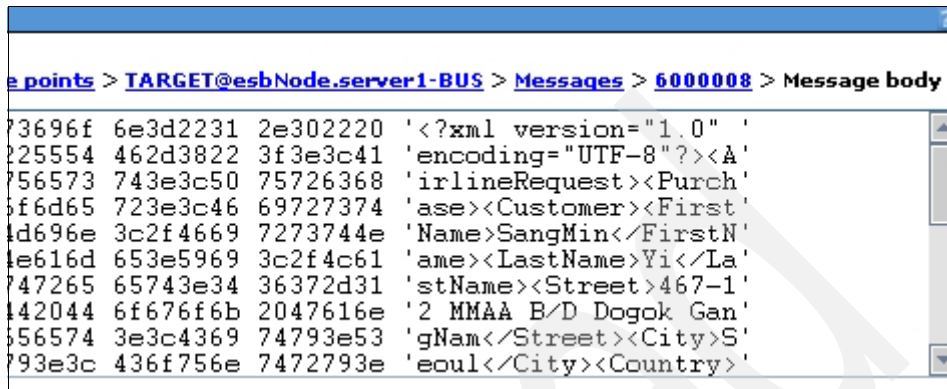
Figure 5-29   Queue point for TARGET queue in the bus

8. Click **Messages**.

9. Click the message identifier number.

10. You will see the JMS message properties and runtime message properties.

11. Click **Message Body**. You should see the Airline1.xml message you sent
    through WebSphere MQ.

Figure 5-30   Delivered message body

Next, try a request/reply test:

1. Start RFHUtil.

2. Select **MQ** in the Queue Manager field.

3. Select **REQUEST** in the Queue Name field.

4. Click the **Read File** button to read the same XML file.

5. Click **Write Q** to send the message.
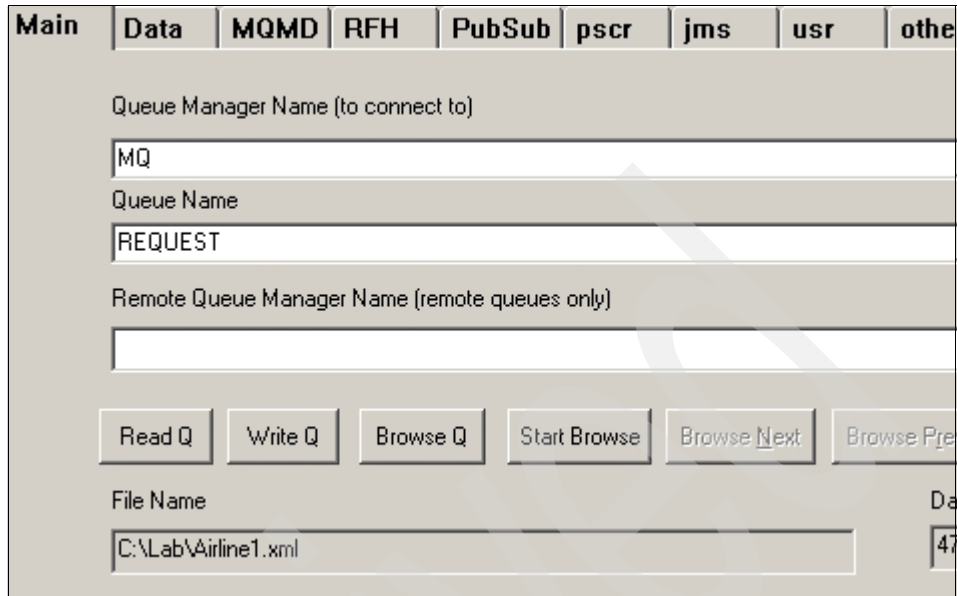
Figure 5-31   Send a test message using RFHUtil

The sample program:

- Puts the message on the REQUEST queue in MQ.

- The message is then transferred to the REPLY queue in WAS through the TO.WAS channel.

- But the REPLY queue in WAS is an alias queue targeting the RESULT queue in MQ. So this message is sent to the RESULT queue.

So you should be able to see the message in the RESULT queue in MQ.
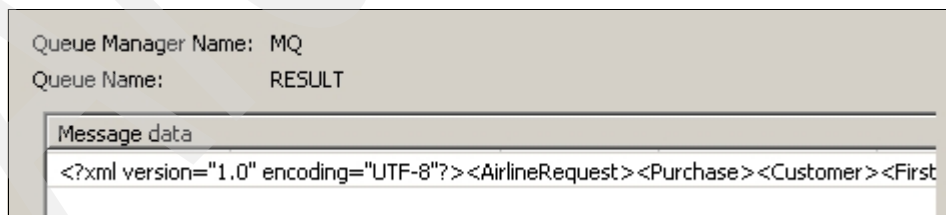
Figure 5-32   Result of the request/reply test

To use RFHUTIL to read the data from the RESULT queue:

1. Start RFHUtil.

2. Select **MQ** in the Queue Manager field.

3. Select **RESULT** in the Queue Name field.

4. Click the **Read Q** button to read the same XML file.

5. Switch to the Data tab to see the message.

5.3 Configuring a queue sharing group

In this section we describe how to configure WebSphere MQ to use a queue sharing group in a sysplex, and list the tasks needed to define Q1.QUE and Q2.QUE shared queues:

1. Set up the DB2 environment to support MQ shared queue.
2. Set up the CFRM policy with the MQ structures.
3. Add the MQ data sharing group entry to the DB2 table.
4. Update the ZPARM.
5. Add the sample queue sharing definitions to the queue manager procedures.
6. Define the shared queues between the two MQ subsystems.

Before we started, MQQ1 and MQQ2 were individual queue managers running on separate z/OS systems in our sysplex. After we finish the customization, MQQ1 and MQQ2 will fully participate in a queue sharing group, as shown in Figure 5-33.

Figure 5-33   z/OS QSG

5.3.1 Set up the DB2 environment to support MQ shared queue

The following jobs must be executed to define the MQ environment on the DB2 data-sharing group:

► Create the storage group.
► Create the database.

- ▶ Create the tablespaces.
- ▶ Create the DB2 tables and associated indexes.
- ▶ Bind the DB2 plans.
- ▶ Grant execute authority.

Create the storage group

The first job, CSQ45CSG, is used to create the storage group that is to be used for the WebSphere MQ database, tablespaces, and tables.

Example 5-1   Create the storage group - CSQ45CSG

```
//MQQGCSG JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//****************************************************************
//*                IBM WebSphere MQ for z/OS                    *
//*                                                             *
//* Sample job to create the DB2 storage group used by WebSphere MQ*
//* using the DB2 TSO batch interface.                          *
//*                                                             *
//****************************************************************
//*                                                             *
//* MORE INFORMATION - See:                                     *
//*   "WebSphere MQ for z/OS System Setup Guide"                *
//*     for information about this customization job            *
//*                                                             *
//****************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//****************************************************************
//*
//CREATESG EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   DSN SYSTEM(D8QG)
   RUN PROGRAM(DSNTIAD) -
       PLAN(DSNTIA81)   -
       LIB('DB8QU.RUNLIB.LOAD')
/*
//SYSIN    DD *
  CREATE STOGROUP "CSQQ1Q2"
       VOLUMES('TOTDB9') VCAT DB8QU;
/*
//
```
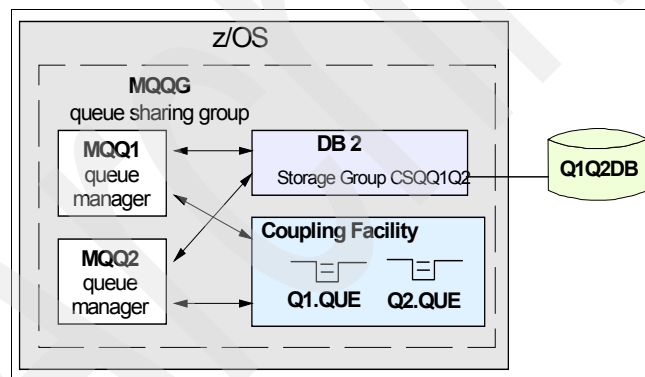
Create the database

Job CSQ45CDB is used to create the database to be used by all queue managers that will connect to this DB2 data-sharing group.

Example 5-2   Create the database - CSQ45CDB

```
//MQQGCDB JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*****************************************************************
//*                    IBM WebSphere MQ for z/OS                 *
//*                                                              *
//* Sample job to create the DB2 database used by WebSphere MQ   *
//* using the DB2 TSO batch interface.                           *
//*                                                              *
//*****************************************************************
//*                                                              *
//* MORE INFORMATION - See:                                      *
//*   "WebSphere MQ for z/OS System Setup Guide"                 *
//*     for information about this customization job             *
//*                                                              *
//*****************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//*****************************************************************
//*
//CREATEDB EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(D8QG)
    RUN PROGRAM(DSNTIAD) -
        PLAN(DSNTIA81)   -
        LIB('DB8QU.RUNLIB.LOAD')
/*
//SYSIN    DD *
  CREATE DATABASE "Q1Q2DB"
  BUFFERPOOL BP32K1
  STOGROUP CSQQ1Q2;
/*
//
```

Create the tablespaces

This creates the tablespaces that will contain the queue manager and channel initiator tables used for queue-sharing groups.

Example 5-3   Create the tablespaces - CSQ45CTS

```
//MQQGCTS JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*****************************************************************
//*              IBM WebSphere MQ for z/OS                        *
//*                                                               *
//* Sample job to create the DB2 tablespaces used by WebSphere MQ *
//* using the DB2 TSO batch interface.                            *
//*                                                               *
//*****************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//*****************************************************************
//*
//CREATETS EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   DSN SYSTEM(D8QG)
   RUN PROGRAM(DSNTIAD) -
       PLAN(DSNTIA81)  -
       LIB('DB8QU.RUNLIB.LOAD')
/*
//SYSIN    DD *

  CREATE TABLESPACE "CQMGR4K"
  USING STOGROUP CSQQ1Q2
  PRIQTY 1024
  SECQTY 4096
  PCTFREE 20
  SEGSIZE 64
  BUFFERPOOL BP2
  LOCKSIZE ANY
  CLOSE NO
  IN Q1Q2DB;

  CREATE TABLESPACE "CQMGR32K"
  USING STOGROUP CSQQ1Q2
  PRIQTY 1024
  SECQTY 4096
  BUFFERPOOL BP32K2
  LOCKSIZE ANY
  CLOSE NO
  IN Q1Q2DB;

  CREATE TABLESPACE "CCHIN"
  USING STOGROUP CSQQ1Q2
```

```
PRIQTY 1024
SECQTY 4096
FREEPAGE 10
PCTFREE  30
SEGSIZE 64
BUFFERPOOL BP1
LOCKSIZE ANY
CLOSE NO
IN Q1Q2DB;

CREATE TABLESPACE "CLOBMB4K"
USING STOGROUP CSQQ1Q2
BUFFERPOOL BP7
NUMPARTS 4
LOCKSIZE ANY
CLOSE NO
IN Q1Q2DB;

CREATE LOB TABLESPACE "CLOB132K"
IN Q1Q2DB
USING STOGROUP CSQQ1Q2
PRIQTY 7200
SECQTY 7200
LOCKSIZE LOB
GBPCACHE SYSTEM
BUFFERPOOL BP32K3
LOG NO
CLOSE NO;

CREATE LOB TABLESPACE "CLOB232K"
IN Q1Q2DB
USING STOGROUP CSQQ1Q2
PRIQTY 7200
SECQTY 7200
LOCKSIZE LOB
GBPCACHE SYSTEM
BUFFERPOOL BP32K3
LOG NO
CLOSE NO;

CREATE LOB TABLESPACE "CLOB332K"
IN Q1Q2DB
USING STOGROUP CSQQ1Q2
PRIQTY 7200
SECQTY 7200
LOCKSIZE LOB
GBPCACHE SYSTEM
BUFFERPOOL BP32K3
LOG NO
```

```
   CLOSE NO;

   CREATE LOB TABLESPACE "CLOB432K"
   IN Q1Q2DB
   USING STOGROUP CSQQ1Q2
   PRIQTY 7200
   SECQTY 7200
   LOCKSIZE LOB
   GBPCACHE SYSTEM
   BUFFERPOOL BP32K3
   LOG NO
   CLOSE NO;

/*
//
```

Create the DB2 tables and associated indexes

Job CSQ45CTB is used to create the twelve DB2 tables and associated indexes.

Example 5-4   Create the DB2 tables - CSQ45CTB

```
//MQQGCTB JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*******************************************************************
//*                IBM WebSphere MQ for z/OS                        *
//*                                                                 *
//* Sample job to create the DB2 tables used by WebSphere MQ        *
//* using the DB2 TSO batch interface.                              *
//*                                                                 *
//*******************************************************************
//*                                                                 *
//* MORE INFORMATION - See:                                         *
//*   "WebSphere MQ for z/OS System Setup Guide"                    *
//*     for information about this customization job                *
//*   "WebSphere MQ for z/OS Concepts and Planning Guide"           *
//*     for information about DB2 table sizes                       *
//*                                                                 *
//*******************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//*******************************************************************
//*
//CREATETB EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   DSN SYSTEM(D8QG)
```

```
            RUN PROGRAM(DSNTIAD) -
                PLAN(DSNTIA81)   -
                LIB('DB8QU.RUNLIB.LOAD')
        /*
        //SYSIN    DD *
          CREATE TABLE CSQ.ADMIN_B_QSG
                (
                QSGNAME        CHAR(4)  NOT NULL   ,
                ARRAY_QMGR     CHAR(32)            ,
                ARRAY_STRUC    CHAR(64)            ,
                PRODLVL        CHAR(3)  NOT NULL   ,
                VERSIONCOUNT   INT                 ,
                UPDT_QMGR      CHAR(48)            ,
                UPDT_QMGRNUM   SMALLINT            ,
                UPDT_STAMP     CHAR(8)             ,
                CREATE_QMGR    CHAR(48)            ,
                CREATE_QMGRNUM SMALLINT            ,
                CREATE_STAMP   CHAR(8)             ,
                RECON_STAMP    CHAR(8)  WITH DEFAULT X'00'  ,
                RECON_QMGRNUM  SMALLINT WITH DEFAULT 0      ,
                PRIMARY KEY (QSGNAME)
                )
                IN Q1Q2DB.CQMGR4K;

          CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_QSG
                ON CSQ.ADMIN_B_QSG (QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                CLOSE NO;

          CREATE TABLE CSQ.ADMIN_B_QMGR
                (
                QMGRNAME       CHAR(48) NOT NULL   ,
                QSGNAME        CHAR(4) NOT NULL    ,
                QMGRNUM        SMALLINT            ,
                ACTSTATE       CHAR(1)             ,
                DESCR          CHAR(64)            ,
                PLATFORM       CHAR(10)            ,
                IGQAUT         CHAR(10)            ,
                CPILEVEL       INT                 ,
                CMDLEVEL       INT                 ,
                CCSID          INT                 ,
                MAXPRTY        INT                 ,
                MAXMSGL        INT                 ,
                SYNCPT         CHAR(10)            ,
                COMMANDQ       CHAR(48)            ,
                DEADQ          CHAR(48)            ,
                TRIGINT        INT                 ,
```

```
            MAXHANDS     INT               ,
            AUTHOREV     CHAR(10)          ,
            INHIBTEV     CHAR(10)          ,
            LOCALEV      CHAR(10)          ,
            REMOTEEV     CHAR(10)          ,
            STRSTPEV     CHAR(10)          ,
            PERFMEV      CHAR(10)          ,
            CHAD         CHAR(10)          ,
            CHADEXIT     CHAR(8)           ,
            CLWLDATA     CHAR(32)          ,
            CLWLEXIT     CHAR(8)           ,
            REPOS        CHAR(48)          ,
            REPOSNL      CHAR(48)          ,
            QMID         CHAR(48)          ,
            DEFXMITQ     CHAR(48)          ,
            VERSIONCOUNT INT               ,
            MVERSIONL    INT               ,
            MVERSIONH    INT               ,
            QSGCREATE    CHAR(8)           ,
            UPDT_QMGR    CHAR(48)          ,
            UPDT_QMGRNUM SMALLINT          ,
            UPDT_STAMP   CHAR(8)           ,
            CREATE_QMGR  CHAR(48)          ,
            CREATE_QMGRNUM SMALLINT        ,
            CREATE_STAMP CHAR(8)           ,
            VSOBJECT     VARCHAR(2560)     ,
            BSDS_NAME1   CHAR(44) WITH DEFAULT ' ',
            BSDS_NAME2   CHAR(44) WITH DEFAULT ' ',
            BSDS_STATUS1 CHAR(1) WITH DEFAULT X'00',
            BSDS_STATUS2 CHAR(1) WITH DEFAULT X'00',
            CONFIGEV     CHAR(10) WITH DEFAULT 'DISABLED',
            MAXUMSGS     INT WITH DEFAULT 10000,
            SSLTASKS     INT WITH DEFAULT 0   ,
            SSLCRLNL     CHAR(48) WITH DEFAULT ' ',
            SSLKEYR      VARCHAR(256) WITH DEFAULT ' ',
            EXPRYINT     INT WITH DEFAULT 0   ,
            PRIMARY KEY (QMGRNAME),
            FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
            ON DELETE RESTRICT
            )
            IN Q1Q2DB.CQMGR4K;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_QMGR
            ON CSQ.ADMIN_B_QMGR (QMGRNAME ASC)
            USING STOGROUP CSQQ1Q2
            PRIQTY 512
            SECQTY 1024
            CLOSE NO;
```

```
CREATE TABLE CSQ.EXTEND_B_QMGR
    (
    QMGRNAME    CHAR(48) NOT NULL            ,
    QSGNAME     CHAR(4) NOT NULL             ,
    ACCTQ       CHAR(10) WITH DEFAULT 'ON'   ,
    MONQ        CHAR(10) WITH DEFAULT 'OFF'  ,
    CHLEV       CHAR(10) WITH DEFAULT 'ENABLED',
    BRIDGEEV    CHAR(10) WITH DEFAULT 'ENABLED',
    SSLEV       CHAR(10) WITH DEFAULT 'ENABLED',
    CMDEV       CHAR(10) WITH DEFAULT 'DISABLED',
    IPADDRV     CHAR(10) WITH DEFAULT 'IPV4' ,
    ACTCHL      INT     WITH DEFAULT 200     ,
    ADOPTCHK    CHAR(10) WITH DEFAULT 'ALL'  ,
    ADOPTMCA    CHAR(10) WITH DEFAULT 'NO'   ,
    CHIADAPS    INT     WITH DEFAULT 8       ,
    CHIDISPS    INT     WITH DEFAULT 5       ,
    CHISERVP    CHAR(32) WITH DEFAULT X'00'  ,
    DNSGROUP    CHAR(18) WITH DEFAULT ' '    ,
    DNSWLM      CHAR(10) WITH DEFAULT 'NO'   ,
    LSTRTMR     SMALLINT WITH DEFAULT 60     ,
    LUGROUP     CHAR(8)  WITH DEFAULT ' '    ,
    LUNAME      CHAR(8)  WITH DEFAULT ' '    ,
    LU62ARM     CHAR(2)  WITH DEFAULT ' '    ,
    LU62CHL     INT     WITH DEFAULT 200     ,
    MAXCHL      INT     WITH DEFAULT 200     ,
    OPORTMIN    INT     WITH DEFAULT 0       ,
    OPORTMAX    INT     WITH DEFAULT 0       ,
    RCVTIME     INT     WITH DEFAULT 0       ,
    RCVTTYPE    CHAR(10) WITH DEFAULT 'MULTIPLY',
    RCVTMIN     INT     WITH DEFAULT 0       ,
    TCPCHL      INT     WITH DEFAULT 200     ,
    TCPKEEP     CHAR(10) WITH DEFAULT 'NO'   ,
    TCPNAME     CHAR(8)  WITH DEFAULT 'TCPIP' ,
    TCPSTACK    CHAR(10) WITH DEFAULT 'SINGLE',
    TRAXSTR     CHAR(10) WITH DEFAULT 'YES'  ,
    TRAXTBL     INT     WITH DEFAULT 2       ,
    SSLRKEYC    INT     WITH DEFAULT 0       ,
    SQQMNAME    CHAR(10) WITH DEFAULT 'USE'  ,
    MONACLS     CHAR(10) WITH DEFAULT 'QMGR' ,
    MONCHL      CHAR(10) WITH DEFAULT 'OFF'  ,
    CLWLMRUC    INT     WITH DEFAULT 999999999,
    CLWLUSEQ    CHAR(10) WITH DEFAULT 'LOCAL' ,
    ROUTEREC    CHAR(10) WITH DEFAULT 'MSG'  ,
    ACTIVREC    CHAR(10) WITH DEFAULT 'MSG'  ,
    PRIMARY KEY (QMGRNAME),
    FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
            ON DELETE RESTRICT
    )
    IN Q1Q2DB.CQMGR4K;
```

```
CREATE TYPE 2 UNIQUE INDEX CSQ.EXTEND_QMGR
       ON CSQ.EXTEND_B_QMGR (QMGRNAME ASC)
       USING STOGROUP CSQQ1Q2
       PRIQTY 512
       SECQTY 1024
       CLOSE NO;

CREATE TABLE CSQ.ADMIN_B_STRUCTURE
       (
       STRUCNAME    CHAR(12) NOT NULL   ,
       QSGNAME      CHAR(4)  NOT NULL   ,
       LH_ARRAY     CHAR(64)            ,
       STRUC_INTRST CHAR(32)            ,
       STRUCNUM     SMALLINT            ,
       PRODLVL      CHAR(3)  NOT NULL   ,
       CFSTATUS     CHAR(1)             ,
       VERSIONCOUNT INT                 ,
       MVERSION     INT                 ,
       UPDT_QMGR    CHAR(48)            ,
       UPDT_QMGRNUM SMALLINT            ,
       UPDT_STAMP   CHAR(8)             ,
       CREATE_QMGR  CHAR(48)            ,
       CREATE_QMGRNUM SMALLINT          ,
       CREATE_STAMP CHAR(8)             ,
       RECOVER      CHAR(1) WITH DEFAULT 'N',
       DESCR        CHAR(64) WITH DEFAULT ' ',
       PRIMARY KEY (STRUCNAME, QSGNAME) ,
       FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
       ON DELETE CASCADE
       )
       IN Q1Q2DB.CQMGR4K;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_STRUCTURE
       ON CSQ.ADMIN_B_STRUCTURE (STRUCNAME ASC, QSGNAME ASC)
       USING STOGROUP CSQQ1Q2
       PRIQTY 512
       SECQTY 1024
       CLOSE NO;

CREATE TABLE CSQ.ADMIN_B_STRBACKUP
       (
       STRUCNAME    CHAR(12) NOT NULL   ,
       QSGNAME      CHAR(4)  NOT NULL   ,
       QMGRNAME     CHAR(48)            ,
       STRUCNUM     SMALLINT            ,
       FAIL_STAMP   CHAR(8)             ,
       BSTART_RBA   CHAR(6)             ,
       BEND_RBA     CHAR(6)             ,
```

```
                BSTART_STAMP CHAR(8)              ,
                BEND_STAMP   CHAR(8)              ,
                VERSIONCOUNT INT                 ,
                MVERSION     INT                 ,
                UPDT_QMGR    CHAR(48)             ,
                UPDT_QMGRNUM SMALLINT             ,
                UPDT_STAMP   CHAR(8)              ,
                CREATE_QMGR  CHAR(48)             ,
                CREATE_QMGRNUM SMALLINT           ,
                CREATE_STAMP CHAR(8)              ,
                STRUC_INTRST VARCHAR(256)         ,
                PRIMARY KEY (STRUCNAME, QSGNAME)  ,
                FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
                ON DELETE CASCADE
                )
                IN Q1Q2DB.CQMGR4K;

        CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_STRBACKUP
                ON CSQ.ADMIN_B_STRBACKUP
                (STRUCNAME ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                CLOSE NO;

        CREATE TABLE CSQ.OBJ_B_QUEUE
                (
                QNAME          CHAR(48) NOT NULL   ,
                QSGNAME        CHAR(4)  NOT NULL   ,
                QTYPE          CHAR(8)             ,
                CFCACHE        CHAR(1)             ,
                DELCOMMIT      CHAR(1)             ,
                CLUSTER        CHAR(48)            ,
                CLUSNL         CHAR(48)            ,
                DEFBIND        CHAR(10)            ,
                DEFPRTY        INT                 ,
                DEFPSIST       CHAR(10)            ,
                QSGDISP        CHAR(10)            ,
                DEFTYPE        CHAR(10)            ,
                DESCR          CHAR(64)            ,
                PUT            CHAR(10)            ,
                BOQNAME        CHAR(48)            ,
                BOTHRESH       INT                 ,
                DEFSOPT        CHAR(10)            ,
                GET            CHAR(10)            ,
                HARDENBO       CHAR(10)            ,
                INDXTYPE       CHAR(10)            ,
                INITQ          CHAR(48)            ,
                LHNUMBER       SMALLINT            ,
```

```
            MAXDEPTH        INT                     ,
            MAXMSGL         INT                     ,
            MSGDLVSQ        CHAR(10)                ,
            PROCESS         CHAR(48)                ,
            QDEPTHHI        INT                     ,
            QDEPTHLO        INT                     ,
            QDPHIEV         CHAR(10)                ,
            QDPLOEV         CHAR(10)                ,
            QDPMAXEV        CHAR(10)                ,
            QSVCIEV         CHAR(10)                ,
            QSVCINT         INT                     ,
            RETINTVL        INT                     ,
            RNAME           CHAR(48)                ,
            RQMNAME         CHAR(48)                ,
            SHARE           CHAR(7)                 ,
            STGCLASS        CHAR(8)                 ,
            STRUCNAME       CHAR(12)                ,
            STRUCNUM        SMALLINT                ,
            TARGQ           CHAR(48)                ,
            TRIGDATA        CHAR(64)                ,
            TRIGDPTH        INT                     ,
            TRIGGER         CHAR(9)                 ,
            TRIGMPRI        INT                     ,
            TRIGTYPE        CHAR(10)                ,
            USAGE           CHAR(10)                ,
            XMITQ           CHAR(48)                ,
            VERSIONCOUNT    INT                     ,
            MVERSION        INT                     ,
            STRUCSTAMP      CHAR(8)                 ,
            UPDT_QMGR       CHAR(48)                ,
            UPDT_QMGRNUM    SMALLINT                ,
            UPDT_STAMP      CHAR(8)                 ,
            CREATE_QMGR     CHAR(48)                ,
            CREATE_QMGRNUM  SMALLINT                ,
            CREATE_STAMP    CHAR(8)                 ,
            VSOBJECT        VARCHAR(2560)           ,
            ACCTQ           CHAR(10) WITH DEFAULT 'ON',
            MONQ            CHAR(10) WITH DEFAULT 'QMGR',
            CLWLRANK        INT     WITH DEFAULT 0,
            CLWLPRTY        INT     WITH DEFAULT 0,
            CLWLUSEQ        CHAR(10) WITH DEFAULT 'QMGR',
            NPMCLASS        CHAR(10) WITH DEFAULT 'NORMAL',
            PRIMARY KEY (QNAME, QSGNAME)            ,
            FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
            ON DELETE CASCADE
            )
            IN Q1Q2DB.CQMGR4K;

CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_QUEUE_IX1
```

```
                ON CSQ.OBJ_B_QUEUE (QNAME ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                FREEPAGE 10
                PCTFREE  20
                CLOSE NO;

        CREATE TYPE 2 INDEX CSQ.OBJ_QUEUE_IX2
                ON CSQ.OBJ_B_QUEUE
                (QSGNAME ASC, QSGDISP ASC, CREATE_STAMP ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                FREEPAGE 10
                PCTFREE  20
                CLOSE NO;

        CREATE TABLE CSQ.OBJ_B_PROCESS
                (
                PROCNAME      CHAR(48) NOT NULL   ,
                QSGNAME       CHAR(4)  NOT NULL   ,
                DESCR         CHAR(64) NOT NULL   ,
                QSGDISP       CHAR(10)            ,
                APPLTYPE      CHAR(4)             ,
                APPLICID      VARCHAR(256)        ,
                ENVRDATA      CHAR(128)           ,
                USERDATA      CHAR(128)           ,
                VERSIONCOUNT INT                  ,
                UPDT_QMGR     CHAR(48)            ,
                UPDT_QMGRNUM SMALLINT             ,
                UPDT_STAMP    CHAR(8)             ,
                CREATE_QMGR  CHAR(48)             ,
                CREATE_QMGRNUM SMALLINT           ,
                CREATE_STAMP CHAR(8)              ,
                VSOBJECT      VARCHAR(2560)       ,
                PRIMARY KEY (PROCNAME, QSGNAME)   ,
                FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
                ON DELETE CASCADE
                )
                IN Q1Q2DB.CQMGR4K;

        CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_PROCESS
                ON CSQ.OBJ_B_PROCESS (PROCNAME ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                CLOSE NO;
```

```
CREATE TABLE CSQ.OBJ_B_STGCLASS
        (
        STGCNAME     CHAR(48) NOT NULL   ,
        QSGNAME      CHAR(4)  NOT NULL   ,
        DESCR        CHAR(64) NOT NULL   ,
        QSGDISP      CHAR(10)            ,
        PSID         INT                 ,
        XCFGNAME     CHAR(8)             ,
        XCFMNAME     CHAR(16)            ,
        VERSIONCOUNT INT                 ,
        UPDT_QMGR    CHAR(48)            ,
        UPDT_QMGRNUM SMALLINT            ,
        UPDT_STAMP   CHAR(8)             ,
        CREATE_QMGR  CHAR(48)            ,
        CREATE_QMGRNUM SMALLINT          ,
        CREATE_STAMP CHAR(8)            ,
        VSOBJECT     VARCHAR(2560)       ,
        PASSTKTA     CHAR(8)       WITH DEFAULT ' ',
        PRIMARY KEY (STGCNAME, QSGNAME)  ,
        FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
        ON DELETE CASCADE
        )
        IN Q1Q2DB.CQMGR4K;

CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_STGCLASS
        ON CSQ.OBJ_B_STGCLASS (STGCNAME ASC, QSGNAME ASC)
        USING STOGROUP CSQQ1Q2
        PRIQTY 512
        SECQTY 1024
        CLOSE NO;

CREATE TABLE CSQ.OBJ_B_NAMELIST
        (
        NLNAME       CHAR(48) NOT NULL   ,
        QSGNAME      CHAR(4)  NOT NULL   ,
        DESCR        CHAR(64) NOT NULL   ,
        QSGDISP      CHAR(10)            ,
        VERSIONCOUNT INT                 ,
        UPDT_QMGR    CHAR(48)            ,
        UPDT_QMGRNUM SMALLINT            ,
        UPDT_STAMP   CHAR(8)             ,
        CREATE_QMGR  CHAR(48)            ,
        CREATE_QMGRNUM SMALLINT          ,
        CREATE_STAMP CHAR(8)            ,
        VSOBJECT     VARCHAR(2560)       ,
        NAMES        VARCHAR(12288)      ,
        NLTYPE       CHAR(10)  WITH DEFAULT 'NONE',
        PRIMARY KEY (NLNAME, QSGNAME)    ,
        FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
```

```
            ON DELETE CASCADE
            )
            IN Q1Q2DB.CQMGR32K;

CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_NAMELIST
            ON CSQ.OBJ_B_NAMELIST (NLNAME ASC, QSGNAME ASC)
            USING STOGROUP CSQQ1Q2
            PRIQTY 512
            SECQTY 1024
            CLOSE NO;

CREATE TABLE CSQ.OBJ_B_CHANNEL
            (
            CHLNAME      CHAR(48) NOT NULL   ,
            QSGNAME      CHAR(4)  NOT NULL   ,
            CHLTYPE      CHAR(10) NOT NULL   ,
            DESCR        CHAR(64) NOT NULL   ,
            QSGDISP      CHAR(10)            ,
            TRPTYPE      CHAR(8)             ,
            CLUSTER      CHAR(48)            ,
            CLUSNL       CHAR(48)            ,
            XMITQ        CHAR(48)            ,
            BATCHINT     INT                 ,
            BATCHSZ      INT                 ,
            CONVERT      CHAR(10)            ,
            DISCINT      INT                 ,
            HBINT        INT                 ,
            LONGRTY      INT                 ,
            LONGTMR      INT                 ,
            MAXMSGL      INT                 ,
            MCANAME      CHAR(20)            ,
            MCAUSER      CHAR(12)            ,
            MODENAME     CHAR(8)             ,
            MSGDATA      CHAR(32)            ,
            MSGEXIT      CHAR(128)           ,
            NETPRTY      INT                 ,
            NPMSPEED     CHAR(10)            ,
            PASSWORD     CHAR(12)            ,
            PUTAUT       CHAR(10)            ,
            QMNAME       CHAR(48)            ,
            RCVDATA      CHAR(32)            ,
            RCVEXIT      CHAR(128)           ,
            SCYDATA      CHAR(32)            ,
            SCYEXIT      CHAR(128)           ,
            SENDDATA     CHAR(32)            ,
            SENDEXIT     CHAR(128)           ,
            SEQWRAP      INT                 ,
            SHORTRTY     INT                 ,
            SHORTTMR     INT                 ,
```

```
            TPNAME       CHAR(64)              ,
            USERID       CHAR(12)              ,
            VERSIONCOUNT INT                   ,
            UPDT_QMGR    CHAR(48)              ,
            UPDT_QMGRNUM SMALLINT              ,
            UPDT_STAMP   CHAR(8)               ,
            CREATE_QMGR  CHAR(48)              ,
            CREATE_QMGRNUM SMALLINT            ,
            CREATE_STAMP CHAR(8)               ,
            CONNAME      VARCHAR(264)          ,
            VSOBJECT     VARCHAR(8190)         ,
            BATCHHB      INT WITH DEFAULT 0    ,
            KAINT        INT WITH DEFAULT 120,
            LOCLADDR     CHAR(48) WITH DEFAULT ' ',
            MSGEXITS     VARCHAR(1024)  WITH DEFAULT ' ',
            MSGDATAS     VARCHAR(256)   WITH DEFAULT ' ',
            SENDEXITS    VARCHAR(1024)  WITH DEFAULT ' ',
            SENDDATAS    VARCHAR(256)   WITH DEFAULT ' ',
            RCVEXITS     VARCHAR(1024)  WITH DEFAULT ' ',
            RCVDATAS     VARCHAR(256)   WITH DEFAULT ' ',
            SSLCAUTH     CHAR(10)       WITH DEFAULT 'REQUIRED',
            SSLCIPH      CHAR(32)       WITH DEFAULT ' ',
            SSLPEER      VARCHAR(256)   WITH DEFAULT ' ',
            MREXIT       CHAR(8)        WITH DEFAULT ' ',
            MRDATA       CHAR(32)       WITH DEFAULT ' ',
            MRRTY        INT            WITH DEFAULT 0,
            MRTMR        INT            WITH DEFAULT 1000,
            COMPHDR      CHAR(10)       WITH DEFAULT 'NONE',
            COMPMSG      CHAR(10)       WITH DEFAULT 'NONE',
            MONCHL       CHAR(10)       WITH DEFAULT 'QMGR',
            CLWLRANK     INT            WITH DEFAULT 0,
            CLWLPRTY     INT            WITH DEFAULT 0,
            CLWLWGHT     INT            WITH DEFAULT 50,
            PRIMARY KEY (CHLNAME, QSGNAME)     ,
            FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
            ON DELETE CASCADE
            )
            IN Q1Q2DB.CQMGR32K;

CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_CHANNEL
            ON CSQ.OBJ_B_CHANNEL (CHLNAME ASC, QSGNAME ASC)
            USING STOGROUP CSQQ1Q2
            PRIQTY 512
            SECQTY 1024
            CLOSE NO;

CREATE TYPE 2 UNIQUE INDEX CSQ.DEL_OBJ_CHANNEL
            ON CSQ.OBJ_B_CHANNEL (CHLNAME ASC, CHLTYPE ASC, QSGNAME ASC)
            USING STOGROUP CSQQ1Q2
```

```
                PRIQTY 512
                SECQTY 1024
                CLOSE NO;

CREATE TABLE CSQ.ADMIN_B_SCST
                (
                CHLCRESTAMP      CHAR(10) NOT NULL        ,
                CHLUPDSTAMP      CHAR(10) NOT NULL        ,
                XMITQ            CHAR(48) NOT NULL        ,
                CHLNAME          CHAR(20) NOT NULL        ,
                REMOTEQMGR       CHAR(48) NOT NULL        ,
                QSGNAME          CHAR(4)  NOT NULL        ,
                OWNINGQMGR       CHAR(4)  NOT NULL        ,
                CHLTOKEN         CHAR(20) NOT NULL        ,
                REMOTEMACH       CHAR(20)                 ,
                CHANNELSTATUS    CHAR(1)                  ,
                CHANNELTYPE      INT                      ,
                LONGRETRYCOUNT   INT                      ,
                SHORTRETRYCOUNT  INT                      ,
                NEXTRETRYTIME    INT                      ,
                DATA             CHAR(4) FOR BIT DATA     ,
                UPDT_QMGR        CHAR(48)                 ,
                UPDT_QMGRNUM     SMALLINT                 ,
                UPDT_STAMP       CHAR(8)                  ,
                CREATE_QMGR      CHAR(48)                 ,
                CREATE_QMGRNUM   SMALLINT                 ,
                CREATE_STAMP     CHAR(8)                  ,
                PRIMARY KEY (XMITQ, CHLNAME, REMOTEQMGR, QSGNAME) ,
                FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
                ON DELETE CASCADE
                )
                IN Q1Q2DB.CCHIN;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SCST_IX1
                ON CSQ.ADMIN_B_SCST
                (XMITQ ASC, CHLNAME ASC, REMOTEQMGR ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                FREEPAGE 5
                PCTFREE  30
                CLOSE NO;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SCST_IX2
                ON CSQ.ADMIN_B_SCST
                (CHLCRESTAMP ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
```

```
                  FREEPAGE 5
                  PCTFREE  30
                  CLOSE NO;

CREATE TYPE 2 INDEX CSQ.ADMIN_SCST_IX3
                  ON CSQ.ADMIN_B_SCST
                  (QSGNAME ASC, CHLNAME ASC, CHANNELSTATUS ASC)
                  USING STOGROUP CSQQ1Q2
                  PRIQTY 512
                  SECQTY 1024
                  FREEPAGE 5
                  PCTFREE  30
                  CLOSE NO;

CREATE TYPE 2 INDEX CSQ.ADMIN_SCST_IX4
                  ON CSQ.ADMIN_B_SCST
                  (QSGNAME ASC, CHLCRESTAMP ASC, CHLUPDSTAMP ASC)
                  USING STOGROUP CSQQ1Q2
                  PRIQTY 512
                  SECQTY 1024
                  FREEPAGE 5
                  PCTFREE  30
                  CLOSE NO;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SCST_IX5
                  ON CSQ.ADMIN_B_SCST
                  (QSGNAME ASC, CHLTOKEN ASC)
                  USING STOGROUP CSQQ1Q2
                  PRIQTY 512
                  SECQTY 1024
                  FREEPAGE 5
                  PCTFREE  30
                  CLOSE NO;

CREATE TABLE CSQ.ADMIN_B_SSKT
                  (
                  XMITQ          CHAR(48) NOT NULL          ,
                  CHLNAME        CHAR(20) NOT NULL          ,
                  REMOTEQMGR     CHAR(48) NOT NULL          ,
                  KEY            INT      NOT NULL          ,
                  QSGNAME        CHAR(4)  NOT NULL          ,
                  UPDT_QMGR      CHAR(48)                   ,
                  UPDT_QMGRNUM   SMALLINT                   ,
                  UPDT_STAMP     CHAR(8)                    ,
                  CREATE_QMGR    CHAR(48)                   ,
                  CREATE_QMGRNUM SMALLINT                   ,
                  CREATE_STAMP   CHAR(8)                    ,
                  PRIMARY KEY (XMITQ, CHLNAME, REMOTEQMGR, QSGNAME) ,
                  FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
```

```
                ON DELETE CASCADE
                )
                IN Q1Q2DB.CCHIN;

        CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_SSKT_IX1
                ON CSQ.ADMIN_B_SSKT
                (XMITQ ASC, CHLNAME ASC, REMOTEQMGR ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                FREEPAGE 5
                PCTFREE  30
                CLOSE NO;

        CREATE TABLE CSQ.OBJ_B_AUTHINFO
                (
                AUTHINFO      CHAR(48) NOT NULL   ,
                QSGNAME       CHAR(4)  NOT NULL   ,
                AUTHTYPE      CHAR(10) NOT NULL   ,
                DESCR         CHAR(64)            ,
                QSGDISP       CHAR(10)            ,
                MVERSION      INT                 ,
                LDAPPWD       CHAR(32)            ,
                LDAPUSER      VARCHAR(256)        ,
                VERSIONCOUNT INT                  ,
                UPDT_QMGR    CHAR(48)             ,
                UPDT_QMGRNUM SMALLINT             ,
                UPDT_STAMP   CHAR(8)              ,
                CREATE_QMGR  CHAR(48)             ,
                CREATE_QMGRNUM SMALLINT           ,
                CREATE_STAMP CHAR(8)              ,
                CONNAME       VARCHAR(264)        ,
                VSOBJECT      VARCHAR(2560)       ,
                PRIMARY KEY (AUTHINFO, QSGNAME)   ,
                FOREIGN KEY (QSGNAME) REFERENCES CSQ.ADMIN_B_QSG
                ON DELETE CASCADE
                )
                IN Q1Q2DB.CQMGR4K;

        CREATE TYPE 2 UNIQUE INDEX CSQ.OBJ_AUTHINFO
                ON CSQ.OBJ_B_AUTHINFO (AUTHINFO ASC, QSGNAME ASC)
                USING STOGROUP CSQQ1Q2
                PRIQTY 512
                SECQTY 1024
                CLOSE NO;

        CREATE TABLE CSQ.ADMIN_B_MESSAGES
                (
                LEID          CHAR(12) NOT NULL   ,
```

```
             QSGNAME     CHAR(4)  NOT NULL   ,
             SEGMENTNUM  SMALLINT NOT NULL   ,
             MROWID      ROWID    NOT NULL GENERATED ALWAYS,
             STRUCNUM    SMALLINT            ,
             LHNUMBER    SMALLINT            ,
             MVERSION    INT                 ,
             BASEPART    CHAR(2)             ,
             MESSAGE     BLOB(512K)          ,
             DELCOMMIT   CHAR(1) WITH DEFAULT 'N'    ,
             RECON_STAMP CHAR(8) WITH DEFAULT X'00'  ,
             PERSISTENT  CHAR(1),
             MSG_STAMP   CHAR(8),
             PRIMARY KEY (LEID, QSGNAME, SEGMENTNUM)
             )
             IN Q1Q2DB.CLOBMB4K;

CREATE TYPE 2 UNIQUE INDEX CSQ.ADMIN_MESSAGES_IX1
        ON CSQ.ADMIN_B_MESSAGES
        (LEID ASC, QSGNAME ASC, SEGMENTNUM ASC)
        USING STOGROUP CSQQ1Q2
        PRIQTY 720
        SECQTY 1024
        CLOSE NO;

CREATE AUX TABLE CSQ.ADMIN_MSGS_BAUX1
        IN Q1Q2DB.CLOB132K
        STORES CSQ.ADMIN_B_MESSAGES
        COLUMN MESSAGE
        PART 1;

CREATE INDEX CSQ.ADMIN_MSGS1
        ON CSQ.ADMIN_MSGS_BAUX1
        USING STOGROUP CSQQ1Q2
        PRIQTY 720
        SECQTY 1024
        CLOSE NO;

CREATE AUX TABLE CSQ.ADMIN_MSGS_BAUX2
        IN Q1Q2DB.CLOB232K
        STORES CSQ.ADMIN_B_MESSAGES
        COLUMN MESSAGE
        PART 2;

CREATE INDEX CSQ.ADMIN_MSGS2
        ON CSQ.ADMIN_MSGS_BAUX2
        USING STOGROUP CSQQ1Q2
        PRIQTY 720
        SECQTY 1024
        CLOSE NO;
```

```
CREATE AUX TABLE CSQ.ADMIN_MSGS_BAUX3
       IN Q1Q2DB.CLOB332K
       STORES CSQ.ADMIN_B_MESSAGES
       COLUMN MESSAGE
       PART 3;

CREATE INDEX CSQ.ADMIN_MSGS3
       ON CSQ.ADMIN_MSGS_BAUX3
       USING STOGROUP CSQQ1Q2
       PRIQTY 720
       SECQTY 1024
       CLOSE NO;

CREATE AUX TABLE CSQ.ADMIN_MSGS_BAUX4
       IN Q1Q2DB.CLOB432K
       STORES CSQ.ADMIN_B_MESSAGES
       COLUMN MESSAGE
       PART 4;

CREATE INDEX CSQ.ADMIN_MSGS4
       ON CSQ.ADMIN_MSGS_BAUX4
       USING STOGROUP CSQQ1Q2
       PRIQTY 720
       SECQTY 1024
       CLOSE NO;

CREATE TYPE 2 INDEX CSQ.CLUS_INDEX
       ON CSQ.ADMIN_B_MESSAGES
       (BASEPART ASC)
       USING STOGROUP CSQQ1Q2
       PRIQTY 720
       SECQTY 1024
       CLUSTER
        (PART 1 VALUES(X'3FFF'),
         PART 2 VALUES(X'7FFF'),
         PART 3 VALUES(X'BFFF'),
         PART 4 VALUES(X'FFFF'))
       CLOSE NO;

/*
//
```

