WebSphere software

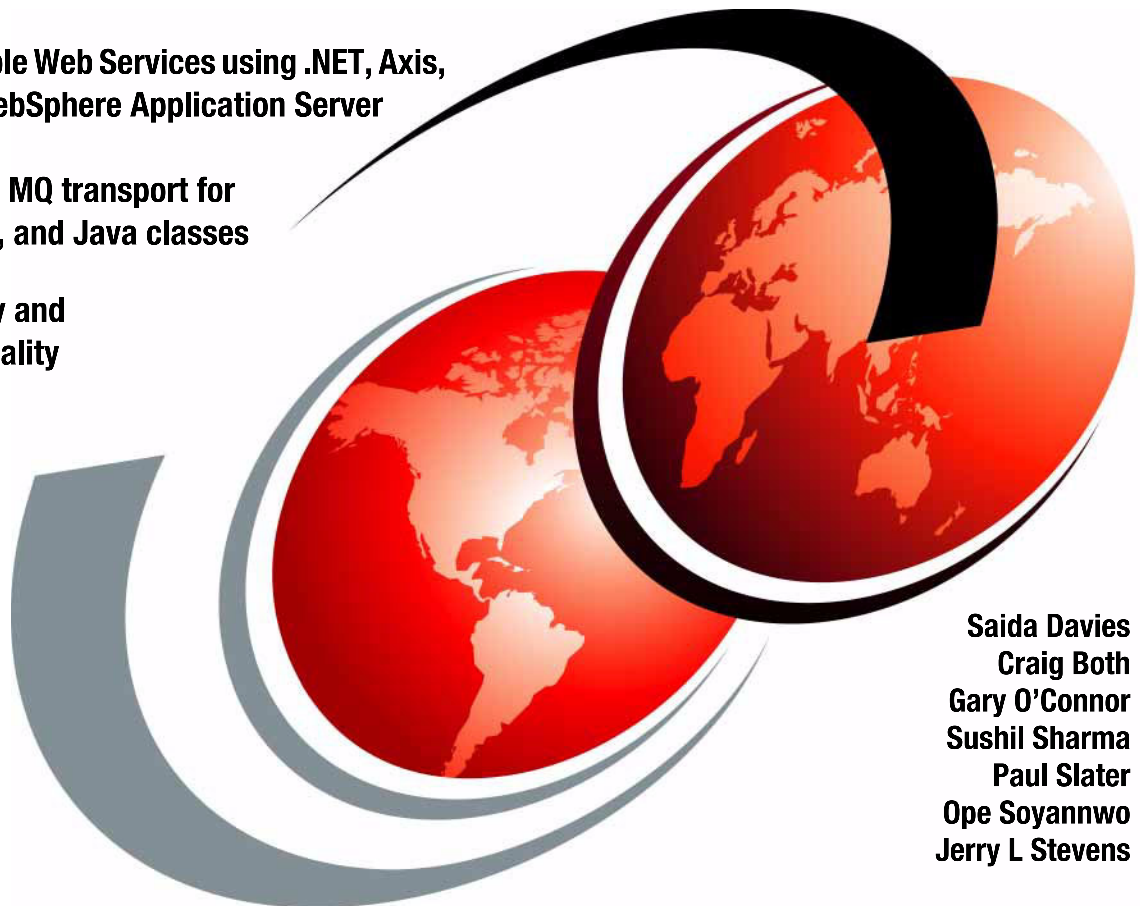
IBM

WebSphere MQ Version 6 and Web Services

Interoperable Web Services using .NET, Axis, and IBM WebSphere Application Server

WebSphere MQ transport for SOAP, .NET, and Java classes

Asynchrony and transactionality



Saida Davies
Craig Both
Gary O'Connor
Sushil Sharma
Paul Slater
Ope Soyannwo
Jerry L Stevens

Redbooks

ibm.com/redbooks

International Technical Support Organization

WebSphere MQ Version 6 and Web Services

October 2006

SG24-7115-00

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

First Edition (October 2006)
This book includes an update of the material present in *WebSphere MQ Solutions in a Microsoft .NET Environment*, **SG24-7012**

Version 1 Release 1 (or earlier) of Microsoft .NET Framework
Version 1 Release 1 (or earlier) of Microsoft .NET SDK
Version 6 Release 0 of IBM WebSphere MQ
Version 6 Release 0 of IBM Rational Application Developer
Version 6 Release 0 of IBM WebSphere Application Server for IBM AIX
Version 6 Release 0 of IBM WebSphere MQ V6.0 - SupportPac MA0V
Version 1 Release 4 of IBM Java SDK

© Copyright International Business Machines Corporation 2006. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures xi

