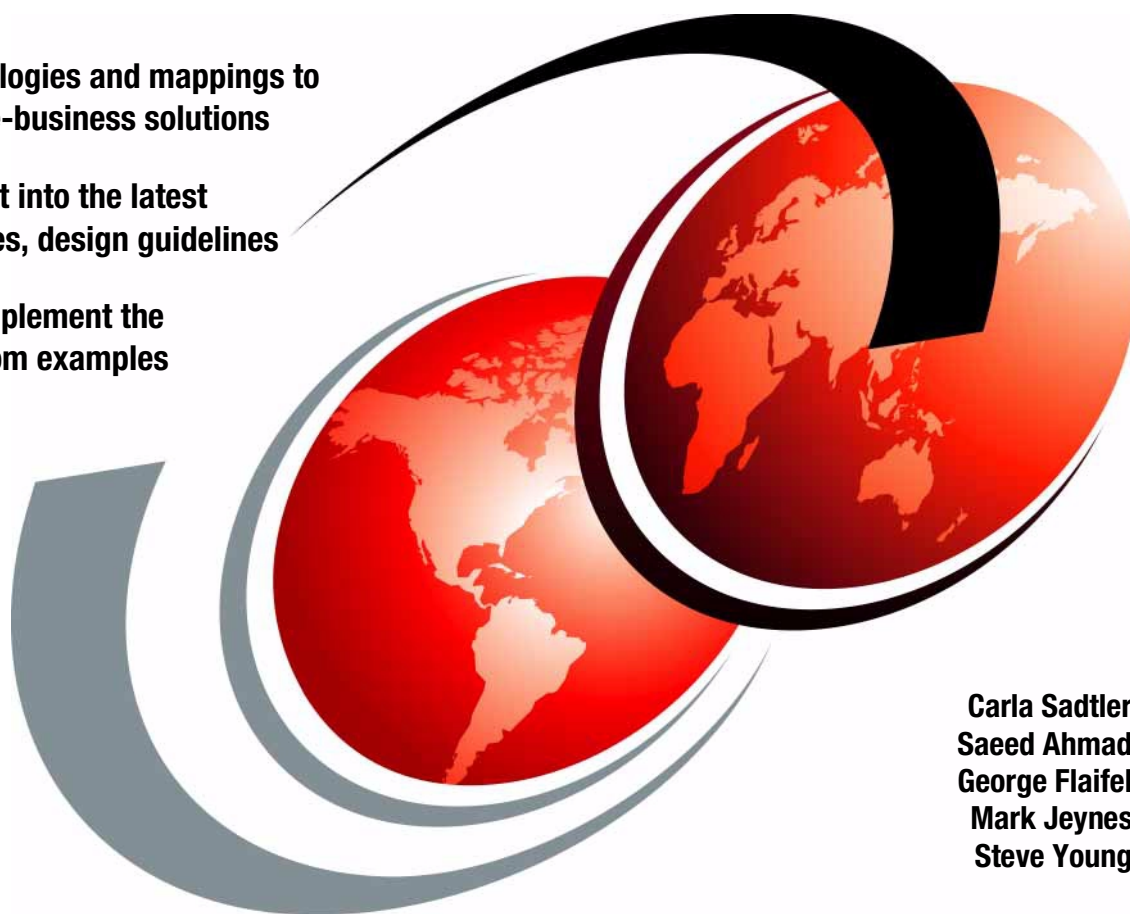


User-to-Business Patterns Using WebSphere Advanced and MQSI Patterns for e-business Series

Select topologies and mappings to
build U2B e-business solutions

Gain insight into the latest
technologies, design guidelines

Learn to implement the
solution from examples



Carla Sadtler
Saeed Ahmad
George Flaifel
Mark Jaynes
Steve Young

ibm.com/redbooks

Redbooks



International Technical Support Organization

**User-to-Business Patterns Using
WebSphere Advanced and MQSI
Patterns for e-business Series**

December 2000

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix C, "Special notices" on page 401.

First Edition (December 2000)

This edition applies to IBM WebSphere Application Server Advanced Edition Version 3 Release 5, Program Number 5648-C84 and MQSeries Integrator Version 2.0.1

Note

This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.

Comments may be addressed to:

IBM Corporation, International Technical Support Organization

Dept. HZ8 Building 678

P.O. Box 12195

Research Triangle Park, NC 27709-2195

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

© Copyright International Business Machines Corporation 2000. All rights reserved.

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Preface	xi
The team that wrote this redbook	xi
Comments welcome	xiii
 Chapter 1. Introduction	1
1.1 Patterns for e-business	1
1.1.1 Components of the Patterns for e-business	2
1.1.2 Defined Patterns for e-business	3
1.1.3 How to use these patterns	4
1.1.4 Patterns for e-business Web site	5
1.2 The User-to-Business Pattern	5
1.3 IBM MQSeries	6
1.4 IBM MQSeries Integrator	7
 Part 1. User-to-Business Patterns: topology 5	9
 Chapter 2. Choosing the application topology	11
2.1 Application topologies	11
2.1.1 Web-up	12
2.1.2 Enterprise-out	12
2.2 Application topology 5	14
2.2.1 Application topology 5: business driver	14
2.2.2 Application topology 5: key features	15
2.2.3 Application topology 5: considerations	16
 Chapter 3. Choosing the runtime topology	17
3.1 An introduction to the node types	17
3.2 Runtime topology A	20
3.3 Topology A variation 1	21
 Chapter 4. Product mapping	25
4.1 Product mappings for the basic topology	25
4.2 Runtime topology variation 1	27
4.3 Extending the topologies with workload management	29
4.3.1 MQSeries and MQSeries Integrator	29
4.3.2 WebSphere Advanced Edition	30
 Part 2. User-to-Business Patterns: guidelines	33
 Chapter 5. Technology options	35
5.1 Web client	36

5.1.1	Web browser	37
5.1.2	HTML	38
5.1.3	Dynamic HTML (DHTML)	39
5.1.4	XML (client-side)	39
5.1.5	JavaScript	40
5.1.6	Java applets	40
5.2	Web application server	42
5.2.1	Java servlets	43
5.2.2	JavaServer Pages (JSP)	44
5.2.3	JavaBeans	44
5.2.4	XML	45
5.2.5	JDBC	45
5.2.6	Enterprise JavaBeans	46
5.3	Integration server	48
5.3.1	Connectors	48
5.3.2	Message-oriented middleware	51
5.4	Additional enterprise Java APIs	53
5.5	References and where to find more information	53
Chapter 6. Java application design: using commands and MQSeries		55
6.1	Command framework	55
6.1.1	What are commands?	55
6.1.2	The command package	57
6.1.3	Command caching	58
6.1.4	Command classes	59
6.1.5	Command shipping example	60
6.1.6	Compensable commands	81
6.1.7	Local command example	86
6.2	Using MQSeries to send and retrieve data	95
6.2.1	MQSeries classes for Java	95
6.2.2	Java Messaging Service (JMS)	99
Chapter 7. MQSI application design guidelines		103
7.1	MQSeries and MQSI as message-oriented middleware	103
7.1.1	MQSeries - the MOM transport layer	103
7.1.2	MQSeries Integrator - transformation and integration	103
7.2	MQSeries Integrator topology	105
7.2.1	WebSphere-to-MQSI connection options	105
7.2.2	Queue manager roles and relationships	108
7.2.3	Placement of MQSI databases	113
7.3	MQSI message flow design	114
7.3.1	Design contract with the application	115
7.3.2	Message flow structure	115

7.3.3 Defining document types	116
7.4 Message flow components	117
7.4.1 Message flow inputs and outputs	117
7.4.2 IBM primitive nodes	120
7.4.3 Transformation nodes	121
7.4.4 Database nodes	127
7.4.5 Logic control nodes	129
7.4.6 Reusable message flows	135
7.4.7 Testing message flow components	139
Chapter 8. Application development guidelines	141
8.1 The scope of this book	142
8.2 Application development tools	143
8.2.1 Rational Rose	143
8.2.2 VisualAge for Java	144
8.2.3 WebSphere Studio	144
8.2.4 How these tools fit together	144
8.3 WebBank problem domain	146
8.4 Solution outline	146
8.5 Macro design	147
8.5.1 Creating a business process model	147
8.5.2 Information architecture	147
8.5.3 Technology choices	151
8.5.4 Deployment model	152
8.6 Micro design	153
8.6.1 Use cases	153
8.6.2 Storyboard	157
8.6.3 Activity diagrams	162
8.6.4 Class models and class diagrams	164
8.6.5 Interaction diagrams	172
8.6.6 Component model	176
8.6.7 Deployment model	179
Chapter 9. Developing the MQSI application	183
9.1 The contract with WebSphere	183
9.2 Design considerations	183
9.2.1 Customer profile lookup	184
9.2.2 Customer profile update	184
9.3 Operational entities	184
9.3.1 Application databases and tables	185
9.3.2 Messages and documents	187
9.4 Identify the general operations	188
9.4.1 Customer profile lookup operational components	188

9.4.2	Customer profile update operational components	188
9.5	Identify the operational components	189
9.5.1	Customer profile lookup functional components	189
9.5.2	Customer profile update functional components	191
9.6	Building the message flows	193
9.6.1	Creating a message flow	193
9.6.2	Organizing message flows	194
9.6.3	Message flow: "ITSO Cache Lookup: Profile from Request" . . .	195
9.6.4	Message flow: "ITSO Cache Update: from Profile"	200
9.6.5	Message flow: "ITSO Accounts from Request"	204
9.6.6	Message flow: "ITSO First Account From Accounts"	206
9.6.7	Message flow: "ITSO Profile from Savings"	212
9.6.8	Message flow: "ITSO Profile from Checking"	214
9.6.9	Message flow: "ITSO Profile: add Accounts"	218
9.6.10	Message flow: "ITSO Looper"	220
9.6.11	Message flow: "ITSO Updates from Profile w/accounts"	221
9.6.12	Message flow: "ITSO Update Router"	223
9.6.13	Message flow: "ITSO Update Savings: from Update"	224
9.6.14	Message flow: "ITSO Update Checking: from Update"	226
9.7	Piecing together the lookup components	228
9.7.1	Customer profile lookup	228
9.7.2	Customer profile update	231
9.8	Tracing	234
Chapter 10.	System management guidelines	239
10.1	MQSeries system management	239
10.1.1	MQSeries administration interfaces	240
10.1.2	Remote administration	246
10.1.3	Administration interface guidelines	248
10.1.4	Overview of the MQSeries clustering feature	249
10.1.5	MQSeries security	253
10.1.6	MQSeries monitoring	258
10.1.7	MQseries restart and recovery	262
10.2	.MQSeries Integrator system management	265
10.2.1	Message brokers	267
10.2.2	The Configuration Manager	269
10.2.3	The Control Center	269
10.2.4	The User Name Server	270
10.2.5	MQSeries guidelines for MQSI	271
10.2.6	MQSI databases	273
10.2.7	MQSeries Integrator commands and operations	274
10.2.8	Control Center operations	278
10.2.9	Resource definition management	288

10.2.10 MQSeries Integrator security	290
10.2.11 MQSeries Intergrator backup and recovery.	294
10.2.12 MQSeries Integrator monitoring	300
Part 3. Working example	301
Chapter 11. Introduction to the working example	303
11.1 Sample application	303
11.1.1 Application flow	303
11.2 Runtime topologies.	305
11.2.1 Product documentation, software, and support	307
11.3 Web application server.	308
11.3.1 Web application server running on Windows NT.	309
11.3.2 Web application server running on AIX.	310
11.4 MQSI broker.	310
11.4.1 Running the broker on Windows NT	311
11.4.2 Running the broker on AIX	311
11.4.3 MQSI service	312
11.5 MQSI Configuration Manager.	312
11.6 User Name Server	313
11.6.1 Running the User Name Server on Windows NT.	313
11.7 Database server.	314
11.7.1 Running DB2 on Windows NT.	314
11.7.2 Running DB2 on AIX.	315
11.8 Planning user IDs.	315
11.8.1 User IDs for the Windows NT mapping	316
11.8.2 User IDs for the AIX mapping	317
Chapter 12. MQSeries and MQSI implementation.	319
12.1 Lab environment.	319
12.1.1 Windows NT test configuration	319
12.1.2 AIX test configuration	320
12.1.3 MQSeries and MQSI configuration methods	322
12.2 Defining user IDs	322
12.2.1 MQSI:	322
12.2.2 DB2 server:	322
12.3 MQSI database setup.	322
12.4 MQSI User Name Server setup	324
12.4.1 Create the queue manager.	325
12.4.2 Create the MQSI User Name Server.	328
12.4.3 Starting the User Name Server.	330
12.5 MQSI Configuration Manager setup	330
12.5.1 Define the databases to the local system	331

12.5.2	Create the queue manager	333
12.5.3	Create the MQSI Configuration Manager	334
12.6	Define the MQSeries cluster.	336
12.7	MQSI broker setup	345
12.7.1	Define the database to the local system	345
12.7.2	Create the queue manager	347
12.7.3	Creating the MQSI broker.	352
12.8	Using the Control Center to deploy an application	355
12.8.1	Connecting to the broker	357
12.8.2	Creating an execution group.	359
12.8.3	Importing message flows	360
12.8.4	Assigning the message flows to the execution group	362
12.8.5	Saving the configuration and deploying it to the broker.	363
12.9	Preparing the broker for the application	366
12.9.1	Create the application databases	366
12.9.2	Define the required MQSeries queues	367
Chapter 13.	WebSphere Application Server setup	371
13.1	MQSeries SupportPac MA88	371
13.1.1	Classpath settings	372
13.1.2	Configuring JMS.	373
13.2	Deploying the application to WebSphere	375
13.3	Copy the DTDs to the operating system	378
Appendix A.	Rational Rose 2000e and VisualAge for Java.	381
A.1	Forward and reverse engineering with Rational Rose	381
13.3.1	Integration with IBM VisualAge for Java	381
A.1.1	Rose to Java mapping	382
A.2	Installation notes.	383
A.3	Configuration.	383
A.3.1	VisualAge for Java configuration	384
A.3.2	Rational Rose configuration	384
13.4	Linking a Rose model to a VisualAge for Java project	388
A.4	Forward engineering with Rose	388
A.4.1	Generating code from classes	389
A.4.2	Generating code from components	389
A.5	Reverse engineering.	390
Appendix B.	Sample code.	393
B.1	GetCurrentProfileCommandMQJava: retrieveProfile() method	393
B.2	GetCurrentProfileCommandJMS: retrieveProfile() method.	397

Appendix C. Special notices	401
Appendix D. Related publications	405
D.1 IBM Redbooks	405
D.2 IBM Redbooks collections	405
D.3 Other resources	406
D.4 Referenced Web sites	407
Appendix E. Using the additional material	409
E.1 Locating the additional material on the Internet	409
E.2 Using the Web material.	409
E.2.1 System requirements for downloading the Web material	409
E.2.2 How to use the Web material	409
How to get IBM Redbooks	411
IBM Redbooks fax order form	412
Index	413
IBM Redbooks review	419

X User-to-Business Patterns with WebSphere Advanced and MQSI

Preface

Patterns for e-business are a group of proven, reusable assets that can help speed the process of developing applications. The pattern discussed in this book, the User-to-Business Pattern, is the general case of users interacting with enterprise transactions and data. In particular it is relevant to those enterprises that deal with goods and services that cannot be listed and sold from a catalog.

This redbook discusses application topology 5 of the User-to-Business Patterns. Application topology 5 links multiple presentation tiers to any back-end client, but the back-end is not hidden to the user.

The topologies are illustrated using WebSphere Application Server Advanced Edition V3.5, MQSeries V5.1, and MQSeries Integrator V2. The sample application uses the Command Manager Framework, included with WebSphere V3.5.

Part 1 of this redbook takes you through the process of choosing an application topology and a runtime topology. It then gives you possible product mappings for implementation of the chosen runtime topology.

Part 2 provides a set of guidelines for building your e-business application. It includes information on technology options, application design and application development.

Part 3 takes you through a working example, showing the implementation of an e-business application using application topology 5.

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.

The overall manager for Patterns:

Jonathan Adams is an IT consultant with the IBM Software Group and the leader of the Patterns for e-business initiative. He works closely with all areas of IBM and industry consultants. His commitment to the idea of a systematic approach to end-to-end e-business architecture is based on his many years of work in the field with major IBM customers in the United Kingdom.

The ITSO team:

Carla Sadtler is a Senior Software Engineer at the International Technical Support Organization, Raleigh Center. She writes extensively in many areas including WebSphere, SecureWay Communications Servers, network integration, and Web-to-host integration products. Before joining the ITSO 14 years ago, 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.

Saeed Ahmad is an IT Architect working in the Knowledge and Content Management practice of IBM Global Services in the UK. His area of expertise is in using object-oriented methods and technologies. He has worked extensively in enterprise application integration projects and has helped corporate customers to design and build scalable Web services. Saeed has cross-industry experience, providing consultancy in the adoption of new technologies, and performing architect or lead development roles on systems integration projects. Saeed holds a Masters Degree in Engineering, Manufacturing and Management from the University of Manchester.

George Flaifel is a senior IT Specialist in e-business Services and Consulting in IBM Canada. He joined IBM in 1985 as a Software Customer Engineer. His areas of expertise include S/390, CICS, and MQSeries systems programming and technical support, OS/390 UNIX System Services implementation and integration, and most recently WebSphere, specializing in WebSphere administration and back-end integration.

Mark Jeynes is an IT Specialist in the IBM Software Group Services at IBM Laboratories, Hursley. He provides a range of technical services for the EAI product suite offerings including the design and deployment of solutions based on MQSeries, MQSeries Integrator and MQSeries Everyplace. He has experience on the Windows NT, AIX, Solaris, Linux and OS/390 platforms, Java and Perl, and extensive COBOL/CICS/JCL experience from earlier work on what is now OS/390. His IT career spans 16 years, having worked mainly for IT software services companies and also in the finance and retail industries. He is currently studying at Oxford University for an MSc degree in Software Engineering.

Steve Young is a Software Engineer working in the UK Advanced Solutions Group in Hursley, England. He has worked at IBM for two years, producing fully functional demos and test drives. He started out working on CICS for OS/390 demos, and has recently taken over the Pattern Development Kit (PDK). His skills are predominantly Java and WebSphere Application Server. Steve holds

an Honours degree in Drama from the University of Manchester, and a Masters in Computer Science from the University of Bristol.

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

Geert Van de Putte
International Technical Support Organization, Raleigh Center

Geoffrey Hambrick
IBM Austin

Michael Conner
IBM Austin

Comments welcome

Your comments are important to us!

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

- Fax the evaluation form found in “IBM Redbooks review” on page 419 to the fax number shown on the form.
- Use the online evaluation form found at ibm.com/redbooks
- Send your comments in an Internet note to redbook@us.ibm.com

Chapter 1. Introduction

This book is a part of the Patterns for e-business Redbook series. It will discuss User-to-Business Pattern application topology 5. An application topology 5 example will be used to illustrate the use of the command package included in WebSphere Application Server Advanced Edition Version 3.5 and to introduce the use of MQSeries Integrator (MQSI) as a router.

This information will build on what was introduced in *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864. We will refer to that book often and it is highly recommended that you have a copy handy when reading this book.

Another book in the series, *User-to-Business Patterns Using WebSphere Enterprise Edition*, SG24-6151, addresses the use of Component Broker in a topology 5 and 6 situation. This can be considered optional reading.

In addition, there are four other books with supporting technical information on application development that we will reference several times:

1. *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
2. *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application*, SG24-5754
3. *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514
4. *Business Integration Solutions with MQSeries Integrator*, SG24-6154

1.1 Patterns for e-business

The Patterns for e-business aim is to communicate in a highly accessible fashion the business pattern, systems architecture (application and runtime topologies), product mappings, and guidelines required for different classes of applications. For some patterns there is also an associated Pattern Development Kit (PDK), which provides sample application code to illustrate effective use of those patterns.

The goal is to provide the smallest number of Patterns for e-business that will allow IT architects in 80% of cases to quickly develop 80% of their required infrastructure by the re-use of proven:

- Architecture patterns
- Design patterns

- Runtime patterns
- Application development and systems management patterns
- Design, development, and deployment guidelines
- Code

1.1.1 Components of the Patterns for e-business

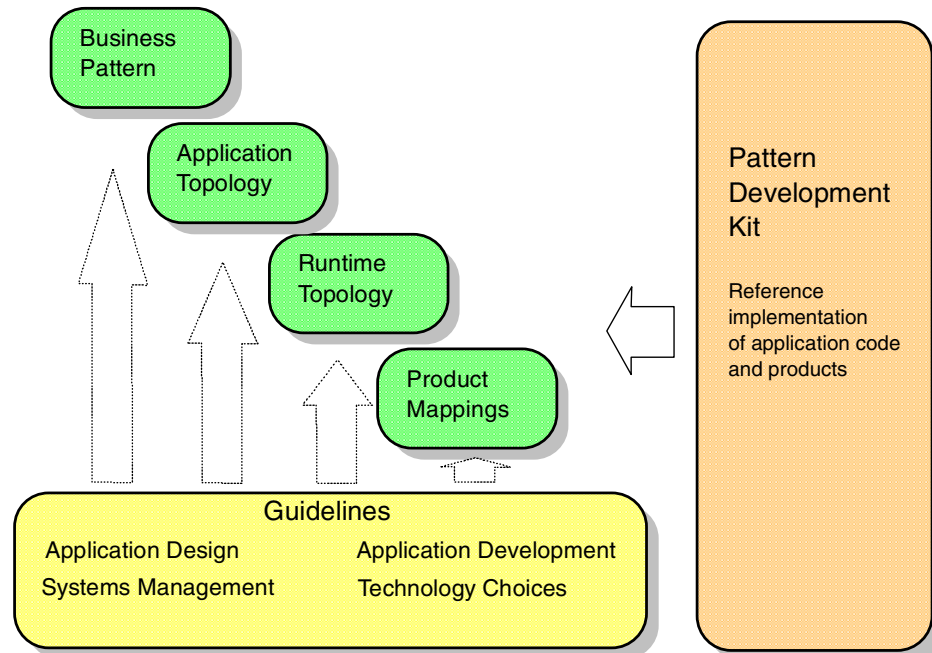


Figure 1. Patterns for e-business

Business patterns describe the interaction between the participants in an e-business solution. The following make up the basic structure of a pattern:

- Application topologies illustrate the various ways to configure the interaction between users, applications, and data. Choosing an application topology will lead to an underpinning runtime topology.
- Runtime topologies use nodes to group functional requirements. The nodes are interconnected to solve a business problem.
- Product mappings show possible combinations of products used to instantiate the runtime topology.
- Guidelines outline and define the processes used to build the e-business application.

1.1.2 Defined Patterns for e-business

There are currently six defined Patterns for e-business:

- **User-to-Business** is the general case of users (internal or external) interacting with enterprise transactions and data. In particular it is relevant to those enterprises that deal with goods and services that cannot be listed and sold from a catalog. It can also be thought of as covering all user-to-business interactions not covered by the User-to-Online Buying pattern.
- **User-to-Online Buying** is used to describe the special case (a subset of the User-to-Business Pattern) where packaged goods, say, are sold through a catalog using a shopping cart, a wallet, etc. This includes both consumers purchasing goods and online buyers purchasing goods from a single supplier. It can also include links to back-end systems to allow for inventory updates and credit checking.
- **Business-to-Business** is used to describe two styles of inter-business to business (Intra-business to business is covered under Application Integration below):
 - The first style (B2Bi) covers programmatic links between arms-length businesses (where potentially a trading partner agreement may be appropriate). A good example of this would be supply chain applications.
 - The second style covers the e-Marketplace where the model supports B2M2B. The “M” represents the e-Marketplace, which supports multiple buyers and suppliers. The buying function may be performed online or programmatically.
- **User-to-User** is used to describe users collaborating with one another by e-mail, shared documents, etc.
- **User-to-Data** is used to describe users needing to take large volumes of data, text, images, video, etc. and use tools to extract useful information from it.
- **Application Integration** is used to link applications together within a business (such as ERP with existing applications). This can be used within a business pattern or between business patterns.

IBM views e-business as an integration of many application domains into systems that connect a business with its customers, partners, and suppliers. These systems are not confined to Web interfaces, although increasingly many of the user interfaces to the combined system will use Web technology.

The common set of node descriptions in the Patterns for e-business enable communication between architects and designers from very different application domains and will suggest areas for shared nodes and infrastructure.

This is similar to the process of using design patterns to solve a programming design problem, where classes in the composed pattern play multiple roles, derived from the source patterns. It is different, however, in that design pattern composition is based on class diagrams and white box by nature, whereas composing architectural patterns is more component-based.

The Patterns for e-business may be applied to e-business solution areas. Here is a guide to where you may find them most applicable:

Table 1. Patterns for e-business and e-business solutions

e-business solution area	Business pattern
Customer relationship management	User-to-Business Pattern
e-commerce	User-to-Online Buying Pattern
Supply chain management, e-Marketplace	Business-to-Business Pattern
Collaboration	User-to-User Pattern
Business Intelligence; Knowledge Management	User-to-Data Pattern
Business application integration	Application Integration Pattern

1.1.3 How to use these patterns

The Patterns for e-business are particularly focused upon addressing common business application problems and providing answers to frequent architecture, design, and implementation questions.

We recommend that you can use the Patterns for e-business in a number of ways according to your needs:

- As a starting point for an end-to-end system architecture.
- As a detailed example and prescriptive approach, following the product mappings and guidance provided.
- As a way to design more complex, multi-channel systems, when several patterns are used together.

As with the design patterns and ESS work, we anticipate that architects and designers will want to combine these patterns to compose solutions to more complex system architectures.

We recommend that you use the Patterns for e-business together with an appropriate development methodology that considers the full set of requirements that are to be understood and implemented, whether these requirements concern the function of the solution or its operational characteristics such as availability, scalability, or performance.

1.1.4 Patterns for e-business Web site

The Patterns for e-business are published on IBM developerWorks, a portal for developers, and can be located at:

<http://www.ibm.com/software/developer/web/patterns>.

This interactive patterns site acts as a guide to aid you in the selection of the pattern and topologies most relevant to your needs. While you can navigate via shortcuts to the information you most need, the site is structured to enable you to “drill down” into the material as you:

1. Select a business pattern.
2. Select an application topology.
3. Review runtime topologies.
4. Review product mappings.
5. Review guidelines.

At the time of writing, the Web site has material for the User-to-Business and User-to-Online Buying Patterns, with material for the other business patterns in the process of development.

You can also register at this site for pattern-related updates, which will include information about the Pattern Development Kit for User-to-Business.

1.2 The User-to-Business Pattern

As mentioned earlier, the User-to-Business Pattern covers the general case of users interacting with enterprise transactions and data. In particular it is relevant to those enterprises that deal with goods and services that cannot be listed and sold from a catalog. It can also be thought of as covering all user-to-business interactions not covered by the User-to-Online Buying Pattern.

1.3 IBM MQSeries

IBM MQSeries provides messaging and queueing capability, allowing programs on a variety of platforms to communicate with each other across a network of unlike components. Applications place messages on an MQSeries queue for delivery to a target application. At that point, MQSeries takes over, providing the transport mechanism and ensuring delivery to the target application. If the target application is not available, the message stays on the queue until it can be delivered.

Two communication models exist with MQSeries. In a *point-to-point* application, the sending application knows the destination of the message. These applications can send messages that do not require a response, or they may specify a destination for the response message. This book will be using a point-to-point application to illustrate the routing capabilities of MQSeries.

Publish/subscribe applications have a publisher that distributes information by sending it to a broker. Subscribers tell the broker what topics of information they are interested in and the broker sends published information to the subscribers for that topic. Brokers can exchange subscription information and publications with each other, allowing subscribers to receive published information from any broker on topics they have subscribed to. Publish/subscribe applications are more appropriate for personalization scenarios described by application topology 7, which is not covered in this book.

Applications communicate with each other by using a local MQSeries queue manager to put messages on, or receive messages from, a queue. MQSeries provides several application programming interfaces:

- Message Queue Interface (MQI) provides a consistent application programming interface that enables programs to talk to the local queue manager in order to send and receive data.
- Application Messaging Interface (AMI) provides a simple interface for application programmers to MQI. AMI is available in the C, COBOL, C++ and Java programming languages.
- MQSeries C++ allows you to write MQSeries application programs in C++.
- MQSeries classes for Java (MQ base Java) allows a program written in Java to connect to MQSeries as a client using TCP/IP or directly to an MQSeries server.

- MQSeries classes for Java Message Service (MQ JMS) implements Sun's Java Message Service (JMS) interfaces to enable JMS programs to access MQSeries systems. JMS is an open standard and offers some additional features that are not present in MQ base Java. MQ JMS supports both the point-to-point and publish/subscribe communication models.

MQSeries messaging and queueing is supported on over 35 platforms. MQSeries also provides a number of connectors and gateways to other products, such as Lotus Domino, Microsoft Exchange, SAP/R3, CICS, and IMS.

Details on MQSeries connectivity can be found at:

<http://www.ibm.com/software/ts/mqseries/platforms/>.

1.4 IBM MQSeries Integrator

MQSeries Integrator (MQSI) Version 2.0 extends the messaging capabilities of MQSeries by adding message broker functionality driven by business rules. It provides the intelligence to route and transform messages, the possibility to filter messages (topic-based or content-based), and database capabilities for enrichment of the messages or for warehousing the messages. It also provides a framework for extending the functionality with plug-ins to user-written or third-party solutions for specific requirements.

MQSI is built on MQSeries. Applications using MQSeries to communicate with other applications put messages on a queue. MQSI can retrieve messages from the queue, perform some action based on business rules, and put the resulting message on an output queue to be sent to the target application. Message flows in MQSI are responsible for performing the desired actions. For example, with message flows, you can:

- Transform the format of a message, allowing dissimilar applications to exchange information.
- Route messages to their destination based on the message or message header contents.
- Store and retrieve information in a database.
- Modify the contents of a message.
- Publish a message to make it available to other applications.
- Create structured topic names, topic-based access control functions, content-based subscriptions, and subscription points.

In other words, MQSI is a simple yet effective way of processing messages en-route to their destination. It is designed to allow customers to extend their technology without changing existing applications.

MQSI V2.0.1 is currently available on the following platforms:

- AIX
- Sun Solaris
- Windows NT

MQSI V1.1 is available on the following platforms:

- AIX
- AS/400
- HP-UX
- Windows NT
- OS/390
- Sun Solaris

In this book we will be discussing MQSI V2.0.1. For the latest information on MQSI availability, check

<http://www.ibm.com/software/ts/mqseries/integrator/>.

Chapter 2. Choosing the application topology

After identifying the business pattern, in this case, the User-to-Business Pattern, the next step in planning an e-business application is to choose the logical application topology that applies to your situation. We will give you a brief overview of the eight defined application topologies for the User-to-Business Pattern. The rest of the book will concentrate specifically on application topology 5.

2.1 Application topologies

The following are typical User-to-Business application topologies. These application topologies do not show middleware, but focus on the shape of the application, the application logic, and associated data.

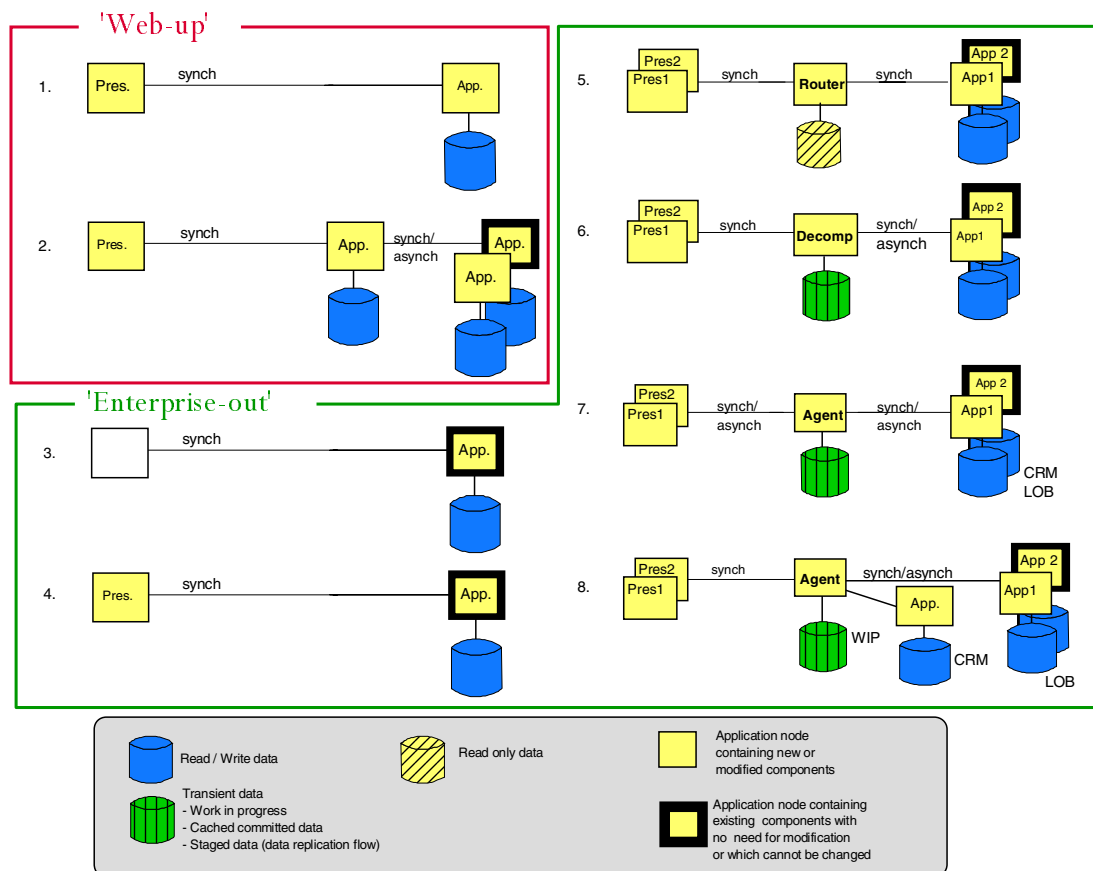


Figure 2. User-to-Business pattern application topologies

2.1.1 Web-up

Web-up is used to represent both an attitude of mind (Web-centric) and the consequent implementation of new Web browser-centric applications focussing only on the Web channel. The data may be acquired by replication from existing sources or typed in by new users from the Web. The two Web-up application topologies (1 and 2) are covered in detail in *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864.

Topology 1 links a presentation node to business logic residing on the second tier. The second tier can access a local database maintaining the application data.

Topology 1 represents the target topology for most Web-up application server vendors. It aims to fix the scalability problems of client/server and at the same time provide re-use of the business logic and data by all styles of Web browsers. Many vendors promote ease of development by a mixture of scripting and components with little attention to layering the application with separate presentation and business logic layers. This should be avoided. It should also be noted that where the business logic and data are held outside the glasshouse there will be a significant system management cost to manage this asset. The business driver for this is currently to extend the current Web-enabled publishing capability with an e-commerce capability with no back-end integration, a classic Web-up strategy.

Topology 2 is similar to topology 1, but the business logic on the second tier can access existing applications and data on the third tier.

Topology 2 represents an extension to the Web-up strategy in topology 1. It allows for one or more point-to-point connections to back-end heritage applications or databases so that Web applications can be integrated with existing back-end applications (for example, an e-commerce application integrated with a back-end inventory management applications).

One of the key issues to address is how this topology will be deployed to avoid systems management complexity arising from corporate data on more than one tier.

2.1.2 Enterprise-out

Enterprise-out is used to represent an enterprise-centric attitude of mind and the consequent extension of existing applications out to the new Web channel. Hence support for multiple channels is required.

Topology 3 represents a very thin client with a 3270/5250/ASCII emulator or Host On-Demand accessing an existing application. The business driver for this is to provide intranet access to existing green-screen applications without having to rewrite/re-engineer these applications. It is a very simplistic enterprise-out scenario.

Topology 4 represents a thin client (for example, a CICS ECI bean or IBM Host Publisher) accessing an existing application. The business driver for this is to provide a customized presentation of existing centralized applications without having to rewrite/re-engineer these applications. Care should be taken in the physical implementation so as not to cause an unsupportable number of connections from the Web. Like topology 1, it is a very simplistic enterprise-out scenario.

Topology 5 links multiple presentation tiers to any back-end client, but the back-end is not hidden from the user, for example a call center or Web browser (multiple presentation) linked to the back-end through an application router.

Topology 5 represents a typical topology used to Web-enable existing robust, highly scalable transactions. The extra scalability requirements generated by enabling thousands of Web users may involve security, protocol conversion, session concentration and routing. The business driver for this design is fast, highly scalable, highly available Web-enablement of existing business transactions, that is, a classic Web server enterprise-out strategy. The key feature is the one-to-one-to-one relationship between the tiers at runtime. This design is also valuable in the case of company mergers and takeovers.

Topology 6 is similar to topology 5, but the mechanics of the back-end applications are hidden. A business request from one of the presentation tiers can be decomposed on the middle tier into multiple back-end transactions, which are then recomposed on the middle tier to return a single business response to the presentation tier.

Topology 6 represents a step beyond topology 5 for the enterprise that wants to make third-tier applications seamless by integrating the business logic at the intermediate tier. This design is increasingly valuable because it can support multiple styles of thin client with a common intermediate tier. The key feature is the one-to-one-to-many relationship between the tiers at runtime. The business driver is to provide customer-oriented support systems rather than product-oriented systems.

Topology 7 represents mass customization. For example when someone uses a browser to check an insurance claim, the application does cross

selling based on personal data. For example, an insurance customer contacts an insurance company to check the status of an insurance claim. While answering his query the system can also retrieve any information about his family, house, car, birthdays, etc. from the corporate data repositories. This data can be held as work-in-progress on the intermediate tier and used to prompt the insurance customer with cross-selling opportunities. The key feature is the provision of a push mechanism from the second to first tier. The business driver for this design is customizing goods and services to a market of one (mass customization).

Topology 8 is similar to topology 7 but the customer data is on a different machine/platform from the application server.

In topology 7 the customer relationship management (CRM) data and the line of business (LOB) data are both held on the same third tier. In topology 8 the CRM data is held on the third tier that owns the customer relationship, while the LOB data is accessed from the third tier of various third-party suppliers. The business driver for this design is the need to support virtual enterprises, intermediaries, or portals on the Web.

2.2 Application topology 5

Topology 5 represents a typical topology used to Web-enable existing, robust, highly scalable transactions. The extra scalability requirements generated by enabling thousands of Web users may involve security, protocol conversion, session concentration, and routing. Topology 5 links multiple delivery channels to any back-end client, but the back-end is not hidden to the user (for example a call center or Web browser linked to the back-end through an application router).

2.2.1 Application topology 5: business driver

The business driver for this design is fast, highly scalable, highly available Web enablement of existing business transactions. There are multiple delivery channels and multiple back-end applications. Each application stands on its own. There is no need to combine information or features from multiple applications into a single response to the user.

For example, a business could provide access to an order entry application from customer server representatives in a call center. At the same time, it could provide access to inventory data to suppliers using an intranet connection.

2.2.2 Application topology 5: key features

The key feature of topology 5 is the one-to-one-to-one relationship between the tiers at runtime. The application is divided into three logical tiers: a presentation tier, a router tier, and a back-end application tier.

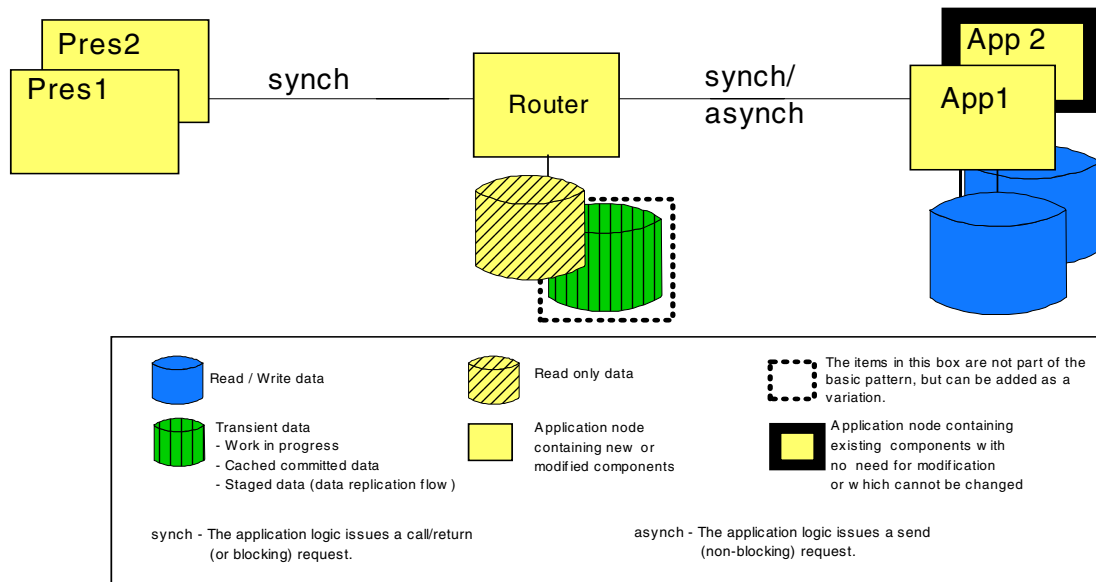


Figure 3. Application topology 5

The presentation tier is responsible for the interface into the Web application. It is responsible for all the presentation logic of the application.

The router tier links a node in the presentation tier to a particular application in the back-end application tier. The router tier provides a common interface to multiple back-end applications, facilitating the connection, but containing no business logic to combine the application data. It has access to data that can affect the routing decision or that can be used to modify the message that is routed. The true application topology 5 calls for this data to be read/only. A variation on this is also possible, allowing the data to be used by the router for caching, logging, or staging data.

The back-end application tier is responsible for all the business logic and data access of the application. It may be a new application developed specifically for the Web application, or it may be an existing legacy application that may or may not require modification.

The communication between the presentation and router tiers is synchronous, meaning that the application logic issues a call/return (or blocking) request.

The connection between the router tier and the back-end application tier can be a fast asynchronous or synchronous connection, depending on the characteristics and capabilities of the back-end system. Asynchronous communication means the application logic issues a send (non-blocking) request. If using synchronous communication, the application must be highly available and scalable, qualities usually found in robust transaction processing systems.

Since the presentation logic and business logic are separated, it is easy to adapt the presentation node to new kinds of clients. The easiest approach is to use thin browser-based clients. But it is also possible to extend the presentation logic to new client platforms, for example, client Java applications or Web appliances such as Web-enabled cellular phones or personal digital assistants (PDAs), without the need to change the business logic in the application node.

2.2.3 Application topology 5: considerations

Traditional transaction monitors have been typically used to support the application router on the middle tier. These may still be appropriate unless you have a requirement to move rapidly to application topology 6 and beyond. For this level of functionality more advanced middleware is required, for example, WebSphere Enterprise Edition or decomposition/recomposition support anticipated in future releases of MQSI.

Chapter 3. Choosing the runtime topology

Application topology 5 represents a starting point for delivering a sophisticated e-business application that delivers information from back-end systems to multiple delivery channels, while exploiting the data integrity and performance of legacy applications.

Now that the application topology has been chosen, it is time to choose the runtime topology that most closely matches the requirements of the application. A runtime topology uses nodes to group functional and operational components. The nodes are interconnected to solve a business problem. Each application topology leads to one or more underlying runtime topologies.

3.1 An introduction to the node types

A runtime topology will consist of several nodes representing specific functions. Most topologies will consist of a core set of common nodes, with the addition of one or more nodes unique to that topology. To understand the runtime topologies, you will need to review the following node definitions.

Integration server

An integration server hosts application logic that can access and use information from existing databases, transaction functions from transaction monitor systems, and application capabilities from application packages. The integration server can access back-end applications individually or combine this information and function in new ways.

At a minimum, the integration server acts as an integration point for multiple presentation tiers (for example, call centers, branch offices, Web browsers) so that they can share the infrastructure and applications on tiers 2 and 3.

Web application server

A Web application server node is an application server that includes an HTTP server (also known as a Web server) and is typically designed for access by HTTP clients and to host both presentation and business logic.

The Web application server node is a functional extension of the informational (publishing-based) Web server. It provides the technology platform and contains the components to support access to both public and user-specific information by users employing Web browser technology. For the latter, the node provides robust services to allow users to communicate with shared

applications and databases. In this way it acts as an interface to business functions, such as banking, lending, and HR systems.

This node would be provided by the company on company premises, or hosted in the enterprise network inside a demilitarized zone (DMZ) for security reasons. In most cases, access to this server would be in secure mode, using services such as SSL or IPSec.

In the simplest design, this node can provide the management of hypermedia documents and diverse application functions. For more complex applications or those demanding stronger security, it is recommended that the application be deployed on a separate Web application server node inside the internal network.

Data that may be contained on the node includes:

- HTML text pages, images, multimedia content to be downloaded to the client browser
- JavaServer Pages (JSP) files
- Servlets, enterprise beans
- Application program libraries, for example, Java applets for dynamic downloading to client workstations.

Public Key Infrastructure (PKI)

PKI is a collection of standards-based technologies and commercial services to support the secure interaction of two unrelated entities (for example, a public user and a corporation) over the Internet. In the context of the topologies defined in this redbook, PKI supports the authentication of the server to the browser client, using the SSL protocol.

Domain Name Service (DNS) node

The DNS node assists in determining the physical network address associated with the symbolic address (URL) of the requested information. The DNS is that of the Internet Service Provider, although DNS is implemented on the accessed site, too.

User node

This node is most frequently a personal computing device (PC, etc.) supporting a commercial browser, for example, Netscape Navigator or Internet Explorer. The level of the browser is expected to support SSL and some level of DHTML. Increasingly, designers should also consider that this node may be a pervasive computing device, such as a Personal Digital Appliance (PDA).

Directory and security services node

This node supplies information on the location, capabilities and various attributes (including user ID/password pairs and certificates) of resources and users known to this Web application system. The node may supply information for various security services (authentication and authorization) and may also perform the actual security processing, for example, to verify certificates. The authentication in most current designs validates the access to the Web application server part of the Web server, but it can also authenticate for access to the database server.

Protocol firewall and domain firewall nodes

Firewalls provide services that can be used to control access from a less trusted network to a more trusted network. Traditional implementations of firewall services include:

- Screening routers (the protocol firewall in this design)
- Application gateways (the domain firewall)

The two firewall nodes provide increasing levels of protection at the expense of increasing computing resource requirements. The protocol firewall is typically implemented as an IP router, while the domain firewall is a dedicated server node.

Web server redirector node

In order to separate the Web server from the application server, a so-called *Web server redirector node* (or just *redirector* for short) is introduced. The Web server redirector is used in conjunction with a Web server. The Web server serves HTTP pages and the redirector forwards servlet and JSP requests to the application servers. The advantage of using a redirector is that you can move the application server behind the domain firewall into the secure network, where it is more protected than within the demilitarized zone (DMZ). Static pages can be served from the DMZ by this node.

The redirector can be implemented, for example, by either a reverse proxy server or by a Web server plug-in such as the remote OSE function of IBM WebSphere Application Server Advanced Edition.

Existing applications and data node

Existing applications are run and maintained on nodes that are installed in the internal network. These applications provide for business logic that uses data maintained in the internal network. The number and topology of these existing application and data nodes is dependent on the particular configuration used by these legacy systems.

3.2 Runtime topology A

Runtime topology A for application topology 5 consists of a basic topology and one variation. In application topology 5, the router tier serves as an integration point for delivery channels in the presentation tier, allowing access to individual back-end applications. In runtime topology A, the functions of the router tier are performed by an integration server. The functions of the presentation tier are performed by a Web application server.

The basic runtime topology features a single Web application server physically located in the DMZ. The integration server, third-tier applications, and data are located in the internal network.

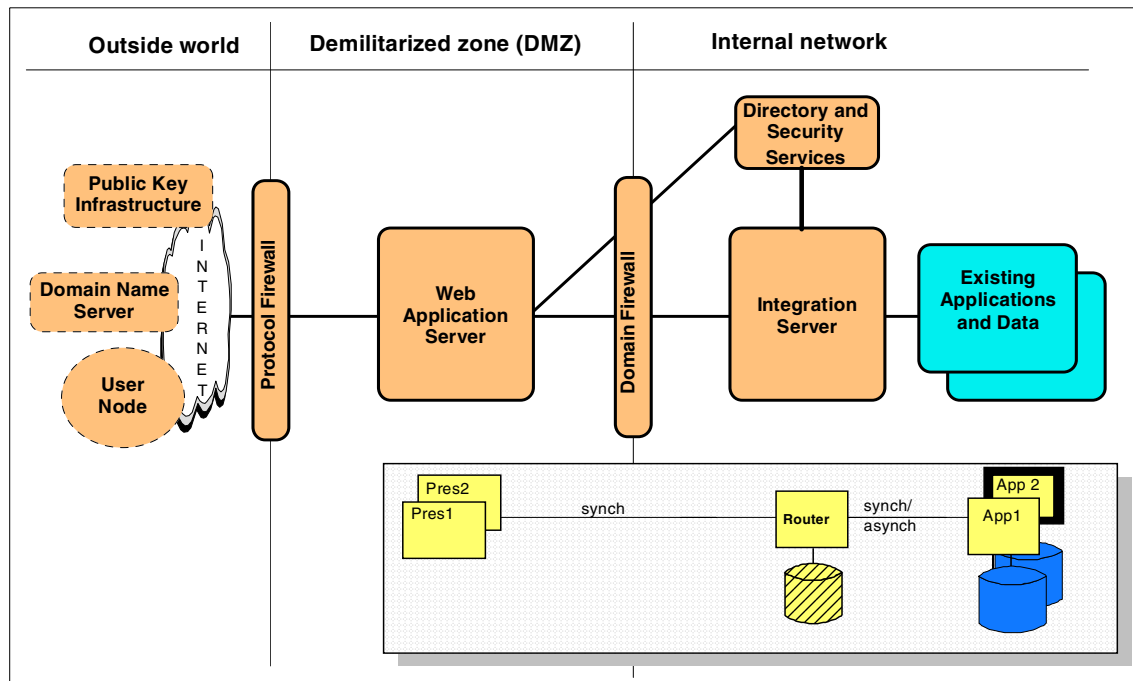


Figure 4. Runtime topology A - basic

The Web application server contains the presentation and controller logic. This includes the HTML pages and JSPs required for interaction with the users, and the controlling servlets required to access the back-end applications. The primary business logic resides in the back-end applications, with just enough logic in the integration server to route the request to the appropriate destination.

The Web application server builds a request based on user input and passes it to the integration server. The integration server examines the request, determines the appropriate destination, and forwards it to the chosen back-end application. This may involve making minor changes to the request.

Access to the Web application server resources is protected by the Web application server's security features, while access to the integration server's resources is protected by the integration server's security features. User information that is needed for authentication and authorization by both servers is stored in the directory and security services node behind the domain firewall in the internal network.

3.2.0.1 Benefits and limitations

This runtime topology offers the following benefits:

- All sensitive persistent data is stored behind the DMZ.
- Business logic can completely reside in the secure network.

Although this topology has limited availability and failover capability, individual products offer features that could be folded into this topology, allowing duplicate servers that can take over in the event of a server failure. Horizontal scalability is also not shown, but there again, individual products may offer workload management features that could be folded into this topology to allow duplicate servers for distributed workload. Vertical scalability can be achieved by adding memory or processors, and/or creating multiple servers on the Web application server and integration server.

The number of clients that access the Web server simultaneously is limited by the capacity of the Web server. The actual numbers depend on the software and hardware platform used. Load balancing among the Web servers is a possibility.

Since the Web server is not separated from the application server, there is no additional security available and the business logic in the Web application server is protected only by the protocol firewall.

3.3 Topology A variation 1

This variation introduces a new layer of security by putting all application logic behind the firewall. Only a portion of the presentation function is left in the DMZ.

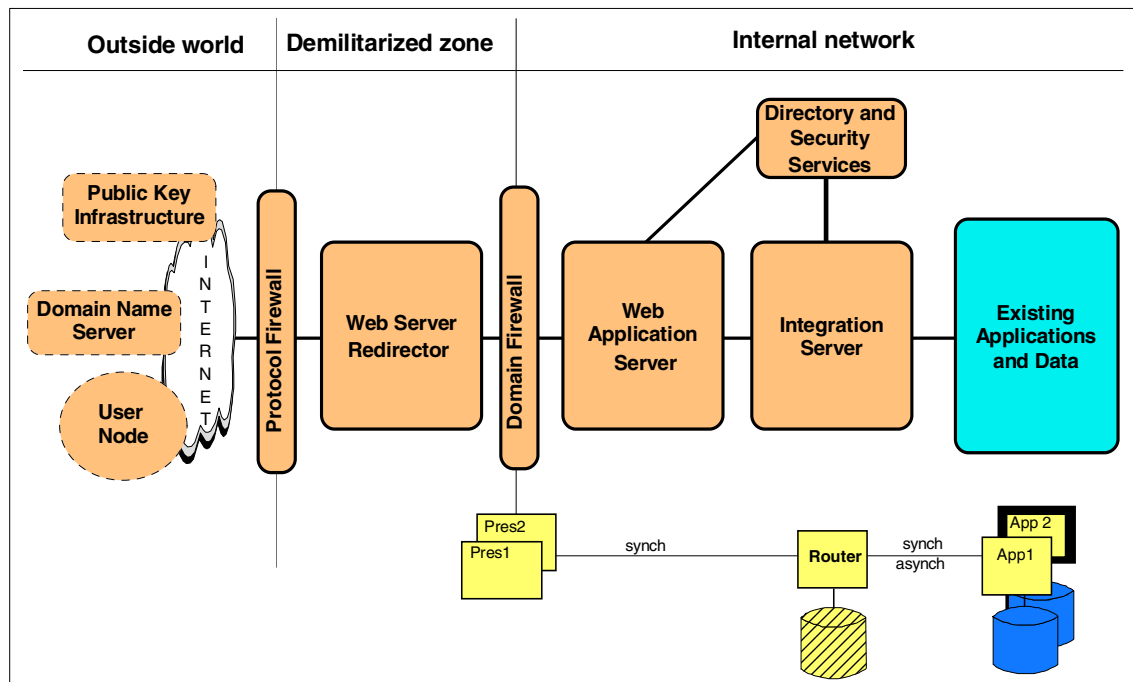


Figure 5. Runtime topology A - variation 1

If you remember from our definitions, a Web application server node is a combination of an HTTP server and an application server on one machine. To provide an even more secure environment than the basic topology provides, it is possible to separate the Web server function from the application server, creating two new nodes. This allows you to leave the HTTP server in the DMZ, but move the application server into the internal network, where it is in a secure environment.

This separation of the Web server from the application server is done by using a *Web server redirector* node (or redirector for short). A Web server redirector combines the HTTP server, which serves static HTTP pages, with a mechanism for forwarding dynamic servlet and JSP requests to a dedicated server.

The presentation logic will span across both the Web server and the application server. The primary business logic resides in the back-end applications, with just enough logic in the integration server to determine the appropriate destination for a request.

3.3.0.1 Benefits and limitations

Since the Web server is separated from the application server, additional security is available. All business logic and the bulk of the presentation logic is protected by both the protocol and the domain firewall.

As with the basic topology, there is limited availability, failover capability, and scalability. Individual products may offer features that could be folded into this topology, allowing duplicate servers and workload management features.

Since the requests to the application server need to be forwarded, you could see a performance degradation, depending on the redirector solution chosen.

Chapter 4. Product mapping

Once the runtime topology is chosen, you will be ready to determine what products and platforms will fit your needs. This chapter outlines our product recommendations.

4.1 Product mappings for the basic topology

These mappings show the products and platforms used in the implementation of the basic runtime topology A. The basic topology features a Web application server between two firewalls and an integration server in the secure network. Figure 6 shows the mapping for an environment where Windows NT is used as the platform of choice for the Web application server and integration server.

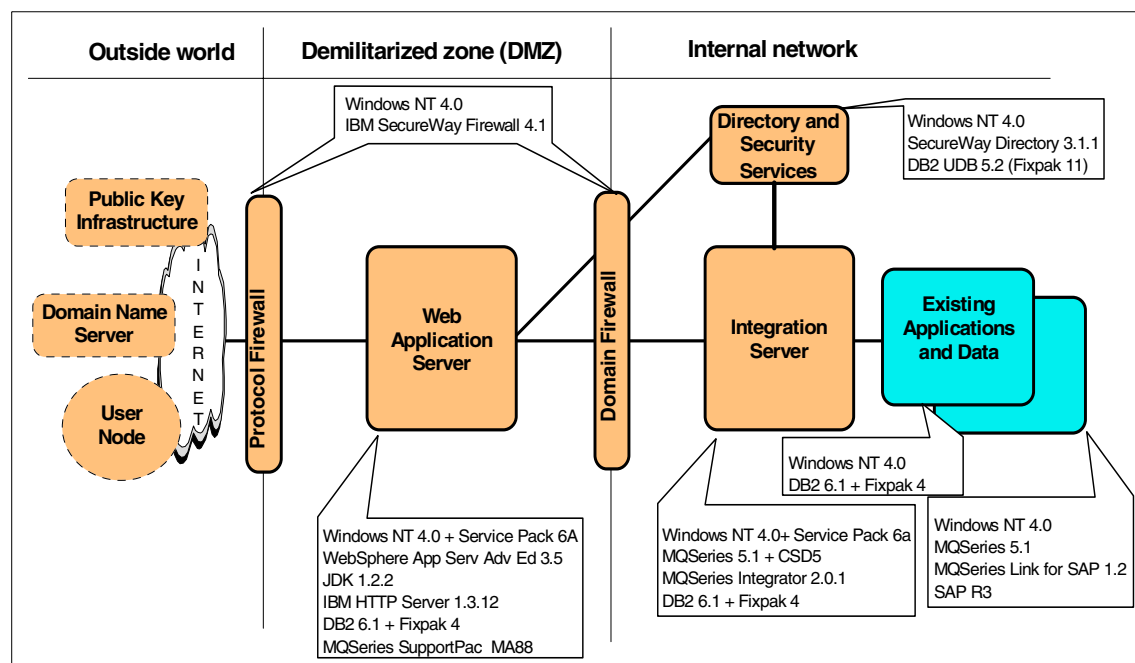


Figure 6. Basic runtime topology: Windows NT

IBM WebSphere Application Server Advanced Edition is used as the application server. Security for the Web applications is provided by WebSphere Advanced's security features. Users must authenticate using the LDAP server or the local operating system security.

The integration server is implemented using IBM's MQSeries and MQSeries Integrator. Both the data and the integration server are protected behind the domain firewall.

The back-end data and applications can be anything that supports a connection to MQSeries, for example:

- CICS
- IMS
- SAP R/3
- Lotus Notes
- PeopleSoft
- Baan

Network security is provided by the two firewalls. The firewall between the Internet and the DMZ is called the protocol firewall. It is configured to be open on port 80 only, thus allowing only traffic using the HTTP protocol to flow from the clients in the Internet to the Web application server in the DMZ.

The domain firewall restricts traffic based not only by protocol, but by host name. The domain firewall is configured to be open on one port to allow the Web application server to access the LDAP server, implemented with the IBM SecureWay Directory. The number of ports open to allow access to the integration server depends on the API method used to access MQSeries and whether you choose to use MQSeries client or server functions on the Web application server.

Our examples assume that an MQSeries client is used on the Web Application Server, putting messages directly on remote queues. For performance reasons (especially if you are not using persistent sessions), you may choose to have the messages put on a local queue in the Web application server for transmission by MQSeries to the remote broker. In this case the open ports needed on the firewall will be those needed to allow MQSeries to transmit traffic to another queue.

Figure 7 shows the mapping for an environment where AIX is used as the platform of choice for the Web application server and integration server. The products on the other nodes were implemented on Windows NT but are also available on AIX.

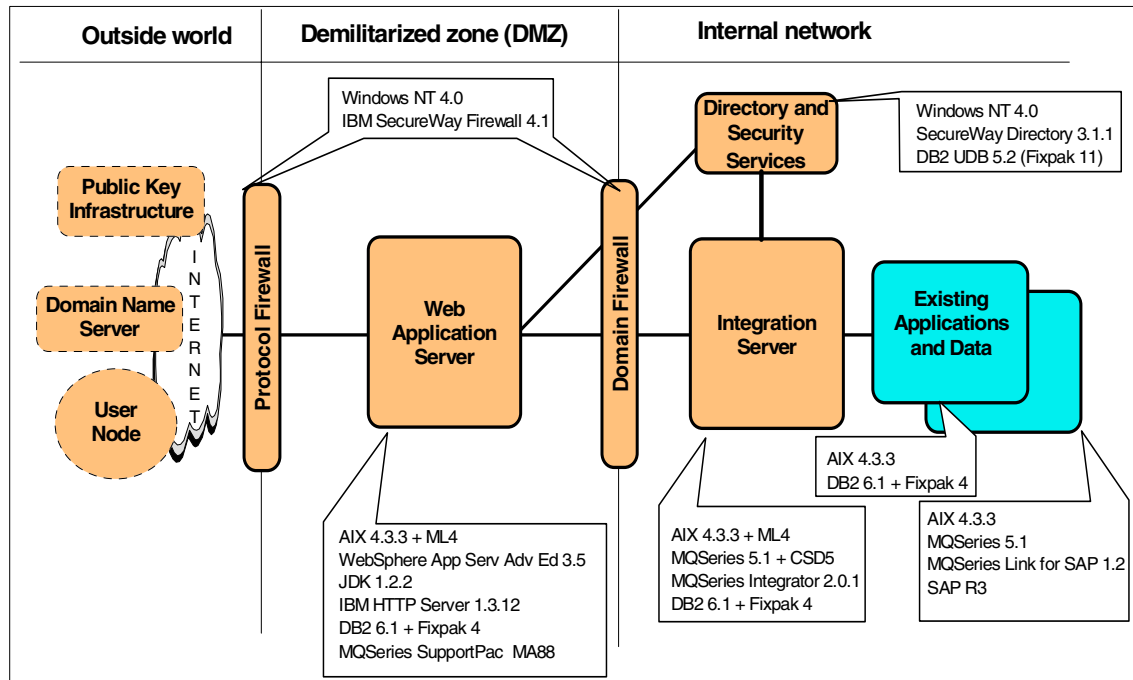


Figure 7. Basic runtime topology: AIX

4.2 Runtime topology variation 1

Variation 1 features separating the Web server functions from the application server functions. The application server and the integration server are placed in the secure network. Requests are routed from the Web server to the application server using a Web server redirector. Figure 8 shows the mapping for an environment where Windows NT is used as the platform of choice for the Web application server and integration server.

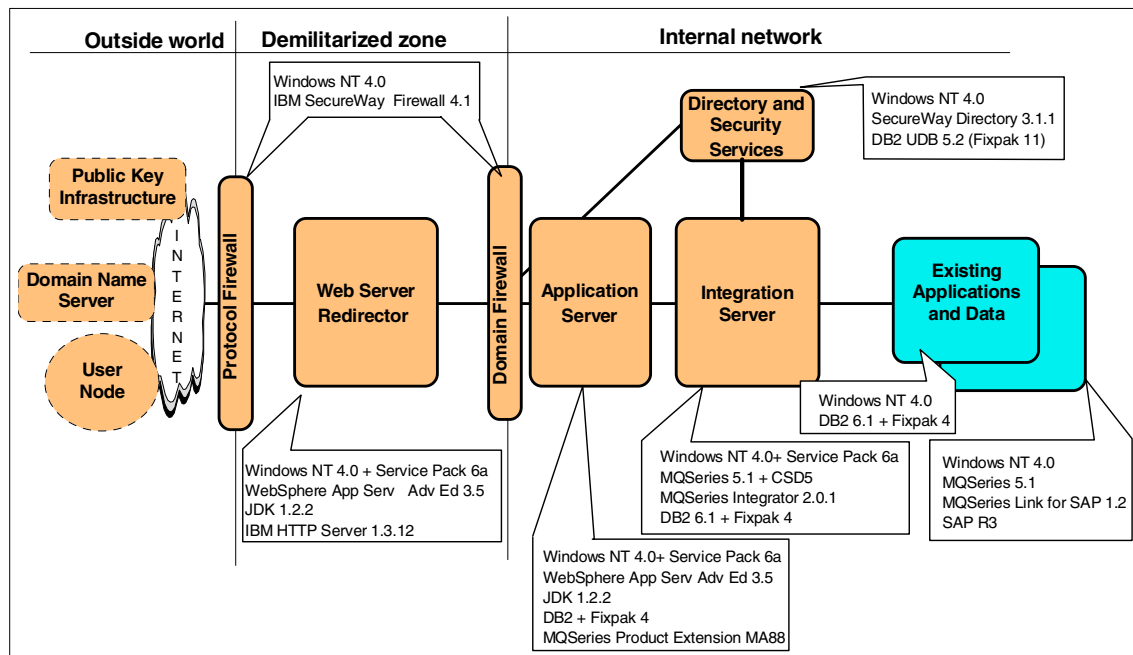


Figure 8. Runtime variation 1: Windows NT

In this product mapping the Web server redirector is implemented by the OSE Remote feature of WebSphere Advanced. More information about this feature can be found in *WebSphere's Remote OSE*, a redpaper available at <http://www.ibm.com/redbooks>.

As with the basic topology, IBM WebSphere Application Server Advanced Edition is used as the application server, which provides Web application security. The integration server is implemented using IBM MQSeries and MQSeries Integrator.

The protocol firewall is identical to that in the basic topology, though the requirements for the domain firewall have changed. A minimum of two ports need to be open on the domain firewall to allow traffic to flow from the redirector to the application server. This number would increase by one for each application server. Additional ports may be needed depending on the the method used to configure the redirector.

Figure 9 shows the mapping for an environment where AIX is used as the platform of choice for the Web application server and integration server. The products on the other nodes were implemented on Windows NT but are also available on AIX.

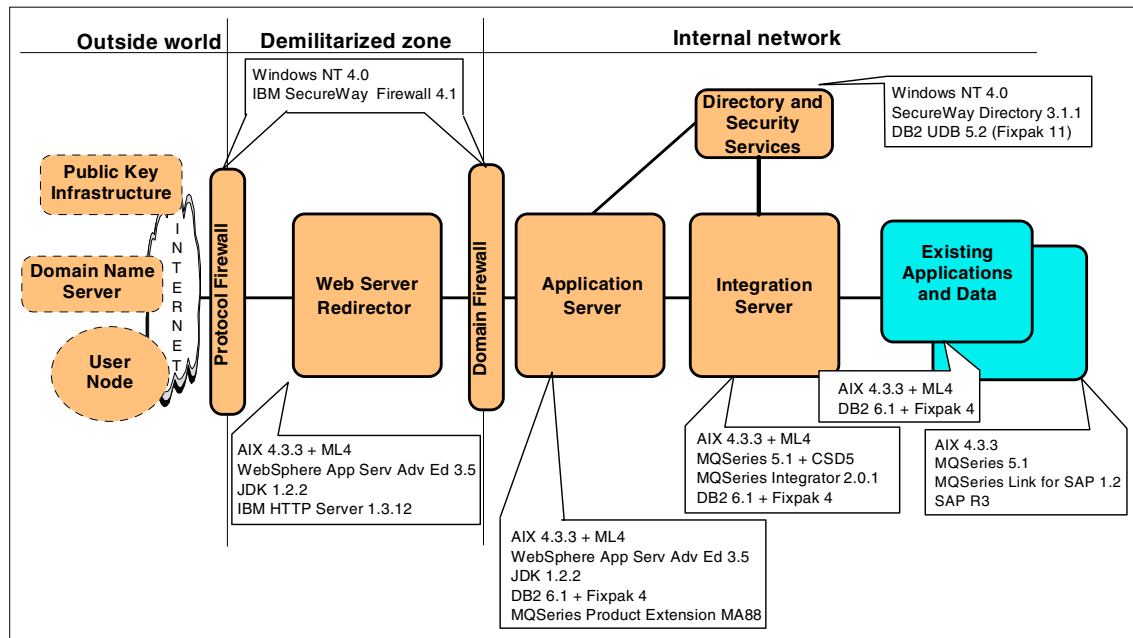


Figure 9. Runtime variation 1: AIX

4.3 Extending the topologies with workload management

The runtime topologies we have been working with do not show any provisions for workload management. The possibilities are largely dependent on the products used to implement the pattern. In our product mappings, we use IBM WebSphere Application Server Advanced Edition, IBM MQSeries, and IBM MQSeries Integrator. Each of these products offers workload management features.

4.3.1 MQSeries and MQSeries Integrator

MQSeries 5.1 has a feature called clustering, which allows you to reduce the complexity of an MQSeries network while reducing the administration overhead required to maintain it. Clustering also allows you to balance workload among queue managers on a round-robin basis. We talk more about MQSeries clustering in 10.1.4, “Overview of the MQSeries clustering feature” on page 249.

MQSI networks have one or more brokers that run the message flow logic, providing the routing, transformation, etc. functions. Workload management can be implemented by building your MQSI network on an MQSeries cluster and defining the application queues to be hosted as local queues on multiple brokers.

This is covered in more detail in *Business Integration Solutions with MQSeries Integrator*, SG24-6154.

4.3.2 WebSphere Advanced Edition

The IBM WebSphere Application Server allows you to leverage processor capacity or additional processors by means of application cloning. Cloning means that a given application is duplicated such that the clients cannot distinguish between the clones. In addition, each WebSphere Application Server (this is WebSphere's term for grouping a servlet engine and related resources) is running in its own JVM. This allows the use of more than one JVM with WebSphere.

Before cloning, you first have to create a model of that resource. Clones are created from a model. After you clone a resource, modifying the model automatically propagates the same changes to all of the clones. You can efficiently administer several copies of a server or other resource by administering its model.

You can also clone servlets, servlet engines, Web applications, EJB containers, and enterprise beans.

The application server balances the workload of the clones running on a server automatically. Thus, you do not have to worry about the machine's utilization. All of that is done automatically by the application server.

There are circumstances where cloning is desirable even if you have only one processor installed. This can be the case if you have an application that is spending most of its time waiting for some resources. During this time, additional requests can be served by the application's clones. Also, if automatic tasks such as garbage collection take too long to complete, cloning may prove a viable alternative on a single-processor machine.

In addition to all the above-mentioned advantages, cloning also increases the availability of a particular Web application as well as the failover capability. If any of the clones fails, the other clones take over the workload.

Overall, it is highly application dependent as to whether a performance improvement can be achieved by cloning of applications.

As with all alternatives, there are also disadvantages that you have to consider:

1. If you do not require session affinity and want all session-related information to be transparent to the users, there is some performance penalty because all session information needs to be saved and retrieved from a database. Depending on the amount of data, this penalty may prove to be very expensive.
2. If your application assumes that it is running on a dedicated machine (even on a dedicated JVM), cloning will not be an issue for you because it cannot be determined in advance on which machine your application will execute the next time a request is served.

See *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153 for more information on using clones in a WebSphere Advanced environment.

Part 2. User-to-Business Patterns: guidelines

Chapter 5. Technology options

This chapter looks at some of the technologies that you should consider for Web applications based upon the open standards and Java-based programming model of the IBM Application Framework for e-business. We will look at the technologies as they apply to both the client and the server side of the application. Some technologies, such as Java and XML, can apply to both. Also, the selection of client-side technologies used in your design will require consideration for the server side such as whether to store, or dynamically create, elements for the client side.

For an outline of the technologies recommended by the IBM Application Framework, see the *IBM Application Framework for e-business Architecture Overview* Web page at:

http://www.ibm.com/software/ebusiness/arch_overview.html

The sections that follow detail a number of technologies that you will want to consider in your design.

We recommend the following as technologies that are central to the Application Framework and its programming model:

- HTML
- Java servlets and JavaServer Pages
- XML
- Connectors
- Enterprise JavaBeans
- JDBC
- Additional enterprise Java APIs

These technologies are used in the context of the following logical model for an e-business application. This model, which has similarities to the Model-View-Controller approach in GUI development, characterizes the presentation logic as consisting of interaction control (implemented by Java servlets) and page construction (implemented by JavaServer Pages). The business logic may be implemented using beans and/or enterprise beans, depending on the transactional characteristics of the application. The business logic may need to access external resources using the appropriate connector technology.

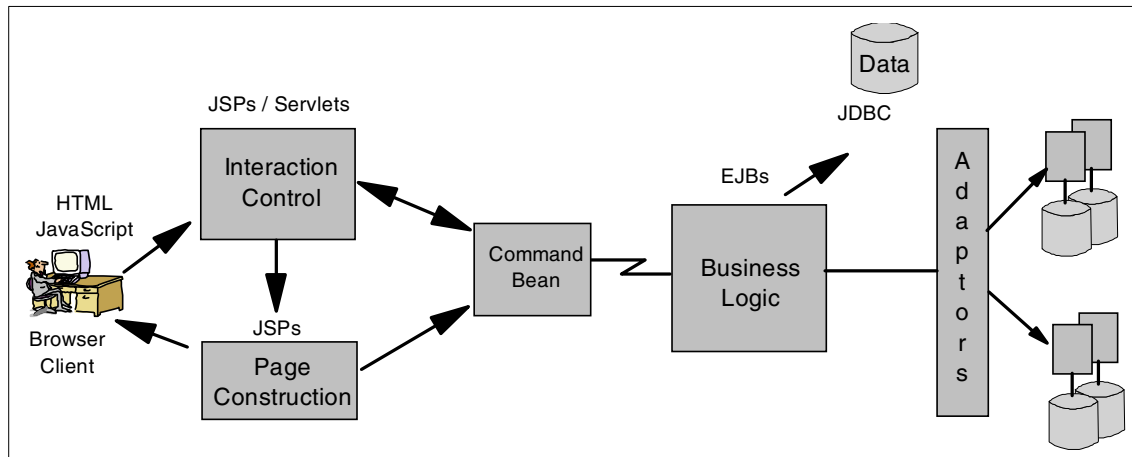


Figure 10. The logical structure of an e-business application using the recommended core technologies

We also include some discussion of the following technologies and the limitations involved in their usage:

- DHTML
- JavaScript
- Java applets

For more information, see the *IBM Application Framework for e-business Architecture Overview: Understanding Technology Choices* Web page, upon which significant portions of this chapter are based:

<http://www.ibm.com/software/ebusiness/buildapps/understand.html>

5.1 Web client

The Application Framework recommends the following technology model for a Web client.

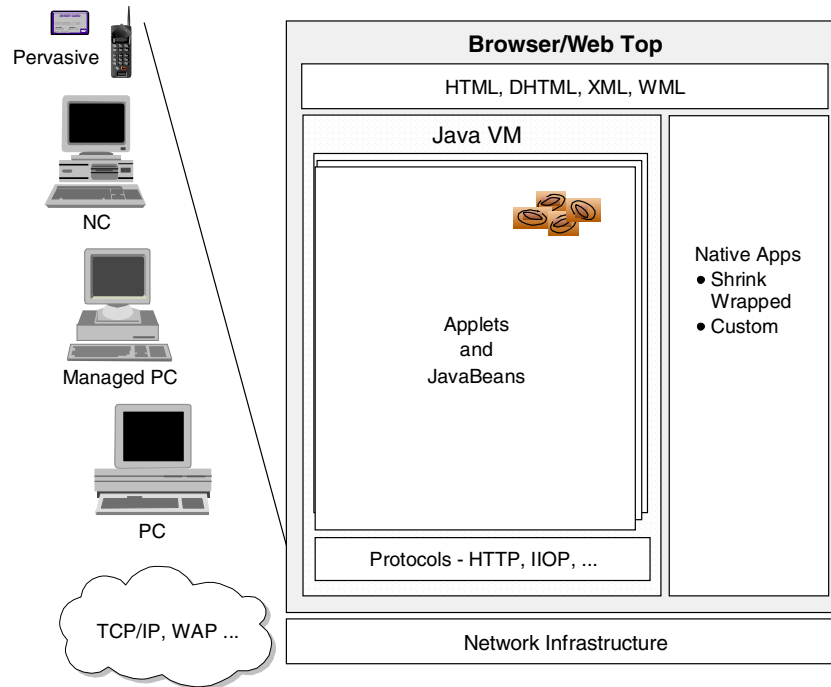


Figure 11. Web client technology model

The clients are “thin clients” with little or no application logic. Applications are managed on the server and downloaded to the requesting clients. The client portions of the applications should be implemented in HTML, dynamic HTML (DHTML), XML, and Java applets.

The following sections outline some of the possible technologies that you should consider, but remember that your choices may be constrained by the policy of your customer or sponsor. For example, for security reasons, only HTML is allowed on the Web client at some government agencies.

Although the Application Framework recommends thin clients, there is also opportunity for developing applications geared toward other client types.

5.1.1 Web browser

A Web browser is a fundamental component of the Web client. For PC-based clients, the browser typically incorporates support for HTML, DHTML, JavaScript, and Java. Some browsers are beginning to add support for XML as well. Under user control, there is a whole range of additional technologies

that can be configured as “plug-ins”, such as RealPlayer from RealNetworks or Macromedia Flash.

As an application designer you must consider the level of technology you can assume will be available in the user’s browser, or you can add logic to your application to enable slight modifications based upon the browser level. Regarding plug-ins, you need to consider what portion of your intended user community will have that capability.

For an e-business application that is to be accessed by the broadest set of users with varying browser capabilities, the client is often written in HTML with no other technologies. On an exception basis, limited use of other technologies, such as using JavaScript for simple edit checks, can then be considered based on the value to the user and the policy of the organization for whom the project is being developed.

The emergence of pervasive devices introduces new considerations to your design with regard to the content streams that the device can render and the more limited capabilities of the browser. For example, Wireless Application Protocol (WAP) enabled devices render content sent in Wireless Markup Language (WML).

5.1.2 HTML

HTML is a document markup language with support for hyperlinks, that is rendered by the browser. It includes tags for simple form controls. Many e-business applications are assembled strictly using HTML. This has the advantage that the client-side Web application can be a simple HTML browser, enabling a less-capable client to execute an e-business application.

The HTML specification defines user interface (UI) elements for text with various fonts and colors, lists, tables, images, and forms (text fields, buttons, checkboxes, and radio buttons). These elements are adequate to display the user interface for most applications. The disadvantage, however, is that these elements have a generic look and feel, and they lack customization. As a result, some e-business application developers augment HTML with other user interface technologies to enhance the visual experience, subject to maintaining access by the intended user base and compliance with company policy on Web client technologies.

Because most Web browsers can display HTML Version 3.2, this is the lowest common denominator for building the client side of an application.

5.1.3 Dynamic HTML (DHTML)

DHTML allows a high degree of flexibility in designing and displaying a user interface. In particular, DHTML includes cascading style sheets (CSS) that enable different fonts, margins, and line spacing for various parts of the display to be created. These elements can be accurately positioned using absolute coordinates.

Another advantage of DHTML is that it increases the level of functionality of an HTML page through a document object model and event model. The document object enables scripting languages such as JavaScript to control parts of the HTML page. For example, text and images can be moved about the window, and hidden or shown, under the command of a script. Also, scripting can be used to change the color or image of a link when the mouse is moved over it, or to validate a text input field of a form without having to send it to the server.

Unfortunately there are several disadvantages with using DHTML. The greatest of these is that two different implementations (Netscape and Microsoft) exist and are found only on the more recent browser versions. A small, basic set of functionality is common to both, but differences appear in most areas. The significant difference is that Microsoft allows the content of the HTML page to be modified by using either JScript or VBScript, while Netscape allows the content to be manipulated (moved, hidden, shown) only using JavaScript.

Because of browser compatibility issues, DHTML is not recommended in environments where mixed levels and brands of browsers are present.

5.1.4 XML (client-side)

XML allows you to specify your own markup language with tags specified in a Document Type Definitions (DTDs). Actual content streams are then produced that use this markup. The content streams can be transformed to other content streams by using XSL (eXtensible Stylesheet Language).

For PC-based browsers, HTML is well established for both document content and formatting. The leading browsers have significant investments in rendering engines based on HTML and a Document Object Model (DOM) based on HTML for manipulation by JavaScript.

XML seems to be evolving to a complementary role for active content within HTML documents for the PC browser environment.

For new devices, such as WAP-enabled phones and voice clients, the data content and formatting is being defined by new XML schema (DTD), WML for WAP phone, and VoiceXML for voice interfaces.

For most Web application designs, you should focus your attention on the use of XML on the server side. See 5.2.4, “XML” on page 45 for additional discussion of the server side use of XML.

5.1.5 JavaScript

JavaScript is a cross-platform object-oriented scripting language. It has great utility in Web applications because of the browser and document objects that the language supports. Client-side JavaScript provides the capability to interact with HTML forms. You can use JavaScript to validate user input on the client and help improve the performance of your Web application by reducing the number of requests that flow over the network to the server.

ECMA, a European standards body, has published a standard (ECMA-262) that is based on JavaScript (from Netscape) and JScript (from Microsoft) called ECMAScript. The ECMAScript standard defines a core set of objects for scripting in Web browsers. JavaScript and JScript implement a superset of ECMAScript. You can find the ECMAScript Language Specification at:

<http://www.ecma.ch>.

To address various client-side requirements, Netscape and Microsoft have extended their implementations of JavaScript in Version 1.2 by adding new browser objects. Because Netscape's and Microsoft's extensions are different from each other, any script that uses JavaScript 1.2 extensions must detect the browser being used, and select the correct statements to run.

The use of JavaScript on the server side of a Web application is not recommended, given the alternatives available with Java. Where your design indicates the value of using JavaScript, for example for simple edit checking, use JavaScript 1.1, which contains the core elements of the ECMAScript standard.

JavaScript: The Definitive Guide, Third Edition, by David Flanagan, is an excellent book on JavaScript that details the JavaScript objects and methods listing their origin and JavaScript level.

5.1.6 Java applets

The most flexible of the user interface (UI) technologies that can be run in a Web browser is offered by the Java applet. Java provides a rich set of UI elements that include an equivalent for each of the HTML UI elements. In

addition, because Java is a programming language, an infinite set of UI elements can be built and used. There are many widget libraries available that offer common UI elements, such as tables, scrolling text, spreadsheets, editors, graphs, charts, etc.

A Java applet is a program written in Java that is downloaded from the Web server and run on the Web browser. The applet to be run is specified in the HTML page using an APPLET tag:

```
<APPLET CODEBASE="/mydir" CODE="myapplet.class" width=400 height=100>
  <PARAM NAME="myParameter" VALUE="myValue">
</APPLET>
```

For this example, a Java applet called myapplet will run. An effective way to send data to an applet is with the use of the PARAM tag. The applet has access to this parameter data and can easily use it as input to the display logic.

Java can also request a new HTML page from the Web application server. This provides an equivalent function to the HTML FORM submit function. The advantage is that an applet can load a new HTML page based upon the obvious (a button being clicked), or the unique (the editing of a cell in a spreadsheet).

A characteristic of Java applets is that they seldom consist of just one class file. On the contrary, a large applet may reference hundreds of class files. Making a request for each of these class files individually can tax any server and also tax the network capacity. However, packaging all of these class files into one file reduces the number of requests from hundreds to just one. This optimization is available in many Web browsers in the form of either a JAR file or a CAB file. Netscape and HotJava support JAR files simply by adding an `ARCHIVE="myjarfile.jar"` variable within the APPLET tag. Internet Explorer uses CAB files specified as an applet parameter within the APPLET tag. In all cases, executing an applet contained within a JAR/CAB file exhibits faster load times than individual class files. While Netscape and Internet Explorer use different APPLET tags to identify the packaged class files, a single HTML page containing both tags can be created to support both browsers. Each browser simply ignores the other's tag.

A disadvantage of using Java applets for UI generation is that the required version of Java must be supported by the Web browser. Thus, when using Java, the UI part of the application will dictate which browsers can be used for the client-side application. Note that the leading browsers support variants of the JDK 1.1 level of Java and they have different security models for signed applets.

A second disadvantage of Java applets is that any classes such as widgets and business logic that are not included as part of the Java support in the browser must be loaded from the Web server as they are needed. If these additional classes are large, the initialization of the applet may take from seconds to minutes, depending upon the speed of the connection to the internet.

Because of the above shortcomings the use of Java applets is not recommended in Internet environments where mixed levels and brands of browsers are present. Small applets may be used in rare cases where HTML UI elements are insufficient to express the semantics of the client-side Web application user interface. If it is absolutely necessary to use an applet in an Internet environment, care should be taken to include UI elements that are core Java classes whenever possible. Applets work better in an intranet environment, where there is some expectation of consistent browser levels.

5.2 Web application server

The Application Framework recommends the following technology model for a Web application server.

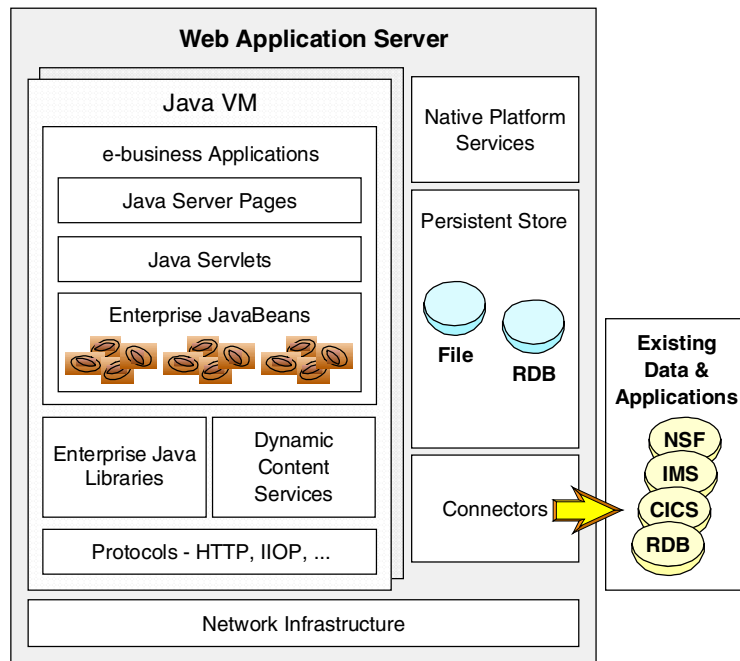


Figure 12. Web application server technology model

We will assume in this section that you will be using a Web application server and server-side Java. While there have been many other models for a Web application server, this is the one that is experiencing widespread industry adoption. For more details on the Java APIs discussed in this section see *Java Enterprise in a Nutshell* by David Flanagan, Jim Farley, William Crawford and Kris Magnusson.

Before looking at the technologies and APIs available in the Web application programming environment, let's have a word about two fundamental operational components on this node, the HTTP server and the Java Virtual Machine (JVM). For production applications, these essential components should be chosen for their operational characteristics in areas such as robustness, performance, and availability.

Relating the Model-View-Controller design structure so often used in user interfaces to the Web application programming model:

- The View is generally best implemented using JavaServer Pages.
- The Interaction Controller, which is primarily concerned with processing the HTTP request and invoking the correct business or UI logic, often lends itself to implementation as a servlet.
- The Model is represented to the View and Interaction Controller via a set of JavaBeans or Enterprise JavaBeans components.

5.2.1 Java servlets

Servlets provide a replacement for CGI-based techniques in Web programming. Servlets are small Java programs that run on the Web application server. They interact with the servlet engine running on the Web application server through HTTP requests and responses, which are encapsulated as objects in the servlet.

One of the attractions of using servlets is that the API is a very accessible one for a Java programmer to master. The most current level of the servlet API is 2.2. To learn more about the servlet API visit:

<http://www.javasoft.com/products/servlet/>.

Servlets are a core technology in the Web application programming model. They are the recommended choice for implementing the “Interaction Controller” classes that handle HTTP requests received from the Web client.

5.2.2 JavaServer Pages (JSP)

JSP files were designed to simplify the process of creating pages by separating Web presentation from Web content. In the page construction logic of a Web application, the response sent to the client is often a combination of template data and dynamically generated data. In this situation, it is much easier to work with JSP files than to do everything with servlets.

The chief advantage JSP files have over Java servlets is that they are closer to the presentation medium. A JavaServer Page is basically an HTML page. JSP files can contain all the HTML tags that Web authors are familiar with. A JSP may contain fragments of Java code that encapsulate the logic that generates the content for the page. These code fragments may call out to beans to access re-usable components and back-end data. To learn more about JSP files visit:

<http://www.javasoft.com/products/jsp/>.

JSP files are compiled into servlets before being executed on the Web application server. The most current level of the JSP API is 1.1.

JSP files are the recommended choice for implementing the “View” that is sent back to the Web client. For those cases where the code required on the page will be a large percentage of the page, and the HTML only a small percentage, writing a Java servlet will make the Java code much easier to read and therefore maintain.

5.2.3 JavaBeans

JavaBeans is an architecture developed by Sun Microsystems, Inc. describing an API and a set of conventions for re-usable, Java-based components. Code written to Sun’s JavaBeans architecture is called Java beans or just beans. One of the design criteria for the JavaBean API was support for builder tools that can compose solutions that incorporate beans. Beans may be visual or non-visual.

Beans are recommended for use in conjunction with servlets and JSP files in the following ways:

- As the client interface to the “Model” layer. An interaction controller servlet will use this bean interface.
- As the client interface to other resources. In some cases this may be generated for you by a tool.

- As a component that incorporates a number of property-value pairs for use by other components or classes. For example, the JavaServer Pages specification includes a set of tags for accessing JavaBean properties.

5.2.4 XML

XML and XSL style sheets can be used on the server side to encode content streams and parse them for different clients, thus enabling you to develop applications for both a range of PC browsers and for the emerging pervasive devices. The content is in XML and an XML parser is used to transform it to output streams based on XSL style sheets.

This general capability is known as transcoding and is not limited to XML-based technology. The appropriate design decision here is how much control over the content transforms you need in your application. You will want to consider when it is appropriate to use this dynamic content generation and when there are advantages to having servlets or JSP files specific to certain device types.

XML is also used as a means to specify the content of messages between servers, whether the two servers are within an enterprise or represent a business-to-business connection. The critical factor here is the agreement between parties on the message schema, which is specified as an XML DTD. An XML parser is used to extract specific content from the message stream. Your design will need to consider whether to use an event-based approach, for which the SAX API is appropriate, or to navigate the tree structure of the document using the DOM API.

For more detail on the use of XML in the server side of your Web applications, see *XML and Java: Developing Web Applications* by Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto.

5.2.5 JDBC

The business logic in a Web application will access information in a database for a database-centric scenario. JDBC is a Java API for database-independent connectivity. It provides a straightforward way to map SQL types to Java types. With JDBC, you can connect to your relational databases, and create and execute dynamic SQL statements in Java.

JDBC drivers are RDBMS specific, provided by the vendor, but implement the standard set of interfaces defined in the JDBC API. Given common schemas between two databases, an application can be switched between one and the other by changing the JDBC driver name and URL. A common practice is to

place the JDBC driver name and URL information in a property or configuration file.

There are four types of JDBC drivers from which you can choose, based on the characteristics of your application:

- Type 1: JDBC-ODBC bridge drivers. This type of driver, packaged with the JDK, requires an ODBC driver and was introduced to enable database access for Java developers in the absence of any other type of driver.
- Type 2: Native API Partly Java drivers. This type of driver uses the client API of the DBMS and requires the binaries for the database client software. This type of driver offers performance advantages but introduces native calls from the JVM.
- Type 3: Net-protocol All Java drivers. A generic network protocol is used with this type of driver. Portability is a major advantage of this type of driver, but it has the limitation that it requires intermediate middleware to convert the Net-protocol to the DBMS protocol.
- Type 4: Native-protocol All Java drivers. This type of driver is portable and uses the protocol of the DBMS. Type 3 and 4 drivers are well suited for applets that access a database server on an intranet, as they only require Java code to be downloaded.

An important technique used to enhance the scalability of Web applications is connection pooling, which may be provided by the application server. When application logic in a user session needs access to a database resource, rather than establishing and later dropping a new database connection, the code requests a connection from an established pool, returning it to the pool when no longer required.

The most recent level of the JDBC specification is 2.0, but many JDBC drivers you use will still implement 1.0.

5.2.6 Enterprise JavaBeans

“Enterprise JavaBeans” is Sun's trademarked term for their EJB architecture (or “component model”). When writing to the EJB specification you are developing “enterprise beans” (or, if you prefer, “EJB beans”).