Bind the DB2 plans

Job CSQ45BPL is used to bind the DB2 plans for the queue manager, utilities,
and channel initiator.

Example 5-5 Bind the DB2 plans - CSQ45BPL

```
//MQQGBPL JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*****************************************************************
//*               IBM WebSphere MQ for z/OS                       *
//*                                                               *
//* Sample job to bind the DB2 plans used by WebSphere MQ         *
//* using the DB2 TSO batch interface.                            *
//*                                                               *
//*****************************************************************
//*                                                               *
//* MORE INFORMATION - See:                                       *
//*   "WebSphere MQ for z/OS System Setup Guide"                  *
//*     for information about this customization job              *
//*                                                               *
//*****************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//*****************************************************************
//*
//BINDPLAN EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
   DSN SYSTEM(D8QG)

   BIND PLAN(CSQ5A600) -
   MEM(CSQ5A600) -
   ACQUIRE(USE) RELEASE(COMMIT) -
   CURRENTDATA(NO) -
   ACT(REP) RETAIN ISOLATION(CS)  -
   LIB('MQ600.SCSQDEFS')

   BIND PLAN(CSQ5C600) -
   MEM(CSQ5C600) -
   ACQUIRE(USE) RELEASE(COMMIT) -
   CURRENTDATA(NO) -
   ACT(REP) RETAIN ISOLATION(CS)  -
   LIB('MQ600.SCSQDEFS')

   BIND PLAN(CSQ5D600) -
   MEM(CSQ5D600) -
   ACQUIRE(USE) RELEASE(COMMIT) -
   CURRENTDATA(NO) -
   ACT(REP) RETAIN ISOLATION(CS)  -
   LIB('MQ600.SCSQDEFS')
```

```
BIND PLAN(CSQ5L600) -
MEM(CSQ5L600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5M600) -
MEM(CSQ5M600) -
ACQUIRE(USE) RELEASE(DEALLOCATE) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5R600) -
MEM(CSQ5R600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5T600) -
MEM(CSQ5T600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5U600) -
MEM(CSQ5U600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5W600) -
MEM(CSQ5W600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')

BIND PLAN(CSQ5B600) -
MEM(CSQ5B600) -
ACQUIRE(USE) RELEASE(COMMIT) -
CURRENTDATA(NO) -
ACT(REP) RETAIN ISOLATION(CS)  -
LIB('MQ600.SCSQDEFS')
```

```
        BIND PLAN(CSQ52600) -
        MEM(CSQ52600) -
        ACQUIRE(USE) RELEASE(COMMIT) -
        CURRENTDATA(NO) -
        ACT(REP) RETAIN ISOLATION(CS)  -
        LIB('MQ600.SCSQDEFS')

        BIND PLAN(CSQ5S600) -
        MEM(CSQ5S600) -
        ACQUIRE(USE) RELEASE(COMMIT) -
        CURRENTDATA(NO) -
        ACT(REP) RETAIN ISOLATION(CS)  -
        LIB('MQ600.SCSQDEFS')

        BIND PLAN(CSQ5K600) -
        MEM(CSQ5K600) -
        ACQUIRE(USE) RELEASE(COMMIT) -
        CURRENTDATA(NO) -
        ACT(REP) RETAIN ISOLATION(CS)  -
        LIB('MQ600.SCSQDEFS')

        BIND PLAN(CSQ5Z600) -
        MEM(CSQ5Z600) -
        ACQUIRE(USE) RELEASE(COMMIT) -
        CURRENTDATA(NO) -
        ACT(REP) RETAIN ISOLATION(CS)  -
        LIB('MQ600.SCSQDEFS')

/*
//
```

Grant execute authority

Job CSQ45GEX is used to grant execute authority to the respective plans for the user IDs that will be used by the queue manager, utilities, and channel initiator.

► The user IDs for the queue manager and channel initiator are the user IDs under which their started task procedures run.

► The user IDs for the utilities are the user IDs under which the batch jobs can be submitted.

Example 5-6   Grant execute authority - CSQ45GEX

```
//MQQGGEX JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*****************************************************************
//*                IBM WebSphere MQ for z/OS                     *
//*                                                              *
```

```
//* Sample job to grant execute authority for the DB2 plans used    *
//* by the WebSphere MQ queue manager and utility programs.          *
//*                                                                  *
//********************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//********************************************************************
//*
//* WARNING: Completion code 04 may be returned if the grantee
//* already has execute authority on the specified plan.
//*
//********************************************************************
//*
//GRANT    EXEC PGM=IKJEFT01,REGION=4M,DYNAMNBR=20
//STEPLIB  DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
    DSN SYSTEM(D8QG)
    RUN PROGRAM(DSNTIAD) -
        PLAN(DSNTIA81)   -
        LIB('DB8QU.RUNLIB.LOAD')
/*
//*
//* Repeat the GRANT statement for CSQ5B600
//*   for each user ID that may use the CSQ5PQSG utility.
//* Repeat the GRANT statement for CSQ52600
//*   for each user ID that may use the CSQUTIL  utility.
//* Repeat the GRANT statement for CSQ5Z600
//*   for each user ID that may use the CSQUZAP  utility.
//SYSIN    DD *
   GRANT EXECUTE ON PLAN CSQ5A600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5C600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5D600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5L600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5M600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5R600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5T600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5U600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5W600 TO STC;

   GRANT EXECUTE ON PLAN CSQ5S600 TO STC;
   GRANT EXECUTE ON PLAN CSQ5K600 TO STC;

   GRANT EXECUTE ON PLAN CSQ5B600 TO RCONWAY;
   GRANT EXECUTE ON PLAN CSQ52600 TO RCONWAY;
   GRANT EXECUTE ON PLAN CSQ5Z600 TO RCONWAY;
/*
//
```

5.3.2 Set up the CFRM policy with the MQ structures

You need to define the coupling facility structures used by the queue managers in the queue-sharing group in the Coupling Facility Resource Management (CFRM) policy data set, using IXCMIAPU. All the structures for the queue sharing group start with the name of the queue sharing group (in our case, it is MQQG). You must have the following:

► An administrative structure called qsg-name CSQ_ADMIN (in our case, it is MQQGCSQ_ADMIN). This structure is used by WebSphere MQ itself and does not contain any user data.

► One or more structures used to hold messages for shared queues. These can have any name you choose, up to 16 characters in length. The first four characters must be the queue-sharing group name (in our case, the structure name is MQQGWEBQUEUE).

Figure 5-34 shows the statements added to the CFRM policy to define the two MQ structures.

```
STRUCTURE NAME(MQQGCSQ_ADMIN)
      INITSIZE(10240)
      SIZE(20480)
      PREFLIST(CF03,CF06)
      REBUILDPERCENT(5)
      FULLTHRESHOLD(85)


STRUCTURE NAME(MQQGWEBQUEUE)
      INITSIZE(10240)
      SIZE(20480)
      PREFLIST(CF03,CF06)
      REBUILDPERCENT(5)
      FULLTHRESHOLD(85)
```

Figure 5-34   Define structures in CFRM policy

When you have defined your structures successfully, activate the CFRM policy that is being used with the SETXCF command.

5.3.3 Add the MQ data sharing group entry to the DB2 table

You need to define the MQ subsystems that are participating in the MQ queue-sharing groups and the MQ sharing group itself in the DB2 table. You can use the CSQ5PQSG utility to add queue-sharing group and queue manager entries to the WebSphere MQ tables in the DB2 data-sharing group. You can run

the utility for the queue-sharing group and for the queue manager that is to be a member of the queue-sharing group.

To accomplish this requires the following:

► Add a queue sharing group entry.
► Add a queue manager entry for MQQ1.
► Add a queue manager entry for MQQ2.

Add a queue sharing group entry

Add a queue-sharing group entry into the WebSphere MQ DB2 tables using the ADD QSG function of the CSQ5PQSG program by running the CSQ45AQS job. Note the following:

► You need to perform this function once for *each* queue-sharing group that is defined in the DB2 data-sharing group.

► The queue-sharing group entry must exist *before* adding any queue manager entries that reference the queue-sharing group.

Example 5-7   Add a queue-sharing group entry - CSQ45AQS

```
//MQQGAQS JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//*****************************************************************
//*                IBM WebSphere MQ for z/OS                      *
//*                                                               *
//* Sample job to add a queue-sharing group record into the      *
//* DB2 administration table CSQ.ADMIN_B_QSG used by WebSphere MQ *
//* using the CSQ5PQSG utility.                                   *
//*                                                               *
//*****************************************************************
//*                                                               *
//* MORE INFORMATION - See:                                       *
//*    "WebSphere MQ for z/OS System Setup Guide"                 *
//*       for information about this customization job            *
//*    "WebSphere MQ for z/OS System Administration Guide"        *
//*       for information about CSQ5PQSG                          *
//*      and managing queue-sharing groups                       *
//*                                                               *
//*****************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//*****************************************************************
//*
//ADDQSG   EXEC PGM=CSQ5PQSG,REGION=4M,
//         PARM='ADD QSG,MQQG,DB8QU,D8QG'
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=MQ600.SCSQANLE,DISP=SHR
```

```
//          DD DSN=MQ600.SCSQAUTH,DISP=SHR
//          DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//
```

Add a queue manager entry for MQQ1

Add a queue manager entry for MQQ1 into the WebSphere MQ DB2 tables using
the ADD QMGR function of the CSQ5PQSG program by running the
CSQ45AQM job.

Example 5-8   CSQ45AQM, add a queue manager entry

```
//MQQGAQM JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//******************************************************************
//*                  IBM WebSphere MQ for z/OS                     *
//*                                                                *
//* Sample job to add a queue manager record into the             *
//* DB2 administration table CSQ.ADMIN_B_QMGR used by WebSphere MQ *
//* using the CSQ5PQSG utility.                                    *
//*                                                                *
//******************************************************************
//*                                                                *
//* MORE INFORMATION - See:                                        *
//*   "WebSphere MQ for z/OS System Setup Guide"                   *
//*     for information about this customization job               *
//*   "WebSphere MQ for z/OS System Administration Guide"          *
//*     for information about CSQ5PQSG                             *
//*     and managing queue-sharing groups                         *
//*                                                                *
//******************************************************************
//* CUSTOMIZE THIS JCL HERE FOR YOUR INSTALLATION
//******************************************************************
//*
//ADDQMGR  EXEC PGM=CSQ5PQSG,REGION=4M,
// PARM='ADD QMGR,MQQ1,MQQG,DB8QU,D8QG'
//SYSPRINT DD SYSOUT=*
//STEPLIB  DD DSN=MQ600.SCSQANLE,DISP=SHR
//         DD DSN=MQ600.SCSQAUTH,DISP=SHR
//         DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//
```

Add a queue manager entry for MQQ2

Add a queue manager entry for MQQ2 by running the same job and updating the
parm field specifying the queue manager name.

5.3.4 Update the ZPARM

You need to update the ZPARM of each MQ subsystem participating in the sharing group to define the sharing group and provide information about the DB2 sharing group that is holding the MQ tables. In the CSQ6SYSP section of the ZPARM table, the parameter QSGDATA (queue-sharing group data) needs to be updated with the following information:

QSGDATA=*(Qsgname,Dsgname,Db2name,Db2serv,Db2blob)*

Qsgname	The name of the queue-sharing group to which the queue manager belongs
Dsgname	The name of the DB2 data-sharing group to which the queue manager is to connect
Db2name	The name of the DB2 subsystem or group attachment to which the queue manager is to connect
Db2serv	The number of server tasks used for accessing DB2
Db2blob	The number of server tasks used for processing DB2 blobs

In our case, this was QSGDATA=(MQQG,DB8QU,D8QG,10,4).

Example 5-9   Update ZPARM - CSQ4ZPRM

```
//MQQGZPRM JOB (999,POK),'CONWAY',CLASS=A,MSGCLASS=T,
// NOTIFY=&SYSUID,TIME=1440,REGION=0M
/*JOBPARM L=999,SYSAFF=SC55
//******************************************************************
//*             IBM WebSphere MQ for z/OS                    *
//*
//*    This job assembles and links a new system parameter module.
//*
//*    Edit the parameters for the
//*    CSQ6LOGP, CSQ6ARVP, and CSQ6SYSP macros
//*    to determine your system parameters.
//*
//******************************************************************
//*                                                          *
//* MORE INFORMATION - See:                                  *
//*   "WebSphere MQ for z/OS System Setup Guide"             *
//*     for information about this customization job         *
//*     and a full description of the parameters.            *
//*                                                          *
//******************************************************************
//*
//*             Assemble step for CSQ6LOGP
//*
```

```
//LOGP   EXEC PGM=ASMA90,PARM='DECK,NOOBJECT,LIST,XREF(SHORT)',
//            REGION=4M
//SYSLIB   DD DSN=MQ600.SCSQMACS,DISP=SHR
//         DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&LOGP,
//            UNIT=SYSDA,DISP=(,PASS),
//            SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
         CSQ6LOGP INBUFF=60,          ARCHIVE LOG BUFFER SIZES (KB)   X
                  OUTBUFF=4000,        - INPUT AND OUTPUT             X
                  MAXRTU=2,           MAX ALLOCATED ARCHIVE LOG UNITS X
                  DEALLCT=0,          ARCHIVE LOG DEALLOCATE INTERVAL X
                  OFFLOAD=YES,        ARCHIVING ACTIVE                X
                  MAXARCH=500,        MAX ARCHIVE LOG VOLUMES         X
                  TWOACTV=YES,        DUAL ACTIVE LOGGING             X
                  TWOARCH=YES,        DUAL ARCHIVE LOGGING            X
                  TWOBSDS=YES,        DUAL BSDS                       X
                  WRTHRSH=20          ACTIVE LOG BUFFERS
         END
/*
//*
//*          Assemble step for CSQ6ARVP
//*
//ARVP   EXEC PGM=ASMA90,COND=(0,NE),
//            PARM='DECK,NOOBJECT,LIST,XREF(SHORT)',
//            REGION=4M
//SYSLIB   DD DSN=MQ600.SCSQMACS,DISP=SHR
//         DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPUNCH DD DSN=&&ARVP,
//            UNIT=SYSDA,DISP=(,PASS),
//            SPACE=(400,(100,100,1))
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
         CSQ6ARVP ALCUNIT=BLK,        UNITS FOR PRIQTY/SECQTY         X
                  ARCPFX1=CSQARC1,    DSN PREFIX FOR ARCHIVE LOG 1    X
                  ARCPFX2=CSQARC2,    DSN PREFIX FOR ARCHIVE LOG 2    X
                  ARCRETN=9999,       ARCHIVE LOG RETENION (DAYS)     X
                  ARCWRTC=(1,3,4),    ARCHIVE WTO ROUTE CODE          X
                  ARCWTOR=YES,        PROMPT BEFORE ARCHIVE LOG MOUNT X
                  BLKSIZE=28672,      ARCHIVE LOG BLOCKSIZE           X
                  CATALOG=NO,         CATALOG ARCHIVE LOG DATA SETS   X
                  COMPACT=NO,         ARCHIVE LOGS COMPACTED          X
                  PRIQTY=4320,        PRIMARY SPACE ALLOCATION        X
                  PROTECT=NO,         DISCRETE SECURITY PROFILES      X
                  QUIESCE=5,          MAX QUIESCE TIME (SECS)         X
                  SECQTY=540,         SECONDARY SPACE ALLOCATION      X
```

```
                     TSTAMP=NO,              TIMESTAMP SUFFIX IN DSN           X
                     UNIT=TAPE,              ARCHIVE LOG DEVICE TYPE 1         X
                     UNIT2=                  ARCHIVE LOG DEVICE TYPE 2
              END
         /*
         //*
         //*         Assemble step for CSQ6SYSP
         //*
         //SYSP    EXEC PGM=ASMA90,COND=(0,NE),
         //              PARM='DECK,NOOBJECT,LIST,XREF(SHORT)',
         //              REGION=0M
         //SYSLIB   DD DSN=MQ600.SCSQMACS,DISP=SHR
         //         DD DSN=SYS1.MACLIB,DISP=SHR
         //SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
         //SYSPUNCH DD DSN=&&SYSP,
         //            UNIT=SYSDA,DISP=(,PASS),
         //            SPACE=(400,(100,100,1))
         //SYSPRINT DD SYSOUT=*
         //SYSIN    DD *
              CSQ6SYSP CTHREAD=300,       TOTAL NUMBER OF CONNECTIONS      X
                     CLCACHE=STATIC,       CLUSTER CACHE TYPE              X
                     CMDUSER=CSQOPR,       DEFAULT USERID FOR COMMANDS     X
                     EXITLIM=30,           EXIT TIMEOUT (SEC)              X
                     EXITTCB=8,            NUMBER OF EXIT SERVER TCBS      X
                     IDBACK=20,            NUMBER OF NON-TSO CONNECTIONS   X
                     IDFORE=100,           NUMBER OF TSO CONNECTIONS       X
                     LOGLOAD=500000,       LOG RECORD CHECKPOINT NUMBER    X
                     OTMACON=(,,DFSYDRU0,2147483647,CSQ),  OTMA PARAMETERS X
                     QINDXBLD=WAIT,        QUEUE INDEX BUILDING            X
                     QMCCSID=0,            QMGR CCSID                      X
                     QSGDATA=(MQQG,DB8QU,D8QG,10,4),                       X
                     RESAUDIT=YES,         RESLEVEL AUDITING               X
                     ROUTCDE=1,            DEFAULT WTO ROUTE CODE          X
                     SMFACCT=NO,           GATHER SMF ACCOUNTING           X
                     SMFSTAT=NO,           GATHER SMF STATS                X
                     STATIME=30,           STATISTICS RECORD INTERVAL (MIN) X
                     TRACSTR=YES,          TRACING AUTO START              X
                     TRACTBL=99,           GLOBAL TRACE TABLE SIZE X4K     X
                     WLMTIME=30,           WLM QUEUE SCAN INTERVAL (SEC)   X
                     WLMTIMU=MINS,         WLMTIME UNITS                   X
                     SERVICE=0             IBM SERVICE USE ONLY
              END
         /*
         //*
         //*
         //* Linkedit ARVP, LOGP, and SYSP into a
         //* system parameter module.
         //*
         //LKED    EXEC PGM=IEWL,COND=(0,NE),
```

```
//          PARM='SIZE=(900K,124K),RENT,NCAL,LIST,AMODE=31,RMODE=ANY'
//*
//*   APF-authorized library for the new system parameter module
//SYSLMOD  DD DSN=MQQ1.USERAUTH,DISP=SHR
//*
//SYSUT1   DD UNIT=SYSDA,DCB=BLKSIZE=1024,
//            SPACE=(1024,(200,20))
//SYSPRINT DD SYSOUT=*
//ARVP     DD DSN=&&ARVP,DISP=(OLD,DELETE)
//LOGP     DD DSN=&&LOGP,DISP=(OLD,DELETE)
//SYSP     DD DSN=&&SYSP,DISP=(OLD,DELETE)
//*
//*   Load library containing the default system
//*   parameter module (CSQZPARM).
//OLDLOAD  DD DSN=MQ600.SCSQAUTH,DISP=SHR
//*
//SYSLIN   DD *
   INCLUDE SYSP
   INCLUDE ARVP
   INCLUDE LOGP
   INCLUDE OLDLOAD(CSQZPARM)
 ENTRY CSQZMSTR
 NAME  CSQZPARM(R)                     Your system parameter module name
/*
//
```

Restart the MQ subsystem

During the startup of MQ, you should see the MQ subsystem successfully connecting to RRS, to DB2, and to the CF structure previously allocated.

5.3.5 Update the queue manager procedures

Sample initialization member CSQ4INSS, found in hlq.SCSQPROC, contains a set of commands for defining the objects used in a queue-sharing group. Add the customized member to the CSQINP2 DD in both queue manager procedures, as shown in Example 5-10.