Tables xv

Notices xvii
Trademarks xviii

Preface xix
The team that wrote this IBM Redbook xx
Become a published author xxiii
Comments welcome xxiv

Part 1. Overview 1

Chapter 1. Introduction 3
1.1 Object-orientation 4
1.2 Self-description 4
1.3 Messaging 5

Chapter 2. Objectives 7
2.1 Programming models 8
2.2 Overview of the chapters 9

Chapter 3. Technologies 13
3.1 Web Services 14
3.1.1 Universal Resource Identifier 14
3.1.2 Extensible Markup Language 15
3.1.3 Universal Description Discovery and Integration 16
3.1.4 Understanding Web Services Description Language 16
3.2 Simple Object Access protocol (SOAP) 18
3.3 Microsoft .NET 19
3.3.1 .NET Framework and the Common Language Runtime 20
3.3.2 Internet Information Services and Active Server Pages 20
3.3.3 COM+ 21
3.3.4 Visual Studio .NET 21
3.4 IBM WebSphere Application Server 22
3.4.1 Java 2 Platform, Enterprise Edition 22
3.4.2 IBM Rational Application Developer for WebSphere Software 23
3.4.3 SOAP/Java Message Service 23
3.5 Apache Axis 24

© Copyright IBM Corp. 2006. All rights reserved.

3.6 WebSphere MQ V6 24

Part 2. Web Services and security considerations 27

Chapter 4. WebSphere Services with WebSphere MQ 29
4.1 SOAP over Hypertext Transfer Protocol 30
4.2 SOAP over WebSphere MQ 31
4.3 Client applications 32
4.3.1 Axis clients 33
4.3.2 Microsoft .NET clients 34
4.3.3 Registration 34
4.4 The SOAP layer 35
4.4.1 SOAP message styles and encodings 35
4.4.2 Interoperability 37
4.4.3 WebSphere MQ SOAP Uniform Resource Indicator 39
4.5 SOAP/WebSphere MQ sender 41
4.6 SOAP/WebSphere MQ listener 42
4.7 Service applications 42
4.8 Service deployment 43
4.9 WebSphere MQ infrastructure 44
4.9.1 The request queue and the response queue 44
4.9.2 Queue manager connections 44
4.9.3 WebSphere MQ channels 45
4.9.4 Security and error handling 46
4.9.5 Advanced features 46

Chapter 5. SOAP/WebSphere MQ implementation 49
5.1 Setting up the environment and using the samples 51
5.1.1 Setting the environment variable WMQSOAP_HOME 51
5.1.2 Running the amqwsetcp.cmd/sh command 51
5.1.3 Using the Installation Verification Test to verify installation 52
5.1.4 Executing the setupWMQSOAP.cmd/sh script 53
5.2 The development process 53
5.3 SOAP formatting 57
5.3.1 Specifying Remote Procedure Call-style encoding or Document-style encoding 58
5.4 The deployment process 59
5.4.1 Deployment utility syntax 64
5.4.2 The SOAP/WebSphere MQ Universal Resource Indicator 65
5.4.3 Request queues 70
5.4.4 Response queues 72
5.4.5 Queue manager connection types 72
5.5 Customizing the deployment process 73
5.5.1 Illustrating the Microsoft .NET customized deployment 74

5.5.2 Illustrating the Axis customized deployment 77
5.5.3 Using complex objects in Java and Microsoft .NET 80
5.5.4 The use of mixed package names 81
5.6 The WebSphere MQ transport for SOAP listener 81
5.6.1 Microsoft .NET listener runtime syntax 83
5.6.2 Methods to start listeners 83
5.6.3 Stopping a listener 85
5.6.4 The role of identity context 85
5.6.5 Listener transactionality 86
5.7 Permanent and temporary dynamic response queues 88
5.8 WebSphere MQ transport for SOAP error handling 90
5.8.1 Report messages 91
5.8.2 Message integrity options 92
5.9 Microsoft .NET asynchronous interface 93
5.9.1 Using Microsoft .NET short-term asynchrony 94
5.10 WebSphere Application Server and CICS Transaction Server interoperability 100
5.10.1 WebSphere Application Server interoperation 101
5.10.2 CICS interoperation. 104
5.11 Summary 104

Chapter 6. Security 107
6.1 Concepts of security 108
6.1.1 Security services 109
6.1.2 Security mechanisms 110
6.2 Security considerations 110
6.2.1 Application layer security 111
6.2.2 Transmission layer security 112
6.3 Concepts of cryptography 113
6.3.1 Cryptography. 113
6.3.2 Message digest 115
6.3.3 Digital signature 116
6.3.4 Digital certificates 117
6.4 Introduction to Secure Sockets Layer 120
6.4.1 Concepts of Secure Sockets Layer 120
6.4.2 CipherSuites and cipherSpecs 121
6.5 Secure Sockets Layer support in WebSphere MQ 121
6.6 Working with WebSphere MQ and Secure Sockets Layer 123
6.6.1 Configuring WebSphere MQ for secured communication 123