The EJB framework specifies clearly the responsibilities of the EJB developer and the EJB container provider. The intent is that the “plumbing” required to implement transactions or database access can be implemented by the EJB container. The EJB developer specifies the required transactional and security characteristics of an EJB in a deployment descriptor (this is sometimes referred to as declarative programming). In a separate step, the

EJB is then deployed to the EJB container provided by the application server vendor of your choice.

There are two types of Enterprise JavaBeans:

- Session beans
- Entity beans

A typical session bean has the following characteristics:

- Executes on behalf of a single client.
- Can be transactional.
- Can update data in an underlying database.
- Is relatively short lived.
- Is destroyed when the EJB server is stopped. The client has to establish a new session bean to continue computation.
- Does not represent persistent data that should be stored in a database.
- Provides a scalable runtime environment to execute a large number of session beans concurrently.

A typical entity bean has the following characteristics:

- Represents data in a database.
- Can be transactional.
- Shared access from multiple users.
- Can be long lived (lives as long as the data in the database).
- Survives restarts of the EJB server. A restart is transparent to the client.
- Provides a scalable runtime environment for a large number of concurrently active entity objects.

Typically an entity bean is used for information that has to survive system restarts, while in session beans, the data is transient and does not survive when the client's browser is closed. For example, a shopping cart containing information that may be discarded uses a session bean, and an invoice issued after the purchase of the items is an entity bean.

An important design choice when implementing entity beans is whether to use Bean Managed Persistence (BMP), in which case you must code the JDBC logic, or Container Managed Persistence (CMP), where the database access logic is handled by the EJB container.

The business logic of a Web application often accesses data in a database. EJB entity beans are a convenient way to wrap the relational database layer in an object layer, hiding the complexity of database access. Because a single business task may involve accessing several tables in a database,

modeling rows in those tables with entity beans makes it easier for your application logic to manipulate the data.

The latest EJB specification is 1.1. The most significant changes from EJB 1.0 are the use of XML-based deployment descriptors and the need for vendors to implement entity bean support to claim EJB compliance.

To learn more about Enterprise JavaBeans, visit:
<http://www.javasoft.com/products/ejb/index.html>

5.3 Integration server

The Application Framework defines the application integration component as that which allows disparate applications to communicate with each other. In this case, we are going to be using the IBM MQSeries and MQSeries Integrator as the integration server and will base this discussion on those products.

5.3.1 Connectors

e-business connectors are gateway products that enable you to access enterprise and legacy applications and data from your Web application. Connector products provide Java interfaces for accessing database, data communications, messaging and distributed file system services.

IBM provides a significant set of e-business connectors with tool support, for CICS, Encina, IMS, MQSeries, SAP R/3, Lotus Domino, DB2 and other relational databases. IBM is basing its tool support for most of these connectors on a Common Connector Framework (CCF). For resources on System/390, IBM is delivering native connectors based on CCF. The command bean model of the CCF allows you to code to the specific connector interface(s) of your choice while hiding the connector logic from the rest of the Web application.

5.3.1.1 Common Connector Framework (CCF)

The task of connecting an application to a back-end data store is relatively standard and follows the same basic pattern whether you are considering the interactions between applications, servlets, EJBs, message queueing systems, relational databases, transactional systems or some other pieces of enterprise infrastructure. The typical approach to integration has been to hand-craft the code required to drive each component of a system.

IBM's Common Connector Framework (CCF) recognizes that most interactions follow a standard pattern and provides a standard Java-based

infrastructure for integrating various system components together. The CCF is a framework made up of client and server interfaces. IBM provides an implementation of the CCF in VisualAge for Java.

The CCF simplifies enterprise connectivity development by providing:

- A common client programming model for connectors that greatly reduces the learning curve for an application developer
- A common infrastructure programming model for connectors
- A plug-in interface for higher-level tools, making them independent of a particular connector

VisualAge for Java provides the Enterprise Access Builder (EAB) tool, allowing you to create an EAB command that hides the complexities of enterprise connectivity. The programmer can easily use this EAB command as a bean that provides enterprise access functionality in the same manner for all the connectors.

Connectors currently exist for:

- CICS—both External Call Interface (ECI) and External Presentation Interface (EPI) modes
- IMS
- MQSeries
- Host On-Demand
- Encina DE-Light
- SAP R/3

CCF connectors use three main interfaces:

1. **ConnectionSpec:** A ConnectionSpec implementation uniquely identifies a connection and holds all connection-relevant attributes of a CCF Connector such as host name, port number, and timeout specifications. It may also encapsulate connector-specific connection data. For example, an MQSeries connector would require a channel identifier.
2. **InteractionSpec:** An InteractionSpec retains all interaction-relevant attributes of a CCF connector such as a program name argument to a CCF Connector Communication when a particular interaction has to be carried out.
3. **Communication:** An implementation of Communication “drives” a particular interaction along via its execute method. Three arguments are passed to the execute method to carry out an interaction via a Communication. An instance of an InteractionSpec must be provided to

identify the concrete interaction characteristics. The other arguments are an input and an output record that are used to carry the exchanged data.

In addition, it is necessary to specify input/output records to allow data to flow through the framework. These represent the parameters to, and data returned from, the addressed Enterprise Information System (EIS). They may be a simple byte array but are usually a Java bean. Each bean is either based on the Java Record Library (a number of helper classes in the `com.ibm.record`, `com.ibm.record.ctypes`, and `com.ibm.record.util` packages intended to facilitate working with both fixed-length and variable-length records), or is defined specifically for a particular connector.

Applying the connector is relatively straightforward, even if performed by hand, but the preferred way of using the CCF connectors is through the integrated tools.

The CCF is easily applied within a feature-rich component environment such as that provided by WebSphere Application Server to support servlets or enterprise beans. However, it is not restricted to such environments and can be readily applied within applets or “fat” client applications. In these situations, it is necessary for the application to directly supply an implementation of the CCF runtime support infrastructure, since this is normally provided by the supporting component environment.

More information on CCF can be found in *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514.

5.3.1.2 The MQSeries Connector

VisualAge for Java MQSeries Common Connector Framework (CCF) Connector classes provide a higher-level Java interface that conforms to the IBM CCF. This interface simplifies some of the programming tasks associated with the MQSeries Client Classes for Java native programming interface and is consistent with the CCF interfaces implemented by other IBM connectors.

Programs written using the MQSeries CCF Connector classes can communicate with programs that use the standard MQSeries programming interface (the MQI), or with programs that use the MQSeries Client Classes for Java interface. The other applications can be executing on any of the systems to which MQSeries has been ported.

The connector does not make direct calls to the basic MQSeries interfaces. Instead, it makes use of the MQSeries classes for Java (MQ base Java) facility. MQ base Java allows a program written in Java to connect to MQSeries as an MQSeries client using TCP/IP, or directly to an MQSeries

server (bindings mode) using JNI to call directly into the underlying MQSeries queue manager API.

5.3.2 Message-oriented middleware

In this redbook, we shall be studying the role and use of message-oriented middleware (MOM) as the integration component of the topology.

First, we will look at the principles of MOM, then examine the products we will use to illustrate its role in the e-business pattern.

5.3.2.1 Technology principles

To understand the role of MOM and its component technologies, we must examine the problem for which it is the answer.

The need for Enterprise Application Integration (EAI) in a typical back-office distributed systems environment presents a problem that can be split into distinct parts:

- How to transport information between systems
- How to translate information so that the output of one system can be accepted and understood by another.

It is important to be clear about how these issues differ from each other.

Clearly the former requires that the network infrastructure be in place to support the transport. In addition, and in the absence of an MOM solution, applications must contain code that can open the right connections across the network and send or receive information over them. In a typical MOM solution, the *transport layer* handles this function on behalf of the application. A *transformation and routing* application is used where information output from an application must be changed in order to be of use to receiving applications.

Transport layer

To use the MOM transport layer, a sending application packages its information in the form of a *message*, labels it to an addressee and hands it to the transport layer for delivery to the intended recipient. The receiving application gets the messages by simply connecting to the transport layer with a request for its messages. The use of the transport layer enables (but does not require) an interface (*messaging*) to become an asynchronous operation between applications.

Each node forming part of the transport layer is configured with the necessary network connections and other definitions that enable it to either store

messages, or when possible, forward messages. In this context, we use the word *node* to describe a functional control point of the transport layer.

The transport layer encapsulates the business of message transportation, and potentially translation, between networking protocols and character encoding standards as messages are transported between hosts in a distributed network.

It also provides an Application Programming Interface (API) to enable applications to send and receive messages through conversation with the transport layer alone.

The transport layer does not, however, handle any message *transformation* that may be required to make output from one system acceptable to another.

Transformation and routing

Where the output from one system is required in a different form by another system, message transformation is required. Transformation is handled by a specialist application, which gets messages from the transport layer, applies the required transformations, then hands the transformed message back to the transport layer for delivery to the final recipient.

Where transformation is required, the sending application sends its message to the transformation application, which adjusts the message as required and reroutes it to the intended location. The content of a single incoming message may be selectively extracted and routed in differing forms to many destinations.

From this basic concept, the latest specialist applications of this genre have extended the messaging paradigm to provide rich functionality beyond simple message transformation and routing. Messages can be distributed not only in a point-to-point fashion, as in the scenario described so far, but also by “publication and subscription” style through the services of a *broker*.

Message broker services

A message broker, as the name suggests, is a service that holds messages that are *published* to given *topics* so that client applications may obtain these upon *subscription*. This method of message distribution further alleviates an application from the responsibilities associated with exchange of data between it and other applications.

Message enrichment and transaction coordination

An added feature of many modern transformation applications is the ability to perform database operations as part of the message transformation and routing process.

Such database operations can be used to look up and add additional data to a message before onward delivery is made.

More significantly, data from the message can be used to *update* the content of a database as part of the process. This inevitably leads to the need for transactional capability, where all related transformations, routing, and database operations (collectively termed a *message flow*) may be coordinated as a single unit-of-work that may be committed if completely successful or rolled back completely if any part fails.

5.4 Additional enterprise Java APIs

In developing a server-side application, you may also need to be familiar with the following enterprise Java class libraries:

- Java Naming and Directory Interface (JNDI). This package provides a common API to a directory service. Service provider implementations include those for LDAP directories, RMI and CORBA object registries. Sample uses of JNDI include:
 - Accessing a user profile from an LDAP directory
 - Locating and accessing an EJB Home
- Remote Method Invocation (RMI). RMI and RMI over IIOP are part of the EJB specification as the access method for clients accessing EJB services. RMI can also be used to implement limited function Java servers.
- Java Message Service (JMS). The JMS API enables a Java programmer to access message-oriented middle ware such as MQSeries from the Java programming model. Such messaging middle-ware is a popular choice for accessing existing enterprise systems and is one of your options if you are implementing a solution based on application topology 2.
- Java Transaction API (JTA). This Java API for working with transaction services, is based on the XA standard. With the availability of EJB servers, you are less likely to use this API directly.

5.5 References and where to find more information

For more information on topics discussed in this chapter see:

- *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755

- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application*, SG24-5754
- *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514
- *Building e-business Solutions*, SC09-4432
- Flanagan, David, *JavaScript: The Definitive Guide*, Third Edition, O'Reilly & Associates, Inc., 1998
- Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto, *XML and Java: Developing Web Applications*, Addison-Wesley 1999
- Flanagan, David, Jim Farley, William Crawford and Kris Magnusson, *Java Enterprise in a Nutshell*, O'Reilly & Associates, Inc., 1999
- For information on the IBM Application Framework for e-business:
<http://www.ibm.com/software/ebusiness/>
- For information about the ECMAScript language specification:
<http://www.ecma.ch/>
- To learn more about Java technology:
<http://www.javasoft.com/products>

Chapter 6. Java application design: using commands and MQSeries

Designing an e-business application presents many design challenges. We have addressed many of these challenges in *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864. This updates the Command bean information presented in that book, with the new command package available with WebSphere 3.5.

This chapter will also address using MQSeries to exchange data between the Java application and the back-end systems.

As we progress through this chapter and the following guidelines, we will be discussing a WebSphere application example developed specifically for this book. The layout and function of the application, referred to as the WebBank application, is discussed in more detail in Chapter 8, "Application development guidelines" on page 141. For now, we will just be looking at coding techniques that will be used to accomplish specific tasks.

6.1 Command framework

This section discusses the motivation for using the command model in e-business applications. For a full discussion of the WebSphere command framework see *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application*, SG24-5754. More information on the command pattern is available in *Design Patterns - Elements of Reusable Object Oriented Software*.

6.1.1 What are commands?

The command model addresses the need in software to perform an operation without having to understand how or where that operation is completed. It makes it possible to serialize data to ship and cache it. There are also patterns that add compensation, allowing a second command to undo the actions of the first command.

Commands are stylized Java beans that are used to represent requests or actions. They allow us to keep the implementation of business actions, such as querying a database or running a transaction, separate from our servlet code. This allows servlet code to act solely as a controller in the Model-View-Controller paradigm.

In their most basic form, commands simply encapsulate some request for information or action. Commands are particularly useful for significant

program boundaries, such as the boundary between presentation code and business logic.

To use commands you must perform the following steps:

1. Create (instantiate).
2. Initialize by setting some of the command's properties. This may be done in one of the command's constructors (all commands must have a no-argument constructor to comply with the JavaBeans standard, but they may also have convenience constructors that take initial values for the command's properties), by calling some of the command's set methods, or by a combination of two mechanisms.
3. Call the command's `execute()` method. This is a no-argument method that makes the command's output properties ready for access.

And optionally:

4. Inspect the command's output properties by calling get methods.
Depending on how you implement your commands, some operations may not require any output parameters.

Commands can be reset so they can be reused, minimizing the number of creates.

Beans, of which commands are a special case, are the favored component for many visual tools. For example, WebSphere Studio provides visual support for accessing beans within a JavaServer Page (JSP) file. In fact, JSPs provide special support for beans that allows them to be used via special tags without requiring any code.

Commands use a "set property"/"get property" model for arguments, which scales better to complex requests than the list of parameters model. In the set property model each of the arguments has a different name so ordering is not critical. The set-property model handles optional arguments easily by providing default values for properties. In the list of parameters model the caller must pass in a special illegal value, such as null or -1, for optional parameters.

The get property model for obtaining the results easily accommodates multiple result values, whereas the single result model of method or procedure invocation is limited to a single value unless output parameters (which are not Java's strong suit) are used.

6.1.2 The command package

With the release of WebSphere 3.5 the command model is formalized in the command package (`com.ibm.websphere.command`) and extended to accommodate command shipping (called `TargetableCommands`). The concept behind command shipping is to intercept the `execute` method, ship the command to a better execution point (say on a remote server), execute it there and then ship it back to the caller.

The command package is available to any WebSphere Java application. For example, you can implement command shipping by using an entity bean. When `execute()` is called on a command, `performExecute(TargetableCommand targetableCommand)` is called on the entity bean. Or for example, you could also implement command shipping by using a “catcher servlet”. In this approach, the `CommandTarget` class would construct and send an `HttpServletRequest` to a servlet on the EJB server. The servlet would retrieve the command from the request, execute it, then store the executed command in the `HttpServletResponse` object for return to the `CommandTarget`.

Both these approaches are valid. The choice of which to use may depend on whether or not you want your servlet and EJB environments completely separate. Security is also a factor. The EJB approach transports over IIOP, which may present a problem in environments with strict firewall rules. The servlet catcher uses HTTP for the protocol, but in some environments Internet protocols are not allowed to pass through the firewall between the presentation layer to the application layer.

The command shipping model has several compelling advantages:

1. It is a direct extension of the base command model and therefore maintains the same programming style and tooling advantages.
2. It isolates application logic from communication protocols and routing policies. This allows the best protocol to be selected without requiring extensive application changes. Indeed, using techniques such as dynamic class loading, new protocols can be supported “on-the-fly” without the need to change or recompile existing code.
3. It supports an agent-oriented service definition model in which the service provider provides a functional interface without consideration for distribution overhead. The service client then defines commands based on the service interface. The commands are shipped to the service and run there. This allows the service client to control the granularity of remote communication and avoids many of the performance and complexity issues associated with remote interfaces.

4. In an EJB environment, command shipping allows multiple EJB calls to be made without the need for multiple round trips to the EJB server. All calls are made locally by the command server.

And some disadvantages.

1. The simplest implementation of command shipping uses Java serialization to generate the messages that flow between servers. This may hamper the use of a messaging infrastructure such as MQSI, since it is difficult to interpret a serialized bean at an intermediate point. However, command shipping does not dictate an encoding for requests. It is perfectly reasonable to provide a command target that encodes the command in XML, or even SOAP for transport. Currently this will require per-command encoding logic, but this logic would be required on a per-request basis with any approach.
2. The simplest implementation of command shipping uses the same class for both the server-side and the client-side implementation of the command. Thus, if the server-side implementation of the execute method needed to be changed, it would be necessary to redeploy the command class to all clients as well, for example, by including the command classes in a deployed EJB JAR file. Given the goal of agent-oriented service definitions, this does not seem like a serious issue. However, if this is a concern, then a simple dynamic delegation pattern can be followed where the execute method of a command is implemented by delegation to a dynamically linked (via `classForName`) helper class. In this way the helper class can be changed at any time with no impact on the client code.

6.1.3 Command caching

Command caching is beyond the scope of this book. However, it warrants a brief discussion to show future direction and to further justify the use of the command model.

Command caching extends the command model to allow executed commands to be saved in a cache and then retrieved when they are needed, thus avoiding the cost of re-executing the command. To do this, commands are extended with IDs and other metadata such as dependencies. The usage model for cacheable commands is exactly like that for non-cacheable commands. However, when `execute()` is called on the command the caching infrastructure checks to see if a command with exactly the same ID is already in the cache. If it is, then the contents of the cached command are copied into the newly executed command using the `setOutputProperties()` method added by `TargetableCommand`. `Execute()` then simply returns without really executing the command.

The advantages of command caching are:

1. Caching is transparent to application code.
2. It is a true caching model. The application works correctly if items are not in the cache. Just as an application using a command does not know or care *how* the action is carried out, an application using a caching command does not know or care if the action is carried out. It interacts with the command in the same way, regardless of implementation.
3. It provides a unified caching model. The model is the same for expensive computations, remote requests, database queries, etc.
4. A consistent caching model helps to contain the complexity of invalidation.

The disadvantages of command caching are:

1. It mixes logic and data solution by using data objects for output properties of commands.
2. It requires the application developer to implement invalidation logic; however:
 - Time-outs work very well for most non-user specific commands.
 - There is a special pattern for user-specific commands that keeps things quite simple.
 - A consistent, regular framework is better than ad hoc caching, which is the only real alternative.

6.1.4 Command classes

The complete command hierarchy is shown in Figure 13. This shows the Command interface as the base for all commands. Each command has to implement at least the Command interface. When using the base Command interface, the command is executed locally in the same JVM as the calling servlet. An application that requires a command to be executed remotely (a shippable command) needs to implement the TargetableCommand and TargetableCommandImpl interfaces. Finally, if a command is to undo the work done by another command then it must implement the CompensableCommand interface.

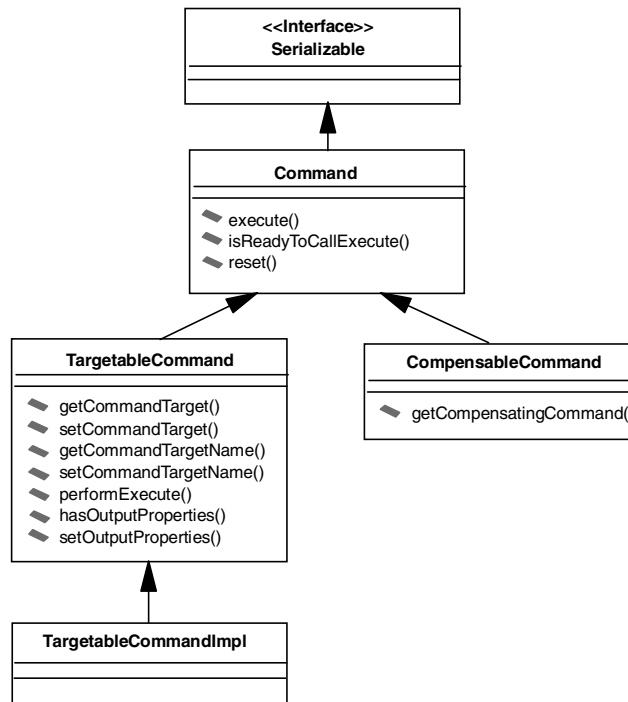


Figure 13. Command hierarchy

6.1.5 Command shipping example

The example addressed in this section shows the implementation of a shippable command for a client and server target. This example is used here only as a guide for the command shipping section. It is not used in the Personal Profile Services WebBank solution described at the end of this chapter and in Chapter 8, “Application development guidelines” on page 141.

This example uses an entity bean with Container Managed Persistence (CMP) called `CheckingAccountBean`. It allows a client to deposit money, withdraw money, set a balance, get a balance, and retrieve the name on the account.

In order to write a targetable command there are three steps involved:

1. Create the command interface
2. Implement the command
3. Implement the target

To write a command interface, it is necessary to extend one of the three interfaces included in the command package. The base interface for all commands is the `Command` interface. This provides the client-side interface for generic commands and declares the three following basic methods:

- `isReadyToCallExecute()` - This method is called on the client side before the command is passed to the server for execution.
- `execute()` - This method passes the command to the target and returns any data.
- `reset()` - This method reverts any output properties to the values they had before the `execute` method was called so that the object can be reused.

The implementation classes for the interface must contain implementations for the `isReadyToCallExecute()` and `reset()` methods. The `execute()` method is implemented for you elsewhere (see 6.1.5.2, “Implementing the command” on page 62).

Most commands, however, do not extend `Command` directly but use one or both of its extensions, the `TargetableCommand` interface and the `CompensableCommand` interface.

6.1.5.1 Creating the command interface

As mentioned above, in practice, a command interface is extended by the `TargetableCommand` interface and, if required, the `CompensableCommand` interface.

Figure 14 shows the most common implementation, which is that of a command interface for a targetable command.

```
import com.ibm.websphere.exception.*;
import com.ibm.websphere.command.*;
public interface AccountUpdateCmd extends TargetableCommand {
    float getAmount();
    float getBalance();
    float getOldBalance();
    float setBalance(float amount);
    float setBalance(int amount);
    CheckingAccount getCheckingAccount();
    void setCheckingAccount(CheckingAccount newCheckingAccount);
    TargetPolicy getCmdTargetPolicy
    .....
}
```

Figure 14. *AccountUpdateCmd* interface for a targetable command

In addition, the TargetableCommand interface declares the methods shown in Table 2.

Table 2. TargetableCommand methods

Method	Purpose
setCommandTarget()	Specify the target object of a command
getCommandTarget()	Returns the target object of a command
setCommandTargetName()	Specify the target name to a command
getCommandTargetName()	Returns the target name of a command
hasOutputProperties()	Indicates whether the command has output that must be copied back to the client (The implementation class also provides a method, setHasOutputProerties(), for setting the output of this method. By default, hasOutputProperties() returns true.
setOutputProperties()	Saves output values from the command for return to the specific client
performExecute()	Encapsulates the application specific work. It is called by the execute() method declared in the Command Interface.

The performExecute() method is the only method that must be implemented by the application developer. The remaining are implemented by the TargetableCommandImpl class, which also implements the execute() method declared in the Command interface.

6.1.5.2 Implementing the command

If a command is to be shipped, it must implement the TargetableCommand class by extending the TargetableCommandImpl class. This class implements all of the methods in the TargetableCommand interface except the performExecute() method. This method must be written by the application developer. It also implements the execute() method from the Command interface.

Continuing our example, the AccountUpdateCmdImpl must do the following:

- Define instance and class variables
- Implement Command specific methods
- Implement methods from the Command interface
- Implement methods from the TargetableCommand interface

Let's take a look at each of these items.

Defining instance and class variables

The AccountUpdateCmdImpl class declares the variables used by the methods in the class, including the remote interfaces for the CheckingAccount entity bean, the variables used to capture operations on the checking account, and a compensating command.

```
public class AccountUpdateCmdImpl extends TargetableCommandImpl
implements AccountUpdateCmd {

    // Define Variables
    public float balance;
    public float amount;
    public CheckingAccount checkingAccount;
    ....
}
```

Figure 15. Declare variables

Implementing command-specific methods

The AccountUpdateCmd interface defines several command-specific methods in addition to extending other interfaces in the command package. The command-specific methods are implemented in AccountUpdateCmdImpl.

The command developer must also provide a way of instantiating the command. The command package does not specify the mechanism, so there are number of options available to the developer. The fastest and most efficient is to use constructors; the most flexible is to use a factory; and since the commands are implemented as a JavaBeans component, the standard Beans.instantiate() method may also be used. The constructor also sets the Target policy (we cover this later in 6.1.5.3, "Implementing targets" on page 67).

Figure 16 shows two constructors used by this class. The difference between the two is that the first constructor uses the default target policy and the second allows a custom policy to be specified. They both take a CommandTarget object as an argument and cast it to the CheckingAccount type. The CheckingAccount interface extends both the CommandTarget interface and the EJBObject (see Figure 20 on page 69). The resulting checkingAccount object routes the command to the desired server using the bean's remote interface.

```

public class AccountUpdateCmdImpl extends TargetableCommandImpl
implements AccountUpdateCmd {
    // Define Variables
    .....
    // Constructors
    // First Constructor: relies on default target policy
    public AccountUpdateCmdImpl(CommandTarget target, float
newAmount)
    {
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    // Second Constructor: allows for a customized target policy
    public AccountUpdateCmdImpl(CommandTarget target, float
newAmount, TargetPolicy targetPolicy)
    {
        setTargetPolicy(targetPolicy);
        amount = newAmount;
        checkingAccount = (CheckingAccount)target;
        setCommandTarget(target);
    }
    .....
}

```

Figure 16. Constructors in the AccountUpdateCmdImpl class

In addition to the constructors, there are also business-specific methods that fall into this category. Figure 17 shows the following methods:

- **setBalance()** - Sets the balance of the account
- **getAmount()** - Returns the amount of the deposit or withdrawal.
- **getCmdTargetPolicy()** - Retrieves the current target policy
- **setCheckingAccount()** and **getCheckingAccount()** - Sets and retrieves the current checking account


```

public class AccountUpdateCmdImpl extends TargetableCommandImpl
implements AccountUpdateCmd {
    // Define Variables
    .....
    // Constructors
    .....
    // Business Methods
    public float getAmount() {
        return amount();
    }
    public float getBalance() {
        return balance;
    }
    public float getOldBalance() {
        return oldbalance;
    }
    public float setBalance(float amount) {
        balance = balance + amount;
        return balance;
    }
    public float setBalance(int amount) {
        balance += amount;
        return balance;
    }
    public TargetPolicy getCmdTargetPolicy() {
        return getTargetPolicy();
    }
    public void setCheckingAccount(CheckingAccount
newCheckingAccount) {
        if (checkingAccount == null) {
            checkingAccount = newCheckingAccount;
        }
        else
            System.out.println("Incorrect Checking Account (" +
                newCheckingAccount + ") specified");
    }
    public CheckingAccount getCheckingAccount() {
        return checkingAccount;
    }
}

```

Figure 17. Business-specific methods in the AccountUpdateCmdImpl class

The AccountUpdateCmd command is a stylized JavaBean, meaning that its input and output properties are managed using standard JavaBean

techniques such as “getter” and “setter” methods. The getter methods do not work until after the command’s execute method has been called.

Implementing methods from other interfaces

There are potentially methods from three other interfaces that need to be implemented.

Command interface

This declares two methods that require implementation, `isReadyToCallExecute()` and `reset()`.

```
{
    .....
    //Methods from the Command Interface
    public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void reset() {
        amount =0 ;
        balance = 0;
        checkingAccount = null;
        targetPolicy = new TargetPolicyDefault();
    }
    .....
}
```

Figure 18. Implementing methods from the Command interface

TargetableCommand Interface

This declares one method that needs to be implemented, `performExecute()`. It may also be appropriate to override the default implementation of `setOutputProperties()` since it does not save final, transient or static fields.

```

public class AccountUpdateCmdImpl extends TargetableCommandImpl
implements AccountUpdateCmd
{
    ...
    // Method from the TargetableCommand interface
    public void performExecute() throws Exception {
        CheckingAccount checkingAccount = getCheckingAccount();
        oldBalance = checkingAccount.getBalance();
        balance = oldBalance+amount;
        checkingAccount.setBalance(balance);
        setHasOutputProperties(true);
    }
    public void setOutputProperties(TargetableCommand fromCommand) {
        try {
            if (fromCommand != null) {
                AccountUpdateCmd accountUpdateCmd =
                    (AccountUpdateCmd) fromCommand;
                this.oldBalance = accountUpdateCmd.getOldBalance();
                this.balance = accountUpdateCmd.getBalance();
                this.checkingAccount =
                    accountUpdateCmd.getCheckingAccount();
                this.amount = accountUpdateCmd.getAmount();
            }
        }
        catch (Exception ex) {
            System.out.println("Error in setOutputProperties.");
        }
    }
    ...
}

```

Figure 19. Implementing methods from the TargetableCommand interface

6.1.5.3 Implementing targets

The object that is the target of a TargetableCommand must implement the CommandTarget interface. This object can be an actual server-side object, such as an entity bean, or it can be a client-side adapter for a server. The implementer of the CommandTarget interface is responsible for ensuring the proper execution of a command in the desired target server environment. This typically requires the following steps:

1. Copying the command to the target server by using a server-specific protocol.
2. Running the command in the server.

3. Copying the executed command from the target server to the client by using a server-specific protocol.

Common ways to implement the `CommandTarget` interface include:

- A local target, which runs in the client's JVM.
- A client-side adapter for a server.
- An enterprise bean (either a session bean or an entity bean). Figure 20 on page 69 shows the structure of the remote interface and enterprise bean class for an entity bean that implements the `CommandTarget` interface.

Since targetable commands can be run remotely in another JVM, the command package provides mechanisms for determining where to run the command. A target policy associates a command with a target and is specified through the `TargetPolicy` interface. You can design customized target policies by implementing this interface, or you can use the provided `TargetPolicyDefault` class. We will discuss this in 6.1.5.4, "Targets and target policies" on page 75.

The following two sections will show how to implement a server-side command target followed by a client-side adapter.

Writing a command target (server)

In order to accept commands the server must implement the single method of the `CommandTarget` interface, the `executeCommand()` method. This can be implemented in an enterprise bean (session or entity beans) or a servlet.

We shall concentrate on an entity bean for the remainder of this example. It is possible to write a target enterprise bean that forwards the commands to a specific server, such as another entity bean, in which case all commands directed to a specific target go through the target enterprise bean. Or we can write a target enterprise bean that does the work of the command locally.

In order to make an enterprise bean the target of a command, it is necessary to:

- Extend the `CommandTarget` interface when you define the bean's remote interface, which must also extend the `EJBObject` interface.
- Implement the `CommandTarget` interface when you implement the bean class, which must also implement either the `SessionBean` or `EntityBean` interface.

The target in this example is an enterprise bean called `CheckingAccountBean`. The bean's remote interface, `CheckingAccount`,

extends the `CommandTarget` interface in addition to the `EJBObject` interface. This is shown in Figure 20. The methods declared in the remote interface are independent of those used by the command. The `executeCommand()` is declared in neither the bean's home or remote interfaces.

```
...
import com.ibm.websphere.command.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface CheckingAccount extends CommandTarget, EJBObject
{
    float deposit (float amount) throws RemoteException;
    float deposit (int amount) throws RemoteException;
    String getAccountName() throws RemoteException;

    float getBalance() throws RemoteException;
    float setBalance(float amount) throws RemoteException;

    float withdrawal (float amount) throws RemoteException,
    Exception;
    float withdrawal (int amount) throws RemoteException,
    Exception;
    ...
}
```

Figure 20. The remote interface of the `CheckingAccount` entity bean also a command target

Figure 21 shows the `CheckingAccountBean` implementing the `EntityBean` interface as well as the `CommandTarget` interface. The class contains the business logic for the methods in the remote interface, the necessary life cycle methods (`ejbActivate`, `ejbStore`, etc.), and the `executeCommand()` declared by the `CommandTarget` interface. This method is the only command-specific code in the enterprise bean class. It attempts to run the `performExecute()` method on the command and throws a `CommandException` if an error occurs. If the `performExecute()` method runs successfully, the `executeCommand()` method uses the `hasOutputProperties()` method to determine if there are output properties that must be returned. If the command has output properties, the method returns the command object to the client.

```

...
public class CheckingAccountBean implements EntityBean, CommandTarget
{
    // Bean variables
    ...
    // Business methods from remote interface
    ...
    // Life-cycle methods for CMP entity beans
    ...
    // Method from the CommandTarget interface
    public TargetableCommand executeCommand(TargetableCommand
command)throws RemoteException, CommandException {
        try {
            command.performExecute();
        }
        catch (Exception ex) {
            if (ex instanceof RemoteException) {
                RemoteException remoteException = (RemoteException)ex;
                if (remoteException.detail != null) {
                    throw new CommandException(remoteException.detail);
                }
                throw new CommandException(ex);
            }
        }
        if (command.hasOutputProperties()) {
            return command;
        }
        return null;
    }
}

```

Figure 21. Implementing the *CommandTarget*

Writing a client-side adapter

Commands can be used with any Java application, but the means of sending the command from the client to the server varies. The application we have been describing uses enterprise beans. Now we will take a look at an example that shows how to send a command to a servlet over the HTTP protocol.

In this example, the client implements the *CommandTarget* interface locally. Figure 22 shows the structure of the client-side class; it implements the *CommandTarget* interface by implementing the *executeCommand* method.

```

...
import java.io.*;
import java.rmi.*;
import com.ibm.websphere.command.*;
public class ServletCommandTarget implements CommandTarget,
Serializable
{
    protected String hostName = "localhost";
    public static void main(String args[]) throws Exception
    {
        ....
    }
    public TargetableCommand executeCommand(TargetableCommand command)
        throws CommandException
    {
        ....
    }
    public static final byte[] serialize(Serializable serializable)
        throws IOException {
        ... }
    public String getHostName() {
        ... }
    public void setHostName(String hostName) {
        ... }
    private static void showHelp() {
        ... }
}

```

Figure 22. The structure of a client-side adapter for a target

The main method in the client-side adapter constructs and initializes the CommandTarget object, shown in Figure 23.

```

public static void main(String args[]) throws Exception
{
    String hostName = InetProfile.getLocalHost().getHostName();
    String fileName = "MyServletCommandTarget.ser";
    // Parse the command line
    ...
    // Create and initialize the client-side CommandTarget adapter
    ServletCommandTarget servletCommandTarget = new
ServletCommandTarget();
    servletCommandTarget.setHostName(hostName);
    ...
    // Flush and close output streams
    ... }
}

```

Figure 23. Instantiating the client-side adapter

Implementing a client-side adapter

The CommandTarget interface declares one method, `executeCommand()`, which the client implements. The `executeCommand()` method takes a `TargetableCommand` object as input and also returns a `TargetableCommand`. Figure 24 shows the implementation of the method used in the client-side adapter. This implementation does the following:

- Serializes the command it receives
- Creates an HTTP connection to the servlet
- Creates input and output streams, to handle the command as it is sent to the server and returned
- Places the command on the output stream
- Sends the command to the server
- Retrieves the returned command from the input stream
- Returns the returned command to the caller of the `executeCommand` method


```

public TargetableCommand executeCommand(TargetableCommand command)
    throws CommandException {
    try {
        // Serialize the command
        byte[] array = serialize(command);
        // Create a connection to the servlet
        URL url = new URL
            ("http://" + hostName +
            "/servlet/com.ibm.websphere.command.servlet.CommandServlet");
        HttpURLConnection httpURLConnection =
            (HttpURLConnection) url.openConnection();
        // Set the properties of the connection
        ...
        // Put the serialized command on the output stream
        OutputStream outputStream =
            httpURLConnection.getOutputStream();
        outputStream.write(array);
        // Create a return stream
        InputStream inputStream = httpURLConnection.getInputStream();
        // Send the command to the servlet
        httpURLConnection.connect();
        ObjectInputStream objectInputStream =
            new ObjectInputStream(inputStream);
        // Retrieve the command returned from the servlet
        Object object = objectInputStream.readObject();
        if (object instanceof CommandException) {
            throw ((CommandException) object);
        }
        // Pass the returned command back to the calling method
        return (TargetableCommand) object;
    }
    // Handle exceptions
    ....
}

```

Figure 24. A client-side implementation of the executeCommand() method

Running the command in the servlet

The servlet that runs the command is shown in Figure 25. The service method retrieves the command from the input stream and runs the performExecute() method on the command. The resulting object, with any output properties that must be returned to the client, is placed on the output stream and sent back to the client.

```

...
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.ibm.websphere.command.*;
public class CommandServlet extends HttpServlet
{
    ...
    public void service(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException
    {
        try {
            ...
            // Create input and output streams
            InputStream inputStream = request.getInputStream();
            OutputStream outputStream = response.getOutputStream();
            // Retrieve the command from the input stream
            ObjectInputStream objectInputStream =
                new ObjectInputStream(inputStream);
            TargetableCommand command = (TargetableCommand)
                objectInputStream.readObject();
            // Create the command for the return stream
            Object returnObject = command;
            // Try to run the command's performExecute method
            try {
                command.performExecute();
            }
            // Handle exceptions from the performExecute method
            ...
            // Return the command with any output properties
            ObjectOutputStream objectOutputStream =
                new ObjectOutputStream(outputStream);
            objectOutputStream.writeObject(returnObject);
            // Flush and close output streams
            ...
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

Figure 25. Running the command in the servlet

In this example, the target invokes the `performExecute()` method on the command, but this is not always necessary. In some applications, it can be preferable to implement the work of the command locally. For example, the command can be used to send input data to the target, which retrieves the data and runs a local database procedure based on the input. You must decide the appropriate way to use commands in your application.

6.1.5.4 Targets and target policies

As we have discussed, a targetable command extends the `TargetableCommand` interface, which allows the client to direct a command to a particular server. The `TargetableCommand` interface and the `TargetableCommandImpl` class provide two ways for a client to specify a target: the `setCommandTarget()` and `setCommandTargetName()` methods. The `setCommandTarget()` method allows the client to set the target object directly on the command. The `setCommandTargetName()` method allows the client to refer to the server by name, an approach that is useful when the client is not directly aware of server objects. A targetable command also has corresponding `getCommandTarget()` and `getCommandTargetName()` methods.

The command package needs to be able to identify the target of a command. Because there is more than one way to specify the target and because different applications can have different requirements, the command package does not specify a selection algorithm. Instead, it provides a `TargetPolicy` interface with one method, `getCommandTarget()`, and a default implementation. This allows applications to devise custom algorithms for determining the target of a command when appropriate.

The default target policy

The command package provides a default implementation of the `TargetPolicy` interface in the `TargetPolicyDefault` class. If you use this default implementation, the command determines the target by looking through an ordered sequence of four options:

1. The `CommandTarget` value
2. The `CommandTargetName` value
3. A registered mapping of a target for a specific command
4. A defined default target

If it finds no target, it returns null.

The `TargetPolicyDefault` class provides methods for managing the assignment of commands with targets (`registerCommand`,

unregisterCommand, and listMappings) and a method for setting a default name for the target (setDefaultTargetName). The default target name is com.ibm.websphere.command.LocalTarget, where LocalTarget is a class that runs the command's performExecute() method locally. Figure 26 shows the relevant variables and the methods in the TargetPolicyDefault class.

```
...
public class TargetPolicyDefault implements TargetPolicy, Serializable
{
    ...
    protected String defaultTargetName =
"com.ibm.websphere.command.LocalTarget";
    public CommandTarget getCommandTarget(TargetableCommand command) {
        ... }
    public Dictionary listMappings() {
        ... }
    public void registerCommand(String commandName, String targetName)
    {
        ... }
    public void unregisterCommand(String commandName) {
        ... }
    public void setDefaultTargetName(String defaultTargetName) {
        ... }
}
```

Figure 26. The TargetPolicyDefault class

Setting the command target

The AccountUpdateCmdImpl class shown in Figure 16 on page 64 provides two command constructors. The first constructor takes a command target as an argument and implicitly uses the default target policy to locate the target.

In Figure 32 on page 81 we see this constructor used with a null target, causing the default target policy to step through its choices and eventually find the default target name, LocalTarget.

The example in Figure 27 uses the same constructor to set the target explicitly. This differs from using a null target as follows:

- The command target is set to checkingAccount rather than null. The default target policy starts to step through its choices and finds the target in the first place it looks.

- It does not have to call the `setCheckingAccount` method to indicate the account on which the command should operate; the constructor uses the target variable as both the target and the account.

```

...
CheckingAccount checkingAccount
....
try {
    AccountUpdateCmd cmd =
        new AccountUpdateCmdImpl(checkingAccount, 1000);
    cmd.execute();
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
...

```

Figure 27. Identifying a target with CommandTarget

Setting the command target name

If a client needs to set the target of the command by name, it can use the command's `setCommandTargetName()` method. Figure 28 illustrates this technique. This example compares with Figure 32 on page 81 as follows:

- Both explicitly set the command target in the constructor to null.
- Both use the `setCheckingAccount` method to indicate the account on which the command should operate.
- This example sets the target name explicitly by using the `setCommandTargetName()` method. When the default target policy examines its choices, it finds a null for the first choice and a name for the second.

```

...
CheckingAccount checkingAccount
....
try {
    AccountUpdateCmd cmd =
        new AccountUpdateCmdImpl(null, 1000);
    cmd.setCheckingAccount(checkingAccount);

    cmd.setCommandTargetName("com.ibm.sfc.cmd.test.CheckingAccountBean");
    cmd.execute();
} catch (Exception e) {
    System.out.println(e.getMessage());
}
...

```

Figure 28. Identifying a target with `CommandTargetName`

Mapping the command to a target name

The default target policy also permits commands to be registered with targets. Mapping a command to a target is an administrative task that is most appropriately done through a configuration tool. The WebSphere Application Server administrative console does not yet support the configuration of mappings between commands and targets. Applications that require support for the registration of commands with targets must supply the tools to manage the mappings. These tools can be visual interfaces or command-line tools.

Figure 29 shows the registration of a command with a target. The names of the command class and the target are explicit in the code, but in practice, these values would come from fields in a user interface or arguments to a command-line tool. If a program creates a command with null specified for the target, when the default target policy examines its choices, it finds a null for the first and second choices and a mapping for the third.

```

...
targetPolicy.registerCommand(
    "com.ibm.sfc.cmd.test.ModifyCheckingAccountImpl",
    "com.ibm.sfc.cmd.test.CheckingAccountBean");
...

```

Figure 29. Mapping a command to a target in an external application

Customizing target policies

You can define custom target policies by implementing the `TargetPolicy` interface and providing a `getCommandTarget()` method appropriate for your application. The `TargetableCommandImpl` class provides `setTargetPolicy` and `getTargetPolicy` methods for managing custom target policies.

So far, the target of all the commands has been a checking account entity bean. Suppose that someone introduces a session enterprise bean (`MySessionBean`) that can also act as a command target. Figure 30 shows a simple custom policy that sets the target of every command to `MySessionBean`.

```
...
import java.io.*;
import java.util.*;
import java.beans.*;
import com.ibm.websphere.command.*;
public class CustomTargetPolicy implements TargetPolicy, Serializable
{
    public CustomTargetPolicy {
        super();
    }
    public CommandTarget getCommandTarget(TargetableCommand command) {
        CommandTarget = null;
        try {
            target = (CommandTarget) Beans.instantiate(null,
                "com.ibm.sfc.cmd.test.MySessionBean");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 30. Creating a custom target policy

Since commands are implemented as JavaBeans components, using custom target policies requires importing the `java.beans` package and writing some elementary JavaBeans code. Also, your custom target-policy class must also implement the `java.io.Serializable` interface.

Using a custom target policy

The second constructor shown in Figure 16 on page 64 takes a target policy object as an argument, allowing you to use a custom target policy. The example in Figure 31 uses this constructor, passing it a null target and a

custom target policy, so the custom policy is used to determine the target. After the command is executed, the code uses the `reset()` method to return the target policy to the default.

```
...
CheckingAccount checkingAccount
....
try {
    CustomTargetPolicy customPolicy = new CustomTargetPolicy();
    AccountUpdateCmd cmd =
        new AccountUpdateCmdImpl(null, 1000, customPolicy);
    cmd.setCheckingAccount(checkingAccount);
    cmd.execute();
    cmd.reset();
}
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Figure 31. Using a custom target policy

6.1.5.5 Using shippable commands

To use a command, the client creates an instance of the command and calls the command's `execute()` method. Depending on the command, calling other methods may be necessary. The specifics will vary with the application.

In our example application, the server is the `CheckingAccountBean`, an entity enterprise bean. In order to use this enterprise bean, the client gets a reference to the bean's home interface. The client then uses the reference to the home interface and one of the bean's finder methods to obtain a reference to the bean's remote interface. If there is no appropriate bean, the client can create one using a create method on the home interface. All of this work is standard enterprise bean programming.

Figure 32 illustrates the use of the `AccountUpdateCmd` command. This work takes place after an appropriate `CheckingAccount` bean has been found or created. The code instantiates a command, setting the input values by using one of the constructors defined for the command. The null argument indicates that the command should look up the server using the default target policy, and 1000 is the amount the command should attempt to add to the balance of the checking account. After the command is instantiated, the code calls the `setCheckingAccount` method to identify the account to be modified. Finally, the `execute()` method on the command is called.


```

{
    ...
    CheckingAccount checkingAccount
    ...
    try {
        AccountUpdateCmd cmd =
            new AccountUpdateCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
}

```

Figure 32. Using the `AccountUpdateCmd` command

6.1.6 Compensable commands

A compensable command is one that has another command (a compensator) associated with it, so that the work of the first can be undone by the compensator. For example, a command that attempts to make an airline reservation followed by a hotel reservation can offer a compensating command that allows the user to cancel the airline reservation if the hotel reservation cannot be made.

The stages involved in implementing a compensable command are:

- Creating the interface
- Implementing the interface methods
- Writing the compensating command

6.1.6.1 Creating the interface

The first step in creating a compensable command is to write an interface that extends the `CompensableCommand` interface. Such interfaces typically extend the `TargetableCommand` interface as well.

The `CompensableCommand` interface extends the `Command` interface. It declares one method, the `getCompensatingCommand()` method. This method returns the command that can be used to undo the effects of the original command.

Looking at our example, the declaration for the `AccountUpdateCmd` would look like that in Figure 33.

```

....
public interface AccountUpdateCmd extends TargetableCommand,
CompensableCommand {
..// Continue with declarations & methods
.....
}

```

Figure 33. Updated AccountUpdateCmd to show CompensableCommand

6.1.6.2 Implement the interface methods

The getCompensatingCommand() method must be implemented by the application programmer. In our example, this is done in the AccountUpdateCmdImpl class shown in Figure 34.

```

...
public class AccountUpdateCmdImpl extends TargetableCommandImpl
implements AccountUpdateCmd
{
    // Variables
    public AccountUpdateCmdCompensatorCmd accountUpdate
CompensatorCmd;

    ...remaining variables declared here i.e. (Figure 15 in appdes)

    // Method from CompensableCommand interface
    public Command getCompensatingCommand() throws CommandException {
        accountUpdateCompensatorCmd =
            new AccountUpdateCompensatorCmd(this);
        return (Command)accountUpdateCompensatorCmd;
    }
    // Remaining methods from the interface implemented here.
    ..... i.e. Figures 16 to 19 in appdes.
}

```

Figure 34. Implementing the getCompensatingCommand method in AccountUpdateCmdImpl.

This implementation shows the method simply returning an instance of the AccountUpdateCompensatorCmd command associated with the current command.

6.1.6.3 Write the compensating command

An application that uses a compensable command requires two separate commands: the primary command (declared as a `CompensableCommand`) and the compensating command. In the example application, the primary command is declared in the `AccountUpdateCmd` interface and implemented in the `AccountUpdateCmdImpl` class. Because this command is also a compensable command, there is a second command associated with it that is designed to undo its work. When you create a compensable command, you also have to write the compensating command.

Writing a compensating command can require exactly the same steps as writing the original command: writing the interface and providing an implementation class. In some cases, it may be simpler. For example, the command to compensate for the `AccountUpdateCmd` does not require any methods beyond those defined for the original command, so it does not need an interface. The compensating command, called `AccountUpdateCompensatorCmd`, simply needs to be implemented in a class that extends the `TargetableCommandImpl` class. This class must:

- Provide a way to instantiate the command (our example uses a constructor)
- Implement the three required methods:
 - `isReadyToCallExecute()` and `reset()`, both from the `Command` interface
 - `performExecute()` from the `TargetableCommand` interface

Figure 35 shows the structure of the implementation class, its variables (references to the original command and to the relevant checking account), and the constructor. The constructor simply instantiates the references to the primary command and account.

```

...
public class AccountUpdateCompensatorCmd extends TargetableCommandImpl
{
    public AccountUpdateCmdImpl accountUpdateCmdImpl;
    public CheckingAccount checkingAccount;
    public AccountUpdateCompensatorCmd(
        AccountUpdateCmdImpl originalCmd)
    {
        // Get an instance of the original command
        accountUpdateCmdImpl = originalCmd;
        // Get the relevant account
        checkingAccount = originalCmd.getCheckingAccount();
    }
    // Methods from the Command and Targetable Command interfaces
    ....
}

```

Figure 35. Variables and constructor in the AccountUpdateCompensatorCmd

Figure 36 shows the implementation of the inherited methods. The implementation of the `isReadyToCallExecute()` method ensures that the `checkingAccount` variable has been instantiated.

The `performExecute()` method verifies that the actual checking account balance is consistent with what the original command returns. If so, it replaces the current balance with the previously stored balance by using the `AccountUpdateCmd` command. Finally, it saves the most recent balances in case the compensating command needs to be undone. The `reset()` method has no work to do.

```

...
public class AccountUpdateCompensatorCmd extends TargetableCommandImpl
{
    // Variables and constructor
    ....
    // Methods from the Command and TargetableCommand interfaces
    public boolean isReadyToCallExecute() {
        if (checkingAccount != null)
            return true;
        else
            return false;
    }
    public void performExecute() throws CommandException
    {
        try {
            AccountUpdateCmdImpl originalCmd =
            accountUpdateCmdImpl;
            // Retrieve the checking account modified by the original command
            CheckingAccount checkingAccount = originalCmd.getCheckingAccount();
            if (accountUpdateCmdImpl.balance ==
                checkingAccount.getBalance()) {
                // Reset the values on the original command
                checkingAccount.setBalance(originalCmd.oldBalance);
                float temp = accountUpdateCmdImpl.balance;
                originalCmd.balance = originalCmd.oldBalance;
                originalCmd.oldBalance = temp;
            }
            else {
                // Balances are inconsistent, so we cannot compensate
                throw new CommandException(
                "Object modified since this command ran.");
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public void reset() {}
}

```

Figure 36. Methods in the AccountUpdateCompensatorCmd class

6.1.6.4 Using a compensable command

To use a compensating command, you must retrieve the compensator associated with the primary command and call its `execute()` method. Figure 37 shows the code used to run the original command and to give the user the option of undoing the work by running the compensating command.

```
{
    ...
    CheckingAccount checkingAccount
    ....
    try {
        AccountUpdateCmd cmd =
            new AccountUpdateCmdImpl(null, 1000);
        cmd.setCheckingAccount(checkingAccount);
        cmd.execute();
        ...
        System.out.println("Would you like to undo this work? Enter Y or N");
        try {
            // Retrieve and validate user's response
            ...
        }
        ...
        if (answer.equalsIgnoreCase(Y)) {
            Command compensatingCommand = cmd.getCompensatingCommand();
            compensatingCommand.execute();
        }
    }
    catch (Exception e) {
        System.out.println(e.getMessage());
    }
    ...
}
```

Figure 37. Using the `AccountUpdateCompensatorCmd` command

6.1.7 Local command example

In our example application, WebBank, we have identified the need to retrieve data (a user profile) from a database. We will use a command to retrieve this data by sending a request (place a message on an MQSeries queue) to MQSI. The command will be executed locally (no command shipping used for now).

In addition to showing an example of using commands, the code shows two common ways of connecting to MQSeries, namely the use of the Java Messaging Service (JMS) and MQSeries classes for Java.

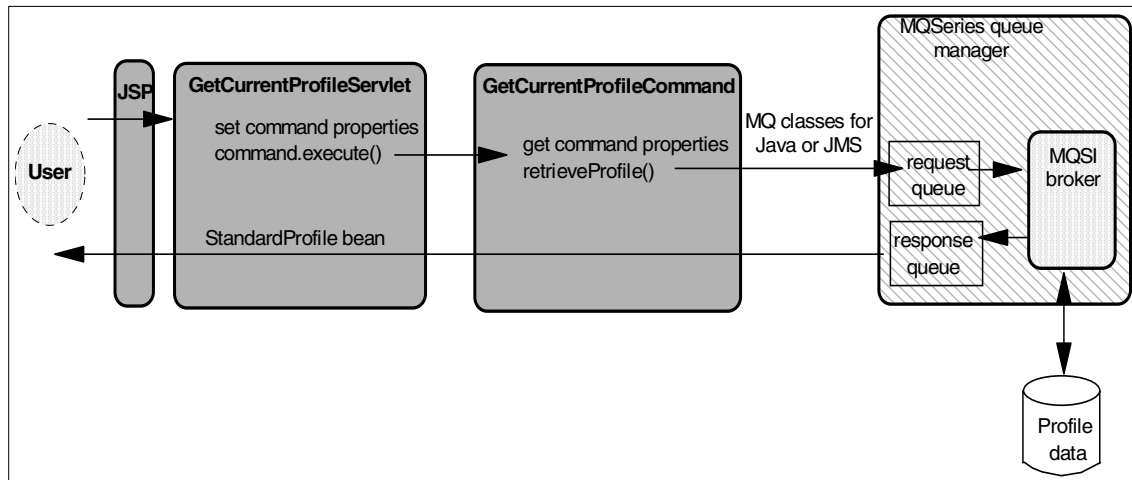


Figure 38. Example application

6.1.7.1 GetCurrentProfileServlet

Figure 39 shows the core function of the performtask() method of the GetCurrentProfileServlet servlet. The servlet is responsible for:

- Instantiating the command

This is shown by the `command =` statement. The `commandClass` parameter is set by the `init()` method of the servlet and is retrieved from a properties file. The value determines whether the MQSeries classes for Java or JMS API is used to construct the XML message sent to and received from the command to the message queues.
- Setting the command properties

These include the user name in the session and MQSeries message queue parameters.
- Calling the command's execute method

This executes locally in the same JVM as the servlet.
- Retrieving the results of the command

The contents of the XML message returned are used to populate the `StandardProfileBean`, which in turn is used by the `profileViewBean` bean to be returned to the view.

```

// Do session check, get username from the string added to session by login servlet
// Instantiate Command
..... more code here .....
GetCurrentProfileCommand command = null;
try{
    command = (GetCurrentProfileCommand)Class.forName(commandClass).newInstance();
}catch(Exception e){
    writeToLog("Exception trying to load the command class");
}
// Set command's properties from file read at initialization time
command.setHostname(properties.getProperty("brokerHostname"));
command.setChannel(properties.getProperty("brokerChannel"));
command.setPort(properties.getProperty("brokerPort"));
command.setQueueManagerName(properties.getProperty("brokerQueueManagerName"));
command.setRequestQueueName(properties.getProperty("brokerSetRequestQueueName"));
command.setReplyQueueName(properties.getProperty("brokerSetReplyQueueName"));
command.setMessageTimeout(Long.parseLong(properties.getProperty("messageTimeout")));
command.setUser((String)session.getValue("user"));
// Now execute
try{
    command.execute();
    if(command.getMessage().startsWith("ERROR")){
        writeToLog("ERROR detected in command message string");
        throw new CommandException(command.getMessage());
    }
}catch(CommandException ce){
    writeToLog("Command exception - message returned by Command is " + ce);
    try{
        RequestDispatcher rd = getServletContext().getRequestDispatcher(errorPage);
        rd.forward(req, res);
        return;
    }catch(ServletException se){
        writeToLog("[performTask] - ServletException " + se.getMessage());
        return;
    }catch(IOException ioe){
        writeToLog("[performTask] - IOException " + ioe.getMessage());
        return;
    } }
.... more code ...
// Add Profile object to StandardProfile bean to be accessed by the view bean & JSPs
req.setAttribute("profile", command.getProfile());

```

Figure 39. Core function of the performtask() method of the GetCurrentProfileServlet servlet

6.1.7.2 GetCurrentProfileCommand

Figure 40 shows the class hierarchy for the GetCurrentProfileCommand.

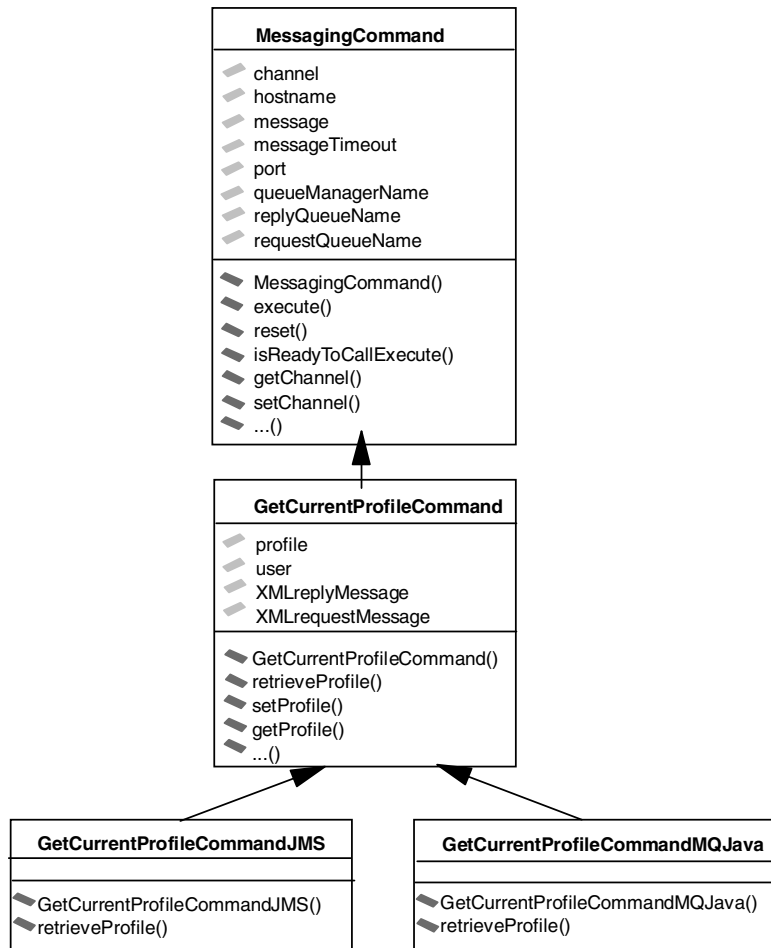


Figure 40. `GetCurrentProfileCommand` class hierarchy

MessagingCommand

The abstract class `MessagingCommand` is a superclass that implements the `Command` and `Serializable` interfaces. Table 3 shows its member field values and their descriptions.

Table 3. Member Fields for MessagingCommand Class

Field name	initial value	Purpose
channel	NULL	The name of the server channel used to connect to the queue manager
hostname	localhost	Host where queue manager lives
message	""	Message field for error conditions
messageTimeout	10000 (ms)	How long to wait for the message response
port	1414	Port of remote machine
queueManagerName	NULL	Name of the queue manager - used only by MQ classes for Java
replyQueueName	NULL	Reply queue name
requestQueueName	NULL	Request queue name

As a command, `MessagingCommand` has the `execute()`, `reset()` and `isReadyToCallExecute()` methods, as shown in Table 4. Note there is no logic in the `execute()` method. This is supplied by the `GetCurrentProfileCommand` subclass. It also has “getter” and “setter” methods for each of the fields listed in Table 3.

Table 4. Methods of the MessagingCommand class

Method	Declaration	Purpose
<code>MessagingCommand()</code>	<pre>public MessagingCommand() { super(); }</pre>	Default Constructor

Method	Declaration	Purpose
execute()	<pre>public void execute() throws CommandException { }</pre>	Declares the work done by the command and throws a CommandException. The logic is implemented by an overriding execute() method in its subclass.
reset()	<pre>public void reset() { }</pre>	This allows the properties to be reset.
isReadyToCallExecute()	<pre>public boolean isReadyToCallExecute() { return true; }</pre>	Confirms that the variables and properties have been set and returns true if the command is ready to execute.

GetCurrentProfileCommand

The GetCurrentProfileCommand is an abstract class that extends the MessagingCommand class. Its declaration, constructor method, and field declarations are shown in Figure 41.

```

package com.ibm.raleigh.pdk.banking.commands;
import com.ibm.websphere.command.*;
import java.rmi.*;
import java.io.*;
import com.ibm.mq.*;
import com.ibm.pdk.banking.StandardProfile;
import com.ibm.xml.parser.*;
import org.w3c.dom.*;
import com.ibm.mq.*;
import java.io.*;
import java.net.*;
import java.rmi.RemoteException;
import java.security.Identity;
import java.util.*;
import java.util.Properties;
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
import javax.naming.directory.*;
public abstract class GetCurrentProfileCommand extends MessagingCommand {
/*
 * user           = The user whose profile we are retrieving
 * XMLrequestmessage = The XML message sent to MQSI
 * XMLreplyMessage = The XML message we get back from MQSI
 * profile        = The profile object that encapsulates a user's personal details
 */
    public String user = null;
    public String XMLrequestMessage = null;
    public String XMLreplyMessage = null;
    public com.ibm.pdk.banking.StandardProfile profile = null;
    public GetCurrentProfileCommand() {
        super();
    }
}

```

Figure 41. *GetCurrentProfileCommand* (abstract and method)

Table 5 shows the setter and getter methods for the fields listed above.

Table 5. "Setter" and "getter" methods for GetCurrentProfileCommand

Method	Declaration
setProfile	<pre>public void setProfile(com.ibm.pdk.banking.StandardProfile newProfile) { Profile = newProfile; }</pre>
getProfile	<pre>public com.ibm.pdk.banking.StandardProfile getProfile() { // Create a new Profile object from the XML retrieved by MQ if(Profile == null){ Profile = new StandardProfile(getXMLreplyMessage()); } return Profile; }</pre>
setUser()	<pre>public void setUser(java.lang.String newUser) { user = newUser; }</pre>
getUser()	<pre>public java.lang.String getUser() { return user; }</pre>
setXMLrequestMessage()	<pre>public void setXMLrequestMessage(java.lang.String newXMLrequestMessage) { XMLrequestMessage = newXMLrequestMessage; }</pre>
getXMLrequestMessage()	<pre>public java.lang.String getXMLrequestMessage() { StringBuffer buffer = new StringBuffer(); buffer.append("<?xml version=\"1.0\" encoding=\"us-ascii\" ?>"); buffer.append("<!DOCTYPE customerrequest SYSTEM \"customerrequest.dtd\">"); buffer.append("<customerrequest>"); buffer.append("<userid>"); buffer.append(user); buffer.append("</userid>"); buffer.append("</customerrequest>"); return buffer.toString(); }</pre>
setXMLReplyMessage()	<pre>public void setXMLreplyMessage(java.lang.String newXMLreplyMessage) { XMLreplyMessage = newXMLreplyMessage; }</pre>

Method	Declaration
getXMLReplyMessage()	<pre>public java.lang.String getXMLreplyMessage() { return XMLreplyMessage; }</pre>

Figure 42 shows the execute() method called by the servlet. This method calls the setter method for XMLReplyMessage with the argument of the retrieveProfile() method.

```
public void execute() throws CommandException {
    // Retrieve the XML message from MQSI and return it as a String
    setXMLreplyMessage(retrieveProfile());
}
```

Figure 42. execute()

From Figure 40 on page 89 you can see that the retrieveProfile() method is overridden by either of the concrete subclasses, depending on which method (JMS or MQ base Java) is chosen.

The retrieveProfile() method does the following:

- Builds an XML message based on the user name and other properties set by the servlet.

The message is constructed by calling the getXMLRequestMessage() method.

- Puts the message on the request queue and waits for a response on the output queue.

If the response message doesn't appear on the queue in time there is a CommandException thrown. Otherwise, the XML message is returned from the queue.

At this point the servlet takes control and calls the getProfile() method of the command. This creates the StandardProfile bean and populates it with the XML message using the getXMLReplyMessage() method.

Table 6 shows the remaining methods defined in the command.

Table 6. Remaining methods in GetCurrentProfileCommand

Method	Declaration	Purpose
reset()	<pre>public void reset() { }</pre>	This allows the properties to be reset.

Method	Declaration	Purpose
isReadyToCallExecute()	<pre>public boolean isReadyToCallExecute() { return true; }</pre>	Confirms that the variables and properties have been set and returns true if the command is ready to execute.
writeToLog()	<pre>System.out.println(" [G etCurrentProfileComman d] - " + message); }</pre>	Logging method.

6.2 Using MQSeries to send and retrieve data

In addition to the command package, this pattern introduces the use of MQSeries and MQSI. We will take a look at two different programming APIs we can use to exchange messages between our Java application and a back-end application using MQSeries.

To illustrate the procedure, we will take a look at some code from a sample application. Our application will request data from a back-end application by building a request for the data in XML format and placing that request on an MQSeries queue. The requested data will be sent back to our application in the form of an XML message on a reply queue. These tasks are performed by a method called `retrieveProfile()`. This method is a part of a command called `GetCurrentProfileCommandMQJava` in our MQSeries classes for Java example and a part of the `GetCurrentProfileCommandJMS` command in our JMS example.

The procedure for both APIs is basically the same:

1. Establish a connection.
2. Create an XML message.
3. Put the message on the MQSeries queue.
4. Wait for a response to appear in the output queue.
5. Get the reply message from the queue.
6. Return control back to the calling method.

6.2.1 MQSeries classes for Java

The MQSeries classes for Java allow a program written in the Java programming language to connect to MQSeries as an MQSeries client, or

directly to an MQSeries server. It enables Java applets, applications, and servlets to issue calls and queries to MQSeries giving access to mainframe and legacy applications, typically over the Internet, without necessarily having any other MQSeries code on the client machine. With the MQSeries classes for Java the user of an Internet terminal can become a true participant in transactions, rather than just a giver and receiver of information.

6.2.1.1 Initialization

The first step in the method is to define the name of the MQ objects to connect to, in this case the “request” and the “reply” queues:

```
String requestQueueName = "ITSO.ADDR.REQ.IN";
String replyQueueName = "ITSO.ADDR.REPLY";
```

All message queues are managed by queue managers, so actual MQ Java object references need to be defined for them. MQQueue provides inquire, set, put, and get operations for MQSeries queues. The MQQueueManager object provides resources for querying and manipulating the queue manager:

```
com.ibm.mq.MQQueueManager qMgr = null;
com.ibm.mq.MQQueue requestQueue = null;
com.ibm.mq.MQQueue replyQueue = null;
```

MQEnvironment contains static member variables which control the environment in which an MQQueueManager object (and its corresponding connection to MQSeries) is constructed. Values set in the MQEnvironment class take effect when the MQQueueManager constructor is called, so you should set the values in the MQEnvironment class before constructing an MQQueueManager instance:

```
MQEnvironment.hostname = hostname;
MQEnvironment.channel = channel;
MQEnvironment.port = port;
```

6.2.1.2 Connect to the queue manager

To place a message on an MQSeries queue, we first need to connect to the queue manager that has the queue:

```
qMgr = new MQQueueManager(queueManagerName);
```

Next, we need to open the queue. The first step is to determine the options needed to open the queue. If we are using one queue for both input and output (requestQueueName = replyQueueName) then we want to open that queue for both put and get operations. Otherwise, we want to open the request queue for put operations only:

```
if (requestQueueName.equals(replyQueueName)) {
```



```

        openOptions = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_OUTPUT;
    }
    else{
        openOptions = MQC.MQOO_OUTPUT;    // Open queue to perform MQPUTs
    }

```

The MQC interface defines all the constants used by the MQSeries classes for Java. To refer to one of these constants from within your programs, prefix the constant name with "MQC."

MQC.MQOO_OUTPUT and MQC.MQ00_INPUT_AS_Q_DEF are interfaces in the MQQueueManager class (MQSeries classes for Java). MQC.MQOO_OUTPUT opens a queue to put messages on it. MQC.MQ00_INPUT_AS_Q_DEF opens a queue to get messages using the queue-defined default.

Now we specify the queue we wish to open (the request queue) and the open options determined in the previous code:

```

requestQueue = qMgr.accessQueue(requestQueueName, openOptions,
                                null,                // default q manager
                                null,                // no dynamic q name
                                null);               // no alternate user id

```

6.2.1.3 Send a message

Now that the queue is open we need to construct the message and put it on the queue. The construction is done with the following code:

```

MQMessage requestMessage = new MQMessage();
requestMessage.replyToQueueName = replyQueueName;
requestMessage.replyToQueueManagerName = queueManagerName;
requestMessage.writeString(getXMLrequestMessage());

```

MQMessage represents both the message descriptor and the data for an MQSeries message. Having created requestMessage, we need specify where our application will be looking for a reply with replyToQueueName() and replyToQueueManagerName.

The writeString() method writes a string into the message buffer at the current position. In this case it uses the getXMLrequestMessage() method of the GetCurrentProfileCommand Command to compose the request to be put on the queue in XML format (see Table 5 on page 93).

Finally the queue options need to be set and the message can be put on the request queue:

```

MQPutMessageOptions pmo = new MQPutMessageOptions();

```

```
requestQueue.put(requestMessage, pmo);
```

This code accepts the default options.

6.2.1.4 Get the reply message

Having put the request message in the queue, the method waits for a response from the output queue. The first thing that needs to be done is that the reply queue needs to be identified, much like the request queue:

```
replyQueue = qMgr.accessQueue(replyQueueName,  
                               openOptions,  
                               null,          // default q manager  
                               null,          // no dynamic q name  
                               null);         // no alternate userid
```

An object needs to be created for the reply message with its properties set to include the original request message. This is done using the correlation ID property of the message queue:

```
MQMessage replyMessage = new MQMessage();  
replyMessage.correlationId = requestMessage.messageId;
```

We set the message options and issue a get with wait indicating the maximum amount of time (2000 milliseconds) that we will wait for the message to arrive:

```
MQGetMessageOptions gmo = new MQGetMessageOptions();  
gmo.options = MQC.MQGMO_WAIT;  
gmo.waitInterval = 2000;  
replyQueue.get(replyMessage, gmo);
```

When the get has been executed, the XML document is extracted from the message and returned to the calling method:

```
int msglen = replyMessage.getMessageLength();  
String msgText = replyMessage.readString(msglen);  
return msgText;
```

6.2.1.5 Clean up

After catching any exceptions and error trapping, the final thing to do is to close the queue and disconnect from the queue manager:

```
requestQueue.close();  
qMgr.disconnect();
```

The complete method, showing the program flow and error trapping can be found in B.1, “GetCurrentProfileCommandMQJava: retrieveProfile() method” on page 393.

6.2.2 Java Messaging Service (JMS)

One of the architectural objectives of JMS is to provide a common way for Java applications to interact with enterprise Message Oriented Middleware (MOM) systems. JMS defines a generic view of a message passing service. It is important to understand this view and how it maps onto the underlying MQSeries transport. The generic JMS model is based around the following interfaces that are defined in Sun's `javax.jms` package:

- *Connection*: Provides access to the underlying transport, and is used to create sessions.
- *Session*: Provides a context for producing and consuming messages, including the methods used to create `MessageProducers` and `MessageConsumers`.
- *MessageProducer*: Used to send messages.
- *MessageConsumer*: Used to receive messages.

It is important to note that a `Connection` is thread safe, but `Sessions`, `MessageProducers` and `MessageConsumers` are not. The recommended strategy is to use one `Session` per application thread.

In MQSeries terms:

- *Connection*: Provides a scope for temporary queues, as well as a place to hold the parameters that control how to connect to MQSeries (for example, the name of the queue manager, and the name of the remote host if using the MQSeries Java client connectivity).
- *Session*: Contains an `HCONN` (a handle to a queue manager) and therefore defines a transactional scope.
- *MessageProducer* and *MessageConsumer*: Contains an object handle (`HOBJ`) that defines a particular queue for writing to or reading from.

Note that the following normal MQSeries rules apply:

- Only a single operation can be in progress per `HCONN` at any given time, so the `MessageProducers` or `MessageConsumers` associated with a `Session` cannot be called concurrently. This is consistent with the JMS restriction of a single thread per `Session`.
- `PUTs` can use remote queues, but `GETs` can only be applied to queues on the local queue manager. The generic JMS interfaces are subclassed into more specific versions for point-to-point and publish/subscribe behavior.

The point-to-point versions are:

- QueueConnection
- QueueSession
- QueueSender
- QueueReceiver

One of the key ideas in JMS is that it is possible, and strongly recommended, to write application programs using only references to the interfaces in `javax.jms`. All vendor-specific information is encapsulated in implementations of:

- QueueConnectionFactory
- TopicConnectionFactory
- Queue
- Topic

These are known as administered objects, which are so named because they can be built using a vendor-supplied administration tool and can be stored in a JNDI namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation.

Once again, let's take a look at our sample application, but this time our `retrieveProfile()` method will be written using JMS.

6.2.2.1 Initialization

During the initialization phase, you will need to declare the local JMS variables.

```
QueueConnection connection = null;
QueueSession session = null;
QueueSender messageProducer = null;
QueueReceiver messageConsumer = null;
TextMessage replyMsg = null;
Queue destQueue=null;
Queue replyQueue=null;
```

Next we need to obtain a context to look up the JMS admin objects (using WebSphere CosNaming JNDI service provider).

```
InitialContext messagingContext = new InitialContext();
```

Having obtained an initial context, objects are retrieved from the namespace with the `lookup()` method. The following code retrieves a

QueueConnectionFactory named queueManagerName from an LDAP based namespace:

```
MQQueueConnectionFactory cf = (MQQueueConnectionFactory)
    messagingContext.lookup(queueManagerName);
```

The MQQueueConnectionFactory set methods are used to customize the factory with MQSeries specific information:

```
cf.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
cf.setHostName(hostname);
cf.setQueueManager(queueManagerName);
cf.setChannel(channel);
cf.setPort(port);
```

MQ JMS can communicate with MQSeries using either the client or bindings transports. Use of the Java bindings requires the JMS application and the MQSeries queue manager to be located on the same machine. The client permits the queue manager to be on a different machine to the application. Our example uses the client transport and is set in the above code with the `cf.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP)` declaration.

The `createQueueConnection()` method is used on the factory object to create a connection:

```
connection = cf.createQueueConnection();
```

The JMS specification defines that connections should be created in the “stopped” state. Until the connection is started, no messages can be received by MessageConsumers that are associated with the connection. To start the connection, issue the following command:

```
connection.start();
```

6.2.2.2 Send a message

To send a message, get the JMS admin objects from the naming provider:

```
destQueue = (Queue)messagingContext.lookup(requestQueueName);
replyQueue = (Queue)messagingContext.lookup(replyQueueName);
session =
    connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

Create a producer of messages:

```
messageProducer = session.createSender(destQueue);
```

Create an XML message string:

```
TextMessage msg = session.createTextMessage();
msg.setText(getXMLrequestMessage());
```

Set the reply to queue for the message:

```
msg.setJMSReplyTo(replyQueue);
```

Put the message on the queue:

```
messageProducer.send(msg);  
String msgID = msg.getJMSMessageID();
```

6.2.2.3 Get the reply message

Create a consumer for the above message, using the JMSCorrelationID as the messageSelector:

```
messageConsumer = session.createReceiver(replyQueue,  
    "JMSCorrelationID = " + "'" + msgID + "'");
```

Wait for the reply:

```
replyMsg = (TextMessage)messageConsumer.receive(messageTimeout);
```

If null is returned on timeout, throw an exception, otherwise store the XML string:

```
return replyMsg.getText();
```

Last, trap for errors.

6.2.2.4 Clean up

After catching any exceptions and error trapping, close the queue and disconnect from the queue manager:

```
messageProducer.close();  
messageConsumer.close();  
session.close();
```

The complete method, showing the program flow and error trapping can be found in B.2, “GetCurrentProfileCommandJMS: retrieveProfile() method” on page 397.

Chapter 7. MQSI application design guidelines

In application topology 5, the design principles are geared towards the direct Web-enablement of back-end system features. In this context, the role of MQSeries Integrator as the application router is to handle requests from the presentation-oriented tiers, routing the request to back-end systems in order to retrieve and/or update information according to the application requirement.

7.1 MQSeries and MQSI as message-oriented middleware

In Chapter 5, “Technology options” on page 35, we discussed the function of message-oriented middleware (MOM). Now, let’s look at how MQSeries and MQSeries Integrator can be combined to provide these functions.

7.1.1 MQSeries - the MOM transport layer

The IBM MQSeries product provides the functionality required in the MOM transport layer.

The transportation of message data across a network is enabled through deployment of a set of MQSeries *queue managers*. Each queue manager hosts *local queues* that are containers used to store messages. Through *remote queue* definitions and message *channels*, messages may be transmitted between queue managers.

To use the services of an MQSeries transport layer, an application must make a connection to an MQSeries queue manager, the services of which will enable it to receive (*get*) messages from local queues, or send (*put*) messages to any queue on any queue manager. The application’s connection may be made directly (where the queue manager runs locally to the application) or as a client to a queue manager that is accessible over a network.

7.1.2 MQSeries Integrator - transformation and integration

MQSeries Integrator (MQSI) is the IBM product that fulfills the needs for message transformation and routing and in addition, provides an integration architecture that extends well beyond these capabilities.

Message flows

In MQSI, each transformation and routing process is developed as a *message flow*. A message flow is built as a sequence of operations, each of which takes the form of a node. Each type of node has the applicable

connectors that allow its input and output flows to be defined visually. Nodes can perform a variety of tasks, for example:

- Get a message from an input queue
- Apply transformation rules to a message
- Perform a database operation
- Put a message to a target queue

There is a wide variety of nodes available, many of which are highly configurable.

Message flows may be configured with generic input and output terminals so that they may be re-used as components of greater message flows.

Broker topologies

MQSI provides the means to configure and deploy message flows to a managed network of message brokers. In the MQSI context, a broker is essentially a container service that manages the operation of message flows. A typical MQSI network will consist of many brokers.

A broker uses the services of a local MQSeries queue manager to get messages as input to its message flows and to handle the routing of messages that are output from message flows. The queue manager is also used where internal communication with other MQSI components is required.

Configuration management

The management of the operational setup of all brokers is done using a single, central *configuration manager* service. The configuration manager is used to hold details of the broker domain topology, and the definition of all message flows. It is also used to control assignment and deployment of message flows to the broker network.

The configuration manager hosts a multi-user configuration management (and development) platform for broker operations and message flow definition.

Message formats and transformation

A wide variety of message formats are accepted by MQSI, and a number of facilities exist to define message format information to MQSI for use in transformation operations.

Most impressive is the internal ESQL language available to the many node types. ESQL may be used to interchangeably manage the content of

structured message or database tables. This is particularly powerful in the handling of message content held in XML.

7.2 MQSeries Integrator topology

Although topology 5 depicts the application router as a single logical entity, MQSI is a highly scalable and potentially highly distributed operational environment unto itself. For this reason, effective implementation of an MQSI router involves far more than a case of good message flow design. How the MQSeries infrastructure is built, as well as the distribution of the MQSI components, will have a definite role in determining how the application performs.

When designing the MQSeries and MQSI topology, the following topics should be considered:

- WebSphere-to-MQSI connectivity options used
- MQSeries queue manager roles and relationships
- MQSI services and brokers in the MQSeries network
- Placement of databases within the network

7.2.1 WebSphere-to-MQSI connection options

The request from the application server may originate directly from a servlet, or may be sent from a command bean or EJB. Regardless of the method used, the MQSeries message will be sent to a queue manager using one of the available MQSeries Java APIs. Each API has certain characteristics that make it appropriate for a situation, depending on the priorities you have. However, the API chosen can have an effect on the options you have for distributing the application components.

The two APIs that we will discuss here are:

- The MQSeries classes for Java (MQ base Java) package, `com.ibm.mq.jar`. MQ base Java enables Java applets, applications, and servlets to issue calls and queries to MQSeries.
- The MQSeries classes for Java Message Service (MQ JMS) package, `com.ibm.mqjms.jar`. MQ JMS implements Sun's Java Message Service (JMS) to enable JMS programs to access MQSeries.

If application portability and vendor independence is of importance, JMS is the obvious winner for your choice of technology. JMS uses abstracted concepts of messaging to provide a vendor-independent API to messaging, while underneath lies the MQSeries implementation of the JMS interfaces.

Objects, the real world entities that are MQSeries queue managers and queues, are defined to JMS through use of a directory naming service (MQSeries message service). MQ JMS supports both the point-to-point and publish/subscribe models of JMS.

MQ base Java and MQ JMS provide two connection options to MQSeries:

- Bindings mode to connect to a queue manager directly
- Client mode using TCP/IP to connect to a queue manager

7.2.1.1 Java bindings mode

The fastest link from Java to MQSeries is to use the Java bindings mode. This provides a direct connection to an MQSeries queue manager that resides on the same host as the application. The key connection parameter in this case is the queue manager name.

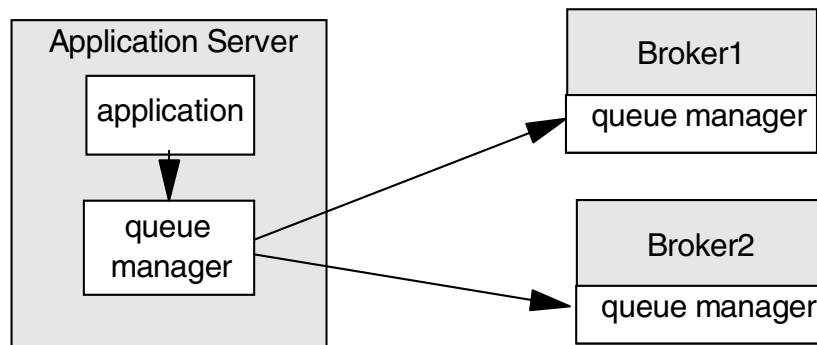


Figure 43. Java bindings mode

Connecting to the local queue manager has several major advantages. First, the probability of establishing a connection to a queue manager in your own host is high as opposed to a connection with a remote queue manager. Second, the time it takes to establish a network connection to the queue manager is avoided. Third, the local queue manager can distribute the work among multiple brokers. If connection performance is a high priority in your network, then using bindings mode is the clear choice.

Using bindings mode also has the advantage of using MQSeries as an XA resource coordinator for units of work that involve MQSeries updates and, for instance, DB2 updates.

7.2.1.2 Java client mode

MQSeries classes for Java and MQSeries classes for JMS provide for connectivity to MQSeries in client mode. This is similar to bindings mode, but

the connection to the queue manager is made through a server connection channel, meaning applications may connect to queue managers on other hosts. The key connection parameters are host name, TCP/IP port, and server connection channel name.

The client mode is best used when you do not want MQSeries to reside on the same machine as the application server, possible for security reasons. It allows you to connect directly to a remote MQSeries.

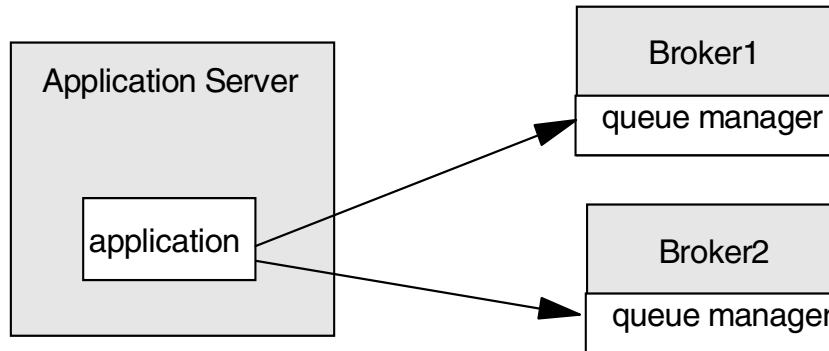


Figure 44. Client mode to remote brokers

When you connect directly to a queue manager on a broker, as in Figure 44, you relinquish any workload distribution the queue manager offers. The application must decide which broker to send the work to and any workload distribution would have to be done in the application itself. Even having the queue managers in a cluster does not help, since a queue manager will always send the work to the local instance of the broker.

One way around this is to connect to a remote queue manager that does not have a broker instance, but is there purely for workload distribution, as shown in Figure 45.

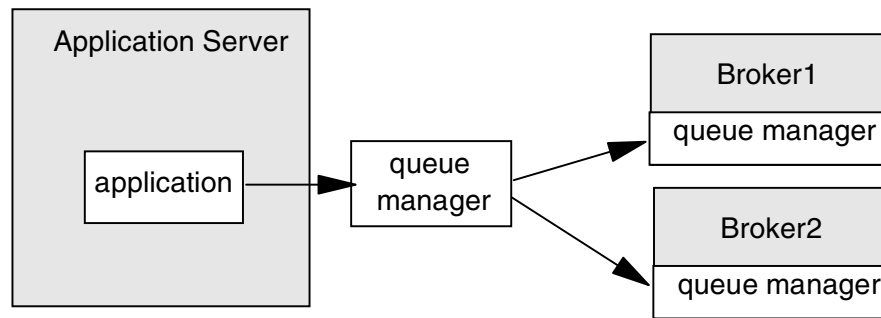


Figure 45. Client mode to a remote queue manager

You still have the network connectivity time; in fact, you have made it worse by introducing an intermediate system. But you do have the advantage of the queue manager workload distribution and the ability to connect to a remote queue manager.

The client mode can also be used to connect to the local queue manager by passing through the internal TCP/IP stack. This is obviously not as efficient as using the bindings mode, but it does allow your program to be used in a generic environment where you do not know if the queue manager will be local or not.

Both JMS and MQ base Java allow you to put MQ messages from the Java application in WebSphere directly to the remote broker's queue. If you are thinking of doing this, you should consider the performance implications. The cost of creating a network connection is added to the total cost of each request. For each request, an MQ-to-client session is created. There is no long-lasting network connection. This will impact the ability to run thousands of sessions in parallel. If you create a local MQ session for each request, the overhead will be much lower. The network connection is now maintained by a sender-receiver channel pair and is then long-running. Ideally, long-running MQ sessions are preferable. In the next version of MQSeries, the overhead of creating a local session is expected to be reduced dramatically, making the performance gap between client and bindings even larger.

7.2.2 Queue manager roles and relationships

Queue managers are the heart of the MQSeries network, providing the messaging services to the applications. Careful thought should be put into how the queue managers are defined, their roles with regard to MQSeries and MQSI, and their relationships to each other.

7.2.2.1 Application design

To be flexible in design, it is good practice for application code to be independent of its underlying infrastructure. Use of specific queue manager and queue names within application code will inevitably lead to high maintenance overhead in the future.

To avoid this, your application code should use its own aliases for the queue manager and queues with which it interacts, the interpretation of which is left to the supporting infrastructure.

When JMS has been used as the API, one option is to use the JMS configuration detail to handle the translation between application object names and real object references. Where the MQSeries classes for Java have been used, queue managers must be commissioned using the applicable set of queue and queue manager aliases that will enable messaging functions to target the applicable queues.

Another API, Application Messaging Interface (AMI), is a policy-based programming interface. With AMI, you send a message to a service according to a certain policy and AMI will translate this into queue names, queue manager names, and other MQSeries options. This means the programmer is shielded from the implementation details of your MQSeries network.

For maximum availability, the application object should be designed to be able to connect to one of a set of queue managers, so that if the normal point of connection is not available, an alternative connection may be achieved without compromising system functionality.

7.2.2.2 Use of clustering

All MQSI components other than the administration client (Control Center) require the services of an MQSeries queue manager for their intercommunication (since this is handled using MQSeries messages).

For ease of administration, the queue managers that support the MQSI components can be configured in a cluster. To establish a clustered network of queue managers, a pair of queue managers should be configured as the cluster repository queue managers. These queue managers have a special role, in that they hold a full record of the queue managers and shared objects within the cluster. Additional queue managers can be added as members of the cluster, with a defined cluster-sender channel to one of the repository queue managers.

To get a better view of how this should be deployed in relation to the network, we need a good understanding of what we will use the queue managers for.

7.2.2.3 Queue manager roles

When designing the queue manager network to support the application router, we must consider the distinct roles that queue managers will be expected to fulfill, together with the design priorities we may have from the application. This will enable us to define templates that can be used to deploy instances of the roles to a complete network, making scalability easier to manage.

Some of the queue managers may be required to support the key MQSI services, the Configuration Manager, User Name Server, or a broker. Other queue managers may be specifically designed to directly serve the calls from an application program presentation tier, for example, an EJB container on WebSphere Application Server Advanced Edition.

A queue manager that serves a broker will need to host *local queues* that are used as input source for the broker's message flows. An application's queue manager has similar requirements, plus we need to consider how the *alias queue* definitions deployed to the queue manager relate to the real queue definitions. We must also consider the channel definitions that will be necessary when defining a new instance of the role to a network.

In practice, it may be beneficial to deploy more than one role to a single queue manager for reasons of performance or cost constraint. This is of course possible, but by keeping the design role-focused, you will achieve flexibility in deployment.

The application may also have specific design requirements. For example, it may be vital to achieve maximum performance of message throughput between EJBs and the message broker. Since performance is affected by network connectivity issues, hosting the roles on the same queue manager will usually mean better performance.

Alternatively, performance may be less of a priority but achieving once-only assured delivery of a given message may be imperative. Where separate queue managers are used to host the application handler and broker roles, assured messaging is achieved.

MQSI Configuration Manager role

In any MQSI domain, the Configuration Manager service is the entity that supports the maintenance of the MQSI configuration. While, strictly speaking, a Configuration Manager is not required in order for brokers to operate, the Configuration Manager service is likely to be active at all times. The volume of message traffic to and from the Configuration Manager's

queue manager is limited to the messages that are concerned with management of the broker domain configuration.

Given the high availability requirements and central role this queue manager plays, it is a clear choice as a cluster repository in the MQSeries network. For details of how to configure a queue manager as a cluster repository, refer to 10.1.4, “Overview of the MQSeries clustering feature” on page 249.

There are no application-specific MQSeries configuration steps to add to a given Configuration Manager queue manager.

MQSI User Name Server role

The User Name Server service provides a topic-oriented authentication mechanism for the publish/subscribe features of MQSI. For this reason, if publish/subscribe is implemented, all brokers must have access to the User Name Server at all times.

Given the high availability requirements and central role this queue manager plays, it is a clear choice as a cluster repository in the MQSeries network. For details of how to configure a queue manager as a cluster repository, refer to 10.1.4, “Overview of the MQSeries clustering feature” on page 249.

In MQSI environments that do not use publish/subscribe, the User Name Server service is not required. However it does no harm to implement it at initial setup and of course makes it easier to introduce a publish/subscribe architecture at a later stage. Though topology 5 uses MQSI as a router and does not require publish/subscribe, this is a key feature of MQSI and will most likely be implemented at some point in time.

MQSI broker role

The broker services are the workhorses of the MQSI product. A typical implementation of MQSI will involve many brokers that may have different or identical roles. Deployment of brokers to the network is a design issue in itself. The location of these will affect their performance according to the nature of the flows they are configured to handle.

The application involved will have an influence on the design of the broker topology and the operation of message flows deployed to each broker in the domain. Typically, a request from the presentation tier will take the form of an MQSeries message supplied as input to a message flow. This act initiates a sequence of events that is designed to accomplish a business function of some kind. The request may or may not require a reply, and where a reply is expected, the request/reply interaction may need to be handled synchronously or asynchronously. Clearly where synchronous interaction is

required, performance is a key issue that will affect the decisions made on component distribution.

Applications running on a broker may send (*put*) reply or output messages to any remote queue and queue manager in the network, subject to authorization. However, applications can only retrieve (*get*) messages from queues defined to the local queue manager.

To achieve workload balancing, broker queue managers can be duplicated across the network. Each broker queue manager has a unique name, but will contain identical local cluster queue definitions. The administration task of setting up multiple brokers identically is made easy through the use of scripts using MQSeries commands. You can see this in 10.2.1.1, “Administering multiple brokers” on page 268.