Example 5-10   mqq1mstr process

```
//          PROC
//PROCSTEP EXEC PGM=CSQYASCP,REGION=0M
//*
//STEPLIB   DD DSN=MQQ1.USERAUTH,DISP=SHR
//          DD DSN=MQ600.SCSQANLE,DISP=SHR
//          DD DSN=MQ600.SCSQAUTH,DISP=SHR
//          DD DSN=DB8Q8.SDSNLOAD,DISP=SHR
//*
```

```
//********************************************************************
//* BOOTSTRAP DATA SETS                                              *
//*                                                                  *
//* This sample shows dual BSDS. To run with a single BSDS remove    *
//* the BSDS2 entry.                                                 *
//********************************************************************
//BSDS1     DD DSN=MQQ1.BSDS01,DISP=SHR
//BSDS2     DD DSN=MQQ1.BSDS02,DISP=SHR
//*
//********************************************************************
//* SYSTEM INITIALIZATION INPUT FILES                                *
//*                                                                  *
//* This sample shows the IBM supplied samples being used for the    *
//* initialization input files. These sample initialization input    *
//* files should be copied into a user library and tailored.         *
//********************************************************************
//CSQINP1   DD DSN=MQQ1.INSTALL.JCL(MQQ1INP1),DISP=SHR
//CSQINP2   DD DSN=MQQ1.INSTALL.JCL(MQQ1INSG),DISP=SHR
//          DD DSN=MQQ1.INSTALL.JCL(MQQ1INSX),DISP=SHR
//          DD DSN=MQQ1.INSTALL.JCL(MQQ1INSS),DISP=SHR
//          DD DSN=MQQ1.INSTALL.JCL(MQQ1INYS),DISP=SHR
//*         DD DSN=MQQ1.INSTALL.JCL(MQQ1STAC),DISP=SHR
//CSQOUT1   DD SYSOUT=*
//CSQOUT2   DD SYSOUT=*
//*
//********************************************************************
//* PAGE SET DATA SETS                                               *
//*                                                                  *
//* This sample shows five page set data sets.                       *
//* You must have page set 00.                                       *
//********************************************************************
//CSQP0000  DD DSN=MQQ1.PSID00,DISP=SHR
//CSQP0001  DD DSN=MQQ1.PSID01,DISP=SHR
//CSQP0002  DD DSN=MQQ1.PSID02,DISP=SHR
//CSQP0003  DD DSN=MQQ1.PSID03,DISP=SHR
//CSQP0004  DD DSN=MQQ1.PSID04,DISP=SHR
//*
//********************************************************************
//* USER EXIT LIBRARY                                                *
//********************************************************************
//*CSQXLIB   DD DSN=++EXITLIB++,DISP=SHR
//*
//********************************************************************
//* USER EXIT DATA SETS                                              *
//* Add here DD statements for any data sets used by user exits.     *
//********************************************************************
//*
```

5.3.6 Define the shared queues between the two MQ subsystems

At this point, you can define any shared queue. For this application, two queues have been defined: Q2.QUE and Q1.QUE. Since they are shared queues, the definitions need to be done only once and will be accessible from both MQ subsystems.

Example 5-11 a shows the definitions for Q1.QUE.

Example 5-11   Q1.QUE

```
QUEUE(Q1.QUE)
TYPE(QLOCAL)
QSGDISP(SHARED)
STGCLASS(DEFAULT)
CFSTRUCT(WEBQUEUE)
CLUSTER( )
CLUSNL( )
DESCR( )
PUT(ENABLED)
DEFPRTY(0)
DEFPSIST(NO)
OPPROCS(0)
IPPROCS(0)
CURDEPTH(0)
MAXDEPTH(1000)
PROCESS( )
TRIGGER
MAXMSGL(1000)
BOTHRESH(0)
BOQNAME( )
INITQ( )
USAGE(NORMAL)
SHARE
DEFSOPT(SHARED)
MSGDLVSQ(PRIORITY)
RETINTVL(999999999)
TRIGTYPE(NONE)
TRIGDPTH(1)
TRIGMPRI(0)
TRIGDATA( )
DEFTYPE(PREDEFINED)
NOHARDENBO
CRDATE(2003-08-08)
CRTIME(14.43.13)
GET(ENABLED)
QDEPTHHI(80)
QDEPTHLO(40)
QDPMAXEV(DISABLED)
```

```
QDPHIEV(DISABLED)
QDPLOEV(DISABLED)
QSVCINT(999999999)
QSVCIEV(NONE)
INDXTYPE(NONE)
DEFBIND(OPEN)
```

Example 5-12 a shows the definitions for Q2.QUE.

Example 5-12   Q2.QUE

```
QUEUE(Q2.QUE)
TYPE(QLOCAL)
QSGDISP(SHARED)
STGCLASS(DEFAULT)
CFSTRUCT(WEBQUEUE)
CLUSTER( )
CLUSNL( )
DESCR( )
PUT(ENABLED)
DEFPRTY(0)
DEFPSIST(NO)
OPPROCS(0)
IPPROCS(0)
CURDEPTH(0)
MAXDEPTH(1000)
PROCESS( )
TRIGGER
MAXMSGL(1000)
BOTHRESH(0)
BOQNAME( )
INITQ( )
USAGE(NORMAL)
SHARE
DEFSOPT(SHARED)
MSGDLVSQ(PRIORITY)
RETINTVL(999999999)
TRIGTYPE(NONE)
TRIGDPTH(1)
TRIGMPRI(0)
TRIGDATA( )
DEFTYPE(PREDEFINED)
NOHARDENBO
CRDATE(2003-08-08)
CRTIME(14.43.08)
GET(ENABLED)
QDEPTHHI(80)
QDEPTHLO(40)
QDPMAXEV(DISABLED)
```

```
                QDPHIEV(DISABLED)
                QDPLOEV(DISABLED)
                QSVCINT(999999999)
                QSVCIEV(NONE)
                INDXTYPE(CORRELID)
                DEFBIND(OPEN)
```

5.3.7 Starting WebSphere MQ

Example 5-13 shows the messages generated when WebSphere MQ is started.

Example 5-13   MQ syslog

```
-MQQ1 START QMGR
S MQQ1MSTR
$HASP100 MQQ1MSTR ON STCINRDR
IEF695I START MQQ1MSTR WITH JOBNAME MQQ1MSTR IS ASSIGNED TO USER STC
   , GROUP SYS1
$HASP373 MQQ1MSTR STARTED
IEF403I MQQ1MSTR - STARTED - TIME=01.00.36 - ASID=0061 - SC55
CSQY000I -MQQ1 IBM WebSphere MQ for z/OS V6
CSQY001I -MQQ1 QUEUE MANAGER STARTING, USING PARAMETER MODULE CSQZPARM
CSQ3111I -MQQ1 CSQYSCMD - EARLY PROCESSING PROGRAM IS V6 LEVEL 003-002
CSQY100I -MQQ1 SYSTEM parameters ...
CSQY101I -MQQ1 CTHREAD=300, IDBACK=20, IDFORE=100, LOGLOAD=500000
CSQY102I -MQQ1 CMDUSER=CSQOPR, QMCCSID=0, ROUTCDE=( 1)
CSQY103I -MQQ1 SMFACCT=NO (00000000), SMFSTAT=NO (00000000),
STATIME=30
CSQY104I -MQQ1 OTMACON= 522
(        ,                ,DFSYDRU0,2147483647,CSQ)
CSQY105I -MQQ1 TRACSTR=( 1), TRACTBL=99
CSQY106I -MQQ1 EXITTCB=8, EXITLIM=30, WLMTIME=30, WLMTIMU=MINS
CSQY107I -MQQ1 QSGDATA=(MQQG,DB8QU,D8QG,10,4)
CSQY108I -MQQ1 RESAUDIT=YES, QINDXBLD=WAIT, CLCACHE=STATIC
CSQY110I -MQQ1 LOG parameters ...
CSQY111I -MQQ1 INBUFF=60, OUTBUFF=4000, MAXRTU=2, MAXARCH=500
CSQY112I -MQQ1 TWOACTV=YES, TWOARCH=YES, TWOBSDS=YES
CSQY113I -MQQ1 OFFLOAD=NO, WRTHRSH=20, DEALLCT=0
CSQY120I -MQQ1 ARCHIVE parameters ...
CSQY121I -MQQ1 UNIT=TAPE, UNIT2=, ALCUNIT=BLK, 532
PRIQTY=4320, SECQTY=540, BLKSIZE=28672
CSQY122I -MQQ1 ARCPFX1=CSQARC1, ARCPFX2=CSQARC2, TSTAMP=NO
CSQY123I -MQQ1 ARCRETN=9999, ARCWTOR=YES, ARCWRTC=( 1 ,3 ,4)
CSQY124I -MQQ1 CATALOG=NO, COMPACT=NO, PROTECT=NO, QUIESCE=5
CSQY201I -MQQ1  CSQYSTRT ARM REGISTER for element 536
SYSMQMGRMQQ1 type SYSMQMGR successful
IEC161I 056-084,MQQ1MSTR,MQQ1MSTR,BSDS1,,,MQQ1.BSDS01, 537
IEC161I MQQ1.BSDS01.DATA,MCAT.ZOSR06.Z16CAT
```

```
IEC161I 056-084,MQQ1MSTR,MQQ1MSTR,BSDS1,,,MQQ1.BSDS01, 538
IEC161I MQQ1.BSDS01.INDEX,MCAT.ZOSR06.Z16CAT
IEC161I 062-086,MQQ1MSTR,MQQ1MSTR,BSDS1,,,MQQ1.BSDS01, 539
IEC161I MQQ1.BSDS01.DATA,MCAT.ZOSR06.Z16CAT
IEC161I 056-084,MQQ1MSTR,MQQ1MSTR,BSDS2,,,MQQ1.BSDS02, 540
IEC161I MQQ1.BSDS02.DATA,MCAT.ZOSR06.Z16CAT
IEC161I 056-084,MQQ1MSTR,MQQ1MSTR,BSDS2,,,MQQ1.BSDS02, 541
IEC161I MQQ1.BSDS02.INDEX,MCAT.ZOSR06.Z16CAT
IEC161I 062-086,MQQ1MSTR,MQQ1MSTR,BSDS2,,,MQQ1.BSDS02, 542
IEC161I MQQ1.BSDS02.DATA,MCAT.ZOSR06.Z16CAT
CSQJ127I -MQQ1 SYSTEM TIME STAMP FOR BSDS=2006-01-18 00:58:31.35
CSQJ001I -MQQ1 CURRENT COPY 1 ACTIVE LOG DATA SET IS 544
DSNAME=MQQ1.LOGCOPY1.DS01, STARTRBA=000000000000 ENDRBA=00000464FFFF
CSQJ001I -MQQ1 CURRENT COPY 2 ACTIVE LOG DATA SET IS 545
DSNAME=MQQ1.LOGCOPY2.DS01, STARTRBA=000000000000 ENDRBA=00000464FFFF
CSQJ099I -MQQ1 LOG RECORDING TO COMMENCE WITH 546
STARTRBA=00000002D000
CSQW130I -MQQ1 'GLOBAL' TRACE STARTED, ASSIGNED TRACE NUMBER 01
CSQ5001I -MQQ1 CSQ5CONN Connected to DB2 D8Q1
CSQH021I -MQQ1 CSQHINSQ SUBSYSTEM security switch set 549
OFF, profile 'MQQ1.NO.SUBSYS.SECURITY' found
CSQP007I -MQQ1 Page set 0 uses buffer pool 0
CSQP007I -MQQ1 Page set 1 uses buffer pool 0
CSQP007I -MQQ1 Page set 2 uses buffer pool 1
CSQP007I -MQQ1 Page set 3 uses buffer pool 2
CSQP007I -MQQ1 Page set 4 uses buffer pool 3
CSQY220I -MQQ1 Queue manager is using 123 MB of local 555
storage, 1643 MB are free
CSQV452I -MQQ1 CSQVXLDR Cluster workload exits not available
CSQR001I -MQQ1 RESTART INITIATED
CSQR003I -MQQ1 RESTART - PRIOR CHECKPOINT RBA=00000002BA1D
CSQR004I -MQQ1 RESTART - UR COUNTS - 559
IN COMMIT=0, INDOUBT=0, INFLIGHT=0, IN BACKOUT=0
IXC582I STRUCTURE MQQGCSQ_ADMIN ALLOCATED BY SIZE/RATIOS. 560
  PHYSICAL STRUCTURE VERSION: BE3AC0D4 085E7A8C
  STRUCTURE TYPE:            SERIALIZED LIST
  CFNAME:                    CF03
  ALLOCATION SIZE:     10240 K
  POLICY SIZE:         20480 K
  POLICY INITSIZE:     10240 K
  POLICY MINSIZE:          0 K
  IXLCONN STRSIZE:         0 K
  ENTRY COUNT:          5189
  ELEMENT COUNT:       10224
  EMC COUNT:            1970
  LOCKS:                 256
  ENTRY:ELEMENT RATIO:        1 :    2
  EMC STORAGE PERCENTAGE:  5.00 %
ALLOCATION SIZE IS WITHIN CFRM POLICY DEFINITIONS
```

```
CSQE005I -MQQ1 Structure CSQ_ADMIN connected as 563
CSQEMQQGMQQ1O1, version=BE3AC0D4085E7A8C 00010001
IXL014I IXLCONN REQUEST FOR STRUCTURE MQQGCSQ_ADMIN 561
WAS SUCCESSFUL.  JOBNAME: MQQ1MSTR ASID: 0061
CONNECTOR NAME: CSQEMQQGMQQ1O1 CFNAME: CF03
IXL015I STRUCTURE ALLOCATION INFORMATION FOR 562
STRUCTURE MQQGCSQ_ADMIN, CONNECTOR NAME CSQEMQQGMQQ1O1
 CFNAME     ALLOCATION STATUS/FAILURE REASON
 --------   --------------------------------
 CF03       STRUCTURE ALLOCATED AC001800
 CF06       PREFERRED CF ALREADY SELECTED AC001800
CSQE018I -MQQ1 Admin structure data building started
CSQI049I -MQQ1 Page set 0 has media recovery 565
RBA=00000002BA1D, checkpoint RBA=00000002BA1D
CSQI049I -MQQ1 Page set 1 has media recovery 566
RBA=00000002BA1D, checkpoint RBA=00000002BA1D
CSQI049I -MQQ1 Page set 2 has media recovery 567
RBA=00000002BA1D, checkpoint RBA=00000002BA1D
CSQI049I -MQQ1 Page set 3 has media recovery 568
RBA=00000002BA1D, checkpoint RBA=00000002BA1D
CSQI049I -MQQ1 Page set 4 has media recovery 569
RBA=00000002BA1D, checkpoint RBA=00000002BA1D
CSQE019I -MQQ1 Admin structure data building completed
CSQR030I -MQQ1 Forward recovery log range 571
from RBA=00000002BA1D to RBA=00000002C3E6
CSQR005I -MQQ1 RESTART - FORWARD RECOVERY COMPLETE - 572
IN COMMIT=0, INDOUBT=0
CSQR032I -MQQ1 Backward recovery log range 573
from RBA=00000002C3E6 to RBA=00000002C3E6
CSQR006I -MQQ1 RESTART - BACKWARD RECOVERY COMPLETE - 574
INFLIGHT=0, IN BACKOUT=0
CSQR002I -MQQ1 RESTART COMPLETED
CSQP018I -MQQ1 CSQPBCKW CHECKPOINT STARTED FOR ALL BUFFER POOLS
CSQP019I -MQQ1 CSQP1DWP CHECKPOINT COMPLETED FOR 577
BUFFER POOL 1, 2 PAGES WRITTEN
CSQP019I -MQQ1 CSQP1DWP CHECKPOINT COMPLETED FOR 578
BUFFER POOL 2, 2 PAGES WRITTEN
CSQP019I -MQQ1 CSQP1DWP CHECKPOINT COMPLETED FOR 579
BUFFER POOL 3, 3 PAGES WRITTEN
CSQP019I -MQQ1 CSQP1DWP CHECKPOINT COMPLETED FOR 580
BUFFER POOL 0, 9 PAGES WRITTEN
-MQQ1 DISPLAY CONN(*) TYPE(CONN) ALL WHERE(UOWSTATE EQ UNRESOLVED)
CSQP021I -MQQ1 Page set 0 new media recovery 582
RBA=00000002DF04, checkpoint RBA=00000002DF04
CSQP021I -MQQ1 Page set 1 new media recovery 583
RBA=00000002DF04, checkpoint RBA=00000002DF04
CSQP021I -MQQ1 Page set 2 new media recovery 584
RBA=00000002DF04, checkpoint RBA=00000002DF04
CSQP021I -MQQ1 Page set 3 new media recovery 585
```

```
RBA=00000002DF04, checkpoint RBA=00000002DF04
CSQP021I -MQQ1 Page set 4 new media recovery 586
RBA=00000002DF04, checkpoint RBA=00000002DF04
CSQM297I -MQQ1 CSQMDRTC NO CONN FOUND MATCHING REQUEST CRITERIA
CSQ9022I -MQQ1 CSQMDRTC ' DISPLAY CONN' NORMAL COMPLETION
-MQQ1 DISPLAY SYSTEM
-MQQ1 DISPLAY LOG
CSQJ322I -MQQ1 DISPLAY SYSTEM report ... 591
Parameter    Initial value         SET value
-----------  --------------------  --------------------
CTHREAD      300
IDBACK       20
IDFORE       100
LOGLOAD      500000
CMDUSER      CSQOPR
QMCCSID      0
ROUTCDE      1
SMFACCT      NO
SMFSTAT      NO
STATIME      30
OTMACON
  GROUP
  MEMBER
  DRUEXIT    DFSYDRU0
  AGE        2147483647
  TPIPEPFX   CSQ
TRACSTR      1
TRACTBL      99
EXITTCB      8
EXITLIM      30
WLMTIME      30
WLMTIMU      MINS
QSGDATA
  QSGNAME    MQQG
  DSGNAME    DB8QU
  DB2NAME    D8QG
  DB2SERV    10
   DB2BLOB   4
 RESAUDIT    YES
 QINDXBLD    WAIT
 CLCACHE     STATIC
 End of SYSTEM report
CSQ9022I -MQQ1 CSQJC001 ' DISPLAY SYSTEM' NORMAL COMPLETION
-MQQ1 DISPLAY ARCHIVE
CSQJ322I -MQQ1 DISPLAY LOG report ... 594
Parameter    Initial value         SET value
-----------  --------------------  --------------------
 INBUFF      60
 OUTBUFF     4000
```

```
MAXRTU     2
MAXARCH    2
TWOACTV    YES
TWOARCH    YES
TWOBSDS    YES
OFFLOAD    NO
WRTHRSH    20
DEALLCT    0
End of LOG report
CSQJ370I -MQQ1 LOG status report ... 595
Copy %Full DSName
 1      1  MQQ1.LOGCOPY1.DS01
 2      1  MQQ1.LOGCOPY2.DS01
Restarted at 2006-01-18 01:00:37 using RBA=00000002D000
Latest RBA=00000002EF12
Offload task is AVAILABLE
Full logs to offload - 0 of 8
CSQ9022I -MQQ1 CSQJC001 ' DISPLAY LOG' NORMAL COMPLETION
-MQQ1 DISPLAY USAGE
CSQJ322I -MQQ1 DISPLAY ARCHIVE report ... 598
Parameter   Initial value         SET value
----------- --------------------- ----------------------
UNIT        TAPE
UNIT2
ALCUNIT     BLK
PRIQTY      4320
SECQTY      540
BLKSIZE     28672
ARCPFX1     CSQARC1
ARCPFX2     CSQARC2
TSTAMP      NO
ARCRETN     9999
ARCWTOR     YES
ARCWRTC     1 ,3 ,4
CATALOG     NO
COMPACT     NO
PROTECT     NO
QUIESCE     5
End of ARCHIVE report
CSQJ325I -MQQ1 ARCHIVE tape unit report ... 599
Addr St CorrelID VolSer DSName
---- -- -------- ------ -------------------------------------------
No tape archive reading activity
End of tape unit report
CSQ9022I -MQQ1 CSQJC001 ' DISPLAY ARCHIVE' NORMAL COMPLETION
CSQM050I -MQQ1 CSQMIGQA Intra-group queuing agent 601
starting, TCB=0065F828
CSQI010I -MQQ1 Page set usage ... 602
Page Buffer   Total     Unused  Persistent Nonpersistent  Expansion
```

```
set  pool   pages    pages data pages   data pages     count
_  0   0     5038     5023          15            0 USER   0
_  1   0     5038     5038           0            0 USER   0
_  2   1     5038     5038           0            0 USER   0
_  3   2     5038     5038           0            0 USER   0
_  4   3     5038     5037           1            0 USER   0
 End of page set report
CSQP001I -MQQ1 Buffer pool 0 has 50000 buffers
CSQP001I -MQQ1 Buffer pool 1 has 20000 buffers
CSQP001I -MQQ1 Buffer pool 2 has 50000 buffers
CSQP001I -MQQ1 Buffer pool 3 has 20000 buffers
CSQI024I -MQQ1 CSQIDUSE Restart RBA for system as 607
configured=00000002DF04
CSQ9022I -MQQ1 CSQIDUSE ' DISPLAY USAGE' NORMAL COMPLETION
CSQY022I -MQQ1 QUEUE MANAGER INITIALIZATION COMPLETE
CSQ9022I -MQQ1 CSQYASCP 'START QMGR' NORMAL COMPLETION
IXC582I STRUCTURE MQQGAPPLICATION1 ALLOCATED BY SIZE/RATIOS. 611
  PHYSICAL STRUCTURE VERSION: BE3AC0D6 8BEC2882
  STRUCTURE TYPE:            SERIALIZED LIST
  CFNAME:                    CF03
   ALLOCATION SIZE:      10240 K
   POLICY SIZE:          20480 K
   POLICY INITSIZE:      10240 K
   POLICY MINSIZE:           0 K
   IXLCONN STRSIZE:          0 K
   ENTRY COUNT:           2217
   ELEMENT COUNT:        12929
   EMC COUNT:             1970
   LOCKS:                 1024
   ENTRY:ELEMENT RATIO:           1 :       6
   EMC STORAGE PERCENTAGE:     5.00 %
 ALLOCATION SIZE IS WITHIN CFRM POLICY DEFINITIONS
 IXL014I IXLCONN REQUEST FOR STRUCTURE MQQGWEBQUEUE 612
WAS SUCCESSFUL.  JOBNAME: MQQ1MSTR ASID: 0061
CONNECTOR NAME: CSQEMQQGMQQ101 CFNAME: CF03
IXL015I STRUCTURE ALLOCATION INFORMATION FOR 613
STRUCTURE MQQGAPPLICATION1, CONNECTOR NAME CSQEMQQGMQQ101
 CFNAME     ALLOCATION STATUS/FAILURE REASON
 --------   ----------------------------------
 CF03       STRUCTURE ALLOCATED AC001800
 CF06       PREFERRED CF ALREADY SELECTED AC001800
CSQE005I -MQQ1 Structure WEBQUEUE connected as 614
CSQEMQQGMQQ101, version=BE3AC0D68BEC2882 00010001
CSQY220I -MQQ1 Queue manager is using 714 MB of local 615
storage, 1052 MB are free
```

5.3.8 For more information

For more information about WebSphere MQ queue sharing, see the following:

► *WebSphere MQ Queue Sharing Group in a Parallel Sysplex environment*, REDP-3636

► *WebSphere MQ in a z/OS Parallel Sysplex Environment*, SG24-6864

► *Patterns: Self-Service Application Solutions Using WebSphere for z/OS V5*, SG24-7092

► *IBM WebSphere WAS for z/OS and MQ* at:

http://websphere.sys-con.com/read/148226.htm

6

Integration scenarios with WebSphere ESB

This chapter provides information and examples that illustrate some of the mediation functions possible when using WebSphere ESB as an enterprise service bus. The intent is to give you an idea of how mediations are built using WebSphere Integration Developer and deployed to WebSphere ESB. We introduce you to the concept of using predefind mediations that are provided by WebSphere ESB.

6.1 Using WebSphere ESB

WebSphere ESB provides basic enterprise service bus capabilities, including mediation capabilities.

A mediation intercepts and acts on messages that are passed between a service requester and one or more service providers. Mediations destined for deployment to WebSphere ESB or WebSphere Process Server are referred to as mediation service applications. Mediation service applications are implemented using mediation modules that contain mediation flows. Figure 6-1 shows the elements that make up a mediation service application.
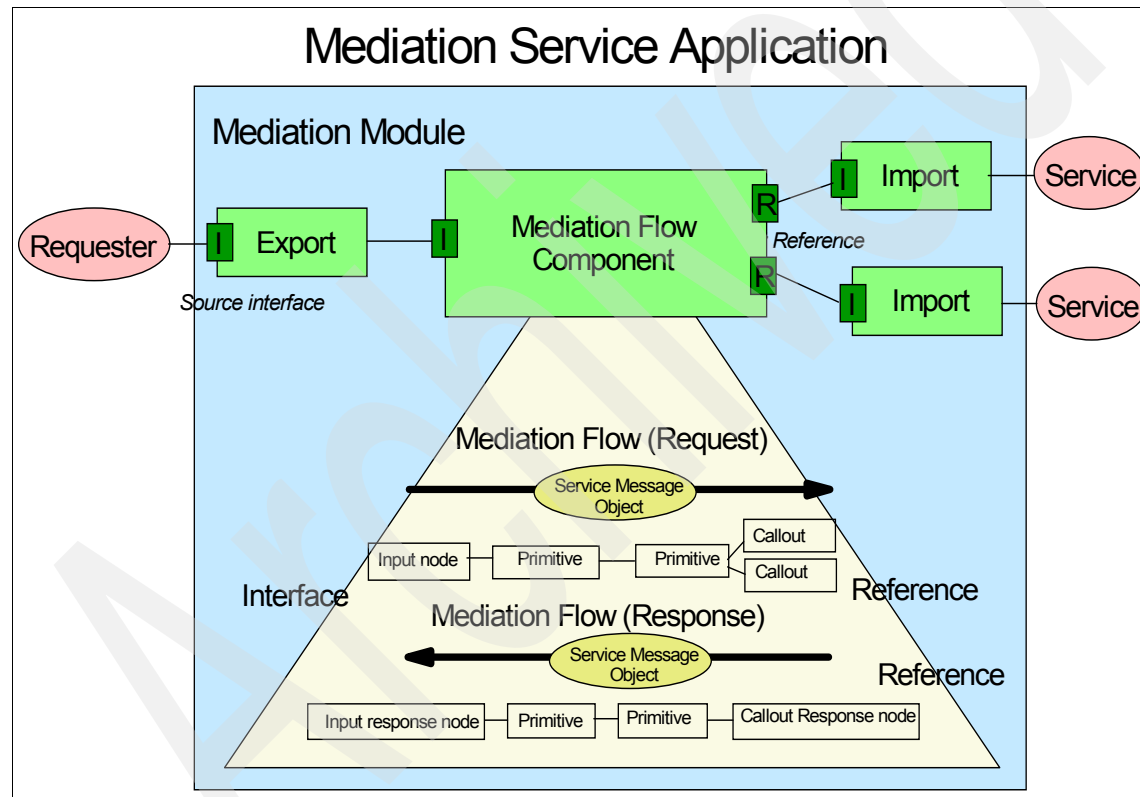
Figure 6-1   Mediation service application

Figure 6-1 is explained below:

► Mediation module

Mediation service applications are assembled and deployed as one or more mediation modules. A mediation module can have the following parts:

– Exports that expose the mediation module to service requesters

– Imports that identify service providers and their interfaces

– A mediation flow component

– Java components that implement custom mediation primitives invoked by the mediation flow component

► Export node

An export node exposes the mediation module to service requesters. Mediation modules and business integration modules have interfaces in their exports, so that the module can be invoked.

► Import node

An import node identifies service providers and their interfaces. It allows you to use functions that are not part of the module that you are assembling.

► Mediation flow component

A mediation flow component consists of the following:

– One or more interfaces that describe how to invoke the flow. An interface must match an export wired to the mediation flow.

– Zero or more references that specify the interfaces of partners the flow can invoke. The references are wired to imports (or Java components) with matching interfaces.

– Mediation flows. Mediation flows that handle messages going from the service requester to the service provider are called request flows. Those that handle responses from the providers to the requester are called response flows.

► Interfaces

An interface provides access to a service and defines the data exchange between components. The interface defines the operations that can be called and the data that is passed. In a mediation flow component, the interface that allows the service requester to access the mediation through an export is called a source interface.

► References

A mediation flow component accesses a service provider (or import) through a reference that specifies the interface that is used by the import to invoke the service. In a mediation flow component, this reference is called a target reference.

► Mediation flow

A mediation flow contains the sequence of steps that process the message.

► Mediation primitive

Mediation flows consist of mediation primitives. A ready-made set of mediation primitives is available from the mediation flow editor palette. If you need mediation capabilities that are not provided by this set of primitives, you can create custom mediation primitives to call your own Java implementation or an imported service.

► Service message objects

Messages in mediation flows are represented as service message objects (SMOs).

Mediation primitives

The following mediation primitives are available to build your mediation flows in WebSphere ESB.

► XSLT

The XSLT mediation primitive can be used to transform an XML message using XSL Transformations (XSLT) 1.0 transformation.

► Message Logger

The Message Logger mediation primitive stores messages in a relational database.

► Database Lookup

The Database Lookup mediation primitive allows you to modify a message using data from a database.

► Message Filter

The Message Filter mediation primitive can make decisions based on the message content, for example, field validation or routing to specific providers or to a specific path in the message flow. This primitive can be used in conjunction with the Database Lookup primitive to enhance the decision-making criteria.

► Fail

The Fail mediation primitive stops a mediation flow and raises an exception.

► Stop

The Stop mediation primitive stops a particular path in the flow.

► Custom mediation

Custom mediation primitives are used to do processing not provided in the other mediation primitives. A custom mediation primitive calls an SCA component that you provide to process the message.

For information about developing custom mediations, see the "contributing your own mediation primitive" plug-in at:

```
http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&uid=swg2
7007001
```

Figure 6-2 shows an example of a using the mediation primitives in a mediation flow.
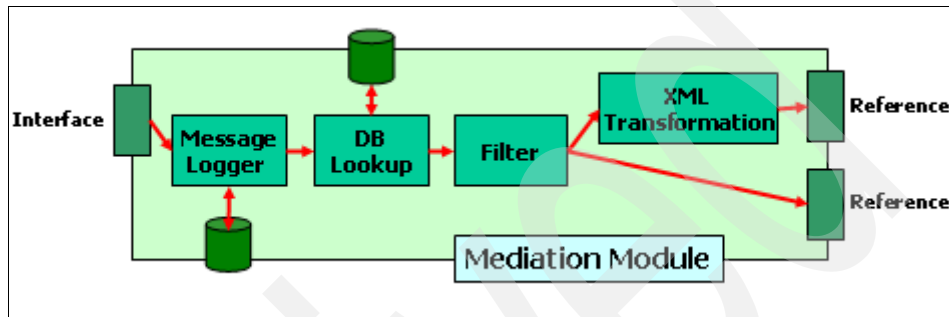
Figure 6-2   Mediation flow

In this example, when a message enters the message flow through the source interface, the mediation logs the message into a database. It then looks up an identifier from the database for authentication, filters the message based on its contents, transforms the message based on XSLT, and routes the message to the target services.

Bindings

A mediation is made available to clients through its export binding. The mediation invokes external services through import bindings.

The following bindings are available for use with imports:

- ► Web services
- ► JMS
- ► EIS adapters
- ► SCA (for connection to other modules)
- ► Stateless session EJB

The following bindings are available for use with exports:

- ► Web services
- ► JMS
- ► EIS adapters - with the exception of CICS and IMS™
- ► SCA (for connection to other modules)

The EIS adapters supported include the following J2C-compliant IBM WebSphere Adapters:

- ► IBM CICS ECI Resource Adapter 6.0.1
- ► IBM IMS Connector for Java 9.1.0.2
- ► IBM WebSphere Adapter for Flat Files 6.0
- ► IBM WebSphere Adapter for JDBC 6.0
- ► IBM WebSphere Adapter for PeopleSoft Enterprise 6.0
- ► IBM WebSphere Adapter for SAP Software 6.0
- ► IBM WebSphere Adapter for Siebel Business Applications 6.0

The IBM WebSphere Business Integration Adapters are also supported. To see a complete list of supported WebSphere Business Integration Adapters, go to:

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

6.1.1 Developing mediations

WebSphere Integration Developer provides the tools required to develop, deploy, and test mediations for WebSphere ESB. WebSphere Integration Developer consists of a Workbench window that displays one or more perspectives. A *perspective* is a group of views and editors required to perform tasks associated with a role. Each perspective consists of views that provide alternative presentations of resources or ways of navigating through information in the Workbench. As a user works with the Workbench, the data representing the projects and working environment are stored in a workspace directory on the local file system.

Mediation development is done using the Business Integration perspective. This perspective is used to develop and test mediations. The perspective consists of views designed specifically for development.

- ► The Business Integration view (top left) lists the mediation modules and their resources.

- ► The Editor view (top right) is where you build modules and flows. The view consists of a canvas where you can assemble and wire components and a Palette that contains elements, such as mediation primitives, that you can drag and drop to the canvas. The content and options for the canvas and Palette vary depending on the editor. For example, there is an assembly editor for modules and a Mediation Flow editor for working with mediation flows. The appropriate editor is invoked when you open an element such as a module or flow for editing.

- ► The lower-half portion of the screen contains multiple views that you can access by clicking the appropriate tab. Among these are the Problems and Outline views. The Problems view shows informational, warning, and error

messages that indicate problems within resources. The Outline view displays a structured outline of the file open in the editor area (palette).

6.1.2 Deploying mediations

Mediations are deployed as EAR files to the WebSphere ESB server. WebSphere Integration Developer builds the files necessary and provides tools for exporting the EAR files.

6.2 Integration scenario

In the scenario introduced in "End-to-end scenario" on page 60, Airline B uses a J2EE messaging application that runs in WebSphere Application Server. Both the travel bureau and Airline B use XML format for messages, but the two formats are not the same. A mediation is required to transform the XML message to the proper format. This can be done using WebSphere ESB or WebSphere Message Broker. In this chapter we look at how to do the mediation with WebSphere ESB.