Part 3. Implementing synchronous Web Services 139

Chapter 7. Environment setup 141
7.1 Software prerequisites. 142

7.2 Software installation . . . 142
7.2.1 Installing IBM WebSphere MQ V6 . . . 142
7.2.2 Installing Microsoft .NET Framework Redistributable V1.1 . . . 145
7.2.3 Installing Microsoft .NET Software Development Kit V1.1 . . . 145
7.2.4 Verifying the installation of WebSphere MQ transport for SOAP . . 146
7.2.5 Installing WebSphere Application Server V6 for AIX . . . 146
7.2.6 Installing Rational Application Developer V6 . . . 147
7.3 Environment setup . . . 147
7.3.1 Basic WebSphere MQ administration . . . 151

Chapter 8. Axis Web Service . . . 159
8.1 Design . . . 160
8.2 Requirements . . . 162
8.3 Implementation . . . 162
8.3.1 Implementation of Web Service . . . 163
8.3.2 Preparing the WebSphere MQ environment . . . 163
8.4 Deployment . . . 164
8.4.1 Common deployment steps . . . 165
8.4.2 Executing a simple deployment to a local default queue manager . 167
8.4.3 Executing a deployment to a local queue manager with specific request and response queues . . . 170
8.4.4 Executing a deployment to a remote queue manager . . . 171
8.5 Error handling . . . 180
8.6 Security . . . 184
8.7 Using the Web Service . . . 186
8.8 Summary . . . 186

Chapter 9. Axis client . . . 187
9.1 Design . . . 188
9.2 Requirements . . . 190
9.3 Implementation . . . 190
9.3.1 Proxy code . . . 190
9.3.2 A client for a local Axis service . . . 192
9.3.3 A client for a remote .NET service . . . 200
9.3.4 The WebSphere MQ environment . . . 205
9.4 Error handling . . . 208
9.4.1 Unable to put a request to queue . . . 208
9.4.2 Specified request queue does not exist . . . 208
9.4.3 Response not received . . . 209
9.4.4 Cannot find the client-config.wsdd file . . . 209
9.4.5 Incorrect message format . . . 210
9.5 Security . . . 210
9.6 Summary . . . 211

Chapter 10. .NET Web Service 213
10.1 Design 215
10.2 Requirements 217
10.3 Implementation 217
10.3.1 Implementation of the Web Service 217
10.3.2 Compiling the Web Service 221
10.4 Preparing the WebSphere MQ environment 222
10.5 Deployment 227
10.5.1 Common deployment steps 228
10.5.2 Executing a simple deployment to a local default queue manager 229
10.5.3 Executing a deployment to a local queue manager with specific request and response queues 230
10.5.4 Executing a deployment to a remote queue manager 232
10.6 The SOAP/WebSphere MQ listener 235
10.7 Error handling 236
10.8 Security 239
10.9 Using the Web Service 241
10.10 Summary 241

Chapter 11. .NET client 243
11.1 Design 244
11.2 Requirements 244
11.3 Implementation 245
11.3.1 Proxy code 245
11.3.2 Implementing .NET client to make synchronous calls 246
11.3.3 Implementing the .NET client to make asynchronous calls 252
11.3.4 Preparing the WebSphere MQ environment 256
11.3.5 Setup for client mode and server binding mode connection 258
11.4 Error handling 262
11.5 Security 266
11.6 Summary 266

Chapter 12. WebSphere Application Server Web Service 269
12.1 Design 270
12.2 Requirements 272
12.3 Implementation 272
12.3.1 Creating and implementing the Web Service skeleton 272
12.3.2 WebSphere MQ and WebSphere Application Server setup 277
12.3.3 Deployment 284
12.4 Security 285
12.5 Summary 287

Chapter 13. WebSphere Application Server client 289

13.1 Design . . . 291
13.2 Requirements . . . 292
13.3 Implementation . . . 292
13.3.1 WebSphere MQ setup . . . 294
13.4 Deployment . . . 295
13.5 Security . . . 298
13.6 Summary . . . 300