Proximity to the back-end application data will clearly result in a more efficient message flow performance where message flows involve interaction with legacy databases or transactions. Proximity to the client application will gain a more efficient response when only local operations are performed within the message flow. Clearly these two dynamics may conflict and the best solution may be to divide the workload into distinct subflows. Here, more than one broker would be involved in the complete operation, each handling the work it is best placed to. In this scenario, your configuration would have more than one broker queue manager role.

MQSI application request handler role

Queue definitions must be defined to those queue managers that will be assigned the application request handler role. Input queues for the application must be defined locally to these queue managers.

Often, it is desirable to define an alias for the real queue names to be used by the application. This eliminates the need to change and recompile the application if the real queue name changes. The aliases provide the translation between queue names as defined to the application and actual queue names on the MQSeries network.

High availability and load balancing may be achieved here too, through deployment of many queue managers with a duplicate role. Where an application finds itself unable to connect to one queue manager, it is possible for an alternative connection to be made. The responsibility for handling this routing lies in the application code design itself. For example, a set of hosts could be deployed, each with an EJB container and queue manager to handle MQ calls from the EJBs.

7.2.3 Placement of MQSI databases

The time required to access a database can be a significant factor in application performance.

There will be both system databases and user databases involved in running an MQSI application. System databases are required to support the MQSI system services. User databases will contain the data required by the application. These can be legacy databases, new databases created for the application, or databases that are used by the application for temporary storage (or caching).

The frequency and type of database access, security considerations, and stability requirements will all factor into the decision of where to place them in the network.

7.2.3.1 System databases

The MQSI Configuration Manager and MQSI brokers store information in database tables. Each type of information is stored in a unique table. The Configuration Manager uses two tables, one as a configuration repository and one as a message repository. Each broker will also have a unique table where it stores persistent information.

These tables may be combined in one database or may be separated into multiple databases. For our discussion and examples we will assume that the databases have been split into separate databases, referred to as the configuration database, message repository database, and broker databases. The configuration and message repository databases must be defined using DB2. The broker databases can be DB2, Microsoft SQL Server, Oracle, or Sybase. These databases can be local or remote, though obviously the highest system performance is achieved when a database is defined on the same host as the system that uses it.

A broker database may be best situated on the broker machine where performance is of greater importance due to intense message flow activity. Management issues raised due to the scattering of these databases, such as centralized backup capability, are not as much of a concern. If the data in a broker database becomes corrupt or lost it can be readily redeployed from the central configuration.

Database performance is a lesser concern in the configuration and message databases, since these are utilized in development and system management operations only. It is of greater importance to preserve the integrity of these databases, so you may elect to hold these in a central repository where backup and security features are already established.

7.2.3.2 User databases

Message flows deployed to reference or update existing application databases will most likely need to access remote databases. These requests are often for legacy data and are passed to a back-end system. When this type of operation involves many steps and perhaps many references to remote databases, there is clearly some performance penalty. Use of a “cache” database that is local to the broker as a store for commonly accessed data can lead to a considerable performance benefit.

A cache database can be used to enable efficient access to data that is used frequently in the message flows. In this design, a database local to the broker is used to hold a cache for frequently accessed but seldom changed information. When designing an application to use a caching database, consideration must be given to the possibility that there will be duplicate broker deployment for load-balancing purposes. In this case, it may be necessary for the brokers to use a common (but not necessarily local) cache. In this case the issue becomes one of which is faster, the cache or the legacy database.

7.3 MQSI message flow design

From our logical design (see Chapter 8, “Application development guidelines” on page 141), we will have a clear understanding of the business operations (the *model*) that message flows are required to fulfill. The logical design describes the requirements at a high level and forms the basis for defining operational contracts between the WebSphere application components and the MQSI domain. To implement a solution on MQSI that satisfies an operational contract may involve several steps of message transformation and routing. A given process may require the involvement of more than one broker role.

To design complex message flows, it is helpful to consider the role they play within the application in a wider context. Essentially, message flows are all about document exchange, involving actions such as document transformation and database operations. It is very easy to leap into the development environment in MQSI and throw together a large set of interconnected nodes to carry out a complete, complex message transformation. Unfortunately, this style of message flow development can lead to high maintenance overhead in the future.

One of the key features of MQSI is its ability to store message flows as reusable components that can be linked together and used as part of larger operational message flows (which we will refer to as *deployable* message

flows). In this way, each component in a complete operation is developed and tested in its own right with its own clear specification of its inputs, actions and outputs. For example, some transformations can require writing complex code in the form of ESQL. There is a clear benefit to only having to do this once, and importantly, only having one instance of a given transformation definition in the MQSI shared configuration. Once the components of a complex operation are defined in this way, it becomes a straightforward task to string subflows together to compile a complete business operation.

The golden rule is: keep your individual message flow components as simple as possible! They will be easier to test and easier to work with later. It can be very hard to trace a fault in a complex flow with dozens of nodes and many more connections.

7.3.1 Design contract with the application

As a high-level concept, the application's contract with MQSI defines the message document to be supplied, what the expected actions are, and the expected reply documents. It will include such details as whether an operation will be carried out synchronously or asynchronously.

From the MQSI end, the process will involve receipt of a message to initiate the required actions. These may involve retrieving and returning information in the form of a reply, updating data in other systems, or any combination of such things according to the details of the contract.

7.3.2 Message flow structure

The complete message flow deployed to handle this contract will be designed to take the incoming document and perform the required series of functions that carry out the business operations set out in the contract.

The need to develop standard documents that support communication between MQSI and the WebSphere application is clear. However, to complete the entire unit of work, it may be necessary to perform many transformations of the original document into alternative forms so that individual actions can be carried out. To this end, there is also a great benefit to be gained from developing document standards that support the interaction between subflows.

Common database operations based on a standard incoming document type can be saved as message flow components. In order to use such components, common document transformations must also be defined and saved as message flow components.

Here we can begin to visualize how business operations modelled as MQSI message flow component parts are linked together to form complete business processes. For example, an incoming message of a “profileupdate” document type, may involve some transformation to create a “savingsupdate” document that can be used to perform a “savings account update” database operation.

It is also possible to build a message flow that can analyze the content of the incoming message and route the message to a subflow that is capable of working with that type of message.

7.3.3 Defining document types

To enable message flow components to be compatible with each other, the document types that are exchanged between them must be defined. The definitions must detail the parser that will be used to interpret the message, and also the structure and content of the message itself.

For XML documents, this is achieved through using a Document Type Definition (DTD). In this Redbook, shall assume some prior knowledge of XML and the use of DTDs. For information on XML, see *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104.

7.3.3.1 Example DTD

Here is a simple example of an XML DTD. This DTD describes a document known as “profilemessage”. Notice how the DTD defines the data elements that make up the profilemessage. You can see that “profilemessage” comprises three child elements, “userid”, “custname” and “custaddr”. The DTD then defines each of these child elements. Definitions continue in this way until leaf elements (the data itself) are defined using a given data representation. All elements shown here are of type #PCDATA, a term meaning “parseable character data”... a string, basically.

```
<!-- profile.dtd -->
<!ELEMENT profilemessage (userid,custname,custaddr)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT custname (namepref, forename, middlename, surname)>
<!ELEMENT namepref (#PCDATA)>
<!ELEMENT forename (#PCDATA)>
<!ELEMENT middlename (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT custaddr (houenum, housename, street, district, city, state,
country, zip, telephone)>
<!ELEMENT houenum (#PCDATA)>
<!ELEMENT housename (#PCDATA)>
```

```
<!ELEMENT street (#PCDATA)>
<!ELEMENT district (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
```

7.4 Message flow components

Message flows are implemented as sequences of logically connected events, defined in the form of message flow *nodes*. Each node performs a specific function in the flow. MQSI comes with a set of message flow nodes called the IBM primitives, which supply a wide range of functions.

MQSI provides a programming interface that allows you to create new nodes that provide new message processing function or supersede existing function. You can also create new message parsers to access types of messages not defined by MQSI.

Message flows are built in the Control Center by using a drag-and-drop user interface. We will look at how this is done later in Chapter 9, “Developing the MQSI application” on page 183.

7.4.1 Message flow inputs and outputs

MQSeries Integrator is principally based on the messaging transport layer provided by the base MQSeries product. Message flows that operate within MQSI must take MQSeries messages as their input and may result in one or many MQSeries messages as output.

You should note, however, that MQSI is designed as an extensible message broker framework and in the future, this framework could potentially be utilized to take input from other, non-MQSeries sources. Consider, for example, the potential for an MQSI node that feeds a direct output stream from an input stream to an EJB.

Given this extensibility, it is always advisable to visit the MQSeries Family SupportPacs page at <http://www.ibm.com/software/ts/mqseries/txppacs/> for the latest information on supported product extensions.

For this redbook, we shall of course base our discussions around the use of MQSeries messages as the medium used for communication between the application server and the application router.

7.4.1.1 Message flow nodes

One of the powerful features of MQSI is the ability to define message flows as reusable objects. By defining a message flow using the InputTerminal and OutputTerminal primitives as its source of input and output, the message flow may be used over and over again as a node forming part of greater message flows.

Message flow nodes cannot be deployed to a broker, since they don't operate in a stand-alone fashion. They will only be used as a part of a greater message flow.

7.4.1.2 Message types

MQSI can be configured to accept a wide variety of message types. As a starting point, an MQSeries message with a straightforward string message body is acceptable to MQSI at an MQInput node.

However to be useful within MQSI, the message must be acceptable to one of its many available parsers. The sending application is responsible for including sufficient information for MQSI to be able to work with its messages. To do so, it is possible to include a secondary message header known as the MQRFH2 header at the beginning of the standard MQSeries message body, containing details MQSI can use to interpret the message body. In the absence of an MQRFH2 header, MQSI will attempt to select the most appropriate parser to interpret the message content.

Full details of the structure and options available in the MQRFH2 header may be found in the *MQSeries Integrator Programming Guide Version 2.0*, SC34-5603.

Message sets and the MRM

In MQSI Version 2, the Message Repository Manager (MRM) is a built-in graphical tool for defining structured data and message definitions for use in message flows.

Through definition of individual data elements that can be collated into compound data types, reusable data structures can be defined and used as the building blocks in the construction of complete message layouts, thus enabling rapid GUI-driven development of complex message structures. Such structures can be grouped into supersets known as message sets.

A particular strength of the MRM is its ability to create message set content by importing an existing defined message structure that is defined in C or COBOL.

The data structures defined in the MRM may be used in conjunction with more than one parser. This allows the same logical data structure to be used with different physical representations. The two main options are:

- Custom Wire Format (CWF), where the parser will accept or create a structured document where each field is defined with a given data type (and in the case of string data, length attributes)
- Extensible Markup Language (XML), where the parser will accept / create an XML document of the form described by the MRM structure. Note here that this is not the same as the generic XML parser we describe later.

Variable-length strings are supported, provided the length value is also supplied somewhere in the message structure.

When message definitions are defined in the MRM and saved as a message set, message flow node definition can take advantage of the drag-and-drop programming abilities of the Control Center.

While the MRM exhibits some strong features, some applications may find its use restricts the necessary flexibility in handling message field formats. For instance, in the MQSI 2.0.1 release, there is currently no support for tag-delimited data or variable-length strings that have no supporting length field.

NEON formatter

MQSI Version 2 provides the NEON rules and formatter facilities present in MQSI Version 1, in the form of primitive nodes. This has been done to provide a direct migration path for customers that have an existing implementation of MQSI version 1 product.

The NEON formatter is a powerful data format definition tool that provides similar functionality to that of message sets and the MRM. This tool is particularly strong when the need is to parse tag-delimited input.

XML

One of the greatest strengths of MQSI is its ability to parse and manage message data expressed in the form of XML documents.

Message structures defined in the MRM may be parsed as XML messages, but this does have inherent limitations, since the MRM has been designed principally to provide compatibility with legacy system data formats, which defeats the benefits that XML can provide in its flexible, self-defining representation of data.

For this reason, a generic XML document parser is available that is able to parse any well-formed XML message body. Although it is possible for an MQRFH2 header to be supplied to explicitly declare a message body as XML, this is not necessary as MQSI will automatically detect that an incoming message body is an XML message.

The pace of adoption of XML as the new standard for data representation has been dramatically high. XML is a truly portable medium for exchange of data, making it the natural choice for data representation in Web applications. For this reason, we have chosen to use generic XML message formats in our topology 5 example.

One current restriction within MQSI Version 2 is that transformation node programming based on generic XML documents cannot be carried out graphically alone. It is, however, perfectly possible to code such transformation logic in the MQSI node programming language, ESQL. This is in fact much easier than it may first appear.

7.4.1.3 Outputs

The end of the line for many message flows will be to output a new message that has in some way been derived from the original input.

The MQReply node is one way this can be handled. This node will take the transformed message body and put a message to the reply-to queue and queue manager name from the original message.

The MQOutput node will put the transformed message to the queue and queue manager or destination list specified in the MQOutput node properties.

7.4.2 IBM primitive nodes

At the beginning of any implementation, the MQSI Control Center makes available a set of fundamental nodes described as the IBM primitives. These are the basic building blocks that can be combined to make up a complete message flow.

Each primitive node has a set of input and output terminals that can be used to connect the outputs from a given node to the input of another. For example, an MQInput node is a node that can be configured to get a message from a nominated MQSeries queue. It has an out terminal which can be connected to the in terminal of a compute node - another type of configurable function that is designed to facilitate message transformation.

It is possible to set up a connection between a given out terminal and the in terminals of more than one node, as illustrated in Figure 46 on page 121. In

this way, one incoming message can be made to cause more than one flow of events. For instance, a message designed to carry a change of name and address details of a customer could be connected to several subflows, each of which is designed to update a different back-end system.

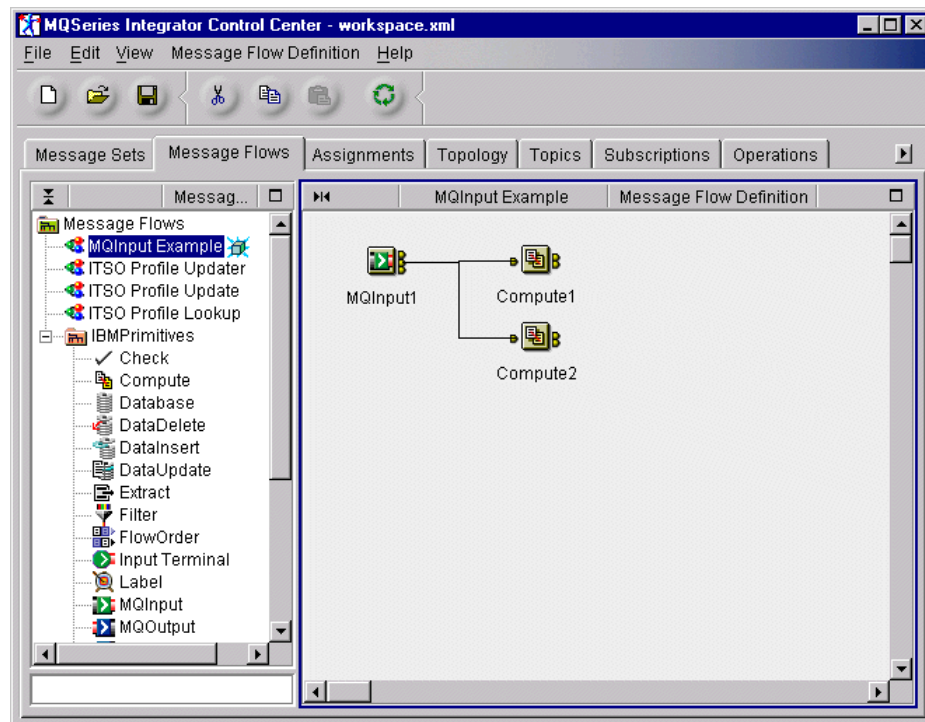


Figure 46. Multiple subflows

The full set of primitive nodes available in MQSeries Integrator is of course described in detail in the product documentation, and we shall not repeat this here. We shall, however, highlight certain features we have found useful in building an example to illustrate topology 5.

7.4.3 Transformation nodes

Transformation nodes perform the task of taking the message and changing it as required.

7.4.3.1 The compute node

The compute node is one of the most versatile transformation nodes available from the palette. It is designed to create an output message. In order to do so, it may take more than one input data source. Obviously the main input source

is the incoming message via the node's input terminal. The node may optionally refer to a database as part of its function.

A compute node has one input terminal through which an input message is received, and two output terminals, one to handle output under normal successful operation and the other to redirect the original message in the event of failure of the transformation process.

The compute node may be configured through its Properties dialog, shown in Figure 47.

Let's look at the layout of the Properties dialog. The main dialog is divided into two sections, *Inputs* and *Output Messages*. The simplest operation a compute node can perform is to copy the input message to the output message. This can be achieved simply by checking the Copy entire message radio button. Checking the Copy message headers radio button automatically generates code that will copy all parts of the input message prior to the message body to output.

The code generated to perform these tasks is displayed in the ESQL tab.

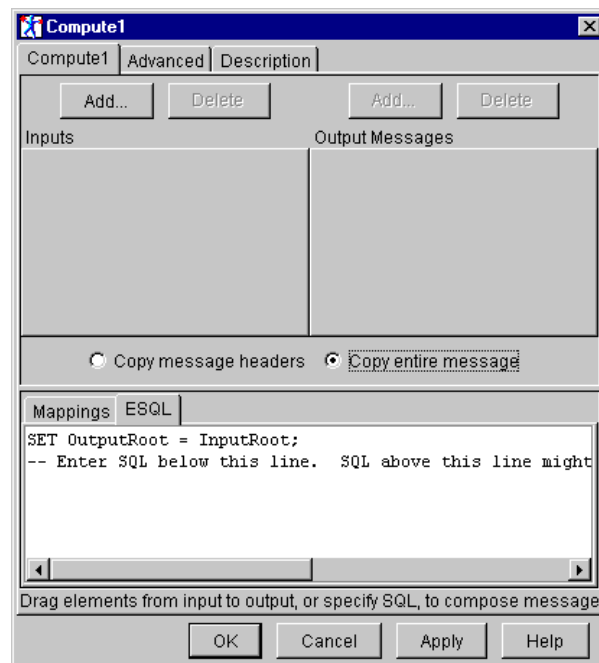


Figure 47. Compute node properties

In each case, a comment line is generated that recommends that any custom code you may add to the node be added below the comment line. This is done because the transformation details may be constructed using the GUI. The effect of these actions will impact the ESQL code that is contained in the node properties. Automatically generated code remains above the line; user code may be entered below the line.

In using the Control Center to configure compute nodes, you will soon notice the limited size of the ESQL pane, which can make it difficult to read the code that lies behind the node. One technique to make working with ESQL easier is to use Ctrl+a to select all the code behind a node, Ctrl+c to copy the selection and Ctrl+v to paste the contents into a text editor such as the Windows Notepad to perform the editing. When code changes are complete, the code can be copied/pasted back to the ESQL pane.

One of the features of the ESQL pane is its dynamic validation of the code syntax as you type. If you choose to edit code in this pane, a red “X” will appear below the code pane if there are any syntax errors in the code. This is useful when you try new techniques or combinations of functions within the structure of your code.

7.4.3.2 Using ESQL for message transformation

ESQL is a powerful data transformation language available to many of the MQSI transformation nodes including the compute node. It is capable of manipulating structured data within a message in the same way as performing regular database operations in an SQL fashion. In this way, data may be freely interchanged between the message content and database data sources as part of the Compute operation.

The *Root* element is the highest level element of a message structure. Since the compute node has both input and output message terminals, these elements are referred to as *InputRoot* and *OutputRoot* respectively. The elements that make up the whole document are held as child elements of the Root element, and are referred to in the following manner:

```
InputRoot.element1  
InputRoot.element2
```

The following is an example of an MQSeries XML message:

```
<Trade type='buy'  
Company='IBM'  
Price='2 .2 '  
Date='2 -1-1'  
Quantity='1 ' />
```

It has a structure that takes the following form:

```
Root
  Properties
    CreationTime=GMTTIMESTAMP '1999-11-24 13:1 : '
      (a GMT timestamp field)
      ..and other fields ...
  MQMD
    PutDate=DATE '19991124'
      (a date field)
    PutTime=GMTTIME '131 '
      (a GMTTIME field)
      ...and other fields ...
  MQRFH
    mcd
    msd='xml '
      (a character string field)
      ..and other fields ...
  XML
    Trade
      type='buy'
        (a character string field)
      Company='IBM'
        (a character string field)
      Price='2 '
        (a character string field)
      Date='2 -1-1'
        (a character string field)
      Quantity='1 '
        (a character string field)
```

Here we can see the various substructures found within a message:

- Generic message properties
- The MQSeries Message Descriptor (MQMD)
- The MQSeries Integrator Rules and Formatting Header (MQRFH2)
- The message body itself, in this case an XML document

To refer to the Trade data element of an input message, we can use the ESQL expression:

```
InputRoot.XML.Trade
```

or

```
InputRoot.XML.(XML.tag)Trade
```

The latter shows an example where the Trade node is fully qualified as an XML tagged element. Although in the above example it is not strictly

necessary to use this fully qualified notation, it is good practice to include the (XML.tag) qualifier, because in a well-formed XML document with an XML document header, the parser may misinterpret your XML message body tag references as being references to attributes of the XML document header.

The above does little more than introduce the basic shape of data element references in ESQL. The above example message has been taken from *MQSeries Integrator Using the Control Center Version 2 Release 0*, SC34-5602, which has a comprehensive ESQL reference appendix. The manual covers the full set of valid constructs you can make in ESQL, which we will not repeat here. Instead, we shall continue by focusing on the techniques used in building our example scenario.

Example: copying elements from input to output

Consider the following code extract:

```
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."userid" =
  "InputBody".(XML.tag)"customerrequest"."userid";
```

Here we see two ESQL code statements over three lines. To enhance readability, statements can wrap to as many lines as you like, but each statement must end in a semicolon.

The first line of code copies the XML type declaration element from input to output. The second line copies the “userid” child element of the input “customerrequest” XML document element of the “userid” child element of the output “profilemessage” XML document element.

In this kind of assignment operation, the referenced elements may be leaf elements (that is, they have no child elements themselves) or may contain substructures of any number of levels of child elements. Whichever applies, the assignment statement will copy the full element from input location to output location inclusive of any child elements that may exist.

Example: more complex tasks

Now consider this example:

```
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr" = THE
(
  SELECT
    T."HOUSENO" AS "houenum",
    T."HOUSENAME" AS "houename",
    T."STREET" AS "street",
    T."DISTRICT" AS "district",
    T."CITY" AS "city",
```

```

T."STATE" AS "state",
T."COUNTRY" AS "country",
T."ZIP" AS "zip",
T."TELEPHONE" AS "telephone"
FROM InputBody.(XML.tag) "customerrequest"."savingsqueryresult" AS T
);

```

This introduces a number of concepts.

ESQL may be used to extract lists of data elements from structured messages similar to the way that one uses SQL to query a relational database, such as DB2. In the above example, the target of the select query is the compound data element “savingsqueryresult” which is itself a child element of the “customerrequest” XML document. The SELECT statement assigns the shortname of T to this element, and using this, it extracts the fields HOUSENO, HOUSENAME, STREET, DISTRICT, CITY, STATE, COUNTRY, ZIP and TELEPHONE from its child elements, substituting the values found to new child elements of “housenum”, “housename”, “street”, “district”, “city”, “state”, “country”, “zip” and “telephone” within the “custaddr” element of the output XML document “profilemessage”.

We can also see use of the THE predicate. The inclusion of this is an indication that only one row is expected (and to be taken) from the result of the query. Omission of the THE predicate indicates more than one row may result from the query, and so the assignee of the SET statement may be expected to have multiple occurrences. Where this is the case, the assignee (in this case “custaddr”) should carry the array [] suffix to indicate more than one instance of it may be created as a result of the operation.

Note in each of the above examples it is common good practice to wrap all data element names within “double quotes”. This approach will avoid any potential confusion that may arise in using special characters in element names.

If assigning literal string values in code, these should be represented within ‘single quotes’.

7.4.3.3 Using database input

In addition to the input message, input from a data source may be taken by giving a data source name (ODBC database name) and the table name.

Compute nodes can include any basic, valid SQL operation as part of their function in the form of ESQL queries to the content of a database. Consider the example code that follows:

```

SET "OutputRoot"."XML".(XML.tag) "customerrequest"."profilequeryresult" [] =
(
  SELECT T.*
  FROM Database.ITSO_CUSTOMER AS T
  WHERE T.USERID = TRIM("InputBody".(XML.tag) "customerrequest"."userid")
);

```

This is a simple ESQL assignment statement that will select all columns from the ITSO_CUSTOMER table of our database, and extract all rows where the USERID column contains the same value as the “userid” element from the input “customerrequest” XML document. As a precaution, the input document element is subjected to a TRIM function before it’s used as the selection criteria of the query. This will eliminate any potential problem with leading or trailing blanks in the supplied “userid” value.

Note that since more than one row may be returned, the values selected are loaded to a repeating element described as “profilequeryresult”[]. Due to the use of “*” in the selection criteria, in the output document the new child elements will be assigned the same names as the column names found in the ITSO_CUSTOMER table. Note that we have not specifically declared these to the compute node at any point.

7.4.4 Database nodes

At points where no message transformation is necessary, but database operations are required, a better alternative to using a compute node is to use a database node. In a compute node, ESQL can be included to select, update, or delete entries in the database data source.

MQSI contains a number of specialized database nodes that are suited to each of the key SQL operations, the DataInsert, DataUpdate, and DataDelete nodes. These are most useful when working with message data that can be manipulated graphically, but above this they have no added benefit.

In our examples, we have used the generic database node for all database operations. The example message flow shown in Figure 48 includes two database nodes, the Update Checking Profile node and the Update Savings Profile Node. These database nodes are constructed to take values from the message data that passes through the node, and perform a database update based on those values.

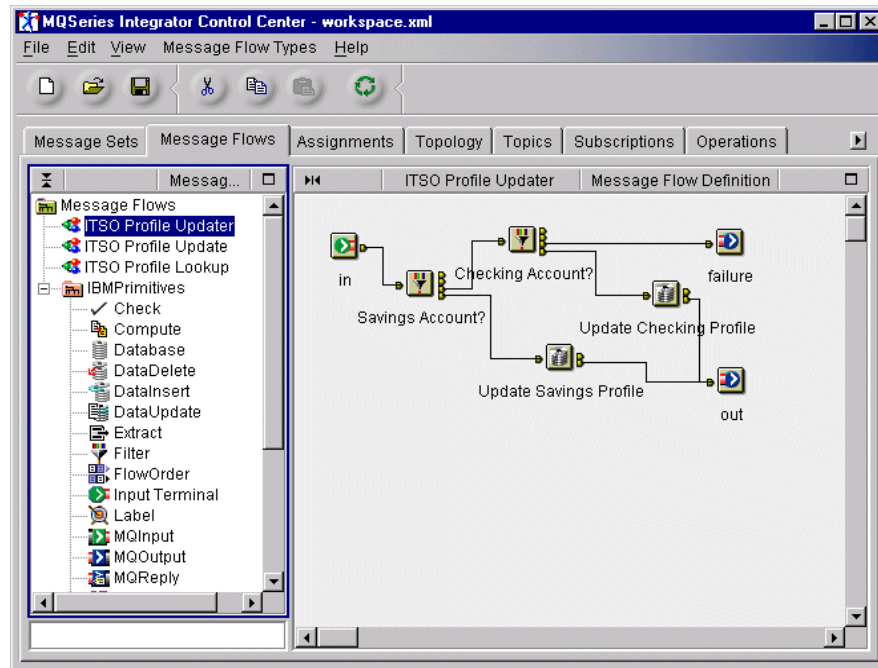


Figure 48. Example message flow incorporating database nodes

Let's take a close look at the Update Checking Profile database node.

The 'Update Checking Profile' dialog box is shown with the 'Advanced' tab selected. It contains fields for 'Input' and 'Output'. The 'Output' field is set to 'ITSO_CHECKING'. Below it, 'Transaction Mode' is set to 'Automa...'. A tree view shows the database structure: 'ITSO_CHEC' containing 'ITSO_CHECKING', which contains 'NAME' and 'ADDRESS1'. The 'UPDATE' statement is displayed in the bottom text area:

```
UPDATE Database.ITSO_CHECKING AS T
SET
  NAME = SUBSTRING(
    "Body".(XML.tag)"updatemessage"."profile"."cu
    "Body".(XML.tag)"updatemessage"."profile"."cu"
```

Buttons for 'Add...', 'Delete', 'OK', 'Cancel', 'Apply', and 'Help' are present.

Figure 49. Update Checking Profile database node

The complete ESQL code is shown in Figure 50. Note the use of SUBSTRING functions and concatenation || operators that are used to combine several input elements into one column value, but protect the column from the potential for update using a value that exceeds its maximum string length value.

```
UPDATE Database.ITSO_CHECKING AS T
SET
  NAME = SUBSTRING (
    "Body".(XML.tag) "updatemessage"."profile"."custname"."forename" || ' ' ||
    "Body".(XML.tag) "updatemessage"."profile"."custname"."middlename" || ' ' ||
    "Body".(XML.tag) "updatemessage"."profile"."custname"."surname"
    FROM 1 FOR 40),
  ADDRESS1 = SUBSTRING (
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"." housename" ||
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"." housenum" || ', ' ||
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."street"
    FROM 1 FOR 40),
  ADDRESS2 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."district",
  ADDRESS3 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."city",
  ADDRESS4 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."state",
  ADDRESS5 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."zip",
  TELEPHONE = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."telephone"
WHERE T."NUMBER" = "Body".(XML.tag) "updatemessage"."account"."ACCOUNT";
```

Figure 50. ESQL for the database node

7.4.5 Logic control nodes

Several nodes are available to insert logic into the message flow.

7.4.5.1 The filter node

In a message flow, it is useful to control the flow direction according to a condition that exists at execution time. For this purpose we use a filter node.

You can use ESQL to construct an expression that is evaluated to a true or false value. Control is propagated to the applicable output terminal based on the outcome of resolving your expression.

Consider the message flow shown in Figure 51. You will notice there are four possible output terminals to a filter node. In addition to output on a true or false condition, the expression may be “unknown”, or a failure.

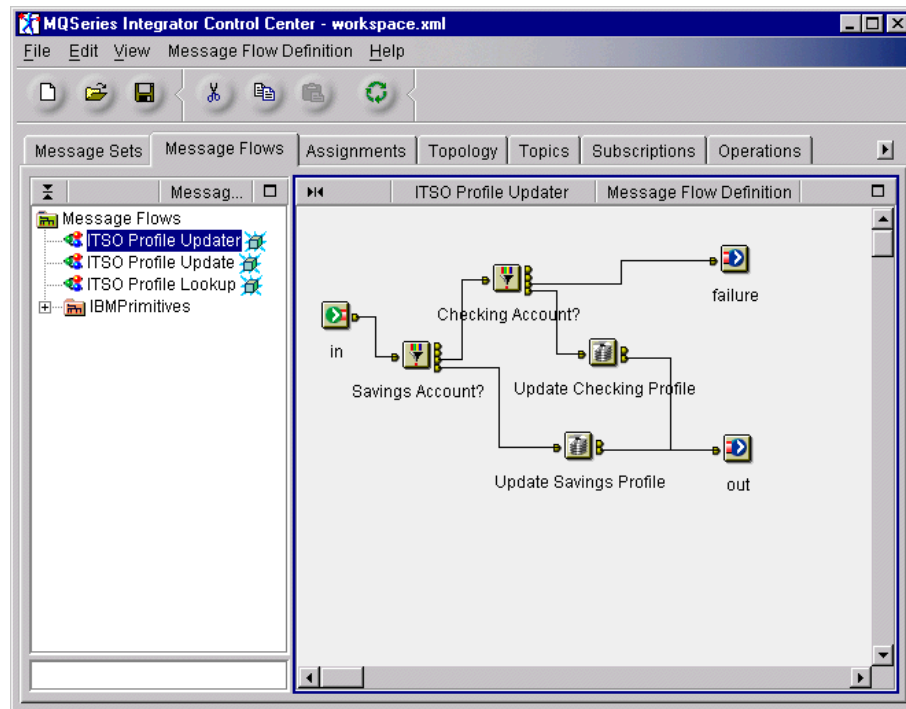


Figure 51. Example message flow using a filter node

In Figure 52, we can see the properties of the filter node. The ESQL expression describes an element from the message that is input to the node. The value is tested for a value of "S". If this condition is true, control is propagated to the True output terminal.

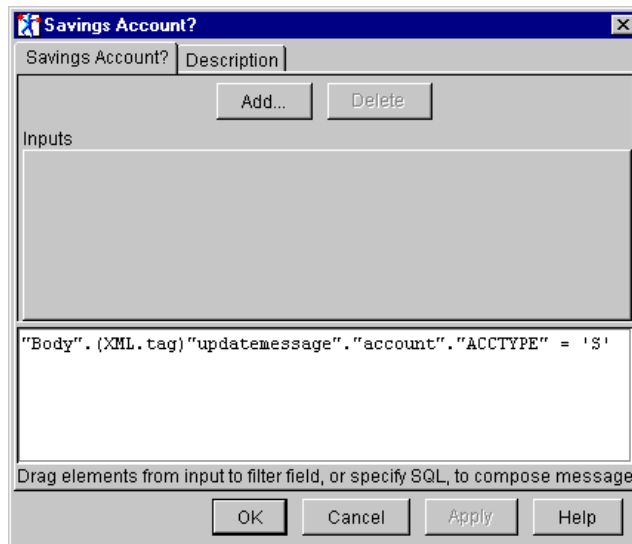


Figure 52. Filter node properties

You can add further input to the filter node, allowing you to add database tables as a data source for use in the expression. In this way you can use incoming data elements to look up values from a database, which can be tested in the expression.

7.4.5.2 Looping

A database lookup that retrieves many rows can add a very large repeating element to a message. This is often performed in order to take a set of elements and make repeated entries to a subflow for each element in the set.

This requires a loop in a message flow that counts the number of entries in the list and works through them, creating a separate output message at each iteration of the loop. Take a look at the example message flow shown in Figure 53.

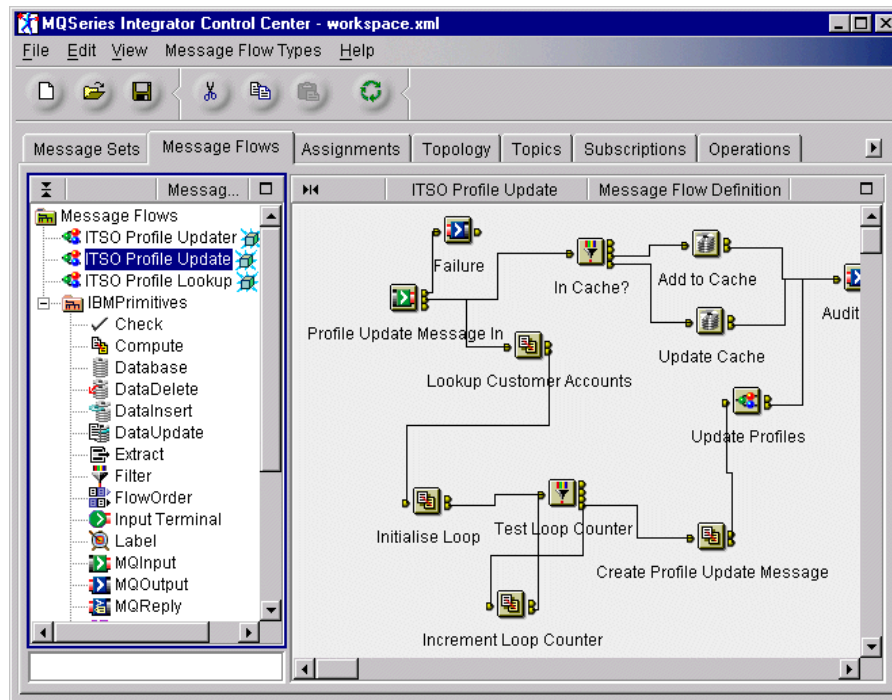


Figure 53. Example message flow showing a loop

The message is prepared by the compute node named *Lookup Customer Accounts*. This node contains ESQL that adds a repeating element based on the output of a database operation as follows:

```
SET "OutputRoot".XML.(XML.tag)"groupupdatemessage"."accounts" [] =
(SELECT T.* FROM Database.ITSO_CUSTOMER_ACCOUNTS AS T WHERE T.CUSTOMER =
"InputBody".(XML.tag)"profilemessage"."userid");
```

This populates the repeating data element “accounts” with the rows found in table ITSO_CUSTOMER_ACCOUNTS.

Looping starts with the *Initialise Loop* compute node. This contains the following ESQL.

```
SET OutputRoot = InputRoot;
-- Enter SQL below this line. SQL above this line might be regenerated,
causing any modifications to be lost.
SET OutputRoot.XML.(XML.tag)"Loop"."Count" =
CARDINALITY("InputBody".(XML.tag)"groupupdatemessage"."accounts" []);
SET OutputRoot.XML.(XML.tag)"Loop"."Index" = 1;
```

As you can see, the message is first copied from input to output in an unchanged form. The code adds a new data element to the message, named “Loop”. The choice of element name is arbitrary. We have used “Loop” so we can hold any data pertinent to loop control within the scope of this element, and recognize it as such. Two child elements are added to “Loop”. These are

- “Count”. This is initialized to the number of entries we have in our list. The CARDINALITY function is used to return the number of data elements in the set.
- “Index”. This is initialized to 1, to indicate looping is to start at the beginning of the list.

Here again, the choice of element names is arbitrary. By keeping the names relevant to their use, this aids the readability of the code.

Loop control is handled using a filter node, shown in Figure 54.

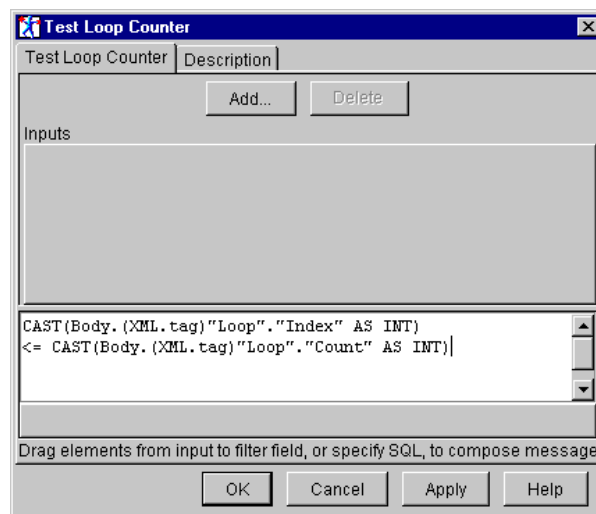


Figure 54. Example loop condition test

Here we can see that the resolved condition will be True for as long as the “Index” does not exceed the “Count”. You will see use of the CAST function. This is particularly important when working with documents in XML, since all data originates as a string. When working with numbers, CAST may be used to force interpretation of the value as a numeric data type such as INT.

Other than maintain a check on loop iterations, the filter node simply propagates the message in full. You will see from Figure 53 on page 132 that the output is propagated twice, first as output to *Create Profile Update*

Message and second to the *Increment Loop Counter* compute node shown in Figure 55.

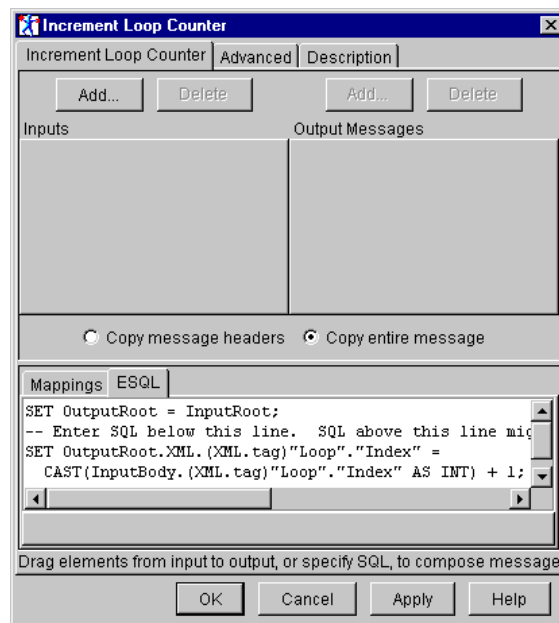


Figure 55. Incrementing a loop counter

This compute node simply copies the message, but also increments the "Index" element of the "Loop" construct. The output from this node is piped back to the *in* terminal of the *Test Loop Counter* filter node which repeats its operation for this new instance of the message.

You will see from the above that this technique can be used to cause many messages to be propagated to the *Create Profile Update Message* node, from a single message that was output from the *Lookup Customer Accounts* node.

However, we have not finished the job. Every instance of the message sent to *Create Profile Update Message* is identical, save the content of the Loop construct. This compute node will use the value of "Loop"."Index" value to select the element it requires from the repeating list. Take a look at this ESQL code taken from *Create Profile Update Message*.

```

DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag) "updatemessage"."account" =
InputBody.(XML.tag) "groupupdatemessage"."accounts" [CAST (InputBody.(XML.
tag) "Loop"."Index" AS INT)];
SET "OutputRoot"."XML".(XML.tag) "updatemessage"."profile" =
"InputBody".(XML.tag) "groupupdatemessage"."profile";

```

Figure 56. ESQL from Create Profile Update Message node

Using the value found in the line `InputBody.(XML.tag) "Loop"."Index"`, the required element from the repeating group in the input message is added as a single element to the output document.

Here we should issue a warning: clearly this technique makes it possible to initiate a single message flow operation that comprises a large number of individual node-processes, which may exceed the system's capacity in one flow execution. Where larger result sets are expected, you may wish to consider limiting the number of iterations in one message flow execution, and passing the remaining workload out as a message for input to a further execution of the message flow.

7.4.6 Reusable message flows

Common operations can be built as reusable message flow components that may be used as nodes of a greater message flow. Figure 57 shows an example of how the updates to a set of database tables could be packaged as a message flow for incorporation in other message flows where needed. The message flow defined here is saved as a message flow node called ITSO Profile Updater.

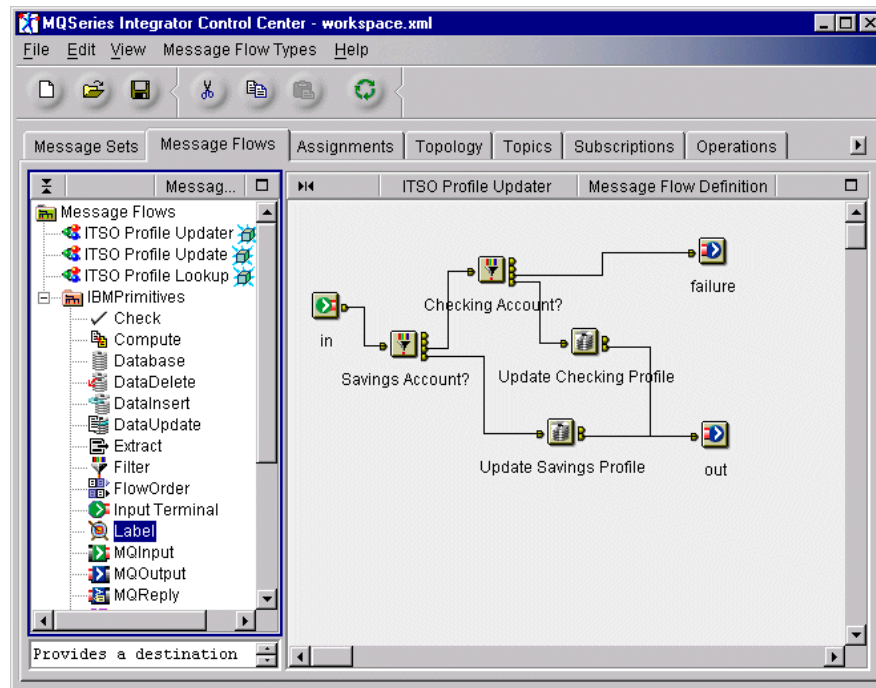


Figure 57. Reusable message flow - ITSO Profile Updater

The inputs and outputs are defined using terminal node types InputTerminal and OutputTerminal. The remainder of the message flow is defined in the same way as a deployable message flow. From this statement, you can infer that component message flows are not deployable in isolation, but only as part of a deployable message flow.

When this message flow is saved, it can be dragged into another message flow, where it will be displayed as a message flow node. You can see an example of this in Figure 53 on page 132, where the node named Update Profiles is an instance of the ITSO Profile Updater component message flow defined above.

Clearly, use of a message flow node requires that its input is prepared in accordance with its specification. In the case of the ITSO Profile Updater flow, it requires an XML document of type "updatemessage", the content of which matches that which has been prepared by the Create Profile Update Message compute node of the ITSO Profile Update message flow.

7.4.6.1 Using property promotion in message flow nodes

Saving commonly used operations as message flow nodes is a useful method that enables reuse of code. An extension of this feature is the ability to make message flow nodes configurable through property promotion.

Consider an adaptation of the loop scenario described earlier, shown in Figure 58. The ITSO Looper message flow node takes the loop scenario from “Looping” on page 131, and builds a general-purpose version of it for re-use. In the earlier example, the loop was built from individual nodes, each of which was configured with ESQL code specific to the application.

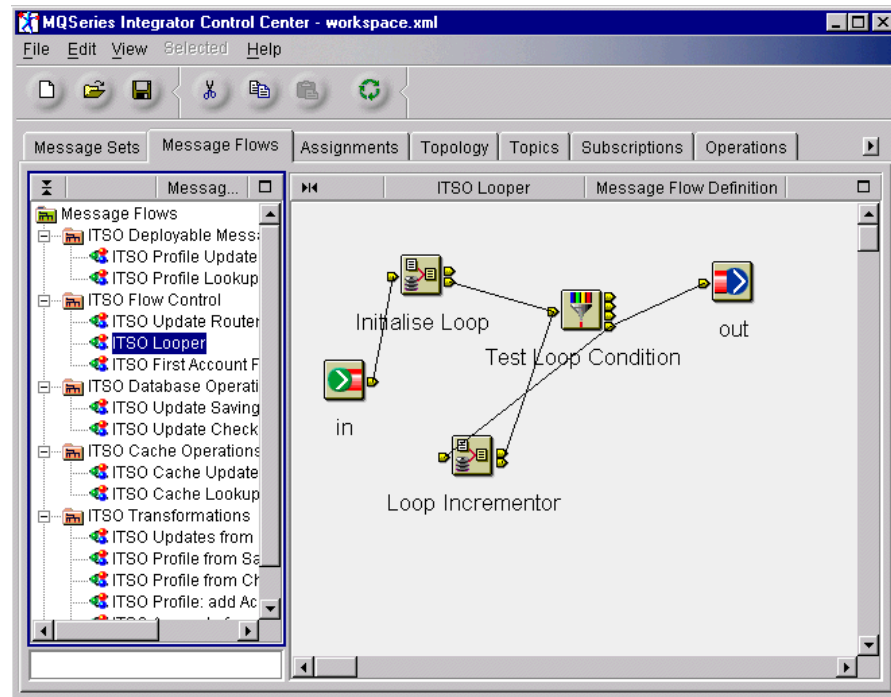


Figure 58. The ITSO Looper message flow node

In the ITSO Looper message flow node, we provide the basic construction of the loop within the message flow itself, and promote those properties that are involved in re-use of the message flow, so these may be configured in each re-use scenario. The properties that affect loop operation are:

- The ESQL code from the *Loop Initialization* compute node
- The ESQL expression that controls loop execution from the *Test Loop Condition* filter node

- The ESQL code that is executed in the *Loop Incrementor* compute node

The Promote Property dialog (see Figure 59) allows you to select from the properties available for promotion and add them to the set of properties that will be made available for configuration when the message flow node is used. Here, we see how the properties listed above will be presented to the user of the ITSO Looper message flow node.

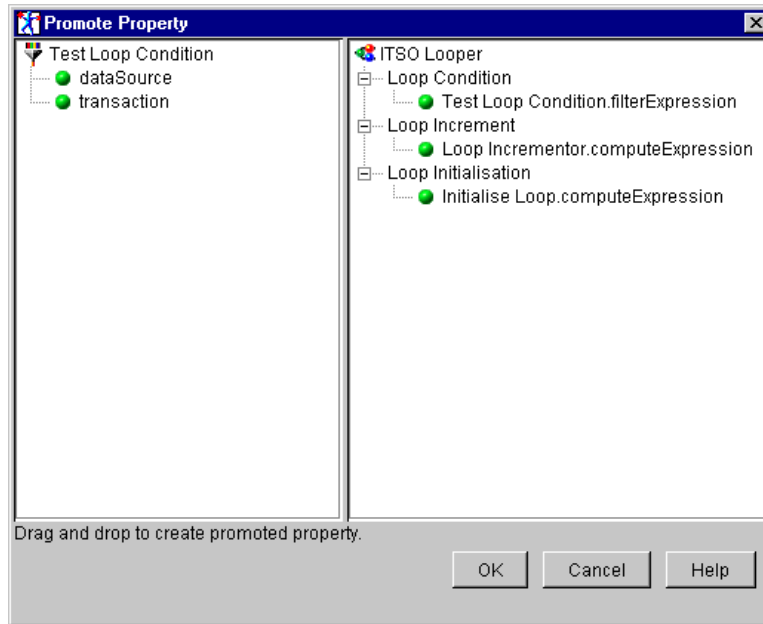


Figure 59. ITSO Looper promoted properties

Figure 60 shows the properties dialog that gets displayed when the ITSO Looper message flow node is configured as part of a message flow.

Using this technique, it is also possible to configure promoted properties so that a value entered once as a message flow node property may be utilized in more than one of its component nodes.

Create Update for Each Account	
	Description
Loop Condition	CAST(Body.(XML.tag)"Loop"."Index" AS INT) <= CAST(Body.(XML.tag)"Loop"."Count" AS INT)
Loop Increment	SET OutputRoot = InputRoot; SET OutputRoot.XML.(XML.tag)"Loop"."Index" = CAST(InputBody.(XML.tag)"Loop"."Index" AS INT) + 1;
Loop Initialisation	SET OutputRoot=InputRoot; SET OutputRoot.XML.(XML.tag)"Loop"."Count" = CARDINALITY("Input SET OutputRoot.XML.(XML.tag)"Loop"."Index" = 1;
MessageProcessingNodeType	ITSO Looper

OK Cancel Apply Help

Figure 60. ITSO Looper message flow node configuration

7.4.7 Testing message flow components

By creating complete message flows as a set of smaller and simpler components, the testing process becomes a less arduous task.

Each subflow can be tested in isolation, through use of a test harness like the example shown in Figure 61 on page 140. By connecting the message flow node into a harness like this, you can use a simple MQSeries application to put a message of the required format to the input queue, and monitor the result using a simple application that gets a message.

Where a message flow component is designed with more than one terminal in or out, a variation of this method will be needed.

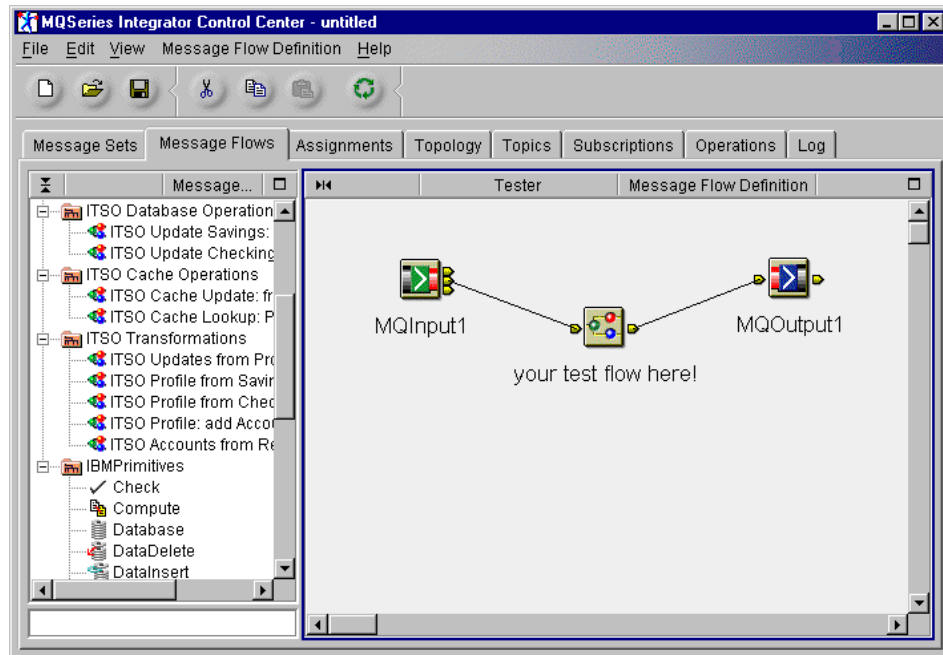


Figure 61. Example test harness

For our testing, we used a simple adaptation of the MQSample application that is supplied with MQSeries SupportPac MA88. Where database operations are expected, we used the DB2 Command Center to run test queries against database contents.

Chapter 8. Application development guidelines

Now that you are familiar with the application design considerations, it is time to get down to business and develop the application. We define the stages of application development as consisting of the following stages:

- Solution outline
- Macro design
- Micro design
- Build cycle
- Deployment model



Figure 62. Development process overview

In the solution outline phase you decide the scope of the project, explore what the essential business needs are, come up with an idea of the base architecture, and get the commitment from the project sponsor to start.

Then you start with the macro design, which concentrates on the detailed requirements gathering, business process modelling, high-level analysis and design, the base architecture, and a plan for the following development phases, including a development release plan. The solution outline and macro design phases are usually done once in a project.

It is likely that there will be multiple releases of an application. New releases are usually required to add new function, add maintenance, or to improve processes. The rest of the phases of development are completed for each release of the application.

The micro design focuses on transforming the business model into a design model by taking the selected use cases and running them through a typical object-oriented development phase. Transforming means that we use the business model to bring it to such a technically detailed level that it can be implemented. This is done by adding all the architecture and implementation-specific classes and components to the existing business model.

In the build cycle the results of the micro design are turned into code:

- Write and unit test the source code.
- Build the executable code if necessary, for example, all Java code.
- Perform various tests on the executable code.
- Test the application in a runtime environment.
- Prepare for deployment.

The incremental approach used to run the release cycles is also used for the different activities of the build cycle. It is run in several iterations for one release, with each iteration transforming more of the design into tested executable code that is ready to be deployed.

The last step is to create and execute a deployment plan that encompasses not only when and how to install and set up the newly developed application, but it must include all hardware and prerequisite software requirements. The deployment plan should also include plans for system management, taking into consideration what has to be managed and how, how to establish the required security, and what has to be done for availability and recovery.

8.1 The scope of this book

As with the design guidelines, we will be building on to the information presented in *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864, where application development guidelines for e-business Java applications were introduced. We will not cover the development cycle processes again here since they have not changed. Instead, we will expand on those guidelines by using our WebBank application to illustrate two new concepts.

First, our WebBank application will use the command package introduced in WebSphere Application Server Advanced Edition 3.5 and VisualAge for Java 3.5.

Second, our application will use MQSeries and MQSI to perform functions that in previous U2B applications were done with Java classes. This presents an interesting challenge to the designer using such classic design tools as Rational Rose.

8.2 Application development tools

There are many good application development tools available on the market today and choosing the right tool for your enterprise will depend on many things. We have chosen to use the following for application development:

- Rational Rose for analysis and design of our application
- VisualAge for Java for developing the Java code
- WebSphere Studio to create the HTML and JavaServer Pages
- MQSeries Control Center to develop the MQSI flows

For more information on how to use VisualAge for Java and WebSphere Studio to build and deploy the application code, see *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755.

Developing the MQSI flows using the MQSeries Control Center is discussed in detail in Chapter 9, “Developing the MQSI application” on page 183.

8.2.1 Rational Rose

Rational Rose is a visual modeling tool product by Rational Software Corporation (<http://www.rational.com>). It is based on the Unified Modelling Language (UML) .

Building an e-business solution requires careful and detailed design before the actual code can be developed. Rational Rose provides visual design and analysis capabilities. Starting with only a concept of what the application should do, you can go through a logical progression of design activities to build a detailed model of the application and generate the initial code.

Rational Rose provides the ability to:

- Capture user and business requirements, and present them in a common format.
- Identify and design business objects.
- Transpose business objects to software components.
- Design the distribution and communications of the components across an enterprise.
- Forward engineer Java code directly from the model.
- Reverse engineer code from existing components and applications into Rose models.
- Round-tripping facilities to keep the model and code synchronized.

Rose 2000e V2.0 provides a seamless integration between Rose and VisualAge for Java, including full support of forward engineering Rose models into the VisualAge for Java workspace as well as reverse engineering from the VAJ environment into Rose. Prior releases of Rose required the XMI Toolkit to provide this integration.

8.2.2 VisualAge for Java

VisualAge for Java provides extensive functionality across the entire development life cycle and includes tools for Java code editing and debugging, JavaServer Page debugging, and the WebSphere Test Environment. VisualAge for Java uses a repository to store project source and compiled code, and an import/export facility that enables interaction with the file system.

One of the most important features of VisualAge for Java is the WebSphere Test Environment. This feature provides application and Web server environments on a development machine, enabling you to test and debug the resources of a Web site locally. This environment provides much of the functionality of a full application server, including access to services such as LDAP and enterprise resources.

8.2.3 WebSphere Studio

WebSphere Studio is used to develop, manage and deploy the resources for a Web site. It maintains project files in a file system and provides support for team development and version control tools. The deployment features of WebSphere Studio enable you to configure the projects to deploy to a number of locations, such as the WebSphere Application Server or the WebSphere Test Environment of VisualAge for Java.

WebSphere Studio also contains a number of wizards that guide you through tasks such as SQL statement generation and creation of Web pages to interact with databases and Java beans. You can also use the WebSphere Studio Page Designer to edit these generated pages, or create your own HTML and JSP pages.

Any Java source code within WebSphere Studio can be compiled using the supplied Java compiler.

8.2.4 How these tools fit together

These three development tools work nicely together to give you a comprehensive development environment.

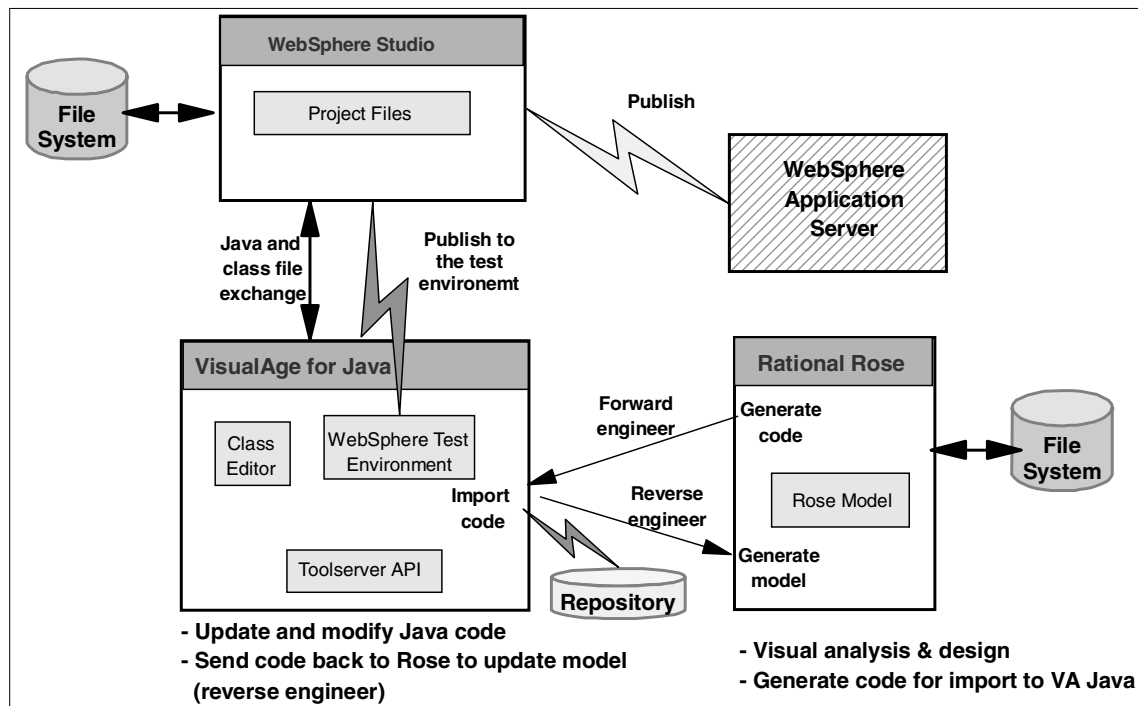


Figure 63. Development tool interaction

In the micro design phase, Rational Rose is used to capture the design model. The initial code for the servlets, Java beans, and other Java classes in this model can be generated by Rose and then be imported into VisualAge for Java (forward engineered). The link between Rose and VisualAge for Java is discussed in Appendix A, “Rational Rose 2000e and VisualAge for Java” on page 381.

Development of the code is then continued in VisualAge for Java, producing the required Java classes. Any changes that would alter the original design model can be reverse engineered back to the Rose model. The class files are imported to WebSphere Studio for management and publishing.

WebSphere Studio is used to develop the view element (HTML and JSP) portions of the code and to manage and publish code. It also interacts with VisualAge for Java, to exchange source and class files. Imported classes from VisualAge for Java are extracted from the repository, converted into files, and stored in the file system structure used by WebSphere Studio. They can subsequently be published, possibly back to the VisualAge for Java WebSphere Test Environment for testing.

8.3 WebBank problem domain

The example chosen to demonstrate the development issues in this chapter is that of a fictitious bank, WebBank. WebBank wishes to provide a number of services over the Internet for its existing customer base and new customers. They have decided to implement their expansion into the Web in stages, the first stage being addressed by our sample application.

The first banking service they have chosen to open up to customers on the Web is a personal profile service. This service allows customers to review and update their own personal profiles at the bank. A profile includes information such as contact address and telephone numbers.

To implement the personal profile service, WebBank would like an application set that leverages existing investments in its MQSeries and MQSeries Integrator middleware infrastructure. The goal is to prevent any major adaptation or rewriting of existing legacy applications and databases.

8.4 Solution outline

The first phase of a development project is the startup. This phase normally begins with a small team of domain experts, analysts and IT architects exploring the requirements for the new solution. Beyond the pure business requirements, it is important to explore the existing environment to find out how the new application can fit. The target audience for the solution has to be named and their experience has to be determined.

Based on this initial information an architecture for the application has to be determined. The team has to decide about the overall strategy of the solution that will drive the whole project, based on the business impact the solution has on the organization. The architectural decisions made are a separate work product and should be well documented.

Because this problem domain is centered around banking customers (users) interacting with enterprise transactions and data (banking applications and data), the appropriate business pattern to use is the User-to-Business Pattern.

In this application, customers will be allowed to view and update their customer profile. Customer requests need to be routed to the appropriate back-end systems with the result communicated back to the customer. This routing mechanism can provide a common interface to the back-end systems, thus simplifying the connectivity options and reducing the cost of access to

enterprise data. This suggests topology 5 as described Chapter 2, “Choosing the application topology” on page 11 as the appropriate choice.

It is also evident that in order to prevent constantly traversing multiple back-end systems to retrieve commonly used data, some sort of caching system at the router node needs to be employed. This will improve performance and further reduce the cost of access.

The outcome of the solution outline feeds into the next phase, which is the macro design.

8.5 Macro design

In the macro design phase the project team is usually extended from the few domain experts, analysts and IT architects working in the solution outline phase to a broader skill set. They will refine the requirements identified in the solution outline, going into more detail in several design areas.

8.5.1 Creating a business process model

The macro design phase should produce a business process model that defines what the application needs to accomplish. This is generally done by identifying the key use cases. In our account services application we have identified the following as the key use cases:

- **Login** - This is the same login mechanism used for the Account Services. It is designed to prevent unauthorized access to the customer profiles and the banking data.
- **Update Profile**: This retrieves the customer profile from cache and allows the customer to update it. The updates are made to the cache and to the back-end systems.
- **Sign Off**: The customer logs off the system and all session information is cleared.

8.5.2 Information architecture

An information architecture will need to be developed that defines what data needs to be accessed and how. The information architecture is a key component in any e-business architecture. Its focus is on improving the clarity and functionality of a Web site or Web application. It is not just concerned with the design of the user interface. The fundamental questions it addresses are in four areas, namely presentation, organization, navigation and adaptability.

8.5.2.1 Presentation

This addresses the issues around how the information is to be conveyed; what words will be used, charts, illustrations, rich multimedia, streaming technologies, etc. There are a number of issues that, when addressed, will help with the presentation of information.

Who are the users?

Fundamental to addressing this issue is the need to understand who the users of the site are. By identifying the target audience, content can be customized and targeted for that audience, therefore enhancing overall site experience. Taking this a step further, personalization opens the door to countless possibilities for e-commerce, e-CRM, Web marketing and other areas of application. By knowing the audience, it may also be possible to assume their familiarity with Web technologies and the channels of delivery, thereby putting in more advanced features.

What are the channels?

Another concern is based on what channels are used to access the site. Are they PC browser based, PDAs, WAP phones, iTV or maybe a kiosk? Knowing this allows the content to be tailored for the specific environment. For example, compare the relative richness of a PC-based Web browser to that of a WAP phone. In the first instance the real estate is large enough to hold not only the items of interest but also a very rich set of supporting content, using the latest multimedia technologies, any number of navigation mechanisms and advertising features as well. There is a standard keyboard and mouse to take user input, from simple point-and-click dialog boxes to rich-text areas to handle copious amounts of typed or pasted information. The WAP phone offers few luxuries by comparison. The screen can handle a few lines of text and navigation to other areas and that's probably the limit. Eliciting information is probably best done by a series of predetermined lists, requiring the use of up, down and select functions, limiting the need to use the phone keypad to type information, an altogether cumbersome and painful experience for anything more than one or two words.

What are the technology constraints?

Having identified the channels, technology constraints within each channel also need to be accommodated. Consider again a Web site accessed from a PC-based browser. Will the site support all Version 3.x and 4.x browsers or only the latest versions from particular vendors? If the Web site is a corporate intranet, it's more than likely that there would be a standard supported version of a browser. However, as we consider extranets, e-Marketplaces, and full-access consumer Web sites, we can see the proliferation of browser types and versions that customers may use. This adds a layer of technical

complexity as developers need to accommodate the different technologies supported by browsers, as well as understanding the different interpretation of what is standard by vendor-specific browsers. This is a significant consideration when estimating the development and testing effort. Obviously writing for one browser will take less time and effort than writing for multiple browsers.

How will it perform?

What are the users expectations of response time and what can you deliver? This will depend largely on knowing where the users are. Are the majority accessing the site from a dial-up line or from a corporate network? If it is slow access, then what are the limitations on page sizes? Consider the fact that on a 28.8Kbps modem, with no noise or interference on the line, it takes a 10KB page approximately three seconds to download. Taking these kinds of metrics into consideration has a profound effect on just how much information is displayed per page and how it is presented.

8.5.2.2 Organization

This aspect deals with the manner in which the information within the system needs to be organized. A good organization allows for very quick identification of where it is on the site and its retrieval, almost intuitive for the end users.

How will it be arranged?

Will the information be arranged alphabetically, spatially, by time or topic? Whatever the answer, the nature of the content itself and how people will try to locate it need to be understood.

So, for example, a site that has a high turnover of news-related items may wish to show the top 10 news items on the home page, with the remaining news items accessible in a separate news section. Since the news section would have a large number of items, it may be worthwhile categorizing along the lines of interest to the business (financial, R&D, competitors, etc.) to make them easier to locate. Finally, if we imagine the news archive with almost thousands of items, that could be arranged in date order.

Is the content structured or unstructured?

News items are fairly well-structured pieces of content; they have a headline, maybe a sub-headline, opening paragraph and content area, author, release dates, and maybe expiration dates. We can begin to see how each of these elements can be used to construct a page. On the home page we could show our 10 news item headlines and opening paragraphs and then allow users to click through to get to the remaining article. The experience is quite predictable and can be easily put together by site developers.

Unstructured content presents a different challenge. How do we even begin to identify content that may be of value to users from a completely unstructured source that changes day to day? There is, of course, the manual way, where someone scours endless pages of content for any relevant information and then brings it into a structured page. Fortunately, there are tools available on the market that are very sophisticated and will parse all manner of content to retrieve the relevant information.

Currency

One issue with site credibility is the currency of the content. It is important to have all content owned so there are clear lines of responsibility for its maintenance. Also, expiration dates that force turnover of content help ensure that content is updated.

Security

Probably one of the greatest concerns with Web sites is that of security. Just what are the access requirements and who should be able to look at what? It's clear that there can be no effective definition and enforcement of security without a clearly stated security policy as a benchmark for system developers, content providers, and maintainers.

8.5.2.3 Navigation

Next you need to address the question of how users find what they are looking for. The experience of navigation should allow users to find their way to the content without, at any point, feeling lost in the maze of pages. Clear guides on each page showing where they are and where they have come from and a consistent navigation scheme allows for a better experience. Again, there are some pointers to keep in mind when designing for navigation.

Search or navigate?

One of the issues with navigation devices is that on very large or complex sites, navigation systems aren't the best way of finding information. In this case the use of a search engine would be more effective. However, search engines may return a varying number of responses. In a list of 100 responses, where is the item of interest? There are very advanced search engines that employ very complex algorithms in order to return more meaningful responses; however, the problem still remains the same. The answer in most cases lies with the use of both systems. Navigation should allow users to easily locate a middle-to-high percentage of content and search engines should search not only the whole site but targeted portions of the site.

What roles do users play?

When designing the navigation path, it may be of value to keep in mind the roles that users play. Are they authors, approvers, or consumers? What type of consumers are they? What information are they most interested in? What other information would they find of use? If we take the example of a stock share Web site, users are obviously buyers and sellers of stock. They will have a clear interest in locating the latest stock quotes as quickly possible and then making the transaction. They would be very interested in information that would help them make the decision to buy or sell, such as stock reports, market watch reports, financial statements, latest breaking news, etc.

What are the channels of access?

Depending on the type of device accessing the information the navigation experience will be different. Once again an obvious comparison is that of the PC-based browser and WAP phone.

8.5.2.4 Adaptability

One of the great lessons learned in the relatively short history of the Internet is that sites need to adapt if they are to survive. User expectations of what technology can deliver and what services should be provided are constantly being raised. This raises several concerns.

What are the measures for effectiveness?

Is there a clear understanding of what determines if your site is effective and how to measure the success? Are these measurements the right ones and have they been tested? How will the metrics be captured?

Site evolution

How will your Web site cope with changes in user demand? Let us say that all of a sudden there is a surge in interest in the site. Will it be able to cope with a higher number of concurrent users? How will it cope with a change in user behavior? What are the inhibitors to a fast effective change? This could be organizational as well as being technology based.

8.5.3 Technology choices

The design pattern for the application will be based on the Model-View-Controller (MVC) pattern. In the MVC pattern, the “model” represents the business logic, while the “view” represents the page displays. The “controller” is charged with the responsibility of channeling requests to the appropriate business component (model) and calling the appropriate page construction component (view) based on the output of the business logic commands. Technology choices that will best accomplish each of these

pieces of the application will need to be made. The options we have considered are covered in Chapter 5, “Technology options” on page 35.

The client environment is Web browser-based. The supporting technologies for the view have been identified as HTML, JavaServer Pages, JavaScript and JavaBeans.

The controller represents server-side code directing requests and returning responses. Java servlets are platform and protocol independent, implementing a simple request-and-response framework for communications. This makes them the ideal choice for the controller function.

When considering the technology choices for the model, we have a more interesting challenge. The key prerequisites for the model is that it needs to be built using a server-side component architecture that is distributed, scalable, secure and reliable. It should allow the portability and re-use of business logic. In the bank’s case, they are also considering the fact that their core databases vary in structure and reside on different operating systems. In addition, they believe there may be a merger with another bank in the near future, which will introduce even more complexity into the picture.

Although almost anything can be done with Java programming, there are technologies and products available that will do some tasks much more efficiently than a Java program, and alleviate the need to program these complicated tasks.

For WebBank, the decision was made that combining two sets of customers and sending their requests to the correct back-end systems can best be handled by using some type of messaging technology as the routing mechanism. After evaluating the requirements for the model portion of the application, they chose to implement messaging technology using IBM MQSeries. They used Java commands to put messages on a queue and to receive messages from a queue. MQSeries Integrator will be used as a messaging broker that will route the messages and, if necessary, transform them to the appropriate format and structure.

8.5.4 Deployment model

Another task is to begin planning the deployment model. This will include:

- Finalizing the operational model. The model chosen will depend primarily on the technology options chosen and security needs. The runtime models we chose to consider as final solutions are discussed in Chapter 3, “Choosing the runtime topology” on page 17.

- Determining the product mapping from the logical architecture. There will most likely be several choices of products available for implementing the chosen runtime topology. While considering the choices you should consider possible future application expansion. Choose products that offer enough flexibility and function to accommodate future business growth. Our decision for these options is covered in Chapter 4, “Product mapping” on page 25.
- Selecting system management methods and tools. Planning for system management begins here. It is a crucial piece of the puzzle and must be considered early in the planning cycle. This includes not only how applications and servers are controlled, but security issues as well. These options are covered in Chapter 10, “System management guidelines” on page 239.

Planning for development phases and testing should also begin in this phase.

8.6 Micro design

In the micro design phase we need to expand the business process model by refining these into actual work products. These work products will define the interactions necessary between the external users (actors) and the proposed system needed to accomplish the business goals. We have chosen to use Rational Rose to produce the work products.

8.6.1 Use cases

The first step in the macro design phase is to capture the functional requirements of the system, that is, what the system should provide in terms of services to its users, in the form of use cases. The main purpose of a use case model is as a communications tool. It is utilized by end users, developers and domain experts to establish the boundary of the proposed system and to fully state the behavior and functional capabilities to be delivered to end users. It is also the primary basis for defining the user interface requirements.

The principal use cases should typically represent a piece of functionality that is complete from beginning to end and delivers value to the user (actor). Unfortunately there is no formula for identifying good use cases. This comes from the experience of building systems that employ modelling rigor and from a good understanding of the problem domain.

The use cases identified in the macro design have been captured in the Rational Rose use case diagram shown in Figure 64.

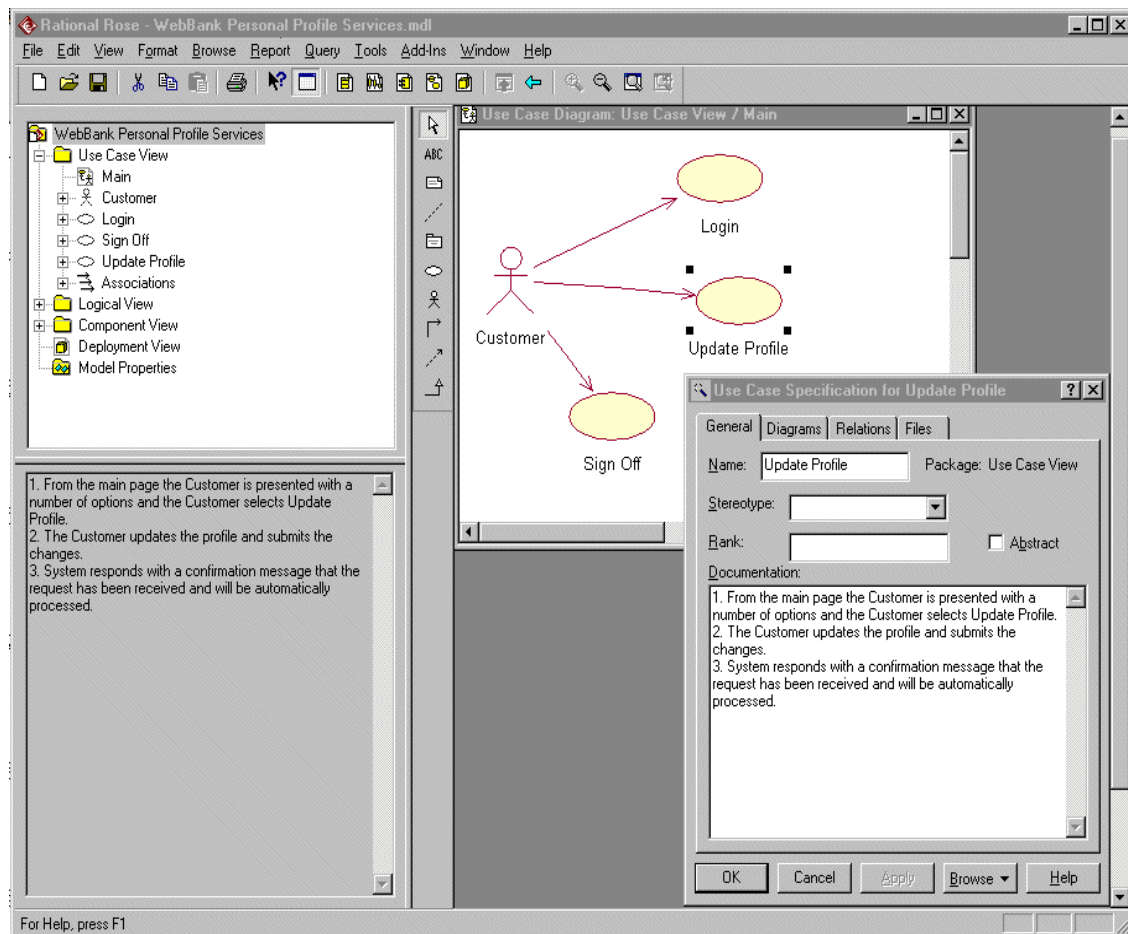


Figure 64. Rational Rose use case model

The stickman figure in the use case diagram is used to define the actor, called Customer in our example. The actor is not a part of the system as such, but interacts with the system to either submit and/or receive information to/from the system.

Customer, as an actor, is quite easy to identify since it is a “person” role that most people can empathize with. However, actors may not be, and in many cases are not, “person” roles, but rather another system that is outside the boundary of consideration that maintains the interaction.

Typically, the initial list of actors identified will be subject to change as their interaction with the system is understood. For example, in the case of

Customer, is a new Customer equivalent to an existing Customer? If the interaction with the system is different, then a new actor will need to be defined. If the interaction is the same then the same actor will suffice.

There are three use cases depicted in Figure 64: Login, Update Profile and Sign Off. To illustrate our development process, the remainder of this chapter will focus on the Update Profile use case. It will be assumed that the Login and Sign Off use cases are already implemented.

For each case, there will be a description, or specification, that defines each aspect of the use case. Table 7 shows the layout of a typical use case. The title identifies the use case with a name and number that will be referred to throughout the development cycle. The specification includes a description of the functionality to be delivered, any relevant assumptions made, definition of actors, and any necessary preconditions that need to be satisfied before the use case can begin.

Table 7. Update Profile use case

UC06 - Update Profile	
Description	This facility will allow Customer to remotely update personal profile information for all accounts.
Assumptions	All accounts have a personal profile. All profiles held for Customer are the same.
Primary Actor	Customer
Preconditions	Customer has completed the Login use case and is trusted by the system. A Customer Accounts entry exists.
Main Scenario	1. From the main page, Customer is presented with a number of options and selects the Profile Update. 2. Customer updates the profile and submits the changes. 3. The system responds with a confirmation message that the request has been received and will be automatically processed.

UC06 - Update Profile	
Extensions	<p>1a. System is unable to retrieve details. A message is displayed to that effect with an option to return to the main menu.</p> <p>1b. The system returns Customer's profile details. The system displays the Customer's title, name, address and telephone number. For each address line entry there is a corresponding input field for updates and an Update button to submit the changes.</p> <p>2. The page displays a link back to the main menu page.</p>

These specifications can be defined in Rational Rose (see the dialog box called "Use Case Specification for Update Profile" in Figure 64) or in some external way, as in our table. Where external methods are used to capture the information, there is the ability to attach them to Rose use cases as external files.

The main scenario describes the principal flow of events for the completion of the use case and the extensions capture alternative paths in the flow.

The second precondition defines a Customer Accounts entry. A single customer may hold several accounts, each with its own account number. To simplify things for the customer and to head off any problems in the future when the two banks merge, each customer is assigned one account number to keep up with. The Customer Accounts entry is used by the application to take that number and determine the accounts held by the user and the real account codes that go with them. The customer profile will need to be updated at each back-end system where he holds an account.

The Customer Accounts entry is kept in a staged database, which is synchronized with legacy systems. This allows the application to determine whether a customer account exists without interrogating each and every back-end system.

The customer profile is also held here and synchronized with the back-end systems, although the format of this data may differ in each system. For instance, it is possible that a customer could hold both a checking and savings account. However, these two accounts represent different subsystems that were developed by different teams, over different time periods. The information held by each type of account is similar but in a different format. For this reason, provisions have to be made to allow the synchronization of the staged data with back-end data of various formats. The

customer data format for the two back-end systems in the WebBank example is illustrated in Table 8.

Table 8. Checking Account table and Savings Account table

Checking Table	Savings Table
NUMBER INTEGER	NUMBER INTEGER
	TITLE CHAR(6)
NAME CHAR(40)	SURNAME CHAR(20)
	FORENAMES CHAR(20)
ADDRESS1 CHAR(40)	HOUSENO CHAR(6)
ADDRESS2 CHAR(40)	HOUSENAME CHAR(40)
ADDRESS3 CHAR(40)	STREET CHAR(40)
ADDRESS4 CHAR(40)	DISTRICT CHAR(40)
ADDRESS5 CHAR(40)	CITY CHAR(40)
	STATE CHAR(40)
	COUNTRY CHAR(40)
	ZIP CHAR(8)
TELEPHONE CHAR(20)	PHONENUM CHAR(20)
BALANCE DECIMAL(12,2)	BALANCE DECIMAL(12,2)

The distinction represents the typical problems faced in enterprise environments in keeping data sets of different formats synchronized.

8.6.2 Storyboard

The storyboard is a visual prototype tool that is used as an aid to validate the requirements. It can be created using a graphics package or an HTML wizard. No great length of time should be spent on getting the “look” right. The functionality is the key point. Storyboards are excellent communication tools and should be used to play back the requirements to the end users and help validate use cases.

In our example, from the use case, we can derive five windows that hold functionality or show pertinent information. Figure 65 shows the storyboard, which is represented as a flow from window to window cascading down.



Figure 65. Storyboard

8.6.2.1 Page 1

The use case starts with the assumption that the user has already authenticated. This is reflected in Figure 66, which shows the welcome message and the user's name displayed in the top right-hand corner. The first page shows the option under consideration, in our case, the ability for the user to view and update his personal profile.

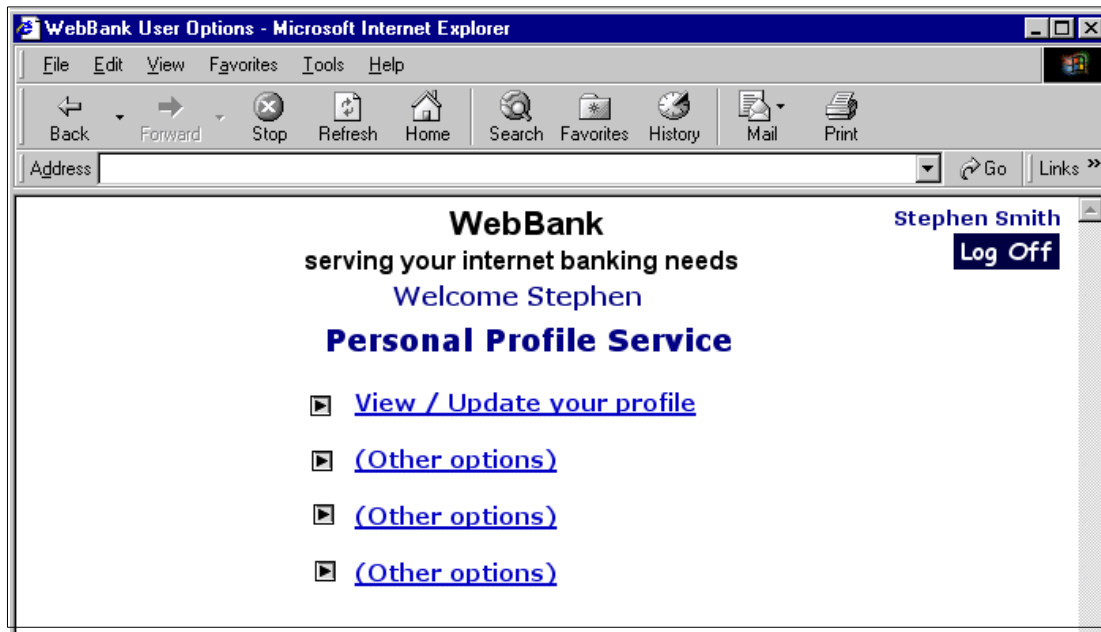


Figure 66. Page 1 - main menu

8.6.2.2 Page 2a

Figure 67 shows the next page, Page 2a, where the user profile has been returned successfully after the selection was made on Page 1. From this page the user may choose to change his profile. To do this, he updates the fields and clicks the **Update** button to submit the changes.

WebBank
serving your internet banking needs
Personal Profile Services

Stephen Smith [Log Off](#)

We currently hold these details for you.
Are they Correct?

Title	Mr
First name	Stephen
Middle name	J
Surname	Smith
House no.	199
House name	The Vale
Street	Acacia Avenue
District	
City	New York
State	NY
Country	USA
Zip	NY 0012
Tel	917 111 2222

[Update](#)

Figure 67. Page 2a - personal profile

8.6.2.3 Page 2b

An alternate outcome of the selection from Page 1 is that the system was unable to retrieve any details, in which case Figure 68 shows the specific error returned.

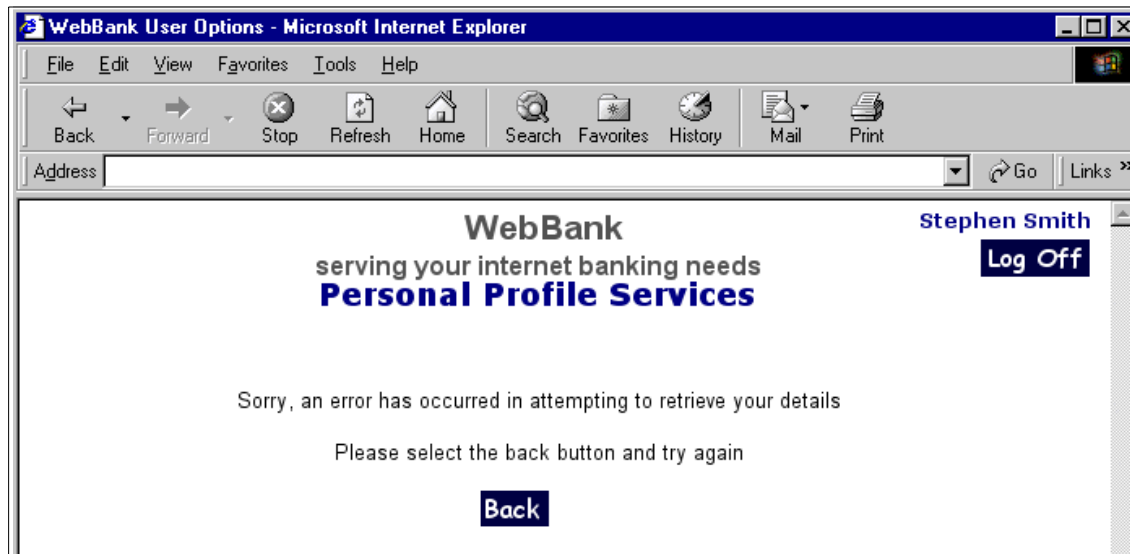


Figure 68. Page 2a - error returning profile information

8.6.2.4 Page 3a

Assuming that the user submitted changes to the profile and all has gone well, an acknowledgement in the form of Figure 69 is displayed to the user, with the options to log off or go back to the main menu.

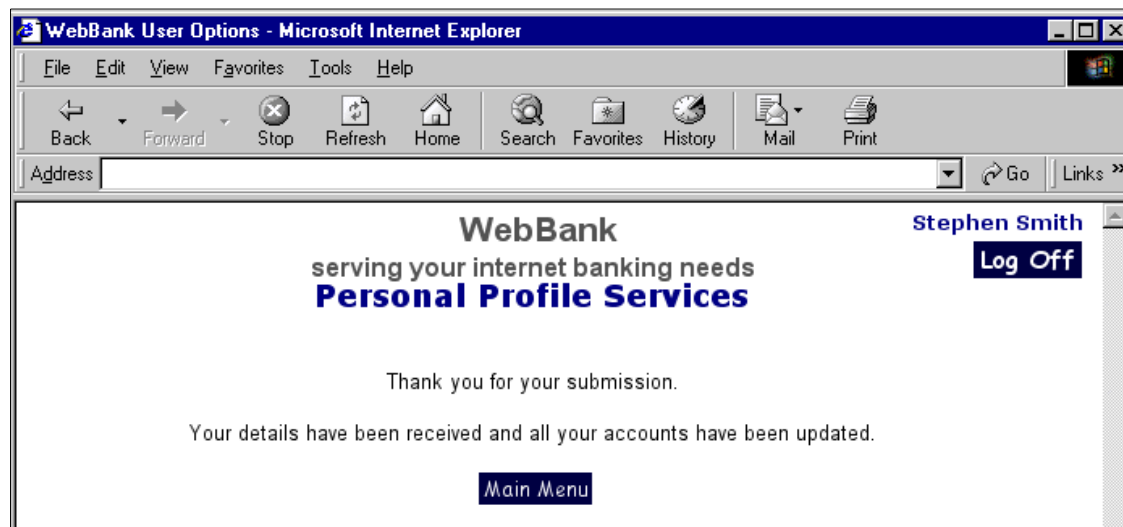


Figure 69. Page 3a - update acknowledgment

8.6.2.5 Page 3b

If the update from Page 2a is unable to take place then the appropriate error message is displayed to the user, as in Figure 70.

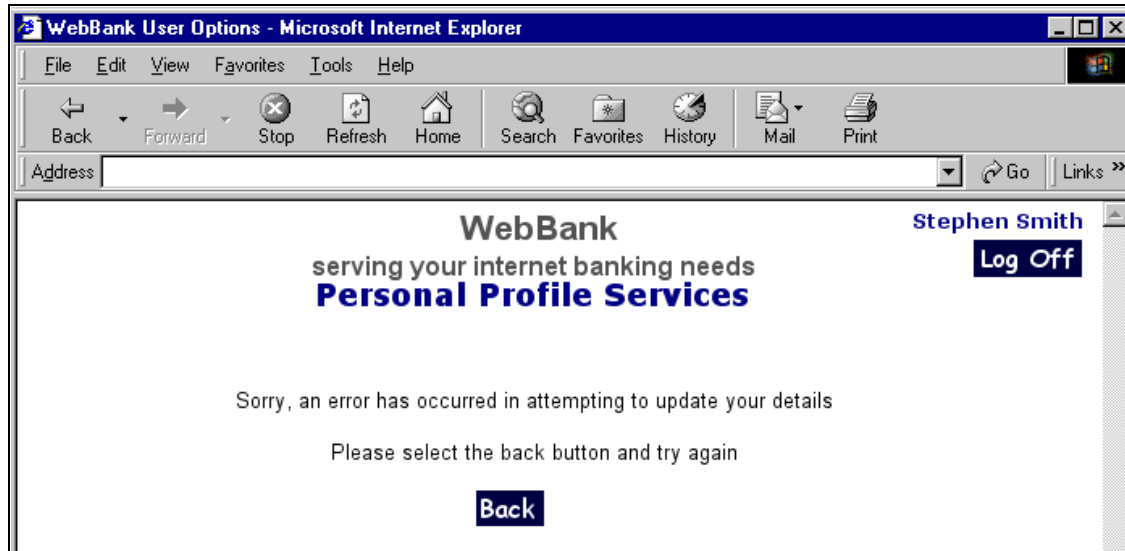


Figure 70. Page 3b error updating the profile

8.6.3 Activity diagrams

The next set of views in Rose are grouped into a category called Logical View. In this category, we logically construct the various elements required to deliver the functionality described in the use cases. This means looking at the process involved and decomposing them to their various classes and interactions.