Figure 6-3 shows the runtime infrastructure for this sample.
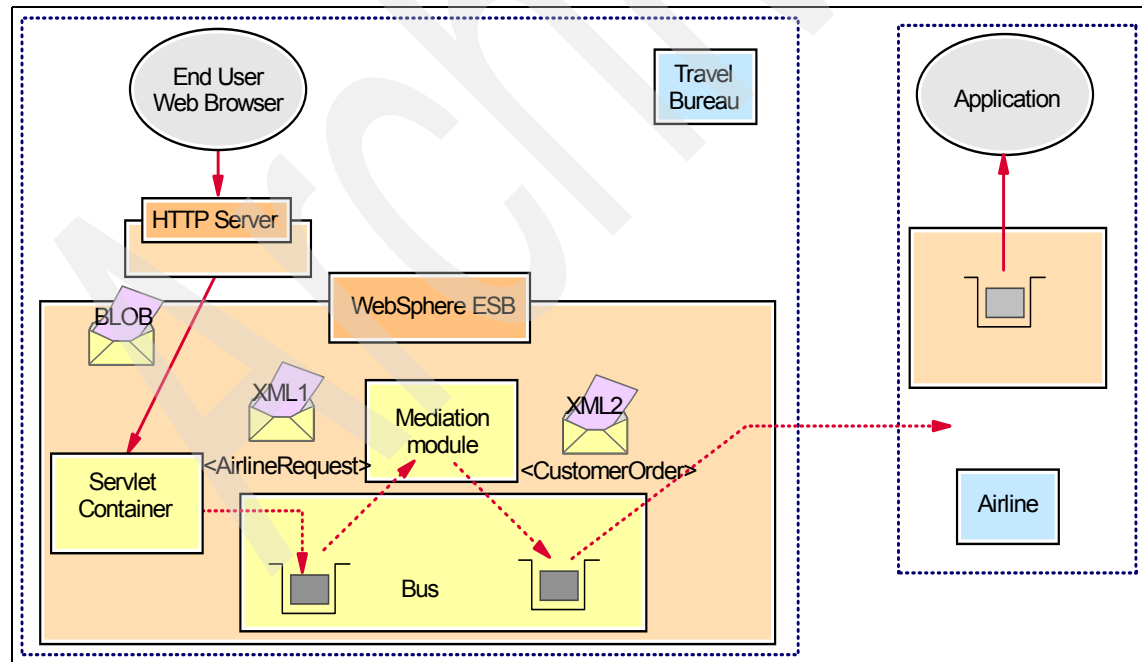
Figure 6-3   XML transformation with WebSphere ESB

6.3 XML-to-XML mapping using a mediation flow

This section shows how to build a mediation flow that transforms a message in one XML format to another XML format. WebSphere ESB includes a mediation primitive called the XSLT mediation primitive that can be placed in a mediation flow to provide this function.

In this example, two queue destinations are defined in the service integration bus. Messages enter the flow through the AIRLINE queue destination, are mediated, and leave the flow through the ORDER queue destination. Applications access these queues using the JNDI names.

The Export is bound to JMS and specifies jms/AIRLINE as the destination to receive messages from. The Import is also bound to JMS and specifies jms/ORDER as the destination to send messages to.

Configuration details are shown in Figure 6-4.
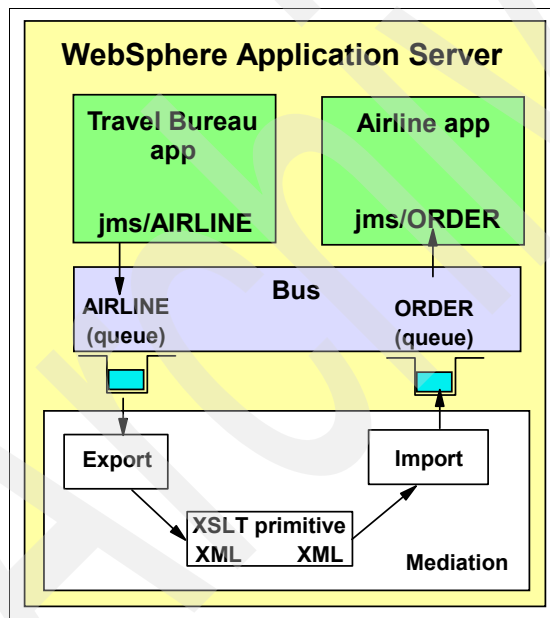
Figure 6-4   XML-to-XML mapping using the XSLT primitive

6.3.1 Mediation overview

Figure 6-5 on page 189 shows the mediation module contents we are going to build.
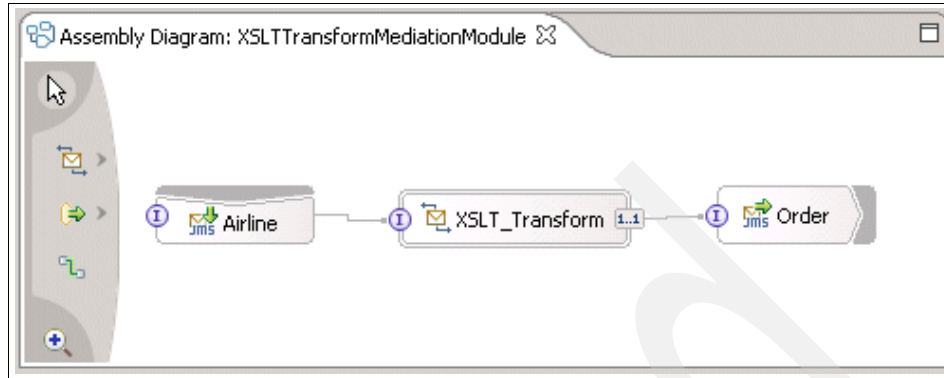
Figure 6-5   XSLT Transformation mediation component

XSLT_Transform is the mediation component. Its mediation flow implementation looks like Figure 6-6.
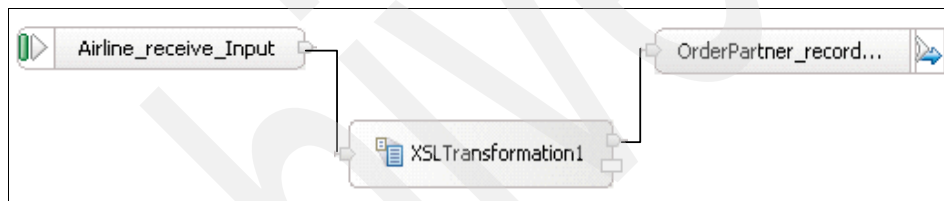
Figure 6-6   Mediation flow

This flow uses the XSLT mediation primitive. This performs the translation of the input XML to the output XML using an XSL style sheet.

6.4 XML-to-XML transformation using XSLT mapping

XSL Transformations (XSLT) is a language for transforming the structure of XML documents. It is designed for use as part of Extensible Stylesheet Language (XSL).

The steps required to build the mediation for XML transformation in WebSphere ESB are:

1. Create the mediation module.
2. Create the business objects.
3. Build the interfaces.
4. Build the mediation module.
5. Bind the export and import nodes to JMS.

6.4.1 Create the mediation module

The first step is to create a mediation module. You can optionally create a library first to hold resources, such as business objects or interfaces, that you may use in other modules. If you create a library, you can add it as a dependency while you create the mediation module.

1. Create a mediation module by right-clicking in the Business Integration view and selecting **New** → **Mediation module**.

2. Name the mediation module XSLTTransformMediationModule. Allow the wizard to create the mediation flow component.

6.4.2 Create the business objects

Next, we take you through the process of creating the seven business objects listed under Data Types in Figure 6-7. These data objects will be used to represent the XML data as it flows through the mediation.

Note that the business objects will be structured in the same manner as the Airline1.xml and Order1.xml files seen in "Sample XML files" on page 270.
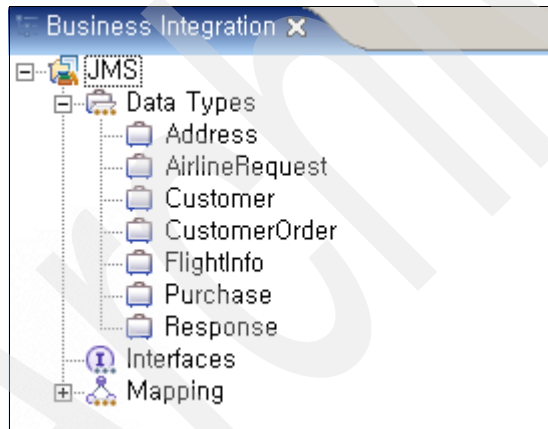
Figure 6-7   Create business objects

To create each business object:

1. Select **Data Types** in the XSLTTransformMediationModule and right-click.

2. Select **New** → **Business Object**.

3. Enter the name in the Name field.

4. Click **Finish**.

To add an attribute to a business object:

1. Select the object, right-click, and select **Add Attributes**.

2. Type over the name of the attribute with the new name, or change it in the Properties view.

3. Select the correct type of attribute (the default is string).

Tip: If you are adding a business object as an attribute (versus a string), you can simply drag and drop the business object from the Business Integration view onto the business object in the canvas.

Create the input business objects

The following business objects are used for the XML input:

1. Create the following business objects:

   – AirlineRequest
   – Customer
   – Purchase
   – Response

   As each business object is created it is opened in the editing area. Leave them open since we will be adding attributes to them next.

2. Add the following attributes to the Customer business object, as shown in Figure 6-8 on page 192:

   – FirstName - Type string
   – LastName - Type string
   – Street - Type string
   – City - Type string
   – Country - Type string
   – Zip - Type string

   You can add an attribute by clicking the **Add** icon in the window. The new attribute will be added to the list under Customer. Type over the name and choose the proper data type in the Properties view.
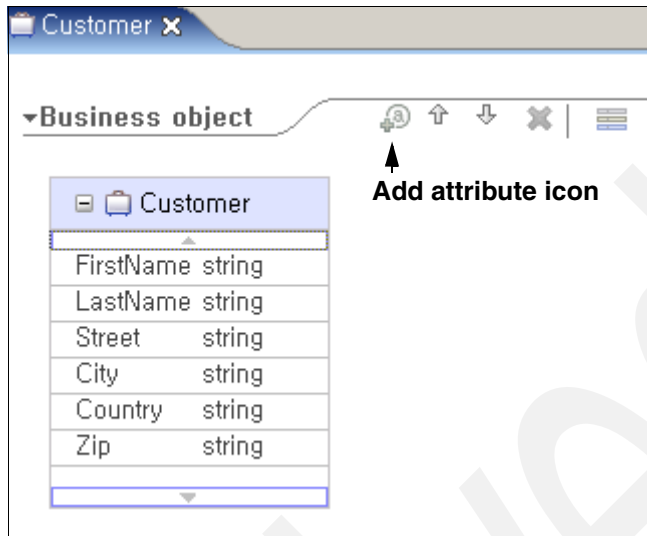
Figure 6-8   Create the Customer business object

Close the Customer business object.

3. Add the following attributes to AirlineRequest, as shown in Figure 6-9 on page 193:

   – Purchase - Type Purchase business object
   – Response - Type Response business object

Since these are business objects, you can drag and drop them onto the AirlineRequest object.
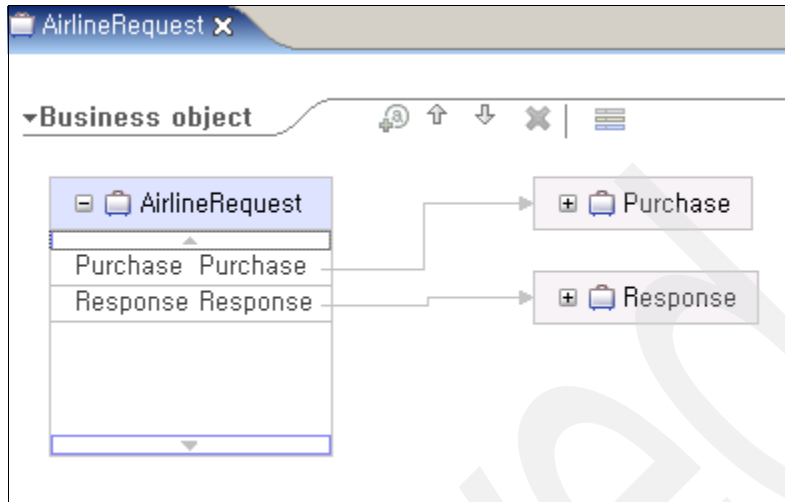
Figure 6-9   Create the AirlineRequest business object

Close AirlineRequest.

4. Add the following attributes in the Purchase business object, as shown in Figure 6-10 on page 194:

   – Customer - Type Customer business object
   – FlightNumber - Type string
   – Date - Type string
   – Price - Type string
   – CreditCard - Type string
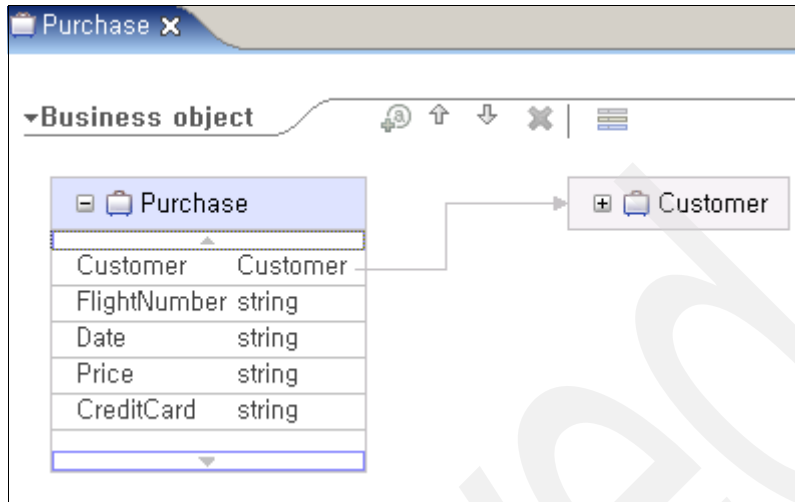
Figure 6-10   Create the Purchase business object

Close the Purchase business object.

5. Add the following attributes in the Response business object, as shown in Figure 6-11:

   – Status - Type string
   – Details - Type string
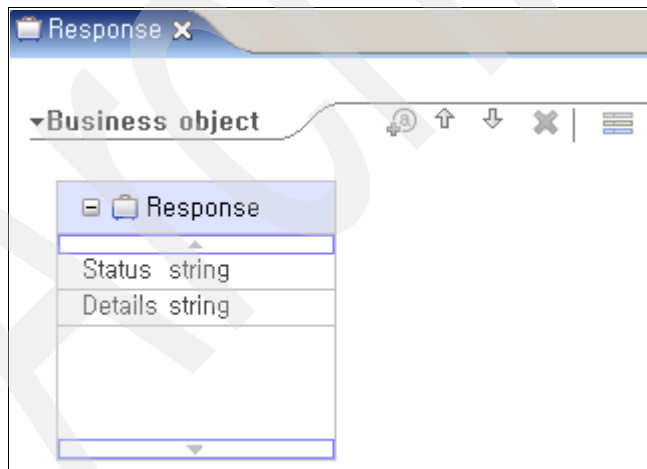
Figure 6-11   Create the Response business object

Close the Response business object.

Create the business objects for output

In this section we create the business objects for the XML output:

1. Create the following business objects:

    – CustomerOrder
    – Address
    – FlightInfo

2. In CustomerOrder, add the following attributes:

    – CustomerName - Type string
    – Address - Type is the Address business object
    – FlightInfo - Type is the FlightInfo business object

    You can see the changes in Figure 6-12.
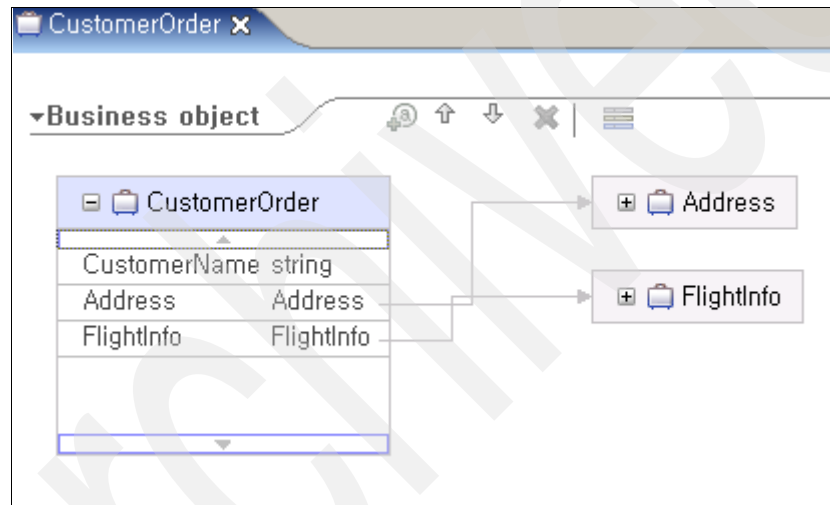
Figure 6-12   Create Business Object of CustomerOrder

Close the CustomerOrder business object.

3. Add the following attributes to Address, as shown in Figure 6-13 on page 196:

    – Street - Type string
    – City - Type string
    – Country - Type string
    – PostCode - Type string
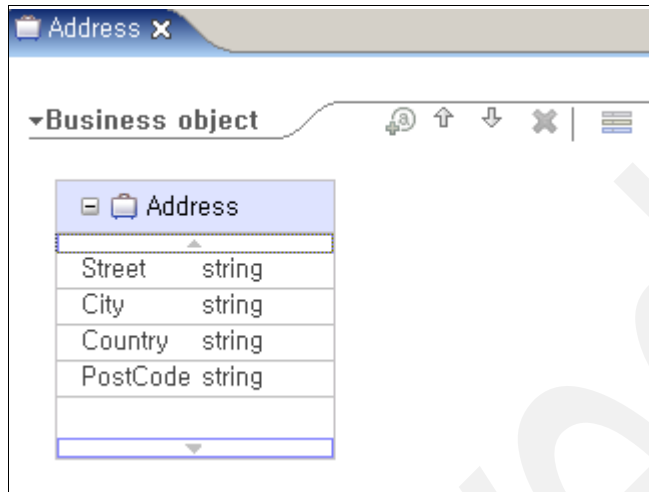
Figure 6-13   Create the attributes for the Address business object

Close the Address business object.

4. Create the following attributes for the FlightInfo business object, as shown in Figure 6-14 on page 197:

   – FlightNo - Type string
   – Date - Type string
   – Cost - Type string
   – CardNo - Type string
   – Membership - Type string
   – Status - Type string
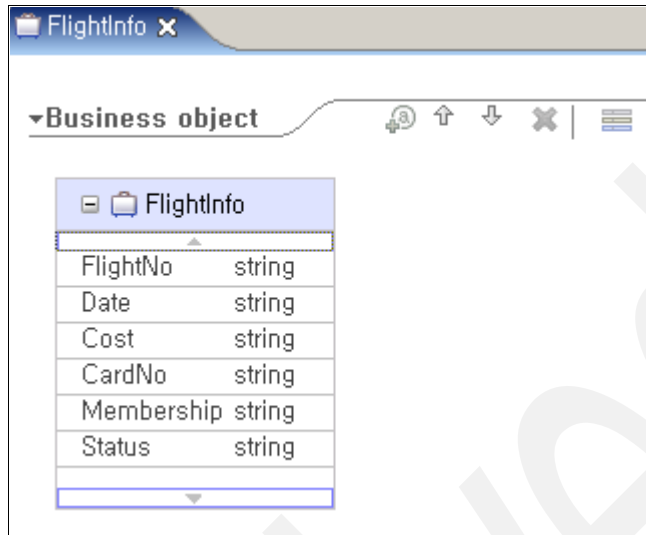
Figure 6-14   Create the attributes for the FlightInfo business object

Close the FlightInfo business object.

6.4.3 Build the interfaces

We need two interfaces for this service. A service can be called synchronously or asynchronously.

A component can be called synchronously or asynchronously; this is independent of whether the implementation is synchronous or asynchronous. The interfaces on the component are defined in the synchronous form and asynchronous support is also generated for them. For an interface you can specify a preferred interaction style as synchronous or asynchronous.

The asynchronous type advertises to users of the interface that it contains at least one operation that may take a significant time to complete. As a consequence, the calling service should avoid keeping a transaction open while waiting for the operation to complete and send its response. The interaction style applies to all the operations in the interface.

These interfaces define an asynchronous invocation.

Build the input interface

An interface to a component contains one or more operations that describe the action implemented by the component. An operation may be a request/response type, meaning a request is sent and a response is returned to the interface, or a

one way type, meaning only an input is sent and there is no response needed. Each operation in the interface defines the data that can be passed in the form of inputs to and outputs from the component when the operation is invoked.

This first interface describes how to call the mediation. It identifies the AirlineRequest business object as the expected input.

To build the interface:

1. In XSLTTransformMediationModule, select **Interfaces**, right-click, and select **New** → **Interface**.

    – Enter `Airline` in the Name field.

    – Click **Finish**.

    The new interface will open in the editor area.

2. Since our example flow is not expecting a response to be returned, the operation will be a one-way operation. In the editor area, click the Add One Way Operation icon (see Figure 6-15).

    A new operation will be added to the interface with the default name of Operation1. Change this name to `receive` by typing over it.

3. Click the **receive** operation and click the Add Input icon (see Figure 6-15). A new Input entry will appear on the canvas.

    – Enter `request` in the Name field.

    – Click in the Type field and select the **AirlineRequest** business object. The results are shown in Figure 6-15.

    (You can also use the Properties view to enter these values.)
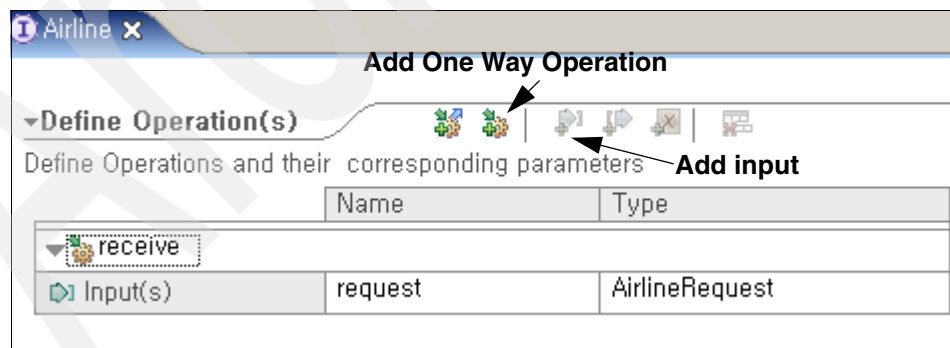
Figure 6-15   Interface for Input

4. Close and save the interface.

Build the output interface

The next interface is used for the import that invokes the Airline company. It specifies the CustomerOrder business object as the data to pass.

1. Create a second interface named Order.

2. Click the Add One Way Operation icon.

   Change the name to `record`.

3. Click the Add Input icon.

   Enter `order` in the Name field and select the **CustomerOrder** business object as the Type. The results are shown in Figure 6-16.
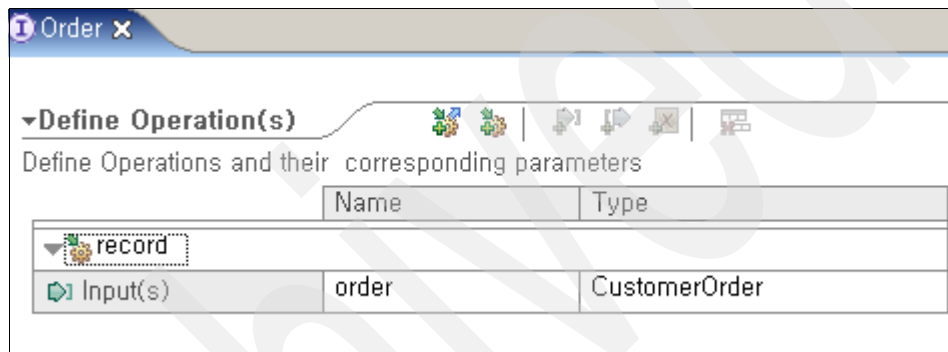
Figure 6-16   Interface for Output

4. Close and save the interface.

6.4.4 Build the mediation module

Next we populate the mediation module. First we add the SCA components (export, import, and mediation flow components) in the module assembly and wire them together. Next we generate and refine the implementation for the mediation flow component. The implementation is the mediation flow.

Add the components to the module assembly

First we add the export to be used to invoke the module. Then we add the import that invokes the airline service. Then we add the interfaces and references and wire the components together.

1. Navigate to the module assembly in the Business Integration view (Figure 6-17 on page 200). This was created automatically when you created the meditation module. Open the module assembly by double-clicking it.
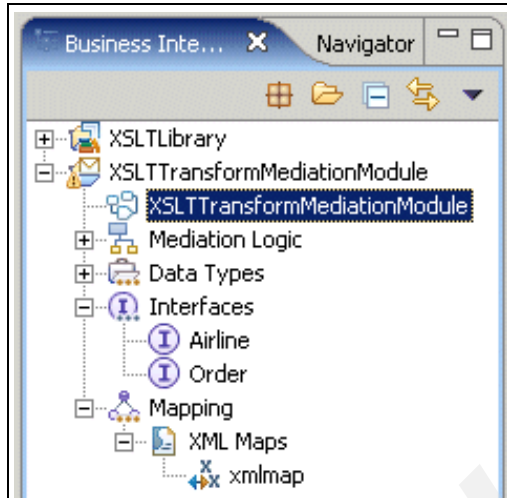
Figure 6-17   Module assembly

The assembly editor will open and you will see that a mediation flow component called Mediation1 has been added for you.
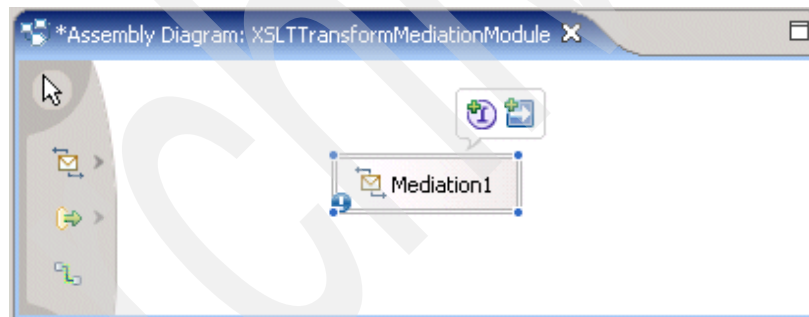
Figure 6-18   Mediation flow component

2. Select the mediation flow component. In the Properties view, change the display name from Mediation1 to XSLT_Transform.

3. Right-click in the canvas and select **Add Node** → **Export**. Move the new Export (Export1) to the left of the mediation flow component.

   In the Properties view, change the display name to Airline.

4. Right-click in the canvas and select **Add Node** → **Import**. Move the new Import (Import1) to the right of the mediation flow component.

   In the Properties view, change the display name to Order.

5. Right-click the Airline export and select **Add Interface**. Select **Airline**.

6. Right-click the **XSLT_Transform** component and select **Add → Interface**. Select **Airline** and click **OK**.

7. Right-click the **XSLT_Transform** component and select **Add → Reference**. Select **Order**. The new reference will be added with the name of OrderPartner.

8. Right-click the **Order** import and select **Add Interface**. Select **Order**.

9. Click the Wire icon and then click the **Airline** export. Drag the wire to the interface on XSLT_Tranform. Do the same to connect the XSLT_Transform reference to the Order import node.
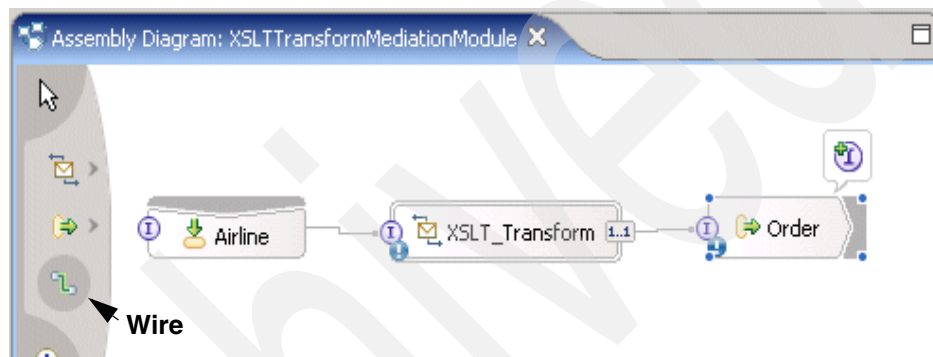
Figure 6-19   Wire the components

Click the arrow at the top left of the palette to get out of the wiring mode.

Build the mediation flow

The implementation of the mediation flow component is the next item we build. The implementation is a mediation flow.

1. Right-click **XSLT_Tranform** and select **Generate Implementation** to generate the mediation flow. Click **OK** in the next window. The Mediation Flow editor opens with the new flow (Figure 6-20 on page 202).
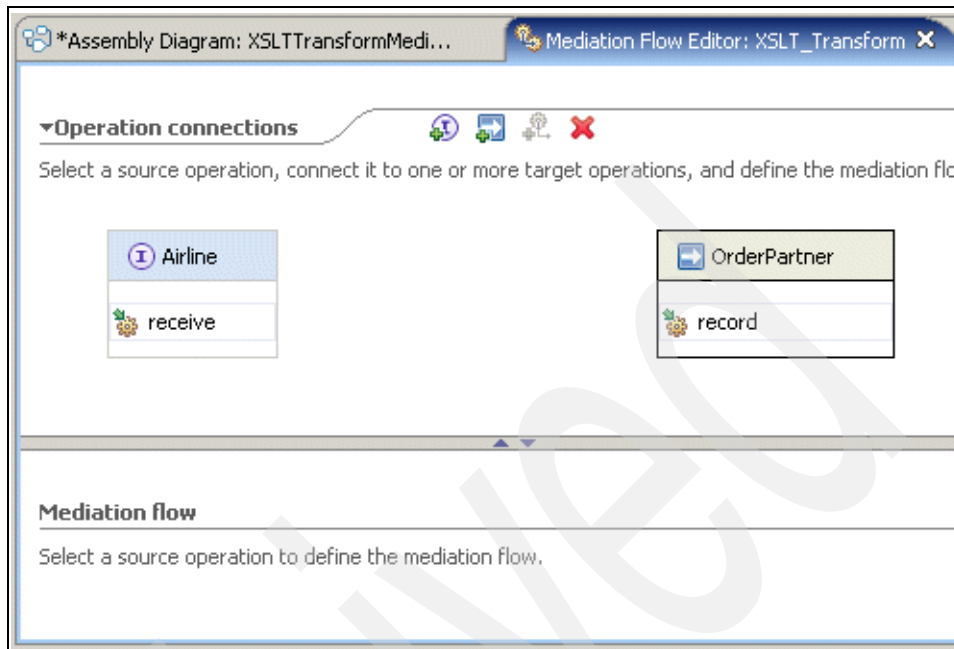
Figure 6-20   Mediation flow editor

2. The end points of the mediation flow component are defined by source interfaces and target references. Source operations are connected to target operations and a flow is defined for each connection.

   In the Operation connections window, you will see the interface and the reference used in this flow. Click the receive operation (you will see an input node in the Mediation flow pane) and drag it to the record operation, creating an operation connection between the two. This will generate the Callout node in the flow.

3. In the pane below, you will see the input and callout nodes. To the left you will find icons representing mediation primitives.

   Click the icon for XSL Transformation, and then click in the editor pane to add the primitive between the two generated nodes.

   The results are shown in Figure 6-21 on page 203.
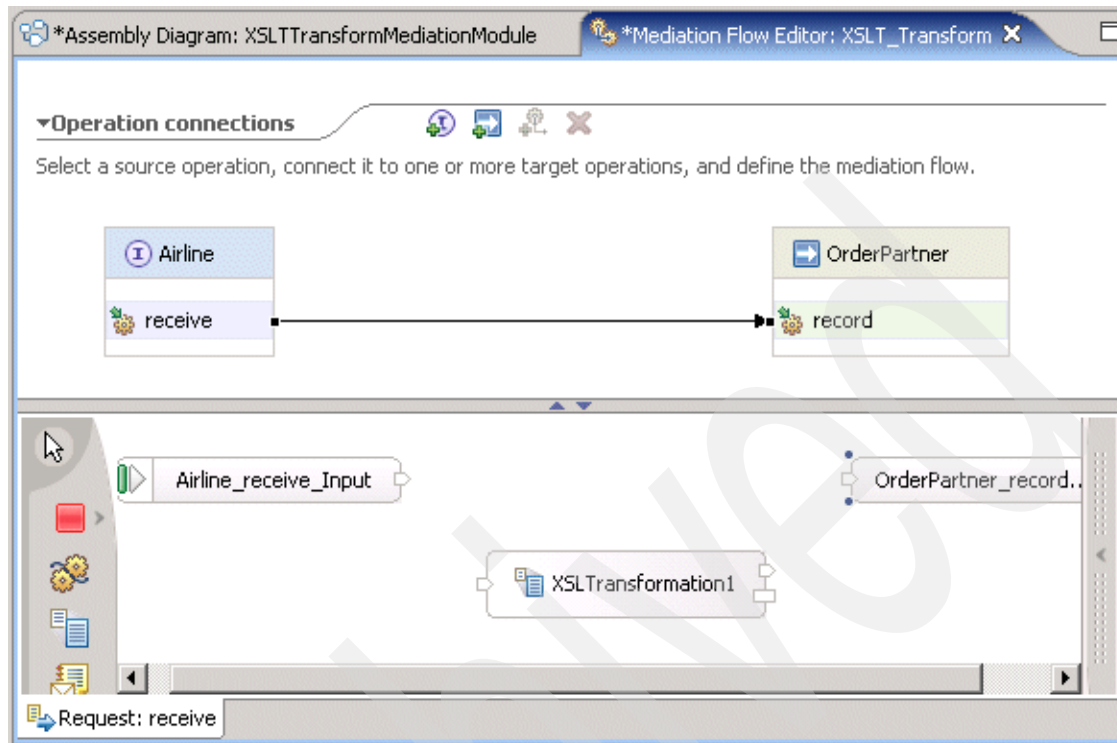
Figure 6-21   Meditation flow

4. Wire the out terminal of the Airline receive Input node to the in terminal of the
    XSLTransformation1 node. To do this, click the out terminal and drag it to the
    in terminal.

5. Wire the out terminal of the XSLTransformation1 node to the in terminal of the
    OrderPartner record Callout node.

> **Note:** When you dropped the XSLT primitive onto the canvas, the
> message types of the in and out terminals were null. As you wire the input
> node to the XSLT primitive, the type of the input node's out terminal
> (receiveRequestMsg) is propagated to the in terminal of the XSLT
> primitive. Similarly, when you wire the out terminal of the XSLT primitive to
> the callout node, the message type of the callout node's in terminal
> (recordRequestMsg) is propagated to out terminal of the XSLT primitive.
>
> This becomes important later when the mapping editor is used to define
> the mapping for the XSLT primitive.
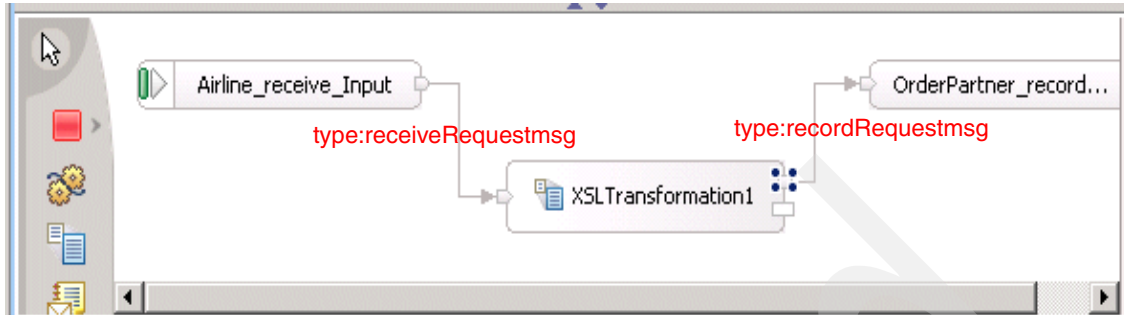
Figure 6-22   Connect the nodes in the flow

6. To set the properties of the XSLTransformation1 node, select the node:

a. In the Properties view click the **Details** tab.
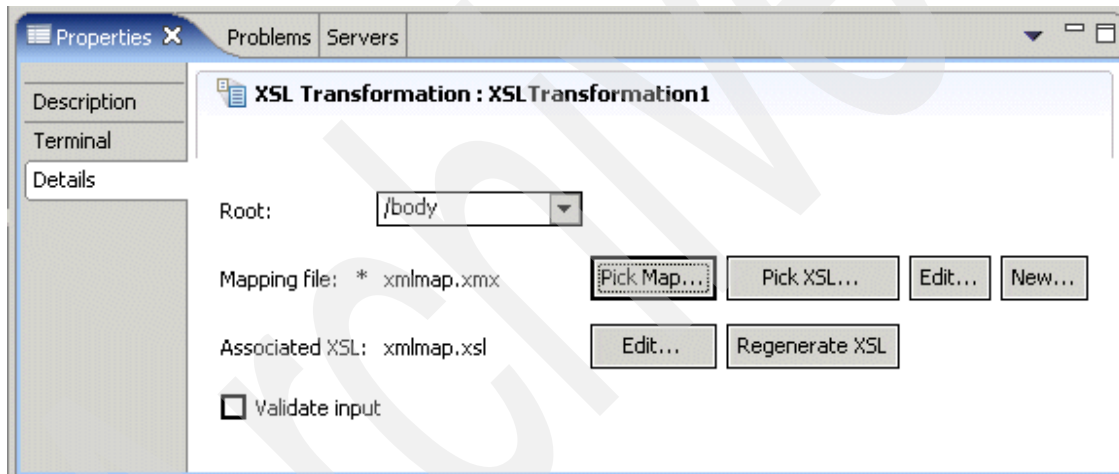
Figure 6-23   XSL transformation properties

The Root field specifies the root of the SMO message to use for the source and target message during transformation. Valid values are:

- / for the complete SMO
- /body for the body section of the SMO
- /headers for the headers of the SMO
- /context for the context of the SMO

For this example, the Root field has /body specified, so the mapping is made of the message body.

If you have an existing XSL style sheet file you can specify it here instead by using the **Pick Map** button. If not, click **New** to create an XML mapping.

b. Since the XSLT primitive is wired to the input and callout nodes, the wizard knows the input and output message types that to be mapped. Click **Finish** to launch the XML Mapping editor.

7. A mapping editor will be opened. Elements can be mapped by dragging the source element and dropping it on the target element. Map the elements as shown in Table 6-1.

   To map FirstName and LastName to CustomerName, click both source fields using the Ctrl key, and then drag and drop them onto CustomerName.

Table 6-1   XML Mapping

AirlineRequest	**CustomerOrder**
FirstName	CustomerName
LastName	CustomerName
Street	Street
City	City
Country	Country
Zip	PostCode
FlightNumber	FlightNo
Date	Date
Price	Cost
CredirCard	CardNo
Status	Status
Details	Membership

– The final mapping should look like Figure 6-24 on page 206.
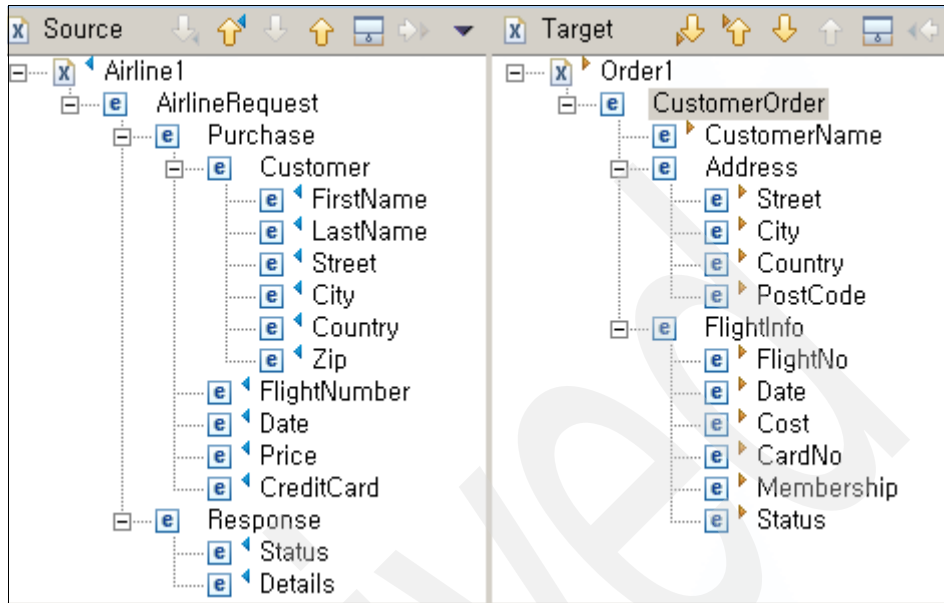
Figure 6-24   XSLT mapping

- To add a blank between FirstName and LastName:

    i.   Select **CustomerName** in the Overview view. Right-click and select
         Define XSLT Function.

    ii. Select **String**, and click **Next**.

    iii. Select **concat** as the function (default). Then click **Add**.

    iv. Enter a blank surrounded by quotes (' ') as the parameter value and
         click **OK**.

    v.   Select the quotes in the Input Parameters window and move them
         between FirstName and Last Name using the Up and Down buttons, as
         shown in Figure 6-25 on page 207.
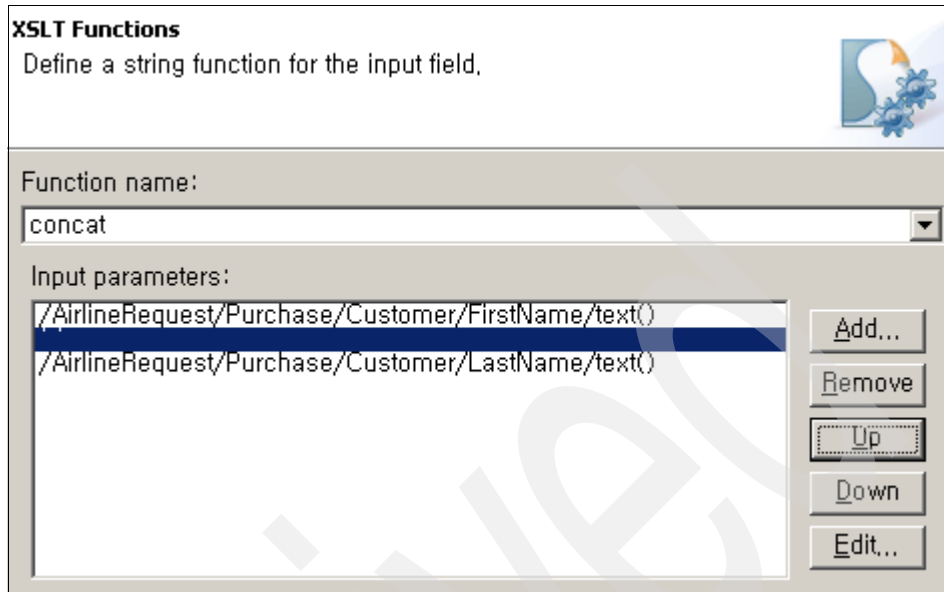
Figure 6-25   Define an XSLT function

      vi. Click **Finish**.

      If the function is defined correctly, you will see the concat function listed in the Applied Function column (Figure 6-26).

Figure 6-26   New applied XSLT function

8. Close and save the map.

9. In the Properties view click the **Regenerate XSL** button to generate an XSL style sheet from the XML map.

10. Close the XSL file.

11.Close and save the mediation flow.

6.4.5 Bind the export and import nodes to JMS

The next step is to set up the export and import nodes to use JMS.

Export node setup

Open the XSLTTransformMediationModule in the assembly editor and do the following:

1. Select the Airline node and right-click. Select **Generate Binding** → **JMS Binding**.

   a. In the JMS Binding Attributes Selection window, select **Point-to-Point** as the JMS messaging domain and **Text** for data serialization.

   b. Click **OK**.

2. In the Properties view:

   a. Select the Binding tab. Enter `jms/AS` as the JNDI Lookup Name.

   b. Select the JMS Destinations tab and expand **Receive Destination Properties**. Enter the JNDI Lookup Name as `jms/AIRLINE`.

Import node setup

In the Assembly editor:

1. Select the **Order** node in the Assembly editor and right-click. Select **Generate Bindings** → **JMS Binding**.

   a. In the JMS Binding Attributes Selection window, select **Point-to-Point** as the JMS messaging domain and **Text** for data serialization.

   b. Click **OK**.

2. In the Properties view:

   a. Select the **Binding** tab. On the JMS Import Binding tab, enter `jms/QCF` as the JNDI Lookup Name.

   b. Select the JMS **Destinations** tab and expand **Send Destination Properties**. Enter `jms/ORDER` as the JNDI lookup name.

   The assembly diagram will now look like Figure 6-27 on page 209.
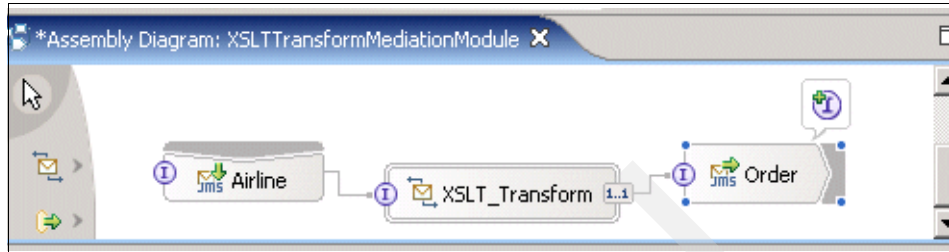
Figure 6-27   Mediation module with a JMS binding

3. Save the changes.

6.4.6 Prepare the runtime

Before building the mediation, some preparation is needed to set up the rest of the runtime environment.

The following needs to be done to prepare WebSphere ESB:

► Create the bus destinations (queues).
► Create the JMS objects.

Create the bus destinations (queues)

Using the administrative console for the WebSphere ESB server:

1. Select **Service Integration** → **Buses**. Note that three buses have been created for you during installation.

2. Click the **SCA.APPLICATION.esbCell.Bus**, then click **Destinations**.

3. Create the following queues using default settings:

   – AIRLINE
   – ORDER
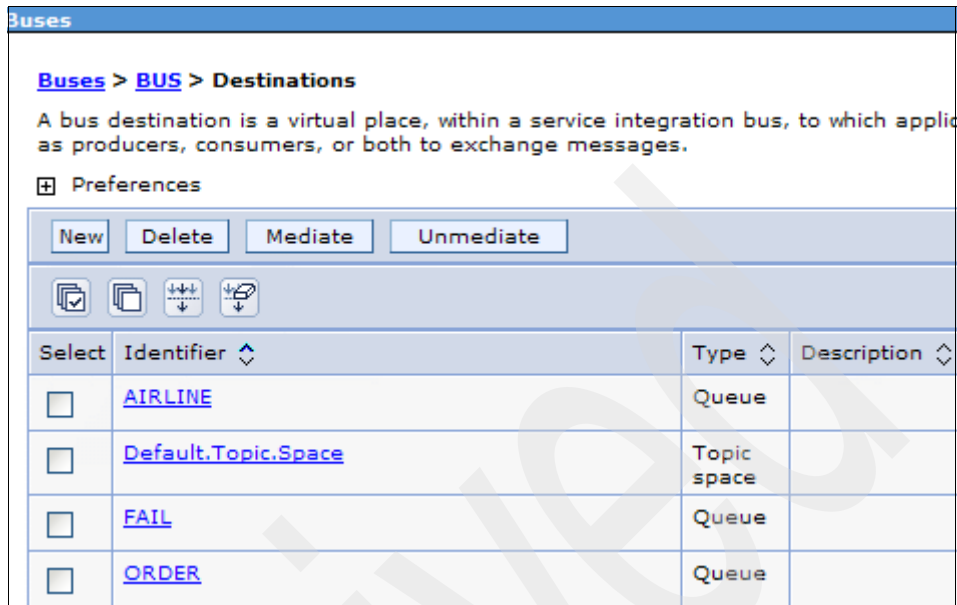
Figure 6-28   Create the queue destinations in WebSphere ESB

4. Save all changes.

WebSphere ESB and WebSphere MQ integration: Earlier, in "Connect WebSphere ESB to WebSphere MQ" on page 118, you saw how to connect the service integration bus in WebSphere ESB to WebSphere MQ. In this example you can use the same techniques to send the translated message to a queue on WebSphere MQ. The basic steps are:
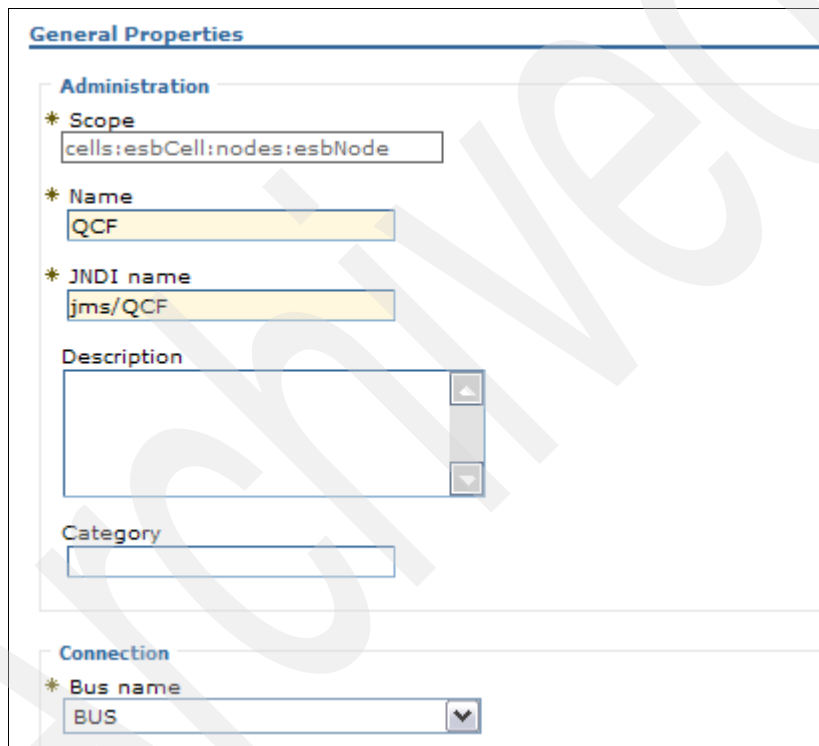
1. On WebSphere ESB:

   – Define WebSphere MQ as a foreign bus to SCA.APPLICATION.esbCell.Bus.

   – Define the WebSphere MQ link, including the sender and receiver channels.

   – Delete the ORDER queue destination and create an alias destination named ORDER, pointing to a queue on WebSphere MQ.

2. On WebSphere MQ:

   – Create the queue manager, sender channel, receiver channel, and transmission queue to complete the connection with WebSphere ESB.

   – Create the queue that the ORDER alias points to.

Create the JMS objects

Create the JMS objects required by doing the following:

1. Select **Resources** → **JMS Providers** → **Default Messaging**.

2. Click **JMS Queue Connection Factory** in Connection Factories.

3. Click **New**.

    a. Enter `QCF` in the Name field.
    b. Enter `jms/QCF` in the JNDI Name field.
    c. Select **SCA.APPLICATION.esbCell.Bus** in the Bus Name field.
    d. Click **OK**.
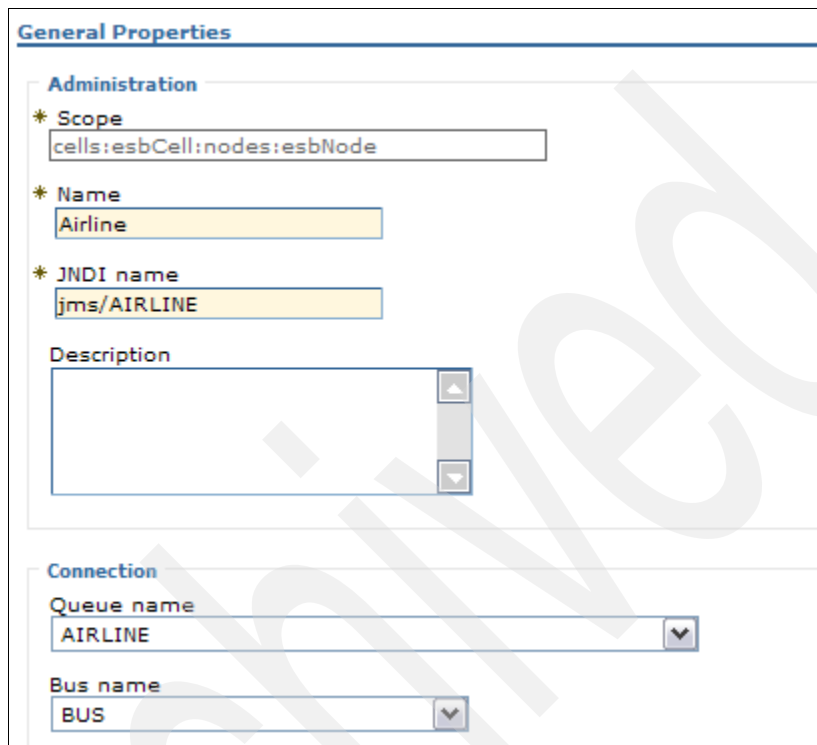
Figure 6-29   Create a New JMS queue connection factory

4. Save the changes.

5. Select **Resources** → **JMS Providers** → **Default Messaging**.

6. Click **JMS Queue**.

7. Click **New**.

    a. Enter `Airline` in the Name field.
    b. Enter `jms/AIRLINE` in the JNDI Name field.

c. Select **SCA.APPLICATION.esbCell.Bus** as the Bus Name.

d. Select **AIRLINE** as the queue name.

e. Click **OK**.

Figure 6-30   Attributes of the JMS queue

8. In the same way create the Order JMS queue using the following values:

   – Order queue:

     • The name is `Order`.
     • The JNDI name is `jms/ORDER`.
     • The bus name is `SCA.APPLICATION.esbCell.Bus`.
     • The queue name is `ORDER`.

   You can see the new JMS queues in Figure 6-31 on page 213.
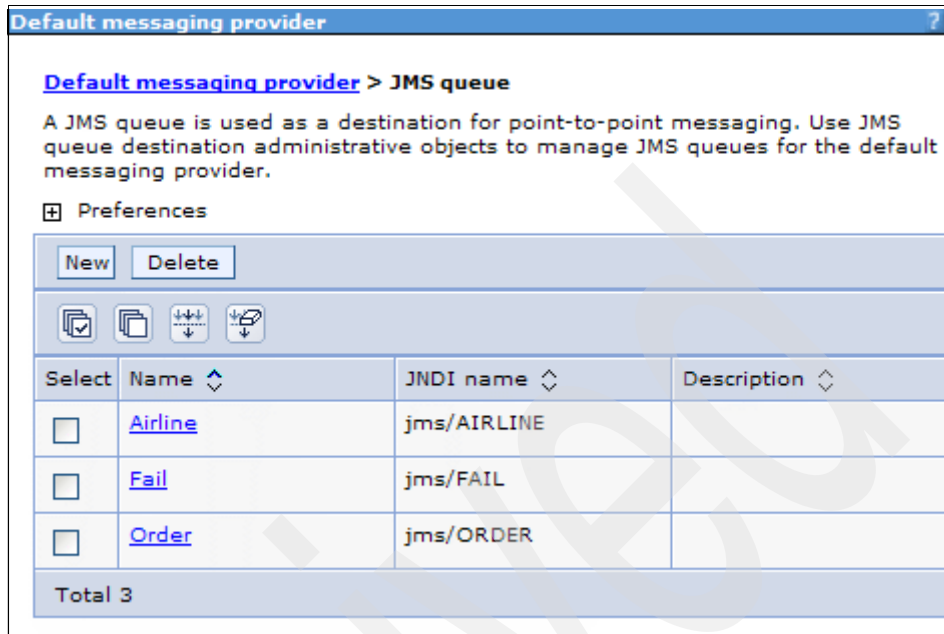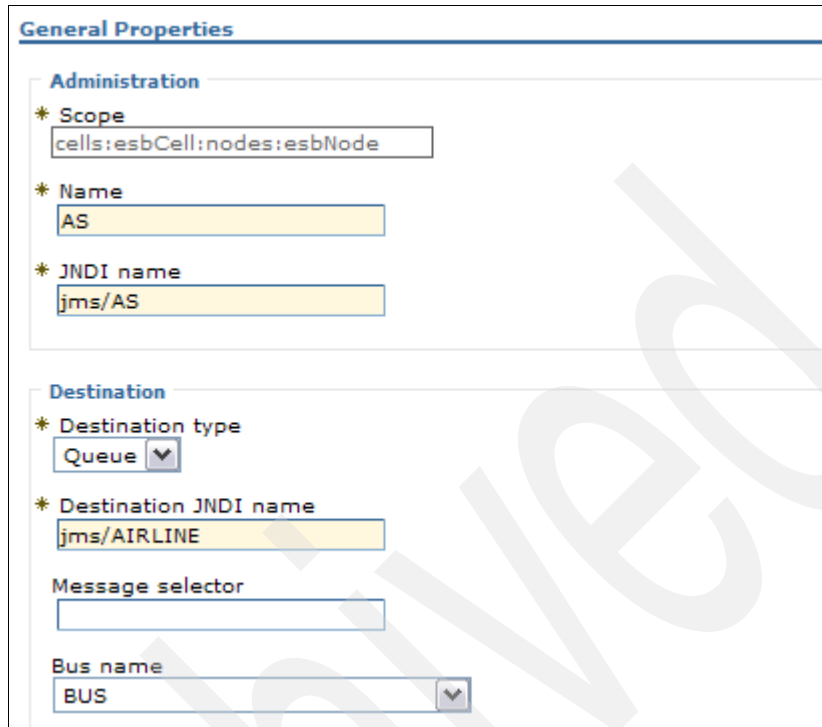
Figure 6-31   Create the new JMS queues

9. Save the changes.

10. Select **Resources** → **JMS Providers** → **Default Messaging**.

11. Click **JMS Activation Specification**.

12. Click **New**.

   a. Enter `AS` in the Name field.
   b. Enter `jms/AS` in the JNDI Name field.
   c. Enter `jms/AIRLINE` in the Destination JNDI Name field.
   d. Select **SCA.APPLICATION.esbCell.Bus** as the Bus Name.
   e. Click **OK** to create the activation specification.

Figure 6-32   Create a new JMS activation specification

13.Save the changes and restart the server.

Deploy the mediation

Mediation modules are deployed to the WebSphere ESB server as EAR files. To deploy to the server you must first export the module as an EAR file and make it available to the server. Then you can install the module as an application using the WebSphere administrative console.

To export modules as EAR files:

1. Select **File** → **Export**.

2. Select **Integration module** and click **Next**.

3. Check the box to the left of the mediation module. Select the **EAR files for server deployment** option and click **Next**.

4. In the Target directory field, type the path and name of the target directory where you want to export the EAR file. Note the EAR file name and click **Finish**.

To deploy the mediation, open the WebSphere administrative console and do the following:

1. Select **Applications** → **Install New Applications**.

2. Browse to the EAR file you exported and click **Next**.

3. In the next window, labeled `Preparing for the application installation`, click **Next**.

4. The next series of steps takes you through the application installation. This process is the same as for any WebSphere application. The number of steps and complexity of choices depends on the application and environment. In the sample we are using, all the default values were correct.

   Follow through with the installation process and save your changes.

5. To start the application select **Applications** → **Enterprise Applications**. Place a check in the box to the left of the mediation application and click **Start**.

Test the mediation

To test the mediation, WebSphere Integration Developer provides the integration test client. This is a useful tool for testing and debugging integration modules. For information about using it, refer to the WebSphere Integration Developer help.

To use the tool to test our mediation flow:

1. Open the module assembly.

2. Right-click in the canvas and select **Test Module**.

3. When the test client starts you will see a view that allows you to enter values for the input. Fill in the test values (Figure 6-33 on page 216) and click **Continue**.

Figure 6-33   Enter the values for the test client

The mediation test client sends the data to the mediation and the results are shown. You can also see the server messages in the server console.

7

Integration scenarios with WebSphere Message Broker

This chapter provides information and examples that illustrate some of the mediation functions possible when using WebSphere Message Broker as an enterprise service bus. The intent is to give you an idea of how message flows are built and deployed, and to introduce you to the concept of using predefind nodes that are provided by WebSphere MQ.

This chapter shows the following mediation examples:

► XML-to-XML mapping using a Mapping node
► XML-to-COBOL mapping
► XML-to-XML transformation using XSLT
► Routing messages

7.1 Using WebSphere Message Broker

Messages enter WebSphere Message Broker through an input node and traverse a set of nodes that form a message flow. The message flow processes the message before sending it to its final destination through an output node. This processing can be used to route and transform messages. WebSphere Message Broker provides you with a set of built-in nodes that facilitate message flow development.

WebSphere Message Broker provides the following transport support using the nodes listed.

Table 7-1   Application transport support

Nodes	Transport
► JMSInput ► JMSOutput	*WebSphere Broker JMS Transport* is used to allow a message flow to receive messages from JMS destinations or to send messages to JMS destinations. These destinations are accessible through connection to a JMS provider. The JMS nodes work with the WebSphere MQ JMS provider, WebSphere Application Server Version 6.0, the service integration bus, and any JMS provider that conforms to the Java Message Service Specification Version 1.1.
► SCADAInput ► SCADAOutput	*WebSphere MQ Telemetry Transport* is a lightweight publish/subscribe protocol flowing over TCP/IP. This protocol is used by specialized applications on small footprint devices that require a low bandwidth communication, typically for remote data acquisition and process control.
► Real-timeInput ► Real-timeOptimizedFlow ► Publication	*WebSphere MQ Multicast Transport* is used by dedicated multicast-enabled JMS application clients to connect to brokers. Applications communicate with the broker by writing data directly to TCP/IP ports. This protocol is optimized for high volume, one-to-many publish/subscribe topologies.
► HTTPInput ► HTTPReply ► HTTPRequest ► Publication	*WebSphere MQ Web Services Transport* allows Web services clients using XML messages and the HTTP protocol running over TCP/IP to communicate with applications through message flows in a broker.
► Real-timeInput ► Real-timeOptimizedFlow ► Publication	*WebSphere MQ Real-time Transport* is a lightweight protocol optimized for use with non-persistent messaging. JMS applications communicate with the broker using TCP/IP ports.

Nodes	Transport
► MQInput ► MQOutput ► MQGet ► MQReply ► MQOptimizedFlow ► Publication	*WebSphere MQ Enterprise Transport* supports WebSphere MQ applications that connect to WebSphere Business Integration Message Broker by writing data to and reading data from message queues.
► MQeInput ► MQeOutput ► Publication	*WebSphere MQ Mobile Transport* is used exclusively by WebSphere MQ Everyplace clients. WebSphere MQ Everyplace is an application designed primarily for messaging to, from, and between pervasive devices. These are typically small, handheld devices, such as mobile phones and PDAs. A bridge queue on the broker's queue manager provides an interface for the WebSphere MQ Everyplace clients to the broker services.

WebSphere Message Broker comes with a set of built-in nodes ready to use in building message flows. The built-in nodes shown in Table 7-2 show the broad range of function provided.

Table 7-2   Message manipulation nodes

Node	Function
Nodes for message manipulation	
► Compute ► JavaCompute	Used to examine a message and create new messages. The Compute node logic is written in ESQL. The JavaCompute node uses logic.
► Database ► DataDelete ► DataInsert ► DataUpdate ► Warehouse	Used to interact with an ODBC datasource.
► Extract	Used to extract the exact contents of the input message that you want to be processed by later nodes in the message flow.
► Mapping	Uses the Mapping node to construct one or more new messages by creating new messages and populating them with new information, with modified information from the input message, or with information taken from a database.
► XMLTransformation	Applies a stylesheet to an XML message.

Node	Function
▶ JMSMQTransform ▶ MQJMSTransform	Used to send messages to older message flows and to interoperate with WebSphere MQ JMS and WebSphere Message Broker publish/subscribe.
▶ AggregateControl ▶ AggregateReply ▶ AggregateRequest	Used to combine the generation and fan-out of a number of related requests with the fan-in of the corresponding replies, and compile those replies into a single aggregated reply message.
Nodes for decision making	
▶ Check	Validates the format of a message.
▶ Filter ▶ JavaCompute	Routes a message based on conditional logic. The Filter node uses ESQL logic, while the JavaCompute node uses Java.
▶ FlowOrder	Used to control the order in which a message is processed by a message flow.
▶ Label, RouteToLabel	Uses the Label node in combination with a RouteToLabel node to dynamically determine the route a message takes through the message flow, based on its content.
▶ ResetContentDescriptor	Used to request that the message is reparsed by a different parser.
▶ TimeoutControl ▶ TimeoutNotification	Used together in a message flow for an application that requires events to occur at particular times, or at regular intervals.
▶ Validate	Used to ensure that the message is routed appropriately through the message flow.
Nodes for error handling	
▶ Throw	Used to throw an exception within a message flow.
▶ Trace	Used to generate trace records that can incorporate text, message content, and date and time information, to help you to monitor the behavior of the message flow.
▶ TryCatch	Used to provide a special handler for exception processing.

If you need to perform processing that is not supported by the built-in nodes, WebSphere Message Broker allows the development of custom user-defined nodes. These can be written in either C or Java.

Additional nodes are available as SupportPacs and are free to download from the IBM Web site.

> **Note:** Always check that a specific node is compatible with all of the target platforms on which the message flow will be deployed.

7.1.1 Message flow development

The Message Brokers Toolkit provides the tools required to develop, deploy, and test message flows. The Message Brokers Toolkit consists of a Workbench window that displays one or more perspectives. A *perspective* is a group of views and editors required to perform tasks associated with a role. Each perspective consists of views that provide alternative presentations of resources or ways of navigating through information in the Workbench. As a user works with the Workbench, the data representing the projects and working environment are stored in a workspace directory on the local file system.