The first diagram we will discuss is the activity diagram, which is used to further elaborate on the use cases. More specifically it shows a flow of control from activity to activity, where an activity can be seen to be some behavior within the workflow. This can be seen as the first steps in documenting the dynamics of the system.

Activities undergo transitions to other activities; they may encounter decision points, showing branching of a flow. There may be activities that run in parallel. The diagram may also show people, organizations or components responsible for the execution of those activities.

We have used Rational Rose to produce the activity diagram in Figure 71.

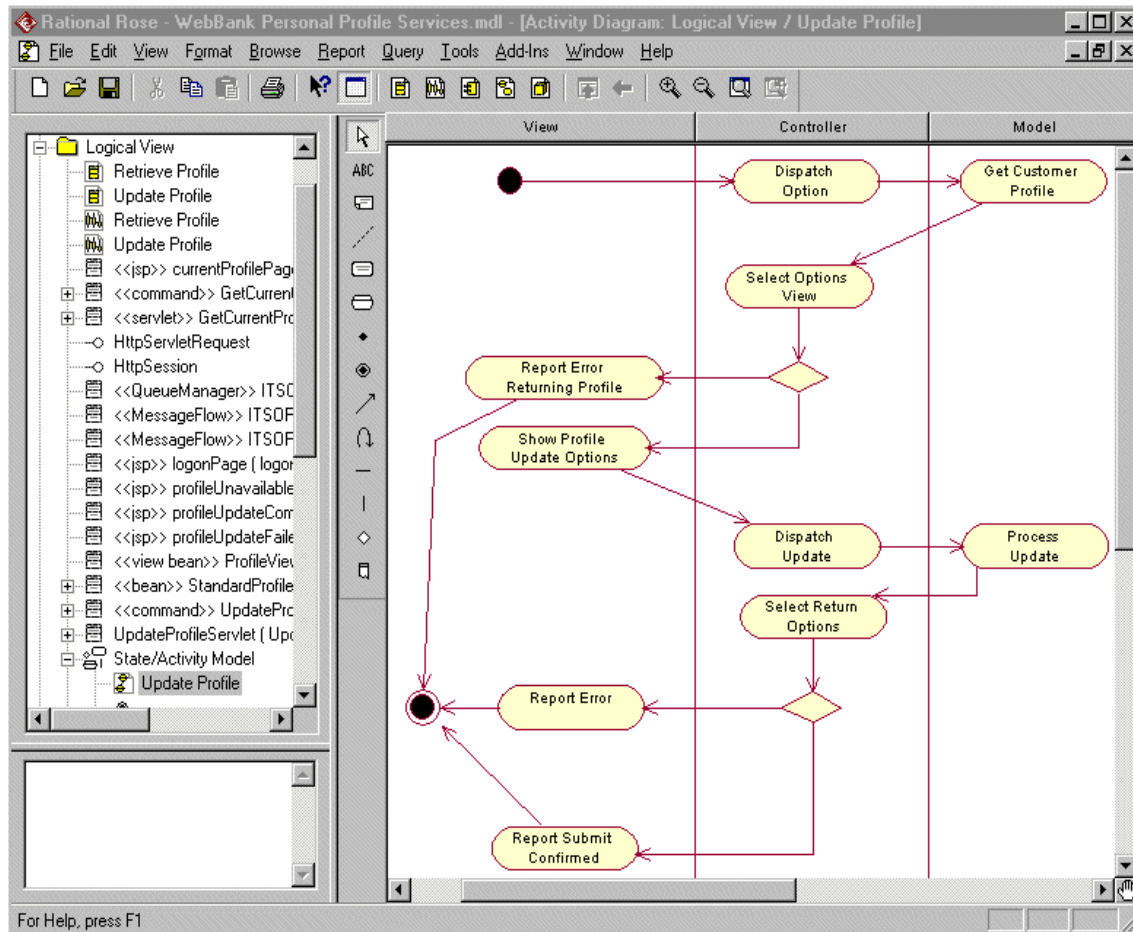


Figure 71. Update Profile activity diagram

The view is the point at which the activity begins and terminates. "Dispatch Option" forwards the request to update the profile to the model. "Get Customer Profile" attempts to retrieve the profile and returns the result back to the controller. This operation may have failed to complete, in which case there is a decision point reached in the controller to either display the "Report Error Returning Profile" view or the "Show Profile Update Options" view. If the former is displayed then the flow terminates. The latter view shows the current profile with the ability to modify any part of it and then to submit the changes for the "Dispatch Update" activity to redirect.

Once redirected to the model, the update either occurs or there is an error. "Process Update" returns the response to the "Select Return Options"

controller, which decides to display the appropriate error or results view. The key thing to note about the positive result is that no data is returned, only an indication that the submission request was successfully received by the model.

The reason for this is that the model may be responsible for updating any number of back-end systems and to do so immediately may leave the customer waiting for a lengthy period of time until the request completed. Or, if one assumes that the checking account system is not available for the update when the request is initiated, then what would be the fate of the request? If an immediate response is required then the system has no choice but to respond with a failure and the customer will need to try again at some point in the future, and even then without the assurance that the request would be processed on the subsequent attempts. This is obviously unacceptable for a service aimed at providing value and high customer satisfaction. Ideally, the customer would want to submit the request once and just forget about it. This needs to be done with the confidence that the processes initiated will assure that the update occurs whether or not a particular subsystem was available at the time the request was initiated.

It may be possible, and in some cases recommended, to put a tracking mechanism in place so that there is clear visibility to the customer of where the request is in the process, or even a confirmation sent by e-mail or on a status page to show that the request completed. There are any number of options available, but for the purposes of this example, we will check the status by re-initiating this use case to display the updated profile.

8.6.4 Class models and class diagrams

The next stage of the process is to start defining classes that will represent the functionality of the system. This is seldom an easy process. Our example business model is quite straightforward and is represented by a customer and a few account objects. In the real world, a great deal of time would be spent in defining the objects and the relationships between them. This process is generally performed without consideration for specific technology implementation classes, making the business logic clear to see and technology independent.

The identification of candidate classes will be made easier if the MVC pattern has been used and there are clear lines of responsibility drawn between the model, the view and the controller. In most cases the candidate classes are a starting point from which further iterations of identification and verification of class definitions occurs until a suitable class model is defined.

The business model implemented here is quite straight forward and can be represented by a profile class and a customer class. There are two tasks performed by the model, retrieve profile and update profile. The point of interest here is the technology used to support the model. From the choices we have focused on so far in this book, it is apparent that this part of the class model covers two implementation technologies: Java classes and message flows implemented by MQSI.

We have determined that our application will use servlets in conjunction with a command pattern implemented using the command package, available in WebSphere Application Server, Advanced Edition 3.5 and VisualAge for Java 3.5.

In order to simplify this discussion, the class diagrams discussed in this section will not show the command interfaces. Details and coding examples of the command package are described in 6.1, “Command framework” on page 55. For information about the MQSI application design please refer to Chapter 7, “MQSI application design guidelines” on page 103.

8.6.4.1 Analysis diagram

Before diving into the class diagram, it may be a good idea to do an analysis of the core classes that will make up the function of the application. The analysis diagram helps us identify the core of the class diagram by hiding much of the complexity of the implementation. With Rational Rose, we will produce this analysis diagram as a class diagram, but will only model the heart of the application program, showing the major classes involved in the interaction.

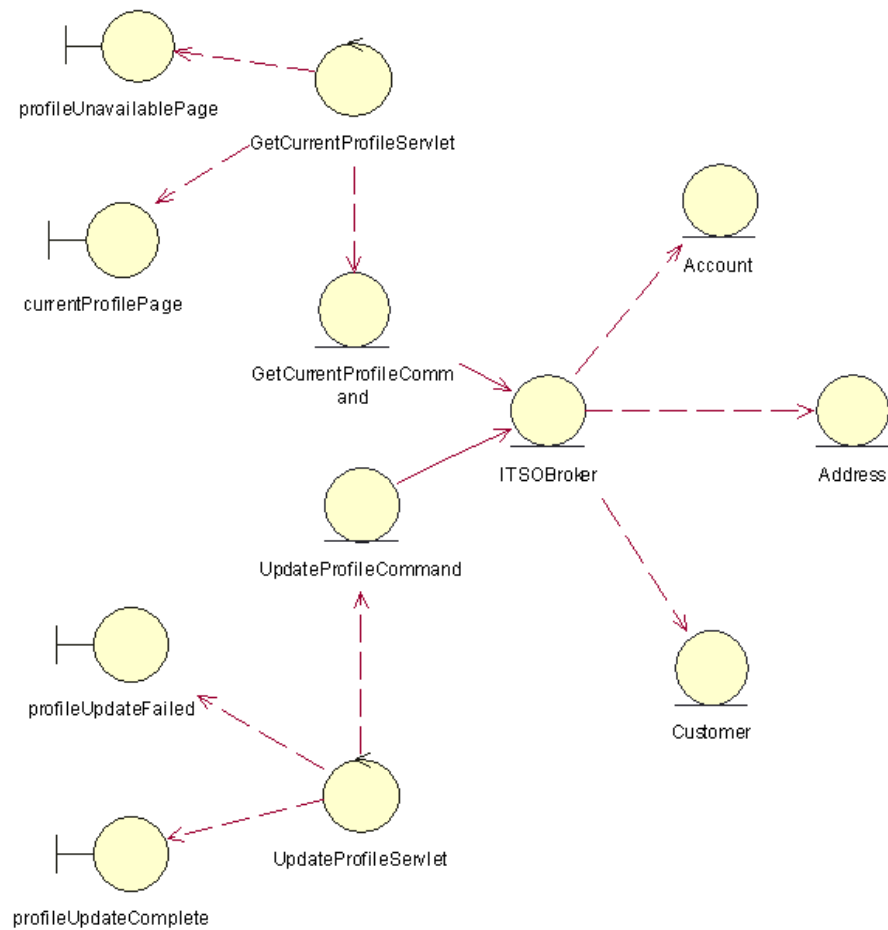


Figure 72. Update Profile analysis diagram

In this diagram, we show that there are two stages to updating the profile. First the profile is retrieved and displayed. In this stage, the `GetCurrentProfileServlet` is the controller and invokes `GetCurrentProfileComm`, which in turn retrieves the current profile information. The second stage is the update. The `UpdateProfileServlet` is the controller and invokes `UpdateProfileCommand`, which causes the messaging broker to update the profile on the staging database and the back-end databases.

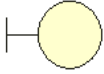


For analysis purposes we can view the MQSI broker role as an object that encapsulates a number of business methods. The broker is actually a named

object with a well-defined interface. In Figure 72, the broker object is named "ITSOBroker". This broker is on the receiving end of the execute() methods of the two commands. Each command elicits a specific response by calling a particular method within the broker (in fact, the getProfile() and updateProfile() methods, respectively).

You may notice it looks different from the more detailed class diagram shown in Figure 73 on page 169. Rational Rose allows you to do class diagrams using different notation. In Figure 72, we chose to use the icon notation option. This is an option available only at the file level, so we made a "supplemental" file to produce these diagrams.

The icons assigned to each node represent the roles each will play in the application. The direction of the arrows between icons also have significance.

Table 9. Class stereotype icons

	<p>Icon: A boundary class icon, representing a point where data or control either enters or exits the application. In our application these will be implemented with JSP files.</p> <p>Arrows: The object at the tail of the arrow will send a message (or pass a parameter list) to the object at the head.</p>
	<p>Icon: A control class icon, representing a point at which a computation or other significant state change is generated. It models use-case specific behavior.</p> <p>Arrows: The object at the tail of the arrow will create the object at the head.</p>
	<p>Icon: An entity class icon, used to represent a data object (usually persistent) and the code that encapsulates it.</p> <p>Arrows: The object at the tail of the arrow is a specialization of the object at the head.</p>

8.6.4.2 Class diagram

When building the class diagram, any components responsible for the MVC view functions of the program, such as windows that gather user input or show the results, will be written using HTML and JSPs. These components will construct HttpServletRequestes and display data encapsulated in a Java bean (a result bean).

The MVC controller component maps user input to tasks in the model and manages the generation of result presentations. In the activity diagram in Figure 71 on page 163, this is shown as two pairs of "Dispatch..." and

"Select..." activities. This dispatch/select (send/receive) pairing suggests that the activity be written as a cohesive unit. The question is whether or not both pairs should be implemented as a single code object. This will depend on the application and what seems to be more logical. The choice made here is to keep the two separate. This is because the function to retrieve a profile is logically independent from updating a profile. Each matched pair will be implemented as its own servlet.

There is value in showing the role of the broker in such a diagram for architects and people interested in the high-level design, as the detailed interactions and complexities are hidden. However, for this very reason, a diagram such as this is of little practical use for system implementors and developers, for whom the complexity of interactions needs to be understood for the system to be built. So, the broker needs to be decomposed into its constituent parts and the interfaces and relationships understood for the interactions to be defined.

In reality, the interface to the broker is actually any number of message queues managed by an MQSeries queue manager. The message queues are where applications "put" message requests and "get" message responses. An application will send a message to a queue manager naming the inbound queue. The queue manager reads the message destination, determines where the inbound queue is, and puts the message in the queue if it is local to itself or hands it off to the queue manager responsible for that queue.

The messages are then involved in a flow within the broker and messaging network that may trigger decisions points, further lookups of data to augment the message, and some transformation to the structure and finally result in a response message in an outbound message queue.

The message flow can be seen as a class of its own that calls a method of the broker object. In this particular instance, the two ITSOBroker methods `getProfile()` and `updateProfile()` are called by the "ITSOProfileLookup" and "ITSOProfileUpdate" message flows respectively. The message flows are defined in Chapter 9, "Developing the MQSI application" on page 183. At this point we are not interested in capturing how message queues, queue managers and brokers relate to one another. However, we are interested in the interface into the messaging infrastructure, that is, the queue manager, the message flows and how our applications submit and receive information from them. To take this part of the model to the next level of detail we can substitute ITSOBroker with a named queue manager, `ITSO.QM.BR`, and add the two message flows.

Retrieve the customer profile class diagram

The class diagram in Figure 73 shows the classes required to retrieve the customer profile. The class types, shown between << >>, are referred to as stereotypes in Rose notation. Typically, all methods and properties would be shown in this diagram. To simplify our discussion, only those we are concerned with will be shown here.

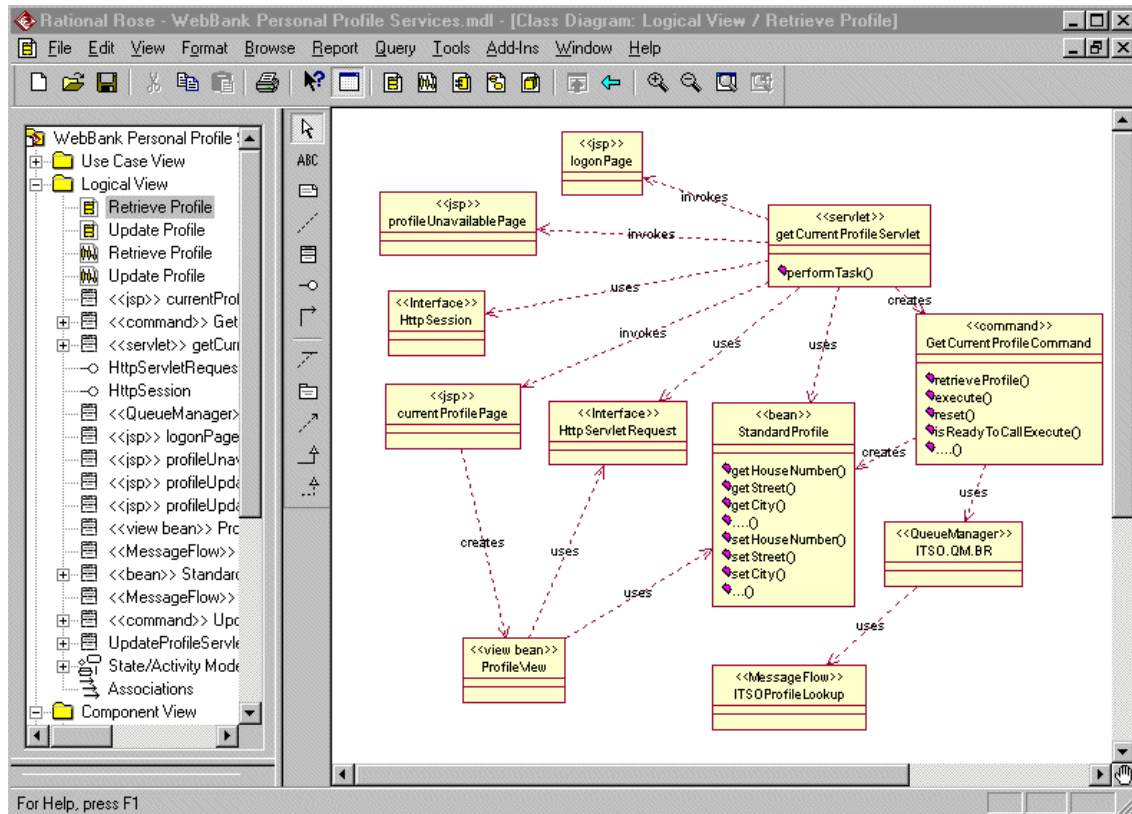


Figure 73. Retrieve Profile class diagram

getCurrentProfileServlet - This servlet fulfills the controller function. Its `performTask()` method checks to see if the session object exists by calling `getSession(false)` which returns null if a session doesn't exist. This means that the user has not been identified, in which case it redirects to the `loginPage` JSP. If the session exists the servlet retrieves the user's identification from the session object using the `getValue()` method. It then creates an instance of the `GetCurrentProfileCommand` and executes it. When the command completes, control is handed back to the servlet which either shows the success view or error view, depending on the command outcome.

GetCurrentProfileCommand - This command is responsible for initiating the request to the broker and holding the result. The request is actually a message created by the command and "put" on an MQSeries message queue. The command then waits for a message response and "gets" it from the MQSeries output queue. Once it has the message, the command instantiates a *StandardProfile* object and populates it with the profile details in the message using a constructor method in the *StandardProfile* class. It then returns control to the servlet.

ITSO.QM.BR - The queue manager that owns all the message flows in *ITSOBroker*. Its primary purpose is to locate the target message queues for the messages.

ITSOProfileLookup - This message flow equates to the *getProfile()* method of the *ITSOBroker* and is owned by the queue manager. The message flow starts by getting the message from the input queue (placed there by *GetCurrentProfileCommand*). It parses the message, doing the necessary database lookups and transformations to produce a result. The result is then encapsulated in another message and "put" into the output queue for the *GetCurrentProfileCommand* to "get". The implementation details for the message flow is described in Chapter 9, "Developing the MQSI application" on page 183.

StandardProfile - This is the result object created by the *GetCurrentProfileCommand* and used by the *ProfileView* bean.

ProfileView - This is a View bean used by the *currentProfilePage* JSP. A View bean is responsible for combining the result data and the display-specific attributes. View beans are described in *Patterns for e-business:*

User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition, SG24-5864.

currentProfilePage - This JSP is presented to the user when there is a successful retrieval of the profile information.

profileUnavailablePage - This page is presented to the user when there is a problem with retrieving the profile information.

loginPage - This is a JSP responsible for gathering the user's login information. It passes it to the *loginServlet*, which does the necessary authentication checks and, if successful, puts the information into the *HttpSession* object.

HttpSession and *HttpServletRequest* - These are the standard objects used by servlets to hold session and request information.

Update the customer profile class model

Figure 74 depicts the key classes required to implement the design for the Profile Update function.

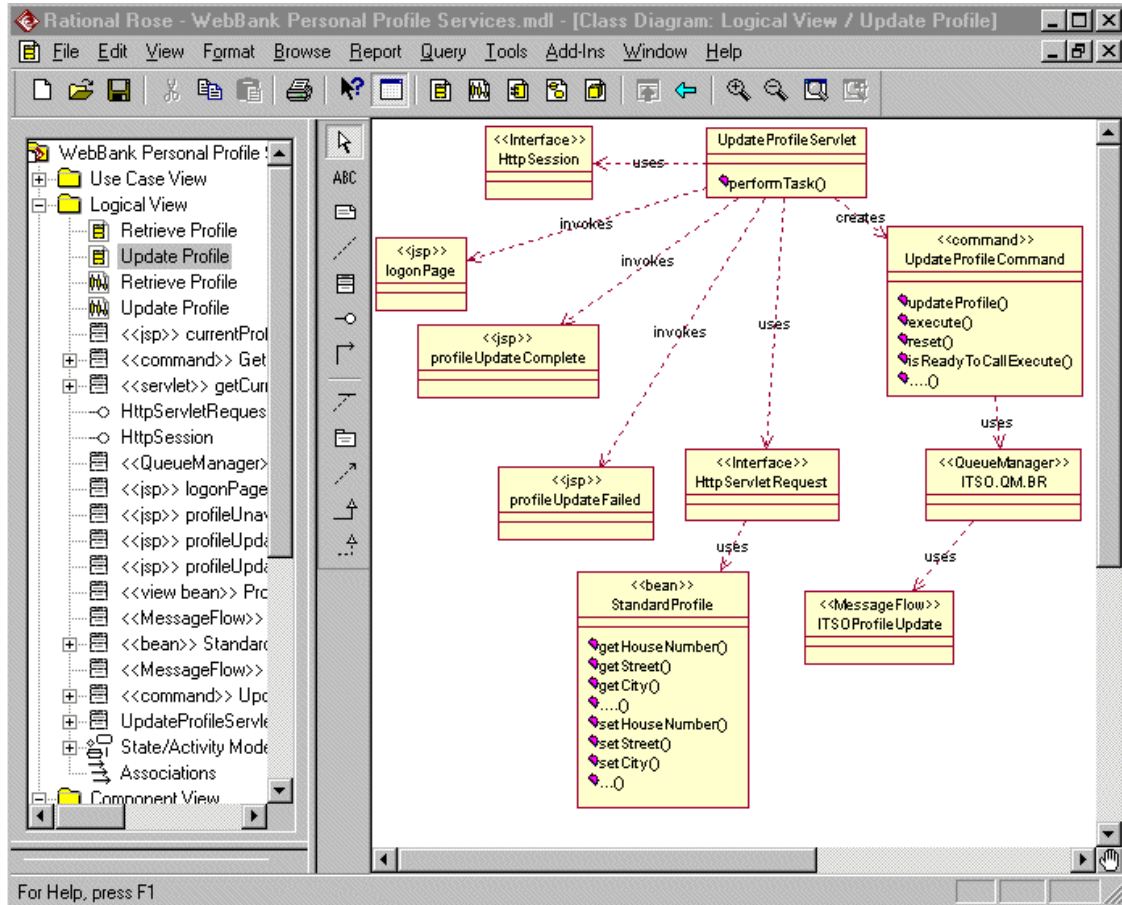


Figure 74. Update Profile class diagram

UpdateProfileServlet - This represents the controller function. It does the session object check, retrieves the updated profile information from the form, and reads it into the standard profile. It then instantiates the **UpdateProfileCommand** and populates it with the user string from the session and the **StandardProfile** object and calls its `execute()` method. It finally checks the command's message string using the `getMessage()` call to verify that the command completed successfully. The servlet invokes the **profileUpdateComplete** JSP, unless there is an error, in which case it invokes the **profileUpdateFailed** JSP.

UpdateProfileCommand - This command is similar to the *GetCurrentProfileCommand* described earlier in that it puts a message into a queue (the new profile). However, there is no response message to “get”. The command checks that the message is sent without error and writes success or failure to the message string.

ITSO.QM.BR - The queue manager that owns all the message flows in the *ITSOBroker*. Its primary purpose is to locate the target message queues for the messages.

ITSOProfileUpdate - The message flow in this instance represents the *updateProfile()* method for *ITSOBroker*. It takes the inbound message and updates the various back-end applications as well as updating the cache. There is no outbound message.

profileUpdateComplete - This JSP is presented to the user on success.

profileUpdateFailed - This JSP is presented to the user if there is an error in the update.

logonPage - This JSP is presented to the user if they have not been authenticated.

8.6.5 Interaction diagrams

A more detailed analysis of the relationships between the elements identified in the class diagrams can be shown by the interaction diagram. There should be at least one interaction diagram for each use case. A use case, as we have described, represents a number of scenarios. The interaction diagram takes the main scenario and captures all the important dynamic issues relevant to implement that functionality.

Figure 75 shows an interaction diagram in Rose.

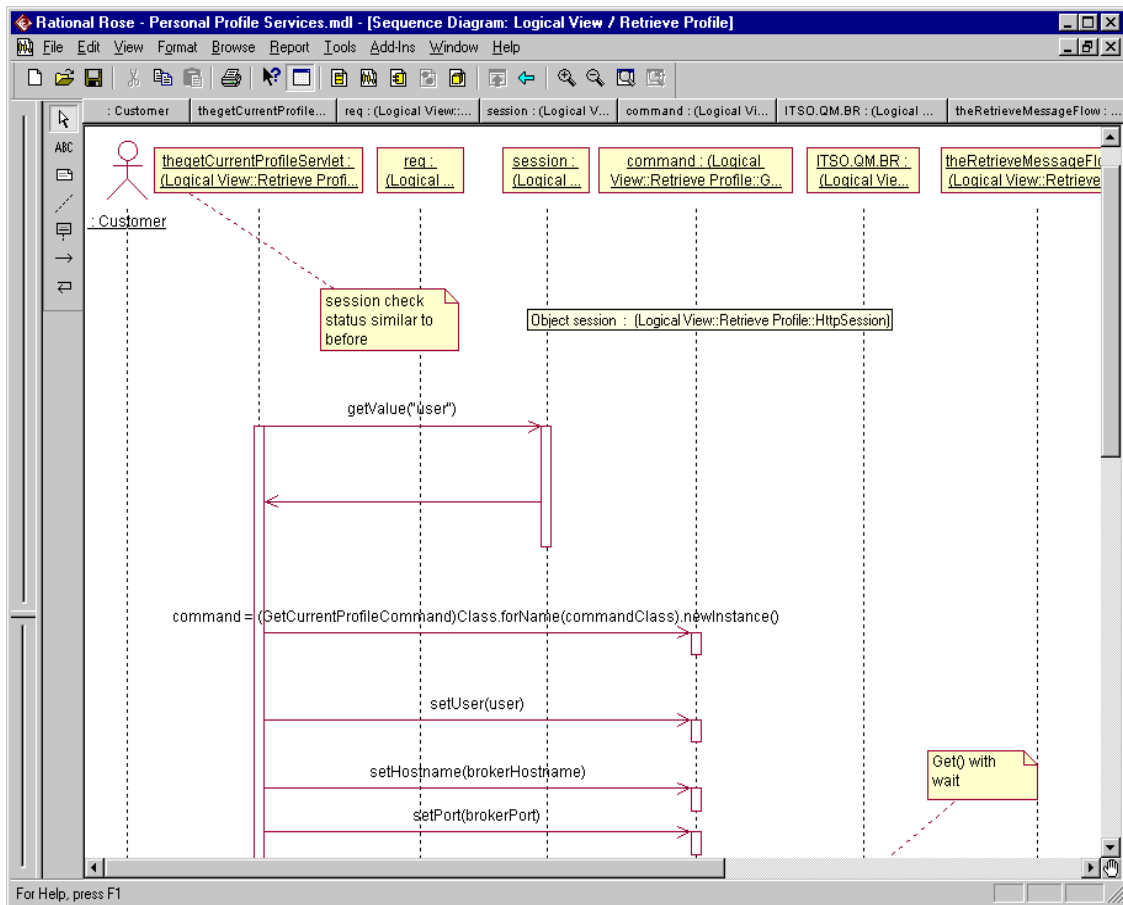


Figure 75. Retrieve profile interaction diagram

Unfortunately, the complete diagram is much too large to show here, but we will list the actions are illustrated:

- The performTask() method of getCurrentProfileServlet checks for a valid session via a call to getSession(false) indicating that a session reference should only be returned if one already exists. If null is returned, then no session exists and the servlet uses the RequestDispatcher to send the user the loginPage.jsp, forcing the user to log on before using this servlet.
- If a valid session object exists, the user is logged on, and processing continues.
- An attempt is made to retrieve the user's ID string using the getValue() method on the AccountDescriptor objects that were stored in the

HttpSession by the logon servlet. If the user string does not exist, a new `InvalidUserException` is thrown, and the error page is sent to the user.

- The servlet instantiates a concrete subclass of `GetCurrentProfileCommand`, based on the servlet init parameter "commandClass". Depending on which class is used, either JMS or MQ classes for Java will be used.
- Properties are set using calls to `setUser()`, `setHostname()` and `setPort()`. The latter two properties are set as servlet init parameters and are used to initialize the JMS/MQ environments.
- Call `execute()`. This results in an XML message being constructed in a predetermined format, using the user identification string. The message is appropriate to the underlying infrastructure (JMS or MQJava classes). The `brokerHostName` and `brokerPort` are used to determine where the message will be sent.
- The message is sent to the `ITSO.QM.BR` queue manager and the command issues a "get with wait" on the reply queue.
- The queue manager reads the incoming message and puts it into the `ITSOProfileLookup` inbound queue.
- `ITSOProfileLookup` runs to completion and a message is put on the outbound queue.
- If the valid reply is received in time, the XML message is used to construct a new instance of `StandardProfile` and the command writes success to the message string.
- When the command returns, the servlet checks the command's message string via a call to `getMessage()` to verify that the command executed successfully. If the string indicates an error, the request is redirected to the error page.
- Otherwise, if the command has completed successfully, the `StandardProfile` object is retrieved from the command and placed in the `HttpServletRequest`:

```
req.setAttribute("profile", command.getProfile());
```
- The servlet forwards the request to the `currentProfilePage` JSP which creates an instance of `ProfileView` bean. This bean retrieves the `StandardProfile` object from the `HttpServletRequest` and uses the data in it to construct a form to display the user's profile, including a button to submit any updates.
- The following scriptlet allows the Web developer to modify the appearance or the bean's table, without the risk of editing important form fields.

```

<%
profileView.setTD_LABEL_CELL_COLOR("GREEN");
%>

```

- The `getProfileUpdateForm()` method is called, which queries the `HttpServletRequest` to extract the `StandardProfile` object. This enables the view bean to construct an HTML table displaying the profile details, using the properties set by the Web developer to customize its appearance.
- The user views the profile information.

Figure 76 shows an interaction diagram for updating a profile.

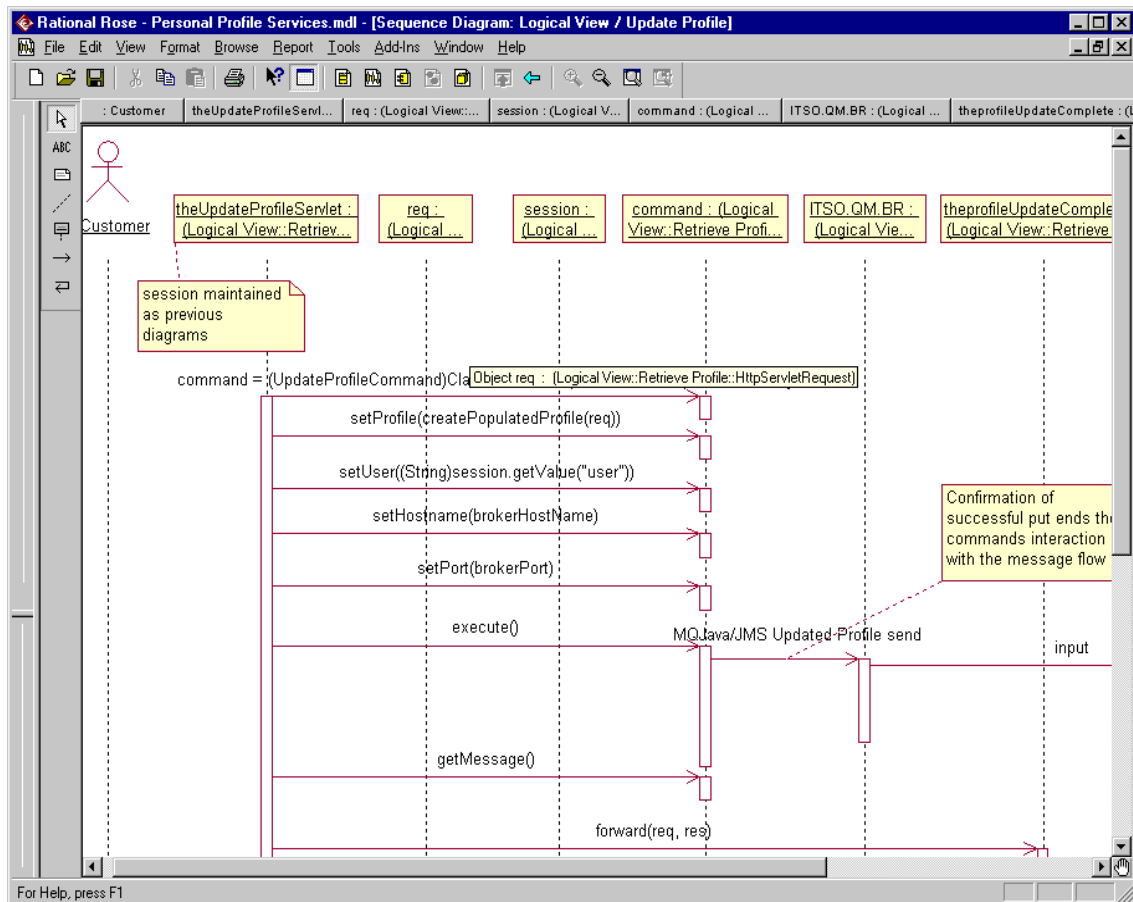


Figure 76. Update Profile interaction diagram

Once again, the complete diagram is too large to show here, so we will list the actions it illustrates:

- The user alters some or all of the profile information displayed in the `profileUpdateForm.jsp`, and clicks the Update button to submit the form.
- The `performTask` method of `UpdateProfileServlet` performs a session check (as described in the Retrieve Profile scenario).
- The servlet's `createPopulatedProfile(HttpServletRequest req)` method retrieves the user's profile information from the form and reads it into the `StandardProfile` object:

```
profile.setSurname((String) req.getParameter("SURNAME"));
```

And so on for all profile values.

- The servlet instantiates a concrete subclass of `UpdateProfileCommand`, based on the servlet init parameter `"commandClass"`. Depending on which class is specified, either JMS or MQ classes for Java will be used.
- Properties are set using calls to `setUser()`, `setProfile()` (passing the new `StandardProfile` object instance), `setHostname()` and `setPort()`. The latter two properties are set as servlet init parameters and are used to initialize the JMS/MQ environment).
- The servlet calls the command's `execute()` method.
- An XML message is constructed with the user identification string and the `StandardProfile`, to the specification and protocol required. The target queue manager of the message is determined using the `brokerHostname` and `brokerPort` and is sent.
- If the message is sent without error, then the execution has completed successfully and the command writes `"success"` to the message string.
- The `ITSO.QM.BR` queue manager determines where the `ITSOProfileUpdate`'s inbound queue is and `"puts"` the message. The flow runs to completion and updates the back-end systems. There is no outbound message.
- When the command returns, the servlet checks the command's message string via a call to `getMessage()` to verify that the command executed successfully. Depending on the string returned, the request is redirected either to the `profileUpdateComplete.jsp` or `profileUpdateFailed.jsp`.

8.6.6 Component model

The component model is concerned with actual software module organization that defines the final application. Its purpose is two-fold:

- Describe the high-level structure of the system.

This entails defining the responsibilities, relationships, and interactions of components. It specifies how existing, acquired, and developed components are related and leads to a clearer view of where they have to be placed on the operational model in order to aid the deployment effort.

- Help organize the development project.

Design and development of large complex systems are simplified as components encapsulate the complexity of individual classes or procedures. They can also act as means to allocate work that can be managed and handed out to build teams for development.

This step takes the model from a logical design, for example the class model, to a physical partitioning of the system as defined by the component model. The model maps one or more classes or even whole packages from the class model to an appropriate language-specific component.

The distinction between a package in the class model and in the component model is, again, one of logical and physical definition. In the class model a package represents a logical grouping of classes based on similar business functionality or technical grouping. It may be true that this maps directly to a package in the component model. However, this may not necessarily be the case for any number of reasons. These reasons usually come down to the constraints of the language, organization of existing models, or the organization of the units of work for the development teams.

In this specific scenario we are modeling Java and non-Java components. The Java components will be used to produce skeleton code in a VisualAge for Java project, with a certain amount of reverse engineering from the project back into the model as the build phase progresses. The non-Java components are MQSeries and MQSI-based and should hold enough information for an MQSeries network engineer and an MQSI developer to build the necessary message flows and message queues.

The components and their connections are shown on a component diagram. It is also worthwhile noting that Rose uses the component definitions as its basis for forward and reverse engineering code.

Figure 77 shows the component view of our model. This view is concerned with the physical software module organization, showing the runtime and software components created for the system. The process involves taking the classes identified in the logical view and mapping them to physical

components. The mapping between the logical classes and the components may or may not be a one-to-one mapping.

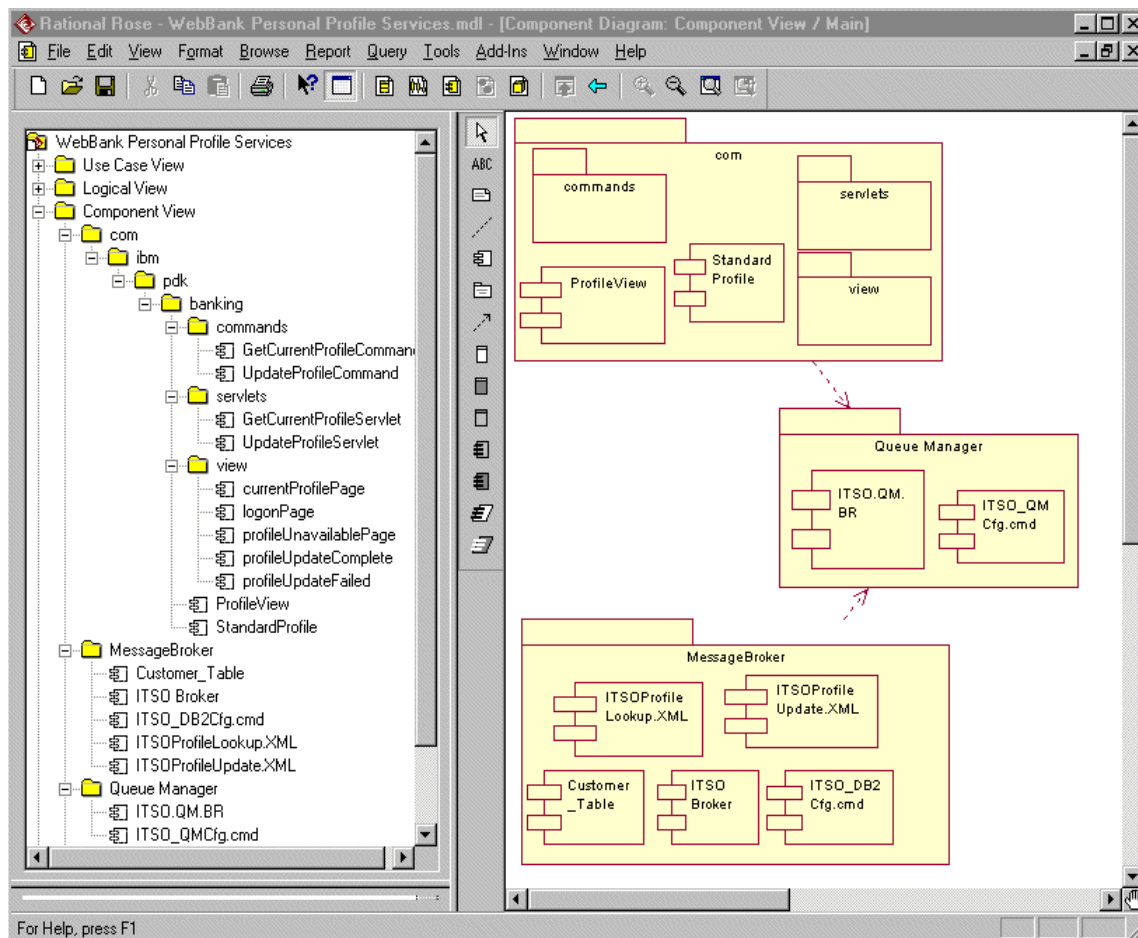


Figure 77. Component View from Rational Rose

The left pane shows the design elements in a directory structure. The directories equate to Java packages for the Java elements, which are all within `com.ibm.pdk.banking.*`. The Queue Manager and MessageBroker entries represent the MQSeries and MQSI elements.

The diagram shows a dependency between the `com.*` package and the Queue Manager; and between the MessageBroker and the Queue Manager.

Looking at the Java components we can see that they are organized by type, i.e. servlets, commands and views (JSPs). The two Java beans,

StandardProfile and ProfileView, reside within the banking package. From the names we can clearly identify a one-to-one mapping of these components with the classes defined in the class models in our earlier analysis (see Figure 73 on page 169 and Figure 74 on page 171).

The non-Java components shown are encapsulated in the MessageBroker and Queue Manager packages.

Within the MessageBroker package there are five components defined:

- ITSOProker
This is the named broker that holds all the message flows and has within itself all the MQSI nodes defined.
- ITSOProfileLookup.XML
This encapsulates the ITSOProfileLookup message flow defined in the analysis section
- ITSOProfileUpdate.XML
This encapsulates the ITSOProfileUpdate message flow defined in the analysis section
- ITSO_DB2Cfg.cmd
This component is a command file responsible for setting up the database tables and files used by the broker.
- Customer_Table
This represents the DB2 table that holds the customer profile information.

The QueueManager package contains two components:

- ITSO.QM.BR
This is the actual queue manager that delivers messages to the two message flows.
- ITSO_QMCfg.cmd
This is a command file responsible for setting up the various message queues, channels and other components required by the MQSeries network.

8.6.7 Deployment model

This phase is really concerned with the physical makeup and distribution of the system. It maps the runtime applications to their processing devices. The applications, devices and their relationships are shown on a deployment

diagram. This view takes into consideration other requirements such as system availability, reliability, scalability and performance.

The diagram illustrates nodes as the runtime processing elements with associations, indicating communication paths, as the connections between them. Software and applications are shown as text attached to a node or group of nodes.

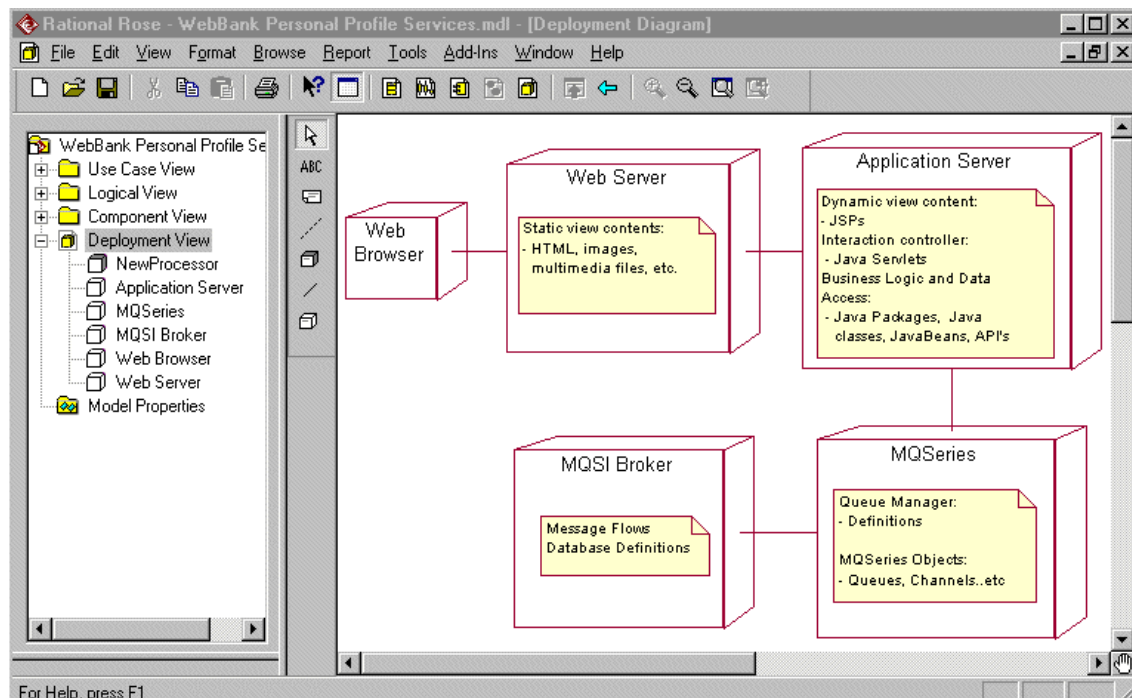


Figure 78. Deployment diagram

From the diagram, five nodes are illustrated with the components running on them.

- Web browser

This is the standard way of accessing the application, rendering HTML pages or JSPs.

- Web server

This is a standard HTTP server delivering static HTML pages and supporting media files. It also forwards all requests for dynamic content to the application server.

- Application server

The application server serves dynamic content in the form of:

- View: The page constructor implemented with JSPs
- Controller: The interaction controller as Java servlets
- Business Logic: Java beans and the command framework using JMS messaging and MQSeries classes for Java to connect to the MQSeries network.

- MQSeries

The MQSeries network is the messaging transport used between the application server and the broker. It consists primarily of queue managers and MQSeries objects such as queues and channels, that are necessary for the transport functionality.

- MQSI broker

The broker contains the message flows that hold the business logic for accessing, transforming, augmenting and updating the data in the local cache database and to the back-end systems.

It also holds the database definitions used by the broker for administration purposes.

Chapter 9. Developing the MQSI application

In our sample application, we have two use cases that require MQSI to act upon an instruction from WebSphere. These are:

- The customer profile lookup
- The customer profile update

The role of the MQSI application has been defined in Chapter 8, “Application development guidelines” on page 141. This chapter will describe the process of developing the MQSI portion of this application. It will take the design criteria produced by the modelling work and transform it into a working, deployable example MQSI application.

Please note that the following sample application scenarios have been designed to illustrate some of the techniques that may be used to take advantage of the products that play a role in this redbook. The scenarios have not been subjected to the full, rigorous design validation and testing processes that should apply in development of a solution for production use.

9.1 The contract with WebSphere

During the design of the WebBank application, the role of MQSI with regard to the WebSphere application were defined. These requirements are fairly simple.

Customer profile lookup: WebSphere will request details of a customer profile by sending a “customerrequest” XML document containing the “userid” of the customer. As a synchronous response, it requires a “profilemessage” XML document containing all details of the customer profile.

Customer profile update: WebSphere will request an update of a customer profile for all accounts held by that customer by sending a “profilemessage” XML document containing the new details of the customer profile. The action is to be performed asynchronously.

9.2 Design considerations

As we progressed through the application design and development phases, the following facts have been established:

- A locally available database has already been configured to hold details regarding which accounts each customer holds. This takes the simple form of a three-column table where customer user ID is matched to an

account type indicator and an account number. The customer holds a user ID which is the key to the table. A separate table holds the latest customer profile information, acting as a cache for the information in the back-end databases.

- There are currently two remote systems maintained with account details on them, one for savings accounts and one for checking accounts. The customer profile details are held in different forms in each case.
- All databases used are DB2 databases.
- All messages exchanged are valid, well-formed XML documents.

9.2.1 Customer profile lookup

For efficiency, the profile lookup will refer to the locally maintained cache of customer details as its first point of reference. If the details can be found here, they are returned from the details found in the cache.

If no details are found in the cache, the local database is consulted to determine the accounts held by the customer and a database lookup is performed on one of the remote systems where an account is held by that customer.

The local cache is updated from the details found.

9.2.2 Customer profile update

The update message received will be used to update all of the following:

- The local cache of customer details
- The customer details held on the checking account database
- The customer details held on the savings account database

An audit trail of activity is required for systems purposes, but no synchronous response is required by the user from the update request.

9.3 Operational entities

The first logical step in developing the application is to get the supporting structure in place. The databases that will be accessed by the application need to be created, or if existing, defined to the operating system. Any tables required should be in place. Any messages to be used as input or output should also be defined.

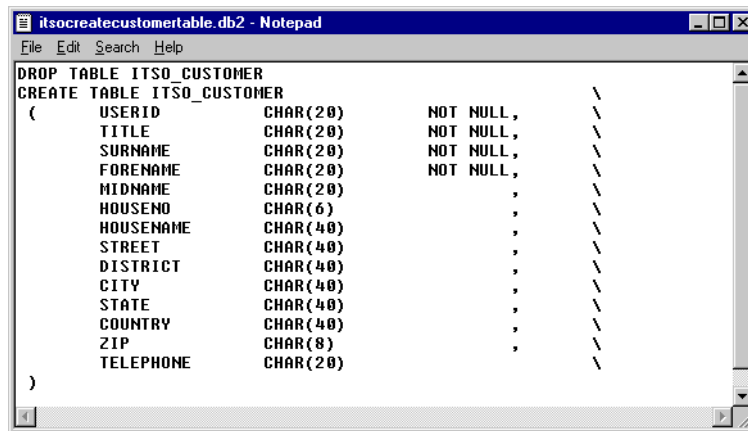
9.3.1 Application databases and tables

Based on the application design, we know we will need a database local to the broker to act as a cache and local lookup, plus the back-end databases. The cache database will be a new database. The back-end databases are assumed to be existing legacy databases, but for the purposes of testing, we will create two new databases representing a checking and a savings database.

9.3.1.1 Local (cache) database

A database local to the broker called “ITSOCUST” will hold the following table definitions:

- ITSOCUSTOMER, the local customer profile details repository (that is, the “cache”).



```
itsocreatecustomertable.db2 - Notepad
File Edit Search Help
DROP TABLE ITSOCUSTOMER
CREATE TABLE ITSOCUSTOMER
(
    USERID          CHAR(20)      NOT NULL,
    TITLE           CHAR(20)      NOT NULL,
    SURNAME         CHAR(20)      NOT NULL,
    FORENAME        CHAR(20)      NOT NULL,
    MIDNAME         CHAR(20)
    HOUSENO        CHAR(6)
    HOUSENAME       CHAR(40)
    STREET          CHAR(40)
    DISTRICT        CHAR(40)
    CITY            CHAR(40)
    STATE           CHAR(40)
    COUNTRY         CHAR(40)
    ZIP             CHAR(8)
    TELEPHONE       CHAR(20)
)
```

Figure 79. ITSOCUSTOMER table

- ITSOCUSTOMER_ACCOUNTS, the local customer accounts repository (that is, customer user ID-to-account type/number matching table) - see Figure 80.

```

itscreatecustomeraccountstable.db2 - Notepad
File Edit Search Help
DROP TABLE ITSO_CUSTOMER_ACCOUNTS
CREATE TABLE ITSO_CUSTOMER_ACCOUNTS
(
    CUSTOMER      CHAR(20)      NOT NULL,
    ACCTYPE       CHAR(1)       NOT NULL,
    ACCOUNT       INTEGER       NOT NULL
)
INSERT INTO ITSO_CUSTOMER_ACCOUNTS
VALUES ( '1', 'C', 82635273 ),
       ( '1', 'S', 76725264 ),
       ( '2', 'C', 82626545 ),
       ( '2', 'S', 72625537 ),
       ( '3', 'C', 82736365 ),
       ( '3', 'S', 73762736 )

```

Figure 80. ITSO_CUSTOMER_ACCOUNTS table

9.3.1.2 Savings accounts database

A remote back-end database will contain a table called ITSO_SAVINGS that will contain the savings accounts information for each customer.

```

itscreatesavingstable.db2 - Notepad
File Edit Search Help
DROP TABLE ITSO_SAVINGS
CREATE TABLE ITSO_SAVINGS
(
    NUMBER        INTEGER       NOT NULL,
    TITLE         CHAR(6)
    SURNAME       CHAR(20)      NOT NULL,
    FORENAMES     CHAR(40)      NOT NULL,
    HOUSENO       CHAR(6)
    HOUSENAME     CHAR(40)
    STREET        CHAR(40)
    DISTRICT      CHAR(40)
    CITY          CHAR(40)
    STATE         CHAR(40)
    COUNTRY       CHAR(40)
    ZIP           CHAR(8)
    PHONENUM      CHAR(20)
    BALANCE       DECIMAL(12,2) NOT NULL
)
INSERT INTO ITSO_SAVINGS
VALUES ( 76725264, 'Mrs.', 'Powell', 'Jan', '11272', NULL, 'Finsbury
Ave.', 'Nuffield', 'Redukville', 'Farley', NULL, NULL, NULL,
71562.28 ),
       ( 72625537, 'Miss.', 'Francis', 'M', '888A', NULL, 'Trenston

```

Figure 81. ITSO_SAVINGS table with test data

9.3.1.3 Checking account database

A remote back-end database will contain a table called ITSO_CHECKING that will contain checking account information for each customer.

```

DROP TABLE ITS0_CHECKING
CREATE TABLE ITS0_CHECKING
(
    NUMBER          INTEGER          NOT NULL,
    NAME            CHAR(40)         NOT NULL,
    ADDRESS1        CHAR(40)         NOT NULL,
    ADDRESS2        CHAR(40)         ,
    ADDRESS3        CHAR(40)         ,
    ADDRESS4        CHAR(40)         ,
    ADDRESS5        CHAR(40)         ,
    TELEPHONE       CHAR(20)         ,
    BALANCE         DECIMAL(12,2)    NOT NULL
)
INSERT INTO ITS0_CHECKING
VALUES ( 82635273, 'J Powell', '11272, Finsbury Avenue,',
'Nuffield', 'Reddockville', 'Farleigh', 'HA 28362', '555-826-7263',
72.38),
( 82626545, 'Matt Francis', '888A, Trenston Court,', 'Winton
Artford', 'Grenswillow', NULL, 'CW 26253', '555-838-7252', 7263.11),
( 82736365, 'Michelle Anna Brown', 'Highfields,', 'Spanton
Hill', 'Dustonbury', NULL, 'GI 92173', '555-927-9172', 54143.22)

```

Figure 82. ITS0_CHECKING table with test data

9.3.2 Messages and documents

The WebSphere application will work with messages in the form of XML documents. These must be agreed upon by the MQSI application programmer and the WebSphere application programmer. Message formats are defined for MQSI through the Control Center and stored in the message repository. A DTD can be generated from the message definition to be passed to the WebSphere application programmer. In our discussions, we will use the DTDs as a common way to describe the messages used in our application.

9.3.2.1 DTD for “customerrequest” XML document

The following DTD describes the customerrequest XML document used by WebSphere and MQSI:

```

<!-- customerrequest.dtd -->
<!ELEMENT customerrequest (userid)>
<!ELEMENT userid (#PCDATA)>

```

9.3.2.2 DTD for “profilemessage” XML document.

The following DTD describes the profilemessage XML document used by WebSphere and MQSI:

```

<!--profile.dtd -->
<!ELEMENT profilemessage (userid,custname,custaddr)>
<!ELEMENT userid (#PCDATA)>

```

```

<!ELEMENT custname (namepref, forename, middlename, surname)>
<!ELEMENT namepref (#PCDATA)>
<!ELEMENT forename (#PCDATA)>
<!ELEMENT middlename (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT custaddr (houenum, housename, street, district, city, state,
country, zip, telephone)>
<!ELEMENT houenum (#PCDATA)>
<!ELEMENT housename (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT district (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>

```

9.4 Identify the general operations

Next, we break down each of the application contract requirements into component operations. From this, we will determine common operational components, which may be transformations or operations.

9.4.1 Customer profile lookup operational components

The following operational components make up the customer profile lookup contract.

- Receive the “customerrequest” message
- Find the profile in ITSO_CUSTOMER
- Find the customer accounts list from ITSO_CUSTOMER_ACCOUNTS
- Choose an account for reference
- Get the customer’s savings account information from ITSO_SAVINGS
- Get the customer’s checking account information in ITSO_CHECKING
- Update the profile in ITSO_CUSTOMER
- Create “profilemessage”
- Return “profilemessage” to application

Not all of these may be involved in any one invocation of the process.

9.4.2 Customer profile update operational components

The following operational components make up the customer profile update contract.

- Receive the “profilemessage”

- Update customer profile in ITSO_CUSTOMER
- Find the list of customer accounts in ITSO_CUSTOMER_ACCOUNTS
- Update customer profile in ITSO_SAVINGS
- Update customer profile in ITSO_CHECKING
- Write to audit trail

9.5 Identify the operational components

Having defined general operations to be performed we can divide these into functional components. This is done in order to decide which operations should be developed as reusable components.

Even in the case of operations that are used only once in a given scenario, it is a good idea to define the operation in a re-usable form.

9.5.1 Customer profile lookup functional components

The following functions will be required to look up a customer profile:

1. Use “customerrequest” to look up details in ITSO_CUSTOMER and create “profilemessage”.
2. Use “customerrequest” to look up accounts in ITSO_CUSTOMER_ACCOUNTS and pass the details on.
3. Transform details from “customerrequest” and row from ITSO_SAVINGS to create “profilemessage”.
4. Transform details from “customerrequest” and row from ITSO_CHECKING to create “profilemessage”.

We can see the above is left somewhat open-ended. The main questions are concerned with how we take what we learn in (2) and use it in either (3) or (4).

9.5.1.1 DTD for “customeraccounts” XML document

New document types will be needed to enable the transformation operations to be linked together. First, we define a new XML document type that will combine the details from “customerrequest” and the result of the ITSO_CUSTOMER_ACCOUNTS table lookup. We call this “customeraccounts”, and the DTD looks like this:

```
<!--customeraccounts.dtd -->
<!ELEMENT customeraccounts (userid,account?)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT account (CUSTOMER, ACCTYPE, ACCOUNT)>
<!ELEMENT CUSTOMER (#PCDATA)>
<!ELEMENT ACCTYPE (#PCDATA)>
```

```
<!ELEMENT ACCOUNT (#PCDATA)>
```

Note how the structure of the “account” element child elements precisely match the columns found in the ITSO_CUSTOMER_ACCOUNT table. “account” has been set up as an optional element (denoted by the “?”) because we do not know how many accounts a customer will hold.

A further transformation will be required, to take the “customeraccounts” document described above, examine the first child of the “account” element, and based on the value of the “ACCTYPE” child element, create either a “savings” document or a “checking” document.

9.5.1.2 DTD for “savings” XML document

We will need a new document type as input to functional component item 3 on page 189. The new document type, called *savings*, combines the “userid” with details we find on the ITSO_SAVINGS table for the customer.

```
<!-- savings.dtd -->
<!ELEMENT savings (userid,account?)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT account (NUMBER, TITLE, SURNAME, FORENAMES, HOUSENO, HOUSENAME,
STREET, DISTRICT, CITY, STATE, COUNTRY, ZIP, PHONENUM, BALANCE)>
<!ELEMENT NUMBER (#PCDATA)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT SURNAME (#PCDATA)>
<!ELEMENT FORENAMES (#PCDATA)>
<!ELEMENT HOUSENO (#PCDATA)>
<!ELEMENT HOUSENAME (#PCDATA)>
<!ELEMENT STREET (#PCDATA)>
<!ELEMENT DISTRICT (#PCDATA)>
<!ELEMENT CITY (#PCDATA)>
<!ELEMENT STATE (#PCDATA)>
<!ELEMENT COUNTRY (#PCDATA)>
<!ELEMENT ZIP (#PCDATA)>
<!ELEMENT PHONENUM (#PCDATA)>
<!ELEMENT BALANCE (#PCDATA)>
```

9.5.1.3 DTD for “checking” XML document

We will need a new document type as input to functional component item 4 on page 189. The new document type, called *checking*, combines the “userid” with details we find on the ITSO_CHECKING table for the customer.

```
<!-- checking.dtd -->
<!ELEMENT checking (userid,account?)>
<!ELEMENT userid (#PCDATA)>
```

```

<!ELEMENT account (NUMBER, NAME, ADDRESS1, ADDRESS2, ADDRESS3, ADDRESS4,
ADDRESS5, TELEPHONE, BALANCE)>
<!ELEMENT NUMBER (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT ADDRESS1 (#PCDATA)>
<!ELEMENT ADDRESS2 (#PCDATA)>
<!ELEMENT ADDRESS3 (#PCDATA)>
<!ELEMENT ADDRESS4 (#PCDATA)>
<!ELEMENT ADDRESS5 (#PCDATA)>
<!ELEMENT TELEPHONE (#PCDATA)>
<!ELEMENT BALANCE (#PCDATA)>

```

9.5.2 Customer profile update functional components

Having defined the general operations for the profile update, we take this and divide it into functional components. The profile update can be considered to consist of the following functional steps, using intermediate document types where necessary:

1. Retrieve the “profilemessage” document containing the updated profile details.
2. Use the “profilemessage” document to update the customer details in cache (table ITSO_CUSTOMER of database ITSOCUST).
3. Find the accounts for the customer.
4. Combine “profilemessage” and accounts into one document - call this “profileaccounts”.
5. Loop through accounts and for each account create an “updatemessage” containing the account and new profile.
6. Route the “updatemessage” to the correct database update function according to account type.
7. Apply “updatemessage” to table ITSO_SAVINGS of database ITSOSAVI where applicable.
8. Apply “updatemessage” to table ITSO_CHECKING of database ITSOCHEC where applicable.
9. Do all in one coordinated transaction.

A few new entities have been born in this analysis. These are:

- The “profileaccounts” intermediate message - a proposed DTD that is defined below.

- An “updatemessage” intermediate document, which contains the account identifier together with all the details from the original “profilemessage” - again, the DTD is shown below.
- A looping mechanism that can count through a repeating “account” element, creating an “updatemessage” for each instance found.
- A filter that can route “updatemessage” documents based on target account type.

The “profileaccounts” DTD will look like this:

```
<!-- profileaccounts.dtd -->
<!ELEMENT profileaccounts (profile,account?)>
<!ELEMENT profile (userid,custname,custaddr)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT custname (namepref, forename, middlename, surname)>
<!ELEMENT namepref (#PCDATA)>
<!ELEMENT forename (#PCDATA)>
<!ELEMENT middlename (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT custaddr (houenum, housename, street, district, city, state,
country, zip, telephone)>
<!ELEMENT houenum (#PCDATA)>
<!ELEMENT housename (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT district (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT account (CUSTOMER,ACCTYPE,ACCOUNT)>
<!ELEMENT CUSTOMER (#PCDATA)>
<!ELEMENT ACCTYPE (#PCDATA)>
<!ELEMENT ACCOUNT (#PCDATA)>
```

The “updatemessage” DTD will look like this - note the subtle difference that the “account” element is not optional:

```
<!-- updatemessage.dtd -->
<!ELEMENT updatemessage (profile,account)>
<!ELEMENT profile (userid,custname,custaddr)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT custname (namepref, forename, middlename, surname)>
<!ELEMENT namepref (#PCDATA)>
<!ELEMENT forename (#PCDATA)>
```



```

<!ELEMENT middlename (#PCDATA)>
<!ELEMENT surname (#PCDATA)>
<!ELEMENT custaddr (houenum, housename, street, district, city, state,
country, zip, telephone)>
<!ELEMENT houenum (#PCDATA)>
<!ELEMENT housename (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT district (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT telephone (#PCDATA)>
<!ELEMENT account (CUSTOMER,ACCTYPE,ACCOUNT)>
<!ELEMENT CUSTOMER (#PCDATA)>
<!ELEMENT ACCTYPE (#PCDATA)>
<!ELEMENT ACCOUNT (#PCDATA)>

```

A loop construct will be required to take a document of type “profileaccounts” and transform it into a set of “updatemessage” documents.

A router will examine the content of the “ACCTYPE” child element from the “account” data element of the “updatemessage” document, and route it to the correct subflow that updates the applicable account database table.

We now have all the necessary transformations defined. Now let’s start building message flows.

9.6 Building the message flows

The development decisions are well under way and now it is time to start building the MQSI message flows. The rest of this chapter shows how the message flows were built for the WebBank application using the MQSI Control Center.

9.6.1 Creating a message flow

To create a message flow, select the **Message Flows** tab of the MQSI Control Center. In the left pane, right-click **Message Flows** and select **Create -> Message Flow**.

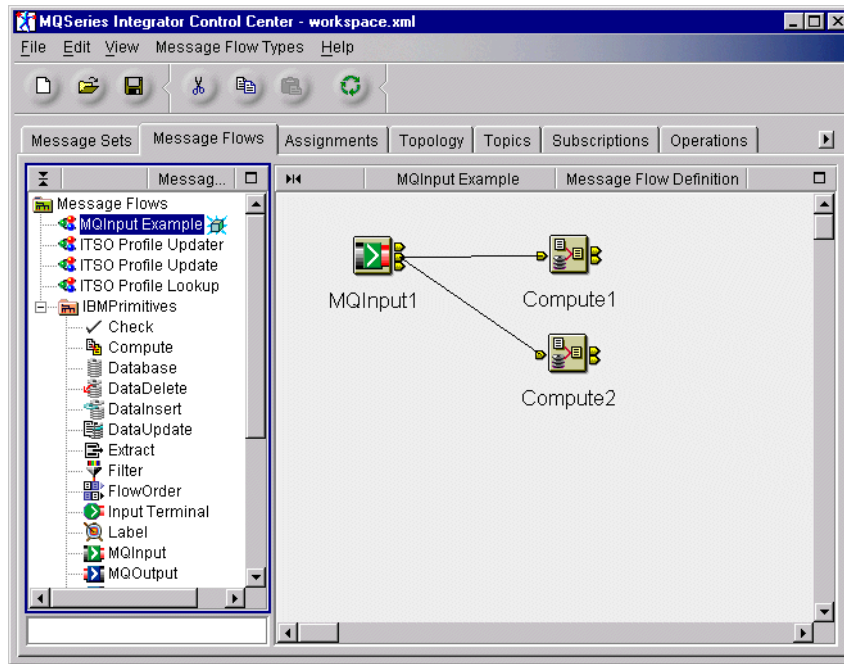


Figure 83. Building message flows with the Control Center

Message flows are built by dragging message flow nodes from the Message Flow Types pane (the leftmost pane) to the Message Flow Definition pane (the rightmost pane) and connecting the input/output terminals of the nodes together.

It is possible to set up a connection between a given output terminal and the input terminals of more than one node. In this way, one incoming message can be made to cause more than one flow of events. For instance, a message designed to carry a change of name and address details of a customer could be connected to several subflows, each of which is designed to update a different back-end system.

9.6.2 Organizing message flows

Since message flow development involves the development of message flows as components, large sets of message flow components can result. It is useful to organize message flows into *categories*. By creating a category, message flow components can be organized into logical groups according to their use.

We will be placing our message flows into the following categories:

- ITSO Deployable Message Flows
- ITSO Flow Control
- ITSO Database Operations
- ITSO Cache Operations
- ITSO Transformations

To create a message flow category, select the **Message Flows** tab in the Control Center, right-click **Message Flows** and select **Create -> Message Flow Category**. To create a new message flow within a category, simply right-click the category folder and select **Create -> Message Flow**.

You will see these categories in the Message Flows pane (left pane) of the Control Center as we build our message flows.

9.6.3 Message flow: “ITSO Cache Lookup: Profile from Request”

Figure 84 shows an outline of the message flow for the cache lookup.

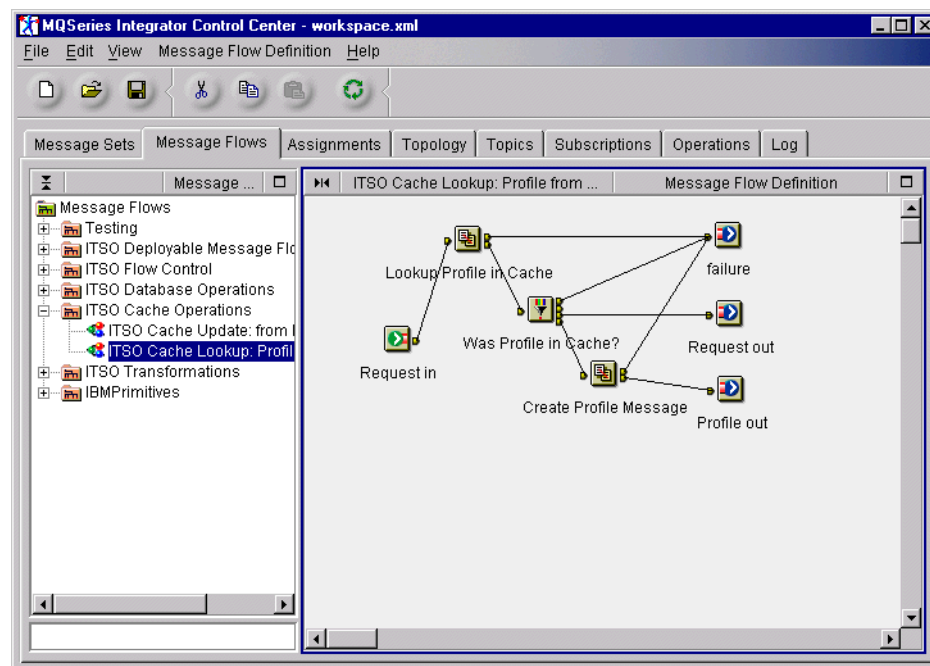


Figure 84. The ITSO Cache Lookup: profile from update

The purpose of this message flow is to take a “customerrequest” message and transform it into a “profilemessage” message, by looking up the customer

details from the cache, in this case, the ITSO_CUSTOMER table of the ITSOCUST database.

Input

Input is taken in via the InputTerminal *Request in*, in the form of a “customerrequest” XML document.

Output

When an entry can be found in the cache, this message flow creates a “profilemessage” XML document. The document is passed out through the OutputTerminal node named *Profile out*.

When an entry is not found in the cache, the original “customerrequest” message is passed intact, through the OutputTerminal named *Request out*.

9.6.3.1 Compute node: “Lookup Profile in Cache”

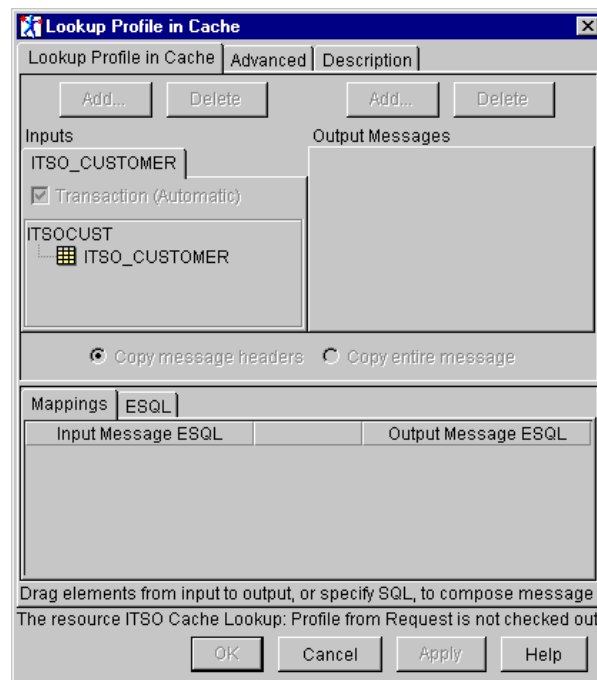


Figure 85. Compute node: Lookup Profile in Cache

This compute node contains the following ESQL:

```
DECLARE I INTEGER;  
SET I = 1;
```

```

WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag) "customerrequest"."userid" =
    "InputBody".(XML.tag) "customerrequest"."userid";
SET "OutputRoot"."XML".(XML.tag) "customerrequest"."profilequeryresult" [] =
(
    SELECT T.*
    FROM Database.ITSO_CUSTOMER AS T
    WHERE T.USERID = TRIM("InputBody".(XML.tag) "customerrequest"."userid")
);

```

In creating the output document, the ESQL

- Copies the message headers (the section before the comment line)
- Copies the XML document header
- Copies the “userid” element
- Adds a new data element document called “profilequeryresult”. This is populated with the elements retrieved from a query against the database. The query retrieves all details from the ITSO_CUSTOMER_ACCOUNTS table for the incoming “userid”.

The result is passed to the out terminal of the node which is connected to the in terminal of the *Was Profile in Cache?* node.

All internal failure conditions are wired to an OutputTerminal labelled *failure*.

9.6.3.2 Filter node: “Was Profile In Cache?”

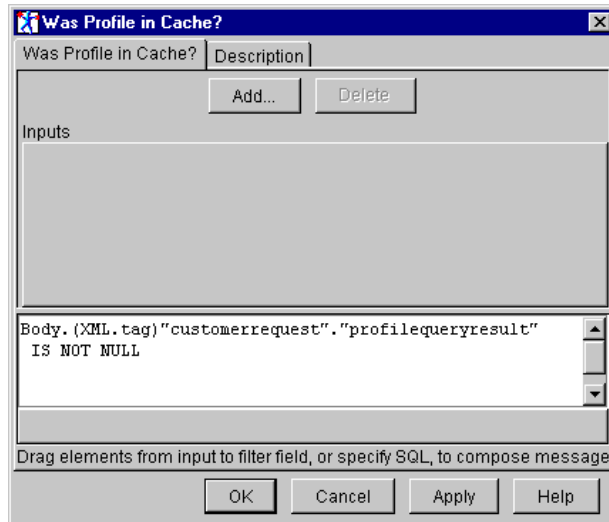


Figure 86. Filter node - Was Profile In Cache

This filter node evaluates the expression shown in the above figure.

If the query against ITSO_CUSTOMER_ACCOUNTS has found data, “profilequeryresult” will not be null and the expression evaluates to *true*. If no data is found, the *false* condition is met.

On *false*, the original message is propagated to the OutputTerminal named *Request out*.

On *true*, the transformed message is passed to the *in* terminal of the *Create Profile Message* node.

All internal failure conditions are wired to an OutputTerminal labelled *failure*.

9.6.3.3 Compute node: “Create Profile Message”

This compute node contains the following ESQL transformation code:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
causing any modifications to be lost.
```

```

SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."userid" =
"InputBody".(XML.tag)"customerrequest"."userid";
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custname" = THE
(
  SELECT
    TRIM(T."TITLE") AS "namepref",
    TRIM(T."FORENAME") AS "forename",
    TRIM(T."MIDNAME") AS "middlename",
    TRIM(T."SURNAME") AS "surname"
  FROM InputBody.(XML.tag)"customerrequest"."profilequeryresult" AS T
);
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr" = THE
(
  SELECT
    TRIM(T."HOUSENO") AS "houenum",
    TRIM(T."HOUSENAME") AS "houename",
    TRIM(T."STREET") AS "street",
    TRIM(T."DISTRICT") AS "district",
    TRIM(T."CITY") AS "city",
    TRIM(T."STATE") AS "state",
    TRIM(T."COUNTRY") AS "country",
    TRIM(T."ZIP") AS "zip",
    TRIM(T."TELEPHONE") AS "telephone"
  FROM InputBody.(XML.tag)"customerrequest"."profilequeryresult" AS T
);

```

This is a straightforward message transformation compute node that uses ESQL to compile an output XML message of document type “profilemessage” from the input XML message of document type “customerrequest”.

You will see from the complete ESQL properties content included here that two distinct SELECT operations are used to populate the output data elements “custname” and “custaddr” from the content of “profilequeryresult” that was loaded by the database lookup performed in the *Lookup Profile in Cache* node.

This technique is described in greater detail in 7.4.3.2, “Using ESQL for message transformation” on page 123.

On successful transformation, the “profilemessage” document is routed to the OutputTerminal named *Profile out*.

All internal failure conditions are wired to an OutputTerminal labelled *failure*.

9.6.4 Message flow: “ITSO Cache Update: from Profile”

The purpose of this message flow is to take an XML document of type “profilemessage” as input and, based on its contents, update the details held in the table ITSO_CUSTOMER of database ITSOCUST.

This may involve adding a new row to the table if one does not already exist for the customer, or updating the existing row if a row already exists for the customer. Figure 87 shows the message flow.

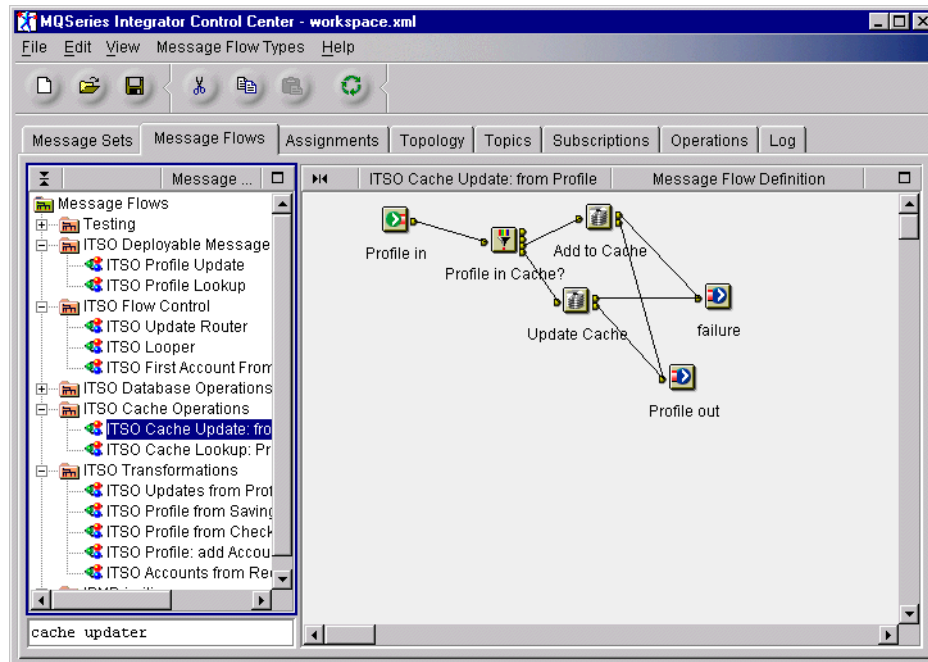


Figure 87. ITSO Cache Update: from Profile

Input

Input is taken via the InputTerminal *Profile in*, in the form of a “profilemessage” XML document.

Output

This message flow passes the document unchanged through *Profile out*.

9.6.4.1 Filter node: “Profile in Cache?”

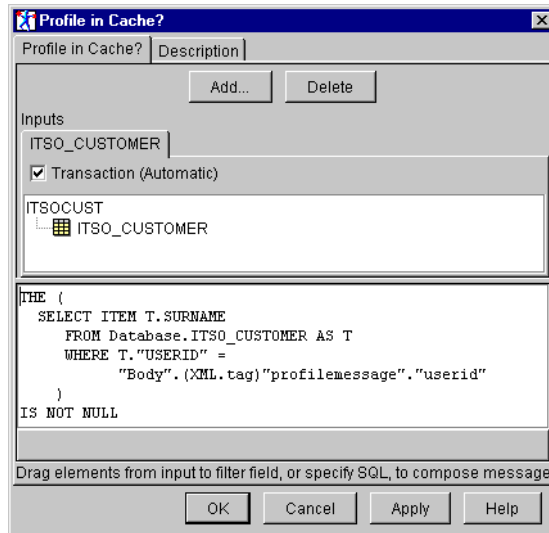


Figure 88. Profile in Cache? node

This filter node involves a database select as part of its expression. The “userid” from the incoming message is used as the search criterion against the ITSO_CUSTOMER table of the ITSOCUST database. If data is found matching the “userid”, the select operation evaluates to “not null”, and so the Filter expression evaluates true. If no row is found, the expression evaluates false.

In the true condition, the message is routed to the *in* terminal of the *Update Cache* database node. In the false condition, the message is routed to the *in* terminal of the *Add to Cache* database node.

9.6.4.2 Database node: “Add to Cache”

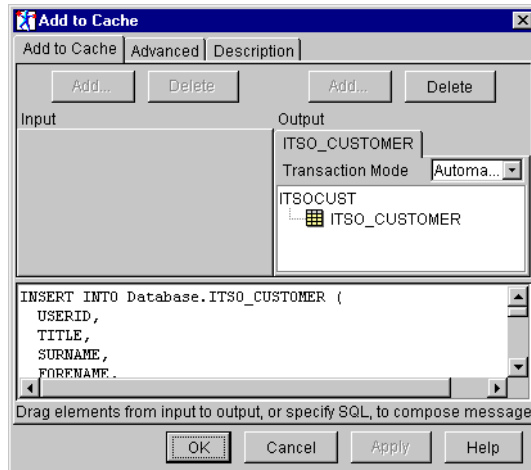


Figure 89. Add to Cache

This database node takes a “profilemessage” document as input and uses its contents to add a new entry to the ITSO_CUSTOMER table of the ITSOCUST database.

The ESQL that performs this transformation is detailed here:

```
INSERT INTO Database.ITSO_CUSTOMER (
    USERID,
    TITLE,
    SURNAME,
    FORENAME,
    MIDNAME,
    HOUSENO,
    HOUSENAME,
    STREET,
    DISTRICT,
    CITY,
    STATE,
    COUNTRY,
    ZIP,
    TELEPHONE
) VALUES (
    "Body".(XML.tag) "profilemessage". "userid",
    "Body".(XML.tag) "profilemessage". "custname". "namepref",
    "Body".(XML.tag) "profilemessage". "custname". "surname",
    "Body".(XML.tag) "profilemessage". "custname". "forename",
    "Body".(XML.tag) "profilemessage". "custname". "middlename",
```

```

"Body".(XML.tag) "profilemessage"."custaddr"."houenum",
"Body".(XML.tag) "profilemessage"."custaddr"."houename",
"Body".(XML.tag) "profilemessage"."custaddr"."street",
"Body".(XML.tag) "profilemessage"."custaddr"."district",
"Body".(XML.tag) "profilemessage"."custaddr"."city",
"Body".(XML.tag) "profilemessage"."custaddr"."state",
"Body".(XML.tag) "profilemessage"."custaddr"."country",
"Body".(XML.tag) "profilemessage"."custaddr"."zip",
"Body".(XML.tag) "profilemessage"."custaddr"."telephone"
);

```

9.6.4.3 Database node: “Update Cache”

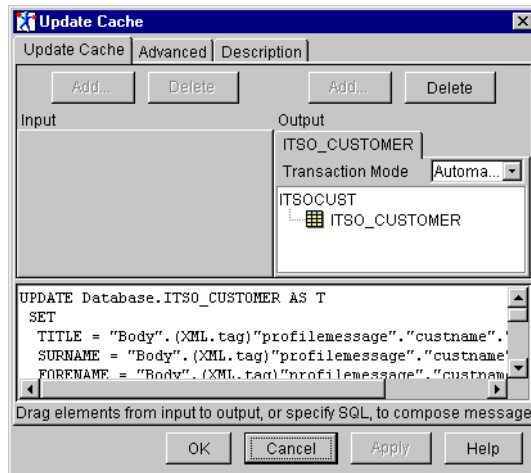


Figure 90. Update Cache

This database node takes a “profilemessage” document as input and uses its contents to update the matching entry to the ITSO_CUSTOMER table of the ITSOCUST database.

The ESQL that performs this transformation is detailed here:

```

UPDATE Database.ITSO_CUSTOMER AS T
SET
  TITLE = "Body".(XML.tag) "profilemessage"."custname"."namepref",
  SURNAME = "Body".(XML.tag) "profilemessage"."custname"."surname",
  FORENAME = "Body".(XML.tag) "profilemessage"."custname"."forename",
  MIDNAME = "Body".(XML.tag) "profilemessage"."custname"."middlename",
  HOUSENO = "Body".(XML.tag) "profilemessage"."custaddr"."houenum",
  HOUSENAME = "Body".(XML.tag) "profilemessage"."custaddr"."houename",
  STREET = "Body".(XML.tag) "profilemessage"."custaddr"."street",
  DISTRICT = "Body".(XML.tag) "profilemessage"."custaddr"."district",

```

```

CITY = "Body".(XML.tag)"profilemessage"."custaddr"."city",
STATE = "Body".(XML.tag)"profilemessage"."custaddr"."state",
COUNTRY = "Body".(XML.tag)"profilemessage"."custaddr"."country",
ZIP = "Body".(XML.tag)"profilemessage"."custaddr"."zip",
TELEPHONE = "Body".(XML.tag)"profilemessage"."custaddr"."telephone"
WHERE T."USERID" = "Body".(XML.tag)"profilemessage"."userid";

```

9.6.5 Message flow: “ITSO Accounts from Request”

The purpose of this message flow is to take an input document of type “customerrequest” and look up all account number details held for that customer from the ITSO_CUSTOMER_ACCOUNTS table. It uses the result of this query to create a document of type “customeraccounts” as output.

Figure 91 shows an outline of the message flow.

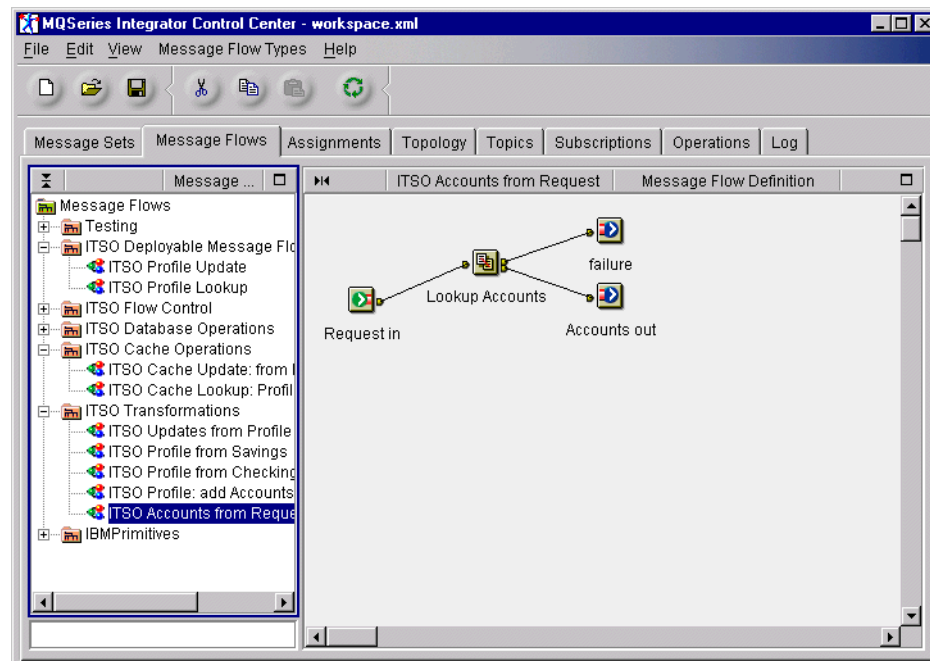


Figure 91. ITSO Accounts from Request

Input

Input is taken via the InputTerminal *Request in*, in the form of a “customerrequest” document.

Output

This message flow creates a “customeraccounts” document. The document is passed out through the OutputTerminal node named *Accounts out*.

The *failure* terminal of the internal compute node is wired to a similarly named OutputTerminal, to allow propagation of the failure condition when required.

9.6.5.1 Compute node: “Lookup Accounts”

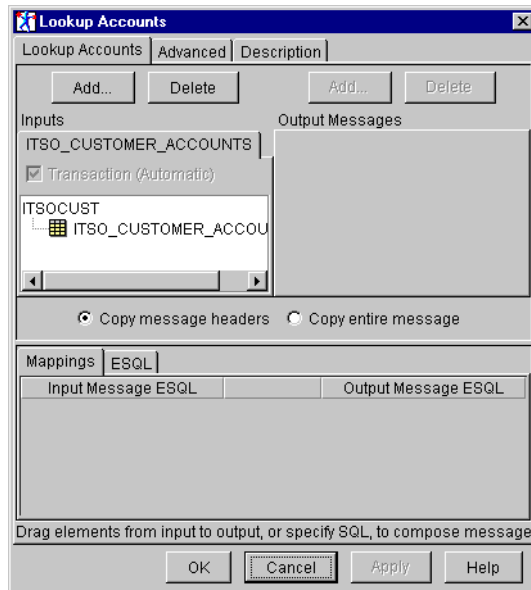


Figure 92. Compute node: Lookup Accounts

Here we see a transformation compute node that uses table `ITSO_CUSTOMER_ACCOUNTS` from database `ITSOCUST` as a data source.

The ESQL behind this compute node is shown here:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
```

```

SET "OutputRoot"."XML".(XML.tag)"customeraccounts"."userid" =
"InputBody".(XML.tag)"customerrequest"."userid";
SET "OutputRoot"."XML".(XML.tag)"customeraccounts"."account"[] =
(
  SELECT T.*
  FROM Database.ITSO_CUSTOMER_ACCOUNTS AS T
  WHERE T.CUSTOMER = "InputBody".(XML.tag)"customerrequest"."userid"
);

```

The ESQL creates a new XML document of type “customeraccounts”. To the “customeraccounts” data element, the ESQL adds:

- The “userid” data element taken from the input “customerrequest” document
- A new repeating element called “account”, which is in turn populated by a query to retrieve all rows from ITSO_CUSTOMER_ACCOUNTS matching the input “userid”.

The transformed message is routed to the OutputTerminal named *Accounts out*.

9.6.6 Message flow: “ITSO First Account From Accounts”

The purpose of this message flow is to take an input document of type “customeraccounts” and select the first occurrence of its “account” data element. The flow will examine the value of the “ACCTYPE” child element from this occurrence and based on the value found look up the account details from one or other of the ITSO_SAVINGS or ITSO_CHECKING tables. According to the route taken at this decision point, the output will either be a “checking” document at the *Checking out* OutputTerminal or a “savings” document at the *Savings out* OutputTerminal.

Figure 93 shows the message flow.

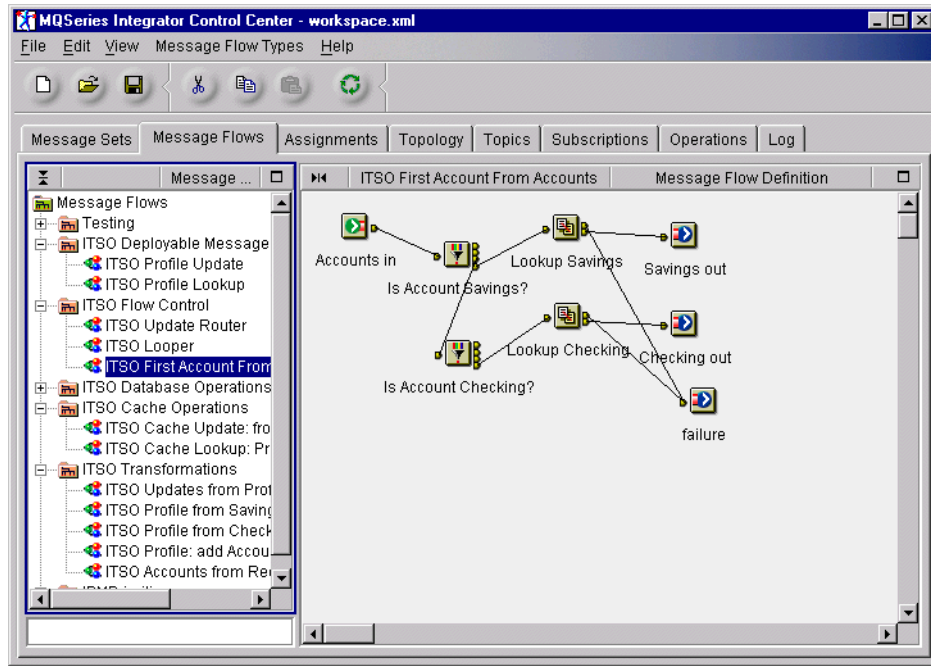


Figure 93. ITSO First Account from Accounts

Input

Input is taken via the InputTerminal *Accounts in*, in the form of a “customeraccounts” document.

Output

If the first account is a savings account, the output via the *Savings out* terminal is a “savings” document.

If the first account is a checking account, the output via the *Checking out* terminal is a “checking” document.