Message flow development is done using the Broker Application Development perspective (shown in Figure 7-1).
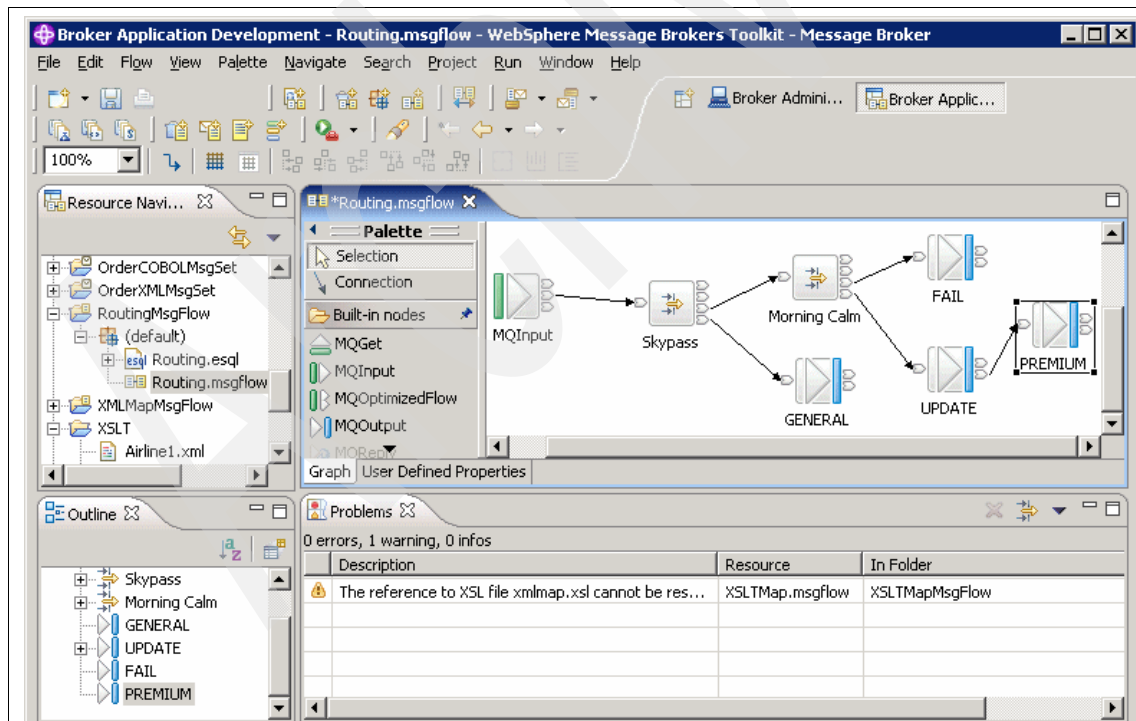
Figure 7-1   Broker Application Development perspective

This perspective is used to develop and test message flows and message sets. You can see in Figure 7-1 on page 221 that the perspective consists of views designed specifically for development. The Resource Navigator view (top left) lists the message flow resources. The editing area (top right) contains the canvas where nodes are placed in the flow, node properties are set, and the nodes connected together. Nodes can be placed on the canvas using drag and drop from the Pallete. The Problems view (bottom right) shows informational, warning, and error messages that indicate problems within resources. The Outline view displays a structured outline of the file open in the editor area (palette).

7.1.2 Message flow deployment and broker administration

Message flows are packaged into broker archive (bar) files and deployed to execution groups on a broker. An execution group is a named grouping of message flows that have been assigned to a broker. The broker does the actual processing of the message flow.

The Message Brokers Toolkit provides broker administration functions through the Broker Administration perspective (Figure 7-2).
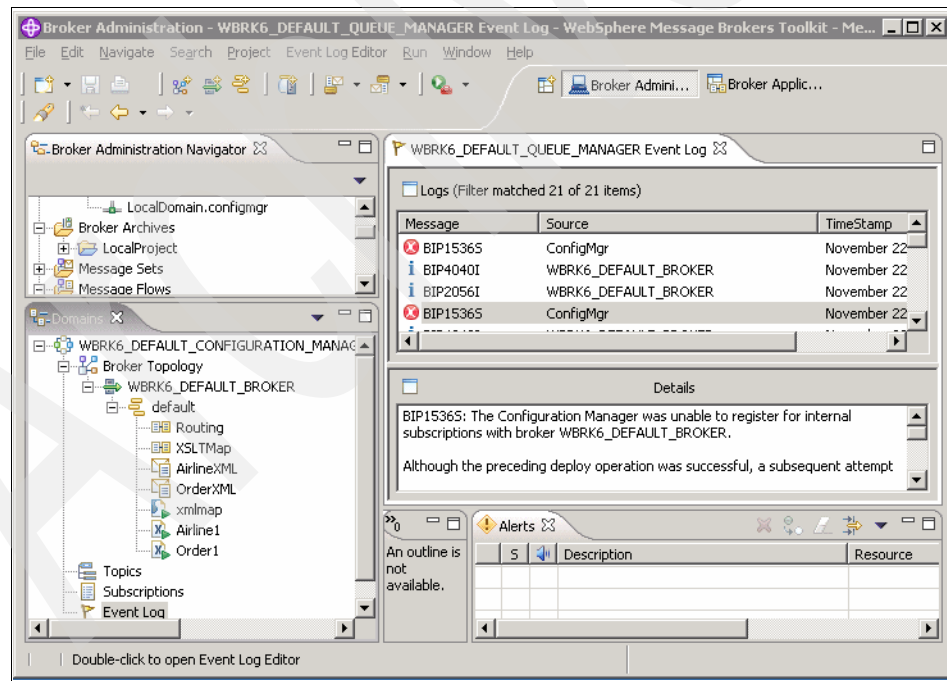
Figure 7-2   Broker Administration perspective

You can use this interface to:

- ► Create and manage broker domains and topology.
- ► Create and manage execution groups.
- ► Create and deploy broker archive (bar) files to execution groups.
- ► Manage publish/subscribe topics and subscriptions.
- ► Manage event logs and alerts.

A single Message Brokers Toolkit can connect to multiple Configuration Managers, thus allowing it to manage multiple broker domains.

7.1.3 Sample message flow

The message flow shown in Figure 7-3 illustrates some of the functionality you can achieve in a message flow. This example is a simple scenario for routing messages from a travel bureau to the proper airline company based on the message content and transforming the messages if needed. This message flow was created using built-in nodes.
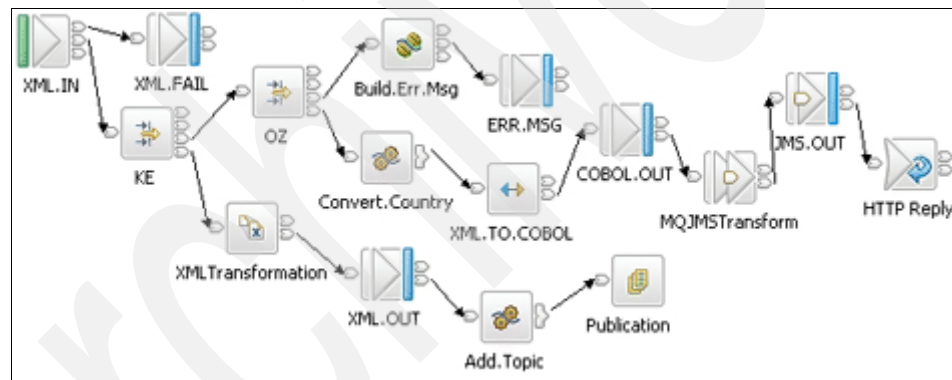
Figure 7-3   Sample message flow

Each segment of the message flow is discussed in the following sections.

Initial processing

Figure 7-4 on page 224 shows the portion of the sample message flows that routes the message to the proper destination based on the message contents.
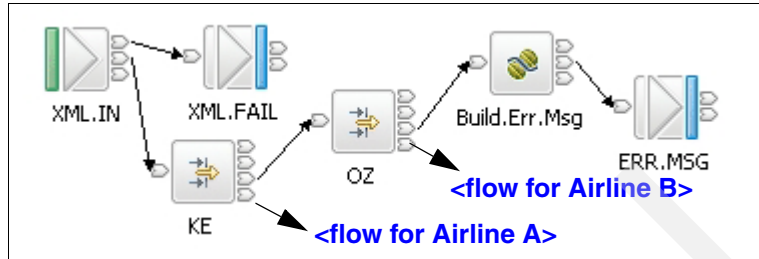
Figure 7-4   Message flow for the travel bureau

The process is:

1. A message is placed on a queue and enters the message flow through the XML.IN MQInput node.

2. If the message is not in XML format it is sent through the failure terminal of the MQInput node to the MQOutput node, XML.FAIL. The message ends up on a queue designated for errors and the flow ends.

   If the message format is valid, the message is sent to the KE Filter node.

3. The KE Filter node looks at the message data. If the FlightNumber string in the XML data starts with "KE", the message is sent over the true terminal to the portion of the message flow that transforms data for Airline A. (See "Airline A" on page 224 for a continuation of the scenario.) Otherwise the message is sent through the false terminal to the OZ Filter node.

4. The OZ Filter node looks at the message data. If the FlightNumber string in the XML data starts with "OZ", the message is sent over the true terminal to the portion of the message flow that transforms the data for Airline B. (See "Airline B" on page 225 for a continuation of the scenario.)

   If the FlightNumber string in the XML data does not start with either, the message is sent to the Build.Err.Msg compute node, where it is changed to an error message. The message is then sent to the ERR.MSG MQOutput node, where it ends up on a queue designated as an error queue.

Airline A

Figure 7-5 on page 225 shows the portion of the message flow that handles data bound for Airline A. The message sender uses a different XML format than Airline A uses so transformation of the message is necessary before sending it on to the airline.
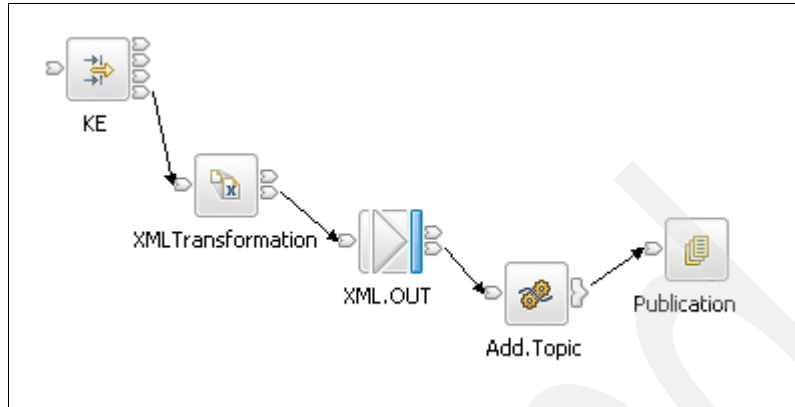
Figure 7-5   Message flow for Airline A

The process is:

1. The KE Filter node has determined the message is to be sent to Airline A. The message is sent to the XMLTransformation node where the XML message is transformed from its original XML format to a new format required by Airline A.

   The message is then sent to the XML.OUT MQOutput node.

2. The MQOutput node sends the message to the Add.Topic Compute node. The message is added to the Publication topic and sent to the Publication node.

3. The Publication node publishes the message to a pub/sub application in Airline A.

Airline B

Figure 7-6 on page 226 shows the portion of the message flow that handles data bound for Airline B. Airline B uses a COBOL application so the message must be transformed before being sent.
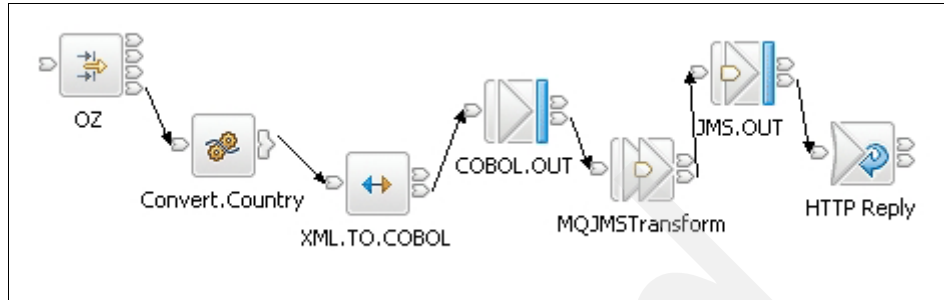
Figure 7-6   Message flow for Airline Company B

The process is:

1. The Convert.Country Compute node updates the Country string of the XML data to a type understood by Airline B. The list of countries used by the travel bureau and their equivalent representation for Airline B is kept in a database. It then sends the message to the next node in the flow.

2. The XML.TO.COBOL Mapping node transforms the XML data to COBOL data and sends the message to the next node.

3. The COBOL.OUT MQOutput node puts the COBOL message on a local queue.

4. The MQJMSTransform node changes the MQ header message to a JMS header and sends the message to the next node.

5. The JMS.OUT JMSOutput node sends the data to a JMS destination.

6. A reply is sent (HTTPReply node).

7.2 Integration scenarios

In the following sections we show you how to do XML-to-XML and XML-to-COBOL using WebSphere Message Broker. We use both MQ input and JMS input.

We show two methods of XML-to-XML mapping. The first uses a Mapping node. In this method, the message formats to be mapped are stored in a message set. The example we use loads DTD files that define the message format into the message set. The Mapping node allows you to map fields from one DTD file to another.

The second uses the XMLTransformation node. This node allows you to transform the input XML data to the output format using an XMLT style sheet.

The third scenario shows how to do XML-to-COBOL transformation using a Mapping node.

The last scenario illustrates how to route a message to a specific destination based on the message contents.

Figure 7-7 shows the runtime configuration for the scenarios illustrated in this chapter.
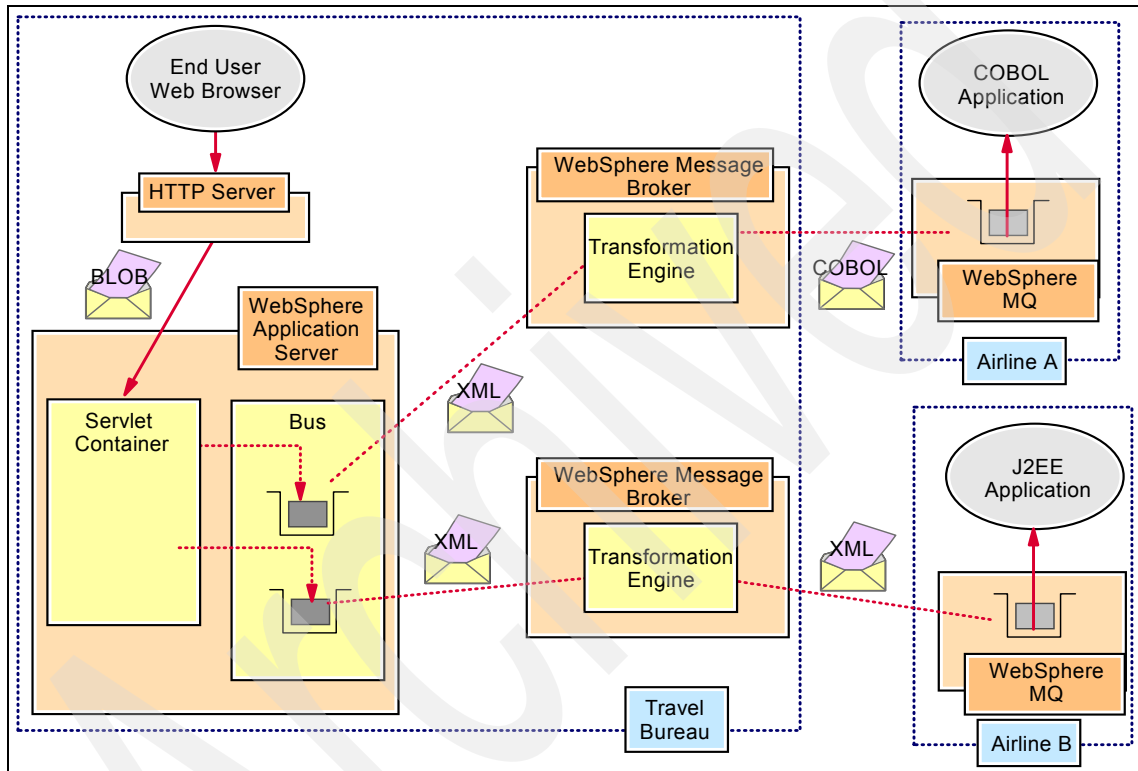
Figure 7-7   Integration scenario with XML to COBOL transformation

7.3 XML-to-XML mapping using a Mapping node

This section shows how to build a message flow that transforms a message in one XML format to another XML format.

The configuration for this scenario is shown in Figure 7-8 on page 228.
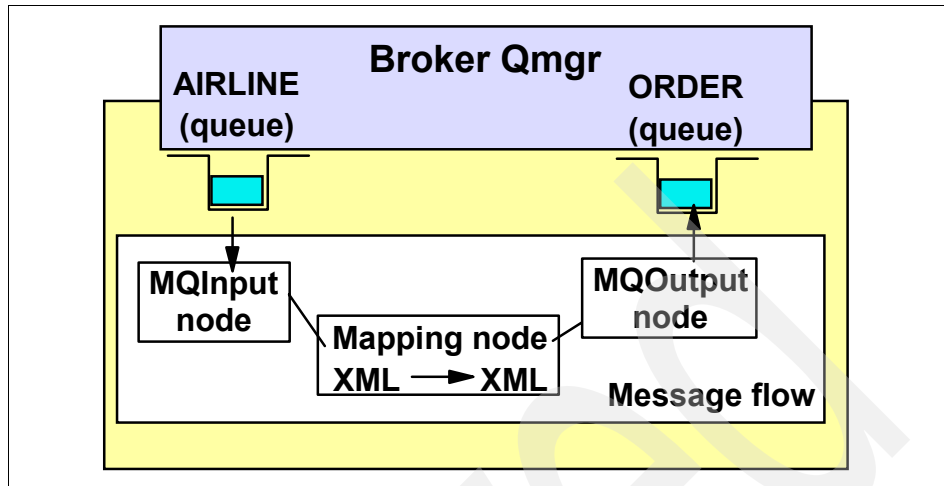
Figure 7-8   XML-to-XML mapping using a Mapping node

To continue the travel bureau theme, assume that the travel bureau is sending the airline reservation request in XML. The airline also uses XML, but the format is different.

The steps required to do this mapping are:

1. Create the message sets containing the XML DTD files.
2. Create the message flow.
3. Define the mapping.
4. Deploy the message flow to the broker.
5. Create the WebSphere MQ queues.
6. Test the message flow.

Last, we alter the message flow to use JMS nodes to illustrate the use of WebSphere Application Server as the JMS provider for input and output versus a WebSphere MQ queue.

7.3.1 Create the message sets containing the XML DTD files

A message set is used to contain message definitions. You would typically populate your message set by importing application message formats described by XML DTD, XML schema, WSDL definitions, IBM-supplied messages, C structures, or COBOL structures. In this example, we use document type definition (DTD) files.

This step shows you how to create two message sets containing the XML DTD files that define the XML formats to be mapped.

Create the airline message set

First, we define the message set for the travel bureau XML format. Using the Broker Application Development perspective in the Message Brokers Toolkit, do the following:

1. Create a message set project named AirlineXMLMsgSet.

    – Create a message set named AirlineXML.
    – Set the XML Wire Format Name to XML1.

    The new message set will open in the editor. Close the file.

2. Create a new DTD file that reflects the structure of the input message XML. In this example, the DTD file is called airline.dtd. You can see the contents in "Sample DTD files" on page 271.

3. Create a new project (**New** → **Project** → **Simple**) and import airline.dtd into it (**File** → **Import** → **File System**).

4. Create a new message definition file:

    a. Right-click **AirlineXMLMsgSet** in the Navigator pane.
    b. Select **New** → **Message Definition File**.
    c. Select **XML DTD File** and click **Next**.
    d. Select **airline.dtd** and click **Next**.
    e. Select the **AirlineXML** message set and click **Next**.
    f. Select **AirlineRequest** in the Global Elements list.
    g. Click **Finish**.

Create the order message set

Next create the message set containing the DTD file that describes the format used by the airline:

1. Create a message set project named OrderXMLMsgSet:

    a. Create a message set named OrderXML.
    b. Select XML Wire Format Name XML1.

2. The new message set will open in the editor. Click **XML1** under Physical Properties in the editor and check **Suppress XML Declarations**.

3. Close the message set.

4. Create a new DTD file that reflects the structure of the output message XML. In this example, the DTD file is called order.dtd. You can see the contents in "Sample DTD files" on page 271.

5. Import order.dtd into the same simple project you used to hold airline.dtd.

6. Create a new message definition file:

   a. Right-click the **OrderXMLMsgSet** in the Navigator pane.

   b. Select **New → Message Definition File** and click **Next**.

   c. Select **XML DTD File**, then in next window select the **order.dtd** file in the project you create for import and click **Next**.

   d. Select **OrderXML** MessageSet and click **Next**.

   e. Select only **CustomerOrder** in the Global Elements list.

   f. Click **Finish**.

You will now have two new XML message sets for the mapping.
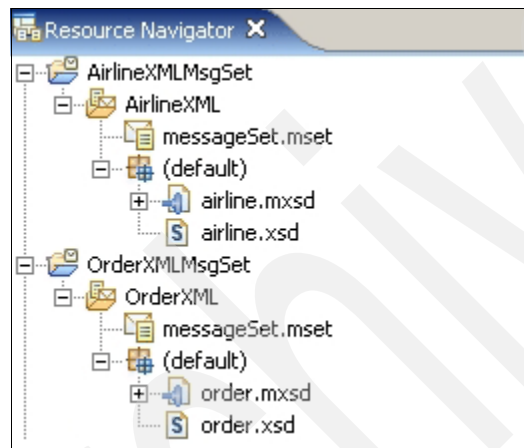
Figure 7-9   Message sets used for XML-to-XML mapping

7.3.2 Create the message flow

To do this:

1. Create a message flow project named XMLMapMsgFlow.

2. Create a message flow named XMLMap in this project.

3. Add an MQInput node, MQOutput node, and Mapping node.

4. Double-click the **MQInput** node.

   On the Basic tab enter `AIRLINE` in the Queue Name field.

   On the Default tab:

   a. Select **MRM** in Message Domain field.
   b. Select **AirlineXML** in the Message Set field.
   c. Select **AirlineRequest** in the Message Type.

d. Select **XML1** in the Message Format field.
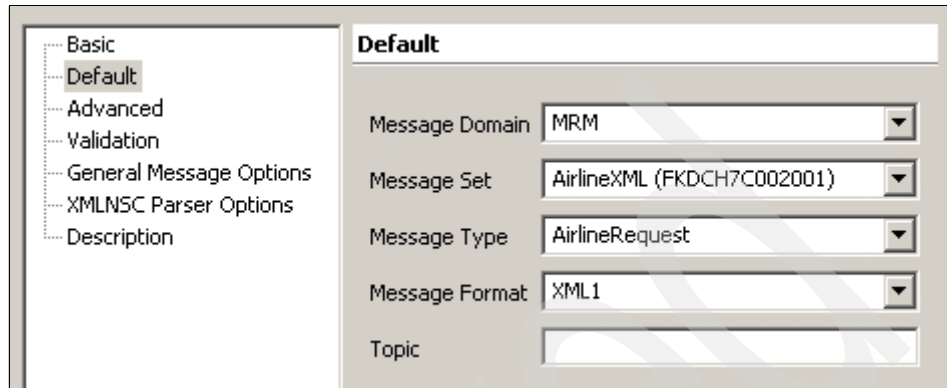
Click **OK**.

Figure 7-10   MQInput node settings

5. Double-click the **MQOutput** node.

    a. Select the **Basic** tab.
    b. Enter `ORDER` in the Queue Name field.
    c. Click **OK**.

6. Connect the nodes as shown in Figure 7-11. Be careful to wire the out terminals to the in terminals.
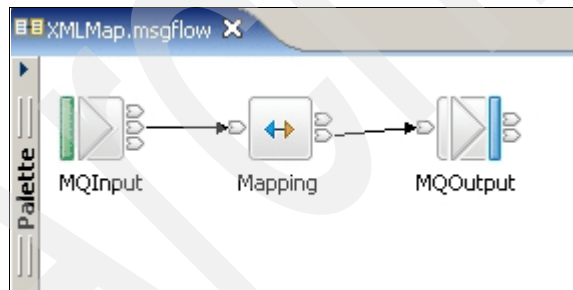
Figure 7-11   Create a message flow

Define the mapping

The Mapping node contains the information required to map a message in one XML format to a message using another format. When you build a Mapping node, you map the source XML fields to the target XML fields.

1. Right-click the **Mapping** node and select **Open Map**.

    a. Click **Next** in the first window.

b. Select **This map is called from message flow node and maps properties and message body** and click **Next**.

c. Select **input message** and click **Next**.

d. The next window lists the existing message sets in the Source and Target window. Select **AirlineRequest** as the source and **CustomerOrder** as the target.
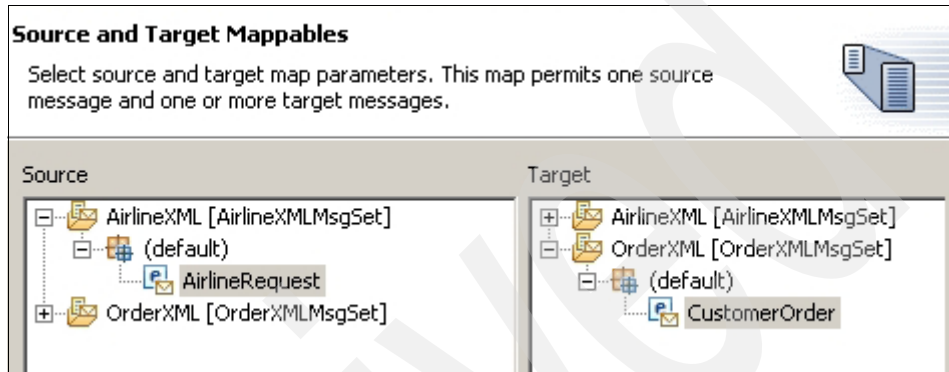
Figure 7-12   Mapping table for source and target

Click **Finish**

2. At the end of the wizard, the mapping will open in an editor. To map a source field to a target field, select the source field and drag and drop it on the target field. As you create a map between two fields, a line will be drawn between the source and target, as shown in Figure 7-13 on page 233.
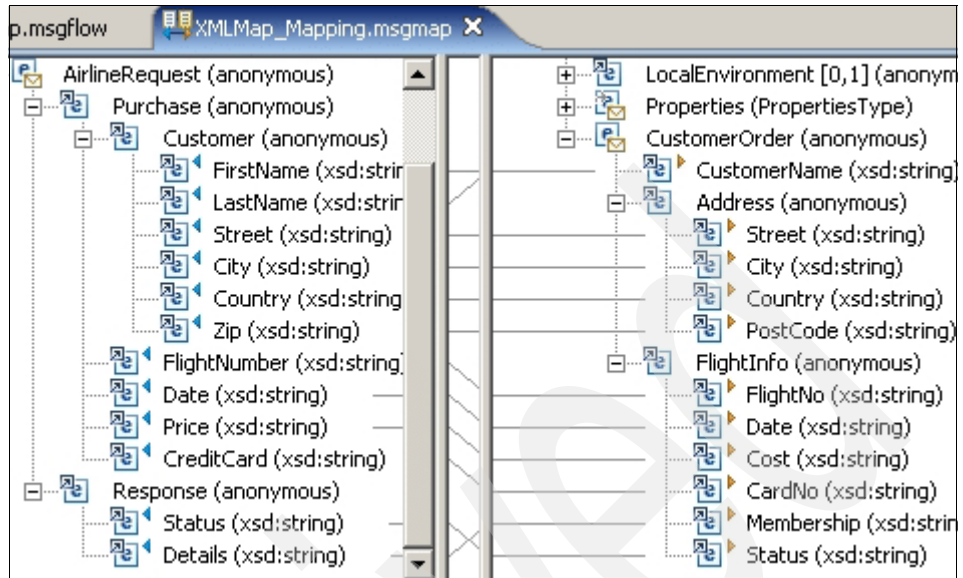
Figure 7-13   Mapping message map

Map the messages as shown in Table 7-3.

Table 7-3   XML mapping

AirlineRequest	CustomerOrder
FirstName	CustomerName[1]
LastName	CustomerName[1]
Street	Street
City	City
Country	Country
Zip	PostCode
FlightNumber	FlightNo
Date	Date
Price	Cost
CredirCard	CardNo
[1]To map both FirstName and LastName to CustomerName, select both names in the source (hold down the Ctrl key while selecting both names), drag them to CustomerName in the target, and drop.	

AirlineRequest	CustomerOrder
Status	Status
Details	Membership
[1]To map both FirstName and LastName to CustomerName, select both names in the source (hold down the Ctrl key while selecting both names), drag them to CustomerName in the target, and drop.	

3. To add a space between the first and last name in the CustomerName field, select **CustomerName** in the Map Script window and change the value to look like the following (adding a space, ' ', between FirstName and LastName):

```
fn:concat($source/AirlineRequest/Purchase/Customer/FirstName, ' ',
$source/AirlineRequest/Purchase/Customer/LastName)
```

4. In the Map Script window, right-click **Properties** under $target and select **Populate**. This will add a list of attributes under Properties. For the following properties, add the following values:

   – 'OrderXML' in MessageSet
   – 'CustomerOrder' in MessageType
   – 'XML1' in MessageFormat

Map Script	Value
— e LocalEnvironment	
⊟ e Properties	
— e MessageSet	'OrderXML'
— e MessageType	'CustomerOrder'
— e MessageFormat	'XML1'
— e Encoding	

Figure 7-14   Output message properties

5. **Save** all changes.

7.3.3 Deploy the message flow to the broker

The next step is to deploy the message flow and message sets to the broker. To deploy, you need to create a new broker archive (bar) file named deploy.bar and add the flow and sets, as shown in Figure 7-15 on page 235.

1. To create the bar file, switch to the Broker Administration perspective.

2. Right-click the **XMLMapMsgFlow** message flow in the navigator and select **New** → **Message Broker Archive**. Select **Local Project** and enter deploy.bar as the name.

3. Click the Add icon and select the message flow and message sets. The BAR file should contain the files shown in Example 7-15.
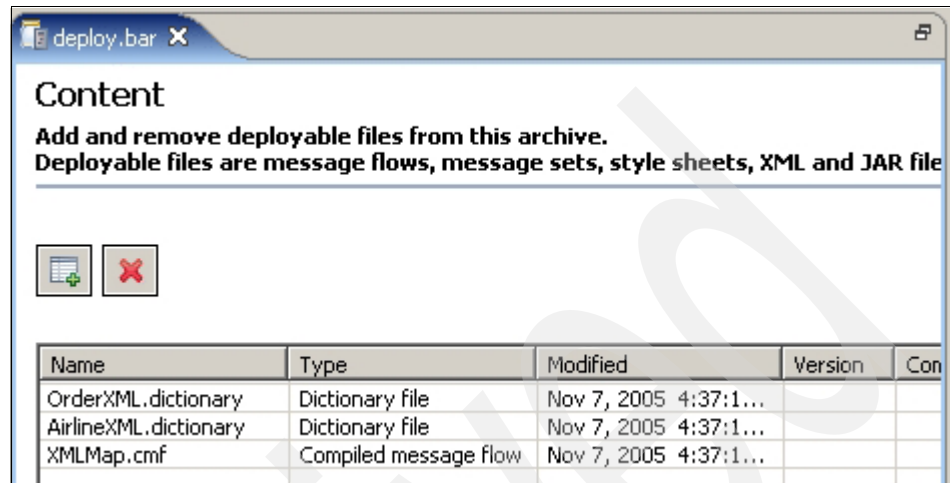
Figure 7-15   Broker archive file

4. Save and close the bar file editor.

5. Connect to the broker.

6. Right-click the bar file and select **Deploy File**.

7. Select the execution group (**Default**) to deploy the file to.

8. If the deploy finishes without error, you will see the following (Figure 7-16).
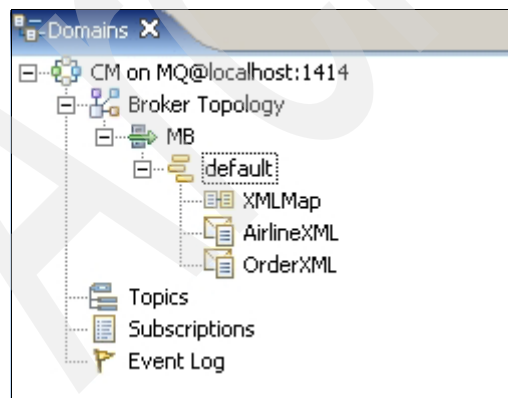
Figure 7-16   Deploy the message flow and message sets to the broker

7.3.4 Create the WebSphere MQ queues

The MQInput and MQOutput nodes specified two queues they use for getting and sending messages. Before testing the message flow, we need to create these queues.
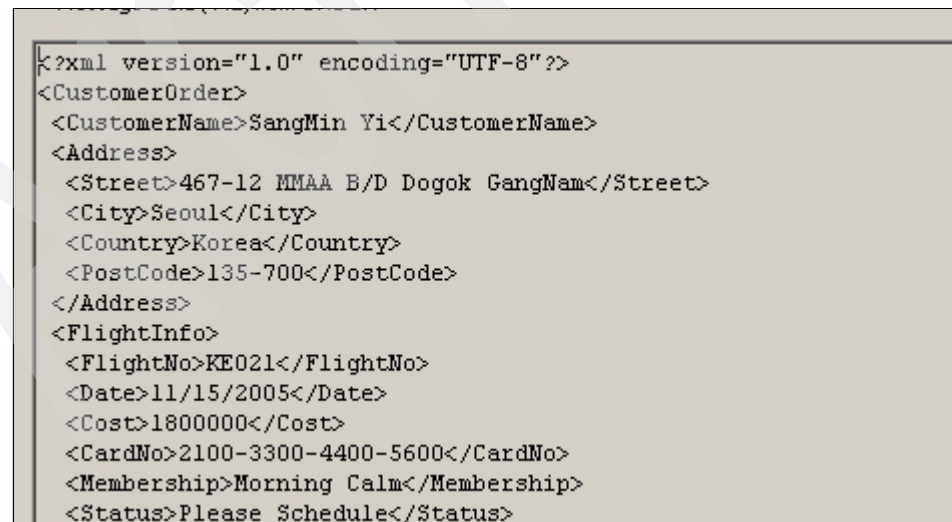
1. The MQInput node gets messages from the AIRLINE queue. Define AIRLINE as a local queue on the broker's queue manager.

2. The MQOutput node puts messages into the ORDER queue. This queue can be defined on any queue manager accessible by the broker's queue manager. For our testing we defined ORDER as a local queue on the broker's queue manager.

7.3.5 Test the message flow

Using RFHUtil, put the Airline1.xml file into the AIRLINE queue. (See "Test the connection" on page 117 for information about using RFHUtil.)

Airline1.xml is in the format defined by the AirlineXML Message Set. If you put any other type of XML file or a non-XML file on the queue, it will be delivered to the dead letter queue.

If the message flow is working as expected, you will be able to see the transformed XML message in the ORDER queue (Figure 7-17). To see the message in RFHUtil, select **ORDER** in the Queue Name field, click **Read Q,** and select the **Data** tab.

```
<?xml version="1.0" encoding="UTF-8"?>
<CustomerOrder>
 <CustomerName>SangMin Yi</CustomerName>
 <Address>
  <Street>467-12 MMAA B/D Dogok GangNam</Street>
  <City>Seoul</City>
  <Country>Korea</Country>
  <PostCode>135-700</PostCode>
 </Address>
 <FlightInfo>
  <FlightNo>KE021</FlightNo>
  <Date>11/15/2005</Date>
  <Cost>1800000</Cost>
  <CardNo>2100-3300-4400-5600</CardNo>
  <Membership>Morning Calm</Membership>
  <Status>Please Schedule</Status>
```

Figure 7-17   Output XML file <CustomerOrder>

7.3.6 Using JMS nodes

Next we use the same message flow, but change it to work with input from a JMS destination instead of input from a queue.

The JMSInput and JMSOutput nodes allow a message flow to receive messages from JMS destinations or to send messages to JMS destinations. These destinations are accessible through a connection to a JMS provider.

Figure 7-18 shows an overview of the configuration in this example.
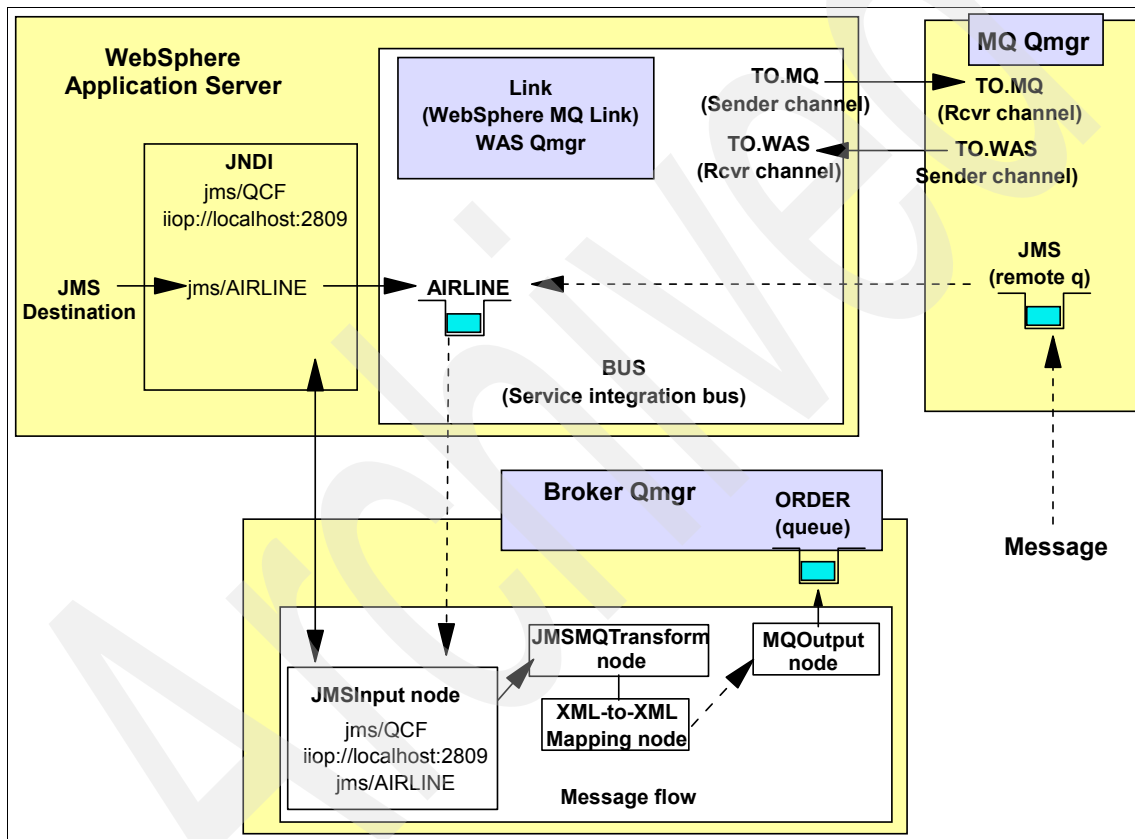
Figure 7-18   XML-to-XML mapping using JMS input and output

In this example, a message is placed on the JMS remote queue in WebSphere MQ, where it is forwarded to the AIRLINE queue on the service integration bus. The JMSInput node in the message flow picks up the message and transforms it into a new XML format. The MQOutput node places the transformed message on the ORDER queue.

Programs use JMS resource objects such as destinations and connection factories instead of pointing directly to queues. These JMS resources are defined and added to the JNDI namespace using the WebSphere Application Server administration tools. The mapping from the JMS resource to the queue is contained in the namespace entries. The JMSInput node in the message flow contains the information it needs to access the JNDI namespace in WebSphere Application Server, where the resource objects are defined.

> **Important:** This scenario requires WebSphere Message Broker to have access to WebSphere Application Server files. In order for this to happen, you need to do the following:
>
> 1. Install WebSphere Application Server or WebSphere Application Server Application Client on the same machine as the WebSphere Message Broker.
>
>   To avoid problems with command-length restrictions on Windows systems when setting the MQSIJVERBOSE variable, install the WebSphere product in a directory with a short name (for example, c:\WebSphere) versus using the default install directory.
>
> 2. Use the following procedure to set a new environment variable called MQSIJVERBOSE:
>
>   a. Assume the following:
>
>   - *<MQSI_ROOT>* is the directory into which the WebSphere Message Broker V6 has been installed.
>
>   - *<CLIENT_ROOT>* is the directory into which the WebSphere Application Server V6 Application Client has been installed
>
>   - *<SERVER_ROOT>* is the directory into which the WebSphere Application Server V6 Server has been installed
>
>   b. Edit the mqsiprofile.cmd file found in *<MQSI_ROOT>*/bin and add one of the following commands depending on whether Application Client or Server is installed:
>
>    ```
>    set
>    MQSIJVERBOSE=-Djava.ext.dirs=<CLIENT_ROOT>\lib;<MQSI_ROOT>\jre\lib;<MQSI_ROOT>\jre\lib\ext;
>    ```
>
>   or
>
>    ```
>    set
>    MQSIJVERBOSE=-Djava.ext.dirs=<SERVER_ROOT>\installedChannels;<SERVER_ROOT>\lib;<MQSI_ROOT>\jre\lib;<MQSI_ROOT>\jre\lib\ext;
>    ```
>
>   c. Restart WebSphere Message Broker.

Configure WebSphere Application Server

In WebSphere Application Server, we need to create the destinations and JNDI definitions for the JMS connection. We assume that you already created a service integration bus named BUS. If not, create a new one using the previous stage described in "Configure the service integration bus" on page 122.

Define the queue as a service integration bus destination

First, we define a queue destination on the service integration bus:

1. Log in to the WebSphere Application Server administrative console.
2. Select **Service Integration** → **Buses**.
3. Click the bus name.
4. Click **Destinations**.
5. Create a new queue named AIRLINE.

Define the JMS resources

Next, define the JMS queue connection factory:

1. Select **Resources** → **JMS Providers** → **Default messaging**.

2. Click **JMS Queue Connection Factory** in Connection Factories.

3. Click **New**.

   a. Enter QCF in the Name field.
   b. Enter jms/QCF in the JNDI Name field.
   c. Select **BUS** in the Bus Name field.
   d. Click **OK**.

Define the JMS queue:

1. Select **Resources** → **JMS Providers** → **Default messaging.**

2. Under Destinations, click **JMS Queue**.

3. Click **Add** to create a new JMS queue.

   a. Enter AIRLINE in the Name field.

   b. Enter jms/AIRLINE in the JNDI Name field.

   c. Select **BUS** in the Bus Name field (created in "Create a bus" on page 122).

   d. Select **AIRLINE** in Queue Name field. This links the JMS queue to the actual destination on the bus.

Figure 7-19   Create a JMS queue

4. Save the changes.

Update the message flow to use JMS nodes

To change our existing message flow to use JMS input do the following:

1. Delete the MQInput node and add a JMSInput node instead.

2. Add a JMSMQTransform node between the JMSInput node and the Mapping node. This node transforms the JMS message into a message that is compatible with the format of messages that are produced by the WebSphere MQ JMS provider.

   The Mapping node can be used without any changes.

3. Make the connections between the nodes as shown in Figure 7-20 on page 241.
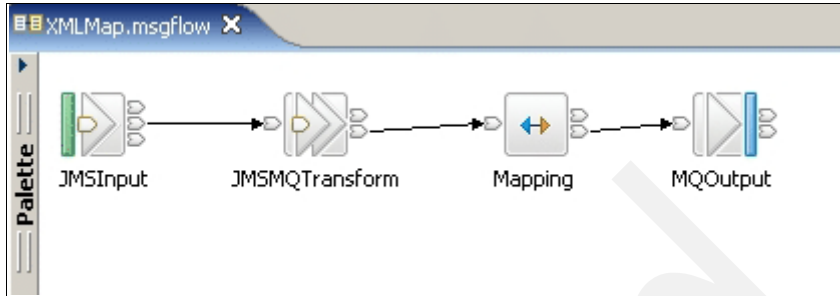
Figure 7-20 Change the XMLMap message flow to use JMS input

4. Double-click the **JMSInput** node.

    a. On the Basic tab change the following:

        • Initial Context Factory:
          `com.ibm.websphere.naming.WsnInitialContextFactory`

          This is used to look up the JMS administered objects in the JNDI. In
          this case, we want to use the WebSphere Application Server JNDI.

        • Location JNDI Bindings: `iiop://localhost:2809`

          The bindings file contains definitions for the JNDI-administered objects
          that are used by the JMSInput node. This URL points to the bootstrap
          address port of the application server. You can check the port number
          in the application server port definitions using the WebSphere
          administrative console. It will be listed as the
          BOOTSTRAP_ADDRESS port.

        • Connection Factory Name: `jms/QCF`

          The connection factory name is used by the JMSInput node to create a
          connection to the JMS provider. The corresponding entry in the JNDI
          will be created later when you add the JMS resource definitions in
          WebSphere Application Server.

Figure 7-21   Basic tab in the JMSInput node

b. Click the **Default** tab and change the following.

- Message Domain: `MRM`
- MessageSet: `AirlineXML`
- MessageType: `AirlineRequest`
- Message Format: `XML1`

Figure 7-22   Default tab in the JMSInput node

c. On the Point to Point tab enter `jms/AIRLINE` in the Source Queue field (thus making the link to the JMS queue resource defined in the application server JNDI).

d. Click **OK**.
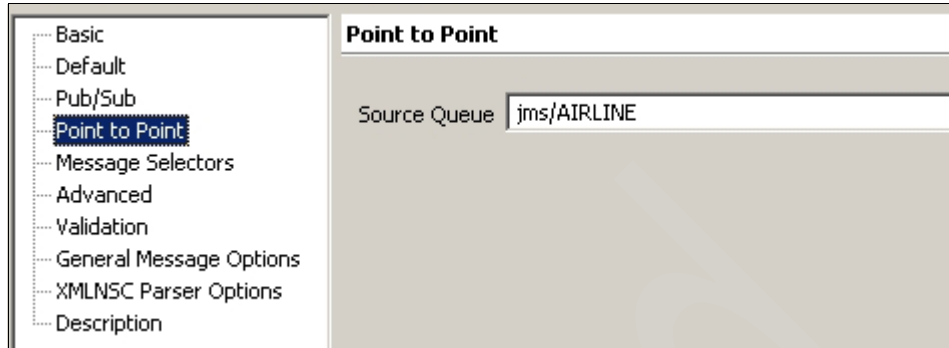
Figure 7-23   Point to Point tab in the JMSInput node

5. Save the changes.

6. To make the JMS provider client available to the JMS nodes, copy the necessary JAR files from the WebSphere Application Server runtime to the shared-classes directory.

   a. Copy all files in the *<WAS_HOME>*\lib directory to C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes.

   b. Copy all files in the *<WAS_HOME>*\ installedChannels directory to C:\Documents and Settings\All Users\Application Data\IBM\MQSI\shared-classes.

7. Update the BAR file and deploy it to the broker.

Test the message flow

Instead of creating a JMS client for testing, we use RFHUtil and a remote queue definition to place the message on the queue destination in the application server.

This assumes that you have done the steps in 5.2, "Connect WebSphere ESB to WebSphere MQ" on page 118, and the connection between the two is running.

1. Using the WebSphere MQ Explorer, create a remote queue definition named JMS, targeting the AIRLINE queue of the WAS queue manager (see "Create a remote queue definition" on page 112).

2. Start RFHUtil.

3. Select **MQ** in the Queue Manager Name field and **JMS** in the Queue Name field.

4. Click **Read File** to read in the Airline1.XML file.

Figure 7-24   Test the flow with RFHUtil

5. Select the **MQMD** tab, and type `MQSTR` in the MQ Message Format field.

Figure 7-25   Select the MQ message format

6. Return to the Main tab and click **Write Q**.

    The message will be put on the AIRLINE queue in WAS, and will be transferred to the ORDER queue via the message flow in WebSphere Message Broker. The message flow will transform the message from the original XML format to the new.

7. In RFHUtil, select the **ORDER** queue and click the **Read Q** button. Click the **Data** tab to see the transformed message.

Figure 7-26   Changed XML data in the target queue

7.4 XML-to-XML transformation using XSLT

This scenario illustrates how to use an XMLTransformation node instead of a Mapping node to transform a message.

The configuration for this scenario is shown in Figure 7-27 on page 246.
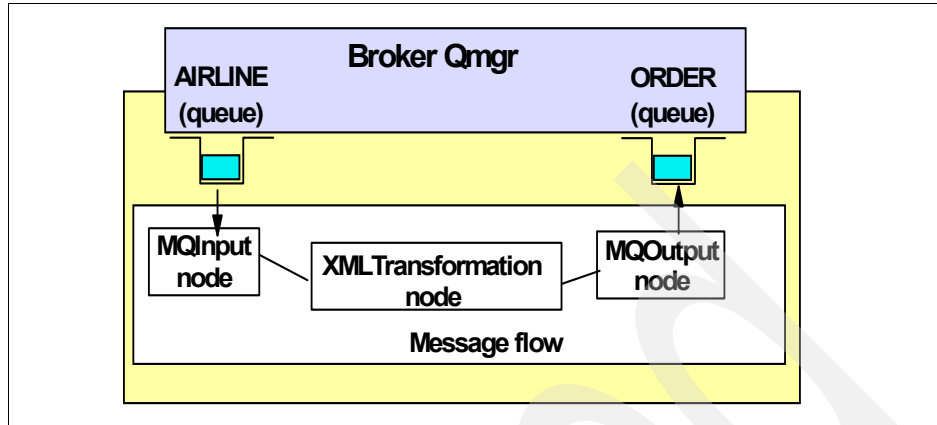
Figure 7-27   XML-to-XML mapping using an XMLTransformation node

The steps required to do this mapping are:

1. Create the message sets.
2. Create the mapping.
3. Create the message flow.
4. Create the WebSphere MQ queues.
5. Deploy and test the message flow.

7.4.1 Create the message sets

This scenario uses the same message sets created in "Create the message sets containing the XML DTD files" on page 228. The two message sets contain the DTD files describing the XML format for the input and output messages.

7.4.2 Create the mapping

The first step is to create a mapping between the source and target XML formats. To do this, create a simple project and import the target and XML files into it. Then map the fields in the source XML to the target XML.

Note that this process also uses the Mapping editor shown in "Build the mediation module" on page 199. In that scenario, WebSphere Integration Developer was used as the development tool. In this scenario, the Message Brokers Toolkit is used. Both tools use the same technology and have the same XML tools.

Create the XML files

To perform XML mapping, you need the XSD file for the XML source or the XML file itself. Using a text editor, create the XML files for the source and target and store them in a temporary directory. In this scenario, we use the Order1.xml file for the target and Airline1.xml for the source. You can see these files in "Sample XML files" on page 270.

Create the mapping

To create the mapping:

1. Create a simple project called XSLT.

2. In the navigator pane, right-click the **XSLT** project and select **New** → **Other**. Select **XML** → **XML To XML Mapping**. If you do not see the XML options, be sure to check the box that says `Show All Wizards`.

3. Click **Next**.

4. Select the **XSLT** project and click the **Next** button.

   a. Click the **Import Files...** button and import the two XML files you created to the XSLT folder.

   b. Select **Airline1.xml** in the Workbench Files pane and send it to the Selected Files pane. Click **Next**.
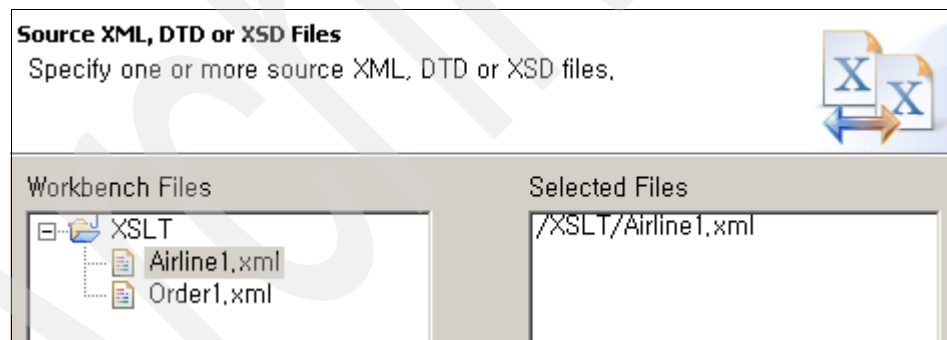
Figure 7-28   Select source file

   c. Select **Order1.xml** as a target and click **Next**.

   d. Confirm that CustomerOrder is the target root element and AirlineRequest is the source root element. Then click **Finish**.

5. A mapping editor will be opened. Map the source elements and the target elements as shown in Table 7-4 on page 248. Mapping is done by dragging an element on the left to an element on the right. To select two elements at once (for example, FirstName and LastName), press the Ctrl key while you are selecting the elements.

Table 7-4   XML mapping

AirlineRequest	CustomerOrder
FirstName	CustomerName
LastName	CustomerName
Street	Street
City	City
Country	Country
Zip	PostCode
FlightNumber	FlightNo
Date	Date
Price	Cost
CredirCard	CardNo
Status	Status
Details	Membership

– The mapping will look like that shown in Figure 7-29 on page 249.
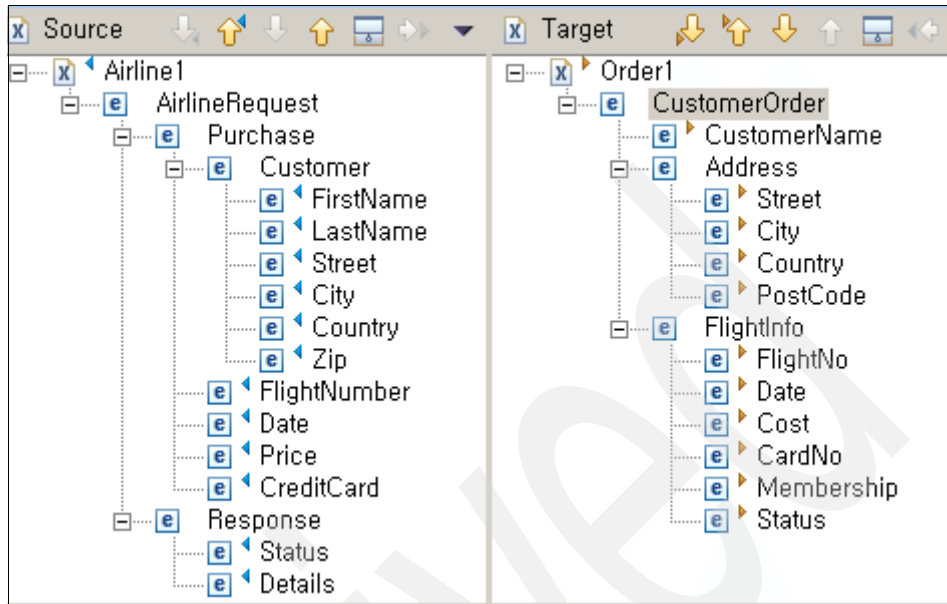
Figure 7-29   XSLT mapping

–   To add a blank between FirstName and LastName:

i.   Select **CustomerName** in the Overview view. Right-click and select **Define XSLT Function**.

ii. Select **String**, and click **Next**.

iii. Select **concat** as the function (default). Then click **Add**.

iv. Enter quotes (' ') as the parameter value and click **OK**.

v.   Select the quotes in the Input Parameters window and move them between FirstName and Last Name using the Up and Down buttons, as shown in Figure 7-30 on page 250.
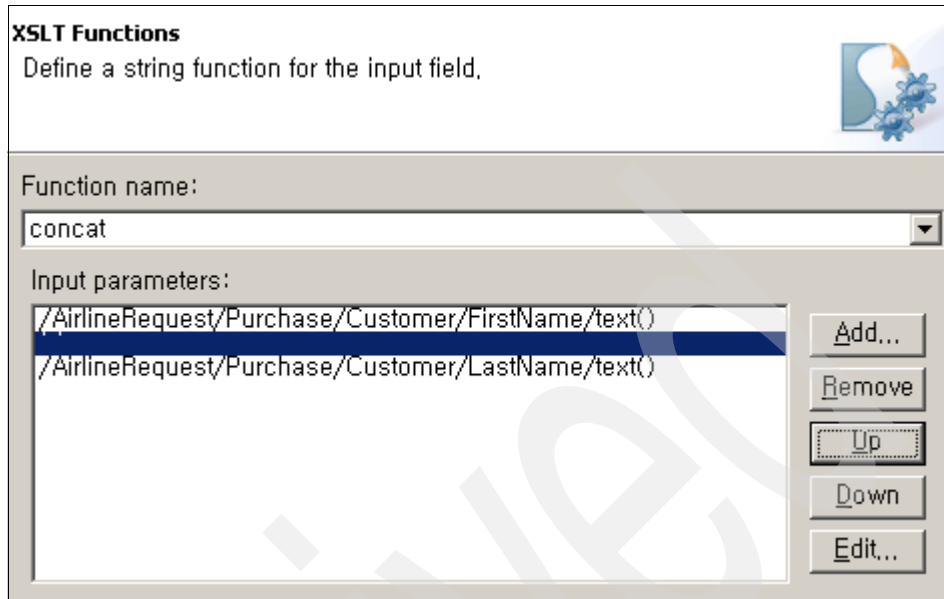
Figure 7-30   Define an XSLT function

vi. Click **Finish**.

6. Save the map.

7. Right-click the **xmlmap.xmx** file in the Navigator view and select **Generate XSLT**. When the window appears, click **Finish**. The new xmlmap.xsl file will be created.

At the completion of this step, the XSLT project in the Navigator view of the Broker Application Development perspective should look like Figure 7-31 on page 251.
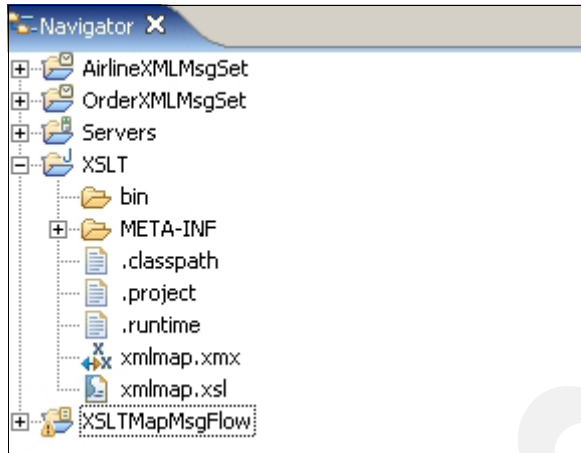
Figure 7-31   XSLT mapping

Test the mapping

Now let us test XML mapping with this file:

1. Right-click **xmlmap.xsl** and select **Run** → **XSL Transformation**.

2. Select the **Airline1.XML** file in the Source XML File field and specify the location for the output in the Output File field.

3. Click **Run**.

4. Open the output file and check to see if the XML data is transformed as you expected.

7.4.3 Create the message flow

To create the message flow:

1. Start the Message Brokers Toolkit and switch to the Broker Application Development perspective.

2. Create a message flow project named `XSLTMapMsgFlow`.

3. Create a message flow named `XSLTMap` in this project.

4. In the message flow, add an MQInput node, an MQOutput node, and an XMLTransformation node.

5. Double-click the **MQInput** node.

   On the Basic tab, enter `AIRLINE` in the Queue Name field.

   On the Default tab:

   a. Select **MRM** in the Message Domain field.

b. Select **AirlineXML** in Message Set field.

c. Select **AirlineRequest** in the Message Type.

d. Select **XML1** in the Message Format field.

Click **OK**.

6. Double-click the **MQOutput** node and enter `Order` in the Queue Name field.

Note that the MQInput and MQOutput nodes used in this flow are exactly the same as those used in the Mapping node example earlier. The use of the XMLTransformation node does not change the input and output requirements of the flow.

7. Wire the nodes as shown in Figure 7-32.
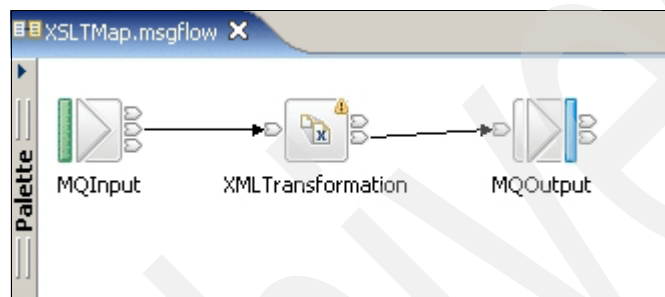
Figure 7-32   Create a message flow using XSLT mapping

8. Right-click **XSLTMapMsgFlow** and select **Properties**.

9. In the Properties window, select the **Project References** tab and check the XSLT project with the XSLT mapping (see Figure 7-31 on page 251), and click **OK**. This allows you to use the files of the XSLT project in the message flow project.
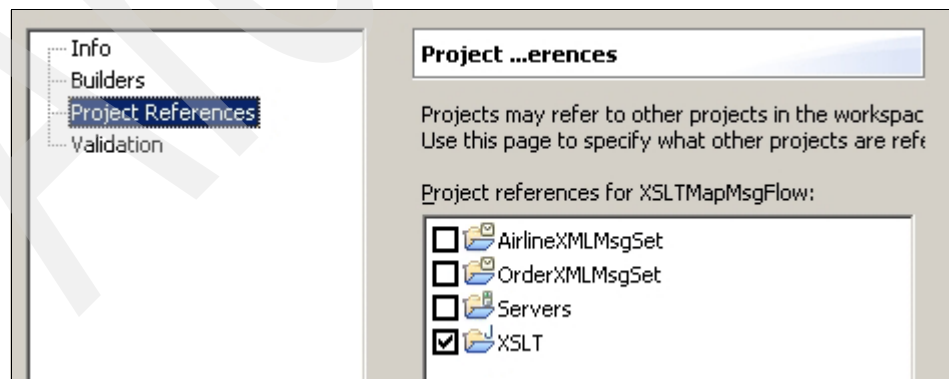
Figure 7-33   Select project references

10. In XSLTMap message flow, double-click the XMLTransformation node. Select the **xmlmap.xsl** file in the Stylesheet Name field and click **OK**.

Figure 7-34   Select the stylesheet name

7.4.4 Create the WebSphere MQ queues

This example uses the AIRLINE and ORDER queues defined previously in "Create the WebSphere MQ queues" on page 236.

7.4.5 Deploy and test the message flow

To do this:

1. Create a bar file for deployment. Select the **xmlmap.xsl** file in addition to the message flow and two message sets, as shown in Figure 7-35 on page 254.
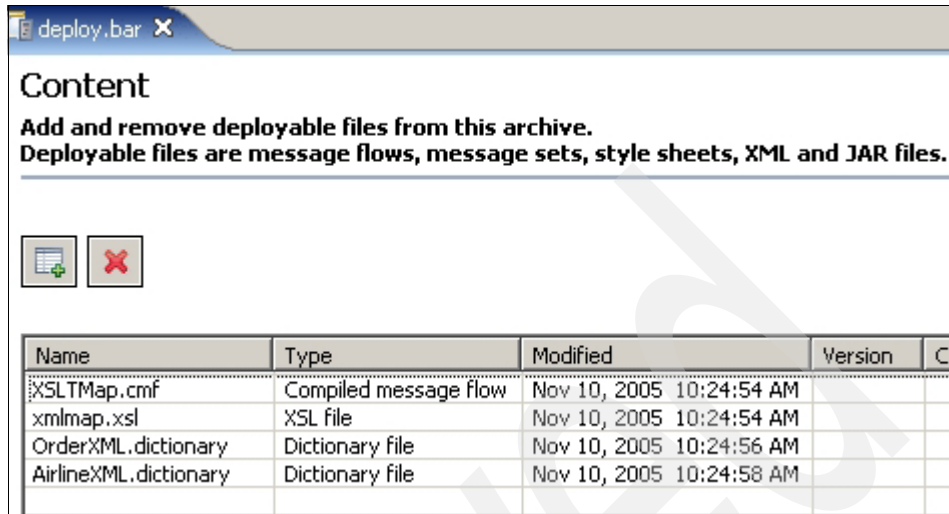
Figure 7-35   Create the BAR file for deployment

2. Deploy the bar file to the broker. You can see in Figure 7-36 that four files are deployed to the broker.
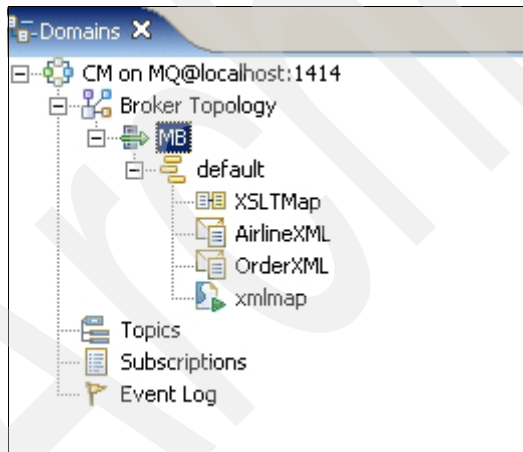
Figure 7-36   Deployed files in the broker

To test, use RFHUTIL (see "Test the connection" on page 117) to put the Airline1.XML file onto the AIRLINE queue. The message should be transformed and sent to the ORDER queue. If the mapping message flow worked correctly, you can see the transformed message by reading the message from the ORDER queue using RFHUTIL.