9.6.6.1 Filter node: “Is Account Savings?”

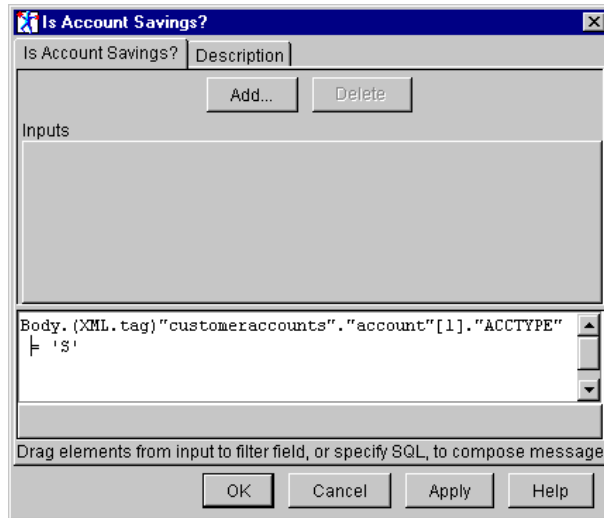


Figure 94. *Is Account Savings?*

This node examines the “ACCTYPE” child element from the first “account” element of the incoming “customeraccounts” message. If the value is “S” this is a savings account and the filter node propagates the incoming message to its *true* terminal, sending it on to the *Lookup Savings* node. If not, the incoming message is sent to the *Is Account Checking* node.

9.6.6.2 Compute node: “Lookup Savings”

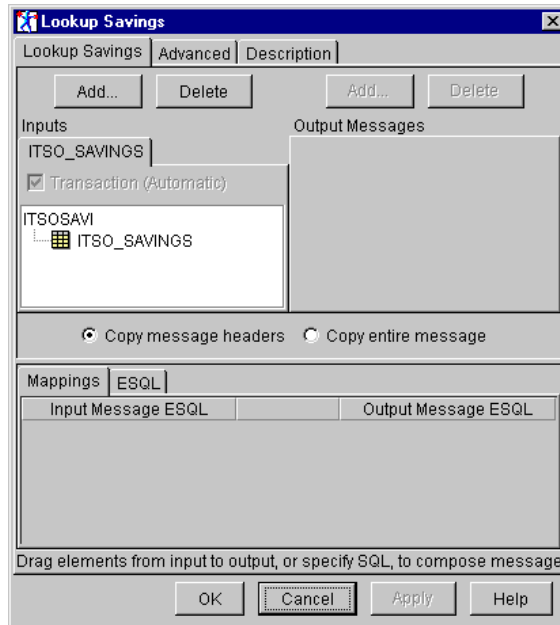


Figure 95. Compute node: Lookup Savings

The transformation ESQL behind this node is as follows:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line.  SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET OutputRoot."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET OutputRoot."XML".(XML.tag) "savings"."customer" =
InputBody.(XML.tag) "customeraccounts"."userid";
SET OutputRoot."XML".(XML.tag) "savings"."account" =
THE (
    SELECT T.*
    FROM "Database"."ITSO_SAVINGS" AS T
    WHERE T."NUMBER" =
    "InputBody".(XML.tag) "customeraccounts"."account" [1] ."ACCOUNT"
);
```

The ESQL code creates a new output XML document of type “savings”, into which it adds the following:

- The incoming “userid” value, to a new element name of “customer”.
- The result of a database query that selects the row from ITSO_SAVINGS for the account number matching the “ACCOUNT” child element from the first “account” element of the input “customeraccounts” document.

The transformed message is sent to the OutputTerminal named *Savings out*.

9.6.6.3 Filter node: “Is Account Checking?”

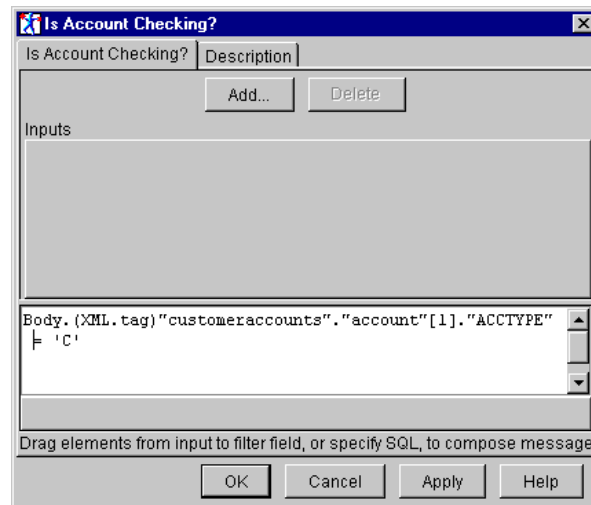


Figure 96. *Is Account Checking?*

This node examines the “ACCTYPE” child element from the first “account” element of the incoming “customeraccounts” message. If the value is “C”, this is a checking account, and the filter node propagates the incoming message to its *true* terminal, sending it on to the *Lookup Checking* node.

9.6.6.4 Compute node: “Lookup Checking”

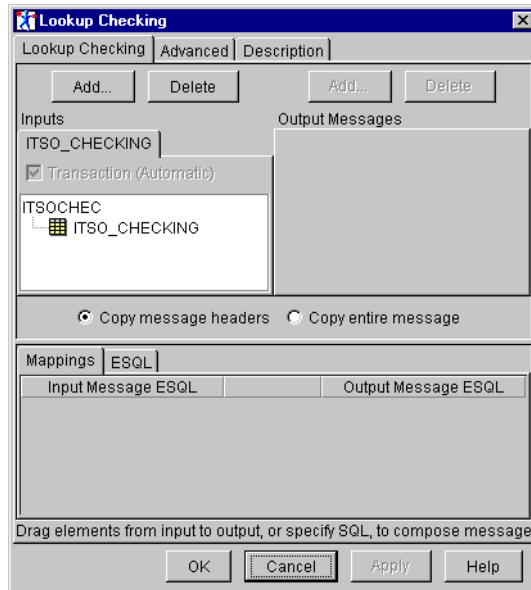


Figure 97. Compute node: Lookup Checking

The transformation ESQL behind this node is as follows:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET OutputRoot."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET OutputRoot."XML".(XML.tag)"checking"."customer" =
InputBody.(XML.tag)"customeraccounts"."userid";
SET OutputRoot."XML".(XML.tag)"checking"."account" =
THE (
    SELECT T.*
    FROM "Database"."ITSO_CHECKING" AS T
    WHERE T."NUMBER" =
"InputBody".(XML.tag)"customeraccounts"."account"[1]."ACCOUNT"
);
```

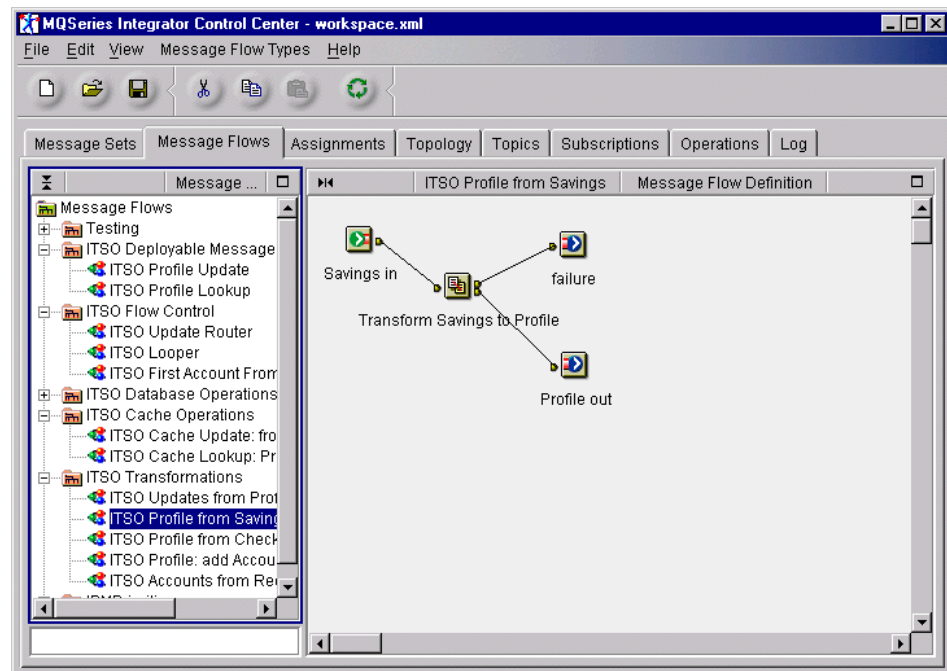
The ESQL code creates a new output XML document of type “checking”, into which it adds:

- The incoming “userid” value, to a new element name of “customer”.
- The result of a database query that selects the row from ITSO_CHECKING for the account number matching the “ACCOUNT” child element from the first “account” element of the input “customeraccounts” document.

The transformed message is sent to the OutputTerminal named *Checking out*.

9.6.7 Message flow: “ITSO Profile from Savings”

The purpose of this message flow is to take an input document of type “savings” and use this to create a document of type “profilemessage”. The following figure shows an outline of the message flow.



Input

Input is taken via the InputTerminal *Savings in*, in the form of a “savings” document.

Output

Output via the *Profile out* terminal is a “profilemessage” document.

9.6.7.1 Compute node: Transform Savings to Profile

This is a straightforward message transformation compute node that uses ESQL to compile an output XML message of document type “profilemessage” from the input XML message of document type “savings”.

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line.  SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."userid" =
    "InputBody".(XML.tag)"savings"."customer";
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custname" =
    THE (
        SELECT
            TRIM(T."TITLE") AS "namepref",
            TRIM(T."FORENAMES") AS "forename",
            '' AS "middlename",
            TRIM(T."SURNAME") AS "surname"
        FROM InputBody.(XML.tag)"savings"."account" AS T
    );
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr" =
    THE (
        SELECT
            TRIM(T."HOUSENO") AS "houenum",
            TRIM(T."HOUSENAME") AS "houename",
            TRIM(T."STREET") AS "street",
            TRIM(T."DISTRICT") AS "district",
            TRIM(T."CITY") AS "city",
            TRIM(T."STATE") AS "state",
            TRIM(T."COUNTRY") AS "country",
            TRIM(T."ZIP") AS "zip",
            TRIM(T."TELEPHONE") AS "telephone"
        FROM InputBody.(XML.tag)"savings"."account" AS T
    );
```

You will see from the complete ESQL properties content included here that two distinct SELECT operations are used to populate the output data elements “custname” and “custaddr” from the content of “savings” that was loaded by the database lookup performed in the earlier compute node described in 9.6.6.2, “Compute node: “Lookup Savings”” on page 209.

This technique is described in greater depth in 7.4.3.2, “Using ESQL for message transformation” on page 123.

On successful transformation, the “profilemessage” document is routed to the OutputTerminal named *Profile out*.

All internal failure conditions are wired to a new OutputTerminal labelled *failure*.

9.6.8 Message flow: “ITSO Profile from Checking”

The purpose of this message flow is to take an input document of type “checking” and use this to create a document of type “profilemessage”. The following figure shows an outline of the message flow.

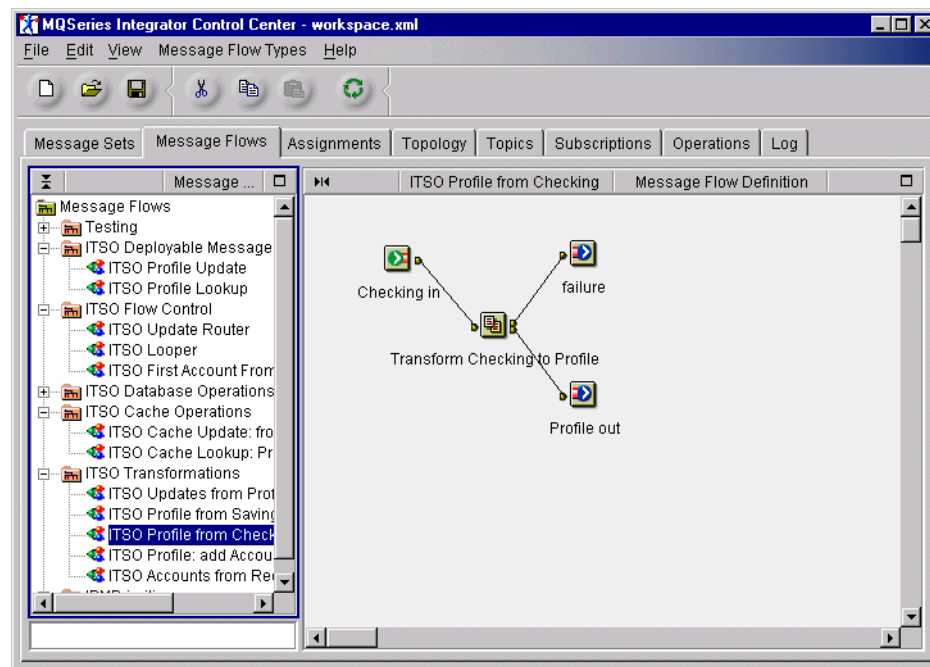


Figure 98. ITSO Profile from Checking

Input

Input is taken via the InputTerminal *Checking in*, in the form of a “checking” document.

Output

Output via the *Profile out* terminal is a “profilemessage” document.

9.6.8.1 Compute node: “Transform Checking to Profile”

This is a more involved transformation node controlled by ESQL, detailed here:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line.  SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."userid" =
    "InputBody".(XML.tag)"checking"."customer";

DECLARE Pos INT;
DECLARE Len INT;
DECLARE NameField CHAR;
DECLARE NamePart CHAR;
DECLARE HouseThing CHAR;

/**
 * this code parses the name field
 */
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custname"."namepref" =
    '';

SET NameField = TRIM(InputBody.(XML.tag)"checking"."account"."NAME");

SET Pos = POSITION(' ' IN NameField);
SET Len = LENGTH(NameField);
IF Pos > 0 THEN
    SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custname"."forename" =
        SUBSTRING(NameField FROM 1 FOR (Pos - 1));
    SET NameField = SUBSTRING(NameField FROM (Pos + 1) FOR (Len - Pos));
    SET Pos = POSITION(' ' IN NameField);
    SET Len = LENGTH(NameField);
    IF Pos > 0 THEN
        SET
            "OutputRoot"."XML".(XML.tag)"profilemessage"."custname"."middlename" =
                SUBSTRING(NameField FROM 1 FOR (Pos - 1));
        SET NameField = SUBSTRING(NameField FROM (Pos + 1) FOR (Len - Pos));
        SET Pos = POSITION(' ' IN NameField);
        SET Len = LENGTH(NameField);
    END IF;
END IF;
```

```

SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custname"."surname" =
NameField;

/**
 * this code parses the first line of the address
 * if there is a comma, the substring is used either as a "houenum" or
"housename" based on its length
 */
SET Pos = POSITION(',', ' IN
InputBody.(XML.tag)"checking"."account"."ADDRESS1");
SET Len = LENGTH(InputBody.(XML.tag)"checking"."account"."ADDRESS1");
IF Pos > 0 THEN
    SET HouseThing =
        SUBSTRING(InputBody.(XML.tag)"checking"."account"."ADDRESS1"
            FROM 1 FOR (Pos - 1)
        );
    IF LENGTH(HouseThing) <= 6 THEN
        SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr"."houenum"
= HouseThing;
        SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr"."housename"
= '';
    ELSE
        SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr"."houenum"
= '';
        SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr"."housename"
= HouseThing;
    END IF;
END IF;
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr"."street" =
    TRIM(
        SUBSTRING(
            InputBody.(XML.tag)"checking"."account"."ADDRESS1"
            FROM (Pos + 1) FOR (Len - Pos)
        )
    );
/**
 * this code populates the remainder of the address fields
 */
SET "OutputRoot"."XML".(XML.tag)"profilemessage"."custaddr" =
THE (
    SELECT
        TRIM(T."ADDRESS2") AS "district",
        TRIM(T."ADDRESS3") AS "city",
        TRIM(T."ADDRESS4") AS "state",
        TRIM(T."ADDRESS5") AS "country",
        '' AS "zip",

```



```
    TRIM(T."TELEPHONE") AS "telephone"  
FROM InputBody.(XML.tag)"checking"."account" AS T  
);
```

This compute node uses ESQL to compile an output message of document type “profilemessage” from the input message of document type “checking”.

You will see from the complete ESQL properties content included here that some complex code is used to populate the output data elements “custname” and “custaddr” from the content of “checking” that was loaded by the database lookup performed in the earlier compute node described in 9.6.6.4, “Compute node: “Lookup Checking”” on page 211.

You will observe the use of the DECLARE function to create internal variables that are used to help manage the transformation logic.

The SUBSTRING, POSITION and LENGTH functions are used to facilitate string manipulation.

The TRIM function is used to remove leading and trailing spaces from strings.

On successful transformation, the “profilemessage” document is routed to the OutputTerminal named *Profile out*.

All internal failure conditions are wired to a new OutputTerminal labelled *failure*.

9.6.9 Message flow: “ITSO Profile: add Accounts”

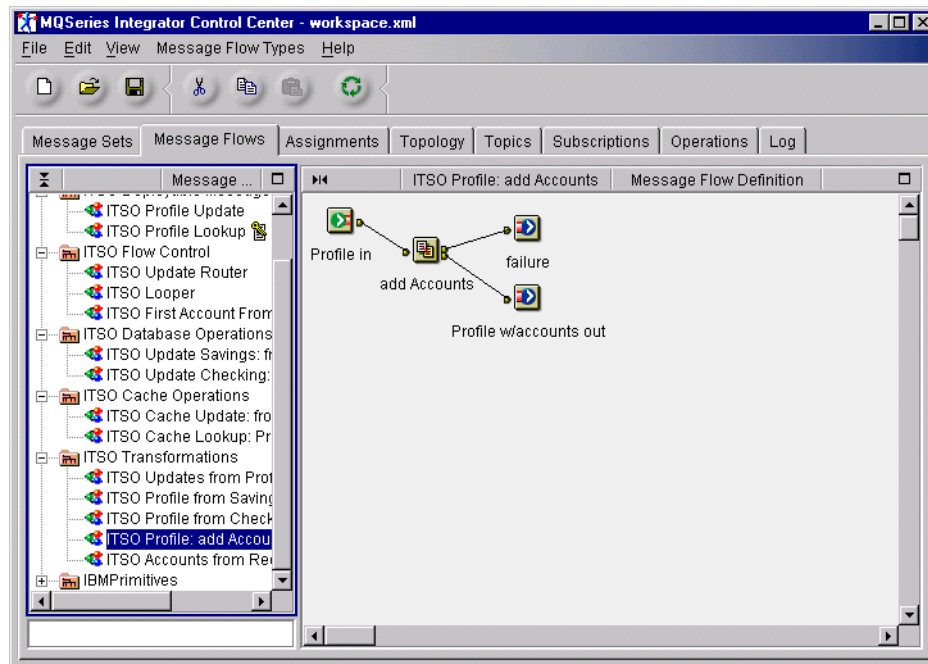


Figure 99. ITSO Profile: add Accounts

Input

At the *Profile in* InputTerminal, the flow will receive a “profilemessage” document.

Output

At the *Profile w/accounts out* OutputTerminal, the flow will output a “profileaccounts” document.

9.6.9.1 Compute node: “add Accounts”

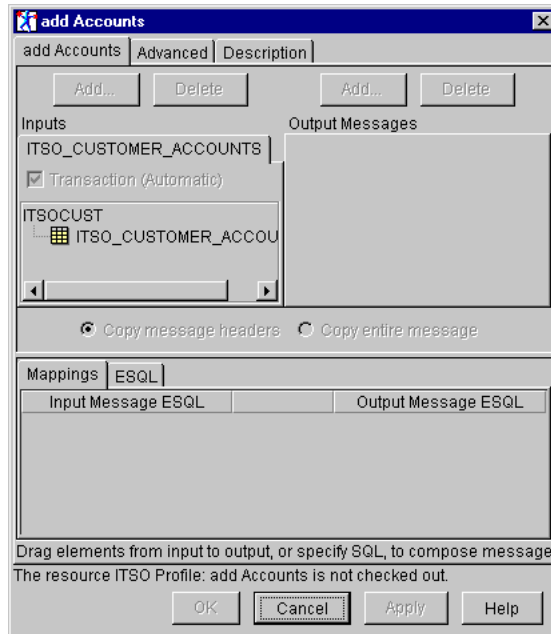


Figure 100. add Accounts

This node performs a database lookup of the ITSO_CUSTOMER_ACCOUNTS table of the ITSOCUST database, based on the “userid” element of the input “profilemessage”. The ESQL code that performs the full operation is as follows:

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET "OutputRoot"."XML".(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET OutputRoot."XML".(XML.tag)"profileaccounts"."profile" =
InputBody.(XML.tag)"profilemessage";
SET OutputRoot."XML".(XML.tag)"profileaccounts"."account"[] =
(
    SELECT T.*
    FROM Database.ITSO_CUSTOMER_ACCOUNTS AS T
    WHERE T.CUSTOMER = "InputBody".(XML.tag)"profilemessage"."userid"
);
```

Here you can see the output “profileaccounts” document comprises the input “profilemessage” document as its “profile” element, and a repeating element named “account” is used to hold the result of the database query.

9.6.10 Message flow: “ITSO Looper”

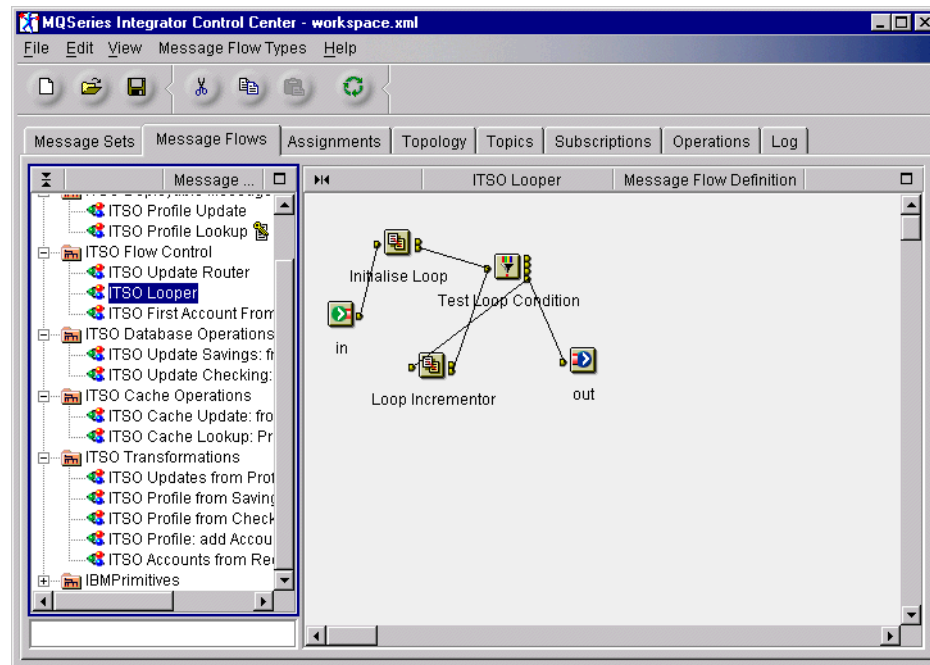


Figure 101. ITSO Looper

Input

ITSO Looper is a re-usable message flow that is designed for general re-use.

It is comprised of:

- A compute node called *Initialise Loop* that can be configured at re-use time to set the loop initialization code.
- A filter node called *Test Loop Condition* which may be configured at re-use time. The condition described here governs whether control will reiterate or cease.
- A compute node that performs the operation that takes place on loop iteration. The node is perhaps wrongly named Loop Incrementor, since this makes the coarse assumption that all loops are controlled using

counters that are incremented. The code that controls the actions in a loop iteration can be configured at message flow re-use time.

The method used to make the ITSO Looper message flow node configurable at re-use time is described in 7.4.6.1, “Using property promotion in message flow nodes” on page 137.

9.6.11 Message flow: “ITSO Updates from Profile w/accounts”

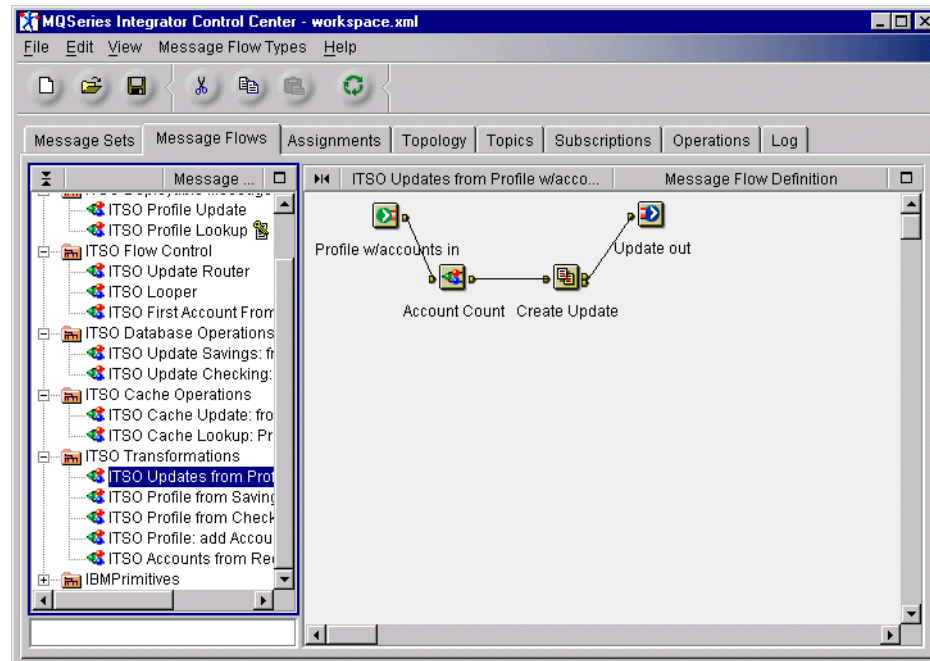


Figure 102. ITSO Updates from Profile w/accounts

Input

At the *Profile w/accounts in* InputTerminal, this message flow receives a “profileaccounts” document.

Output

This message flow is designed to send to the *Update out* OutputTerminal node one or many “updatemessage” documents.

9.6.11.1 Message Flow node: “Account Count”

This is an instance of the ITSO Looper Message Flow node. It has been configured to loop so that a copy of the input “profileaccounts” message is

repeatedly sent as output with an added index value incremented in each instance to make it unique.

The configuration options available are shown in detail here:

Account Count	Description
Loop Condition	CAST(Body.(XML.tag)"Loop"."Index" AS INT) <= CAST(Body.(XML.tag)"Loop"."Count" AS INT)
Loop Increment	SET OutputRoot = InputRoot; SET OutputRoot.XML.(XML.tag)"Loop"."Index" = CAST(InputBody.(XML.tag)"Loop"."Index" AS INT) + 1;
Loop Initialisation	SET OutputRoot=InputRoot; SET OutputRoot.XML.(XML.tag)"Loop"."Count" = CARDINALITY("InputBody".(XML.tag)"profileaccounts"."account"[]); SET OutputRoot.XML.(XML.tag)"Loop"."Index" = 1;

MessageProcessingNodeType: ITSO Looper

OK Cancel Apply Help

Figure 103. Account Count (ITSO Looper) configuration

9.6.11.2 Compute node: "Create Update"

This node makes the simple XML document transformation from "profileaccounts" to "updatemessage", by selecting the required instance of the repeating "account" element from input, to become the fixed and singular "account" element in the output message.

The ESQL code that controls this transformation is detailed below

```
DECLARE I INTEGER;
SET I = 1;
WHILE I < CARDINALITY(InputRoot.*[]) DO
    SET OutputRoot.*[I] = InputRoot.*[I];
    SET I=I+1;
END WHILE;
-- Enter SQL below this line. SQL above this line might be regenerated,
-- causing any modifications to be lost.
SET "OutputRoot".XML.(XML.XmlDecl) = "InputBody".(XML.XmlDecl);
SET "OutputRoot".XML.(XML.tag)"updatemessage"."profile" =
InputBody.(XML.tag)"profileaccounts"."profile";
SET "OutputRoot".XML.(XML.tag)"updatemessage"."account" =
InputBody.(XML.tag)"profileaccounts"."account"[CAST(InputBody.(XML.tag)"Lo
op"."Index" AS INT)];
```

9.6.12 Message flow: “ITSO Update Router”

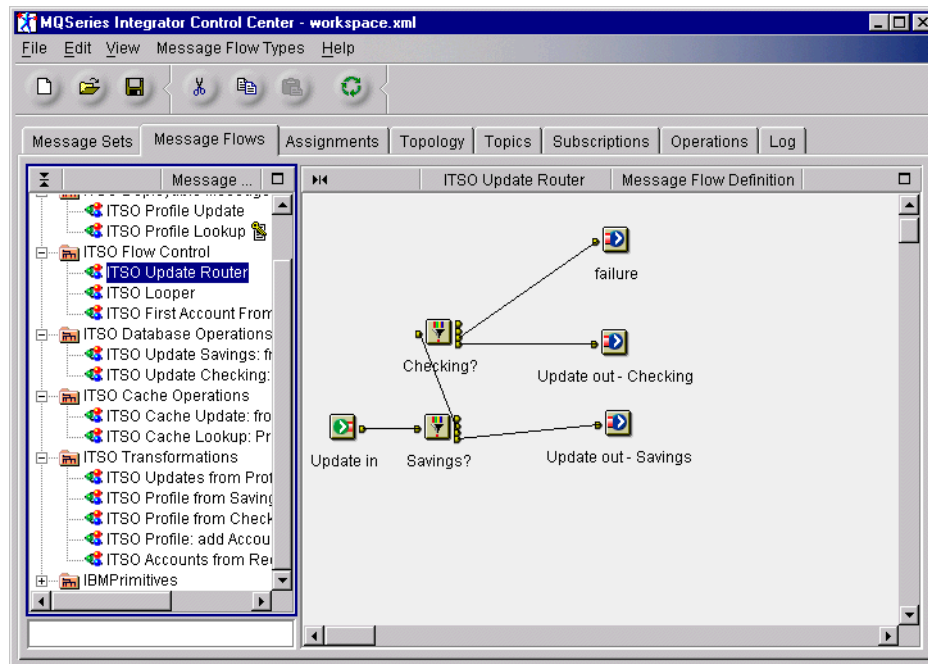


Figure 104. ITSO Updates Router

Input

Input is taken via the InputTerminal *Update in*, in the form of an “updatemessage” document.

Output

This node simply routes the message to either *Update out - Checking* or *Update out - Savings* based on the operation of the filter nodes described below.

9.6.12.1 Filter node: “Savings?”

This node evaluates the following expression:

```
Body. (XML.tag) "updatemessage" . "account" . "ACCTYPE" = 'S'
```

When this expression evaluates to *true*, the update is routed to the *Update out Savings* OutputTerminal.

When this expression evaluates to *false*, the update is routed to the next filter node.

9.6.12.2 Filter node: “Checking?”

This filter node evaluates the following expression:

```
Body.(XML.tag) "updatemessage". "account". "ACCTYPE" = 'C'
```

When this expression evaluates to *true*, the update is routed to the *Update out Checking* OutputTerminal.

9.6.13 Message flow: “ITSO Update Savings: from Update”

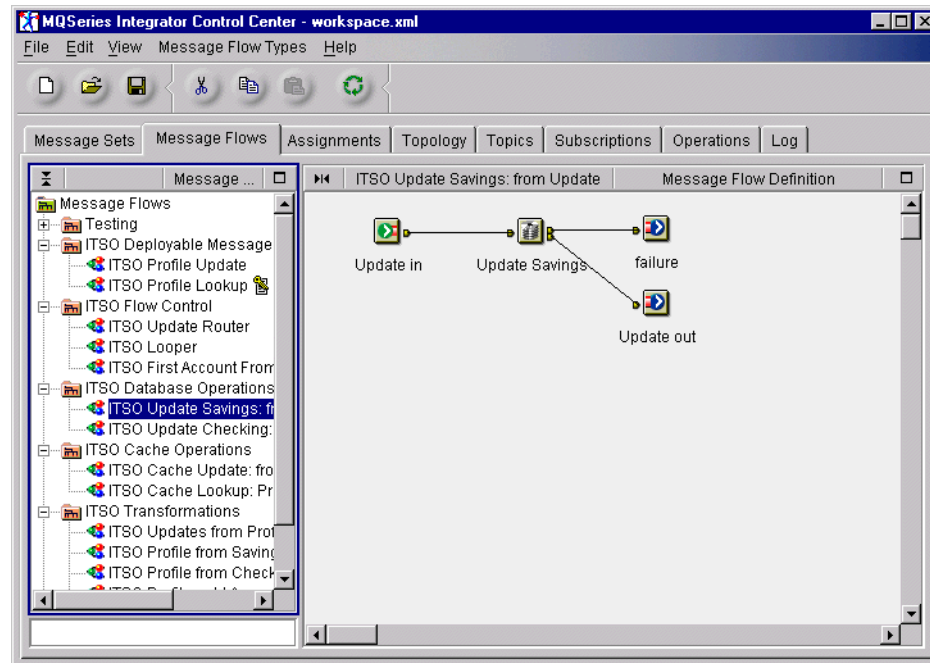


Figure 105. ITSO Update Savings from Update

Input

Input is taken in through *Update in*, in the form of an “updatemessage” document.

Output

This node routes the same message to the OutputTerminal node *Update out*.

9.6.13.1 Database node: “Update Savings”

This database node takes the incoming “updatemessage” and executes ESQL to update the ITSO_SAVINGS table of the ITSOSAVI database based on the content of the input document.

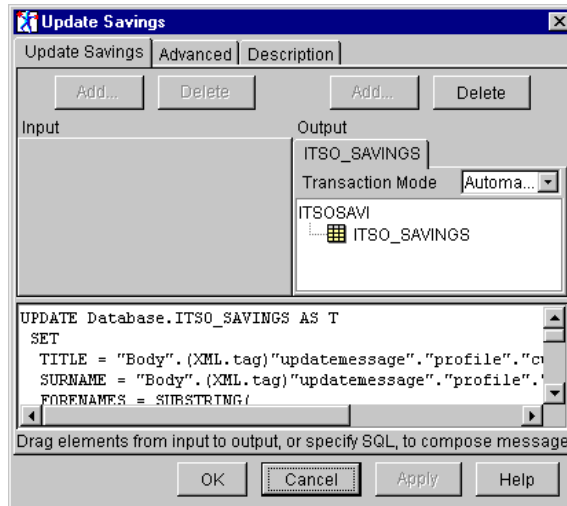


Figure 106. Update Savings from Update

The full ESQL behind this process is listed here:

```
UPDATE Database.ITSO_SAVINGS AS T
SET
  TITLE = "Body".(XML.tag)"updatemessage"."profile"."custname"."namepref",
  SURNAME = "Body".(XML.tag)"updatemessage"."profile"."custname"."surname",
  FORENAMES = SUBSTRING(
    "Body".(XML.tag)"updatemessage"."profile"."custname"."forename"
    || ' ' ||
    "Body".(XML.tag)"updatemessage"."profile"."custname"."middlename"
    FROM 1 FOR 20),
  HOUSENO =
    "Body".(XML.tag)"updatemessage"."profile"."custaddr"."housenum",
  HOUSENAME =
    "Body".(XML.tag)"updatemessage"."profile"."custaddr"."housename",
  STREET = "Body".(XML.tag)"updatemessage"."profile"."custaddr"."street",
  DISTRICT =
    "Body".(XML.tag)"updatemessage"."profile"."custaddr"."district",
  CITY = "Body".(XML.tag)"updatemessage"."profile"."custaddr"."city",
  STATE = "Body".(XML.tag)"updatemessage"."profile"."custaddr"."state",
  ZIP = "Body".(XML.tag)"updatemessage"."profile"."custaddr"."zip",
  PHONENUM =
    "Body".(XML.tag)"updatemessage"."profile"."custaddr"."telephone"
WHERE T."NUMBER" = "Body".(XML.tag)"updatemessage"."account"."ACCOUNT";
```

9.6.14 Message flow: “ITSO Update Checking: from Update”

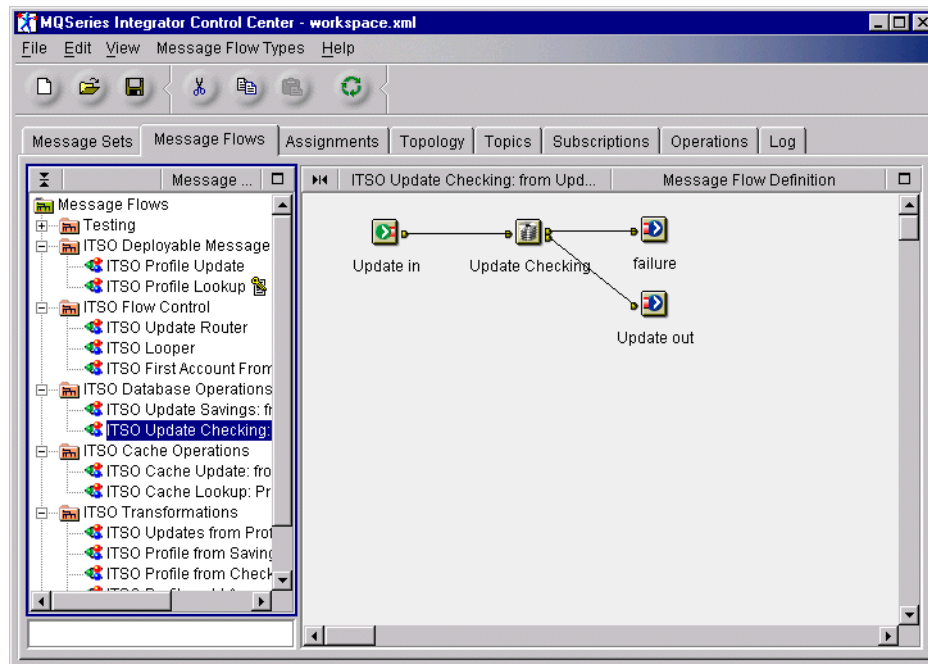


Figure 107. ITSO Update Checking from Update

Input

Input is taken via the InputTerminal *Update in*, in the form of an “updatemessage” document.

Output

This node routes the same message to the OutputTerminal node *Update out*.

9.6.14.1 Database node: “Update Checking”

This database node takes the incoming “updatemessage” and executes ESQL to update the ITSO_CHECKING table of the ITSOCHEC database based on the content of the input document.

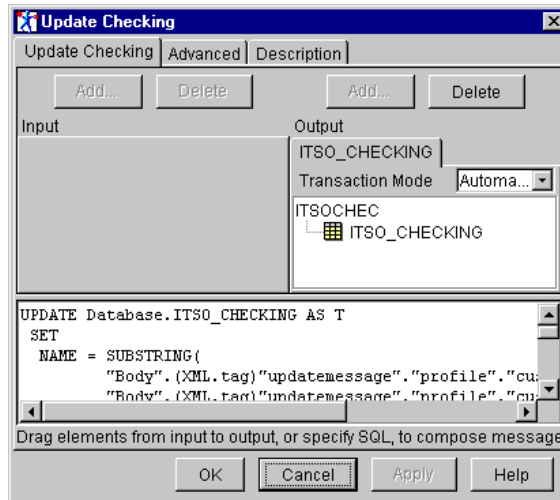


Figure 108. Update Checking from Update

The full ESQL behind this operation is shown here:

```
UPDATE Database.ITSO_CHECKING AS T
SET
  NAME = SUBSTRING(
    "Body".(XML.tag) "updatemessage"."profile"."custname"."forename" ||
    ' ' ||
    "Body".(XML.tag) "updatemessage"."profile"."custname"."middlename"
    || ' ' ||
    "Body".(XML.tag) "updatemessage"."profile"."custname"."surname"
    FROM 1 FOR 40),
  ADDRESS1 = SUBSTRING(
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."house"
    ||
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."house"
    ||
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."street"
    FROM 1 FOR 40),
  ADDRESS2 =
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."district",
  ADDRESS3 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."city",
  ADDRESS4 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."state",
  ADDRESS5 = "Body".(XML.tag) "updatemessage"."profile"."custaddr"."zip",
  TELEPHONE =
    "Body".(XML.tag) "updatemessage"."profile"."custaddr"."telephone"
WHERE T."NUMBER" = "Body".(XML.tag) "updatemessage"."account"."ACCOUNT";
```

9.7 Piecing together the lookup components

We have created the basic message flow components needed to perform the customer profile lookup functions. Each of these message flows has been saved as a message flow node, allowing us to re-use these flows and combine them into larger flows.

9.7.1 Customer profile lookup

In this section, we will describe how the component message flows just described are put together to form the deployable message flow named ITSO Profile Lookup. This message flow will perform the tasks of the customer profile lookup portion of the application, defined in 9.1, “The contract with WebSphere” on page 183.

This message flow will receive an XML message of document type “customerrequest”. From this document, it will examine the local cache of customer details to see if an entry exists. If one exists, this entry is used to construct a “profilemessage” XML document that is returned to the specified reply queue. If there is no matching entry in the local cache, the message flow will determine the accounts the customer holds on remote systems, pick one of these accounts, and look up the customer details on the applicable system. From the account details retrieved from the remote system, a “profilemessage” is constructed that is used to update the local cache, before being sent to the specified reply queue.

The following figure shows an outline of the complete message flow.

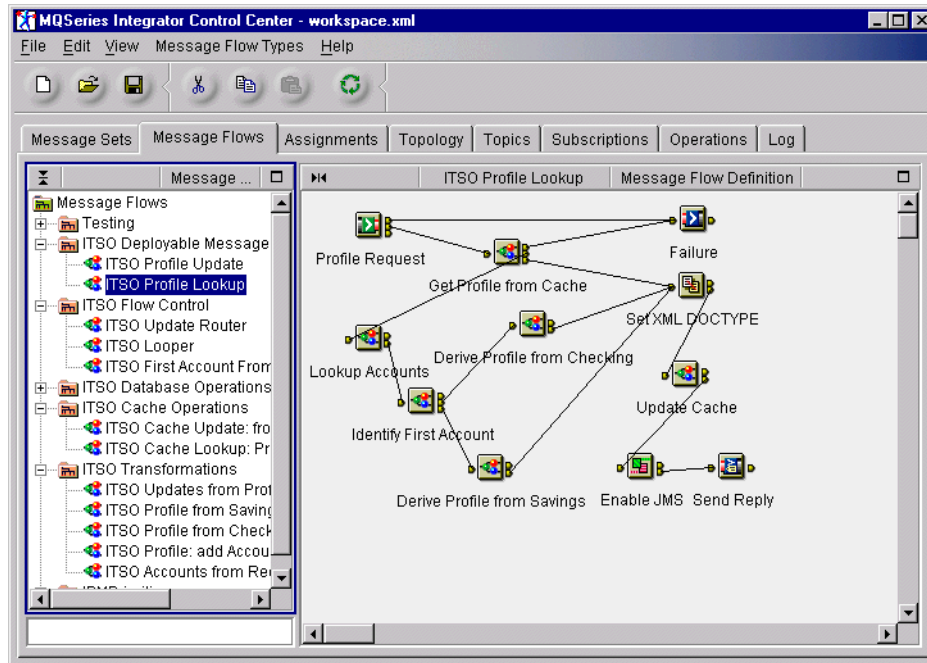


Figure 109. ITSO Profile Lookup

Input

Input is taken in the form of a “customerrequest” document. The Profile Request node is an MQInput node. It defines the MQSeries queue it will use to receive the input message.

Output

This message flow creates an XML “profilemessage” document.

9.7.1.1 Message flow node: “Get Profile from Cache”

This is a message flow node of the type documented in 9.6.3, “Message flow: “ITSO Cache Lookup: Profile from Request”” on page 195.

The “userid” element from the input “customerrequest” XML document is used to look up the customer details in the cache. If these are found, a “profilemessage” XML document is built from the findings and passed to the *SET XML DOCTYPE* compute node detailed below. If no details are found, the original “customerrequest” message is sent to the *Lookup Accounts* node.

9.7.1.2 Message Flow node: “Lookup Accounts”

This is a message flow node of the type documented in 9.6.5, “Message flow: “ITSO Accounts from Request”” on page 204.

The “userid” data element from the input “customerrequest” message is used to look up details of accounts held by this customer. These details are transformed into a “customeraccounts” XML document and passed to the *Identify First Account message flow node*.

9.7.1.3 Message Flow node: “Identify First Account”

This is a message flow node of the type documented in 9.6.6, “Message flow: “ITSO First Account From Accounts”” on page 206.

The first account in the list stored in the “customeraccounts” XML document is selected. Based on its account type, this node creates either a “savings” document, which is sent to the *Derive Profile from Savings* message flow node, or a “checking” document, which is sent to the *Derive Profile from Checking* message flow node.

9.7.1.4 Message flow node: “Derive Profile from Checking”

This is a message flow node of the type documented in 9.6.8, “Message flow: “ITSO Profile from Checking”” on page 214.

This takes the “checking” document and transforms this into a generic “profilemessage”, which is passed on to the *Set XML DOCTYPE* compute node detailed below.

9.7.1.5 Message flow node: “Derive Profile from Savings”

This is a message flow node of the type documented in 9.6.7, “Message flow: “ITSO Profile from Savings”” on page 212.

This takes the “savings” document and transforms this into a generic “profilemessage”, which is passed on to the *Set XML DOCTYPE* compute node detailed below.

9.7.1.6 Compute node: “Set XML DOCTYPE”

This is a straightforward compute node that processes the following ESQL:

```
SET OutputRoot = InputRoot;
-- Enter SQL below this line.  SQL above this line might be regenerated,
causing any modifications to be lost.
SET OutputRoot.XML.(XML.DocTypeDecl)profilemessage='';
SET OutputRoot.XML.(XML.DocTypeDecl).(XML.SystemId)='profilemessage.dtd';
```

The code is designed to add the DOCTYPE descriptor to the XML document, for the benefit of the application.

9.7.1.7 Message flow node: “Update Cache”

This is a message flow node of the type documented in 9.6.4, “Message flow: “ITSO Cache Update: from Profile”” on page 200.

The “profilemessage” document is used to perform a database update to the locally held cache of customer details.

9.7.1.8 Reset Content Descriptor node: “Enable JMS”

The Reset Content Descriptor node adds message definition attributes in the MQRFH2 message header. This usage of the node creates the header necessary to be compatible with the JMS receive operation.

The settings are detailed in the figure below.

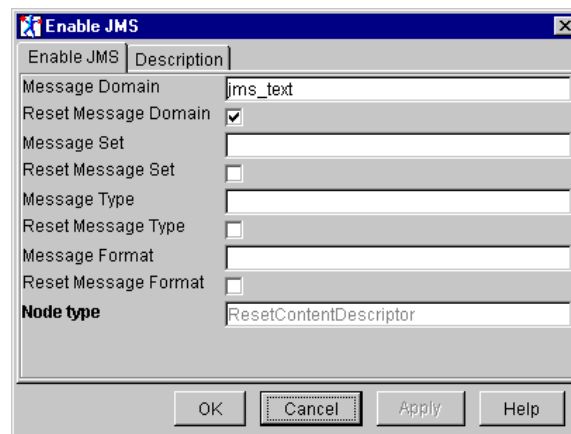


Figure 110. Enable JMS

9.7.1.9 MQReply node: “Send Reply”

The MQReply message takes the message received at its *in* terminal (in our case a “profilemessage” document) and puts it on the reply-to queue defined in the original message header.

9.7.2 Customer profile update

In this section, we describe how the component message flows described in 9.7.2, “Customer profile update” on page 231 are put together to form the deployable message flow named ITSO Profile Update. This message flow will

perform the tasks defined for the customer profile update portion of the application, defined in 9.1, “The contract with WebSphere” on page 183.

This message flow will receive an XML message of document type “profilemessage”. It saves these details in the locally held cache. It also performs a database lookup to determine the number of accounts held for that customer. When this list is obtained, it works through the list, generating individual update documents for each account. These are routed to the correct node for their processing. The whole operation takes place as a single unit of work.

The full message flow is shown in the figure below.

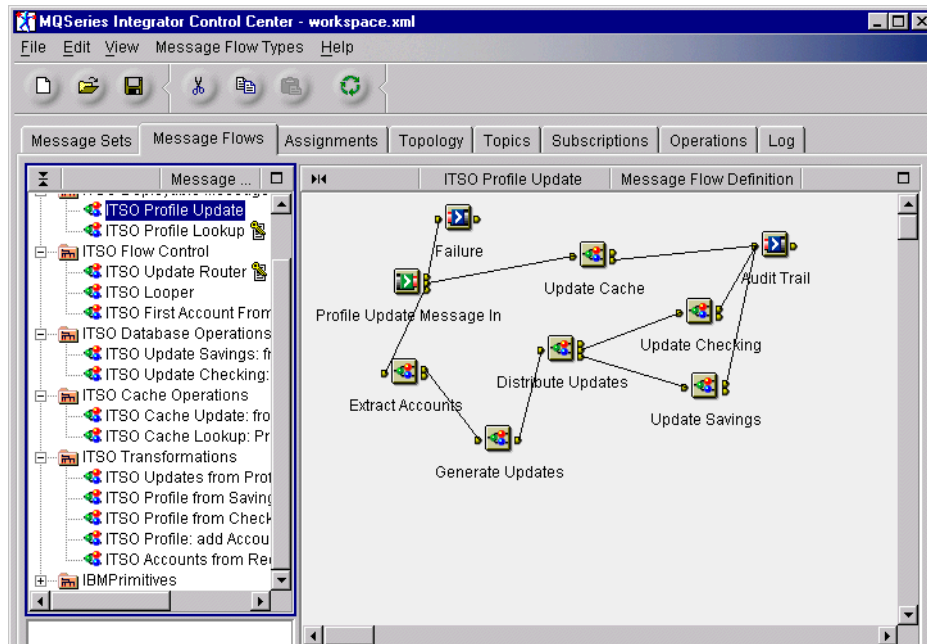


Figure 111. ITSO Profile Update

Input

Input is taken in the form of a “profilemessage” document.

Output

This message flow creates a number of messages that are finally used for database updates to ITSO_CUSTOMER, ITSO_SAVINGS and ITSO_CHECKING. These are nominally routed to a named output queue used as input to an audit trail. This feature has not been implemented in the sample.

9.7.2.1 Message flow node: “Update Cache”

This is an instance of the message flow node described in 9.6.4, “Message flow: “ITSO Cache Update: from Profile”” on page 200.

It takes a “profilemessage” from the main input node, updates the local cache (ITSO_CUSTOMER) and puts the message to the audit trail.

9.7.2.2 Message flow node: “Extract Accounts”

This is an instance of the message flow node described in 9.6.9, “Message flow: “ITSO Profile: add Accounts”” on page 218.

It takes a “profilemessage” from the main input node, performs a database lookup to retrieve the list of accounts held by the customer and puts a “profileaccounts” document as output to the *Generate Updates* node described below.

9.7.2.3 Message flow node: “Generate Updates”

This is an instance of the message flow node described in 9.6.11, “Message flow: “ITSO Updates from Profile w/accounts”” on page 221.

It takes a “profileaccounts” document and uses this to create a sequence of “updatemessage” documents as output, which are each passed to the *Distribute Updates* message flow node described below.

9.7.2.4 Message flow node: “Distribute Updates”

This is an instance of the message flow node described in 9.6.12, “Message flow: “ITSO Update Router”” on page 223.

By this stage in the process more than one incoming message may form part of the same unit of work. Each incoming message is routed either to the *Update Checking* or *Update Savings* nodes, based on the account type held in the “updatemessage”. Both nodes are detailed below.

9.7.2.5 Message flow node: “Update Savings”

This is an instance of the message flow node described in 9.6.13, “Message flow: “ITSO Update Savings: from Update”” on page 224. The “updatemessage” details are used to update the customer profile details as held on the savings account database.

9.7.2.6 Message flow node: “Update Checking”

This is an instance of the message flow node described in 9.6.14, “Message flow: “ITSO Update Checking: from Update”” on page 226. The

“updatemessage” details are used to update the customer profile details as held on the checking account database.

9.7.2.7 MQOutput node: “Audit Trail”

A nominal queue has been defined to hold all “profilemessage” and “updatemessage” documents that have been processed by this message flow.

9.8 Tracing

As you develop your message flows, it will often be useful to view the content of intermediate data in the message flow. To do this, you can use the trace node to extract the data you wish to monitor and write the result to a file.

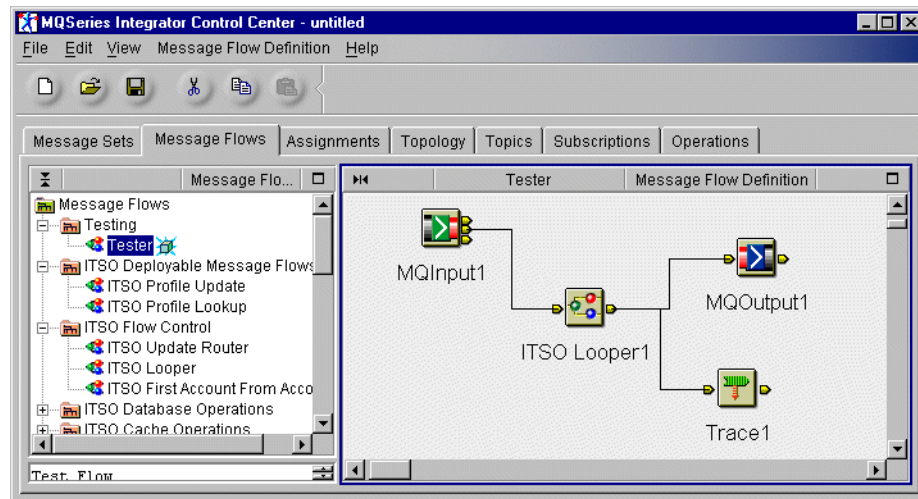


Figure 112. Trace node as part of a message flow

Add the trace node to your message flow and define the type of output you want to see.

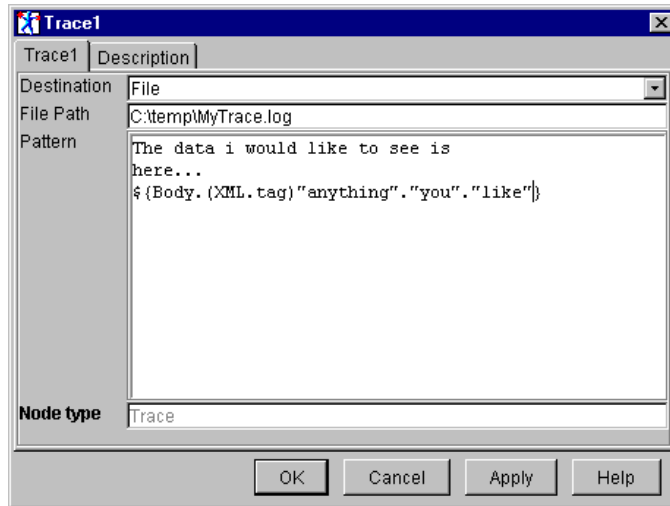


Figure 113. Configuring a trace node

The trace node described above will always produce trace output.

Another type of tracing provides a more general view of broker domain operations in a trace log. Here, you may switch tracing on for an entire broker, an execution group, or an individual message flow in the Operations view of the Control Center. Right-click the item you want to have traced (as shown in Figure 114), and select **User Trace -> Normal**. Debug trace is a greater level of detail again.

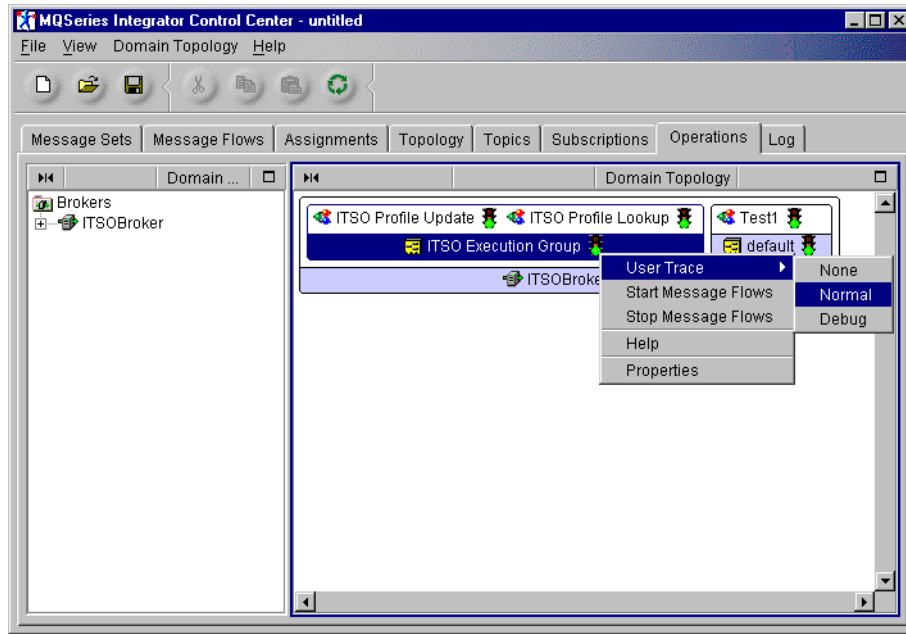


Figure 114. Switching on trace from operations

The trace produced is similar to that detailed in Figure 115. However, in order to produce this listing, you have to use the rather involved commands *mqsireadlog* and *mqsiformatlog*. These commands provide all the flexibility in targeting and formatting the trace you may require.

On Windows NT, to provide an easy route to getting hold of the trace listing but sacrificing full flexibility, we used the a custom-built batch file called *readtrace*, which is executed at the command line using two parameters: *brokername* and *executiongroupname*.

```
C:\>type readtrace.bat

del trace.xml
del trace.txt
mqsireadlog %1 -u -e %2 -o trace.xml
mqsiformatlog -i trace.xml -o trace.txt
start notepad trace.txt
mqsichangetrace %1 -u -e %2 -r

C:\>
```

The `msgichangetrace` command at the tail end of this batch file clears the system's trace file, so that the next execution of `readtrace` does not repeat the same detail.

Figure 115 shows the type of operations trace produced by `readtrace`. As you can see, the output is extensive even though this was a trace at normal level. The debug version is very detailed indeed.

```

Timestamps are formatted in local time, 240 minutes before GMT.

2000-09-21 06:57:04.037000 349 UserTrace BIP4040I: The Execution Group 'ITS0 Execution Group' has processed a configuration message suc
A configuration message has been processed successfully. Any configuration changes have been made and :
No user action required.

2000-09-21 06:57:04.037000 349 UserTrace BIP2638E: MQPUT to queue 'SYSTEM.BROKER.EXECUTIONGROUP.REPLY' on queue manager 'ITS0.QM.BR': M
The node 'ConfigurationMessageFlow.outputNode' attempted to write a message to the specified queue 'SV
No user action required.

2000-09-21 06:57:04.037000 349 UserTrace BIP2622I: Message successfully output by output node 'ConfigurationMessageFlow.outputNode' to
The MQSeries output node 'ConfigurationMessageFlow.outputNode' successfully wrote an output message to
No user action required.

2000-09-21 07:04:37.839000 349 UserTrace BIP6060I: Parser type 'Properties' created on behalf of node 'ConfigurationMessageFlow.InputNo
The message broker has created a parser of type 'Properties' on behalf of node 'ConfigurationMessageFl
No user action required.

2000-09-21 07:04:37.839000 349 UserTrace BIP6061I: Parser type 'MQMD' created on behalf of node 'ConfigurationMessageFlow.InputNode' to
The message broker has created a parser of type 'MQMD' on behalf of node 'ConfigurationMessageFlow.Inp
No user action required.

2000-09-21 07:04:37.839000 349 UserTrace BIP2632I: Message being propagated to the output terminal; node 'ConfigurationMessageFlow.Inpu
An input message received from MQSeries input queue in node 'ConfigurationMessageFlow.InputNode' is be
No user action required.

2000-09-21 07:04:37.839000 349 UserTrace BIP6061I: Parser type 'XML' created on behalf of node 'ConfigurationMessageFlow.InputNode' to
The message broker has created a parser of type 'XML' on behalf of node 'ConfigurationMessageFlow.Inpu
No user action required.

2000-09-21 07:04:42.535999 349 UserTrace BIP2265I: Attribute in message flow 'ITS0 Profile Update' (uuid='2961feb5-e100-0000-0080-a2a23
The message broker received a configuration message containing an instruction to change an attribute i
No user action required.

2000-09-21 07:04:43.536998 349 UserTrace BIP2254I: Message flow node 'Audit Trail' (uuid='593baebc-e100-0000-0080-a2a238596eab') type 'I
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.536998 349 UserTrace BIP2254I: Message flow node 'Update Checking.Update Checking' (uuid='1bd601b6-e100-0000-0080-a
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.547000 349 UserTrace BIP2254I: Message flow node 'Update Savings.Update Savings' (uuid='b45b01b6-e100-0000-0080-a2a
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.547000 349 UserTrace BIP2254I: Message flow node 'Distribute Updates.Checking?' (uuid='249300b6-e100-0000-0080-a2a2
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.547000 349 UserTrace BIP2254I: Message flow node 'Distribute Updates.Savings?' (uuid='249300b6-e100-0000-0080-a2a23
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.547000 349 UserTrace BIP2254I: Message flow node 'Generate Updates.Create Update' (uuid='5cebffb5-e100-0000-0080-a2
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.556999 349 UserTrace BIP2254I: Message flow node 'Generate Updates.Account Count.Initialise Loop' (uuid='5cebffb5-e
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.556999 349 UserTrace BIP2254I: Message flow node 'Generate Updates.Account Count.Loop Incrementor' (uuid='5cebffb5-
The message broker received a configuration message containing an instruction to create a message flow
No user action required.

2000-09-21 07:04:43.556999 349 UserTrace BIP2254I: Message flow node 'Generate Updates.Account Count.Test Loop Condition' (uuid='5cebffb

```

Figure 115. Example operations trace listing

Chapter 10. System management guidelines

Systems management is an important phase of application design and continues to be a factor in the day-to-day operations of the business. Systems management covers many areas, typically involving:

- Application management
- Performance monitoring
- Availability management
- Security management
- Disaster recovery
- Operating system and network administration
- Asset management
- Software distribution
- Problem reporting
- Change management

Many of these are general considerations that span entire enterprise operations. We will not attempt to cover all of these topics here, but will focus on system management techniques specific to MQSeries and MQSeries Integrator.

System management guidelines for WebSphere Advanced Edition V3.5 can be found in the *User-to-Business Patterns Systems Management Guidelines* redpaper, available at the IBM Redbooks home page at

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

10.1 MQSeries system management

The degree of complexity of MQSeries system management is directly proportional to the size of the MQSeries network you are trying to manage. In its simplest form, an MQSeries network is comprised of a single MQSeries queue manager and its resources, but can grow to hundreds or even thousands of queue managers spread throughout the enterprise.

To add to the complexity, given MQSeries' middleware functionality, MQSeries networks could span many platforms with different networks, hardware, operating systems, and applications. Each platform possesses its unique specific systems management needs and requirements, not to mention a varying level of support for the MQSeries administration tools and interfaces.

It is of no surprise, then, to see the growth of MQSeries system management tools available in the marketplace today that strive in some way, shape, or form to deliver one or many combinations of the following MQSeries system management tasks:

- **Configuration management** - The ability to deploy MQSeries code, and create and delete MQSeries objects including queue managers, queues, channels and processes, from a single point of control.
- **Operational management** - The ability to start and stop resources such as queue managers, channels, trigger monitors, channel listeners, and initiators from a single point.
- **Problem management** - The ability to detect, track, and resolve problems with MQSeries objects from a single point of control.
- **Performance management** - The ability to determine performance of MQSeries objects from a single point of control.

in this book we will not be focusing on the growing system management tools available on the market today. Instead, we will focus on introducing the reader to the basic MQSeries system management guidelines along with an introduction to the MQSeries facilities and tools that are available out of the box. It suffices to say that if your MQSeries system management needs surpass those that are delivered with the product, the chosen system management tool should be an extension, or at the least, a good fit with your enterprise-wide system management framework.

The following references discuss MQSeries system management:

- **Managing MQSeries** - MQSeries expert Les Yeamans of NASG investigates the MQSeries systems management market, and provides an independent assessment of the major products available. At <http://www.software.ibm.com/ts/mqseries/library/independent/nasg/vendor.html>
- **MQSeries SupportPac MS08 - *Evaluation of MQSeries System Management Products*** at <http://www.software.ibm.com/ts/mqseries/txppacs>
- **MQSeries SupportPac MS0D - *Selecting MQSeries System Management tools***, available at <http://www.software.ibm.com/ts/mqseries/txppacs>
- **MessageQ.Com** at <http://www.messageq.com>

10.1.1 MQSeries administration interfaces

MQSeries administration tasks include creating, starting, altering, viewing, stopping, and deleting MQSeries objects including:

- MQSeries queue managers
- MQSeries queues
- Process definitions
- Channels
- Clusters
- Namelists

Each MQSeries network has one or more instance of a queue manager known by a name within the network of interconnected queue managers. For all other object types, each object has a name associated with it and can be referenced by that name. These names must be unique within one queue manager and object type. For example, you can have a queue and a process with the same name, but you cannot have two queues with the same name.

These objects are managed by MQSeries administration tasks that can be performed by using any of the following MQSeries interfaces:

- Control commands
- MQSeries commands (MQSC)
- Programmable Command Format (PCF)
- MQSeries Administration Interface (MQAI)
- MQSeries Explorer (available on Windows NT only)
- MQSeries Services 'snap-in' (available on Windows NT only)
- WEB Administration (available on Windows NT only)

10.1.1.1 Control commands

You use control commands to perform operations on queue managers, command servers, and channels. Control commands can be divided into three categories as shown in Table 10.

Table 10. Control command categories

Category	Description
Queue manager commands	Queue manager control commands include commands for creating, starting, stopping, and deleting queue managers and command servers
Channel commands	Channel commands include commands for starting and ending channels and channel initiators

Category	Description
Utility commands	<p>Utility commands include commands associated with:</p> <ul style="list-style-type: none"> - Running MQSC commands - Conversion exits - Authority management - Recording and recovering media images of queue manager resources - Displaying and resolving transactions - Trigger monitors - Displaying the file names of MQSeries objects

MQSeries System Administration, SC33-1873, contains a description of each control command and its syntax.

10.1.1.2 MQSC commands

MQSC commands are used to perform operations on queue manager objects. They are issued using the `runmqsc` command. This can be done interactively from a keyboard, or by redirecting the standard input device (stdin) to run a sequence of commands from an ASCII text file. In both cases, the format of the commands is the same.

MQSeries Command Reference, SC33-1369, contains a description of each MQSC command and its syntax. An example of using MQSC commands can be seen in 10.2.1.1, “Administering multiple brokers” on page 268.

10.1.1.3 PCF and MQSeries Administration Interface (MQAI)

The purpose of MQSeries programmable command format (PCF) commands is to allow administration tasks to be programmed into an administration program. In this way you can create queues and process definitions, and change queue managers, from a program.

PCF commands cover the same range of functions provided by the MQSC facility. Each PCF command is a data structure that is embedded in the application data part of an MQSeries message. Each command is sent to the target queue manager using the MQI function, MQPUT, in the same way as any other message. The command server on the queue manager receiving the message interprets it as a command message and runs the command. To get a reply, the application issues an MQGET call and the reply data is returned in another data structure. The application can then process the reply and act accordingly.

Note

Unlike MQSC commands, PCF commands and their replies are not in a text format that you can read.

You can use MQAI to obtain easier programming access to PCF messages, that are considered cumbersome to work with.

MQSeries Programmable System Management, SC33-1482, and *MQSeries Administration Interface Programming Guide*, SC33- 5390 contain more information about PCF and MQAI respectively.

10.1.1.4 MQSeries Explorer

MQSeries for Windows NT Version 5.1 provides an administration interface called the MQSeries Explorer to perform administration tasks as an alternative to using control or MQSC commands.

The MQSeries Explorer allows you to perform remote administration of your network from a computer running Windows NT simply by pointing the MQSeries Explorer at the queue managers and clusters you are interested in.

The platforms and levels of MQSeries that can be administered using the MQSeries Explorer, and the configuration steps you must perform on remote MQSeries queue managers to allow the MQSeries Explorer to administer them, are outlined in 10.1.2, “Remote administration” on page 246.

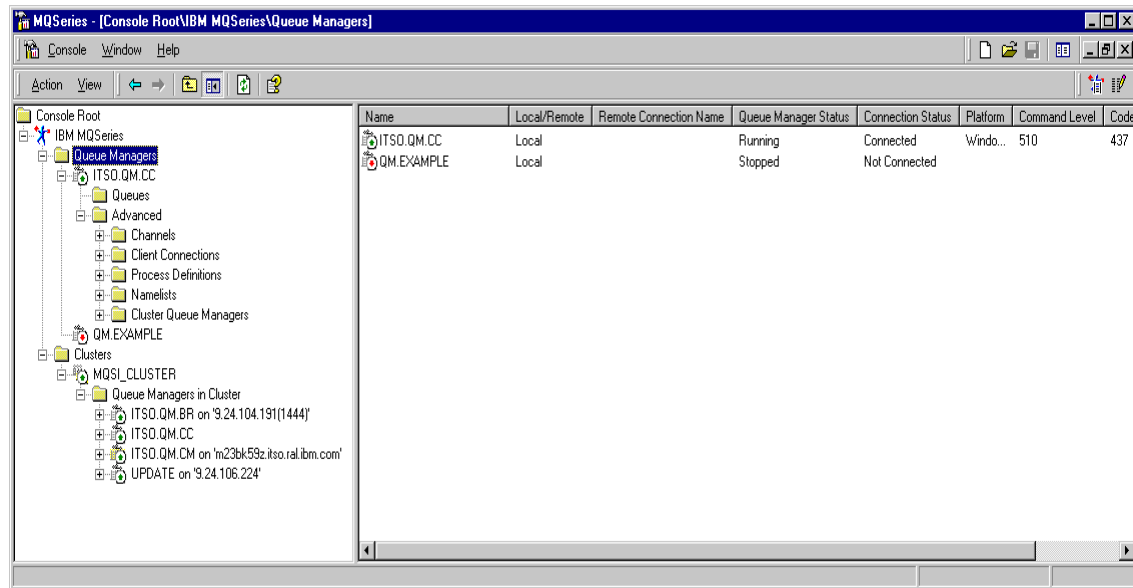


Figure 116. MQSeries Explorer

With the MQSeries Explorer, you can:

- Start and stop a queue manager (on your local machine only).
- Define, display, and alter the definitions of MQSeries objects such as queues and channels.
- Browse the messages on a queue.
- Start and stop a channel.
- View status information about a channel.
- View queue managers in a cluster.
- Create a new queue manager cluster using the Create New Cluster wizard.
- Add a queue manager to a cluster using the Add Queue Manager to Cluster wizard.
- Add an existing queue manager to a cluster using the Join Cluster wizard.

For more details on the MQSeries Explorer, please refer to *MQSeries System Administration*, SC33-1873.

10.1.1.5 MQSeries Services snap-in

The MQSeries Services snap-in can be used to administer local or remote MQSeries for Windows NT servers. It also allows you to monitor alerts created by problems in the local system.

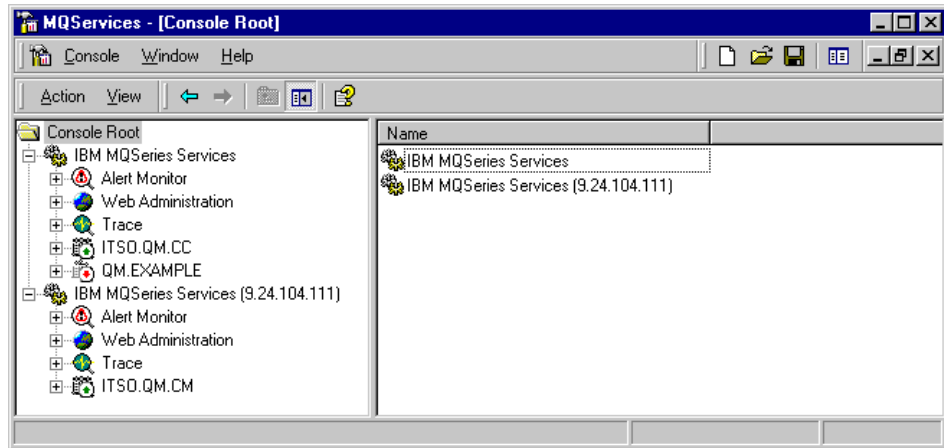


Figure 117. MQSeries Services snap-in

With the MQSeries Services snap-in, you can:

- Start or stop a queue manager (on your local machine or on remote NT machines).
- Start or stop the command servers, channel initiators, trigger monitors, and listeners.
- Create and delete queue managers, command servers, channel initiators, trigger monitors, and listeners.
- Set any of the services to start up automatically or manually during system start up.
- Modify the properties of queue managers. This function replaces the use of stanzas in configuration (mqc.ini and qm.ini) files.
- Change the default queue manager.
- Modify the parameters for any service, such as the TCP port number for a listener, or a channel initiator queue name.
- Modify the behavior of MQSeries if a particular service fails, for example, retry starting the service x number of times.
- Start or stop the service trace.
- Start or stop MQSeries Web Administration.

For more details on the MQSeries Services snap-in, please refer to the *MQSeries System Administration*, SC33-1873.

10.1.1.6 Web Administration

Web Administration enables you to use a Web browser to do everything you can do using `runmqsc`.

In addition, it enables you to construct more sophisticated scripts that can include conditional logic, looping, nesting, and so on. It also includes a simple file management capability that you can use to organize your script files in public and private data stores.

Detailed information about performing these functions can be found in the online help for MQSeries Web Administration.

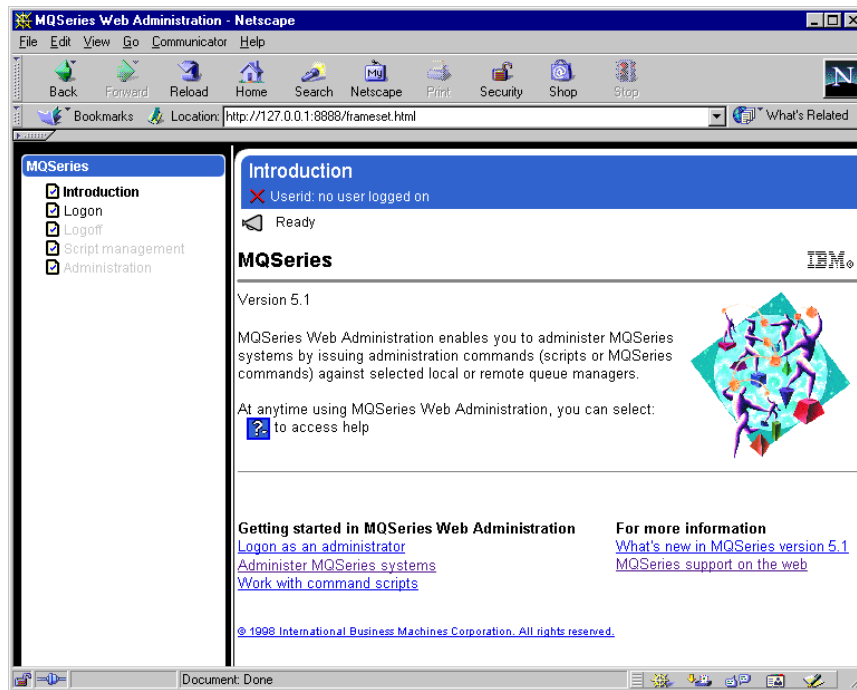


Figure 118. Web Administration on Windows NT

The Web server that hosts these new facilities runs only on Windows NT, but the browser that provides the human interface can run on any platform that supports a Java-enabled Web browser, such as Netscape Navigator or the Microsoft Internet Explorer.

For more details on MQSeries Web Administration, please refer to *MQSeries System Administration*, SC33-1873.

10.1.2 Remote administration

MQSeries Explorer, Web Administration, and MQSeries Services snap-in all offer some form of remote administration.

The MQSeries Explorer can remotely administer MQSeries on the the following platforms:

Table 11. Remote Management

Platform	Minimum Command Level
AIX, UNIX	221
OS/400	320
OS/2	201
VMS and Tandem	221
MQ/390	Indirectly; see 10.1.3.1, "Points to consider when using the MQSeries Explorer" on page 248.

The platform and command level headings of Table 11 refer to the platform and command level queue manager attributes. Both must be used to determine which system control commands are supported. You can see the platform and command level attributes in the MQSeries Explorer window, shown in Figure 116 on page 244.

An important point to be aware of is that both the MQSeries Explorer and Web Administration interfaces can only transmit commands that you would in the past have entered with `runmqsc`, or the platform equivalent. That is, you *can't* use them to create or delete queue managers. Although MQSeries Services snap-in allows you to start and stop remote queue managers and their associated processes, that only works when the remote system is running on Windows NT.

Note

You can do a limited amount of remote operations of queue managers on non-Windows NT platforms, such as starting and stopping channels, but full-blown operations in a heterogeneous multi-platform configuration require more than MQSeries Explorer, MQSeries Services, or Web Administration can deliver.

To remotely administer a queue manager from any of the MQSeries administration interfaces, you will need the following:

- A command server running for any queue manager being administered.
- A suitable TCP/IP listener for every remote queue manager. This may be the MQSeries listener or the INETD daemon as appropriate.

- A server connection channel, called `SYSTEM.ADMIN.SVRCONN`, on every remote queue manager. This channel is mandatory for every remote queue manager being administered.

10.1.3 Administration interface guidelines

Deciding which of the administration interfaces or techniques are the most appropriate for a particular operation depends mainly on the platform type and the MQSeries task at hand.

On Windows NT, you can carry out most common administration and operations tasks using the MQSeries Explorer and MQSeries Services snap-in tools. These tools make Windows NT a very convenient environment for experimentation and development. However, it is much more efficient to use one of the scripting techniques to populate and manipulate queue managers once they have been created.

If your network configuration includes a Windows NT server with MQSeries Version 5.1 installed on it, you can use a combination of the MQSeries Web Administration and MQSeries Explorer to carry out limited remote administration of your non-NT queue managers. Otherwise, you will have to use the `runmqsc` facility interactively or with a script file, as you did with previous versions of MQSeries.

MQSeries script files containing control and MQSC commands are a very common way of administering MQSeries. A whole MQSeries configuration can be stored in such a script file and thus can be used, with minor modifications, to define new queue managers as new MQSeries nodes are added to the network.

Web Administration enables you to use a Web browser to do everything you can do using the `runmqsc` command. In addition, it enables you to construct more sophisticated scripts that can include conditional logic, looping, nesting, and so on. It also includes a simple file management capability that you can use to organize your script files in public and private data stores.

10.1.3.1 Points to consider when using the MQSeries Explorer

When deciding whether to use the MQSeries Explorer at your installation, bear the following points in mind:

- The MQSeries Explorer works best with small queue managers. If you have a large number of objects on a single queue manager you may experience delays while the MQSeries Explorer extracts the required information to present in a view. As a rough guide as to what a “large number” is, if your queue managers have more than 200 queues or 100

channels, you may want to consider using a third-party enterprise console product instead of the MQSeries Explorer.

- MQSeries clusters can potentially contain hundreds or thousands of queue managers. Because the MQSeries Explorer presents the queue managers in a cluster using a tree structure, the view can become cumbersome for large clusters. The physical size of a cluster does not affect the speed of the MQSeries Explorer dramatically because the Explorer does not connect to the queue managers in the cluster until you select them.
- The message browser displays the first 200 messages on a queue. Only the first 1000 bytes of message data contained in a message are formatted and displayed on your screen. Messages containing more than 1000 bytes of message data are not displayed in their entirety.
- The MQSeries Explorer cannot administer a cluster whose repository queue managers are on MQSeries for OS/390. To avoid this problem, nominate an additional repository queue manager on a system that the MQSeries Explorer can administer. By connecting the cluster through this new repository queue manager, you can administer the queue managers in the cluster, subject to the MQSeries Explorer's usual restrictions for supported levels of MQSeries.

10.1.3.2 Points to consider when using Web Administration

When deciding whether or not to use MQSeries Web Administration at your installation, bear the following points in mind:

- The MQSeries Web Administration Web server requires a dedicated IP port number.
- MQSeries Web Administration can be accessed from the Internet if permitted to do so by your network configuration.
- All users of MQSeries Web Administration require an active Windows NT user ID on the server computer with sufficient user rights to run MQSC commands.
- To administer remote queues with Web Administration, MQSeries message channels or a cluster must be configured between the systems.

10.1.4 Overview of the MQSeries clustering feature

The MQSeries clustering feature has been introduced with Version 5.1 of MQSeries on Intel and UNIX platforms and with Version 2.1 on OS/390.

10.1.4.1 Administration benefits of clustering

In a traditional MQSeries network using distributed queuing, every queue manager is independent. If one queue manager needs to send messages to another queue manager, it must have defined a transmission queue, a channel to the remote queue manager, and a remote queue definition for every queue to which it wants to send messages.

A cluster is a group of queue managers set up in such a way that the queue managers can communicate directly with one another over a single network without the need for complex transmission queue, channels, and queue definitions.

Clusters can be set up easily, and typically contain queue managers that are logically related in some way and need to share data or applications.

Once a cluster has been created the queue managers within it can communicate with each other without the need for complicated channel or remote queue definitions. Even the smallest cluster will reduce system administration overheads.

Establishing a network of queue managers in a cluster involves fewer definitions than establishing a traditional distributed queuing environment. With fewer definitions to make, you can set up or change your network more quickly and easily, and the risk in making an error in your definitions is reduced.

10.1.4.2 Clustering details

To set up a cluster, you need to define one cluster sender (CLUSDR) definition and one cluster receiver (CLUSRCVR) definition per queue manager. You do not need to define any transmission or remote queues.

In a cluster environment you should promote one (or ideally two) queue managers as *full repository queue managers*. This means that such a queue manager knows all other queue managers in the cluster. It knows what clustered objects (local queues or any other type of object) are hosted by which queue manager and it knows how to reach those queue managers. This last thing means that the full repository queue manager has a template definition of a sender and receiver channel definition that can be used to automatically create a new sender/receiver channel when needed.

To make two queue managers QM1 and QM2 a full repository queue manager for a cluster named MY_CLUSTER, you need to execute the following MQSeries commands:

On QM1:

```
DEFINE CLUSCHL (TO.QM1) CHLTYPE (CLUSRCVR) +  
    CONNAME (hostname1) +  
    CLUSTER (MY_CLUSTER)  
  
DEFINE CLUSCHL (TO.QM2) CHLTYPE (CLUSSDR) +  
    CONNAME (hostname2) +  
    CLUSTER (MY_CLUSTER)  
  
ALTER QMGR REPOS (MY_CLUSTER)
```

On QM2:

```
DEFINE CLUSCHL (TO.QM2) CHLTYPE (CLUSRCVR) +  
    CONNAME (hostname2) +  
    TRPTYPE (TCP) +  
    CLUSTER (MY_CLUSTER)  
  
DEFINE CLUSCHL (TO.QM1) CHLTYPE (CLUSSDR) +  
    CONNAME (hostname1) +  
    TRPTYPE (TCP) +  
    CLUSTER (MY_CLUSTER)  
  
ALTER QMGR REPOS (MY_CLUSTER)
```

The object of type CLUSRCVR, or cluster receiver, specifies how a queue manager wants other queue managers to talk to him, or more technically, how other queue managers should create a sender channel to send messages to this queue manager.

The object of type CLUSSDR, or cluster sender, should provide the queue manager a sender channel to the other full repository queue manager in the cluster.

The ALTER QMGR command finally makes the queue manager a full repository queue manager.

As soon as these definitions are in place, you will have two-way communication between QM1 and QM2. When you now add a local queue to QM1 and you specify the cluster name MY_CLUSTER, QM1 will pass the definition of that object to the second full repository queue manager QM2.

```
DEFINE QLOCAL (CLUSTERED_QUEUE) CLUSTER (MY_CLUSTER)
```

Given that QM2 has channels to QM1 and that QM2 now knows that CLUSTERED_QUEUE exists on QM1, you do not need to define a remote queue object! Note that we haven't defined transmission queues so far. MQSeries cluster channels are using a common, predefined transmission queue called SYSTEM.CLUSTER.TRANSMIT.QUEUE.

The real benefit of MQSeries clusters becomes clear when adding additional queue managers to the cluster. To add QM3 to the cluster, one needs to define a cluster receiver channel to make clear to the cluster how other queue managers should talk to QM3:

```
DEFINE CLUSCHL(TO.QM3) CHLTYPE(CLUSRCVR) +  
    CONNAME(hostname3) +  
    TRPTYPE(TCP) +  
    CLUSTER(MY_CLUSTER)
```

The next and last thing you need to make QM3 part of the cluster is a cluster sender channel to one full repository queue manager. Because there are two full repository queue managers, you can choose. Which one you select is not important.

```
DEFINE CLUSCHL(TO.QM2) CHLTYPE(CLUSSDR) +  
    CONNAME(hostname2) +  
    TRPTYPE(TCP) +  
    CLUSTER(MY_CLUSTER)
```

When this object is created, QM3 will start the channel TO.QM2 and give QM2 the definition of the channel TO.QM3. QM2 will immediately use that definition to create a sender channel from QM2 to QM3. From this point on QM3 can use any object in the cluster MY_CLUSTER! When an application connects to QM3 and opens the queue CLUSTERED_QUEUE, QM3 will not know where that object lives. At least QM3 knows that it is not hosting this object itself. Thus, it asks its full repository queue manager QM2 about this object. QM2 replies with the definition: CLUSTERED_QUEUE is a local queue hosted by QM1. Because QM3 does not know how to talk to QM1, QM2 sends another request to QM2 to get the communication parameters of QM1. Now, QM2 replies with the cluster receiver channel definition that QM3 uses to create automatically a sender channel from QM3 to QM1. At this point, QM3 is able to send messages to QM1 without any manual definition.

Defining a cluster is even easier if you use the MQSeries Explorer GUI interface. You will see this in Chapter 12, "MQSeries and MQSI implementation" on page 319 when we define a cluster for our test environment.

10.1.4.3 Workload balancing and take-over

The next big advantage of clustering is the possibilities for workload balancing and take-over. Assume that we have a local queue `WORKLOAD` hosted on QM1 and QM2. The queue is defined into the cluster.

When an application connects to QM3 and opens the queue `WORKLOAD`, QM3 can now choose to which queue manager it will send messages. QM3 will select that queue manager to which it was able to set up communication. If channel `TO.QM1` has gone into retry and `TO.QM2` is running, QM3 will choose to send messages to QM2. If both channels are active, QM3 will send the messages to both queue managers on a round-robin basis, if the application or administrator has allowed this.

The `MQOPEN` now has a new option to control the spreading of workload. If you want to choose the destination at `MQOPEN` time, you specify the option `MQOO_BIND_AT_OPEN`. If you want to spread the workload over the active systems, you need to use the option `MQOO_BIND_NOT_FIXED`. By using one of these options, an application can control if all messages generated between `MQOPEN` and `MQCLOSE` are sent to one system or to each system that hosts the target queue.

While the above is definitely not a full coverage of MQSeries clustering, we hope that you now have some basic understanding of this feature and how it can help the MQSeries administrator to create a more powerful and reliable MQSeries network with fewer definitions.

For more information on this topic, please refer to the MQSeries product manual *MQSeries: Queue Manager Clusters*, SC34-5349, and the redbook *MQSeries Version 5.1 Administration and Programming Examples*, SG24-5849.

10.1.5 MQSeries security

Because MQSeries queue managers handle the transfer of information that is potentially valuable, you need the safeguard of an authority system. This ensures that the resources that a queue manager owns and manages are protected from unauthorized access, which could lead to the loss or disclosure of the information.

In a secure system, it is essential that none of the following are accessed or changed by any unauthorized user or application:

- Connections to a queue manager
- Access to MQSeries objects such as queues, clusters, channels, and processes

- Commands for queue manager administration, including MQSC commands and PCF commands
- Access to MQSeries messages
- Context information associated with messages

The following section will briefly present you with an overview of how the above tasks are accomplished. For a more detailed MQSeries security overview, please see *MQSeries System Administration*, SC33-1873.

The mqm group

In MQSeries for UNIX systems, UNIX restrictions mean that all user IDs must be defined in lowercase. All queue manager processes run with these IDs:

User ID	mqm
Group	mqm

A user ID with the name mqm whose primary group is mqm is automatically created during installation. You can create the user ID and group yourself, but you must do this before you install MQSeries.

On MQSeries Windows NT systems, if the local mqm group does not already exist on the local computer, it is created automatically when MQSeries for Windows NT is installed. In addition, a domain mqm group may be created on the domain controller. This global group allows control of mqm user access. All privileged user IDs active within this domain should be added to the Domain mqm group.

Administration user ID

You must be a member of the mqm group to administer MQSeries. In particular you need this authority to:

- Use the `runmqsc` command to run MQSC commands
- Administer authorities using the `SETMQAUT` command
- Create a queue manager using the `crtmqm` command

If you are sending channel commands to remote queue managers, you must make sure that your user ID is a member of group mqm on the target system.

OAM

By default, access to queue-manager resources is controlled through an authorization service installable component formally called the Object Authority Manager (OAM) for MQSeries.

OAM is supplied with MQSeries, and is automatically installed and enabled for each queue manager you create, unless you specify otherwise.

The OAM manages users' authorizations to manipulate MQSeries objects, including queues and process definitions. It also provides a command interface through which you can grant or revoke access authority to an object for a specific group of users. The decision to allow access to a resource is made by the OAM, and the queue manager follows that decision. If the OAM cannot make a decision, the queue manager prevents access to that resource.

The OAM works by exploiting the security features of the underlying operating system. In particular, the OAM uses operating system user and group IDs. Users can access queue manager objects only if they have the required authority.

Through OAM you can control:

- Access to MQSeries objects through the MQI. When an application program attempts to access an object, the OAM checks that the user ID making the request has the authorization for the operation requested. In particular, this means that queues, and the messages on queues, can be protected from unauthorized access.
- Permission to use PCF commands.

Note

Using groups, rather than individual principals, for authorization reduces the amount of administration required.

Try to keep the number of groups as small as possible. For example, dividing principals into one group for application users and one for administrators is a good place to start.

The OAM provides two control commands that allow you to manage the authorizations of users. These are:

SETMQAUT Set or reset authority

DSPMQAUT Display authority

Two of the most common authorizations assigned with the `SETMQAUT` command are the ability to put a message on a specific queue by issuing an `MQPUT` call (put authority) and the ability to retrieve a message from a queue by issuing an `MQGET` call (get authority). You will see these authorities mentioned several times throughout this book.

10.1.5.1 MQSeries Explorer security

Before the MQSeries Explorer is enabled, you must ensure that chosen users have the correct level of authorization. This means being one of the following:

- A member of the mqm group
- A member of the administrator group on the machine running the MQSeries Explorer
- Logged on using the SYSTEM ID

Furthermore, some operations may require you to have authorization to use individual objects or object types. The MQSeries Explorer uses existing MQSeries rules for security to ensure that this happens. For example, you must have display authority for a queue to be able to view its attributes in the MQSeries Explorer.

The MQSeries Explorer connects to remote queue managers as an MQI client application. This means that each remote queue manager must have a definition of a server connection channel and a suitable TCP/IP listener. If you do not specify a non-blank value for the MCAUSER attribute of the channel, or use a security exit, it is possible for a malicious application to connect to the same server connection channel and gain access to the queue manager objects with unlimited authority.

The default value of the MCAUSER attribute is a blank. If you specify a non-blank user name as the MCAUSER attribute of the server connection channel, all programs connecting to the queue manager using this channel run with the identity of the named user and have the same level of authority.

10.1.5.2 MQSeries Services security

Access to the MQSeries Services snap-in can be controlled by using the Windows NT distributed component object model (DCOM) Configuration tool. We won't go into the details here, but will just let you know that the facility is available. For more information on setting up MQSeries Services security, refer to *MQSeries System Administration*, SC33-1873.

To start the DCOM customizing facility, click **Start - > Run -> dcomcnfg**. The first screen will show a list of applications. Choose the MQSeries Services from the list, click **Properties**, then select the **Security** tab.

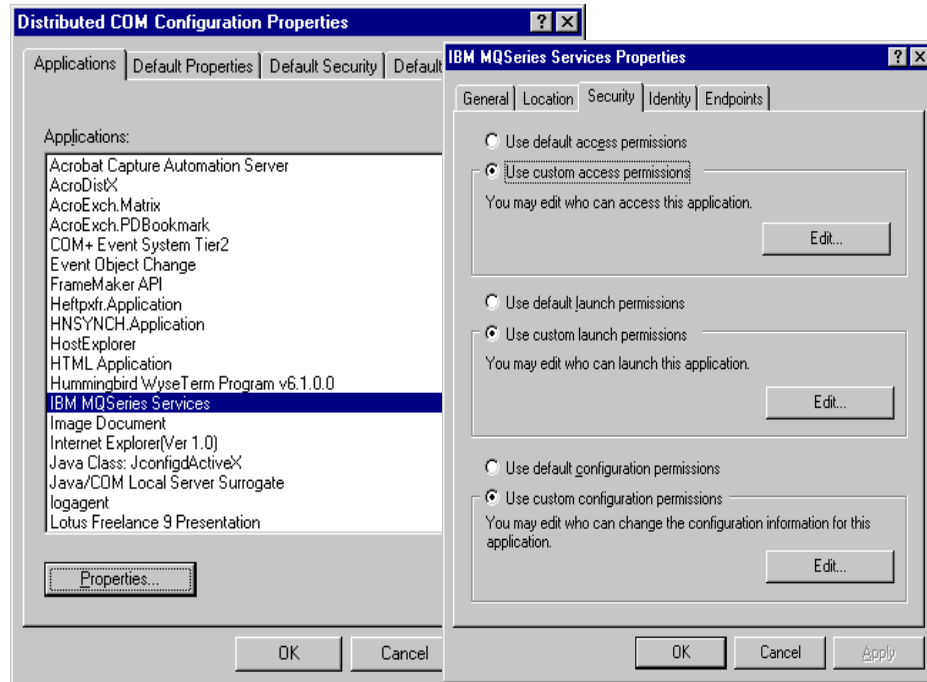


Figure 119. MQSeries Services security using the DCOM customizing facility

10.1.5.3 MQSeries Web Administration security

Your user ID needs the necessary administration privileges on the MQSeries server to perform administration tasks. Therefore, before attempting to log on to MQSeries Web Administration, ensure that you have the correct level of authorization. This means being one or more of the following:

- A member of the mqm group
- A member of the administrator group on the machine running MQSeries Web
- Administration
- Logged on using the SYSTEM ID

Some operations may require you to have authorization to use individual objects or object types. MQSeries Web Administration uses existing MQSeries rules for security to ensure that this happens.

MQSeries Web Administration connects to remote queue managers using MQSC. The Web Administration server adopts the user ID of each logged-on administrator prior to invoking MQSC commands on the administrator's behalf. Therefore, administrators have exactly the same privileges from

MQSeries Web Administration as they would have using the `runmqsc` command locally on the Web Administration server.

10.1.6 MQSeries monitoring

You can use MQSeries instrumentation events to monitor the operation of queue managers.

This section provides a short introduction to instrumentation events, and will introduce the two Windows NT V5.1 service snap-ins that use MQSeries instrumentation to present the user with a GUI event and monitoring tool.

10.1.6.1 Instrumentation events

Instrumentation events cause special messages, called event messages, to be generated whenever the queue manager detects a predefined set of conditions.

For example, the following conditions give rise to a Queue Full event:

- Queue Full events are enabled for a specified queue, and
- An application issues an MQPUT call to put a message on that queue, but the call fails because the queue is full.

Other conditions that can give rise to instrumentation events include:

- A predefined limit for the number of messages on a queue being reached
- A queue not being serviced within a specified time
- A channel instance being started or stopped
- In MQSeries for UNIX systems, an application attempting to open a queue and specifying a user ID that is not authorized

If you define your event queues as remote queues, you can put all the event queues on a single queue manager (for those nodes that support instrumentation events). You can then use the events generated to monitor a network of queue managers from a single node.

MQSeries events are categorized as follows:

Queue manager events	These events are related to the definitions of resources within queue managers.
Performance events	These events are notifications that a threshold condition has been reached by a resource.
Channel events	These events are reported by channels as a result of conditions detected during their operation.

When an event occurs, the queue manager puts an event message on the appropriate event queue (if such a queue has been defined). The event message contains information about the event that you can retrieve by writing a suitable MQI application program.

Each category of event has its own event queue. All events in that category result in an event message being put onto the same queue as shown in Table 12.

Table 12. Event queues

This event queue...	Contains messages from:
SYSTEM.ADMIN.QMGR.EVENT	Queue manager events
SYSTEM.ADMIN.PERFM.EVENT	Performance events
SYSTEM.ADMIN.CHANNEL.EVENT	Channel events

10.1.6.2 MQseries Alert Monitor (Windows NT)

The MQSeries Alert Monitor is an error detection tool that identifies and records problems with MQSeries on a local machine. It displays information about the current status of the local installation of an MQSeries server.

From the MQSeries Alert Monitor, you can:

- Access the MQSeries Services snap-in directly
- View information relating to all outstanding alerts
- Shut down the IBM MQSeries service on the local machine
- Route alert messages over the network to a configurable user account, or to a Windows NT workstation or server

If the task bar icon indicates that an alert has arisen, double-click the icon to open the alert monitor display. This dialog shows a tree view, grouped by queue manager, of all the alerts that are currently outstanding. Expand the nodes of the tree to see which services are alerted and look at the following pieces of information relating to the service:

- The date and time of the most recent alert for the service
- The command line that failed
- The error message describing why the service failed

10.1.6.3 Performance Monitor (Windows NT)

The Performance Monitor is a standard component of Windows NT. It enables you to select and display a variety of data about the performance of the Windows environment, as tabular reports or graphs. You can use it to monitor

the depth of messages on MQSeries queues, and the rates of message arrival and removal.

You access the Performance Monitor by clicking **Start -> Programs -> Settings -> control panel -> Administrative Tools -> Performance**.

When you first start it, the display is empty. To add a resource that you want, click the **+** on the chart pane to add counters as shown in Figure 120.

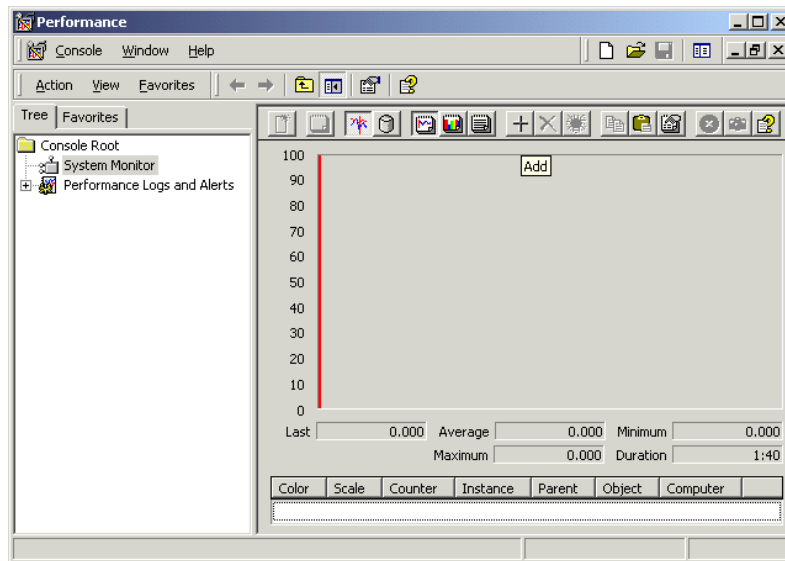


Figure 120. Performance Monitor

In the Add Counters window (Figure 121) chose what you want to monitor:

- Select **MQSeries Queues** from the performance object drop-down.
- Select what you want to monitor:
 - a. The current queue depth, that is, how many messages are in the queue.
 - b. The queue depths as a percentage of the maximum queue depth, that is, how full the queue is.
 - c. The enqueue rate in messages per second, that is, the number of messages placed in the queue. This is not necessarily the number of MQPUTs; each message segment counts as one message.
 - d. The dequeue rate in messages per second, that is, the number of messages removed from the queue.

- Then select a queue from the instance list. The instance list contains only queues that have had messages inserted or removed before the Performance Monitor started.
- Click **Add** for each selected counter.
- Click **Close** when finished.

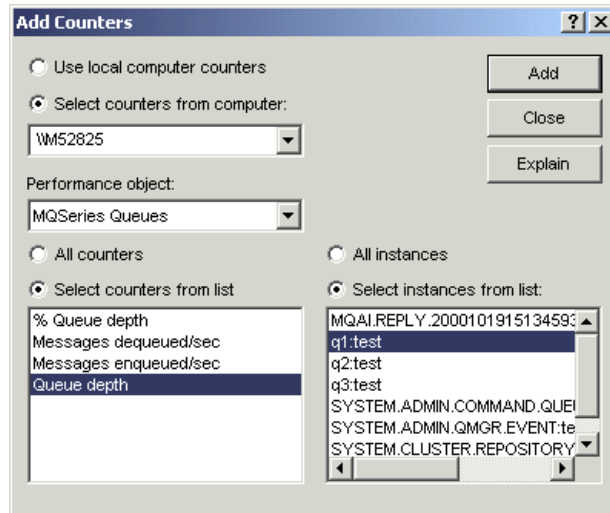


Figure 121. Add counter window

The graph is now displayed. Update the graph properties using the menu bar to control the graph details.

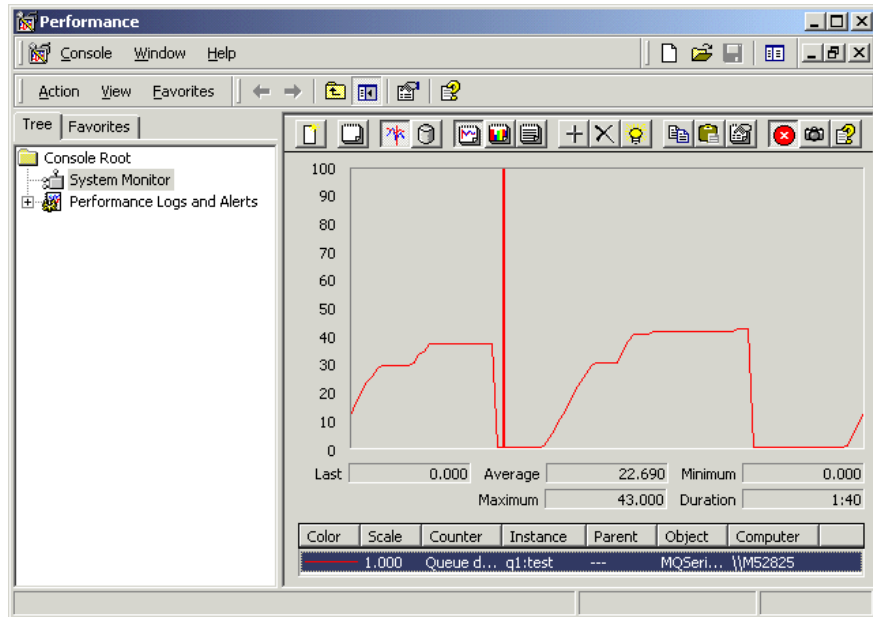


Figure 122. Example queue depth graph

10.1.7 MQSeries restart and recovery

MQSeries ensures that messages are not lost by maintaining records (logs) of the activities of the queue managers that handle the receipt, transmission, and delivery of messages. It uses these logs for three types of recovery:

- Restart recovery, when you stop MQSeries in a planned way.
- Crash recovery, when MQSeries is stopped by an unexpected failure.
- Media recovery, to restore damaged objects.

In all cases, the recovery restores the queue manager to the state it was in when the queue manager stopped, except that any in-flight transactions are rolled back, removing from the queues any messages that were not committed at the time the queue manager stopped. Recovery restores all persistent messages; nonpersistent messages are lost during the process.

MQSeries supports two type of logging:

- Circular logging
- Linear logging

Each type of logging stores the recorded data in a set of files. The differences between the two types of logging are the contents and the way that the files are linked together.

With circular logging, the set of log files are effectively linked together to form a ring. When data is collected, it is written sequentially into the files in such a way as to re-use the log files in the ring. You can use circular logging for crash recovery and restart recovery.

With linear logging, the log is maintained as a continuous sequence of files. When data is collected, it is written sequentially into the log files; the space in the files is not re-used, so that you can always retrieve any record from the time that the queue manager was created.

Because disk space is finite, you might have to plan for some form of archiving. Also, if you are handling a high volume of persistent messages, all your log files will eventually be filled. This will result in operator messages being written to an error log file, and some action will need to be taken by the system administrator to make more log space available, or to reuse the existing space. You can use linear logging for all three types of recover.

10.1.7.1 Managing log files

If you are using circular logging, ensure that there is sufficient space to hold the log files. You do this when you configure your system. The amount of disk space used by the log does not increase beyond the configured size, including space for secondary files to be created when required.

If you are using a linear log, the log files are added continually as data is logged, and the amount of disk space used increases with time. If the rate of data being logged is high, disk space is consumed rapidly by new log files.

Over time, the older log files for a linear log are no longer required to restart the queue manager or perform media recovery of any damaged objects.

Periodically, the queue manager issues a pair of messages to indicate which of the log files is required:

- Message AMQ7467 gives the name of the oldest log file needed to restart the queue manager. This log file and all newer log files must be available during queue manager restart.
- Message AMQ7468 gives the name of the oldest log file needed to do media recovery.

Any log files older than these do not need to be online. You can copy them to an archive medium such as tape for disaster recovery, and remove them from

the active log directory. Any log files needed for media recovery but not for restart can also be off-loaded to an archive.

If any log file that is needed cannot be found, operator message AMQ6767 is issued. Make the log file, and all subsequent log files, available to the queue manager and retry the operation.

10.1.7.2 Logging guidelines

Follow these guidelines when placing and managing your MQSeries logs:

- When choosing a location for your log files, remember that operation is severely impacted if MQSeries fails to format a new log because of lack of disk space.
- To maximize the efficiency of logging, the logging volumes should be separated from the data volumes. On small servers, this may not be practical. However, by doing so, the potential for contention between logging and writing of the queue data is reduced. This eliminates the potential of a single point of failure for both the logs and the associated data.
- Whenever possible, place the log files on multiple disk drives in a *mirrored* arrangement. This gives protection against failure of the drive containing the log. Without *mirroring*, you could be forced to go back to the last backup of your MQSeries system.
- Circular logging is the easiest to manage, since the logs are simply re-used in a circular fashion. However, in some situations, recovery may be impossible while in this mode. Linear logs guarantee recovery but require management to prevent filling up all available disk space. Eventually, linear logs that are no longer required will need to be archived and/or deleted. This should be automated to prevent unexpected outages.
- If you are using a circular log, ensure that there is sufficient space on the drive for at least the configured primary log files. You should also leave space for at least one secondary log file, which is needed if the log has to grow.
- If you are using a linear log, you should allow considerably more space; the space consumed by the log increases continuously as data is logged.

10.1.7.3 Backing up MQSeries resources

To take a backup of a queue manager's data, you must:

1. Ensure that the queue manager is not running. If your queue manager is running, stop it with the `endmqm` command.
2. Locate the directories under which the queue manager places its data and its log files.

3. Take copies of all the queue manager's data and log file directories, including all subdirectories. Make sure that you do not miss any of the files, especially the log control file and the configuration files. Some of the directories may be empty, but they will all be required if you restore the backup at a later date, so it is advisable to save them too.
4. Ensure that you preserve the ownerships of the files. For MQSeries for UNIX systems, you can do this with the tar command.

On Windows NT, starting with V5.1 of MQSeries, all MQSeries resources are saved in the NT registry. As a result, make sure that when backing up MQSeries resources to include a backup of the NT registry as well.

It is a good idea to save MQSeries resource definitions in MQSC command files, and scripts, so that these resources can be quickly re-defined from scratch if required.

10.2 .MQSeries Integrator system management

MQSeries Integrator networks are comprised of broker domains. In this section we will cover the operation and management of the various components that comprise a broker domain:

- Message brokers
- Configuration Manager
- Control Center
- User Name Server

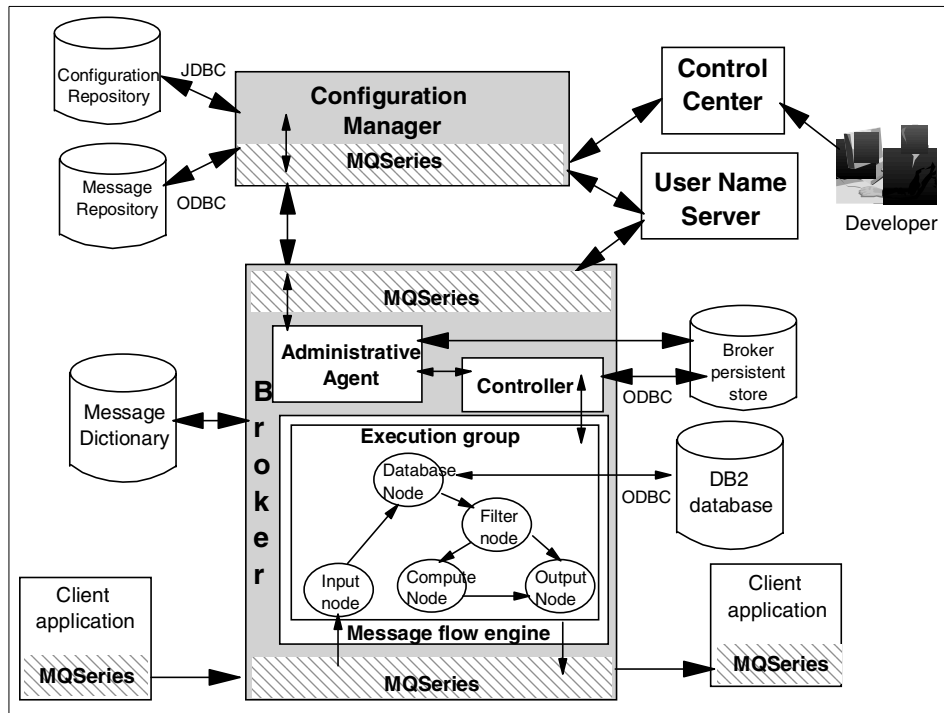


Figure 123. The broker domain

Table 13 shows the platform and database support for MQSI V2.0.1 components.

Table 13. MQSI software requirements

MQSI V2.0.1 components	Platform	Database software for MQSI databases
Configuration Manager	Windows NT 4.0	-DB2 UDB 5.2 + fixpak 12 or 6.1 ¹
Control Center	Windows NT 4.0	
¹ Check the readme.txt for fixpak information. DB2 6.1 is the only DBMS supported by MQSI that permits a database to participate as a Resource Manager in a distributed XA transaction, and coordinated by MQSeries as the XA Transaction Manager. In MQSI, this is referred to as supporting a globally coordinated message flow.		

MQSI V2.0.1 components	Platform	Database software for MQSI databases
Broker	Sun Solaris 7	-DB2 6.1 ¹ -Oracle 8.1.5 -Sybase 11.5 or 12
	AIX V4.3	-DB2 UDB 6.1 ¹ -Oracle 8.1.5 -Sybase 11.5 or 12
	Windows NT 4.0	-DB2 UDB 5.2 + Fixpak 12 or 6.1 ¹ -Microsoft SQL Server 6.5 +SP5a or 7.0 + SP1 -Oracle 8.1.5 -Sybase 11.5 or 12
User Name Server	Sun Solaris 7 AIX V4.3 Windows NT 4.0	
Applications	Sun Solaris 7	-DB2 6.1 ¹ -Oracle 7.3.4 or 8.1.5 -Sybase 11.5 or 12
	AIX V4.3	-DB2 UDB 5.2 SP 12 or above, V6.1 ¹ -Oracle 7.3.4 or Oracle 8.1.5 -Sybase 11.5 or 12
	Windows NT 4.0	-DB2 UDB 5.2 + fixpak 12 or 6.1 ¹ -Microsoft SQL Server 6.5 +SP5a or 7.0 + SP1 -Oracle 7.3.4 or 8.1.5 -Sybase 11.5 or 12
¹ Check the readme.txt for fixpak information. DB2 6.1 is the only DBMS supported by MQSI that permits a database to participate as a Resource Manager in a distributed XA transaction, and coordinated by MQSeries as the XA Transaction Manager. In MQSI, this is referred to as supporting a globally coordinated message flow.		

10.2.1 Message brokers

An MQSI application consists of message flows that perform transformation and routing functions. These message flows are assigned to brokers. Any number of brokers can be added within a broker domain and more than one broker may reside on a physical system. Each broker has a unique name, called the Broker Instance name. Message flows within a broker are grouped by assigning them to execution groups. Each execution group is assigned to a separate message flow engine.

Each broker has a *persistent state datastore*, implemented using a unique set of tables in a relational database. The datastore holds all persistent state data needed by the broker, including:

- The deployed message flow definitions
- Persistent subscriptions
- Publish/subscribe neighbors

There is one *controller* in each broker whose main purpose is to ensure that the defined broker processes are running, including one process for each execution group and a process for the administrative agent. The controller uses an internal cached broker definition table created from information it finds in the persistent datastore. The controller monitors changes to both the definition table and process state, making sure the process state reflects the table. If a process dies, it determines the action to take, generally restarting the process.

The *administrative agent* manages updates to the broker definition table and processes messages to start and stop the broker. It sends requests to start or stop execution engines to the controller and configuration changes to specific message flow engines. When an engine receives a configuration update from the administrative agent, it ensures that the changes are fully made, suspending message processing during the changes and resuming with the new updates. Configuration changes to multiple brokers can be processed transactionally, so that all brokers make the updates, or the updates are backed out. Although the administrative agent can be run in the same process as the controller, it is advisable to run it as a separate process to improve the reliability of the monitor.

Each broker requires a set of MQSeries queues on the queue manager associated with the broker domain. These are created automatically when the broker is created. Each broker also requires its own queue manager. This queue manager can also host the Configuration Manager and/or the User Name Server, but not another broker. If this queue manager does not exist, it is created for you.

10.2.1.1 Administering multiple brokers

When using duplicate brokers, management of MQSeries definitions can be easily handled by using packages of MQSeries commands (MQSC). MQSC can be used to deploy a set of queue definitions that are required on queue managers that have a broker role. In this way, the same set of commands can be re-used when you wish to add an additional instance of a broker to the domain. The following example MQSC extract sets up four cluster queue definitions used by the broker in our example scenario:

```
def ql(MQSI.PROF.REQUESTS) cluster(ITSO.CLUSTER) replace
def ql(MQSI.PROF.UPDATES) cluster(ITSO.CLUSTER) defpsist(YES) replace
def ql(MQSI.FAILURE) cluster(ITSO.CLUSTER) replace
def ql(MQSI.AUDIT) cluster(ITSO.CLUSTER) replace
```

The following sample MQSC sets up the alias queue definitions used by the application code in our example. MQSI.PROF.REPLIES is a local queue that will be used as a reply queue for the application. The remaining definitions define alias queues, which relate the queue names as used by the application to the real queue names which will all be visible because they have been defined as cluster queues either here or elsewhere in the network.

```
def ql(MQSI.PROF.REPLIES) cluster(ITSO.CLUSTER) replace
def qalias(ITSO.PROF.REQ.IN) targq(MQSI.PROF.REQUESTS)
cluster(ITSO.CLUSTER) replace
def qalias(ITSO.PROF.UPD.IN) targq(MQSI.PROF.UPDATES)
cluster(ITSO.CLUSTER) replace
def qalias(ITSO.PROF.REPLY) targq(MQSI.PROF.REPLIES) cluster(ITSO.CLUSTER)
replace
```

10.2.2 The Configuration Manager

The components and resources in a broker domain are controlled by the Configuration Manager, which is responsible for maintaining the broker domain configuration. It manages the initialization, deployment, and configuration updates of broker and message processing operations, including the authorization to perform these actions. These actions are in response to functions performed using the Control Center.

The Configuration Manager uses relational database tables to store information. Two sets of tables are used, one for configuration information known as the configuration repository, and one for message format information known as the message repository.

The Configuration Manager requires a local MQSeries queue manager to host its services. This queue manager is identified or created during the process to create the Configuration Manager and a set of required queues are defined on it. The queues defined will include a server connection queue to be used by the Control Center to communicate with the Configuration Manager and traditional or cluster sender and receiver channels to each broker in the broker domain.

10.2.3 The Control Center

The Control Center is the business administration tool used to define message flows for applications and to access the databases and resources

used by MQSeries Integrator. Any number of Control Center instances can be installed and invoked.

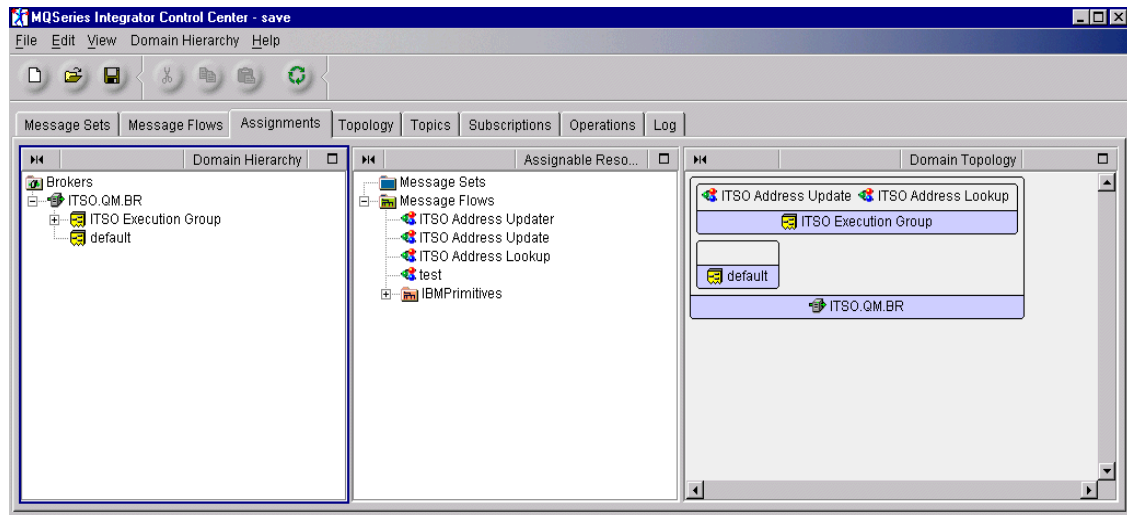


Figure 124. MQSI Control Center

The Control Center depends on MQSeries classes for Java for its connection with the Configuration Manager. The Control Center dynamically creates a client connection to connect to the Configuration Manager's queue manager using the information provided at invocation.

Chapter 9, "Developing the MQSI application" on page 183 shows an example of using the Control Center to build message flows. Chapter 12, "MQSeries and MQSI implementation" on page 319 shows how to start the Control Center to a particular Configuration Manager, create broker execution groups, assign message flows to those groups, and how to deploy a message flow to the broker.

10.2.4 The User Name Server

The User Name Server is an optional resource used to implement topic security in a publish/subscribe environment. It monitors the underlying security subsystem and provides information about users and groups that it shares with the Configuration Manager and brokers.

Message flows that give publish/subscribe service to applications might require topic security. This gives the ability to control the authority of the application, based on the user ID it is running under, to do the following:

- Publish on topics
- Subscribe to topics
- Request persistent delivery of messages

You can build the User Name Server without defining or implementing any of the functions. If you think you will be using publish/subscribe and topic security now or in the future, it may be easier to incorporate the User Name Server initially, implementing the functions later when they are required.

The User Name Server requires an MQSeries queue manager and a set of queues. This queue manager may be shared with the Configuration Manager, broker, or both.

10.2.5 MQSeries guidelines for MQSI

The MQSeries Integrator broker domain is built using MQSeries queue managers and queues as its infrastructure. The MQSeries components required for the operation of the broker domain (exclusive of those required for the applications) are created automatically when the MQSI components are created.

The MQSI Configuration Manager, User Name Server, and brokers are created using commands. These commands will create the component's associated queue manager if it does not exist, and will always create the component's named queues. They do not, however, create the queues required for communication between the queue managers. Transmission queues must be defined to the queue managers for communication between the following:

- Every broker and the Configuration Manager
- Every broker and the User Name Server
- The Configuration Manager and the User Name Server

Table 14 shows which MQSI components require a queue manager and which of these queue managers can be shared.

Table 14. MQSeries Integrator Queue Managers

MQSI Component	Needs a Queue Manager ?	Can Share a Queue Manager with:		
		Configuration Manager	User Name Server	Broker
Configuration Manager	Yes	N/A ¹	Yes	Yes

MQSI Component	Needs a Queue Manager ?	Can Share a Queue Manager with:		
		Configuration Manager	User Name Server	Broker
User Name Server	Yes	Yes	Yes	Yes
Broker	Yes	Yes	Yes	No ³
Control Center	No ²	N/A	N/A	N/A
¹ There is only one Configuration manager in a broker domain ² Client/Server connection to Configuration Manager's queue manager ³ Each broker defined will require a unique MQSeries queue manager				

For each queue manager hosting an MQSI component, you will need to define traditional sender and receiver channels to any other MQSI component queue managers it will need to communicate with. If you are using MQSeries clustering for your queue managers, you will only need to define cluster sender and receiver channels to one of the cluster repository queue managers.

Applications that use broker services also use MQSeries to send and receive messages to the brokers. The MQSeries resources required by your applications are application specific and must be also be created manually.

10.2.5.1 MQSeries clusters

Consider using MQSeries clustering (see 10.1.4.2, "Clustering details" on page 250) when you design the MQSeries network underlying your broker domain. Clustering decreases the amount of MQSeries system administration significantly, increases availability, and provides workload balancing.

By joining the queue managers for each component to a cluster, you will simplify the administrative tasks associated with defining the MQSeries network. This can be a great advantage if you are using a publish/subscribe collective, which requires total MQSeries interconnection between the brokers.

There are, however, implications for applications in a clustering environment. Clustering provides workload management among queue managers by allowing duplicate queues to exist. Messages can be sent to any queue manager that hosts a duplicate copy of a queue. In this situation, the application must be examined for any logic that makes it necessary for messages (reply and request) to be handled by the same queue manager. If this requirement exists, steps must be taken to ensure the integrity of the

message flow. To find out how to handle this see *MQSeries Queue Manager Clusters*, SC34-5349.

Note: SYSTEM.BROKER queues are created automatically to support the MQSI components. They are not defined as cluster queues. Do *not* change this attribute.

10.2.6 MQSI databases

We have already mentioned the way MQSI uses databases for its administrative functions. Three different types of information are stored in distinct tables in relational databases to be used by the MQSI components. The Configuration Manager stores information in two sets of tables called the configuration repository and message repository. Each broker also has a set of tables to be used as a persistent datastore. In order to understand the systems management implications of how these databases are set up, we will take a quick look at how the databases are used.

When you create and modify resources, for example a message flow, in a broker domain using the Control Center, the changes are initially stored in your local system. When you deploy these changes, the Configuration Manager updates the configuration repository. The data in this repository can be viewed and managed using the Control Center, which interacts with the Configuration Manager on your behalf.

The set of tables known as the message repository is also managed by the Configuration Manager. It contains all the message and message set definitions you have created using the Control Center and deployed in your broker domain. If you import externally defined message definitions using the Control Center, these are also stored in this repository.

When changes are made to the broker's environment, the Configuration Manager sends messages to the broker to update its local persistent store. For example, if you assign and deploy a new message flow to the broker, the data is updated.

The database products that can be used for each type of database storage can be seen in Table 13 on page 266. The message repository and the broker databases are accessed using ODBC. ODBC drivers for DB2 and SQL Server are provided with the database products. ODBC drivers for Oracle and Sybase are provided by MQSeries Integrator. A JDBC connection is required for the Configuration Manager access to the configuration repository. An example of setting up these connections can be seen in Chapter 12, "MQSeries and MQSI implementation" on page 319.

MQSI can use tables in a database, whether that database is local or remote to the MQSI component. You can set up a database for each component if you choose, or you can set up a central database on a shared server. The placement of these databases can have an impact on systems performance and systems management. The issues related to performance are discussed in 7.2.3, “Placement of MQSI databases” on page 113.

Where systems management is concerned, there are advantages and disadvantages to using remote databases (see Table 15). You must refer to the documentation supporting the database type you are using to determine the best options for your specific environment.

Table 15. Remote database access advantages vs. disadvantages

Advantages of using a remote db	Disadvantages
<ul style="list-style-type: none"> -A central, shared, DB server is easier to manage, back up, and restore. -A central shared DB server can be optimally tuned for data access workloads -A remote database offloads database processor cycles and utilization -Using a remote database means you only need a thin database client, vs. installing a full DB server on the MQSI node 	<ul style="list-style-type: none"> -Performance may be an issue depending on the network and number of database accesses from different components -Increases network traffic

10.2.7 MQSeries Integrator commands and operations

Now that we have all functional requirements in place, MQSI commands will be used to create, configure, and control the various MQSI components.

10.2.7.1 MQSI commands

MQSI components can be created, deleted, and configured by using MQSI commands. These commands can be entered from the command line or they can be generated by using the MQSI Command Assistant (Windows NT only). MQSI components can be controlled (started or stopped) using commands or if on Windows NT, using the Windows NT services window.

Note that:

- Each command must be issued on the system on which the resource it relates to is defined (or is to be created).
- All MQSI commands have dependencies on MQSeries functions. You must ensure that MQSeries is available before issuing these commands.

With each command, there are flags used to specify parameters. Let's take a brief look at some of these commands. To see the syntax and flags for these commands, refer to the *MQSeries Integrator Administration Guide*, SC34-5792.

Commands to create and delete components

The MQSI commands to create the MQSI components are:

- `mqsicreateconfigmgr`
- `mqsicreatebroker` (for each broker)
- `mqsicreateusernameeserver`

These commands have parameters that are needed to bind the MQSI component to its resources, including:

- Queue manager name to host the component. If the queue manager does not exist, it will be created.
- Database (or data source) name (broker and Configuration Manager). The appropriate tables will be defined in the database.
- User ID and passwords needed to access the database or to run the Windows NT service under.

MQSI commands are also available to delete the MQSI components. Flags will determine if the underlying queue manager and database tables are also deleted.

- `mqsideleteconfigmgr`
- `mqsideletebroker` (for each broker)
- `mqsideleteusernameeserver`

Commands to change components

Once the MQSI components have been created, the MQSI change commands are used to change the MQSI components "bindings", such as the queue manager name or user ID and password used to access the database or for the Windows NT service.

- `mqsichangeconfigmgr`
- `mqsichangebroker`
- `mqsichangeusernameeserver`

Operational (start and stop) commands

MQSI commands exist to start and stop MQSI components. On Windows NT the MQSI components run as Windows NT services and can also be managed as such.

The logical order and commands to start the MQSI components are:

- mqsistart UserNameServer
- mqsistart <broker_instance_name>
- mqsistart ConfigMgr

The logical order to stop MQSI components

- mqsistop <broker_instance_name>
- mqsistop ConfigMgr
- mqsistop UserNameServer

Remember, the commands must be entered on the system hosting the component.

List and trace commands

List and trace commands are available to list MQSI components and to format and read MQSI traces:

- mqsilist
- mqsichangetrace
- mqsiformatlog
- mqsilcc
- mqsireadlog
- mqsireporttrace

10.2.7.2 The Command Assistant (Windows NT only)

The Command Assistant is a graphical interface that supports MQSI commands to create, delete, or change a component. It provides a series of easy-to-use windows that significantly simplify the task of creating and changing components. The create commands, in particular, have a large number of parameters. The Command Assistant displays all the parameters with meaningful labels and provides integrated, context-sensitive help information, and indicates whether each parameter is mandatory or optional.

To invoke the command assistant select **Start -> Programs -> IBM MQSeries Integrator 2.0 -> Command Assistant**.

10.2.7.3 Command results and MQSI messages

For Windows NT, the command results and MQSI messages appear in the Windows NT event application viewer. To open the event viewer select **Start -> Programs -> Administrative Tools (common) -> Event Viewer**. Then select **Log -> application**.

Date	Time	Source	Category	Event	User	Computer
9/14/00	10:19:33 PM	MQSiv201	None	1003	N/A	M23BK59Z
9/14/00	10:19:31 PM	MQSiv201	None	1709	N/A	M23BK59Z
9/14/00	10:19:30 PM	MQSiv201	None	8280	N/A	M23BK59Z
9/14/00	10:19:30 PM	MQSiv201	None	8255	N/A	M23BK59Z
9/14/00	10:19:27 PM	MQSiv201	None	2001	N/A	M23BK59Z
9/14/00	10:10:29 PM	MQSiv201	None	1805	N/A	M23BK59Z
9/14/00	10:01:38 PM	MQSiv201	None	2002	N/A	M23BK59Z
9/14/00	9:03:02 PM	MQSeries	None	9001	N/A	M23BK59Z
9/14/00	9:03:02 PM	MQSeries	None	9545	N/A	M23BK59Z
9/14/00	9:03:02 PM	MQSeries	None	9001	N/A	M23BK59Z
9/14/00	9:03:02 PM	MQSeries	None	9545	N/A	M23BK59Z
9/14/00	8:55:37 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	8:54:17 PM	MQSeries	None	9001	N/A	M23BK59Z
9/14/00	8:49:44 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	7:38:04 PM	MQSeries	None	9001	N/A	M23BK59Z
9/14/00	7:38:04 PM	MQSeries	None	9545	N/A	M23BK59Z
9/14/00	7:23:00 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	7:22:59 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	4:44:13 PM	MQSiv201	None	1003	N/A	M23BK59Z
9/14/00	4:44:11 PM	MQSiv201	None	1709	N/A	M23BK59Z
9/14/00	4:44:11 PM	MQSiv201	None	8280	N/A	M23BK59Z
9/14/00	4:44:10 PM	MQSiv201	None	8255	N/A	M23BK59Z
9/14/00	4:44:07 PM	MQSiv201	None	2001	N/A	M23BK59Z
9/14/00	4:19:55 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	4:18:57 PM	MQSeries	None	9999	N/A	M23BK59Z
9/14/00	4:18:56 PM	MQSeries	None	9202	N/A	M23BK59Z
9/14/00	4:18:55 PM	MQSeries	None	9002	N/A	M23BK59Z
9/14/00	4:17:56 PM	MQSeries	None	9999	N/A	M23BK59Z
9/14/00	4:17:56 PM	MQSeries	None	9202	N/A	M23BK59Z

Figure 125. Windows NT Event Viewer

Double-click an event to see the details. As you can see from Figure 126, the information in the events can be quite useful for debugging purposes.

Event Detail

Date: 9/14/00 Event ID: 8255
Time: 10:19:30 PM Source: MQSiv201
User: N/A Type: Information
Computer: M23BK59Z Category: None

Description:

[Configuration Manager] User Name Services are disabled.

The Configuration Manager communicates with the User Name Server in order to procure user and group information for configuring topic ACLs.

No action unless you want to manage topic ACLs. If you want to manage topic ACLs you must reconfigure the Configuration Manager by providing a queue manager name for the User Name Server.

Data: ☒ Bytes ☐ Words

0000: 32 00 38 00 34 00 00 00 2 . 8 . 4 . . .
0008: 43 00 6f 00 6e 00 66 00 C . o . n . f .
0010: 69 00 67 00 4d 00 67 00 i . g . M . g .
0018: 72 00 2e 00 61 00 67 00 r . . . a . g .
0020: 65 00 6e 00 74 00 00 00 e . n . t . . .

Close Previous Next Help

Figure 126. MQSI event

In AIX, all MQSI V2 messages not generated by the command line utilities are written to the syslog, but the syslog daemon (syslogd) must be configured to write all “user” messages to a file.

To configure the syslog you need to (as root) edit the file /etc/syslog.conf. This file contains the definitions specifying where to write messages written to the syslog. All MQSI messages are written to the “user” facility so you need to add a line starting with the text “user” and then select the level of messages you want to see. For example, to direct all user facility messages to /var/log/syslog.user, add the following line to the end of the syslog.conf file:

```
user.debug /var/log/syslog.user
```

Before committing the changes by restarting syslogd you must create the syslog.user file:

```
touch /var/log/syslog.user
chown root:mqbrkrs /var/log/syslog.user
chmod 75 //var/log/syslog.user
```

You must restart the syslog daemon for your changes to take effect, using the command:

```
refresh -s syslogd
```

10.2.8 Control Center operations

The Control Center can be used to:

- Develop, modify, assign, and deploy message flows.
- Develop, modify, assign, and deploy message sets.
- Define the broker domain topology and create collectives.
- Create and modify access control lists (ACLs) to control publish/subscribe security.
- View status information.

As you can see, the above tasks will most likely be performed by different people in your organization. For instance you would have an MQSI developer develop and modify your message MQSI message flows, but the action of deploying these message flows to a production broker for execution would most likely be performed by the MQSI administration.

MQSI operational roles address the above situation. The role determines what the user can view within the Control Center, and therefore limits the tasks that are available to that user. Later, in 10.2.10, “MQSeries Integrator

security” on page 290, we will see how to map the MQSI roles to the MQSI security groups.

The Control Center roles are:

- All
- Message flow and message set developer
- Message flow and message set assigner
- Operational domain controller
- Topic security administrator

To chose a role click **File -> Preferences -> User’s role** and select the role you are fulfilling. Be sure to check the **Show Log** box.

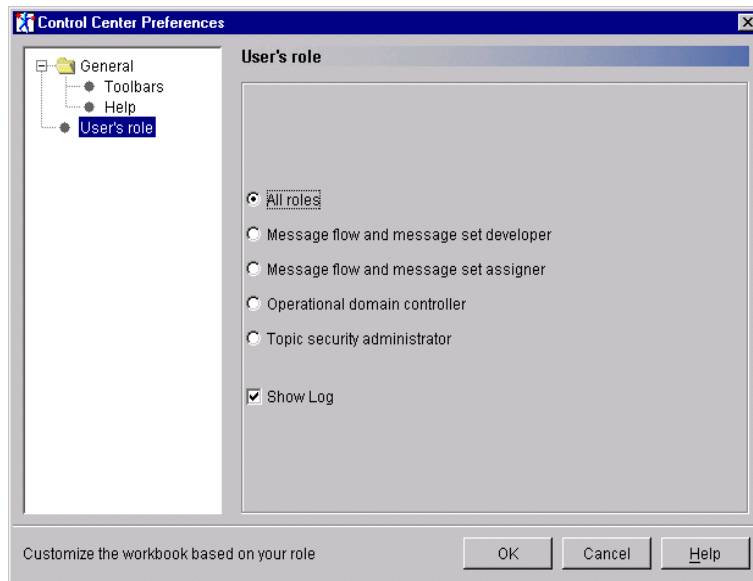


Figure 127. Control Center role selection

The roles that the Control Center user chooses (and is allowed to perform) will determine what Control Center views, and thus functionality, the user has access to. The views are represented as tabs at the top of the window.

Control Center View Tabs

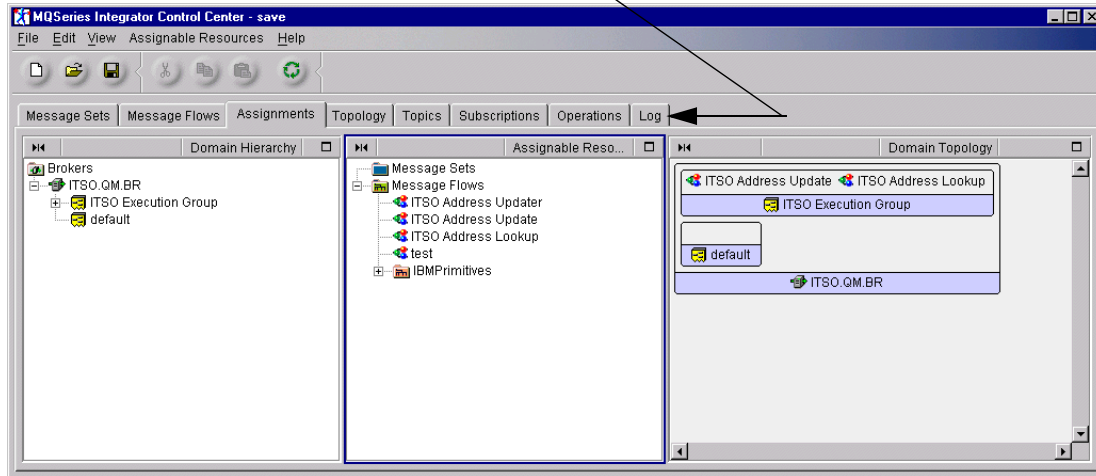


Figure 128. Control Center views

The Control Center Views are:

- Message Sets
- Message Flows
- Assignments
- Topology
- Topics
- Subscriptions
- Operations
- Log (available if **Show Log** is checked as shown in Figure 127 on page 279)

The following table shows the Control Center role to Control Center view mapping.

Table 16. Control Center role authorities

	Message sets	Message flows	Assignments	Topology	Topics	Subscriptions	Operation
All	Y	Y	Y	Y	Y	Y	Y
Message developer	Y	Y					
Message assigner			Y				
Operational controller			Y	Y	Y	Y	Y
Subscription admin				Y	Y		

10.2.8.1 Change control

It is certainly possible that there will be multiple people involved in the design and control of an MQSI network. There are multiple roles that these people will fill, and most likely, there will be multiple people assigned to each role. This makes management of the changes in the MQSI network an important aspect of systems management.

In addition to the roles described earlier and limiting who can make changes, the control center exerts change management control by enforcing a check in/ check out method for preventing simultaneous updates to resources.

There are three different versions of configuration data:

Local A copy of configuration data on which a user is working. When the Control Center is started, the local version of the configuration data is presented. To work with the local copy of configuration data, you need to check out the resources from the shared copy. While checked out, other users are prevented from updating the resource.

Any changes made to a local version will not be visible to other

users until the status of the resource is changed by checking in the resource.

Shared A version of the configuration data that is shared by all the users of the Control Center. Once resources are checked out to the local workspace, they can be modified, and once modified checked in.

Deployed This is the active version of configuration data that is operational at the broker.

10.2.8.2 Managing message sets

MQSI applications are based on receiving messages from a queue, parsing the contents in order to perform some function, then placing the message back on a queue to go to the target application. The sending and receiving applications know the format of the message. In order for MQSI to be able to process the message, it also has to have an understanding of the message structure.

There are two ways MQSI views a message: the logical view and the physical view. The physical view is the physical content of the message. The logical view of a message is the structure of a message: its fields, the order and the relationship between the fields. In MQSI terms this is the *message type*. Message types are grouped in *message sets*. You can think of a message set as the collection of messages types used in a single application or project.

MQSI makes a distinction between self-defined messages and predefined messages. Self-defined messages use the XML standard to structure their content. We will also refer to this type of messages as *generic XML messages*. To use self-defined messages you do not have to do anything specific. For predefined messages, you need to make MQSI aware of the logical view and the physical view of the message.

In MQSI, a *message domain* identifies the parser to use when a broker needs to interpret message. There are four message domains:

- XML for self-defining messages
- MRM for messages defined using the Control Center
- NEON for messages defined using the NEONFormatter
- BLOB for messages that have no definition

Messages managed by the Message Repository Manager (MRM) domain are managed in the Control Center using the Message Sets tab, shown in Figure 129. The definitions built here are stored in the message repository.

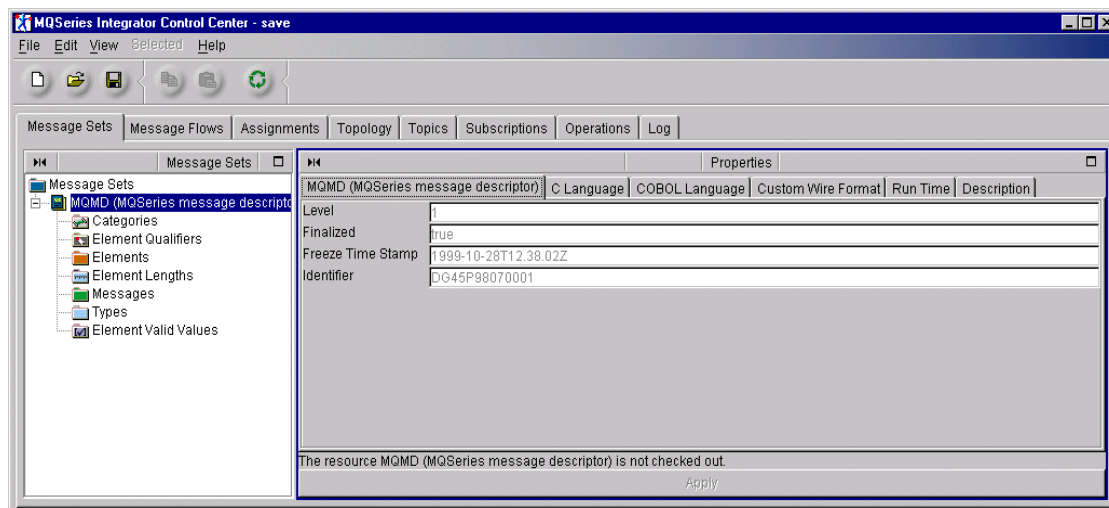


Figure 129. Message sets view

This view allows you to define the format of the messages using message components, and group them into message sets. This will include the specification of the custom wire format, and importing and exporting C structures or COBOL copybooks. The left-hand pane shows a tree view of the message sets in your workspace. The right-hand pane displays the properties of the currently selected entry.

Once the message sets are built, they need to be deployed to the brokers that will need them. The MRM builds a message dictionary for each message set needed and sends it to the broker.

For detailed information on building message definitions, see *MQSeries Integrator Using the Control Center*, SC34-5602.

10.2.8.3 Managing message flows

The logic of the MQSI application is structured using message flows. Message flows are built, maintained, and deployed using the Control Center. To create message flows or to work with their content, choose the Message Flows tab. This view is shown in Figure 130.

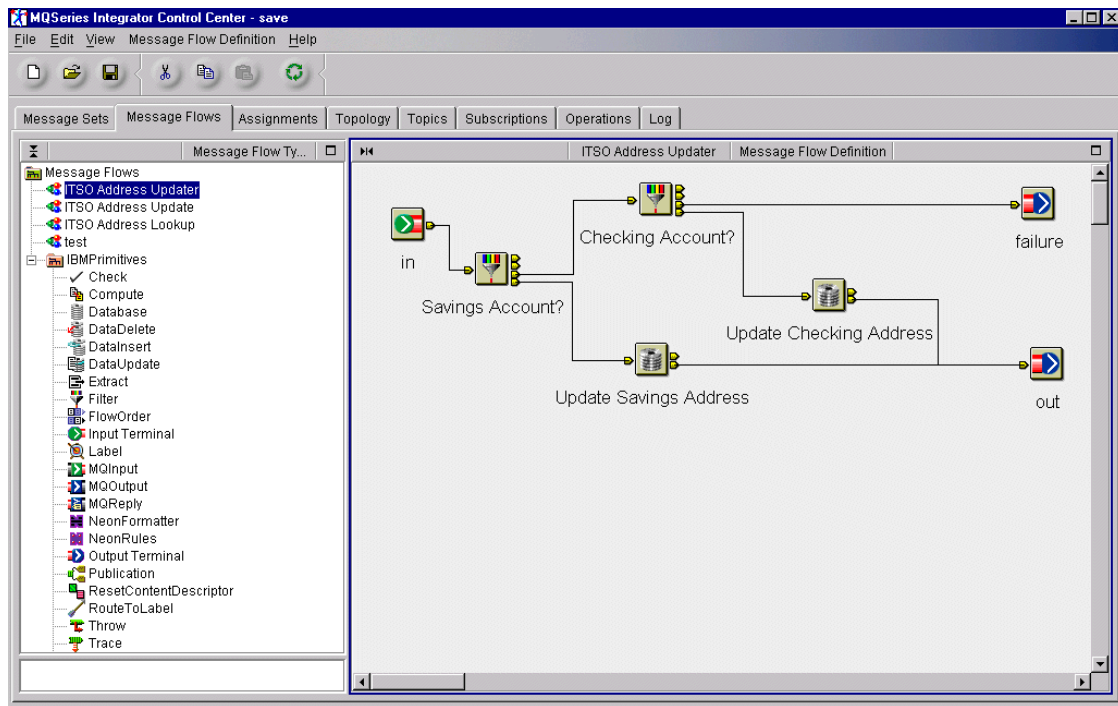


Figure 130. The Message Flows view

The left-hand pane shows a tree view of the message flows and nodes, both user defined and the IBM primitives, in your workspace. The right-hand pane contains an arrangement of graphical symbols that represent the message flow nodes in a selected message flow. Message flows are built by taking nodes in from the left pane, dragging them to the right pane, defining their properties, and then connecting their input and output terminals to other nodes.

Message flow nodes are discussed in 7.4, “Message flow components” on page 117. Creating message flows is discussed in 9.6, “Building the message flows” on page 193.

10.2.8.4 Using the Topology view

The Topology view allows you specify the brokers you will be deploying applications to. The broker is identified by its broker instance name. The MQSeries queue name hosting the broker service is identified, giving MQSI a way to communicate with the broker service. An example of this is shown in 12.8.1, “Connecting to the broker” on page 357.

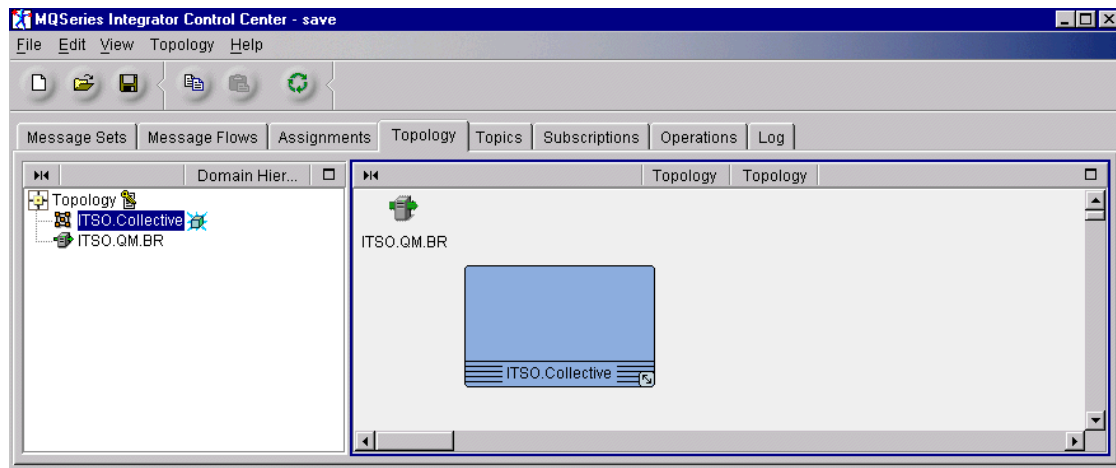


Figure 131. The Topology view

The left-hand pane of the Topology view shows a tree view of the topology of this broker domain. The right-hand pane contains an arrangement of graphical symbols that represent the current topology.

In addition to defining the brokers to use, this tab also allows for the creation of collectives. Once a broker has been defined within the Control Center, this tab can be used to assign it to a collective, for it to be a part of a connected publish/subscribe network. Once a broker has been assigned to a collective, it can be connected to a broker in another collective to ensure that the network of collectives is connected together.

10.2.8.5 Assigning message sets and message flows to a broker

Message flows need to be assigned to the brokers that will execute them. Depending on the type of input the message flow will receive, the broker may also need to have the appropriate message sets assigned to it. Message flows are assigned to execution groups within a broker. A default execution group exists, but you can define more.

The Assignments view allows you to assign message sets to a broker, create execution groups, and assign message flows to an execution group.

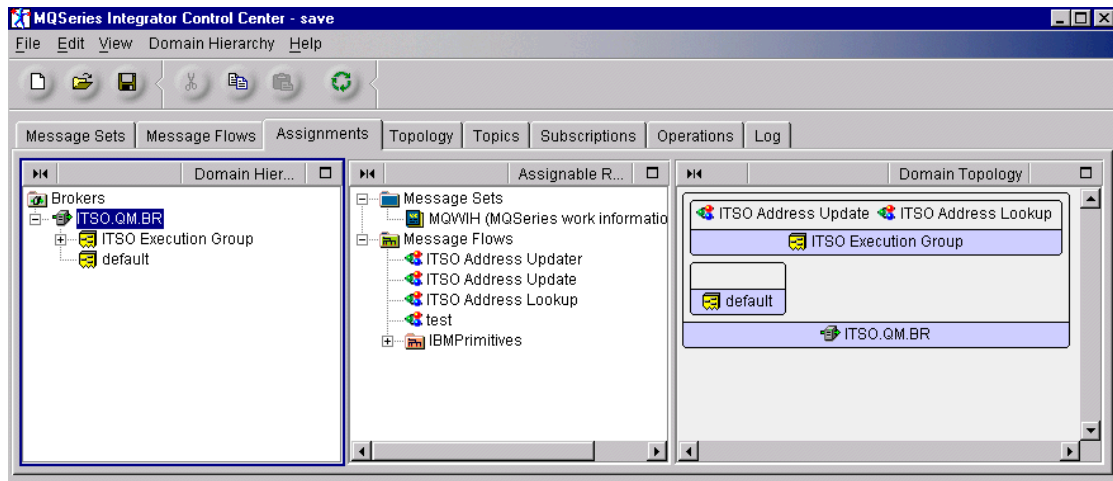


Figure 132. The Assignments view

The left-hand pane of the Assignments view shows the current hierarchy of brokers, execution groups, message flows, and message sets in your workspace. The center pane shows the message sets and message flows in your workspace. The right-hand pane shows the current assignment of execution groups to brokers, of message flows to execution groups, and of message sets to brokers in your workspace.

An example of creating an execution group and assigning a message flow to it can be seen in 12.8, “Using the Control Center to deploy an application” on page 355.

10.2.8.6 Deploying applications

The last step in building and activating an application is deployment. So far, all the activities discussed under each Control Center view have only taken place in the local copy of the configuration. In order for the broker to perform any of the functions defined, they must be deployed to the broker. Data that must be deployed includes assignments, topics data (publish/subscribe), and topology data.

A complete deployment of any type of data deletes all the existing configuration data for that data type from all brokers and creates a new configuration.

A delta deployment deploys only the configuration data that has changed from the currently deployed configuration. This is obviously a much less

drastic action and can save you a lot of time, depending on the size of the data to be deployed and the number of brokers in the domain.

10.2.8.7 Operations

The Operations view provides a basic systems management interface for managing the broker operations. This view will allow an authorized user to see whether brokers, execution groups, and message flows are active or quiesced, and will be able to perform the appropriate operations on these entities.

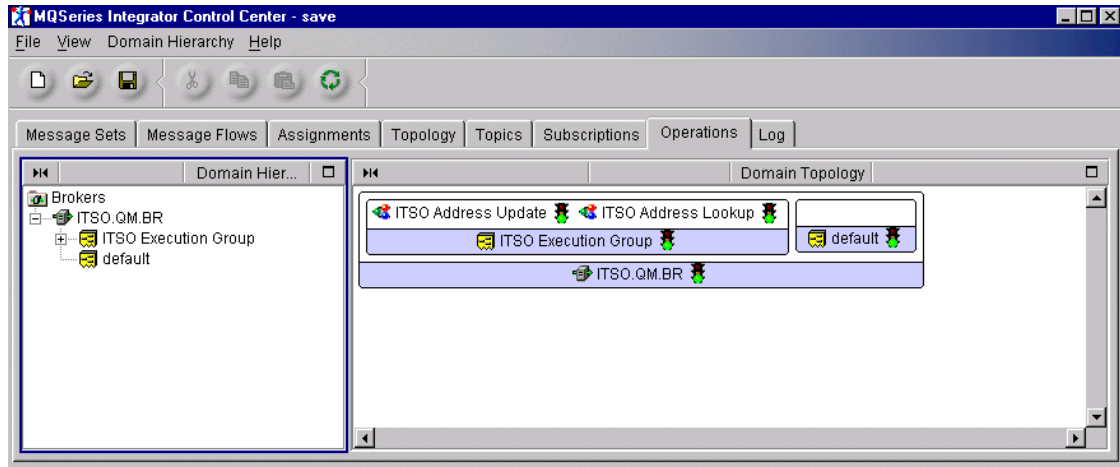


Figure 133. The Operations view

The left-hand pane shows a tree view of the brokers in your broker domain. The execution groups and message sets assigned to a broker are displayed when you expand the broker. The message flows assigned to an execution group are displayed when you expand the execution group.

The right-hand pane contains an arrangement of graphical symbols that represent the current broker domain. Execution groups and message sets appear inside the brokers to which they have been assigned. Message flows appear inside the execution groups to which they have been assigned. The brokers shown in the Operations view are those to which configuration data has been deployed.

The color of the small icons to the right of the resource names indicate the status, either running (green), stopped (red), or unknown (yellow).

In addition to monitoring the status of the MQSI broker domain, you can perform the following actions from the Operations view:

- Start or stop:
 - All message flows in all execution groups assigned to a specified broker
 - All message flows in a specified execution group
 - A single message flow
- Control tracing for:
 - For all message flows in a specified execution group
 - For a single message flow

10.2.9 Resource definition management

The Control Center allows you to export and import the following resource definitions types created for your broker domain:

- Topology definitions
- Topics
- Message flow definitions

Note

Message set definitions can also be exported and imported, but not from the Control Center. You must use the MQSeries Integrator command, `mqsimrmimpexp`, instead.

When you export these definitions, an XML file is generated containing the information retrieved from the configuration repository. You can use definitions exported in this way to populate another configuration repository in another broker domain by using the import function in the Control Center.

10.2.9.1 Exporting resources

When you export a workspace, all resources currently displayed in your current workspace, including topology, message flows, and topics (but excluding message sets), and all resources that they depend on, are exported to an XML file, along with the workspace itself. The export file can then be imported by other Control Center users.

Export does not permit the selection of resource types to export. You can only export a complete workspace.

You export the workspace by selecting **File->Export**.

10.2.9.2 Importing resources

You can import resources from an exported file into the local repository. To import resources:

1. Click **File -> Import**. The import dialog is displayed as shown in Figure 134:

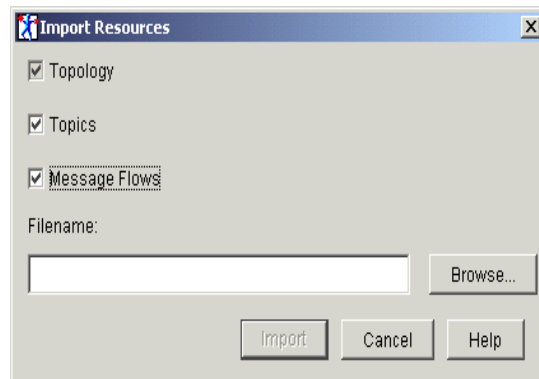


Figure 134. Resource import dialog

2. Specify the fully qualified name of the file to be imported in the Filename field. The import file is examined and you are given a choice of what types of resources to import (see the Import Resources panel inset in Figure 134). You cannot import individual resources. The import action imports all the message flows, all the topics, or all the topology data, depending on the type of resource you select.
3. Select the resource types and click **Import**. The file contents will be imported and your current workspace will be replaced with the workspace in the export file.

When the import action has completed, a report is displayed indicating how many resources have been imported.

10.2.9.3 Message flow versioning utilities

If the application development standards and procedures at your site include a source library system for source control and versioning, then consider downloading and using MQSeries Integrator SupportPac IC01. IC01, called the "MQSeries Integrator V2 - Message flow versioning utilities", is comprised of the following three utilities that enable you to manage your MQSeries message flows in a source library system, such as CMVC:

- **mqsfiltermsgflows** takes an export file (exported from the Control Center) and the name of a message flow, and creates a filtered export file

containing just the named message flow. It does not modify the original export file. This filtered file may then be checked into a library system.

- **mqsicombinmsgflows** takes a list of filtered export files produced by `mqsifiltermsgflows` and combines them into a single export file.
- **mqsdeletemsgflows** looks at the message flows defined in the workspace inside an export file, and deletes them directly from the configuration repository (as long as none of the message flows are locked).

A typical use of these utilities would allow you to save your message flows in a source library system and then apply them as a set to a particular broker domain. For example:

- On your development broker domain:
 - Each time you modify message flows in the configuration repository, export them to an export file using the Control Center.
 - For each message flow in the export file, run `mqsifiltermsgflows` on the export file and check the resulting file into your library system.
- On your production broker domain:
 - Extract the latest version of all message flows from your library system and merge them into a single export file using `mqsicombinmsgflows`.
 - Run `mqsdeletemsgflows` to remove existing message flows from the configuration repository.

A comprehensive list of downloadable MQSeries Integrator SupportPacs, including IC01, is available at:

<http://www.software.ibm.com/ts/mqseries/txppacs/txpm4.html#mqi>

10.2.10 MQSeries Integrator security

An important part of planning your broker domain is considering the security controls that are available and the levels of security you want to implement for those controls.

MQSeries Integrator exploits MQSeries and the operating system facilities to control security for components and tasks:

- Topic-based security.

The MQSeries Integrator User Name Server interacts with the operating system security system to control user and group access to publications and subscriptions.

- Operational control of components.

MQSeries Integrator uses the operating system access control.

- Operational roles used in the Control Center.

MQSeries Integrator uses Windows NT access control. (The Control Center runs on Windows NT only.)

The following sections describe the controls that are available, and how they affect the operation of your broker domain.

10.2.10.1 Security and principals

Security control of MQSeries Integrator components, resources, and tasks depends on the definition of users and groups of users (principals) to the security subsystem of the operating system (the Windows NT User Manager or the UNIX user/group database).

When MQSI is installed it automatically creates five groups in the operating system's security mechanism. Assigning a user to one or more of these groups determines the MQSI tasks they are allowed to perform. The MQSI groups are:

- mqbrkrs
- mqbrasgn
- mqbrdevt
- mqbropts
- mqbrtpic

In a UNIX environment, a user assigned to the MQSI mqbrkrs group and the MQSeries mqm group is authorized to do the following tasks:

- Install/un-install MQSI. A superuser ID (root) is required.
- Create MQSI components using the MQSI create command.
- Start/stop MQSeries components.
- Run MQSI components (the service ID).

In a Windows NT environment:

- Users in the Administrators group can:
 - Install and un-install MQSI
 - Create MQSI components
 - Start/stop MQSI components
- Users in the MQSI mqbrkrs group and MQSeries mqm group can:
 - Run MQSI components (the service ID).

- A user must be assigned to at least one of the MQSI groups other than mqbrkrs to run the Control Center. The particular group (or groups) the user is assigned to maps to a Control Center role, determining the MQSI authority the user possesses. MQSI roles are discussed in 10.2.8, “Control Center operations” on page 278. Table 17 maps this relationship.

Table 17. Principal to role relationship

Role	Group
Message developer	mqbrasgn
Message Assigner	mqbrdev
Operational Controller	mqbrops
subscription admin	mqbrtpic
All	mqbrasgn mqbrdev mqbrops mqbrtpic

10.2.10.2 Using Windows NT security domains

MQSI draws principals from either a Windows NT local account security domain, a Windows NT primary domain, or a Windows NT trusted domain. For more information about Windows NT security domains, refer to the Microsoft Web site at:

<http://www.microsoft.com/ntserver/security/deployment/default.asp>

In particular, review the contents of the Security Deployment Resources Roadmap on this Web page.

10.2.10.3 MQSeries security

MQSI depends on a number of MQSeries resources to operate successfully. You must control access to these resources to ensure that MQSI can access the resources it needs, while limiting access to other users. Some authorizations are granted automatically on your behalf when commands are issued, primarily the authority to put messages on a queue and to retrieve (get) messages from a queue. Others depend on the configuration of your broker domain.

The transmission queues handling the message traffic between the MQSI component queue managers must have put and setall authority granted to the local mqbrkrs group or to the service user ID of the MQSI component.

When you create, assign, and deploy a message flow you must grant the following:

- get authority to each input queue identified in an MQInput node, for the broker's service user ID.
- put authority to each output queue identified in an MQOutput node, or by an MQReply node, for the broker's service user ID.
- get authority to each output queues identified in an MQOutput node or an MQReply node to the user ID under which a receiving or subscribing client application runs.
- put authority to each input queue identified in an MQInput node to the user ID under which a sending or publishing client application runs.

MQSI security is discussed in more detail in *MQSeries Integrator Administration Guide*, SC34-5792.

10.2.10.4 Database security

The Configuration Manager service user ID must be authorized for create and update tasks on the database in which both configuration and message repositories are defined.

Each broker service user ID must be authorized for create and update tasks on the database that contains the broker internal tables. Each broker service user ID must also be authorized for the appropriate access for every database referenced and accessed by a message processing node in any deployed message flow.

Of course, access to the above databases must be controlled and limited to the designated Configuration Manager, broker and User Name Server service IDs.

10.2.10.5 Application security

When you deploy a message flow on one or more brokers, applications can start to feed messages into the message flow by putting messages to the queue that is identified as the input queue. You set up the association between the input node and the queue by setting the queue name as a property of the node.

Similarly, applications access queues to receive messages placed on those queues by MQOutput or Publication nodes, when the message flow has completed processing for those messages.

The user IDs under which applications are executing must therefore be authorized to write to, or read from, the queues used by the message flow the applications are interacting with.

10.2.11 MQSeries Integrator backup and recovery

This section describes the actions needed to recover from errors and restart some or all of the components of your broker domain. It covers the following topics:

- Making sure that messages aren't lost
- Restart scenarios
- Backup
- Recovery scenarios

10.2.11.1 Making sure messages are not lost

It is important to safeguard messages flowing through your broker domain, both application-generated messages and those used internally for inter-component communication. MQSeries provides two techniques that protect against message loss:

- Message persistence

If a message is persistent, MQSeries ensures it is not lost when a failure occurs, by hardening it to disk.

- Syncpoint control

An application can request that a message is processed in an atomic manner in a synchronized unit-of-work (UOW).

For more information about how to use these options, refer to *MQSeries System Administration*, SC33-1873.

Broker internal messages

MQSeries Integrator components use MQSeries messages to communicate events and data between broker processes and subsystems. The broker exploits the available MQSeries to protect against message loss. Therefore, you do not need to take any additional steps to configure MQSeries or MQSeries Integrator to protect against loss of internal messages.

Application messages

If delivery of application messages is critical, you must design application programs and the message flows they use to ensure that messages are not lost.

The default action of a message flow is to respect the persistence of each incoming message. The client program must specify the required message persistence when it puts the message to the input queue of a message flow.

The default action of a message flow is to process incoming messages under syncpoint in a broker-controlled transaction. This means that a message that fails to be processed for any reason is backed out by the broker. Because it was received under syncpoint, the failing message is reinstated on the input queue and can be processed again.

If the error condition persists, the message continues to be passed through the message flow and backed out, causing a processing loop. This is repeated until the value of the MQMD BackoutCount parameter equals or exceeds the value of the backout threshold for the input queue (BOTHRESH attribute). The BackoutCount is incremented automatically by MQSeries every time a message is backed out.

MQSeries Integrator invokes backout processing by attempting to propagate the message as follows:

1. To the failure terminal of the current node.
2. To the queue specified as the input queue's backout requeue name (BOQNAME queue attribute).
3. To the queue manager's dead-letter queue (DLQ).

If none of these queues exist, the message cannot be handled safely without risk of loss. The message cannot be discarded, so the message flow continues to attempt to back out the message. It records the error situation by writing errors to the Windows NT event log, or the AIX syslog.

10.2.11.2 Restart scenarios

This section illustrates the actions you must take to restart the runtime components of MQSeries Integrator and other software on which they are dependent.

Broker

If you need to restart a broker and its environment, do the following (in this order):

1. Stop the broker using the `mqsistop` command.
2. Stop the broker's queue manager using the `endmqm` command.
3. Stop the database manager. Refer to the documentation for your database for instructions on how to complete this task.
4. When everything has stopped, components must be restarted in the following order:

- a. Start the database manager.
- b. Start the broker using the `mqsistart` command. This automatically restarts the queue manager.

The broker does not tolerate abnormal or out-of-sequence termination of the MQSeries queue manager or the database manager. If this occurs, the broker must be stopped using the `mqsistop` command, and all components restarted in the order listed.

If the problem is caused by the queue manager stopping, reissue the MQSeries `endmqm` command specifying the immediate option (`i`) before issuing `mqsistop`.

You do not have to restart the broker execution group processes if they terminate abnormally, because the broker does this automatically.

Configuration Manager

The Configuration Manager operates independently of the brokers, and can be stopped and restarted without affecting the operation of other MQSeries Integrator components in the broker domain. If you need to restart the Configuration Manager and its environment, do the following:

1. Stop the Configuration Manager using the `mqsistop` command.
2. Stop the MQSeries queue manager using the `endmqm` command.
3. Stop DB2.

It is recommended that you complete these tasks in the order shown, but the Configuration Manager tolerates the queue manager and DB2 stopping first.

When everything has been stopped, do the following actions in this order:

1. Restart DB2.
2. Restart the Configuration Manager using the `mqsistart` command. This automatically restarts the queue manager.

If you restart the Configuration Manager first, it will automatically retry its initialization until DB2 is started.

The Configuration Manager also tolerates abnormal termination of the MQSeries queue manager and DB2. If the Configuration Manager detects that either has terminated abnormally, it restarts automatically. If either DB2 or MQSeries, or both, are not up and running, initialization is automatically retried every 30 seconds until it is successful.

User Name Server

The User Name Server operates independently of the brokers, and can be stopped and restarted without affecting the operation of other MQSeries Integrator components in the broker domain. If you need to restart the User Name Server and its environment, do the following:

1. Stop the User Name Server using the `mqsistop` command.
2. Stop the MQSeries queue manager using the `endmqm` command.

It is recommended that you complete these tasks in the order shown, but the User Name Server tolerates the queue manager stopping first.

When everything has been stopped, restart the User Name Server using the `mqsistart` command. This automatically restarts the queue manager.

The User Name Server also tolerates abnormal termination of the MQSeries queue manager. If this occurs, the User Name Server restarts automatically. If MQSeries is not up and running, initialization is retried every 30 seconds until it is successful.

10.2.11.3 Backing up the MQSeries Integrator components

Brokers and the Configuration Manager rely on a database manager to maintain and control all their configuration data. Brokers, the Configuration Manager, and the User Name Server rely on MQSeries to transport and guarantee messages between components. You must establish a backup process that includes these sources of information to preserve the integrity and consistency of your broker domain.

- Back up MQSI database tables frequently and on a regular basis to prevent loss of configuration data if damage occurs. All configuration repository tables, all message repository tables, and the following broker database tables should be included in the backup:
 - BSUBSCRIPTIONS
 - BCLIENTUSER
 - BUSERCONTEXT
 - BRETAINEDPUBS
 - BPUBLISHERS
 - BMQPSTOPOLOGY

The remaining broker database tables will be created when the broker domain is restarted and the Configuration Manager redeploys the configuration.

These backups should be coordinated so a consistent image is available for recovery.

- Back up any user-defined database tables used by the message flows.
- Back up the MQSeries configuration data. See *MQSeries System Administration*, SC33-1873 for further details.
- Consider backing up files that are created by users of the Control Center. When a Control Center user saves a workspace, the workspace XML document, any newly created objects, and any checked-out objects are saved to the local file system. Instruct the users to back up this local file system.

10.2.11.4 Recovery scenarios

You can recover the runtime components of MQSeries Integrator if the environment becomes damaged (for example, if MQSeries objects used by the broker are damaged), or if database contents are damaged.

Broker

If the environment for a particular broker becomes damaged, or if one or more of the broker database tables are unusable, you must perform the following sequence of operations to recover it:

1. Ensure that no Control Center users are deploying to brokers. You must wait until these actions have completed.
2. Stop the broker using the `mqsistop` command.
3. Stop the broker's queue manager using the `endmqm` command.
4. If there is no damage to any of the broker database tables listed in 10.2.11.3, "Backing up the MQSeries Integrator components" on page 297, take a backup of these tables. These tables are interdependent and must all be in a consistent state when restored. You cannot back up or restore individual tables.
5. Delete the broker using the `mqsdeletebroker` command or the Command Assistant.
6. Recreate the broker using the `mqsicreatebroker` command or the Command Assistant.
7. Restore the broker database tables saved in step 4, or from a previous backup if necessary.
8. Start the broker using the `mqsistart` command.
9. Restart the Control Center if it is not currently running. Select the Topology view.
10. Redeploy the domain configuration by selecting **File->Deploy->Complete configuration (all types)->Normal** to ensure that the configuration across the broker domain is consistent.

Configuration Manager

If the Configuration Manager environment is damaged, or one or more of the database tables are corrupted, you must perform the following sequence of operations to recover it:

1. Ensure that all Control Center sessions are stopped.
2. Stop the Configuration Manager using the `mqsistop` command.
3. Stop the Configuration Manager's queue manager using the `endmqm` command.
4. Delete the Configuration Manager using the `mqsdeleteconfigmgr` command or the Command Assistant:
 - a. If you are recovering the Configuration Manager because one or more of the configuration repository or message repository tables is damaged, you must include the `.n` and `.m` flags on the `mqsdeleteconfigmgr` command.
 - b. If the database tables are undamaged, you must omit the `.n` and `.m` flags. This preserves your configuration data in both repositories.
5. If you are recovering the Configuration Manager because one or more of the configuration repository or message repository tables is damaged, you must restore both repositories from a previously successful backup version. The data in the two repositories is interdependent, and you must restore the entire contents of both. You cannot restore individual tables.
6. Recreate the Configuration Manager using the `mqsicreateconfigmgr` command or the Command Assistant.
7. Start the Configuration Manager using the `mqsistart` command.
8. Start the Control Center, and select the Topology view.
9. If you have completed Steps 4a and 5 you must also redeploy the domain configuration by selecting **File->Deploy->Complete configuration (all types)->Normal** to ensure that the configuration across the broker domain is consistent.

User Name Server

If the User Name Server environment becomes damaged, you must perform the following sequence of operations to recover it:

1. Stop the User Name Server using the `mqsistop` command.
2. Stop the User Name Server's queue manager using the `endmqm` command.
3. Delete the User Name Server using the `mqsdeleteusernameserver` command or the Command Assistant.
4. Recreate the User Name Server using the `mqsicreateusernameserver` command or the Command Assistant.
5. Start the User Name Server using the `mqsistart` command.

10.2.12 MQSeries Integrator monitoring

MQSeries Integrator provides facilities that assist in centralized system management. These facilities support the following tasks:

- Monitoring of the status and activity of MQSeries Integrator system components (brokers, the Configuration Manager and the User Name Server). For example, reports are generated whenever a broker starts or stops.
- Monitoring of the status and activity of execution groups.
- Monitoring of the status and activity of message flows.

MQSeries Integrator generates reports (similar in function to MQSeries events in 10.1.6.1, “Instrumentation events” on page 258) to provide information about the operation and status of the broker domain. The nature and format of these report messages, in XML, is described in detail in Appendix A, “Event reporting” of the *MQSeries Integrator Administration Guide*, SC34-5792. The report messages are published with specific associated topics, enabling a center of competence anywhere in the MQSeries network to support the broker domain by simply subscribing to these topics.

There are several vendors that provide MQSI and MQSeries monitors. If your organization has already invested in an MQSeries monitor, check to see if that vendor also provides an MQSI monitor. When selecting an MQSI monitor, make sure it is a good fit within your existing enterprise-wide middleware and system management framework.

Chapter 11. Introduction to the working example

A simple application was developed for this project and implemented on the base and variation 1 topologies for AIX and Windows NT. The design of the MQSI portion of this application is discussed in Chapter 9, “Developing the MQSI application” on page 183. In this part of the book, we will be describing the methods used to implement the application using WebSphere Advanced Edition, MQSeries, and MQSI.

11.1 Sample application

The sample WebBank application described in Chapter 8, “Application development guidelines” on page 141 allows a banking customer to retrieve and update their customer profile. This application demonstrates how MQSeries Integrator can be configured to handle multiple updates to back-end system databases, while also maintaining a local cache of data for reasons of performance and integrity.

We have two versions of the application. In one, Java servlets and command beans use the MQSeries classes for Java to put messages onto and get messages from MQSeries queues. In the second version, the application uses JMS for MQSeries.

11.1.1 Application flow

The first thing a user will see when accessing the application will be a window asking for a user ID and password.



Figure 135. Initial login window

Once logged in, the user is presented with a screen showing several options. The only option we have implemented in this book is option 2a, which allows you to view your profile and to change it.

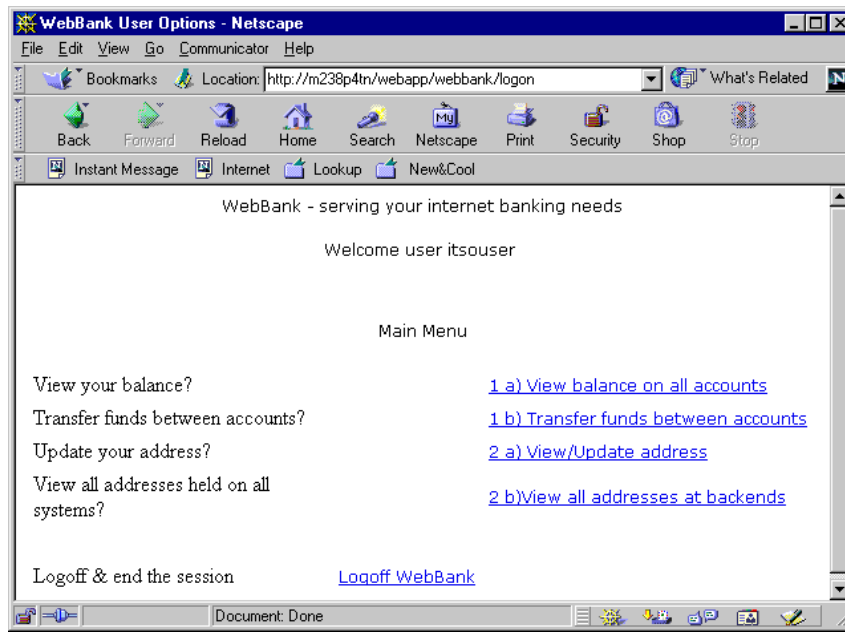


Figure 136. Welcome window

When this option is chosen, the profile information is taken from a database local to the broker used as a cache database, ITSOCUST. If the data has not been previously cached, it is taken from one of the back-end account databases. Updates to the address are sent to all three databases. This illustrates the use of a cache database, and translation to different formats since the two account back-end databases (savings and checking) store the profile in different formats.

11.2 Runtime topologies

In our lab we set up an environment that could be used to show the basic runtime topology and its variation for application topology 5.

The basic runtime topology has the WebSphere Application Server in the DMZ. The servlet in WebSphere uses MQSI as a router. MQSI is in the internal network.

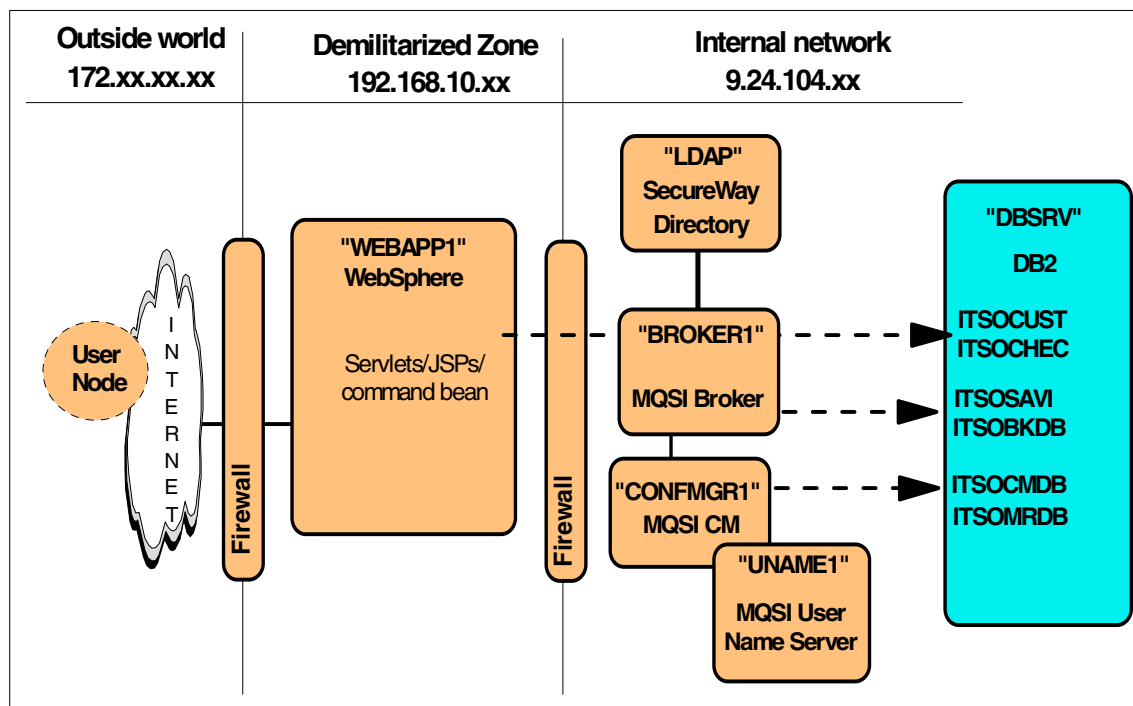


Figure 137. Basic topology

The variation moves WebSphere into the internal network for added security and uses a redirector function in the DMZ to route requests from the Web server to WebSphere.

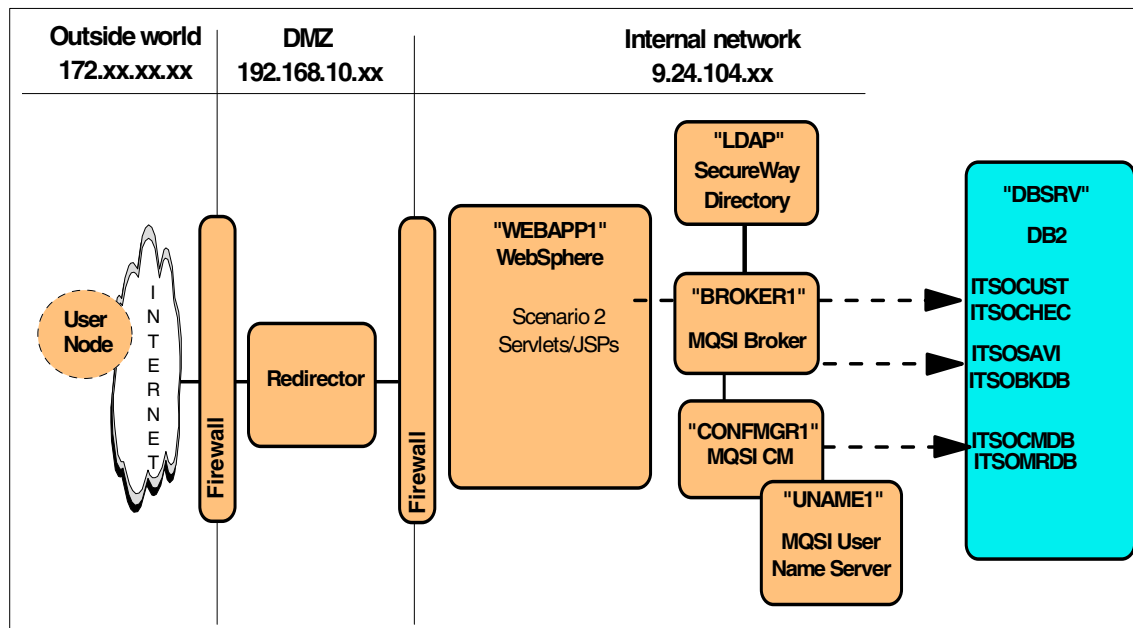


Figure 138. Variation 1

11.2.1 Product documentation, software, and support

Product documentation can be found at the following Web sites:

- Documentation for MQSeries and MQSI can be found at:

<http://www.ibm.com/software/ts/mqseries/library/>

- Documentation for WebSphere can be found at:

<http://www.ibm.com/software/webservers/library.html>

- Documentation for DB2 can be found at:

<http://www.ibm.com/software/data/pubs/>

Product maintenance and upgrades can be found at the following Web sites:

- Software fixes and upgrades for WebSphere Application Server can be found at:

<http://www.ibm.com/software/webservers/appserv/support.html>

- Software fixes and upgrades for MQSeries can be downloaded from:

<http://www.software.ibm.com/ts/mqseries/support/summary/wnt.html>

- The MQSeries support pacs can be downloaded from:

<http://www.ibm.com/software/ts/mqseries/txppacs/>

The MQSeries SupportPac MA88 providing the MQSeries classes for Java and MQSeries classes for Java Message Service (JMS) can be found under category 3 product extensions.

- Microsoft patches can be downloaded from:

<http://support.microsoft.com/directory/>

- MQSeries Integrator patches can be found at:

<http://www.ibm.com/software/ts/mqseries/support/summary/mqsi.html>

11.3 Web application server

The system labeled WEBAPP1 represents the Web application server in the basic topology (Figure 137 on page 306) and the application server in variation 1 (Figure 138 on page 307). WEBAPP1 will provide the interface to the user and will host servlets, JSPs, and EJBs.

Regardless of whether WEBAPP1 is in the DMZ, as in the basic topology, or in the internal network as shown in variation 1, the install is basically the same. The exception will be in the Web server plug-in. You will need the plug-in in the basic configuration, since the Web server and WebSphere are on the same machine. In variation 1, WEBAPP1 is acting only as an application server and will not require the plug-in. In this case, the plug-in and Web server both reside on the redirector node.

In this instance our application is acting as an MQSeries client, meaning the application places messages directly on MQSeries queues on remote systems. This may not always be preferable. You could install MQSeries on the WebSphere machine and have the application place the messages on a local queue. This would add the full benefits of MQSeries to the WebSphere machine (load balancing, asynchronous messaging, etc.).

11.3.1 Web application server running on Windows NT

The following table summarizes the software used to implement our application on the Web application server running on Windows NT.

Table 18. Web application server requirements for Windows NT

Product Installed	Instructions
Windows NT 4.0 + SP6a	
DB2 6.1 + Fixpak 4	Included with the WebSphere Advanced install. DB2 is needed for the WebSphere administrative repository. The database can be local to WebSphere or remote. WebSphere 3.5 also includes the option to use install InstantDB instead of the full DB2, which gives you just enough function for WebSphere to operate.
JDK 1.2.2	Included with the WebSphere Advanced 3.5 install.
IBM HTTP Server 1.3.12	Included with the WebSphere Advanced 3.5 install.
WebSphere Advanced 3.5	Choose full installation for the basic topology. For variation 1 choose Custom Installation and select: <ul style="list-style-type: none">- Application and Administrative Server- Administrator's Console- Samples- IBM JDK 1.2.2- Configure default server and Web application
MQSeries SupportPac MA88	The application will need these classes. They can be copied in with the application or installed as a package.
Banking application	See Chapter 13, "WebSphere Application Server setup" on page 371.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements. Note: For variation 1 you will also need to install build a Web server redirector.	

11.3.2 Web application server running on AIX

The following table summarizes the software used to implement our application on the Web application server running on AIX.

Table 19. Web application server requirements for AIX

Product Installed	Instructions
AIX 4.3.3 + maintenance level 4	
DB2 6.1	DB2 is needed for the WebSphere administrative repository. The database can be local to WebSphere or remote.
IBM HTTP Server 1.3.12	Included with the WebSphere Advanced 3.5 install.
WebSphere Advanced 3.5	From the Install Options window, select Custom Installation and select: <ul style="list-style-type: none">- Production Application Server- Administrator's Console- Documentation- Samples- Configure admin domain with the default applications- The appropriate Web server plug-in.
MQSeries SupportPac MA88	The application will need these classes. They can be copied in with the application or installed as a package.
Banking application	See Chapter 13, "WebSphere Application Server setup" on page 371.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.4 MQSI broker

The system labeled BROKER1 in Figure 137 on page 306 and Figure 138 on page 307 is an MQSI broker running on an MQSeries cluster. It has access to a DB2 database used for caching updated customer profiles and for looking up bank account access information based on the user ID entered by the user.

11.4.1 Running the broker on Windows NT

The following table summarizes the software installed for the MQSI broker running on Windows NT.

Table 20. MQSI broker requirements for Windows NT

Product Installed	Instructions
Windows NT 4.0 + SP6a	
DB2 6.1 + Fixpak 4	Included on the MQSI CD and can be installed as part of the MQSI install process. DB2 is needed for the broker database. The database can be local or remote.
MQSeries 5.1 + CSD05	Choose Typical Install The MQSeries install will call for the following prereqs: -Active Directory Service (ADSI) V2.0 -Microsoft Management Console 1.1 Both are included on the MQSeries CD.
MQSeries Integrator 2.0.1 + IC27806	Choose Custom Install : - Broker Only - Choose appropriate database type for Neon support For configuration setup see: 12.3, "MQSI database setup" on page 322 12.7, "MQSI broker setup" on page 345
Banking application	See Chapter 12, "MQSeries and MQSI implementation" on page 319.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.4.2 Running the broker on AIX

The following table summarizes the software we installed on the AIX system to run the MQSI broker.

Table 21. MQSI broker requirements for AIX

Product Installed	Instructions
AIX V4.3.3 + maintenance level 4	
DB2 6.1 + Fixpak 4	Included with the MQSI. DB2 is needed for the broker database. The database can be local or remote.

Product Installed	Instructions
MQSeries 5.1 + CSD05	Minimum level is CSD04. In our lab we installed CSD05 (U69691) downloaded from: http://www.software.ibm.com/ts/mqseries/support/summary/aix.html .
JDK 1.1.8 + minimum service level PTF8	Included with AIX 4.3.3
MQSeries Integrator 2.0.1 + IY12651	
Banking application	See Chapter 12, "MQSeries and MQSI implementation" on page 319.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.4.3 MQSI service

There are two APARs open for the Windows NT and UNIX environment that should be installed on the MQSI broker when available. They fix a problem that occurs when doing a full deployment of an application to the broker. For Windows NT, the APAR number is IC27806. For AIX, the APAR number is IY12651. You can circumvent the problem by always performing a delta deployment from the Control Center.

11.5 MQSI Configuration Manager

The MQSI Configuration Manager only runs on Windows NT. The following table summarizes the software used for our application on the Configuration Manager running on Windows NT.

Table 22. MQSI Configuration Manager requirements for Windows NT

Product Installed	Instructions
Windows NT 4.0 + SP6a	SP5 is the minimum required level.
DB2 6.1 + Fixpak 4	Included with the MQSI. DB2 is needed for the broker database. The database can be local or remote.

Product Installed	Instructions
MQSeries 5.1 + CSD05	CSD04 is the minimum required level. Choose Typical Install . The MQSeries install will call for the following prereqs: -Active Directory Service (ADSI) V2.0 -Microsoft Management Console 1.1 Both are included on the MQSeries CD.
MQSeries Integrator 2.0.1	Custom install: - Configuration Manager For setup information see: 12.3, "MQSI database setup" on page 322 12.5, "MQSI Configuration Manager setup" on page 330
Banking application	See Chapter 12, "MQSeries and MQSI implementation" on page 319.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.6 User Name Server

The system labeled UNAMEx represents a User Name Server. It is used for publish/subscribe and in this case as a second cluster repository. It is not required for this sample, but we included it in the Windows NT scenario because it is likely that a full MQSI installation will use publish/subscribe and thus will need a User Name Server.

11.6.1 Running the User Name Server on Windows NT

The following table summarizes the software used for the MQSI User Name Server running on Windows NT.

Table 23. MQSI User Name Server requirements for Windows NT

Product Installed	Instructions
Windows NT 4.0 + SP6a	SP5 is the minimum required level.
** The Control Center can be installed on any Windows NT system.	
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

Product Installed	Instructions
MQSeries 5.1 + CSD05	CSD04 is the minimum required level. Choose Typical Install . The MQSeries install will call for the following prereqs: -Active Directory Service (ADSI) V2.0 -Microsoft Management Console 1.1 Both are included on the MQSeries CD.
MQ base Java from MQSeries SupportPac MA88	Needed for the Control Center
MQSeries Integrator 2.0.1	Custom install: - User Name Server - Control Center ** For setup information see: 12.4, "MQSI User Name Server setup" on page 324
** The Control Center can be installed on any Windows NT system.	
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.7 Database server

The DB2 server logically represents one or more database servers. In our example we use one DB2 server for all DB2 requirements, including the databases required by MQSI and the application.

11.7.1 Running DB2 on Windows NT

The following table summarizes the software installed for the DB2 server running on Windows NT.

Table 24. DB2 Server requirements for Windows NT

Product Installed	Instructions
Windows NT 4.0 + SP6a	SP5 is the minimum required level.
DB2 6.1 + Fixpak 4	Included with MQSI.
Banking application and MQSI databases	See 12.3, "MQSI database setup" on page 322 and 12.9.1, "Create the application databases" on page 366.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.7.2 Running DB2 on AIX

The following table summarizes the software installed for the DB2 server running on AIX.

Table 25. DB2 Server requirements for AIX

Product Installed	Instructions
AIX 4.3.3 + maintenance level 4	
DB2 6.1 + Fixpak 4	Included with MQSI.
Banking application and MQSI databases	See 12.3, "MQSI database setup" on page 322 and 12.9.1, "Create the application databases" on page 366.
Note: The software shown here reflects the software levels we chose to use, sometimes exceeding the minimum requirements. Check the product installation manuals for the minimum software requirements.	

11.8 Planning user IDs

When planning the user IDs for this lab setup, we decided to keep things simple by using a small number of user IDs that had a wide range of authority. This is an acceptable approach in a test environment; however, in a production system, user IDs and authorities should be carefully thought out. More information on the required IDs and authorities can be found in the *MQSeries Integrator Administration Guide*, SC34-5792.

There are several roles in this pattern that need to be identified and a user ID assigned:

- System administrator user ID. This will be used primarily to install the products and to set up system security.
- MQM administrator user ID. This will be used to administer the MQSeries environment.
- DB2 administrative user ID. This will be used to define the repository and application databases.
- MQSI service ID. This will be used to run the MQSI components.
- MQSI developer IDs. These are the people who will perform different development roles in the MQSI Control Center.
- WebSphere service ID. The WebSphere administrative server and application servers will run under this ID. It will be used by the operating system to determine access to resources such as files and sockets. If you

are using the operating system registry as the authentication mechanism for checking the identity, then:

- On UNIX platforms, the account must have administrative privileges.
- On Windows NT, the account must be a member of the Administrators group and must have the rights to "Log on as a service" and to "Act as part of the operating system."

Do not use an account whose name matches the name of your machine or Windows domain. The WebSphere administrative server will not work in such a case.

If you are using an LDAP directory service for authentication, then the process identity does not need any special privileges.

11.8.1 User IDs for the Windows NT mapping

The first ID we defined is *itsouser*, which will play multiple roles in our environment, including all administrator and service roles with the exception of DB2. This ID can either be defined on each system or be known to the Windows NT security domain. This ID is a member of the following groups:

- Administrators:
 - This allows the user to install software, create/delete/change/start the MQSI components (broker, etc.).
- mqm:
 - Allows the user to administer MQSeries queues, issue MQSeries control commands, and use MQSeries Explorer (being in the administrator group will accomplish this too). For remote queue administration, itsouser must also be authorized on the remote systems.
- mqsibkrks:
 - Allows the user to run the components (act as the service ID). Membership in both mqsibkrks and mqm is required to act as the service ID for the Configuration Manager, and if fastpath is set on, the broker.
 - List and trace components
- mqbrasgn, mqbrdevt, mqbrops, mqbrtpic:
 - Allows the user to run the Control Center. Different roles require specific group membership. See 10.2.10, "MQSeries Integrator security" on page 290 and 10.2.8, "Control Center operations" on page 278 for more information.

The itsouser ID is also granted the required access to the MQSI administrative databases and application databases.

The second user ID we identified is *db2admin*, which will act as the DB2 administrator. This ID is created during DB2 installation. It is a member of the *Administrators* group.

A third user ID, *MUSR_MQADMIN* is automatically created and put in the *mqm* group during MQSeries installation. This requires no action on your part.

11.8.2 User IDs for the AIX mapping

The first ID we defined in our AIX environment is *mqsi*. This ID will be created on all the systems and will play multiple roles in our environment, including MQSeries, and MQSI administrator. On the DB2 system, it is granted access to the MQSI and user databases.

In the AIX broker, we have the following groups:

- *mqsi*brkrs:
 - Allows the *mqsi* user ID to run the components (act as the service ID) and act as the MQSI administrator.
 - List and trace components
- *mqm*:
 - Allows the *mqsi* user ID to administer MQSeries queues and issue MQSeries control commands. For remote queue administration *itsouser* must also be authorized on the remote systems.

Note: For convenience, we added root to *mqsi*brkrs.

Two IDs are automatically created during product installation. During DB2 installation, a user ID, *db2inst1* in our case, is created as the DB2 administrator. The user ID *mqm* in group *mqm* is automatically created during MQSeries installation.

The MQSI Configuration Manager and Control Center both run on Windows NT, so the *mqsi* user ID must also be defined to the Windows NT domain.

An overview of the user IDs in the AIX broker is shown in Table 26.

Table 26. MQSI broker user IDs and groups

Users	Groups	
	mqm	mqbrkrs
root	x	x
mqm	x	
mqsi	x	x

Chapter 12. MQSeries and MQSI implementation

To verify that the sample application would work in the recommended runtime topologies, we set up four environments, using each topology (basic and variation 1) in both a Windows NT and an AIX environment. This chapter documents the steps taken to set up the MQSeries and MQSI systems. The setup is the same for each topology since the MQSeries/MQSI systems always reside in the internal network. Once you get past the system considerations involving installation, the setup is basically the same for both operating systems used. Therefore, we will cover the setup steps once, pointing out any differences between Windows NT and AIX.

12.1 Lab environment

In our test lab we implemented the following:

- A Configuration Manager on Windows NT
- A User Name Server on Windows NT (optional - needed for publish/subscribe)
- Brokers on Windows NT and AIX
- Database servers on Windows NT and AIX

Our MQSI networks were built using a cluster of MQSeries queue managers.

12.1.1 Windows NT test configuration

In the Windows NT test environment, we chose to include a User Name Server, even though we were not using publish/subscribe. Our thinking is that publish/subscribe is an important feature for many MQSI installations and readers would benefit from seeing the User Name Server included. We will not, however, cover the User Name Server implementation.

In this scenario, the MQSeries cluster repository queue managers will be located on the same machines hosting the MQSI User Name Server and the Configuration Manager. One queue manager on each machine will support both the cluster repository and the MQSI administrative functions. To support the broker service, a third queue manager is created as a member of the cluster, and is configured to refer to the User Name Server queue manager for its repository information.

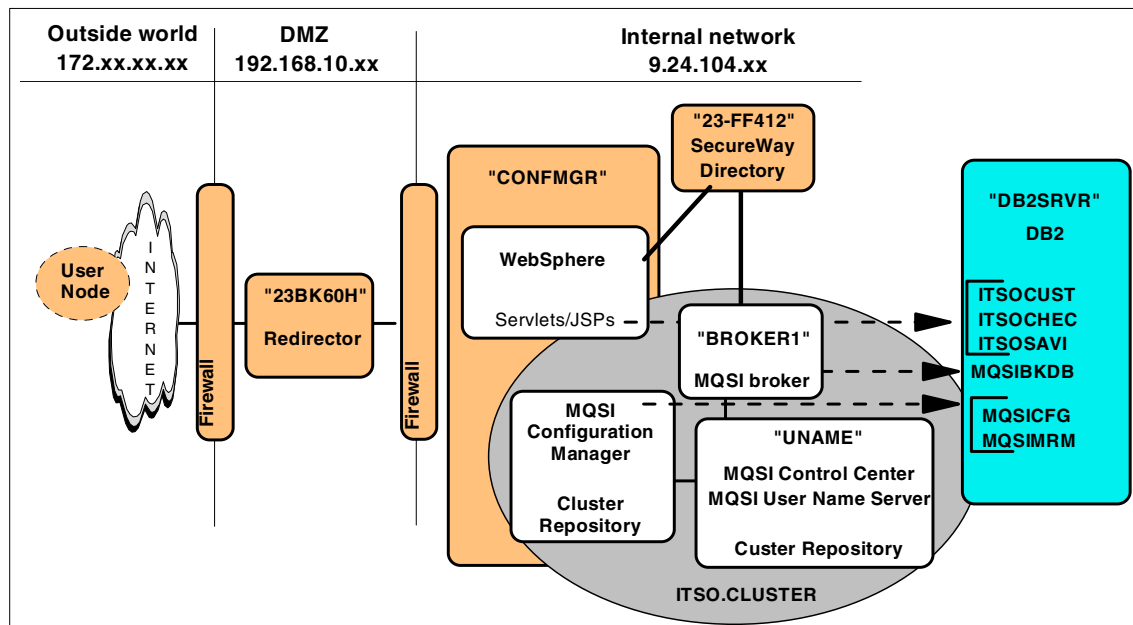


Figure 139. Variation 1 using Windows NT

We will use topology variation 1 to illustrate our environment for Windows NT.

12.1.2 AIX test configuration

Our configuration for the AIX environment is slightly different in that we did not choose to install a User Name Server. The MQSeries cluster repository will be held by the queue managers that will also be performing the MQSI Configuration Manager and broker duties.

Not all of the MQSI components are available on AIX. The MQSI Configuration Manager and Control Center must run in a Windows NT environment. See Table 27 for a summary of the installation options for AIX.

Table 27. Summary of installation options

Product	Component	System Install on
MQSI For AIX V2.0.1	Configuration Manager	Windows NT only
	Control Center	Windows NT only
	Broker	AIX only
	User Name Server	AIX or Windows NT
	SDK	AIX only
	Windows NT Documentation	Windows NT only
	UNIX Documentation	Copy from mqsi_aix_documentation on CD

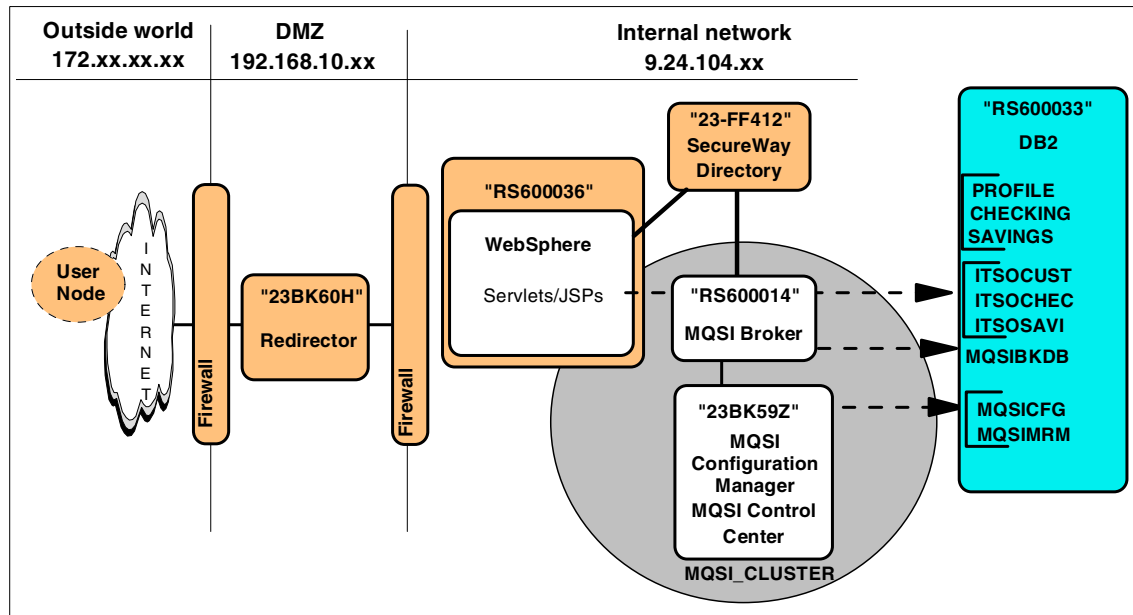


Figure 140. AIX Variation 1

12.1.3 MQSeries and MQSI configuration methods

There are several ways of actually configuring the machines for MQSeries and MQSI. To configure MQSeries components (queue managers, channels, etc) and to control them (start/stop) you can use several methods.

- MQSeries Explorer
- Control commands
- MQSC commands

MQSI provides a Control Center for configuration and operation and also has line commands available. For a discussion on these methods see 10.1.1, “MQSeries administration interfaces” on page 240.

In this chapter we will use a mixture. In addition, we have created scripts that can be used to set up the environment multiple times. See Appendix E, “Using the additional material” on page 409 for information on how to obtain these scripts.

12.2 Defining user IDs

Before starting the configuration of your MQSeries and MQSI network, it is important to choose the user IDs that will be used to fulfill each role. Some of these user IDs will need to be defined before installation. Others will not need to be in place until configuring each product.

Our decisions on which user IDs to use and the roles they will play are discussed in 11.8, “Planning user IDs” on page 315.

12.2.1 MQSI:

The user IDs needed for MQSI will need to be defined on each machine before creating the MQSI components. The MQSI groups will be defined automatically during installation.

12.2.2 DB2 server:

The DB2 administrator ID is chosen and defined at DB2 install time. Any users you grant database authority to must be known to the DB2 server.

12.3 MQSI database setup

The MQSI Configuration Manager and broker services use databases for administration. The options for defining these databases and combining them are discussed in 10.2.6, “MQSI databases” on page 273. In our scenario we

chose to create a distinct DB2 database for each component, message repository, configuration repository, and broker persistent store.

These databases can be local to the MQSI component or remote. We chose to put all of these databases on a DB2 server and define them to DB2 clients on each MQSI system. There are performance implications to consider when deciding where to locate the MQSI databases. In our test lab, performance was not an issue, but in a real network, performance is one of the most important issues. See 7.2.3, “Placement of MQSI databases” on page 113 for more information on this.

The following privileges need to be granted for the user ID that the MQSI service will run under to each database:

- connect
- createtab
- bindadd
- create_not_fenced

DB2 commands can be used to create the databases and grant the required authority.

Windows NT DB2 server

DB2 commands can be executed from the DB2 command line processor window by selecting:

Start->DB2 for Windows NT->Command Line Processor

AIX DB2 server

Log on to the DB2 server machine with the DB2 instance user ID (in our case db2inst1), or switch to the DB2 instance user ID (`su - db2inst1`). Enter:

DB2

You will now be in command line mode and can enter DB2 commands.

The DB2 commands shown in Figure 141 should be executed on the DB2 server for each MQSI component database.