7.5 XML-to-COBOL mapping

This section shows how to use WebSphere Message Broker to transform an XML message to COBOL.
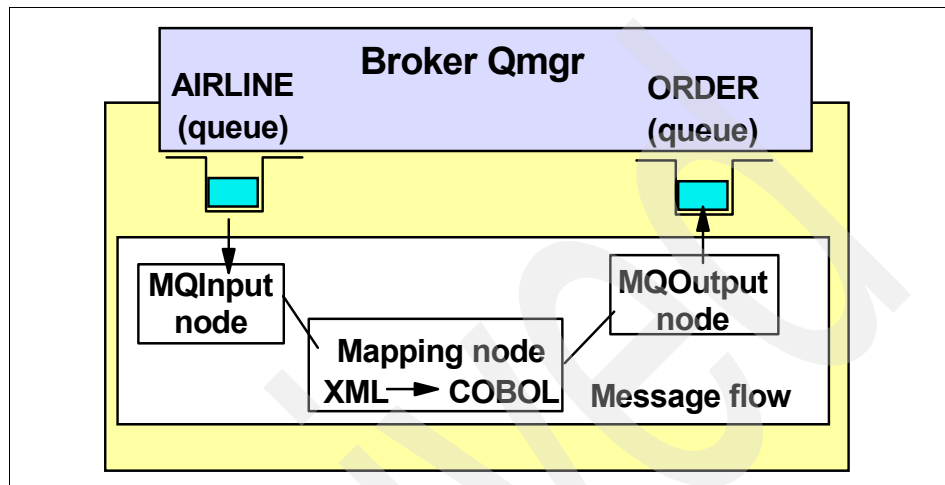
Figure 7-37   XML-to-COBOL mapping using a Mapping node

This scenario illustrates how to create a message flow that transforms an XML message into a COBOL message. In this scenario, the travel bureau sends messages in the XML format shown in Example A-1 on page 270. The airline company expects COBOL messages in the format shown in Figure 7-38.

Figure 7-38   Airline company's COBOL data <msg_AIRLINEREQUEST>

To do this transformation we perform the following steps:

1. Create the message sets.
2. Create the message flow.
3. Define the mapping.
4. Create the WebSphere MQ queues.
5. Test the message flow.

7.5.1 Create the message sets

Two message sets are used to hold the message definitions for this flow. The first message set contains the DTD file for the XML format seen in Example A-1 on page 270. This message set was created earlier in "Create the airline message set" on page 229.

To create the new message set for the COBOL messages, use the WebSphere Message Brokers Toolkit. From the Broker Application Development perspective:

1. Create a message set project named OrderCOBOLMsgSet.

   a. Create a message set named OrderCOBOL in this project.
   b. Select **CWF1** as the Custom Wire Format Name.

2. Create a COBOL copy book file using the statements in Figure 7-1. Name the file order.cpy.

Example 7-1   COBOL copy book file order.cpy

```
01 AIRLINEREQUEST.
        10 CUSTOMER.
            15 GIVENNAME              PIC X(15).
            15 SIRNAME                PIC X(15).
            15 STREET                 PIC X(30).
            15 CITY                   PIC X(15).
            15 COUNTRY                PIC X(15).
            15 ZIPCODE                PIC X(10).
        10 FLIGHTNO                   PIC X(6).
        10 TRANDATE                   PIC X(10).
        10 COST                       PIC X(8).
        10 CCNO                       PIC X(20).
        10 RESPONSE.
            15 DETAILS                PIC X(30).
            15 RESERVATION            PIC X(30).
```

3. Import this file into the same simple project that you used to hold the DTD files in the previous scenarios.

4. Highlight the **OrderCOBOLMsgSet** in the Navigator pane, right-click, and select **New** → **Message Definition File**.

a. Select **COBOL file** and click **Next**.

b. Select the **order.cpy** file you imported.

c. Select **OrderCOBOL** as the message set.

d. Select **AIRLINEREQUEST** in the Source Structures pane and send it (using the **>** button) to Imported Structures.

e. Select the box to the left of AIRLINEREQUEST in the Imported Structures pane.

f. Click **Finish**.
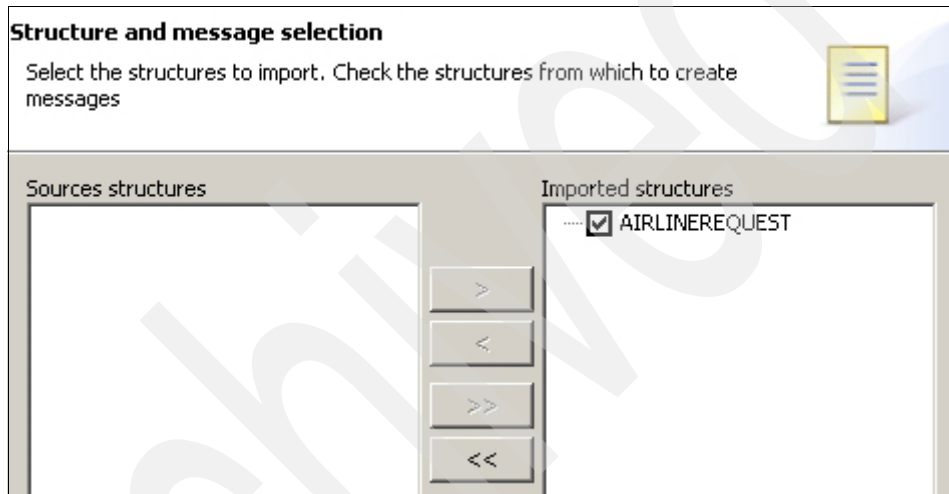
Figure 7-39   Create a message definition file using a COBOL copy book file

7.5.2 Create the message flow

The next step is to build the message flow that will take the message from a WebSphere MQ queue, map it to the new format, and put it on the output queue.

1. Create a message flow project named `COBOLMapMsgFlow`.

2. Create a message flow named `COBOLMap` in this project.

3. Add an MQInput node, MQOutput node, and Mapping node. Note that you could use JMS input or output instead. An example of this is shown in "Update the message flow to use JMS nodes" on page 240.

4. Double-click the **MQInput** node.

   a. On the Basic tab, enter `XML.IN` in the Queue Name field.
   b. On the Default tab, select:
      • **MRM** in the Message Domain field

- **AirlineXML** in the MessageSet field
- **AirlineRequest** in the Message Type field
- **XML1** in the Message Format field

   c. Click **OK**.

5. Double-click the **MQOutput** node and enter `COBOL.OUT` in the Queue Name field. Click **OK**.

6. Wire the nodes as shown in Figure 7-40.
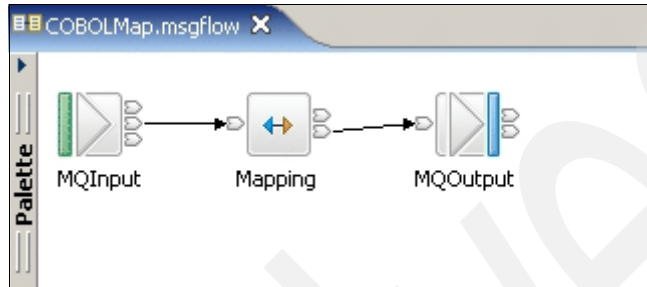
Figure 7-40   Create a new message flow for XML-to-COBOL transformation

Define the mapping

Next we need to define the mapping from the XML format to COBOL:

1. Right-click the **Mapping** node and select **Open Map**.

   a. Click **Next** in the first window.

   b. Select **this map is called from message flow node and maps properties and message body** and click **Next**.

   c. Select **input message** and click **Next**.

   d. Select **AirlineRequest** in the Source pane and **msg_AIRLINEREQUEST** in the Target pane, and click **Finish**.
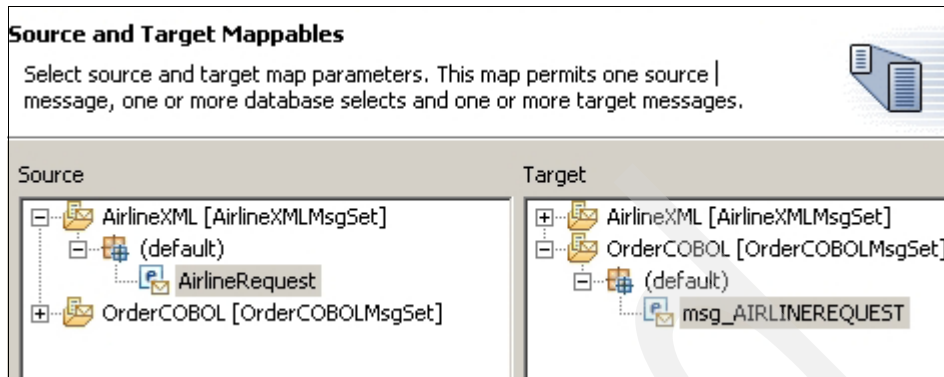
Figure 7-41 Select message sets for mapping

2. Map the messages as shown in Figure 7-5.

Table 7-5 XML-to-COBOL mapping

XML (AirlineRequest)	COBOL (msg_AIRLINEREQUEST)
FirstName	GIVENNAME
LastName	SIRNAME
Street	STREET
City	CITY
Country	COUNTRY
Zip	ZIPCODE
FlightNumber	FLIGHTNO
Date	TRANDATE
Price	COST
CreditCard	CCNO
Status	RESERVATION
Details	DETAILS

You can see the mapping table in Figure 7-42 on page 260.
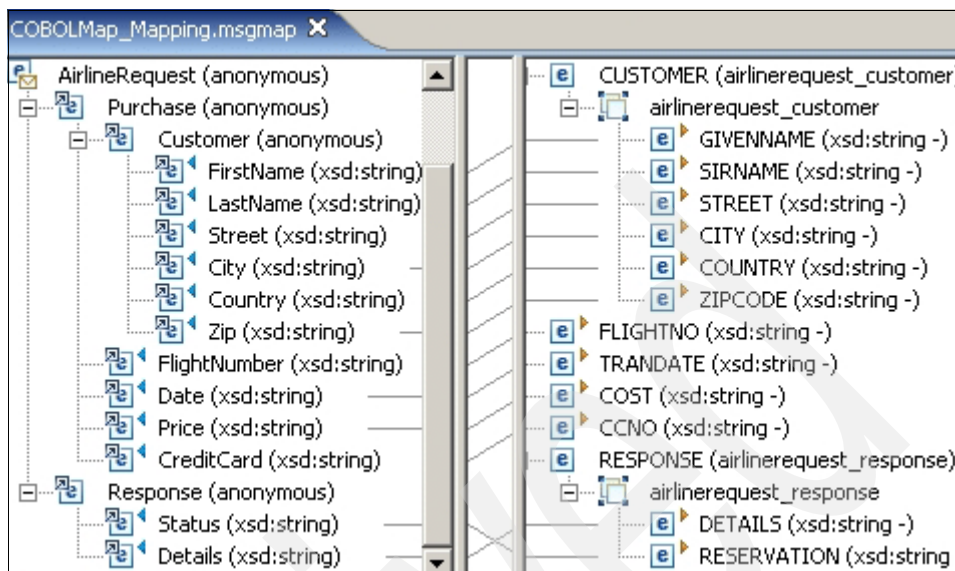
Figure 7-42   XML to COBOL mapping

3. In the Map Script view, right-click **Properties** and select **Populate**. Add the following attributes:

   – `'OrderCOBOL'` in the MessageSet field
   – `'msg_AIRLINEREQUEST'` in the MessageType field
   – `'CWF1'` in the MessageFormat field

Figure 7-43   Output Message Properties in Map Script

4. Save all changes.

5. Create a bar file containing the message flow and message sets.

6. Deploy the bar file to the broker. If the deploy finishes without any error, you will see the following in the execution group of your running broker (Figure 7-44 on page 261).
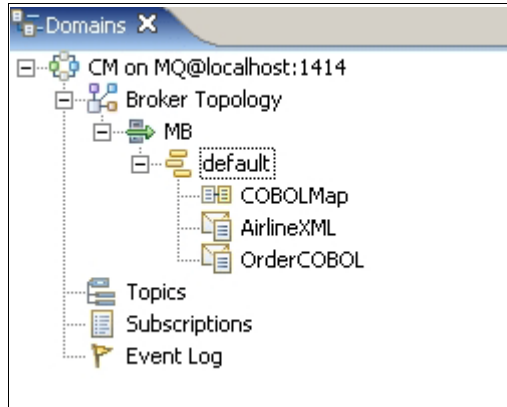
Figure 7-44   Deploy the message flow and message sets to the broker

7.5.3 Create the WebSphere MQ queues

The input and output queues for the message flow need to be created in WebSphere MQ. Using WebSphere MQ Explorer, create two local queues called XML.IN and COBOL.OUT. XML.In is defined in the broker's queue manager. For testing, COBOL.OUT was also created in the broker's queue manager, though in reality it could be defined under any queue manager accessible to the broker's queue manager.

7.5.4 Test the message flow

Using RFHUtil, put the Airline1.xml file into the XML.IN queue.

If the message flow is working properly, you will be able to see the transformed message in the COBOL.OUT queue.

To see the message in its proper format:

1. Select **COBOL.OUT** in the Queue Name field and click **Read Q**.

2. Go to the **Data** tab, select **COBOL** in the Data Format field, and select the **order.cpy** file.

You should be able to see the transformed COBOL message.

```
lain    Data    MQMD   RFH    PubSub  pscr    jms     usr     other

Message Data (204) from COBOL.OUT

Level   Ofs   Len Type   Occ Variable Name   Value
01       0    204            AIRLINEREQUEST
10       0    100            CUSTOMER
15       0     15 CHAR       GIVENNAME        SangMin
15      15     15 CHAR       SIRNAME          Yi
15      30     30 CHAR       STREET           467-12 MMAA B/D Dogok GangN
15      60     15 CHAR       CITY             Seoul
15      75     15 CHAR       COUNTRY          Korea
15      90     10 CHAR       ZIPCODE          135-700
10     100      6 CHAR       FLIGHTNO         KE021
10     106     10 CHAR       TRANDATE         11/15/2005
10     116      8 CHAR       COST             1800000
10     124     20 CHAR       CCNO             2100-3300-4400-5600
10     144     60            RESPONSE
15     144     30 CHAR       DETAILS          Morning Calm
15     174     30 CHAR       RESERVATION      Please Schedule
Copy book size 204, Data area size 204
```

Figure 7-45   Output COBOL file <AIRLINEREQUEST>

7.6 Routing messages

The previous scenarios all focused on transformation of messages from one
format to another. This scenario illustrates how a message flow can route a
message depending on the message content.

This scenario assumes that the message flow developed in "XML-to-XML
transformation using XSLT" on page 245 has been created and successfully
deployed. The message flow that we build here takes the transformed message
from the ORDER queue. One of the new elements in the message is the
Membership. The new message flow looks at the value of Membership. If the
value is 'Skypass', the message will be sent to the GENERAL queue. If the value
is 'Morning Calm', the message will be sent to the UPDATE and PREMIUM
queues.
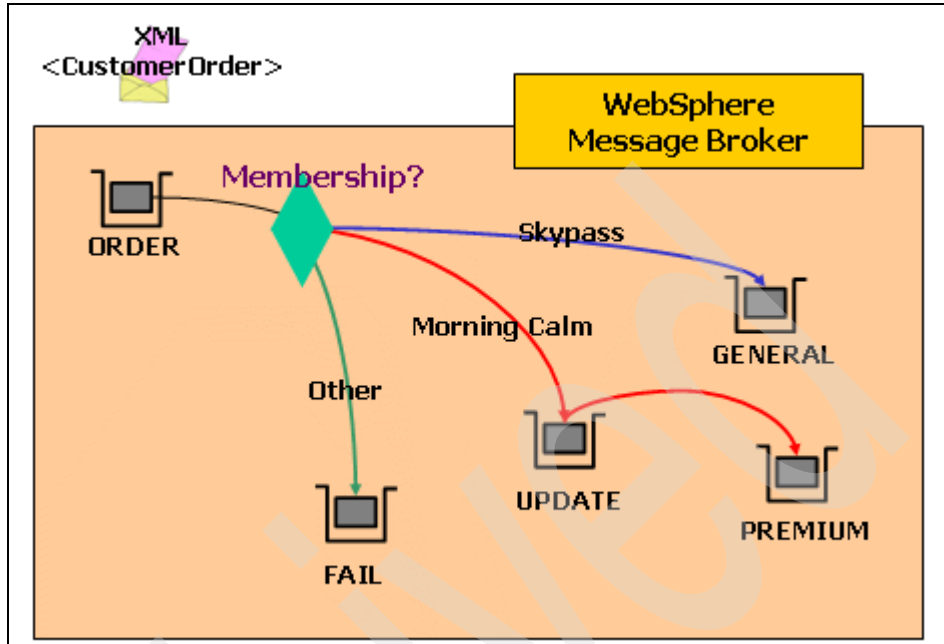
Figure 7-46   A simple routing scenario

To complete this scenario you will:

1. Create the message flow.
2. Define the filters.
3. Create the WebSphere MQ queues.
4. Deploy and test the message flow.

7.6.1 Create the message flow

To create the new message flow do the following:

1. Create a new message flow project named `RoutingMsgFlow`.

2. Create a new message flow named `Routing`.

3. Add a new MQInput node and open it:

   a. On the Basic tab, enter `ORDER` in Queue Name field.
   b. On the Default tab, select **XML** in the Message Domain field.
   c. Click **OK** to close the node.

   Rename the MQInput node to `ORDER`. You can rename a node by right-clicking the node and selecting **Rename**.

4. Add a Filter node and open it. Change Filter Expression to `Filter1`.
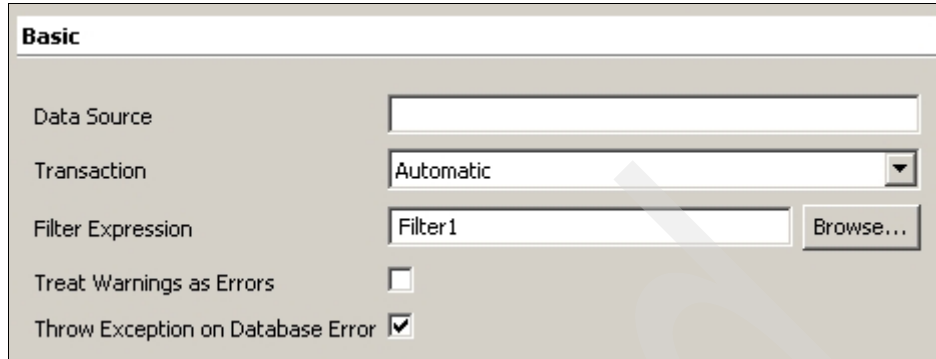
Figure 7-47   Change the parameter of Filter Expression

Close the properties and rename the Filter node to `Skypass`.

5. Add a second Filter node. Open it and change the Filter Expression to `Filter2`. Close the node and rename it `Morning Calm`.

6. Add four MQOutput nodes with the following queue names:

    – GENERAL
    – UPDATE
    – FAIL
    – PREMIUM

    Rename each node to reflect the name of the queue.

7. Connect the nodes as shown in Figure 7-48. You can rename nodes if necessary. Note that for ease of use, we have renamed the nodes to reflect their functions.
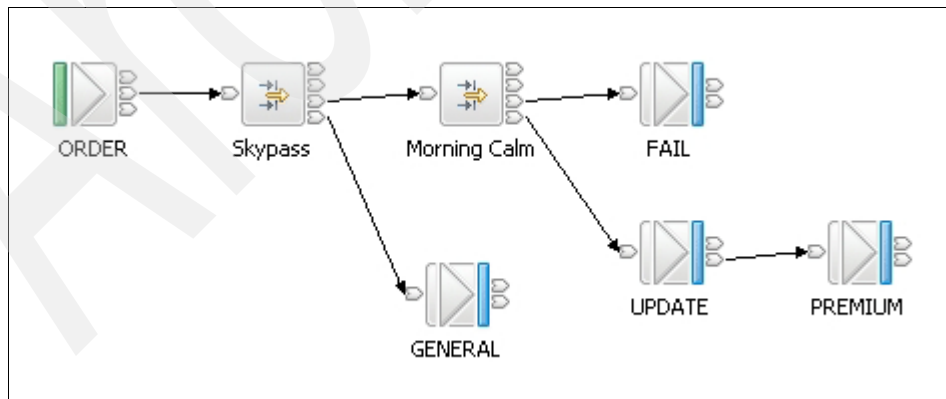
Figure 7-48   Create a new message flow for routing

7.6.2 Define the filters

To do this:

1. Right-click the first Filter node (**Skypass**) and select **Open ESQL**. Change the ESQL code as shown in Figure 7-2.

Example 7-2   ESQL code for Skypass filter

```
CREATE FILTER MODULE Filter1
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      If Root.XML.CustomerOrder.FlightInfo.Membership = 'Skypass'
         Then Return TRUE;
      Else Return FALSE;
      End If;
   END;
END MODULE;
```

2. Open the ESQL for the second Filter node (Morning Calm). Note that the ESQL for both Filter nodes is kept in one file. Change the ESQL code as shown in Figure 7-3.

Example 7-3   ESQL code for Morning Calm filter

```
CREATE FILTER MODULE Filter2
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      If Root.XML.CustomerOrder.FlightInfo.Membership = 'Morning Calm'
         Then Return TRUE;
      Else Return FALSE;
      End If;
   END;
END MODULE;
```

The resulting definitions should look like Figure 7-49 on page 266.

Figure 7-49   Created ESQL file in message flow

3. Close and save the ESQL files and the message flow.

7.6.3 Create the WebSphere MQ queues

Create the following local queues using WebSphere MQ Explorer:

► GENERAL
► UPDATE
► FAIL
► PREMIUM

Since this scenario builds on the XSLT transformation scenario, we assume that you have already created the AIRLINE and ORDER queues.

7.6.4 Deploy and test the message flow

To test the message flow:

1. Add the new message flow to the bar file used in "XML-to-XML transformation using XSLT" on page 245.

2. Deploy the bar file to the broker. You should see five files in the execution group, including the four files you deployed in the XSLT mapping scenario.
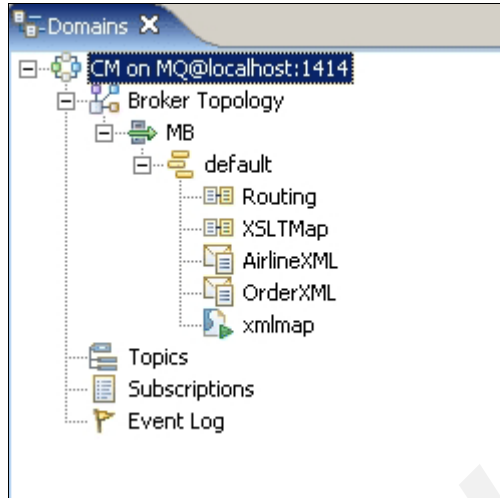
Figure 7-50   Deployed files in the broker

3. Using RFHUtil, put the Airline1.xml file in to the AIRLINE queue.

   If the message flow works as expected, you will be able to see the transformed XML message in the UPDATE queue and PREMIUM queues.

Figure 7-51   Delivered message in the queue

4. Copy the Airline1.XML file and paste it into the same directory. Rename the new copy to `Airline2.XML` and change `Morning Calm` to `Skypass` in the Detail element.

5. Using RFHUtil, put this Airline2.xml file into the AIRLINE queue.

Figure 7-52   Input changed XML data to AIRLINE queue

If the message flow works as expected, you will be able to see the transformed XML message in the GENERAL queue.

If you use a value other than Skypass or Morning Calm, the message will be sent to the FAIL queue.

A

Sample files

This appendix contains the files used in the various samples throughout this book.

269

Sample XML files

Two XML files are used throughout this book, Airline1.xml and Order1.xml.

Airline1.xml file

Example A-1 shows the text used in the Airline1.xml file.

Example: A-1   Airline1.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<AirlineRequest>
    <Purchase>
        <Customer>
            <FirstName>John</FirstName>
            <LastName>Doe</LastName>
            <Street>123 Main</Street>
            <City>Mycity</City>
            <Country>US</Country>
            <Zip>12345</Zip>
        </Customer>
        <FlightNumber>KE021</FlightNumber>
        <Date>11/15/2005</Date>
        <Price>1800000</Price>
        <CreditCard>2100-3300-4400-5600</CreditCard>
    </Purchase>
    <Response>
        <Status>Please Schedule</Status>
        <Details>Morning Calm</Details>
    </Response>
</AirlineRequest>
```

Order1.xml file

Example A-2 shows the text used in the Order1.xml file.

Example: A-2   Order1.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
  - <CustomerOrder>
      <CustomerName>SangMin Yi</CustomerName>
    - <Address>
       <Street>467-12 MMAA B/D Dogok GangNam</Street>
       <City>Seoul</City>
       <Country>Korea</Country>
       <PostCode>135-700</PostCode>
    </Address>
  - <FlightInfo>
```

```
            <FlightNo>KE021</FlightNo>
            <Date>11/15/2005</Date>
            <Cost>1800000</Cost>
            <CardNo>2100-3300-4400-5600</CardNo>
            <Membership>Morning Calm</Membership>
            <Status>Please Schedule</Status>
        </FlightInfo>

</CustomerOrder>
```

Sample DTD files

The following DTD files are used in the WebSphere Message Broker samples.

airline.dtd file

Example A-3 shows the text used in the airline.dtd file. Note that this is the DTD file for the XML file in Example A-1 on page 270.

Example: A-3   airline.dtd

```
<?xml version='1.0' encoding="UTF-8"?>

  <!ELEMENT FirstName    (#PCDATA)>
  <!ELEMENT LastName     (#PCDATA)>
  <!ELEMENT Street       (#PCDATA)>
  <!ELEMENT City         (#PCDATA)>
  <!ELEMENT Country      (#PCDATA)>
  <!ELEMENT Zip          (#PCDATA)>
  <!ELEMENT FlightNumber (#PCDATA)>
  <!ELEMENT Date         (#PCDATA)>
  <!ELEMENT Price        (#PCDATA)>
  <!ELEMENT CreditCard   (#PCDATA)>
  <!ELEMENT Status       (#PCDATA)>
  <!ELEMENT Details      (#PCDATA)>

<!ELEMENT Customer
   (FirstName,
    LastName,
    Street,
    City,
    Country,
    Zip)
 >

 <!ELEMENT Purchase
   (Customer,
```

```
        FlightNumber,
        Date,
        Price,
        CreditCard)
    >

    <!ELEMENT Response
      (Status,
        Details)
    >

    <!ELEMENT AirlineRequest
      (Purchase,
        Response)
    >
```

order.dtd file

Example A-4 shows the text used in the order.dtd file. Note that this is the DTD file for the XML file in Example A-2 on page 270.

Example: A-4   order.dtd

```
<?xml version='1.0' encoding="UTF-8"?>

  <!ELEMENT CustomerName (#PCDATA)>
  <!ELEMENT Street        (#PCDATA)>
  <!ELEMENT City          (#PCDATA)>
  <!ELEMENT Country       (#PCDATA)>
  <!ELEMENT PostCode      (#PCDATA)>
  <!ELEMENT FlightNo      (#PCDATA)>
  <!ELEMENT Date          (#PCDATA)>
  <!ELEMENT Cost          (#PCDATA)>
  <!ELEMENT CardNo        (#PCDATA)>
  <!ELEMENT Membership    (#PCDATA)>
  <!ELEMENT Status        (#PCDATA)>

  <!ELEMENT Address
    (Street,
      City,
      Country,
      PostCode)
  >

  <!ELEMENT FlightInfo
    (FlightNo,
      Date,
      Cost,
      CardNo,
```

```
    Membership,
    Status)
>

<!ELEMENT CustomerOrder
  (CustomerName,
   Address,
   FlightInfo)
>
```

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

IBM Redbooks

For information about ordering these publications, see "How to get IBM Redbooks" on page 277. Note that some of the documents referenced here may be available in softcopy only.

► *Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6*, SG24-6494

► *Patterns: Implementing Self-Service in an SOA Environment*, SG24-6680

► *Technical Overview of WebSphere Process Server and WebSphere Integration Developer*, REDP-4041

► *Getting Started with WebSphere Integration Developer and WebSphere Process Server*, SG24-7130

► *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303

► *WebSphere MQ Queue Sharing Group in a Parallel Sysplex environment*, REDP-3636

► *WebSphere MQ in a z/OS Parallel Sysplex Environment*, SG24-6864

► *Patterns: Self-Service Application Solutions Using WebSphere for z/OS V5*, SG24-7092

Other publications

These publications are also relevant as further information sources:

► *WebSphere MQ Application Programming Guide*, SC23-6595

    http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pb
    i.cgi?CTY=US&FNC=SRX&PBL=SC34659500

Online resources

These Web sites and URLs are also relevant as further information sources:

► IBM Service Oriented Architecture (SOA) Web page

  http://www-306.ibm.com/software/solutions/soa/

► *IBM SOA Foundation: providing what you need to get started with SOA*

  ftp://ftp.software.ibm.com/software/soa/pdf/SOA_g224-7540-00_WP_final.pdf

► WebSphere Application Server home page

  http://www.ibm.com/software/webservers/appserv/was/

► WebSphere MQ home page

  http://www.ibm.com/software/integration/wmq/

► WebSphere ESB home page

  http://www.ibm.com/software/integration/wsesb/

► WebSphere Message Broker home page

  http://www.ibm.com/software/integration/wbimessagebroker/

► WebSphere Process Server home page

  http://www.ibm.com/software/integration/wps/

► IH03: WebSphere Message Broker V6 - Message display, test, and performance utility

  http://www-1.ibm.com/support/docview.wss?rs=203&uid=swg24000637&loc=en_US&cs=utf-8&lang=en

► *PK15976; 6.0.2.3: handling of message headers by the WebSphere default provider*

  http://www-1.ibm.com/support/docview.wss?rs=180&context=SSEQTP&dc=D400&dc=D410&dc=D420&dc=D430&q1=JMS&uid=swg24011220&loc=en_US&cs=utf-8&lang=en

► IBM WebSphere Extended Deployment V6.0 Overview

  http://www-306.ibm.com/software/info/education/assistant/flow/wxd/6.0/Overview/

► Pattern Solutions Web page on IBM developerWorks

  http://www-128.ibm.com/developerworks/rational/products/patternsolutions/

► *IBM WebSphere WAS for z/OS and MQ* at:

  http://websphere.sys-con.com/read/148226.htm

► *Introducing XMS -- The IBM Message Service API*

  http://www-128.ibm.com/developerworks/websphere/library/techarticles/0509_phillips/0509_phillips.html

► *Contributing your own mediation primitive plug-in*

   http://www-1.ibm.com/support/docview.wss?rs=2308&context=SSQQFK&uid=swg2700
   7001

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

   ibm.com/redbooks

Help from IBM

IBM Support and downloads

   ibm.com/support

IBM Global Services

   ibm.com/services

Index

Symbols

.NET   21

A

Access services   15
ACORD   39
Adapter factory   99–100
Adapter support   39
AggregateControl   220
AggregateReply   220
AggregateRequest   220
AL3   39
Alias queue   29
   Create   129
Application clustering   51
Application design   63
Application server
   Bus member   123
Assemble phase   14, 16
Asynchronous communication   68
Augmenting   47

B

Backout threshold   85
Backward compatibility   105
Bar file   234
Binding   33, 41, 185, 208
Biztalk   39
Broker   35, 55, 234, 260
Broker administration   222
Broker domain   36, 223
Built-in node   218
Business application services   15
Business delegate   96, 100
Business delegate / service proxy   95
Business innovation and optimization services   15
Business object   41
   Create   190, 195
Business object map   43
Business process   5, 42
Business process engine   49

Business rule   42, 49
Business services   5
Business state machine   42

C

C++   21
Callout node   202–203
CFRM policy   163
Channel exit   30
Check   220
Client container   25
Cluster   59
   Application server   55
   Hardware   50–51
   Horizontal   56
   Software   51
   Vertical   56
   WebSphere Application Server   22–24, 26–27
   WebSphere ESB   32
     Cluster
       WebSphere Process Server   55
   WebSphere MQ   30, 52, 59
   WebSphere Process Server   40
COBOL   21, 61, 225, 228, 255–256
COBOL copy book   256
COBOL Copybook   39
Command messages   77
Common Business Event   41
Common Event Infrastructure   41
Communication model transparency   8–9, 95, 98
Component services   5
Component specification   93
Componentization   107
Compute   219, 226
Configuration Manager   36, 223
Connection factories   238
Connection Factory Name   241
Consumer   63
CORBA   25
Core group   57–58
Correlation ID   64, 73, 76, 78, 85, 87
CosNaming   25
Coupling facilities   53

279

Enabling SOA Using WebSphere Messaging

Enabling SOA using WebSphere Messaging

IBM®

Redbooks

Service-oriented architecture and messaging

ESB implementation with WebSphere Message Broker

ESB implementation with WebSphere ESB

Successfully implementing a service-oriented architecture (SOA) requires applications and infrastructure that can support the SOA principles. Applications can be enabled by creating service interfaces to existing or new functions hosted by the applications. The service interfaces should be accessed using an infrastructure that can route and transport service requests to the correct service provider. As organizations expose more and more functions as services, it is vitally important that this infrastructure supports the management of SOA on an enterprise scale.

This IBM Redbook looks at how IBM messaging products support an SOA environment. In particular, it looks at WebSphere Application Server, WebSphere Enterprise Service Bus, WebSphere MQ, and WebSphere Message Broker in an SOA environment. We discuss how they support SOA, compare the potential ESB product implementations, and show examples of building the infrastructure and creating mediations.