```
create db dbname
connect to dbname user db2admin_id using db2admin_pw
grant connect, createtab, bindadd, create_not_fenced on database to user mqsiuser
connect reset
```

Figure 141. DB2 commands to create the MQSI administrative databases

In these commands:

- *dbname* will be the database name chosen for each database.
- *db2admin_id* and *db2admin_pw* are the DB2 administrative user ID and password authorized to create databases on the DB2 server and to grant authorities. This user ID will have DBADM authority to the databases.
- *mqsiuser* is the user ID that the relevant MQSI service will run under. You will see this in later steps when we create the Configuration Manager service and the broker service.

The database names chosen for our scenario can be seen in Figure 139 on page 320 and Figure 140 on page 321. For example, to create the broker database for the Windows NT scenario shown in Figure 139, we would enter the following commands:

```
create db mqsibkdb
connect to mqsibkdb user db2admin using db2admin
grant connect, createtab, bindadd, create_not_fenced on database to user itsouser
connect reset
```

Figure 142. Create the broker database for the Windows NT basic topology

12.4 MQSI User Name Server setup

The next node to set up is the MQSI User Name Server. In our scenario we only used this in the Windows NT environment. For our purposes the User Name Server machine will be used as a cluster repository. It will not participate in the application.

At the end of the User Name Server setup procedures, you will have the MQSeries and MQSI infrastructure shown in white in Figure 143.

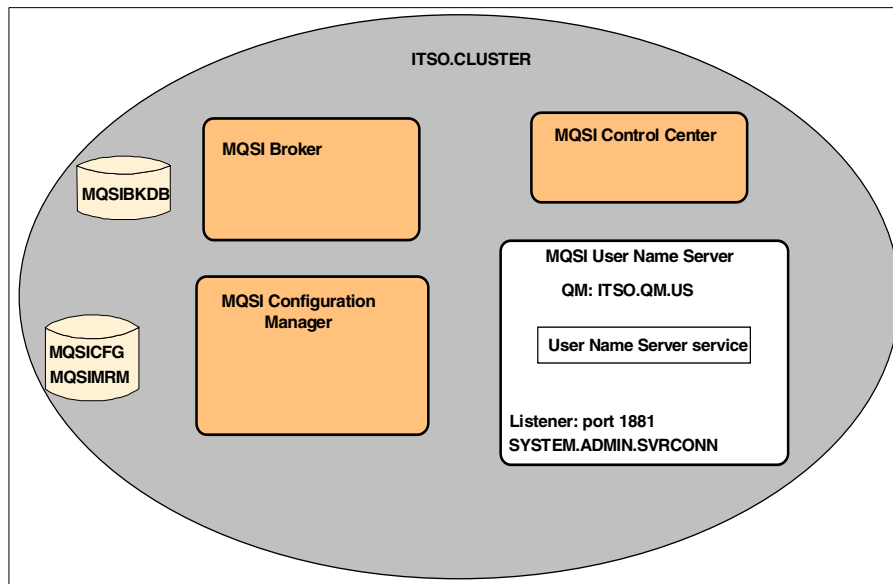


Figure 143. User Name Server

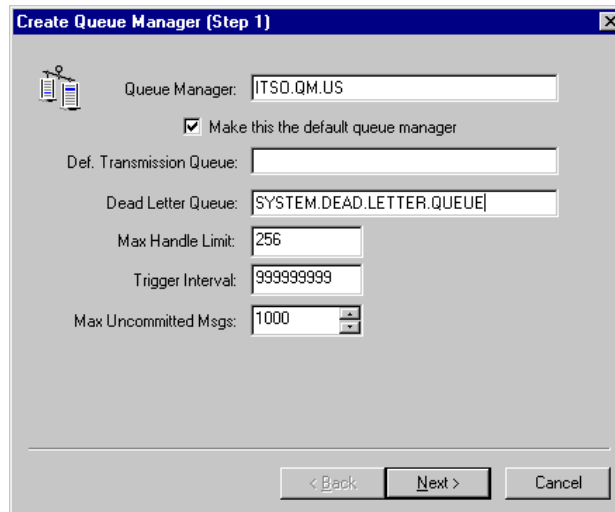
The first step in setting up the User Name Server is to set up the MQSeries infrastructure.

12.4.1 Create the queue manager

The first step involves using the MQSeries Explorer to define the MQSeries infrastructure needed on this system. This involves creating a default queue manager, which we called ITSO.QM.US.

The following steps were used to define the MQSeries definitions required for the MQSI User Name Server.

1. Click **Start->Programs->IBM MQSeries->MQSeries Explorer**.
2. Right-click **Queue Managers** and select **New->Queue Manager**.



Create Queue Manager (Step 1)

Queue Manager:

☒ Make this the default queue manager

Def. Transmission Queue:

Dead Letter Queue:

Max Handle Limit:

Trigger Interval:

Max Uncommitted Msgs:

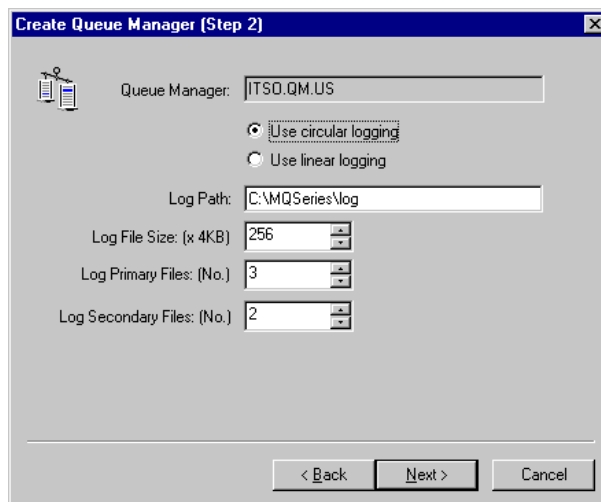
< Back Next > Cancel

Figure 144. Defining a queue manager - window 1

3. Enter the following in the Create Queue Manager window:

- ITSO.QM.US (queue manager name is case sensitive)
- Check **Make this the default queue manager**
- SYSTEM.DEAD.LETTER.QUEUE for the DLQ name

Click **Next** to continue.



Create Queue Manager (Step 2)

Queue Manager:

☒ Use circular logging
☐ Use linear logging

Log Path:

Log File Size: (x 4KB)

Log Primary Files: (No.)

Log Secondary Files: (No.)

< Back Next > Cancel

Figure 145. Defining a queue manager - window 2

4. Accept the defaults for logging and click **Next** to continue.

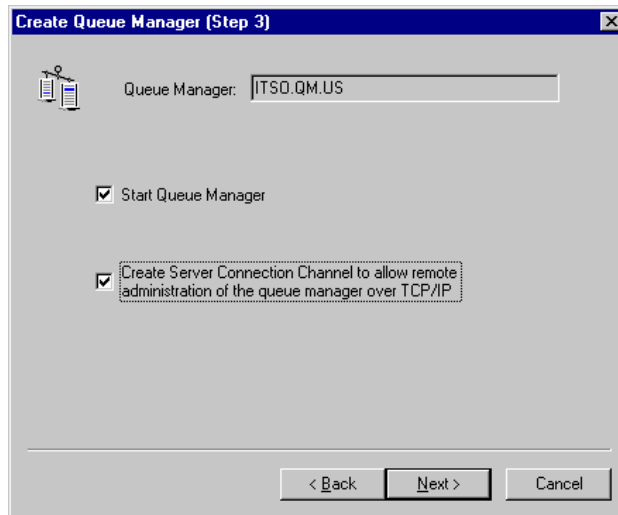


Figure 146. Defining a queue manager - window 3

5. Make sure that **Start Queue Manager** and **Create Server Connection Channel** are both checked. Click **Next** to continue.

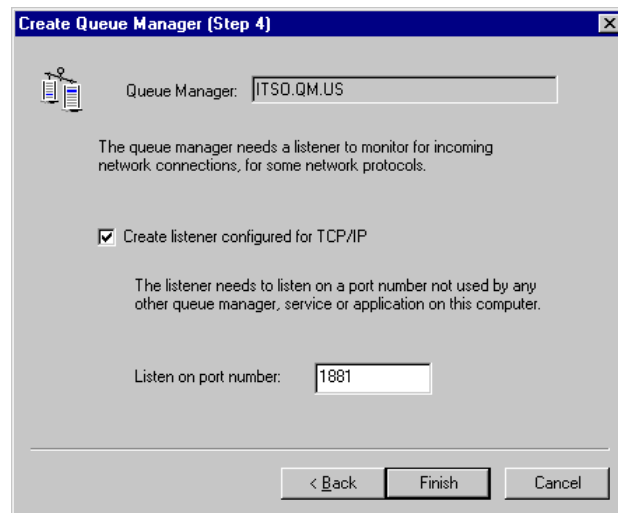


Figure 147. Defining a queue manager - window 4

6. Check **Create listener configured for TCP/IP** and enter a listener port number. We chose 1881.

Choosing listener ports

A listener uses a TCP/IP port to monitor for connection requests from channels that are started from the other end. When it receives a connection request, it starts the channel at its end.

If there are multiple queue managers with listeners on one machine, each listener will require a unique port number. MQSeries defaults to port 1414, the “well-known” port for MQSeries when a listener is created, but you can choose any valid TCP/IP port.

A listener is associated with a queue manager and is started and stopped automatically with the queue manager. You can check the status of the listener by using the MQSeries Services window.

7. Click **Finish** to complete the queue manager creation dialog. Your new queue manager will show up in the MQSeries Explorer window.

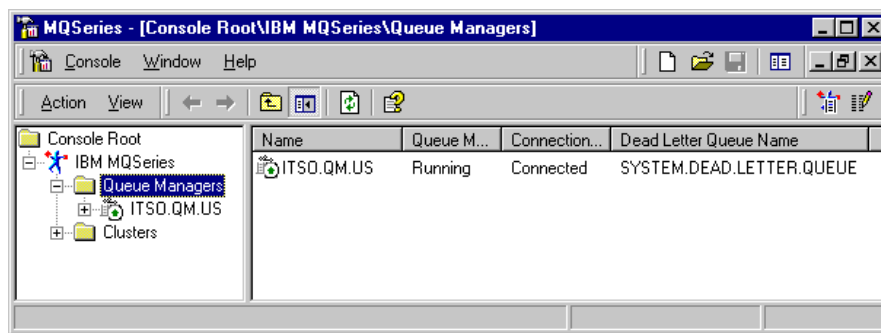


Figure 148. New queue manager in Explorer

12.4.2 Create the MQSI User Name Server

Now that we have all the underlying components installed and configured, we can create the MQSI User Name Server. The User Name Server is created using the `mqsicreateusernameeserver` command. You can enter this from a DOS prompt if you know the correct parameters, or you can use the MQSI Command Assistant to build and execute the command.

We elected to have the Command Assistant generate the User Name Server:

1. Select **Start -> Programs -> MQSeries Integrator Version 2.0 -> Command Assistant -> Create User Name Server**

MQSeries Integrator - Create User Name Server

File View

mqsicreateusernameserver

* Service User ID (-i)	itsouser
* Service Password (-a)	*****
* Queue Manager Name (-q)	ITSO.QM.US
Security Domain (-d)	
Workpath (-w)	
Refresh Interval (secs) (-r)	

* - required parameter

mqsicreateusernameserver -i itsouser -a ***** -q ITSO.QM.US

<<Back Next>> Page 1 of 2 Cancel Help

Figure 149. Creating the User Name Server

The darker boxes indicate the required parameters. As you enter the parameters, you can see the command that will execute being built at the bottom of the window.

For our scenario, we will use the queue manager created in the previous step, ITSO.QM.US.

The User Name Server runs as a Windows NT service. The user ID and password specified are used to start the service. This means the service will operate under any authorities assigned to this ID. See 12.2, “Defining user IDs” on page 322 for information on defining this user ID.

2. Click **Next** and then **Finish**.

Note

In case of errors, you can go to the Windows NT event viewer and review the error messages. To do so:

- **Start->Programs->Administrative Tools(Common)->Event Viewer**
- Then Select **LOG->Application** to see any MQSI messages

12.4.3 Starting the User Name Server

Now we can start the User Name Server, either from a DOS prompt or from the Windows NT services window. From a DOS prompt you would enter:

```
mqsistart usernameserver
```

Check for errors in the Window NT event viewer.

12.5 MQSI Configuration Manager setup

There is one Configuration Manager per broker domain and it must run on Windows NT. It maintains configuration information in the configuration repository, manages the initialization and deployment of brokers and message processing operations in response to actions initiated through the Control Center, and checks the authority of defined user IDs to initiate those actions.

We have already defined the MQSI databases and created a User Name Server. Now, we will create the Configuration Manager, giving us the environment shown in white in Figure 150.

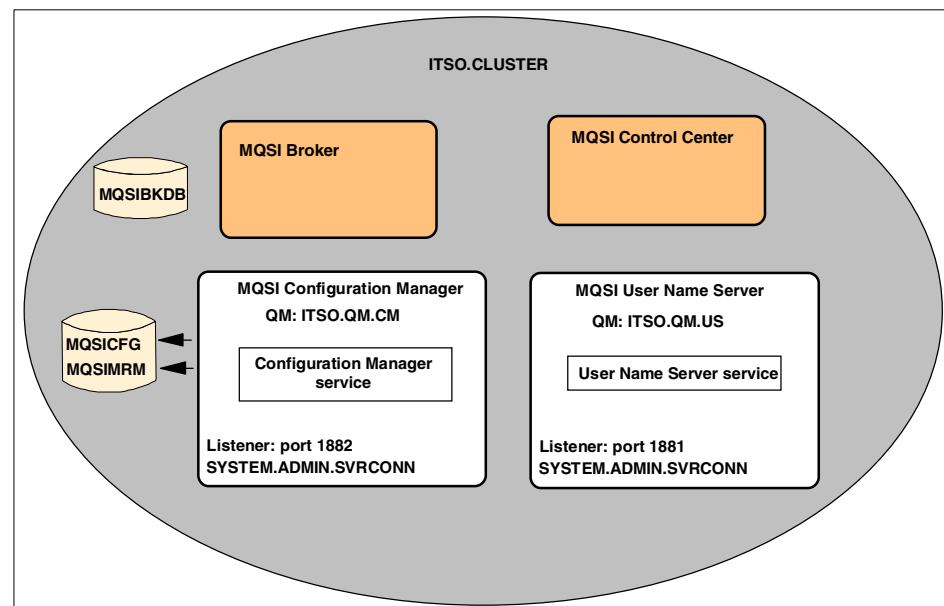


Figure 150. Configuration Manager

12.5.1 Define the databases to the local system

In 12.3, “MQSI database setup” on page 322, two databases were defined for the Configuration Manager on a remote DB2 server. Now, we need to define these databases to the DB2 client on the Configuration Manager system and you register them as ODBC resources.

12.5.1.1 For remote databases

If the database is remote, as in our case, you can use the DB2 Client Configuration Assistant to perform the necessary tasks. The first step is to define the configuration repository database.

1. Start the DB2 Client Configuration Assistant on the Configuration Manager machine:

Start->Programs->DB2 for windows NT->Client Configuration Assistant

2. From the main window click the **Add** button.
3. On the first tab of the Configuration Assistant, select option **Manually configure a connection**.
4. Select **TCP/IP**.
5. Provide the following information about the remote DB2 server in the next tab:
 - Hostname: The TCP/IP host name for our DB2 server is DB2SRVR.
 - Port number: We are using the default DB2 port number of 50000. This is determined when the DB2 server is installed.
6. On the next tab, enter the database name as both the database name and the alias. In our Windows NT scenario shown in Figure 139 on page 320 the configuration database is called MQSICFG.
7. Finally, check the box on the ODBC tab to register the database to ODBC as a system data source. This is mandatory for the message repository and optional for the configuration repository.
8. Click **Done**.
9. You may test the connection by selecting **Test Connection**. This is a good time to make sure the database can be accessed using the user ID and password you have chosen for the Configuration Manager database access.

Repeat the process for the message repository database (MQSIMRM in our scenario).

Finding the DB2 server port number

You can check the DB2 connection port for a DB2 server on AIX by scanning the AIX /etc/services file or on Windows NT by scanning the \Winnt\system32\drivers\etc\services file. In this example, the DB2 connection service is db2cdb2inst1 on the default TCP port 50000.

```
db2cdb2inst1 50000/tcp # Connection port for instance db2inst1
db2idb2inst1 50001/tcp # Interrupt port for instance db2inst1
```

12.5.1.2 Bind the database

On multi-way machines you will need to bind the databases to the db2cli package. To do this, open a DB2 command line processor window and execute the following for each database:

```
db2 CONNECT TO MQSICFG USER userid USING password
db2 BIND C:\SQLLIB\BND\@DB2CLI.LST BLOCKING ALL GRANT PUBLIC
db2 CONNECT RESET
```

12.5.1.3 For local databases

If you defined the Configuration Manager repository on your local machine, you will still need to register it as an ODBC resource:

1. Start the DB2 Client Configuration Assistant on the Configuration Manager machine:

Start->Programs->DB2 for windows NT->Client Configuration Assistant

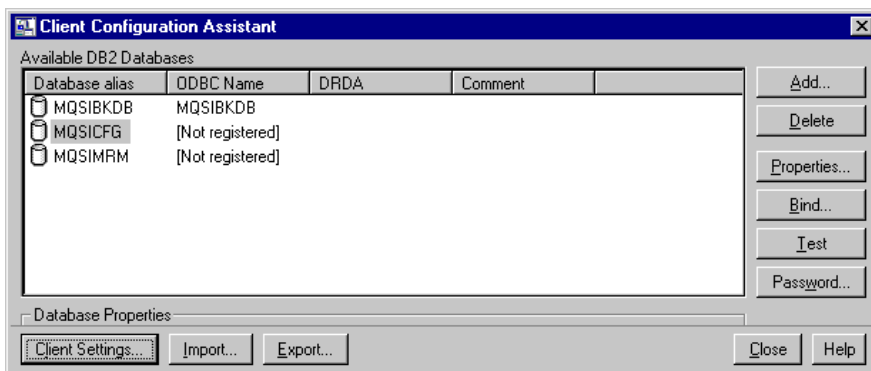


Figure 151. Client Configuration Assistant

2. From the main window select the database and click the **Properties** button.

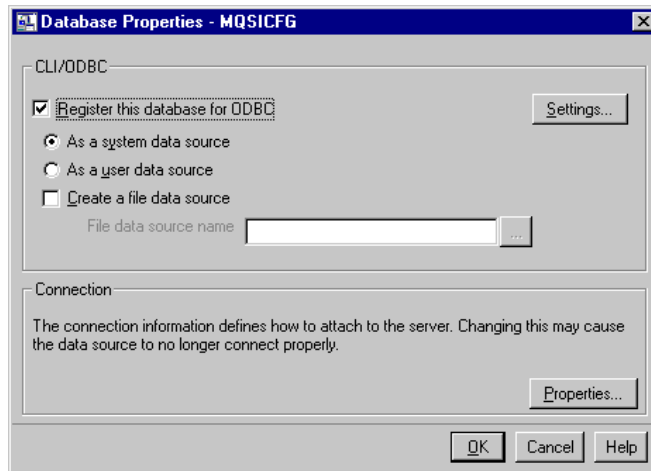


Figure 152. Registering a database as an ODBC resource

3. Check **Register this database for ODBC** and click **OK**.

12.5.2 Create the queue manager

The next step is to build the MQSeries infrastructure required for the Configuration Manager. The steps are the same as for the User Name Server and can be seen in 12.4.1, “Create the queue manager” on page 325. We will call the queue manager for this system ITSO.QM.CM.

The following steps were used to define the MQSeries definitions required for the MQSI Configuration Manager:

1. Click **Start->Programs->IBM MQSeries->MQSeries Explorer**.
2. Right-click **Queue Managers** and select **New->Queue Manager**.
3. Enter the following in the Create Queue Manager window:
 - ITSO.QM.CM (queue manager name is case sensitive)
 - Check **Make this the default queue manager**
 - SYSTEM.DEAD.LETTER.QUEUE for the DLQ name

Click **Next** to continue.

4. Accept the defaults for logging and click **Next** to continue.

5. Make sure that **Start Queue Manager** and **Create Server Connection Channel** are both checked. Click **Next** to continue.
6. Check **Create listener configured for TCP/IP** and enter a listener port number. We chose 1882.
7. Click **Finish** to complete the queue manager creation dialog. The new queue manager can now be seen in the MQSeries Explorer window.

Note: There is still no connection defined between the two queue managers on the User Name Server and Configuration Manager. You will only see the local queue manager from the MQSeries Explorer window. We will connect the two later by putting them both in the same MQSeries cluster.

12.5.3 Create the MQSI Configuration Manager

Now that we have all the underlying components installed and configured, we can create an MQSI Configuration Manager. The Configuration Manager is created using the `mqsicreateconfigmgr` command. You can enter this from a DOS prompt, or you can use the MQSI Command Assistant to build and execute the command. We elected to have the Command Assistant generate the Configuration Manager:

1. Select **Start -> Programs -> MQSeries Integrator Version 2.0 -> Command Assistant -> Create Configuration Manager**.

MQSeries Integrator - Create Configuration Manager

File View

mqsicreateconfigmgr

* Service User ID (-i)	itsouser
* Service Password (-a)	*****
* Queue Manager Name (-q)	ITSO.QM.CM
User Name Server QMgr Name (-s)	
Security Domain (-d)	
Workpath (-w)	

* - required parameter

mqsicreateconfigmgr -i itsouser -a ***** -q ITSO.QM.CM

<<Back Next>> Page 1 of 3 Cancel Help

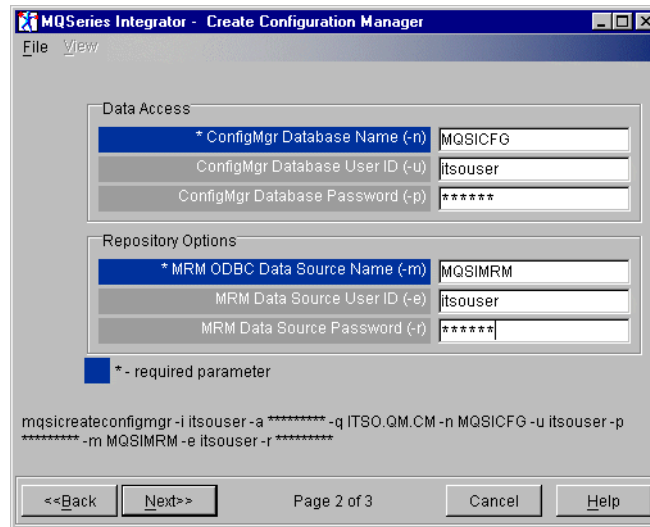
Figure 153. Create Configuration Manager window 1

The darker boxes indicate the required parameters. As you enter the parameters, you can see the command that will execute being built at the bottom of the window.

For our scenario, we will use the queue manager created in the previous step, ITSO.QM.CM.

The Configuration Manager runs as a Windows NT service. The user ID and password specified are used to start the service. This means the service will operate under any authorities assigned to this ID. See 12.2, “Defining user IDs” on page 322 for information on defining this user ID.

Click **Next** to continue.



The screenshot shows the 'MQSeries Integrator - Create Configuration Manager' window, specifically the 'Data Access' and 'Repository Options' sections. The 'Data Access' section has three fields: '* ConfigMgr Database Name (-n)' with value 'MQSICFG', 'ConfigMgr Database User ID (-u)' with value 'itsouser', and 'ConfigMgr Database Password (-p)' with value '*****'. The 'Repository Options' section has three fields: '* MRM ODBC Data Source Name (-m)' with value 'MQSIMRM', 'MRM Data Source User ID (-e)' with value 'itsouser', and 'MRM Data Source Password (-r)' with value '*****'. A legend indicates that fields with an asterisk are required parameters. At the bottom, a command line is displayed: 'mqsicreateconfigmgr -i itsouser -a ***** -q ITSO.QM.CM -n MQSICFG -u itsouser -p ***** -m MQSIMRM -e itsouser -r *****'. Navigation buttons at the bottom include '<<Back', 'Next>>', 'Page 2 of 3', 'Cancel', and 'Help'.

Figure 154. Create Configuration Manager window 2

2. Unlike the User Name Server, the Configuration Manager requires two databases. The next panel configures the database names and the user ID / password used to access them. These databases were created on a remote DB2 server earlier in 12.3, “MQSI database setup” on page 322, and defined to the local system in 12.5.1, “Define the databases to the local system” on page 331.

Click **Next** to continue.

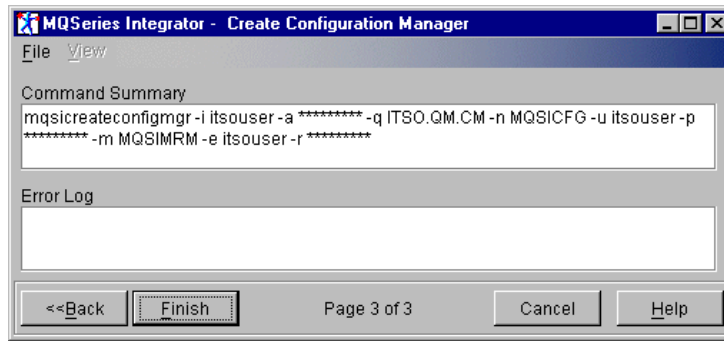


Figure 155. `mqsicreateconfigmgr` command

3. The final window shows the `mqsicreateconfigmgr` command that will execute. Click **Finish** to run the command and create the Configuration Manager.

Note

In case of errors, go to the Windows NT event viewer to see the error messages:

- **Start->Programs->Administrative Tools(Common)->Event Viewer**
- Select **LOG->Application** to see any MQSI messages

4. Now we can start the Configuration Manager, from either a DOS prompt or the Windows NT services window. From a DOS prompt enter:

```
mqsistart configmgr
```

Check for errors in the Window NT event viewer. Open the MQSeries Services to see the status of the listener. It should start automatically when the queue manager starts.

12.6 Define the MQSeries cluster

At this point you have two MQSeries systems and queue managers defined, but no connection between them. We are going to put our MQSeries managers in a common cluster. This will automatically build the definitions needed to connect them. The advantages of clustering is discussed in 10.1.4, "Overview of the MQSeries clustering feature" on page 249.

A cluster needs two systems to act as primary and secondary cluster repositories. In our Windows NT scenario, we have chosen to use the User Name Server and Configuration Manager queue managers as the cluster repository queue managers. Since the MQSeries structure exists on both, this is an appropriate time to build the cluster.

We will define our cluster from the Configuration Manager system using the Create Cluster Wizard, started from the MQSeries Explorer.

Note: In the AIX scenario, we don't build a User Name Server so this step is performed later, after the broker is built. However, since we will be defining the cluster from the Configuration Manager in both cases, which only runs on Windows NT, the steps will look the same. Just substitute the broker where you see references to the User Name Server.

The Create Cluster Wizard will actually define all the channels and queues required for the cluster on both the local queue manager and on the second, remote, queue manager. For this remote MQSeries management to work a few things have to be in place:

- The user ID we are running under, itsouser, has to be defined on the second machine and must be a member of the Windows NT "mqm" group (created during MQSeries install).
- Each queue manager must be able to resolve the connection name of the cluster partner queue manager. For TCP/IP this means the connection name has to be in a domain name server or in the /etc/hosts file of each system.
- The remote queue manager has to have:
 - A running command server
 - A running TCPIP listener
 - An MQSeries SRVCONN type channel

All of these were created in 12.5.2, "Create the queue manager" on page 333 and started automatically when the queue manager was started.

The status of the command server can be seen from the MQSeries Services window by clicking the queue manager.

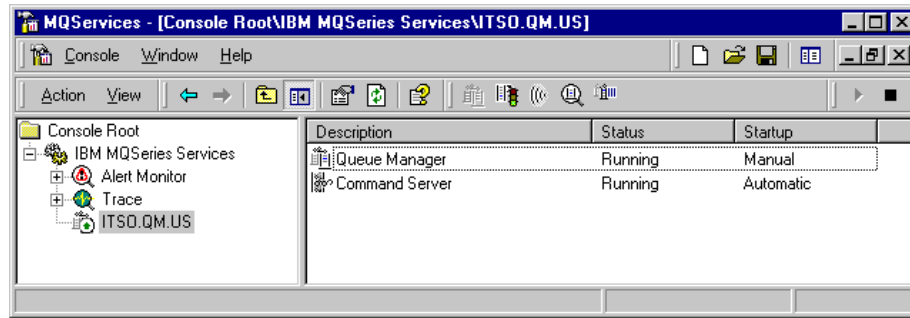


Figure 156. Queue manager command server

A special queue is used by the command server for receiving remote requests. This queue is called the `SYSTEM.ADMIN.COMMAND.QUEUE`. This queue can be seen in the MQSeries Explorer window by clicking on Queues under the queue manager name. It is considered to be a system object, so you will first need to select **View->Show System Objects**.

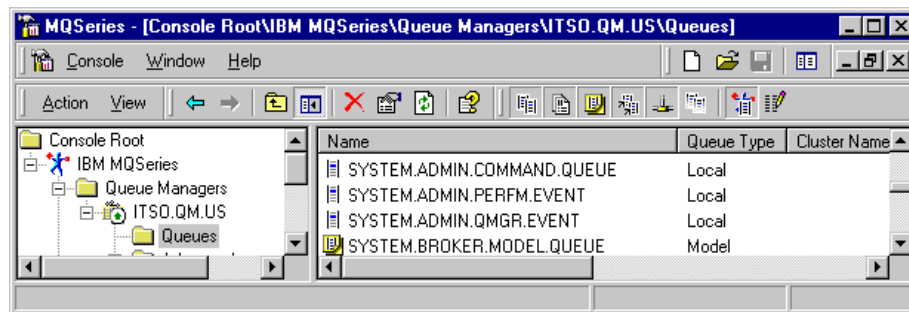


Figure 157. `SYSTEM.ADMIN.COMMAND.QUEUE`

Remote administration is discussed in 10.1.2, "Remote administration" on page 246.

At the end of the procedures to create a cluster, we will have the configuration shown in white in Figure 158.

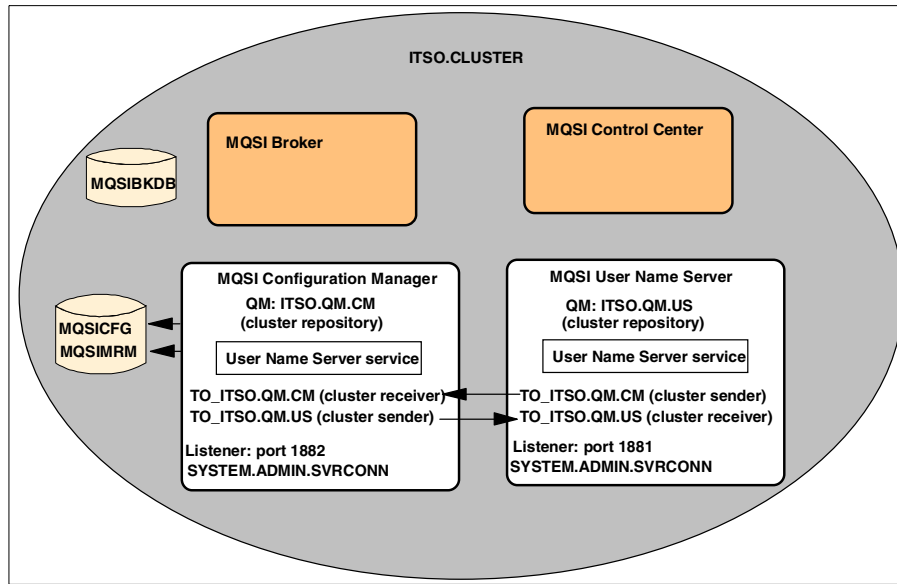


Figure 158. Cluster

To create the new cluster, which we will call ITSO.CLUSTER, do the following:

1. On the Configuration Manager Windows NT system select:
Start -> Programs -> IBM MQSeries -> MQSeries Explorer
2. Right click **Cluster** and select **New->Cluster** to start the Create Cluster Wizard. The first window describes the process you are about to go through. After reading this window, click **Next**.
3. Enter the cluster name. In our scenario we chose ITSO.CLUSTER for our cluster name.

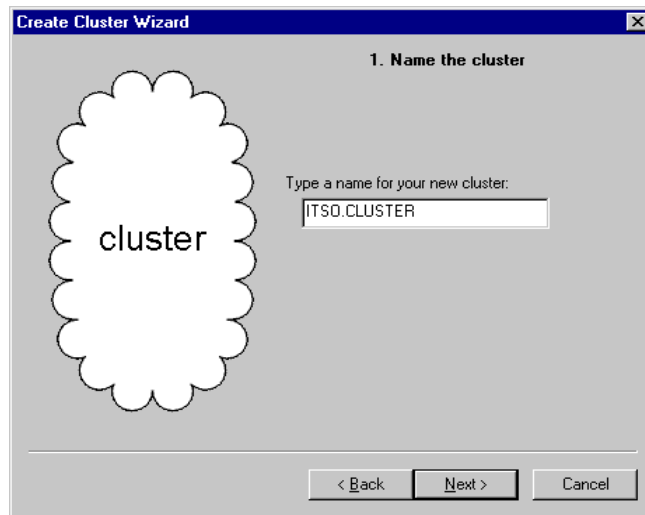


Figure 159. Create cluster window 2

Click **Next** to continue.

4. Enter the first repository queue manager (ITSO.QM.CM in our example).

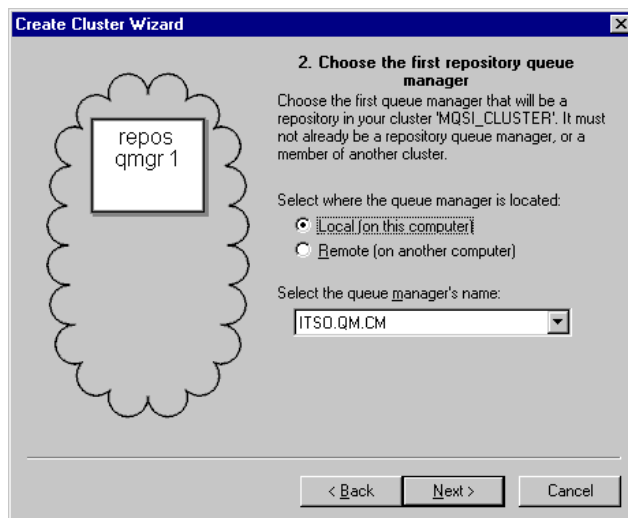


Figure 160. Create cluster window 3

Click **Next** to continue.

5. Enter the second repository queue manager. In our case we are using the User Name Server queue manager, ITSO.QM.US.

The connection name specifies the IP address or TCP/IP host name and the port the queue manager's listener is running on (defined in 12.4.1, "Create the queue manager" on page 325.

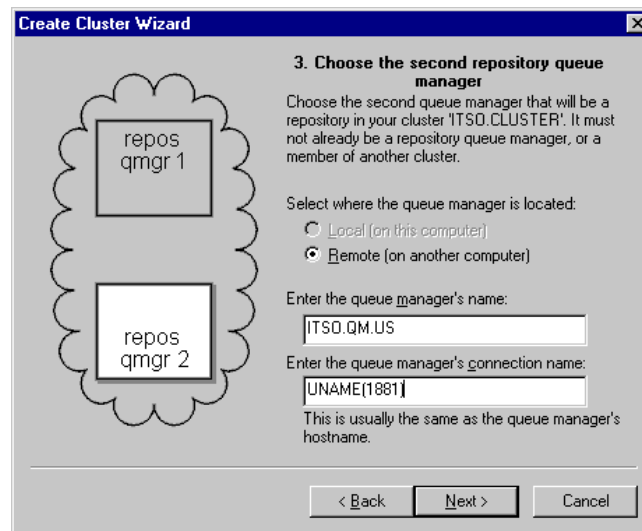


Figure 161. Create cluster window 4

Click **Next** to continue.

6. This next window informs you that you will be defining a cluster sender and a receiver channel on each cluster repository queue manager. Read the information here.

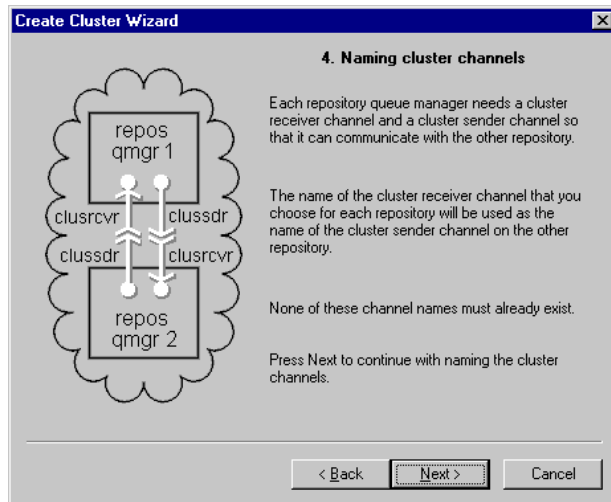


Figure 162. Create cluster window 5

Click **Next** to continue.

7. This panel defines the cluster receiver channel for the primary repository queue manager. The Wizard pre-fills the channel name, TO_ITSO.QM.CM, and the connection name here. Make sure the connection name can be resolved by TCP/IP on both systems.

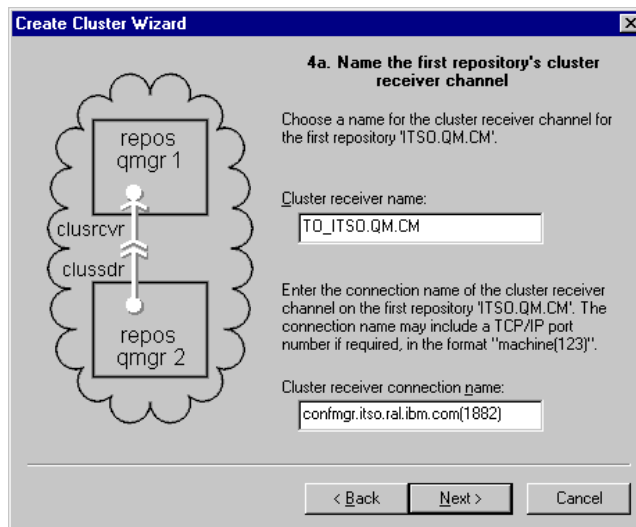


Figure 163. Create cluster window 6

Click **Next** to continue.

8. This panel defines the cluster receiver channel for the secondary repository queue manager. The Wizard pre-fills the channel name, TO_ITSO.QM.US, and the connection name here. Once again, make sure the connection name can be resolved by TCP/IP on both systems.

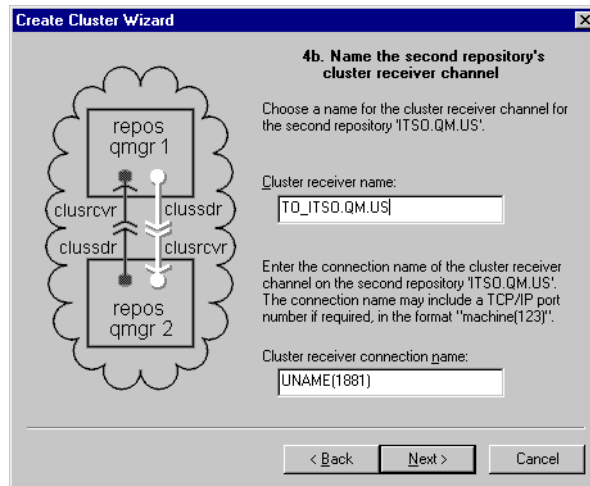


Figure 164. Create cluster window 7

Click **Next** to continue.

9. Confirm your choices and click **Finish** to create the definitions.

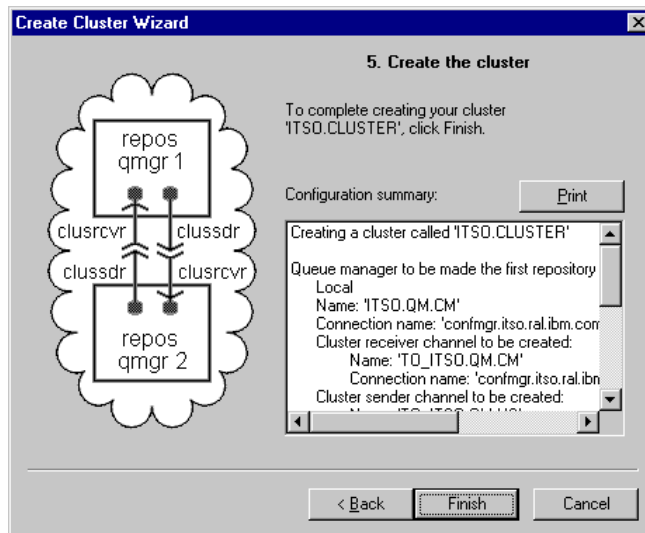


Figure 165. Create cluster window 8

At this point the repository queue managers will contact each other and you will see the current definitions for both in the MQSeries Explorer window.

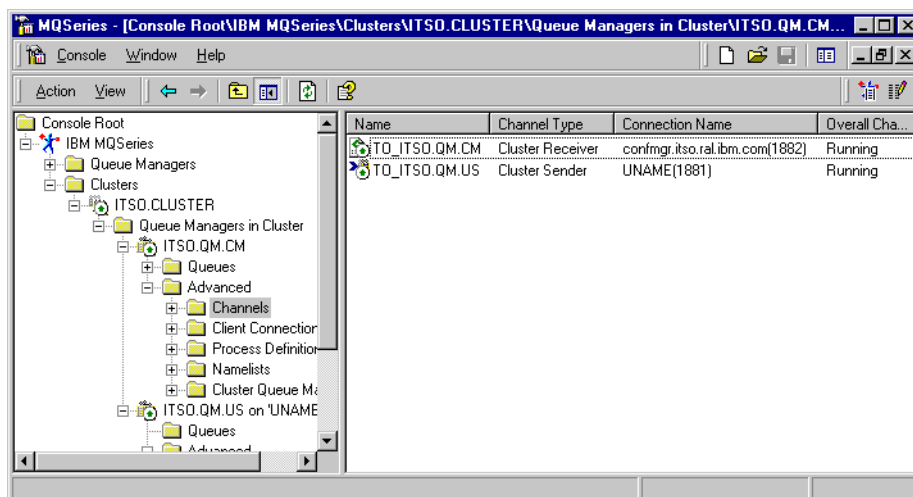


Figure 166. The cluster definitions in the Explorer

12.7 MQSI broker setup

The next step will be to build one or more MQSI brokers to execute the message flows. Your applications communicate with the broker to take advantage of the services provided by these message flows.

You can install, create, and start any number of brokers within a broker domain. In our scenario, we are installing and configuring a single broker, ITSO.QM.BR, on rs600014 (AIX) and BROKER1 (Windows NT).

12.7.1 Define the database to the local system

Each broker requires a database for internal processing. Multiple brokers can share a database. Earlier, in 12.3, “MQSI database setup” on page 322, we showed you how to create the databases. In our scenarios, we have one broker and one database. The database, MQSIBKDB, has been created on a remote DB2 server. We now need to define that database to the DB2 client and register it as an ODBC resource on each broker system.

Note

For performance reasons, you may want to consider putting the broker databases on the same system as the broker. See 7.2.3, “Placement of MQSI databases” on page 113 for more information.

12.7.1.1 Defining the database to AIX

We used the following steps to define the remote broker database, MQSIBKDB, to the DB2 client and to register it as an ODBC resource on the AIX system that will host the broker:

1. As root, edit the /etc/services file to add the db2cdb2inst1 service. this should match the the service defined on the database remote node. Finding the correct entry is discussed in 12.5.1, “Define the databases to the local system” on page 331.

```
db2cdb2inst1 50000/tcp    # Connection port for instance db2inst1
```

2. Log on as the DB2 administrator or switch to that user ID:

```
su - db2inst1
```

3. To define the database to the DB2 client we first need to define, or catalog, the remote DB2 server to the client. This requires giving the server a local name and associating that name with an IP address and DB2 instance.

Assuming the database is on a remote server at host name “RS600014” and that we are going to give our node a local name of “tcpnode”, do the following from an AIX terminal window:

```
db2 catalog tcpip node tcpnode remote rs600014 server db2cdb2inst1
db2 catalog db MQSIBKDB as MQSIBKDB at node tcpnode
```

4. Now bind the DB2 Call Level Interface (CLI) to the database:

```
db2 connect to MQSIBKDB user db2inst1 using db2inst1
db2 bind /home/db2inst1/sqllib/bnd/@db2cli.lst grant public CLIPKG 5
db2 terminate
```

5. The MQSI broker uses ODBC to access its database, so we have to define an ODBC data source for it. In our case we called the data source MQSIBKDB.

Edit /var/mqsi/odbc.odbc.ini (notice the dot before odbc.ini) to add the required information for the broker database:

```
[ODBC Data Sources]
MQSIBKDB=IBM DB2 ODBC Driver
ITSOCUST=IBM DB2 ODBC Driver
ITSOCHEC=IBM DB2 ODBC Driver
ITSOSAVI=IBM DB2 ODBC Driver
MYDB=IBM DB2 ODBC Driver
ORACLEDB=MERANT 3.60 Oracle 8 Driver
ORACLE7DB=MERANT 3.60 Oracle 7 Driver
SYBASEDB=MERANT 3.60 Sybase 11 Driver

[MQSIBKDB]
Driver=/home/db2inst1/sqllib/lib/db2.o
Description=MQSIBKDB DB2 ODBC Database
Database=MQSIBKDB

[ITSOCUST]
Driver=/home/db2inst1/sqllib/lib/db2.o
Description=ITSOCUST DB2 ODBC Database
Database=ITSOCUST
CURRENTSQLID=ITSOUSER
```

6. Make sure that the “root” and “mqsi” user IDs (or any other ID that will be issuing MQSI commands) have access to the DB2 environment.

Executing DB2 commands

The DB2 environment can be initialized for a logged-in user by running the `/home/db2inst1/sqllib/db2profile` shell script.

If a user ID is going to be issuing DB2 commands often, it is recommended that you add `/home/db2inst1/sqllib/db2profile` to either `/etc/profile` or the user's `<home>/.profile` (dot profile), so the DB2 environment is automatically set up when the user logs in.

12.7.1.2 Windows NT

To define the broker database to a Windows NT client, use the method described in 12.5.1, "Define the databases to the local system" on page 331. You will need to define the database to the client, register it as an ODBC resource, and bind it to the `db2cli` package.

12.7.2 Create the queue manager

Now that the broker database is in place and the operating system hosting the broker can find it, the next step is to define the MQSeries infrastructure required for the broker. Initially that means defining a queue manager and joining it to the existing MQSeries cluster.

12.7.2.1 Create the queue manager on Windows NT

The broker queue manager is defined on the broker system in the same manner that we defined the User Name Server and the Configuration Manager. The windows for this process can be seen in 12.4.1, "Create the queue manager" on page 325.

1. Click **Start->Programs->IBM MQSeries->MQSeries Explorer**.
2. Right-click **Queue Managers** and select **New->Queue Manager**.
3. Enter the following in the Create Queue Manager window:
 - ITSO.QM.BR (queue manager name is case sensitive)
 - Check **Make this the default queue manager**
 - SYSTEM.DEAD.LETTER.QUEUE for the DLQ nameClick **Next** to continue.
4. Accept the defaults for logging and click **Next** to continue.
5. Make sure that **Start Queue Manager** and **Create Server Connection Channel** are both checked. Click **Next** to continue.

6. Check **Create listener configured for TCP/IP** and enter a listener port number (we chose 1881). Click **Finish**.

Firewall note

In the basic topology, the WebSphere Application server is in the DMZ and the broker is in the internal network. If this is your topology, you will need to open the listener port chosen in step 6 for TCP on the firewall.

Joining the cluster

Since the cluster for the Windows NT test environment has been created, we will need to join this new queue manager to the existing cluster. We will use the MQSeries Explorer on the broker system to start the Join Cluster Wizard.

1. From the Explorer window, right click on the queue manager and select **All Tasks->Join Cluster**.

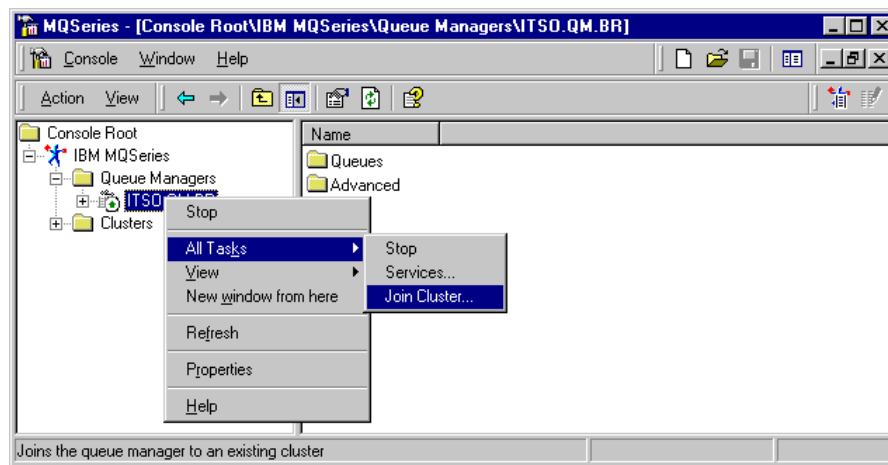


Figure 167. Joining a cluster - window 1

The first window gives you overview information on the steps required. Click **Next** on this window.

2. Enter the cluster name. Our cluster, ITSO.CLUSTER, was created in 12.6, "Define the MQSeries cluster" on page 336. Click **Next**.
3. Enter the queue name and connection information for one of the cluster repository queue managers. In our scenario we will point to ITSO.QM.CM.

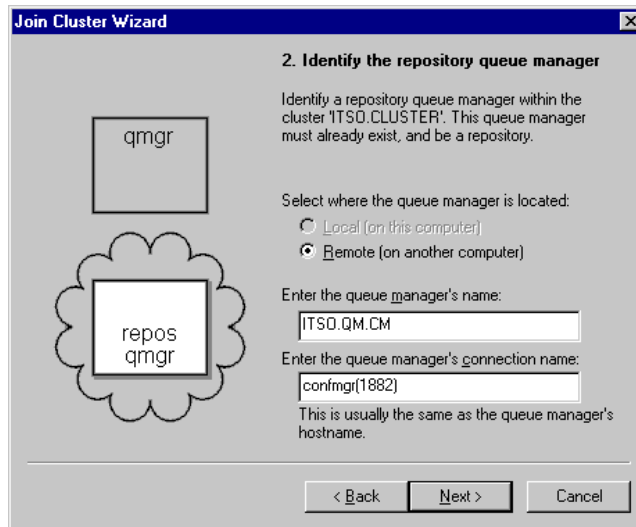


Figure 168. Joining a cluster - window 4

Click **Next**.

4. The next window contains information about the cluster receiver and sender channels required. Read this information and click **Next**.
5. The cluster receiver channel name and connection name are pre-filled in for you. Accept these defaults if they are satisfactory.

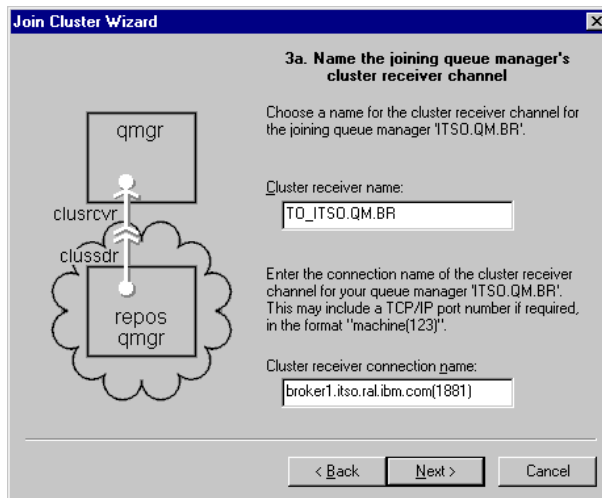


Figure 169. Joining a cluster - window 6

Click **Next**.

- The next window identifies the repository cluster's receiver channel. If you took the default earlier when creating the cluster, you can take the default here.

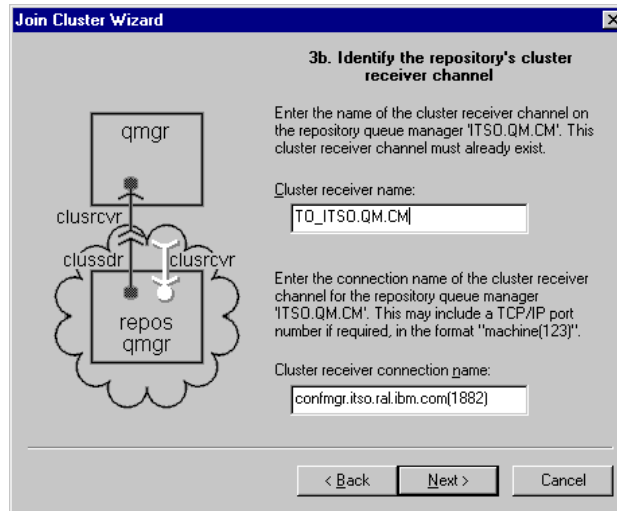


Figure 170. Joining a cluster - window 7

Click **Next** and then click **Finish** to join the broker queue manager to the cluster.

The necessary channels will be created and all the queue managers in the cluster can now be seen in the MQSeries Explorer window.

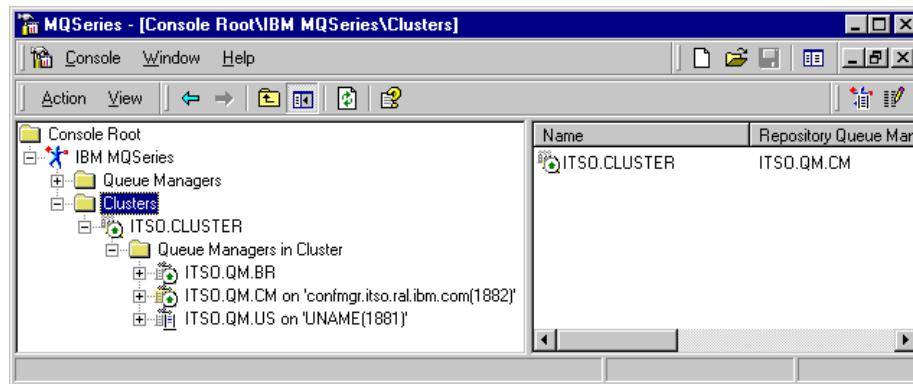


Figure 171. Joining a cluster

12.7.2.2 Defining the MQSeries queue manager on AIX

There is no MQSeries Explorer for AIX, so you will need to use control commands to define the broker's queue manager (ITSO.QM.BR), and prepare it to participate in an MQSeries cluster with the Configuration Manager's queue manager (ITSO.QM.CM).

From the broker system, enter the following to define the queue manager and required resources:

1. Create the queue manager : `crtmqm -q ITSO.QM.BR`
2. Start the the queue manager : `strmqm`
3. Create a file (we will call it broker.mq) with the following MQSC commands:

```
*****
***Broker queue manager definitions***
*****

** Set up the Dead Letter queue *****
ALTER QMGR DEADQ(SYSTEM.DEAD.LETTER.QUEUE

** SRVCONN Channel for remote management from NT MQ explorer**
DEFINE CHANNEL(SYSTEM.ADMIN.SVRCONN) +
CHLTYPE(SVRCONN) TRPTYPE(TCP) REPLACE
```

Run the commands in this file by entering: `runmqsc < broker.mq`

4. Start the command server : `strmqcsv`
5. Start the MQSeries TCP/IP listener : `runmqclsr -t TCP -p 1444 -m ITSO.QM.BR &`

Firewall note

In the basic topology, the WebSphere Application server is in the DMZ and the broker is in the internal network. If this is your topology, you will need to open the listener port chosen in step 6 for TCP on the firewall.

Defining the cluster

Remember that in the AIX lab setup we did not use a User Name Server and have not defined a cluster yet. The cluster can now be defined from the Configuration Manager NT node, using the MQSeries Explorer and Cluster Wizard. This will be the same process followed in 12.6, "Define the MQSeries cluster" on page 336.

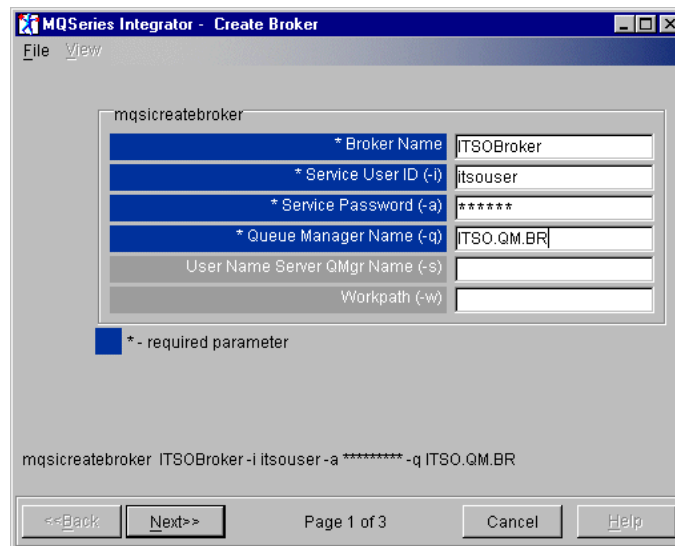
12.7.3 Creating the MQSI broker

The next step is to create the MQSI broker.

12.7.3.1 MQSI broker on Windows NT

Creating the MQSI broker on Windows NT is similar to creating the User Name Server and the Configuration Manager. The command used is the `mqsicreatebroker` command and can be entered from a DOS prompt, or can be created with the MQSI Command Assistant. We chose to use the Command Assistant.

1. Select **Start->Programs->IBM MQSeries Integrator 2.0->Command Assistant->Create Broker**



The image shows a Windows-style dialog box titled "MQSeries Integrator - Create Broker". It has a menu bar with "File" and "View". The main area contains a list of parameters for the `mqsicreatebroker` command. The parameters are: `* Broker Name` (value: ITSOBroker), `* Service User ID (-i)` (value: itsouser), `* Service Password (-a)` (value: *****), `* Queue Manager Name (-q)` (value: ITSO.QM.BR), `User Name Server QMgr Name (-s)` (empty), and `Workpath (-w)` (empty). A legend indicates that asterisks (*) denote required parameters. At the bottom, the command line is displayed: `mqsicreatebroker ITSOBroker -i itsouser -a ***** -q ITSO.QM.BR`. Navigation buttons include "<<Back", "Next>>", "Page 1 of 3", "Cancel", and "Help".

Parameter	Value
* Broker Name	ITSOBroker
* Service User ID (-i)	itsouser
* Service Password (-a)	*****
* Queue Manager Name (-q)	ITSO.QM.BR
User Name Server QMgr Name (-s)	
Workpath (-w)	

* - required parameter

mqsicreatebroker ITSOBroker -i itsouser -a ***** -q ITSO.QM.BR

<<Back Next>> Page 1 of 3 Cancel Help

Figure 172. Creating a broker - window 1

The darker boxes in Figure 172 indicate the required fields. Each broker has a unique name assigned. We will call this one ITSOBroker. The user ID and password to be used to run the service are required, as is the MQSeries queue manager name to be used. We will use the default queue, ITSO.QM.BR.

Click **Next** to continue.

2. Enter the name of the broker database created earlier.

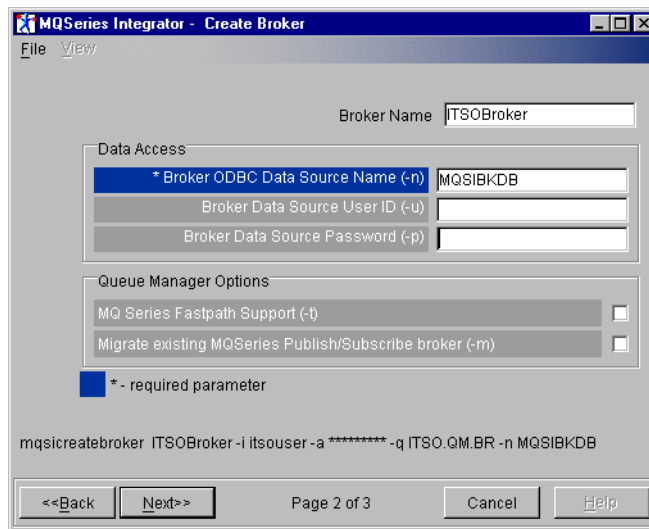


Figure 173. Creating a broker - window 2

Click **Next** to continue.

3. The last window shows the `mqsicreatebroker` command that will execute.

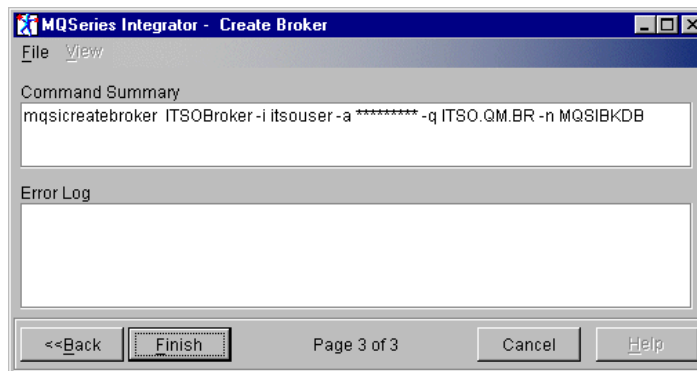


Figure 174. Creating a broker - window 3

Click **Finish** to create the broker.

Common error

ODBC return code -1: make sure the bind was done for the broker database.

Start the broker service

Start the broker service from the Windows NT Services window.

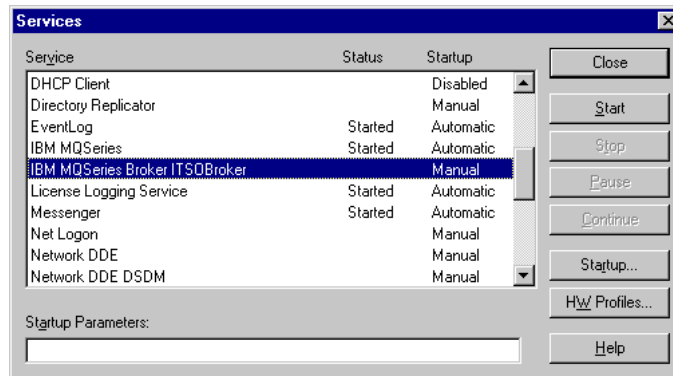


Figure 175. Starting the broker service

12.7.3.2 MQSI broker on AIX

Note

Before creating a broker on AIX, it is recommended that you configure syslog to write all user messages to a file. See 10.2.7.3, “Command results and MQSI messages” on page 276 for instructions on how to do this.

Now we are ready to create an MQSI broker. In AIX the only option we have to do so is by using the `mqsicreatebroker` command.

We will execute this command while logged in with the user ID, `mqsi` (see 11.8, “Planning user IDs” on page 315). Remember that `mqsi` has to be a member of the `mqbrkr` and `mqm` groups, and should have access to the DB2 environment.

1. Execute the following command:

```
mqsicreatebroker ITSO.QM.BR -i mqsi -a mqsipw -u mqsi -p mqsipw -q  
ITSO.QM.BR -n MQSIBKDB
```

-i	The broker will run under this user ID
-a	Password for the broker's user ID
-u	User ID for access to the broker database
-p	Password for database access
-q	Queue manager name
-n	Database name (or alias)

2. Check for status messages in the MQSI syslog file.

Note

You may encounter the following error on an `mqsicreatebroker` command:

```
BIP8040E: Unable to connect to the database
```

This probably means the database cannot be accessed with the user ID and password that were specified when the broker was created. Check that the database is running, that an ODBC connection has been created, and that the user ID/password specified for database access on the `mqsicreatebroker` command are capable of being used to connect to the database.

You may need to execute the following command to set up the DB2 environment for your session:

```
. /home/db2inst1/sqllib/db2profile
```

(Notice the dot and space preceding `/home`)

3. Start the MQSI Broker

```
mqsistart ITSO.QM.BR
```

Check for status messages in the MQSI syslog file.

12.8 Using the Control Center to deploy an application

Our scenario uses the application we created in Chapter 9, “Developing the MQSI application” on page 183. After creating the MQSI application with the Control Center, we exported the message flows to make it easier to set up the various lab scenarios we were using. This means that any time we set up a new MQSI network (including a new Control Center), we only had to import the message flows and deploy them to the appropriate brokers.

The Control Center only runs on Windows NT but can be used to deploy and monitor applications on brokers on any platform.

Note

Before starting the Control Center, you will need to go to EVERY MQSI system and add the user ID you are running the Control Center under. Be sure to add the user ID to the MQSI groups that match the roles you will be using (see 10.2.8, “Control Center operations” on page 278).

Start the Control Center by selecting:

Start -> Program Files -> IBM MQSeries Integrator 2.0 -> Control Center

A Configuration Manager Connection dialog box should appear as shown in Figure 176, prompting you for details required to connect to the Configuration Manager.

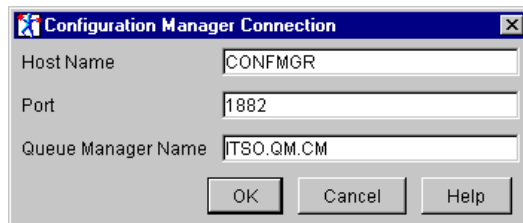


Figure 176. Configuration Manager Connection

These fields tell the Control Center how to connect to the appropriate Configuration Manager. These values were determined when the Configuration Manager was created in 12.5.3, “Create the MQSI Configuration Manager” on page 334.

If this does not appear, you may get an error dialog that reports a problem connecting to the Configuration Manager. This can be the case if the Control Center has been used to work with a different Configuration Manager instance which is no longer active or if this is the first time you have opened the Configuration Manager. You may see no dialog appear at all. This indicates the connection to the Configuration Manager has worked. If the connection dialog has not appeared, you can go to **File->choose Connection**, and the required dialog box will appear.

Click **OK** and the Control Center will open.

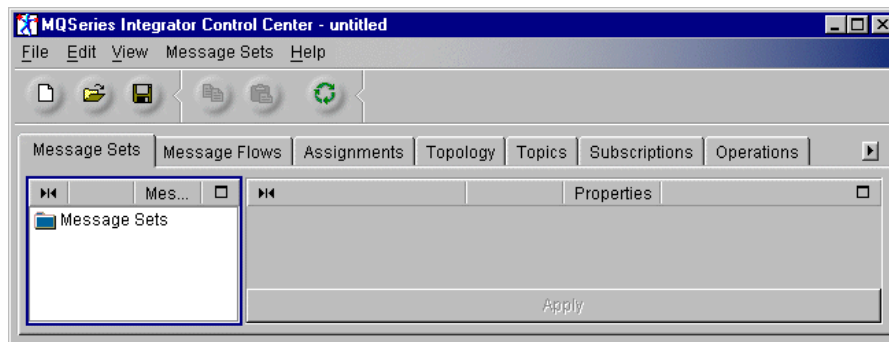


Figure 177. Starting the Control Center

Possible error

If the connection times out, go to the Windows NT services window and make sure the Configuration Manager service is started.

12.8.1 Connecting to the broker

Next, we need to tell the Configuration Manager where its brokers are. In our case we have one broker, called ITSOBroker, that resides on a queue manager called ITSO.QM.BR.

1. In the Control Center, select the **Topology** tab. In the Domain Hierarchy pane (the left-most pane), right-click the **Topology** node and select **Check Out** as in Figure 178.

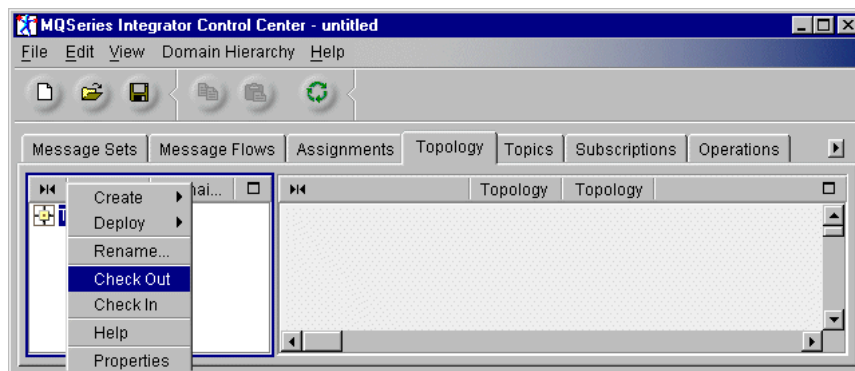


Figure 178. Check out the topology

Note: If you get an authorization failure, make sure the user ID you are operating under is a member of the mq security groups on all the system in the broker domain.

A key symbol appears next to the Topology node to indicate that you have control of it.

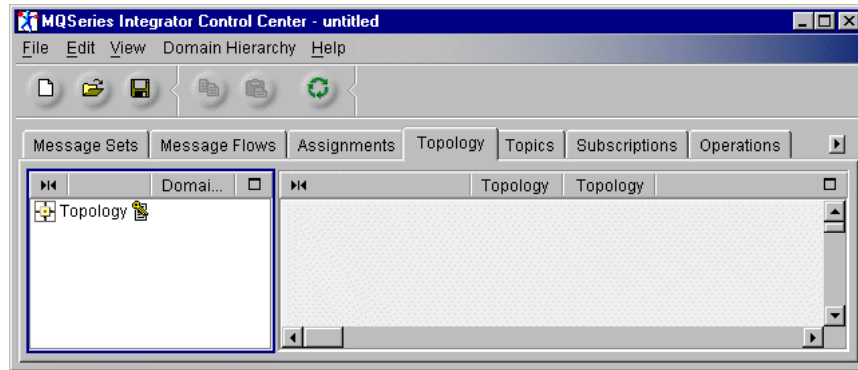


Figure 179. Defining the broker to the Configuration Manager

2. Now right-click the **Topology** node again, and select **Create->Broker**. The following dialog will appear requesting details for the broker.

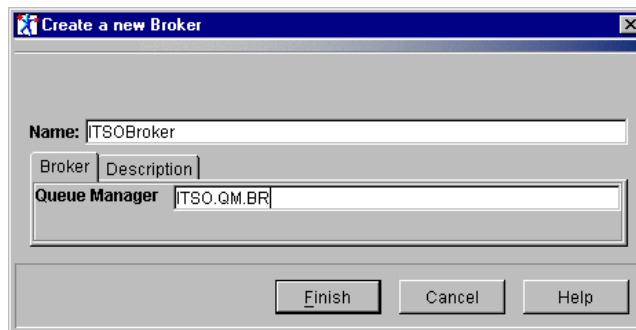


Figure 180. Broker definition

The fields here tell the Control Center how to reach the broker. The broker name was determined when the broker was created in 12.7.3, “Creating the MQSI broker” on page 352. The Control Center will communicate with it by using the MQSeries queue ITSO.QM.BR.

3. Click **Finish** to define the broker to the topology. The newly created broker will appear in the Topology pane.

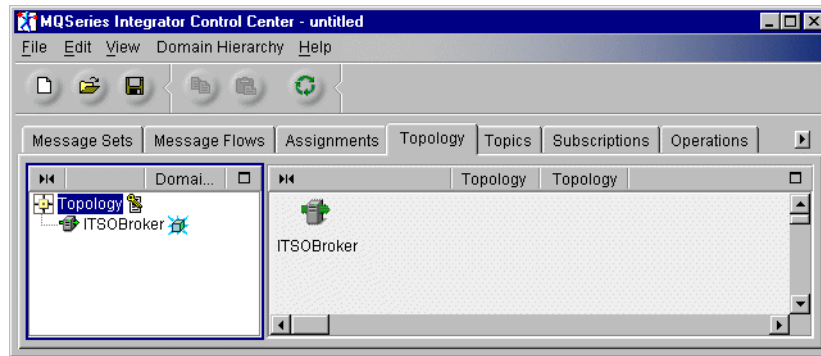


Figure 181. Broker

12.8.2 Creating an execution group

Next, we will create an execution group in the broker to run our application's message flows.

1. Click the **Assignments** tab. In the Domain Hierarchy pane (the left-most pane), right-click **ITSOBroker**, and choose **Create -> Execution Group**.

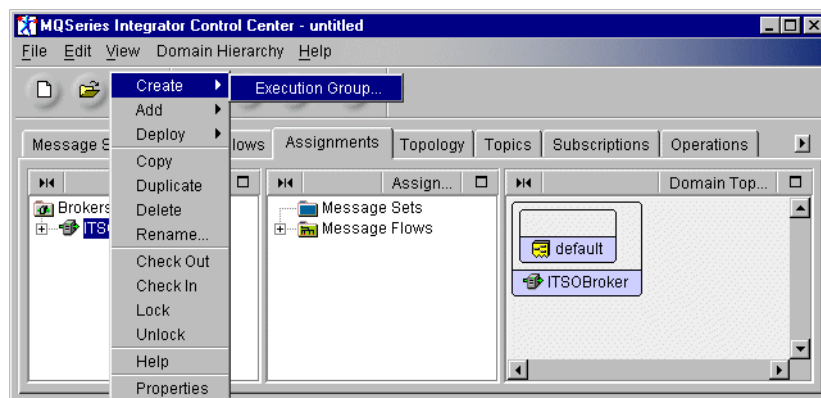


Figure 182. Create an Execution Group

2. In the dialog that appears, enter the name of the new execution group (we entered **ITSO Execution Group**).

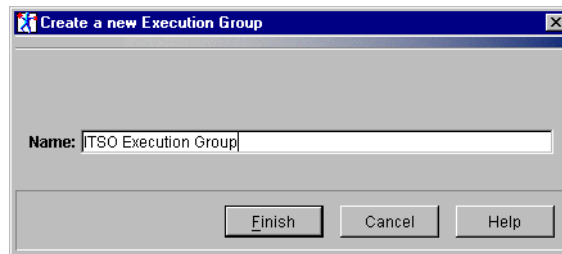


Figure 183. New execution group name

3. Click **Finish**. The new execution group will now appear in the Domain Topology panel in the box for ITSOBroker.

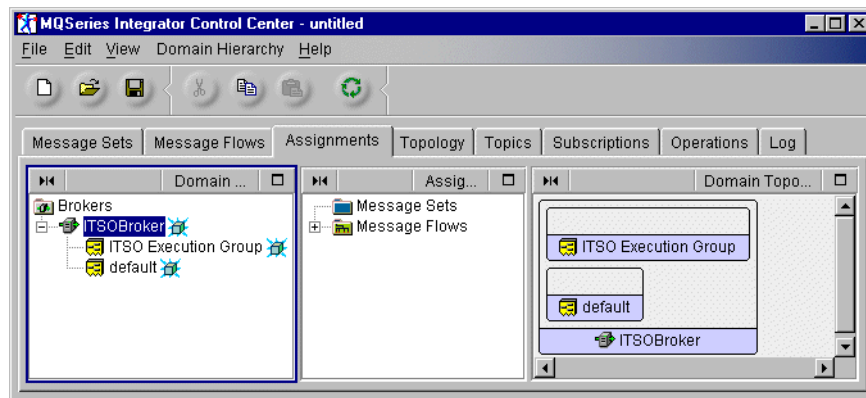


Figure 184. New execution group

12.8.3 Importing message flows

For lab purposes, our message flows were developed in the Control Center and then exported to a file. This allowed us to set up the lab many times, importing the message flow definitions each time to recreate our environment. Importing and exporting definitions is discussed in 10.2.9, “Resource definition management” on page 288.

To import the message flows to our workspace:

1. From the File menu, choose **Import**.

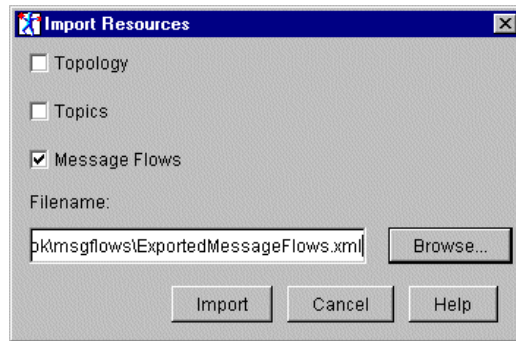


Figure 185. Import message flows

2. Click **Browse** and navigate to the supplied XML file containing the message flows. Then click **Import** to import the message flows.

You will be prompted with a dialog that asks if you wish to save your changes. This is a good time to save your workspace to a configuration file of your choice. The file will be saved as an XML file.

3. Switch to the Message Sets tab to see the imported message flows. In Figure 186, everything you see in the Message Flows pane except the IBM Primitives has been imported.

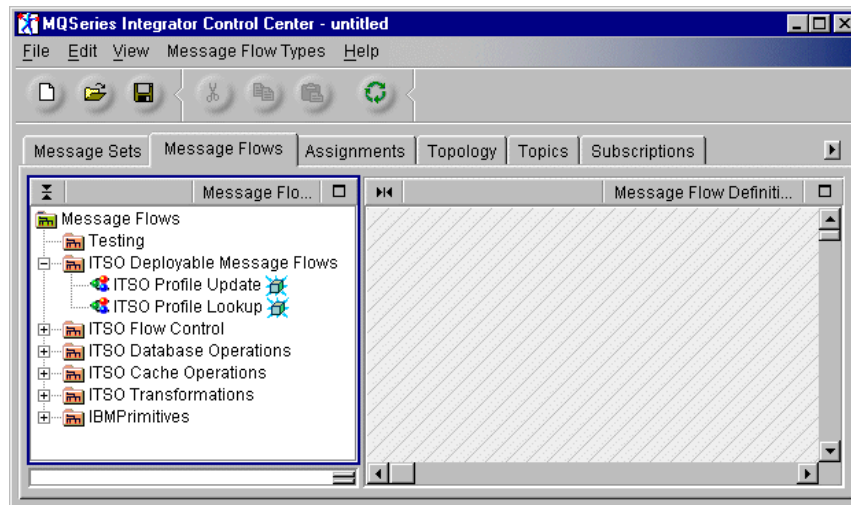


Figure 186. The imported example message flows

12.8.4 Assigning the message flows to the execution group

Next, we must assign the message flows to the broker.

1. Click the **Assignments** tab.

In the Assignable Resources pane (the middle pane), click the **+** symbol next to the Message Flows entry to see all the defined message flows.

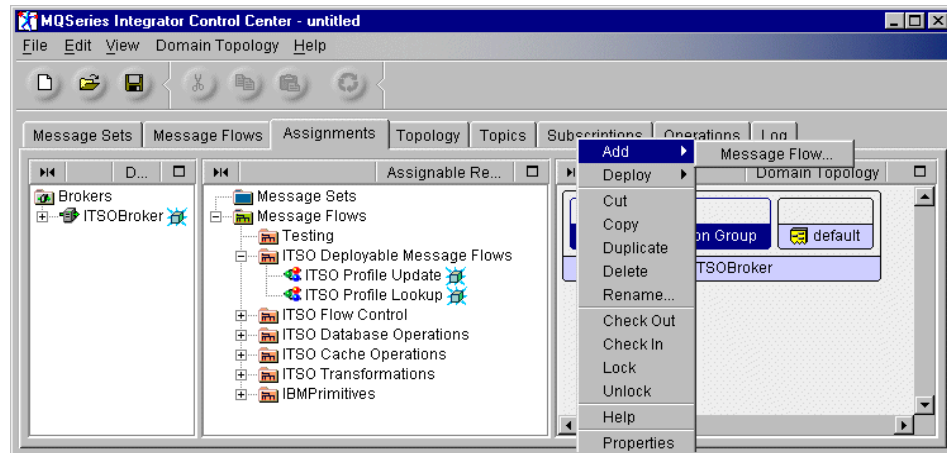


Figure 187. Add message flow

The message flows for our application are listed under ITSO Deployable Message Flows. These two message flows provide the total function of the MQSI portion of the application. The other message flows listed are used as message flow nodes and are included as nodes in the deployable message flows. See 9.7, “Piecing together the lookup components” on page 228 for information on the way this is done.

In the Domain Topology pane (the right-most pane) you will see the broker we added (ITSOBroker) and its execution groups. Right click the desired execution group and select **Add -> Message Flow**.

2. In the dialog that appears, use the mouse and the Ctrl key to select the two message flows shown in Figure 188, and click **Finish**.

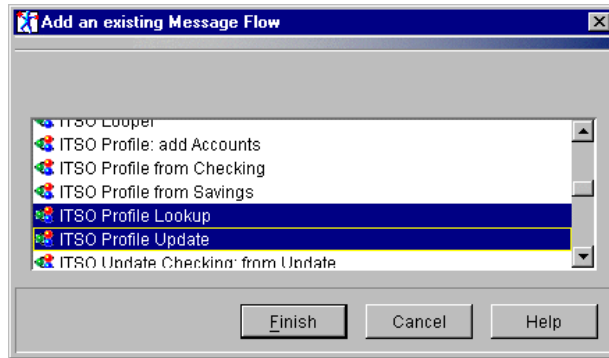


Figure 188. Select the message flows to add

12.8.5 Saving the configuration and deploying it to the broker

Our configuration changes are now complete. Two tasks remain. First we must save our configuration to the shared repository held by the Configuration Manager. Finally we will deploy the changes to the broker domain so they may become operational.

1. From the File menu, select **Check In -> All (Save to Shared)**. This will take all components that exist only in your workspace and check the latest version into the shared configuration.

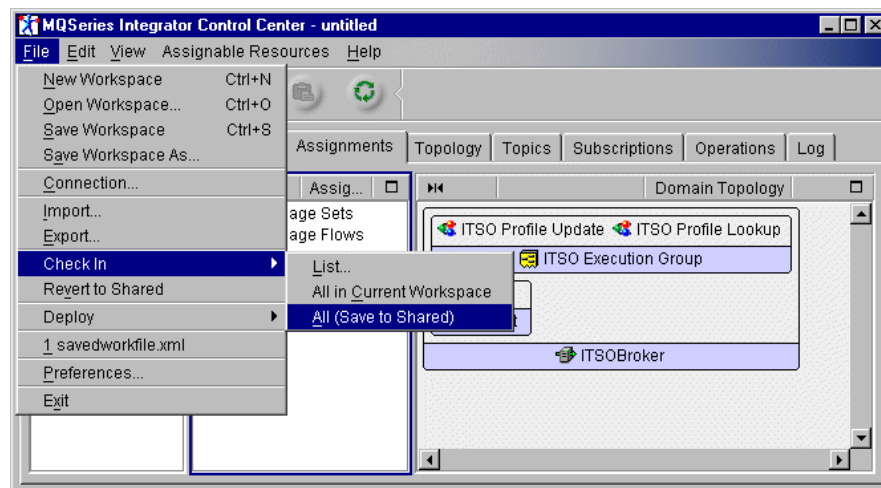


Figure 189. Saving changes to the shared configuration

Next, we must deploy the changes to the broker domain.

2. From the File menu, select **Deploy -> Delta configuration (all types)**.

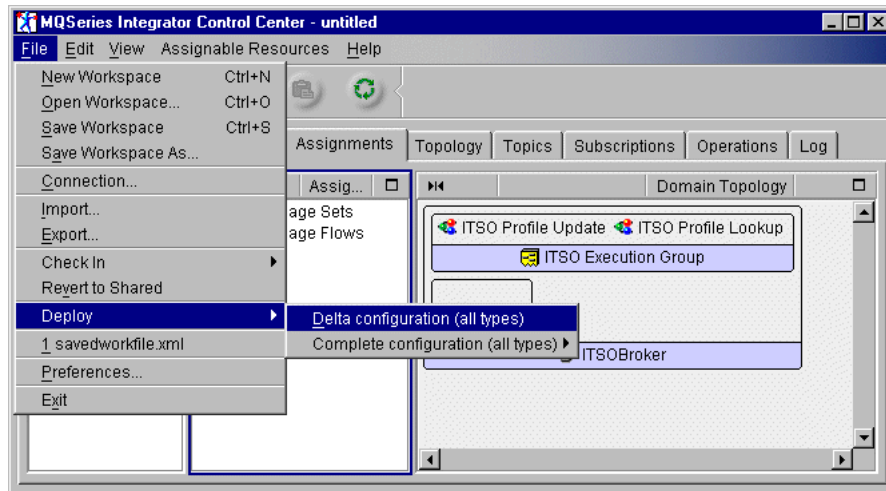


Figure 190. Deploying to the broker domain

Note

There are two types of deployment, complete and delta. See 10.2.8.6, “Deploying applications” on page 286 for a description of each. Before deploying your application make sure you have the following MQSI fixes applied:

- Windows NT: IC27806
- AIX: IY12651

3. A dialog will appear acknowledging your deployment request. This is characteristic of the asynchronous operation of MQSI. It simply means that the request has been placed on the MQSeries queue, not that the deployment is complete. When the deployment is complete, the results can be reviewed in the log pane, as shown in Figure 191.

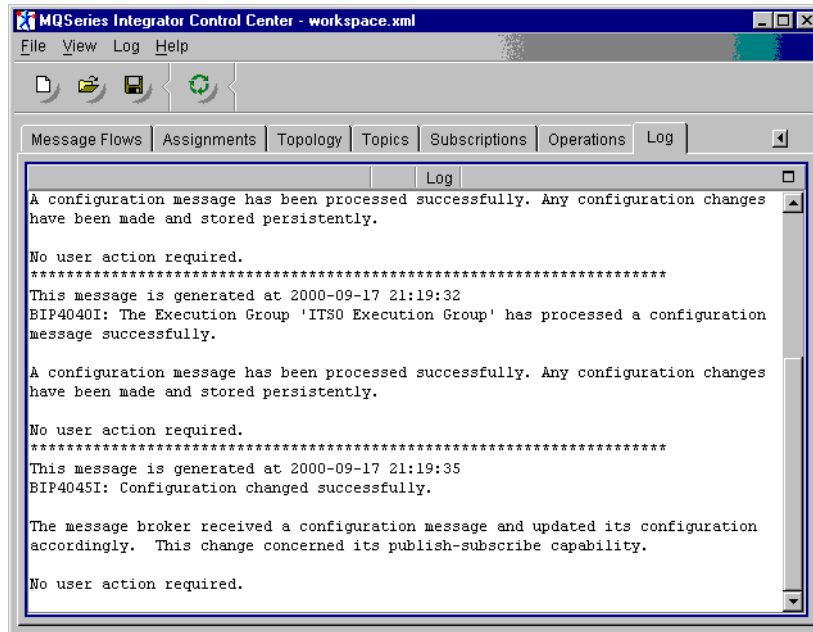


Figure 191. The Log tab, showing successful deployment

- It may take a few seconds for the message to be returned. If no message is shown, right-click the Log tab content and select **Refresh**.
4. Switch to the Operations tab to see the current operational status of the broker domain. It is always necessary to click the **Refresh** button to get a view of the current operational position. If you find that the Refresh button is disabled, click the blank Domain Topology pane (the right-most pane) and then click the **Refresh** button.

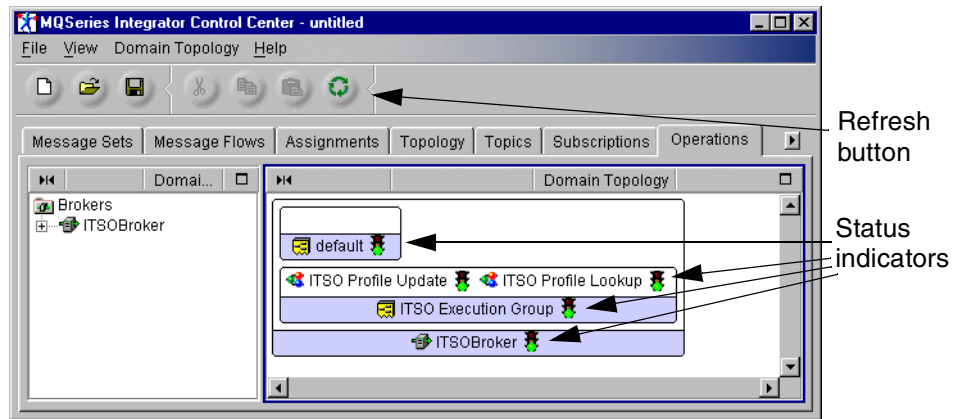


Figure 192. Current broker domain operations

The color of the traffic light symbols indicate the status of the brokers, execution groups and message flows.

12.9 Preparing the broker for the application

The last step in our exercise is to prepare the brokers for the application. Our application will require the following:

- DB2 databases that contain customer account information
- MQSeries queues to handle the requests

12.9.1 Create the application databases

The databases required by our test application are:

- A DB2 database called ITSOCUST that contains customer profile information. This database will reside on the broker machine, since it is primarily used as an intermediate storage hold.
- A DB2 database called ITSOCHEC that contains customer checking account information. This database simulates a legacy banking database and is remote to the broker.
- A DB2 database called ITSOSAVI that contains customer savings account information. This database simulates a legacy banking database and is remote to the broker.

The mechanics of how these databases are created is not important to this discussion. If you are curious as to the process or database layout, see 9.3.1, “Application databases and tables” on page 185.

12.9.2 Define the required MQSeries queues

The next step is to define the MQSeries queues required for the application. Our application will need five queues:

- MQSI.PROF.REQUESTS
- MQSI.PROF.UPDATES
- MQSI.FAILURE
- MQSI.AUDIT
- MQSI.PROF.REPLIES

These queues will be local queues on the broker and will be shared within the cluster. Often, MQSeries administrators will choose to create queues with one name and then create an alias to that queue. The application uses the alias name, thus insulating the application from any real queue name changes that may need to take place. In our scenario, we will do this as well.

First make the real queues:

1. From the MQSeries Explorer right click **Queues** (under the broker queue manager) and select **New->Local Queue**.

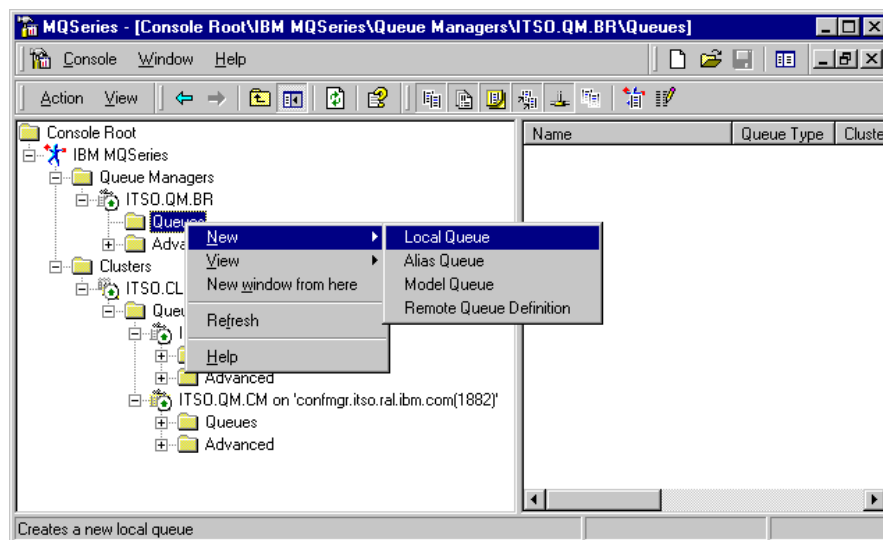


Figure 193. Create a new local queue

You will see a panel with several tabs available for defining the queue.

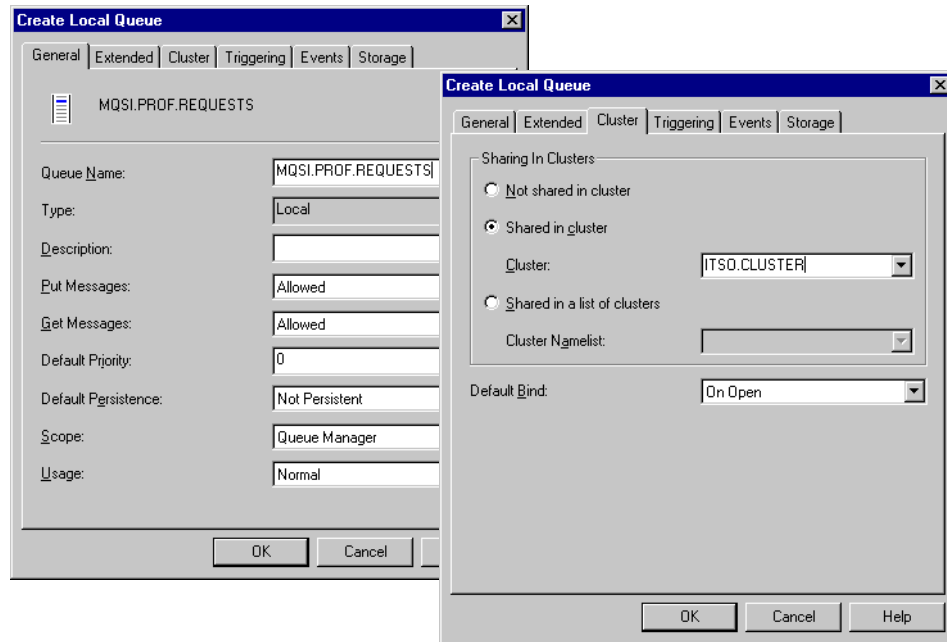


Figure 194. Creating a local queue

2. Enter the queue name on the General tab. On the Cluster tab click **Shared in cluster** and enter the cluster name. Take the defaults for the rest of the fields and click **OK**.
3. Repeat this until you have added all the required queues.

Next we will define alias queues.

1. Right click on Queues under the broker and select **New->Alias Queue**.
Enter the alias name and the base queue name. For example, the alias name for the first queue is ITSO.PROF.REQ.IN. The application will use this queue name, but the actual queue used will be the base queue, MQSI.PROF.REQUESTS.

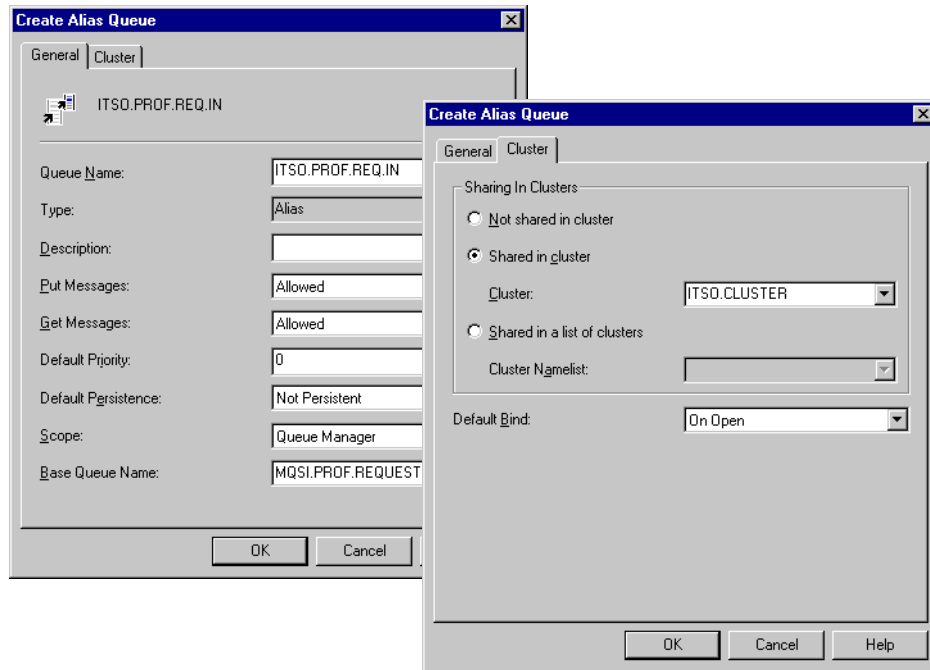


Figure 195. Defining an alias

2. We repeat this procedure, creating the following alias queues:

- ITSO.PROF.UPD.IN as an alias of MQSI.PROF.UPDATES
- ITSO.PROF.REPLY as an alias of MQSI.PROF.REPLIES

When done, both the alias and base queue names can be seen in the MQSeries Explorer.

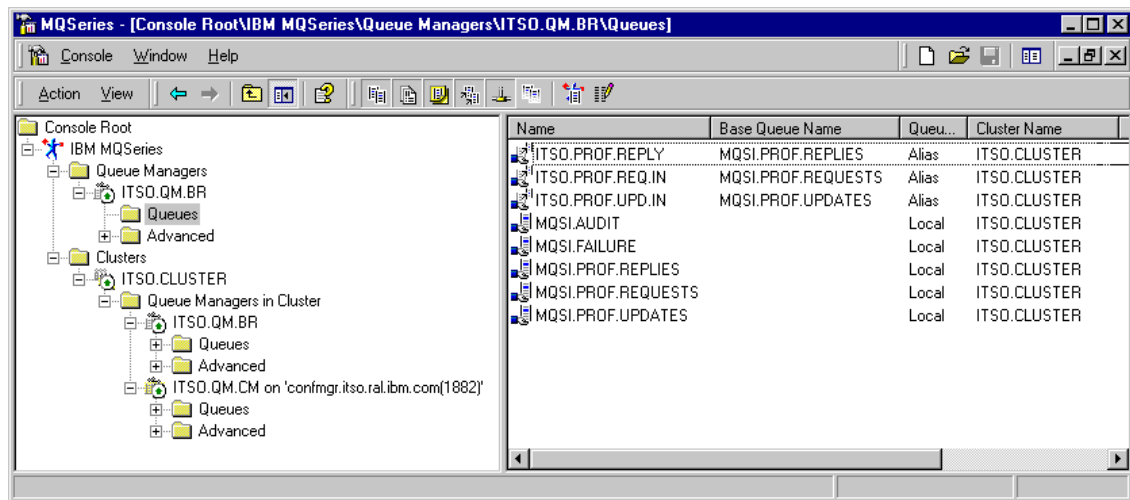


Figure 196. Alias and base queues

At this point, MQSeries and MQSI are ready to run the application. The only thing missing is the WebSphere piece. See Chapter 13, “WebSphere Application Server setup” on page 371 for information on setting up the rest of the application.

Chapter 13. WebSphere Application Server setup

For this project we used WebSphere Application Server Advanced Edition V3.5. When installing WebSphere, check for available fixes at <http://www.ibm.com/software/webservers/appserv/support.html>.

WebSphere 3.5 installs a basic library called the InfoCenter. The full library, which includes search facilities, must be downloaded from:

<http://www.ibm.com/software/webservers/appserv/library.html>

We highly recommend that you download the full InfoCenter and install it.

13.1 MQSeries SupportPac MA88

The Web application code developed for this project required Java classes provided by MQSeries to allow the Java application to put and retrieve messages from an MQSeries queue. These classes include:

- MQSeries classes for Java (MQ base Java)
- MQSeries classes for Java Message Service (JMS or MQ JMS)

They must be present on both the application development machine and on the WebSphere Application Server.

Note

Not all classes are available for all platforms. Before installing, see *MQSeries Using Java*, SG34-5456 It contains product and platform specific information about obtaining and using these classes.

This chapter deals specifically with the Windows NT and AIX platforms.

The Java classes are obtained by installing the MQSeries Product Extension MA88 (also called MQSeries SupportPac MA88). The installation allows you to choose from the following components:

- Java programming interface, which installs MQ base Java V5.1.2.
- JMS programming interface, which installs MQ JMS. It requires MQ base Java to be installed.
- Documentation.

For this example, we installed all three.

13.1.1 Classpath settings

MQ base Java supplies the following JAR files:

- com.ibm.mq.jar - provides support for all connection options
- com.ibm.mq.iiop.jar - provides support for Visibroker (IIOP) connections, Windows only
- com.ibm.mqbind.jar - provides support for bindings connection. This is not recommended for new application.

MQ JMS adds the following JAR files:

- com.ibm.mqjms.jar
- fscontext.jar
- jms.jar
- jndi.jar
- ldap.jar
- providerutil.jar

After installation, ensure that the JAR files supporting the functions you are using are placed in the CLASSPATH. Include the “lib” directory itself in order to access the properties files used by the base Java API. Include providerutil.jar and either fscontext.jar or ldap.jar if you need to access a JNDI namespace.

Our test application was written in a way that allowed us to switch between the two methods. The methods used to connect Java applications to MQSeries are discussed in 7.2.1, “WebSphere-to-MQSI connection options” on page 105.

These classes need to be available to the WebSphere application server at runtime. There are two approaches: you can install the SupportPac on the WebSphere machine, or you can manually copy the classes to a directory and add that directory to the WebSphere node’s dependent classpath.

For MQ base Java acting as an MQSeries client, the following dependent classpath should include com.ibm.mq.jar and the MQ Java lib directory.

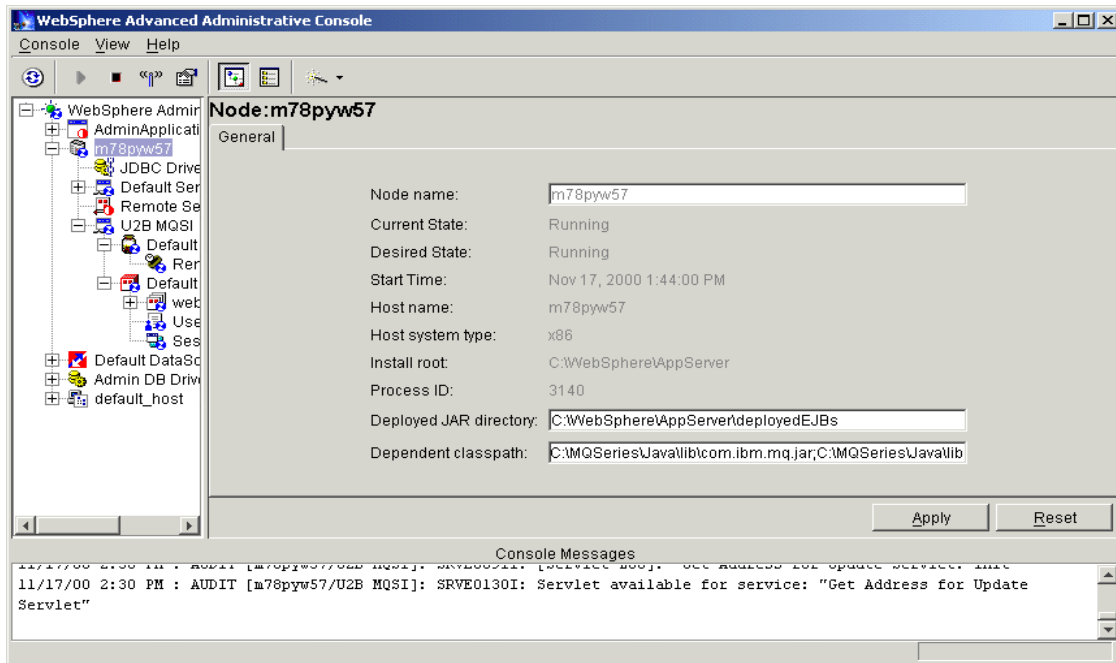


Figure 197. Dependent classpath for MQ base Java

If you are using JMS, using the WebSphere, we add the several JAR files to the dependent classpath. The new classpath is:

C:\MQSeries\Java\lib\com.ibm.mq.jar;C:\MQSeries\Java\lib\com.ibm.mqjms.jar;
C:\MQSeries\Java\lib\jms.jar;C:\MQSeries\Java\lib\jndi.jar;C:\MQSeries\Java\lib

The classes required may vary, depending on the programming options you are using. Check *MQSeries Using Java*, SG34-5456, for complete information on the settings for the classpath and other variables.

13.1.2 Configuring JMS

JMS has an administration tool that allows you to define the properties of four types of JMS objects and to store them within a JNDI namespace. JMS clients can retrieve these objects and use them.

13.1.2.1 Prepare the operating system

The administration tool can be invoked in interactive or batch mode. To use JMS classes and the JMS administration tool, you will need to set up the

CLASSPATH and PATH variables to include the MQ JMS classes and directories.

```
SET classpath=%classpath%;C:\MQSeries\Java\lib;  
C:\MQSeries\Java\lib\com.ibm.mqjms.jar; C:\MQSeries\Java\lib\com.ibm.mq.jar;  
C:\MQSeries\Java\lib\jms.jar;C:\MQSeries\Java\lib\jndi.jar;  
C:\MQSeries\Java\lib\providerutil.jar;C:\MQSeries\Java\lib\ldap.jar;  
C:\WebSphere\AppServer\lib\ujc.jar;C:\WebSphere\AppServer\lib\nc.jar  
  
SET path=C:\WebSphere\AppServer\jdk\bin;%path%;C:\MQSeries\Java\lib
```

In the next figure, you can see an example of calling the administration tool in batch mode. It will take a JMS configuration file (.config) and a script file (.scp file) containing administration commands as input.

```
java -DMQJMS_TRACE_LEVEL=ON -DMQJMS_LOG_DIR=. -DMQJMS_TRACE_DIR=. -DMQJMS_INSTALL_  
PATH=C:\MQSeries\Java\com.ibm.mq.jms.admin.JMSAdmin  
-cfg Banking_JMSAdmin.config < PDK_Banking_JMS_defs.scp
```

The configuration file used for our example is shown next.

```
# Specify which JNDI service provider is in use: LDAP, file system  
# context, or WebSphere's COSNaming repository (our choice).  
#  
INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory  
#  
# Specify the URL of the service provider's initial context: LDAP,  
# file system context, or WebSphere's COSNaming namespace (our choice)  
#  
PROVIDER_URL=iiop://  
#  
# Specifies whether security credentials are passed to the service #  
# provider by JNDI (LDAP only): 'none' (anonymous authentication),  
# 'simple', or 'CRAM-MD5'.  
#  
SECURITY_AUTHENTICATION=none
```

Figure 198. *Banking_JMSAdmin.config*

The configuration input for the administration tool is shown in Figure 199. Our configuration will define the properties for:

- An `MQQueueConnectionFactory` (QCF) object. This represents a factory object for creating connections in the point-to-point domain of JMS.
- Three `MQQueue` (Q) objects. These represent a destination for messages in the point-to-point domain of JMS.

```
def QCF(ITSO.QM.BR) qmanager(ITSO.QM.BR)
def Q(ITSO.PROF.REQ.IN) qmanager(ITSO.QM.BR) queue(ITSO.PROF.REQ.IN)
def Q(ITSO.PROF.REPLY) qmanager(ITSO.QM.BR) queue(ITSO.PROF.REPLY)
def Q(ITSO.PROF.UPD.IN) qmanager(ITSO.QM.BR) queue(ITSO.PROF.UPD.IN)
end
```

Figure 199. *PDK_Banking_JMS_defs.scp*

13.2 Deploying the application to WebSphere

The sample application consists of servlets, command beans, and JSP files that have been created either manually or using tools such as VisualAge for Java and WebSphere Studio. The files produced by these tools can be published using WebSphere Studio, or manually copied to the correct libraries. In our small test environment, we did this manually.

1. The first step is to create a directory structure to store the WebSphere application on the WebSphere application server. We will use “webbank” as the application directory name. We created three new directories under the `default_hosts` directory:
 - webbank
 - webbank\servlets
 - webbank\web

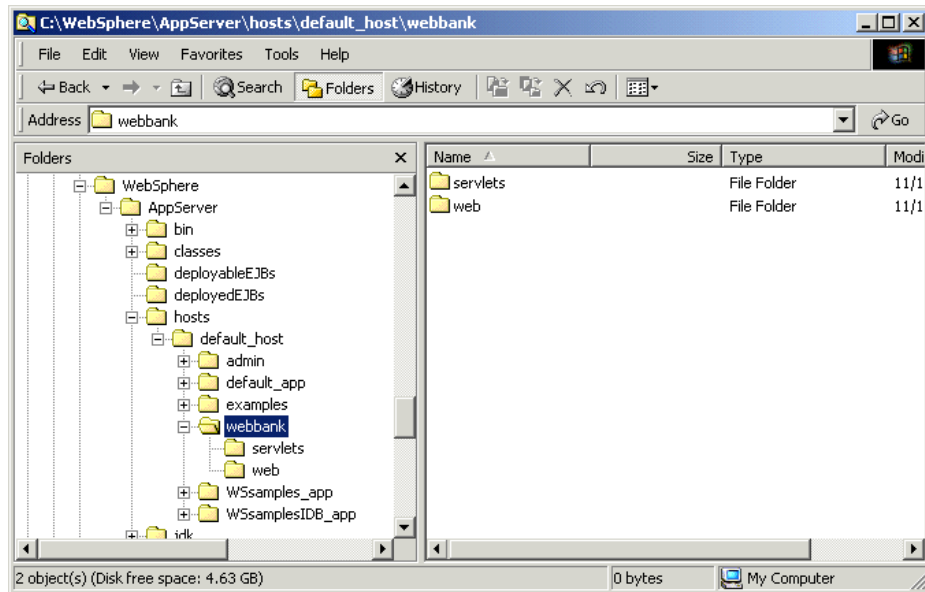


Figure 200. Application directory structure

2. The servlets and command beans were created using VisualAge for Java and combined into one JAR file, we called `pdj_ejb.jar`. This file is copied into the `webbank\servlets` directory.
3. The application reads text files for parameter settings, such as the MQSeries queue name and port. These properties files are placed in the `webbank\servlets` directory.

```
# GetCurrentProfile.properties
# This properties file is used by the GetCurrentProfile Servlet
# to initialize the GetCurrentProfileCommand
brokerHostname=9.24.104.62
brokerChannel=ITSO.SVRCONN
brokerPort=1881
brokerQueueManagerName=ITSO.QM.BR
brokerSetRequestQueueName=ITSO.ADDR.REQ.IN
brokerSetReplyQueueName=ITSO.ADDR.REPLY
messageTimeout=10000
# End of properties file
```

Figure 201. GetCurrentProfile properties


```
# UpdateProfile.properties
# This properties file is used by UpdateProfileServlet
# to initialize the UpdateProfileCommand
brokerHostname=broker1.itso.ral.ibm.com
brokerChannel=SYSTEM.DEF.SVRCONN
brokerPort=1881
brokerQueueManagerName=ITSO.QM.BR
brokerSetRequestQueueName=ITSO.PROF.UPD.IN
messageTimeout=10000
# End of properties file
```

Figure 202. UpdateProfile properties

4. Next, copy the JSP files (produced by WebSphere Studio) into the webbank\web directory.
5. We will use XMLConfig to define the Web application to the application server. We will use a command file to call the XMLConfig tool and another for the XML input. This will create definitions for the webbank Web application and its servlets.

The command file used to call the XMLConfig tool is shown in Figure 203.

```
@ECHO OFF
TITLE Creating Web Application
SET NODENAME=m78pyw57
SET WSAPPSERV=C:\WebSphere\AppServer
echo env vars set...
echo.
CALL %WSAPPSERV%\bin\xmlconfig -import webapp.xml -adminNodeName
%NODENAME% -substitute "NODE_NAME=%NODENAME%"
PAUSE
REM EXIT
```

Figure 203. Calling the xmlconfig tool

A section of the XML input is shown in Figure 204.

```

<?xml version="1.0"?>
<!DOCTYPE websphere-sa-config SYSTEM "$server_root$$dsep$bin$dsep$xmlconfig.dtd" >
websphere-sa-config
  <node name="$NODE_NAME$" action="locate">
    <application-server name="Default Server" action="locate">
      <servlet-engine name="Default Servlet Engine" action="locate">

        <web-application name="webbank" action="update">
          <description>User to Business Topology One</description>
<document-root>C:\WebSphere\AppServer\hosts\default_host\webbank\web</document-root>
          <classpath>
            <path value="C:/WebSphere/AppServer/hosts/default_host/webbank/servlets"/>
          </classpath>
          <error-page>/ErrorReporter</error-page>
          <filter-list/>
          <group-attributes/>
          <auto-reload>true</auto-reload>
          <reload-interval>3000</reload-interval>
          <enabled>true</enabled>
          <root-uri>default_host/webapp/webbank</root-uri>
          <shared-context>false</shared-context>
          <shared-context-jndi-name>SrdSrvltCtxHome</shared-context-jndi-name>
          <isclone>false</isclone>

        <servlet name="Error Reporting Facility" action="update">
          <description>Auto-Generated - Default error reporter servlet</description>
          <code>com.ibm.servlet.engine.webapp.DefaultErrorReporter</code>
          <init-parameters/>
          <load-at-startup>true</load-at-startup>
          <debug-mode>0</debug-mode>
          <uri-paths>
            <uri value="/ErrorReporter"/>
          </uri-paths>
          <enabled>true</enabled>
          <isclone>false</isclone>
        </servlet>
      </web-application>
    </application-server>
  </node>

```

Figure 204. XMLConfig tool input

13.3 Copy the DTDs to the operating system

The MQSeries client Java program needs access to the DTDs that describe the message format to be exchanged with MQSI. The servlet in our application used the default location for DTDs. On Windows NT, these DTDs

are copied to the WINNT\SYSTEM32 directory. This is the default directory for file access from servlets. The DTD needed is described in 9.5.2, “Customer profile update functional components” on page 191.

Appendix A. Rational Rose 2000e and VisualAge for Java

Rational Rose 2000e offers seamless integration between Rose and VisualAge for Java. This appendix discusses the process of taking a complete or partial Rational Rose model and generating Java code into a VisualAge for Java repository (forward engineering). Then it follows the steps involved in reverse engineering the Java code from a VisualAge for Java project into a Rational Rose model. These actions are made possible by the RoseLink Plugin Toggle.

A.1 Forward and reverse engineering with Rational Rose

From our modelling exercise, we can see that it would be of great use for architects and developers alike to be able to use the model as a source for producing the initial skeleton and outline code for development teams to work with. Then as the development goes through its various phases and the initial skeleton is developed into a more detailed implementation, it would be useful to be able to update the model to reflect the latest stage's code changes.

The process of generating source code from a Rose model, then modifying the code and updating the model with the changes, is what we refer to as "round tripping".

Forward and reverse engineering with Rational Rose allows us to do just that. Forward engineering allows Java source code to be generated from one or more classes, packages or components in a Rose model. Reverse engineering involves the analysis of Java source code, mapping it to Rose components and finally storing these components in a Rose model.

13.3.1 Integration with IBM VisualAge for Java

The RoseLink Plugin is the bridge between IBM VisualAge for Java and Rational Rose. Once activated, it enables the code generation of a VisualAge project from a Rose model and the reverse engineering of code from a VisualAge project to a Rose model.

Both forward and reversing engineering is initiated from Rose. It is important to note that Rose is actually doing all the work here, while VisualAge for Java is the passive partner.

When generating the code from the model, Rose first stages the code in a directory called `\Rose\java\YourProjectName`. VisualAge then imports the code into its repository.

Similarly when reverse engineering a VisualAge project into a Rose model, the project is first written to the \Rose\java directory, then the code from the file is reverse engineered into the specified model.

A.1.1 Rose to Java mapping

Although Rose models incorporate classes, forward engineering in Rose is component-centric. This means that Java code generation is based on the component specification as opposed to the class specification. It is still possible to generate Java code from a class. However, the class needs to be assigned to a valid Java component. Rose can create that component for you.

There is a mapping process that takes place between Rose model elements and the corresponding Java constructs when forward and reverse engineering. For example, Rose classes are mapped to Java classes and Rose components are mapped to .java files. Table 28 shows this mapping.

Table 28. Java to Rose mapping

Java element	Rose model element
Package	Package in the Component view.
import	-Dependencies between components and packages in the Component view (forward and reverse engineering). -Relationships between classes that are not located in the same package. In forward engineering, generates a Java import.
Compilation unit	Component (module specification) in the Component view.
Class	Class.
Interface	Class with stereotype of "interface".
Implements relationship	Realizes relationship between Java class (subclass) and Java interface (superclass).
Extends relationship	-Generalization relationship between Java classes. -Generalization relationship between Java interfaces.

Java element	Rose model element
Field	Attribute or supplier relationship between Rose classes. -Java instance variables have Static property value set to FALSE. -Java class variables have Static property set to TRUE.
Method	Operation.
Class modifiers	Properties on classes (e.g. Class.Final). Abstract modifier is an element of a Rose class specification.
Field modifiers	Properties on fields and roles (e.g. Role.Final, Attribute.Volatile, etc.). Static modifier is an element of a Rose role and attribute specification.
Method modifiers	Properties on operations (e.g. Operation.Static, Operation.Final, etc.)
{public, private, protected} access	Public, private, protected access.
package-level access	Implementation class.

A.2 Installation notes

If you are planning to use the Rose-VisualAge link, the correct order to install the products is to install VisualAge for Java first, then Rational Rose. If you have already installed Rational Rose, you can install VisualAge for Java later, but you will need to get the Rational Rose J to IBM VisualAge for Java link patch from the following Rational support site:

<http://www.rational.com/support/downloadcenter/>

A.3 Configuration

The following section covers the configuration of both VisualAge for Java and Rational Rose environments. What follows is an overview of the minimum configuration required to get the link to function.

A.3.1 VisualAge for Java configuration

Forward engineering means we are going to take a Rose model, generate code from it, and then have VisualAge import that code. It is not possible to create a new VisualAge for Java project from Rose, so the first step to take is to create the project in VisualAge to be used.

The next step is to start the RoseLink Plug-in itself. This is done from the VisualAge Quick Start menu.

1. Click **File --> Quick Start (F2)**.

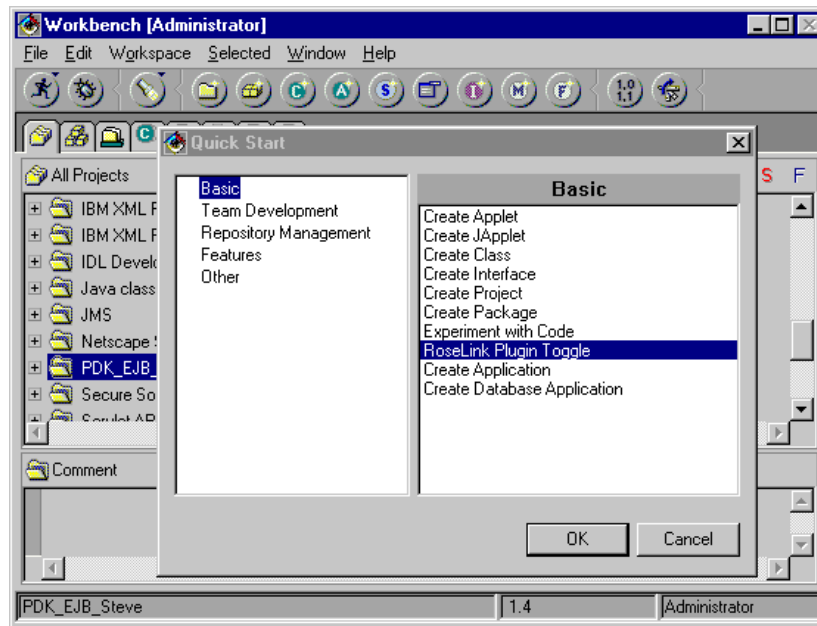


Figure 205. Start the Roselink Plugin Toggle

2. From the Quick Start dialog box choose **Basic** and **RoseLink Plugin Toggle** and click **OK**.
3. A message confirms that the link has successfully established.

A.3.2 Rational Rose configuration

There are also configuration tasks to do in Rose.

A.3.2.1 Language specification

There may be situations in a model where certain components are not Java based and should not be considered by the code generation process. In our modelling exercise we showed queue managers, command files, and queues as components. These were all ignored in the code generation process.

Rose is able to discern between Java and non-Java components by setting the language specification for the components. The language specification can be set for the whole model or for individual components.

Model Language Specification

To set the default language for the whole model:

1. Select **Tools -> Model Properties -> Edit...** or press F4.
2. Choose the **Notation** tab of the dialog box (see Figure 206).

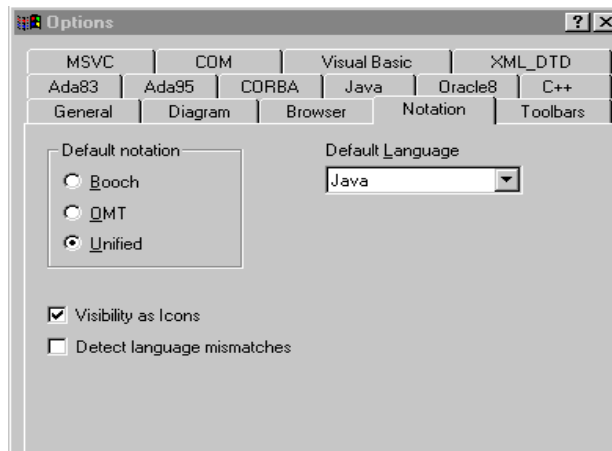


Figure 206. Set the model language specification

3. Select **Java** as the default language and click **OK**.

Note: This will not change the default language of a component if it was created before this property was set.

Component Language Specification

To set the language specification for an individual component:

1. Select the component from either the left-hand browser navigation pane or from an open diagram.
2. Right-click the component and select **Open Standard Specification** from the menu.

3. Select **Java** as the language from the drop-down box (see Figure 207).
4. Select **Ok** or **Apply** for the changes to take effect.



Figure 207. Component language specification

A.3.2.2 Syntax check

You can syntax check your components before attempting to generate Java source from them. This is optional since Rose automatically does a syntax check when it generates the Java code, but checking this earlier may save you some time in the long run.

A.3.2.3 Project specifications

For each model, the project properties that affect code generation can be customized. To customize the properties, from the Tools menu select **Java -> Project Specification**.

Class Path tab

Rose allows the classpath to be set for every model to be used when forward and reverse engineering code. Check to make sure the appropriate libraries are included.

Detail tab

Out of all the options listed in this section the only mandatory one is the Virtual Machine setting. In order to integrate Rose with VisualAge for Java the Virtual Machine specification needs to be set to **IBM**.

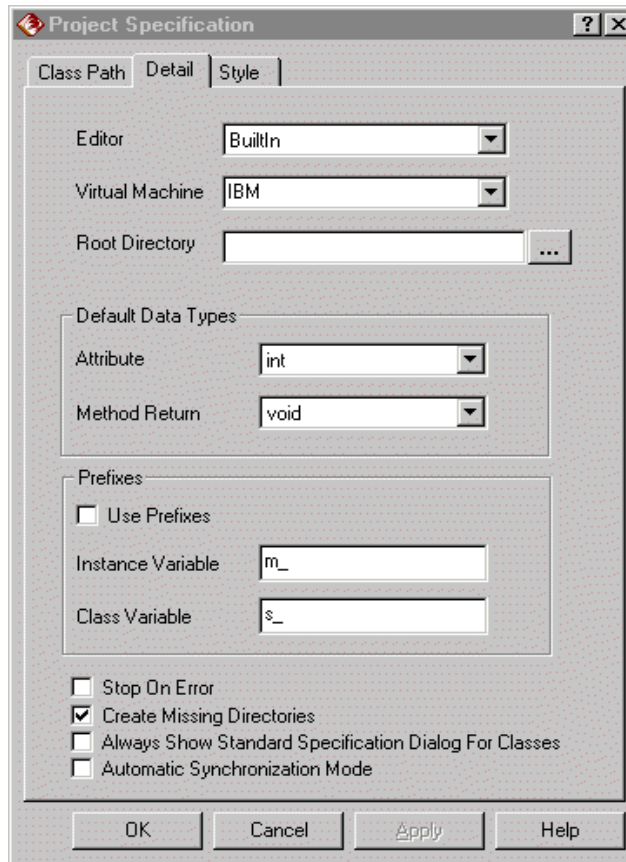


Figure 208. Project Specification options

The other options are:

- Stop On Error - When selected, code generation is halted at the first error encountered. By default this is not selected and code generation continues regardless of errors. All errors can be viewed in the Rose log window.
- Create Missing Directories - This is the default behavior and will create any undefined directories that are referenced as packages in the model.

- Automatic Synchronization mode - This option automatically updates the code every time a Java element is created, deleted, renamed or modified. By default this behavior is not enabled.

13.4 Linking a Rose model to a VisualAge for Java project

Having configured both the VisualAge and Rose environments and started the RoseLink Plugin Toggle, we can now link the model to the project. To do this:

1. From the Tools menu select **Java ->IBM VisualAge for Java Project**.
2. A dialog box entitled VisualAge Link Settings appears showing a list of all the projects within VisualAge. Select the appropriate project and click **OK**. Rose creates a new model property setting called VAJavaProject where it sets the project name specified.

The link between Rose and VisualAge is now active.

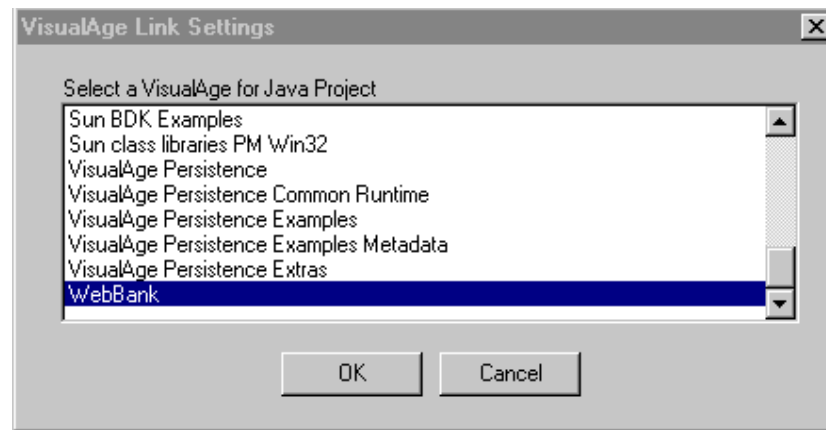


Figure 209. Linking the Rose model to the VisualAge project

A.4 Forward engineering with Rose

In order to generate code it is necessary for classes defined in the Rose model to be assigned to Java components in the component view. In Table 28 on page 382, the compilation unit in VisualAge (a .java file) is mapped to a component in Rose. From this we can see that the component view models the physical file structure of the code generated.

There are two ways of generating code from within Rose, either from classes or from the components.

A.4.1 Generating code from classes

Generating the code from classes relies on Rose creating the components from the class diagrams. The components are created automatically in the code generation process. For this to happen Rose makes the following assumptions:

- Every class directly maps to one component and therefore one .java file is generated.
- If the class is within a package, Rose will create a package in the components view and the .java file will reside in a directory of the same name as the package.

The steps involved in generating the code are:

1. Select one or more classes from a Class Diagram or from the View pane.
2. Right click and select **Java ->Generate Java**.

A timer box will appear, closely followed by a message box confirming the success of the operation.

A.4.2 Generating code from components

The drawback with the previous method is that it may be desirable to have more than one class in a .java file. This can be achieved by manually creating the components in the component view and then assigning the classes to the components by hand.

The manner in which classes are assigned to a component is:

1. Right-click the component from the Component Diagram or the View pane.
2. Select **Open Standard Specification...** and click the **Realizes** tab.
3. From the classes listed, select those required.
4. Right-click to bring up a menu and select **Assign**. This will result in a red tick being displayed on the assigned classes.

The same result can be achieved by following the same steps but for classes, and assigning components to them.

Once assigned, code generation can be started. To do this:

1. Select one or more components from a Component Diagram or from the View Pane.
2. Right-click and select **Java ->Generate Java**.

A timer box will appear, closely followed by a message box confirming the success of the operation.

A.5 Reverse engineering

The reverse engineering process analyzes Java source code, maps it to Rose classes and components, and then stores them in a Rose model.

The source can be either:

- Java source code (.java files)
- Java bytecode (.class files)
- .zip, .cab, or.jar files

There are two ways in which reverse engineering can be initiated.

1. Drag and drop files into a Rose class diagram or component diagram. This will automatically decompress any .zip, .cab or .jar files. However, it will not recompress these files if code is subsequently generated from these classes.
2. Use the **Tools->Java->Reverse Engineer** option or right-click the classes or components and use the shortcut menu.

Either way, the reverse engineering dialog box shown in Figure 210 is displayed.

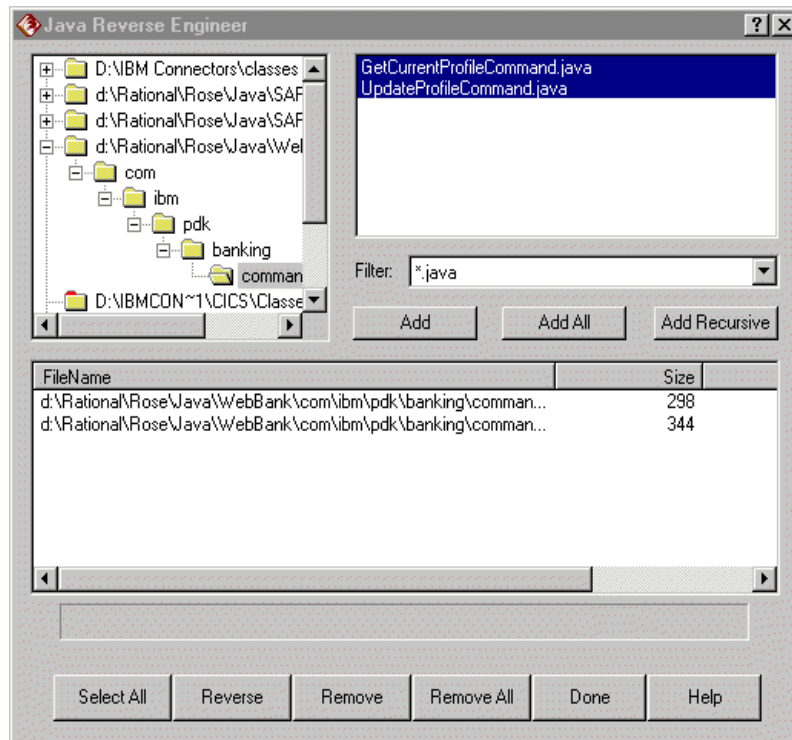


Figure 210. Reverse engineering control

From this dialog box the first selection is the directory that holds the source and the second is the source itself, either .java or .class files. The selected files are then added to the bottom box and from there reverse engineering can begin on the chosen files. After the process is complete the components are visible in the View pane but will not be shown in the component diagrams. They will need to be added to any diagrams manually.

It is also possible to view and edit the Java source code behind the component. This can be done with either the built-in Rose editor or any other user-specified tool. If the AutoSynchronization feature is set, the model will be automatically updated as changes are saved. If not, then the reverse engineering process needs to be done again.

If changes are made in this way (by reverse engineering), it will be necessary to forward engineer again to make sure that the code in the VisualAge repository is also up to date.

Appendix B. Sample code

In this appendix we have placed code listings that were referenced in other parts of the book.

B.1 GetCurrentProfileCommandMQJava: retrieveProfile() method

```
/**
 */
public String retrieveProfile() {
    // *** MQ Variables ***
    // Define names of MQ objects
    //String requestQueueName = "ITSO.ADDR.REQ.IN";
    //String replyQueueName = "ITSO.ADDR.REPLY";
    // Actual MQ Java object refs.
    com.ibm.mq.MQQueueManager qMgr = null;
    com.ibm.mq.MQQueue requestQueue = null;
    com.ibm.mq.MQQueue replyQueue = null;
    // Holds reply queue name for use in MQ message options etc.
    int openOptions;
    int getOptions;
    byte sentmessage[];

    MQEnvironment.hostname = hostname;
    MQEnvironment.channel = channel;
    MQEnvironment.port = port;

    writeToLog("port = " + port);
    writeToLog("hostname = " + hostname);
    writeToLog("channel = " + channel);
    writeToLog("queueManagerName = " + queueManagerName);
    writeToLog("requestQueueName = " + requestQueueName);
    writeToLog("replyQueueName = " + replyQueueName);

    try {
        /*
         * Connect to queue manager
         */
        qMgr = new MQQueueManager(queueManagerName);
        writeToLog("Connected to QManager " + queueManagerName);
    }
```

Figure 211. *retrieveProfile()*

```

// If the name of the request queue is the same as the reply queue...
if(requestQueueName.equals(replyQueueName)){
    openOptions = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_OUTPUT;
}
else{
    openOptions = MQC.MQOO_OUTPUT ;           // Open queue to perform MQPUTs
}
openOptions = MQC.MQOO_INPUT_AS_Q_DEF | MQC.MQOO_OUTPUT ;

/* Request queue */
requestQueue = qMgr.accessQueue(requestQueueName, openOptions,
                                null,           // default q manager
                                null,           // no dynamic q name
                                null);          // no alternate user id
writeToLog("Connected to requestQueue " + requestQueue);
/*
 * Create a request message and set its properties
 */
MQMessage requestMessage = new MQMessage();
requestMessage.replyToQueueName = replyQueueName;
requestMessage.replyToQueueManagerName = queueManagerName;
writeToLog("MQMessage created and properties set");
requestMessage.writeString(getXMLrequestMessage());
writeToLog("MQMessage content written as " + getXMLrequestMessage());

// Put message to queue with options
MQPutMessageOptions pmo = new MQPutMessageOptions();
requestQueue.put(requestMessage, pmo );
writeToLog("MQMessage written to queue");

/* Reply message */
replyQueue = qMgr.accessQueue(replyQueueName,
                               openOptions,
                               null,           // default q manager
                               null,           // no dynamic q name
                               null);          // no alternate user id
writeToLog("Connected to replyQueue " + replyQueue);

```

Figure 212. retrieveProfile()

```

/*
 * Create the reply message and set its properties
 * including the ID of the original request message
 */
MQMessage replyMessage = new MQMessage();
writeToLog("reply message created");
replyMessage.correlationId = requestMessage.messageId;
writeToLog("ID for sent message = " + requestMessage.messageId.toString());
writeToLog("Correlation ID stored = " + replyMessage.correlationId.toString());
/*
 * Put message to queue with options
 */
MQGetMessageOptions gmo = new MQGetMessageOptions(); // accept the defaults
                                                    // same as MQGMO_DEFAULT

gmo.options = MQC.MQGMO_WAIT;
gmo.waitInterval = 2000;
writeToLog("Waiting for reply...");
replyQueue.get(replyMessage, gmo);
writeToLog("Reply received");

/*
 * Extract message data
 */
int msglen = replyMessage.getMessageLength();
writeToLog("reply is " + msglen + " long");

String msgText = replyMessage.readString(msglen);
writeToLog("message is = '" + msgText + "'");

setMessage("GetCurrentProfileCommandMQJava completed successfully");

// Store the message data
return msgText;
}catch (MQException ex){
    writeToLog("MQException occurred : Completion code " +
                ex.completionCode + "\n>MQStatus: Reason code " + ex.reasonCode);
}

```

Figure 213. retrieveProfile()

```

        //throw new CommandException("MQException occurred during MQ operations");
//Detail is " + e.getMessage);
        setMessage("ERROR - GetCurrentProfileCommandMQJava MQException ");
        return "ERROR";
    }catch (Exception e){
        writeToLog("Exception occurred - " + e.toString());
        //throw new CommandException("General Exception during MQ operations");
//Detail is " + e.getMessage);
        setMessage("ERROR - GetCurrentProfileCommandMQJava:Exception ");
        return "ERROR";
    }finally{
        /*
         * Tidy up
         */
        try {
            // Close the queue
            if(requestQueue != null ){
                writeToLog("MQQueue closed");
                requestQueue.close();
            }
            // Disconnect from the queue manager
            if(requestQueue != null ){
                writeToLog("disconnected from QueueManager ");
                qMgr.disconnect();
            }
        }catch(MQException mqe){
            writeToLog("EXCEPTION closing queue or disconnecting from QueueManager ");
            setMessage("ERROR - GetCurrentProfileCommandMQJava: RemoteException");
            return "ERROR";
        }
    }
}
}

```

Figure 214. *retrieveProfile()*

B.2 GetCurrentProfileCommandJMS: retrieveProfile() method

```
/**
 * Insert the method's description here.
 * Creation date: (14/09/00 19:59:25)
 */
public String retrieveProfile() {
    /*
     * Declare local JMS variables
     */
    QueueConnection connection = null;
    QueueSession session = null;
    QueueSender messageProducer = null;
    QueueReceiver messageConsumer = null;
    TextMessage replyMsg = null;
    Queue destQueue=null;
    Queue replyQueue=null;

    try {

        // Obtain context to look up JMS admin objects (websphere cos naming)
        //messagingContext = ContextHelper.createWebsphereContext();
        InitialContext messagingContext = new InitialContext();
        System.out.println("Context OK: create connection factory " + messagingContext);
        MQQueueConnectionFactory cf = (MQQueueConnectionFactory)
messagingContext.lookup(queueManagerName);
        writeToLog("OK 'ConnectionFactory is " + cf);

        writeToLog("Going to setupClientConnection " + hostname);
        cf.setTransportType(JMSC.MQJMS_TP_CLIENT_MQ_TCPIP);
        cf.setHostName(hostname);
        cf.setQueueManager(queueManagerName);
        cf.setChannel(channel);
        cf.setPort(port);
        writeToLog("done setupClientConnection " + hostname);
        writeToLog("About to create Connection ");
        connection = cf.createQueueConnection();
        writeToLog("Connection is " + connection);
        connection.start();
        writeToLog("Connection started OK");
    }
}
```

Figure 215. *retrieveProfile()*

```

//*****
//*****INIT ENDS - MESSAGE SEND BEGINS*****
//*****

//get JMS admin objects from cos naming provider
destQueue = (Queue)messagingContext.lookup(requestQueueName);
replyQueue = (Queue)messagingContext.lookup(replyQueueName);
session = connection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
writeToLog("destQueue = " + destQueue);
writeToLog("replyQueue = " + replyQueue);
writeToLog("session = " + session);
// create a producer of messages
messageProducer = session.createSender(destQueue);
writeToLog("messageProducer = " + messageProducer);

// create xml message string
TextMessage msg = session.createTextMessage();
msg.setText(getXMLrequestMessage());
writeToLog("msg text set to = " + getXMLrequestMessage());

// set the reply to queue for the message
msg.setJMSReplyTo(replyQueue);

// put ('send') message to queue
messageProducer.send(msg);
String msgID = msg.getJMSMessageID();
writeToLog("msg sent with ID = " + msgID);

/*****
 * Get reply message
 *****/
// Create a consumer for the above message,
// Use JMSCorrelationID as messageSelector
messageConsumer = session.createReceiver(replyQueue, "JMSCorrelationID = " + ""
+ msgID + "");

```

retrieveProfile()

```

//wait for reply
writeToLog("Waiting for reply...");
replyMsg = (TextMessage)messageConsumer.receive(messageTimeout);
writeToLog("reply received as = " + replyMsg);

//if null returned on timeout
if(replyMsg == null){
    throw new CommandException("Timeout waiting for reply message");
}
setMessage("GetCurrentProfileCommandJMS completed successfully");

// store the XML string
return replyMsg.getText();

} catch (Throwable t) {
    writeToLog("Exception JMS message operations clean up was " + t);
    t.printStackTrace();
    setMessage("ERROR - GetCurrentProfileCommandJMS encountered an Exception");
    return "No data available";
}

// clean up JMS objects
finally {
    writeToLog("clean up stage");

    try {
        if (messageProducer != null) {
            messageProducer.close();
        }
        if (messageConsumer != null) {
            messageConsumer.close();
        }
        if (session != null) {
            session.close();
        }
    } catch (Throwable t) {
        writeToLog("Exception during clean up was " + t);
        t.printStackTrace();
        setMessage("ERROR - GetCurrentProfileCommandJMS encountered an Exception");
        return "No data available";
    }
}
}

```

Figure 216. *retrieveProfile()*

Appendix C. Special notices

This publication is intended to help IT architects and IT specialists in the design and deployment of e-business applications. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere. See the PUBLICATIONS section of the IBM Programming Announcement for IBM WebSphere Application Server V 3.5 Advanced Edition for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers

attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

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

IBM ®	Redbooks
AIX	Redbooks Logo 
AS/400	S/390
CICS	SecureWay
DB2	SP
Lotus	SP1
Lotus Notes	Storyboard
Domino	SupportPac
MQSeries	System/390
OS/2	VisualAge
OS/390	WebSphere
OS/400	Wizard
RETAIN	

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Københavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

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

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

Appendix D. Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

D.1 IBM Redbooks

For information on ordering these publications see “How to get IBM Redbooks” on page 411.

- *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864
- *User-to-Business Patterns Using WebSphere Enterprise Edition*, SG24-6151
- *Business Integration Solutions with MQSeries Integrator*, SG24-6154
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application*, SG24-5754
- *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153
- *CCF Connectors and Database Connections Using WebSphere Advanced Edition*, SG24-5514
- *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104
- *MQSeries Version 5.1 Administration and Programming Examples*, SG24-5849

D.2 IBM Redbooks collections

Redbooks are also available on the following CD-ROMs. Click the CD-ROMs button at ibm.com/redbooks for information about all the CD-ROMs offered, updates and formats.

CD-ROM Title	Collection Kit Number
IBM System/390 Redbooks Collection	SK2T-2177
IBM Networking Redbooks Collection	SK2T-6022
IBM Transaction Processing and Data Management Redbooks Collection	SK2T-8038
IBM Lotus Redbooks Collection	SK2T-8039
Tivoli Redbooks Collection	SK2T-8044

CD-ROM Title	Collection Kit Number
IBM AS/400 Redbooks Collection	SK2T-2849
IBM Netfinity Hardware and Software Redbooks Collection	SK2T-8046
IBM RS/6000 Redbooks Collection	SK2T-8043
IBM Application Development Redbooks Collection	SK2T-8037
IBM Enterprise Storage and Systems Management Solutions	SK3T-3694

D.3 Other resources

These publications are also relevant as further information sources:

- *MQSeries Primer*, found at <http://www.redbooks.ibm.com>
- *WebSphere's Remote OSE*, a redpaper available at <http://www.redbooks.ibm.com>
- *Choosing the Right EJB Type: Some Design Criteria*, technical article available at <http://www7.software.ibm.com/>
- *Building Business Solutions with WebSphere*, SC09-4432, available at <http://www.ibm.com/software/webservers/appserv/library.html>
- Flanagan, David, Jim Farley, William Crawford and Kris Magnusson, *Java Enterprise in a Nutshell*, O'Reilly & Associates, Inc. 1999, ISBN 1565924835
- Erich Gamma et al, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995, ISBN: 0201633612
- Maruyama, Hiroshi, Kent Tamura and Naohiko Uramoto, *XML and Java: Developing Web Applications*, Addison-Wesley, 1999, ISBN 0201485435
- Flanagan, David, *JavaScript: The Definitive Guide*, Third Edition, O'Reilly & Associates, Inc., 1998, ISBN 1565923928
- Nagaratnam, Nataraj et al. 2000. *Security Overview of IBM WebSphere Standard/Advance 3.02*, IBM white paper, available at: <http://www.ibm.com/software/webservers/appserv/whitepapers.html>
- *Developing Dynamic Web Sites Using the WebSphere Application Server* by Shane Claussen and Mike Conner, available at: <http://service2.boulder.ibm.com/devcon/news0399/artpage2.htm>
- *IBM Application Framework for e-business*: white papers available at: <http://www.ibm.com/software/ebusiness/>
- *Designing e-business Solutions for Performance*, white paper by Maggie Archibald and Mike Schlosser, available at: <http://www.ibm.com/software/developer/library/patterns/performance.html>

These IBM MQSeries and MQSeries Integrator publications are available at <http://www.ibm.com/software/ts/mqseries/library/>:

- *MQSeries Using Java*, SG34-5456
- *MQSeries MQSC Command Reference*, SC33-1369
- *MQSeries Programmable System Management*, SC33-1482
- *MQSeries Administration Interface Programming Guide and Reference*, SC34-5390
- *MQSeries System Administration*, SC33-1873
- *MQSeries: Queue Manager Clusters*, SC34-5349
- *MQSeries Integrator Version 2.0 Technical White Paper*
- *MQSeries Integrator Programming Guide Version 2.0*, SC34-5603
- *MQSeries Integrator Using the Control Center*, SC34-5602
- *MQSeries Integrator Administration Guide*, SC34-5792

D.4 Referenced Web sites

These Web sites are also relevant as further information sources:

- <http://www.ibm.com/software/ebusiness/> For information on the IBM Application Framework for e-business
- <http://www.ecma.ch/> ECMA home page
- <http://www.javasoft.com/products> To learn more about Java technology
- <http://www.ibm.com/redbooks> IBM Redbooks home page
- <http://www7.software.ibm.com/> VisualAge Developer Domain home page
- <http://www.ibm.com/software/ts/mqseries/> IBM MQSeries family home page
- <http://www.ibm.com/software/developer/> PartnerWorld for Developers
- <http://www.rational.com> Rational home page
- <http://www.microsoft.com> Microsoft home page
- <http://www.messageq.com> ebizQ.net online publication dedicated to information about creating and managing messaging infrastructures
- <http://www.ibm.com/software/web servers/appserv/> WebSphere Application Server home page

- <http://www.ibm.com/software/data/pubs> IBM database and data management publications
- <http://support.microsoft.com/directory> Microsoft product support services

Appendix E. Using the additional material

This redbook also contains additional material on the Internet. See the appropriate section below for instructions on using or downloading this material.

E.1 Locating the additional material on the Internet

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246160>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number.

E.2 Using the Web material

The additional Web material that accompanies this redbook includes the following:

<i>File name</i>	<i>Description</i>
6160scripts.zip	Zipped file containing MQSI configuration scripts

E.2.1 System requirements for downloading the Web material

The following system configuration is recommended for downloading the additional Web material.

Hard disk space:	19 KB
Operating System:	Windows NT

E.2.2 How to use the Web material

Create a subdirectory (folder) on your workstation, copy, and unzip the contents of the Web material into this folder. The MQSI configuration scripts can be used as examples for your own configuration scripts. They are intended to serve as examples only and must be modified to suit your own environment. Read the readme.pdf document first for information on how to use the files.

How to get IBM Redbooks

This section explains how both customers and IBM employees can find out about IBM Redbooks, redpieces, and CD-ROMs. A form for ordering books and CD-ROMs by fax or e-mail is also provided.

- **Redbooks Web Site** ibm.com/redbooks

Search for, view, download, or order hardcopy/CD-ROM Redbooks from the Redbooks Web site. Also read redpieces and download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

- **E-mail Orders**

Send orders by e-mail including information from the IBM Redbooks fax order form to:

	e-mail address
In United States or Canada	pubscan@us.ibm.com
Outside North America	Contact information is in the "How to Order" section at this site: http://www.elink.ibm.link.ibm.com/pbl/pbl

- **Telephone Orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	Country coordinator phone number is in the "How to Order" section at this site: http://www.elink.ibm.link.ibm.com/pbl/pbl

- **Fax Orders**

United States (toll free)	1-800-445-9269
Canada	1-403-267-4455
Outside North America	Fax phone number is in the "How to Order" section at this site: http://www.elink.ibm.link.ibm.com/pbl/pbl

This information was current at the time of publication, but is continually subject to change. The latest information may be found at the Redbooks Web site.

IBM Intranet for Employees

IBM employees may register for information on workshops, residencies, and Redbooks by accessing the IBM Intranet Web site at <http://w3.itso.ibm.com/> and clicking the ITSO Mailing List button. Look in the Materials repository for workshops, presentations, papers, and Web pages developed and written by the ITSO technical professionals; click the Additional Materials button. Employees may access MyNews at <http://w3.ibm.com/> for redbook, residency, and workshop announcements.

IBM Redbooks fax order form

Please send me the following:

Title	Order Number	Quantity

First name	Last name
------------	-----------

Company

Address

City	Postal code	Country
------	-------------	---------

Telephone number	Telefax number	VAT number
------------------	----------------	------------

<input type="checkbox"/> Invoice to customer number	
---	--

<input type="checkbox"/> Credit card number	
---	--

Credit card expiration date	Card issued to	Signature
-----------------------------	----------------	-----------

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

Symbols

/etc/syslog.conf 278

A

ACL 278
activity diagram 162, 167
actor 153, 154
administrative agent 268
alias 110, 112, 269, 331, 354, 367, 368, 369
AMQ6767 264
analysis diagram 165, 166
applet 36, 37, 40, 41, 42, 50
Application Framework 35, 36, 37, 42, 48
Application Messaging Interface (AMI) 6, 109
application server 22, 25, 27, 28, 180, 181
ARCHIVE 41
asynchronous 16, 51, 111, 115, 183, 364

B

Baan 26
back-end application tier 15, 16
BackoutCount 295
BCLIENTUSER 297
Bean Managed Persistence (BMP) 47
bindings 51, 101, 106, 275
bindings mode 106
BLOB 282
BMQPSTOPOLOGY 297
BOTHRESH 295
boundary class 167
BPUBLISHERS 297
BRETAINEDPUBS 297
broker database 113
broker instance name 284
broker persistent store 323
BSUBSCRIPTIONS 297
build cycle 141, 142
BUSERCONTEXT 297

C

C structures 283
CAB 41
cache 59, 114, 147, 184, 185, 196, 303
CARDINALITY 132, 133

cascading style sheets (CSS) 39
CAST 133
catcher servlet 57
category 194
CCF 48, 49, 50
CGI 43
channel events 258
check in 281, 282, 363
check out 281, 357
CICS 26, 48, 49
circular logging 262, 263, 264
class diagram 164, 165, 167, 169, 172
class model 164
classForName 58
CLASSPATH 372, 374
client mode 107, 108
cloning 30
CLUSRCVR 250, 251
CLUSSDR 250, 251
cluster 109, 111, 249, 250, 252, 253, 268, 272, 310, 313, 319, 324, 334, 336, 337, 338, 339, 340, 341, 342, 343, 344, 347, 348, 349, 350, 351, 368
cluster receiver 250, 251
cluster sender 109, 250, 251, 269, 272
clustering 29
COBOL copybooks 283
collective 285
com.ibm.mq.iiop.jar 372
com.ibm.mq.jar 372, 373
com.ibm.mqbind.jar 372
com.ibm.mqjms.jar 372, 373, 374
com.ibm.record.ctypes 50
com.ibm.record.util 50
Command Assistant 274, 276, 299, 328, 329, 334, 352
command bean 105
command caching 58
Command interface 59
command package 1, 55, 57, 75, 95, 165
command pattern 55
command shipping 57, 60
CommandTarget 63, 64, 67, 68, 69, 70, 71, 72, 75, 77
CommandTargetName 75, 78
Common Connector Framework (CCF) 48
Communication 49
CompensableCommand 59, 61

- component diagram 177
- component model 176, 177
- compute node 120, 121, 122, 123, 127, 132, 134, 138, 196, 198, 199, 205, 213, 220, 230
- Configuration Manager 110, 113, 265, 266, 268, 269, 270, 271, 273, 275, 293, 296, 297, 299, 312, 319, 320, 321, 322, 324, 330, 331, 332, 334, 335, 336, 337, 347, 351, 356, 357, 363
- ConnectionSpec 49
- connector 35, 48, 49, 50
- container 46, 47, 110, 112
- Container Managed Persistence (CMP) 47, 60
- Control Center 109, 117, 119, 120, 123, 143, 193, 195, 265, 269, 272, 273, 278, 282, 283, 286, 288, 290, 291, 298, 299, 314, 316, 320, 321, 322, 330, 355, 356, 357, 358, 360
- control class 167
- controller 268
- CORBA 53
- crtmqm 254, 351
- Custom Wire Format (CWF) 119
- customer relationship management (CRM) 14

D

- database node 127, 128, 129, 201, 202, 203, 224
- DataDelete 127
- DataInsert 127
- DataUpdate 127
- db2cli 347
- DBMS 46, 266
- DCOM 256
- DECLARE 217
- demilitarized zone 18, 19
- deployable message flows 114
- deployment diagram 179, 180
- deployment model 141, 152, 179
- DHTML 18, 36, 37, 39
- Directory and security services node 19
- directory naming service 106
- DLQ 326, 347
- DMZ 19, 20, 21, 22, 26, 306, 308
- docmcngf 256
- DOCTYPE 231
- Document Object Model (DOM) 39
- Document Type Definition (DTD) 39
 - See also DTD
- domain firewall 19, 21, 23, 26, 28
- Domain Name Service (DNS) node 18

- DSPMQAUT 255
- DTD 116, 187, 189, 190, 191, 192, 378
- dynamic HTML (DHTML) 37, 39

E

- ECMA 40
- ECMA-262 40
- ECMAScript 40
- EJB 46, 47, 48, 53, 105, 110, 112, 117, 308
- eMarketPlace 3
- e-Marketplaces 148
- Encina 48
- Encina DE-Light 49
- endmqm 264, 295, 296, 297, 298, 299
- Enterprise Access Builder (EAB) 49
- Enterprise Application Integration (EAI) 51
- Enterprise Information System (EIS) 50
- Enterprise JavaBeans 35, 43, 48
- Enterprise-Out 12
- entity bean 47, 57, 60, 63, 67, 68, 69, 70, 79
- entity class 167
- ESQL 104, 115, 120, 122, 123, 124, 125, 126, 127, 129, 130, 132, 134, 137, 196, 197, 198, 199, 202, 203, 205, 206, 209, 210, 211, 213, 215, 217, 219, 222, 224, 225, 226, 227
- ESS 5
- execute() 56, 57, 61, 62, 77, 78, 80, 81, 86, 88, 90, 91, 94
- execution group 287
- export 288, 360
- Extensible Markup Language 119
- eXtensible Stylesheet Language 39
 - See also XSL 39
- External Call Interface (ECI) 49
- External Presentation Interface (EPI) 49

F

- filter node 129, 130, 131, 133, 137, 201, 208, 210, 220, 223
- firewall 19, 21, 23, 25, 26
- FORM 41
- forward engineer 143, 145
- fscontext.jar 372
- full repository queue manager 250

G

- generic XML messages 282

get authority 255
getCommandTarget() 62, 75, 76, 79
getCommandTargetName() 62, 75

H

hasOutputProperties() 62, 69, 70
horizontal scalability 21
Host On-Demand 13, 49
HTML 35, 37, 38, 39, 40, 41, 42, 44
HttpServletRequest 57, 167
HttpServletResponse 57

I

IDX1543 263
IIOP 53, 372
import 288, 289, 360
IMS 26, 48, 49
INETD 247
InfoCenter 371
information architecture 147
InputBody 125
InputRoot 123, 124
InputTerminal 118, 136, 196
instrumentation 258
INT 133
integration server 17, 20, 21, 22, 26, 27, 28, 48
interaction diagram 172, 175
InteractionSpec 49
IPSEC 18
isReadyToCallExecute() 61, 66, 83, 84, 85, 90, 91, 95
iTV 148

J

Java applet 40
Java Message Service (JMS) 53
Java Naming and Directory Interface (JNDI) 53
Java Record Library 50
Java Transaction API (JTA) 53
JavaBeans 44
JavaScript 36, 37, 38, 40, 152
JavaServer Pages 35, 43, 44, 143, 144, 152
JDBC 35, 45, 46, 273
JDK 41
JMS 7, 53, 87, 94, 99, 100, 101, 105, 106, 108, 109, 181, 231, 303, 308, 371, 373, 374, 375
jms.jar 372, 373, 374

JMSCorrelationID 102
JNDI 53, 100, 372, 373
jndi.jar 372, 373, 374
JNI 51
JScript 39, 40
JSP 44, 56, 145, 167, 169, 170, 171, 172, 178, 180, 181, 308
JTA 53
JVM 30, 31, 43, 59, 68

K

kiosk 148

L

LDAP 25, 26, 53, 144
ldap.jar 372
LENGTH 217
linear logging 262, 263
listener 247, 256, 328, 334, 336, 337, 341, 348, 351
load balancing 112
local queue 110, 250, 308
Logic control nodes 129
Logical View 162
lookup() 100
loop 131, 137, 191, 192, 193, 221
Lotus Domino 48
Lotus Notes 26

M

macro design 141, 147, 153
Macromedia Flash 38
MCAUSER 256
message broker
 instance name 267
message domain 282
message flow 53, 103, 104, 114
message flow category 195
message flow node 136, 137, 138, 228, 229, 230, 231, 233, 284
Message Queue Interface (MQI) 6
Message Repository Manager 118, 282
 See also MRM
message set 282
message-oriented middleware (MOM)
 See also MOM 51
103

- messaging 51
- micro design 141, 153
- mirror 264
- model 30
- Model-View-Controller 35, 43, 55, 151
- MOM 51, 99, 103
- MQ base Java 94, 106, 108, 314, 371, 372
- MQ JMS 371, 372, 374
- MQAI 242, 243
- mqbrasgn 291, 316
- mqbrdevt 291, 316
- mqbrkrs 291, 354
- mqbrops 291, 316
- mqbrtpic 291, 316
- MQGET 242, 255
- MQInput 118, 120, 229, 293
- MQMD 295
- MQOO_BIND_AT_OPEN 253
- MQOO_BIND_NOT_FIXED 253
- MQOPEN 253
- MQOutput 120, 234, 293
- MQPUT 242, 255, 258
- MQQueue 375
- MQQueueConnectionFactory 375
- MQReply 120, 231, 293
- MQRFH2 118, 120, 124, 231
- MQSC 242, 243, 248, 249, 254, 265, 268, 322, 351
- MQSeries Administration Interface (MQAI)
 - See also MQAI 242
- MQSeries Alert Monitor 259
- MQSeries C++ 6
- MQSeries classes for Java 6, 50, 87, 95, 105, 106, 181, 270, 303, 308, 371
 - See also MQ base Java
- MQSeries classes for Java Message Service 7, 371
- MQSeries Client Classes for Java 50
- MQSeries Connector 50
- MQSeries Explorer 243, 244, 246, 247, 248, 249, 256, 322, 328, 350, 351
- MQSeries Integrator SupportPacs 290
 - IC01 Message flow versioning utilities 289
- MQSeries Message Descriptor (MQMD) 124
 - See also MQMD
- MQSeries Services snap-in 244, 245, 246, 247, 248, 256, 259
- MQSeries SupportPacs
 - MA88 MQSeries classes for Java and MQSeries classes for JMS 140, 308, 309, 310, 314, 371
 - MS08 Evaluation of MQSeries System Management Products 240
 - MS0D Selecting MQSeries System Management tools 240
- mqsisbrkrs 316, 317
- mqsicchangebroker 275
- mqsicchangeconfigmgr 275
- mqsicchangetrace 237, 276
- mqsicchangeusernameserver 275
- mqsiccombinemsgflows 290
- mqsiccreatebroker 275, 298, 352, 353, 354, 355
- mqsiccreateconfigmgr 275, 299, 334, 336
- mqsiccreateusernameserver 275, 299, 328
- mqsideletebroker 275, 298
- mqsideleteconfigmgr 275, 299
- mqsideletemsgflows 290
- mqsideleteusernameserver 275, 299
- mqsisfiltermsgflows 289
- mqsisformatlog 236, 276
- mqsilcc 276
- mqsilist 276
- mqsireadlog 236, 276
- mqsireporttrace 276
- mqsisstart 276, 296, 297, 298, 330, 336, 355
- mqsisstop 276, 295, 296, 297, 298, 299
- MRM 118, 119, 283
 - message domain 282
- MVC 164, 167

N

- NEON 119
 - message domain 282

O

- OAM 254, 255
- Object Authority Manager (OAM) 254
- ODBC 126, 273, 331, 332, 333, 345, 346, 347, 353, 355
- Oracle 267, 273
- OSE Remote 28
- OutputRoot 123, 125, 127
- OutputTerminal 118, 136, 196, 205, 206

P

- Page Designer 144
- PARAM 41

- PATH 374
- Pattern Development Kit (PDK) 1, 5
- PCF 242, 243, 254, 255
- PDA 16, 18, 148
- PeopleSoft 26
- performance events 258
- Performance Monitor 259
- performExecute() 57, 62, 66, 67, 69, 70, 73, 74, 75, 76, 83, 84, 85
- persistence 294
- persistent data store 273
- persistent state datastore 268
- Personal Digital Appliance (PDA) 18
- pervasive 38
- plug-in 38, 308
- POSITION 217
- predefined messages 282
- presentation tier 15, 20, 111
- primitive 118, 120, 121, 284
- promote property 138
- protocol firewall 19, 21, 26, 28
- providerutil.jar 372, 374
- Public Key Infrastructure (PKI) 18
- Publication node 293
- publish/subscribe 6, 111, 268, 271, 272, 285, 286, 313, 319
- put authority 255

Q

- Queue Full event 258
- queue manager events 258

R

- RDBMS 45
- readtrace 237
- RealPlayer 38
- redirector 19, 22, 308
- Remote Method Invocation (RMI) 53
- remote OSE 19
- remote queue 250, 256, 258
- repository 109, 111, 113, 249, 250, 272, 273, 288, 299, 313, 319, 320, 323, 324, 330, 331, 332, 340, 341, 342, 343, 344, 348, 350, 363
- Reset Content Descriptor node 231
- reset() 61, 66, 83, 91
- result bean 167
- reverse engineer 143, 144, 145, 177
- RMI 53

- role 151, 154, 166, 278, 279, 280, 281, 316
- Root element 123
- Rose 143, 144, 145, 153, 156, 162, 165, 167, 169, 172, 177
- round-tripping 143
- router tier 15, 16, 20
- runmqtsr 351
- runmqsc 248, 258, 351

S

- SAP R/3 26, 48, 49
- SAX 45
- SELECT 126, 213
- self-defined messages 282
- session bean 47, 68
- SET 126
- setCommandTarget() 62, 64, 75
- setCommandTargetName() 62, 75, 77, 78
- setHasOutputProperties() 67
- SETMQAUT 254, 255
- setOutputProperties() 58, 62
- SOAP 58
- solution outline 141, 146, 147
- SQL 45
- SSL 18
- storyboard 157
- strmqcsv 351
- strmqm 351
- SUBSTRING 129, 217
- SVRCONN 351
- Sybase 267, 273
- synchronous 16, 111, 115, 183, 184
- syncpoint 294
- syslog daemon 278
- syslogd 278
- SYSTEM.ADMIN.SVRCONN 248
- SYSTEM.BROKER 273

T

- TargetableCommand 57, 58, 59, 61, 62, 66, 67, 70, 71, 72, 73, 74, 75, 79, 81, 82, 85
- TargetableCommandImpl 59, 62, 63, 64, 65, 67, 75, 79, 82, 83, 84, 85
- TargetPolicy 68
- TargetPolicyDefault 68
- THE 126
- thin client 13, 37
- topic 52, 271, 286, 290, 300

trace node 234, 235
transformation 52, 103, 115, 116, 120, 121, 123,
188, 190, 199, 205, 209, 211, 213, 214, 215, 217,
267
transport layer 52
TRIM 127

206, 211, 213, 222, 228, 230, 232, 282, 288, 298,
300, 361
 message domain 282
XML.tag 125
XMLConfig 377
XSL 45

U

ujc.jar 374
use case 147, 153, 154, 155, 156, 157, 158, 172
User Name Server 110, 111, 265, 267, 268, 270,
271, 272, 293, 297, 299, 314, 319, 320, 321, 324,
325, 328, 329, 330, 333, 334, 335, 337, 347, 351
User node 18

V

VBScript 39
versioning 289
vertical scalability 21
View bean 170
Visibroker 372
VisualAge for Java 49, 50, 143, 144, 145, 165,
177, 375, 376
VoiceXML 40

W

WAP 40, 148, 151
Web Administration 245, 246, 247, 248, 249, 257
Web application server 17, 20, 21, 22, 27, 28, 42
Web server redirector 19, 22, 27, 28
WebSphere Studio 56, 144, 145, 377
 development environment 144
WebSphere Test Environment 144, 145
Web-Up 12
widget 41, 42
Wireless Application Protocol (WAP) 38
Wireless Markup Language (WML) 38
workload balancing 112
workload management 29
writeToLog() 95

X

XA resource coordinator 106
XMI Toolkit 144
XML 35, 37, 39, 40, 45, 48, 58, 87, 94, 95, 102,
105, 116, 119, 123, 124, 125, 126, 127, 133, 136,
176, 183, 184, 187, 189, 190, 196, 197, 199, 200,

IBM Redbooks review

Your feedback is valued by the Redbook authors. In particular we are interested in situations where a Redbook "made the difference" in a task or problem you encountered. Using one of the following methods, **please review the Redbook, addressing value, subject matter, structure, depth and quality as appropriate.**

- Use the online **Contact us** review redbook form found at ibm.com/redbooks
- Fax this form to: USA International Access Code + 1 845 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Document Number	SG24-6160-00						
Redbook Title	User-to-Business Patterns Using WebSphere Advanced and MQSI Patterns for e-business Series						
Review	<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>						
What other subjects would you like to see IBM Redbooks address?	<table><tr><td></td></tr><tr><td></td></tr><tr><td></td></tr></table>						
Please rate your overall satisfaction:	<input type="radio"/> Very Good <input type="radio"/> Good <input type="radio"/> Average <input type="radio"/> Poor						
Please identify yourself as belonging to one of the following groups:	<input type="radio"/> Customer <input type="radio"/> Business Partner <input type="radio"/> Solution Developer <input type="radio"/> IBM, Lotus or Tivoli Employee <input type="radio"/> None of the above						
Your email address: The data you provide here may be used to provide you with information from IBM or our business partners about our products, services or activities.	<input type="radio"/> Please do not use the information collected here for future marketing or promotional contacts or other communications beyond the scope of this transaction.						
Questions about IBM's privacy policy?	The following link explains how we protect your personal information. ibm.com/privacy/yourprivacy/						



User-to-Business Patterns Using WebSphere Advanced and MQSI

(0.5" spine)

0.475" <-> 0.875"

250 <-> 459 pages



Redbooks

User-to-Business Patterns Using WebSphere Advanced and MQSI Patterns for e-business Series

Select topologies and mappings to build U2B e-business solutions

Gain insight into the latest technologies, design guidelines

Learn to implement the solution from examples

Patterns for e-business are a group of proven, reusable assets that can help speed the process of developing applications. The pattern discussed in this book, the User-to-Business Pattern, is the general case of users interacting with enterprise transactions and data. In particular it is relevant to those enterprises that deal with goods and services that cannot be listed and sold from a catalog.

This redbook discusses application topology 5 of the User-to-Business Patterns. Application topology 5 links multiple presentation tiers to any back-end client, but the back-end is not hidden to the user.

The topologies are illustrated using WebSphere Application Server Advanced Edition V3.5, MQSeries V5.1, and MQSeries Integrator V2. The sample application uses the Command Manager Framework, included with WebSphere V3.5.

Part 1 of this redbook takes you through the process of choosing an application topology and a runtime topology. It then gives you possible product mappings for implementation of the chosen runtime topology.

Part 2 provides a set of guidelines for building your e-business application. It includes information on technology options, application design and application development.

Part 3 takes you through a working example, showing the implementation of an e-business application using application topology 5.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-6160-00

ISBN 0738418307