Part 4. Asynchrony and transactionality . . . 301

Chapter 14. Long-term asynchronous functionality (MA0V) . . . 303
14.1 Overview of asynchronous facilities . . . 304
14.2 Installation of MA0V . . . 305
14.3 The SOAP/WebSphere MQ Installation Verification Testing and MA0V . . . 306
14.4 Developing a client to use long-term asynchrony . . . 307
14.5 Response queues and asynchronous clientID . . . 311
14.6 Illustration of client software modification . . . 313
14.6.1 Asynchronous request notification . . . 313
14.6.2 Trapping an AsyncResponseExpectedException . . . 314
14.6.3 Instantiating an asynchronous response listener . . . 314
14.6.4 Implementing an asynchronous callback . . . 315
14.6.5 Stopping the response listener . . . 316
14.7 Building client applications . . . 316
14.7.1 Microsoft .NET client applications . . . 317
14.7.2 Java client applications . . . 317
14.8 Long-term asynchrony and error handling . . . 317
14.9 ResponseListener start/finish notification . . . 319
14.10 Maintaining the side queue . . . 321
14.10.1 Removing queue mapping entries from the side queue . . . 321
14.10.2 Removing redundant context objects from the side queue . . . 324
14.11 Uninstalling MA0V SupportPac . . . 325
14.12 Summary . . . 326

Chapter 15. Implementing long-term asynchronous Web Service clients . . . 327
15.1 The Web Service . . . 328
15.2 Implementation of long-term asynchrony . . . 329
15.3 Executing the .NET client . . . 337
15.4 Executing the Axis client . . . 338

Chapter 16. Transactional functionality (MA0V) . . . 339
16.1 Overview of MA0V transactional functionality . . . 340
16.2 Transactional demonstration samples . . . 343

16.2.1 Microsoft .NET client transactionality 343
16.2.2 Developing a transactional Microsoft .NET client 345
16.3 Axis client transactionality 348
16.3.1 Developing a transactional Axis client. 350
16.4 Summary 354

Chapter 17. Implementing transactionality 355
17.1 Overview 356
17.2 Java. 358
17.2.1 Invoking the service within a transaction. 358
17.2.2 Processing the response within a transaction. 364
17.3 Microsoft .NET. 366
17.3.1 Invoking the service within a transaction. 367
17.4 Summary 372

Part 5. Web Services and WebSphere MQ clustering 373

Chapter 18. Using WebSphere MQ clustering with Web Services 375
18.1 Benefits of WebSphere MQ clustering with Web Services 376
18.2 An example scenario 376
18.2.1 The client invocation and the WebSphere MQ sender 377
18.2.2 The Web Service and the WebSphere MQ listener 379
18.3 Summary 379

Appendix A. WebSphere MQ using .NET classes 381
WebSphere MQ .NET classes 382
Overview 382
Environment setup 386
Interacting with queues 387
Working with messages 387
Putting a message on a WebSphere MQ queue. 388
Getting a message off a WebSphere MQ queue 388
Sending messages 389
Receiving messages 389
Application development 389
Simple WebSphere MQ put operation 389
Simple WebSphere MQ get operation 392
Request and reply 395
The .NET monitor 404

Appendix B. WebSphere MQ using Java classes. 405
Overview 406
Using the WebSphere MQ Java classes 406
What are WebSphere MQ Java classes? 407

Environment setup . . . 409
Interacting with queues . . . 409
Working with messages . . . 409
Putting a message on a WebSphere MQ queue . . . 410
Getting a message off a WebSphere MQ queue . . . 411
Application development . . . 411
Simple WebSphere MQ put operation . . . 412
Simple WebSphere MQ get operation . . . 414
Request-and-reply messaging pattern . . . 416
Transaction participation with SOAP/WebSphere MQ . . . 422
Web Service client transaction participation . . . 423
Web Service transaction participation . . . 424

Appendix C. Deployment utility quick reference . . . 425
Sample deployment command lines . . . 429

Appendix D. Additional material . . . 431
Locating the Web material . . . 431
Using the Web material . . . 432
How to use the Web material . . . 432

Abbreviations and acronyms . . . 433

Related publications . . . 435
IBM Redbooks . . . 435
Other publications . . . 435
Online resources . . . 435
How to get IBM Redbooks . . . 436
Help from IBM . . . 437

Index . . . 439

Figures

4-1 Web Services over HTTP 30
4-2 Web Services over WebSphere MQ 31
4-3 Use of a proxy by a Web Service client 33
4-4 WebSphere Application Server SOAP/JMS client accessing SOAP/WMQ service 38
4-5 SOAP/WebSphere MQ client accessing a CICS service 38
5-1 WebSphere MQ transport for SOAP deployment 60
5-2 The three levels of WebSphere MQ transport for SOAP transactionality . 87
5-3 Comparing WsnInitCtxFactory and Nojndi 103
6-1 Example of eavesdropping and tampering 108
6-2 Symmetric key encryption 114
6-3 Asymmetric key encryption 114
6-4 Obtaining a digital certificate 119
6-5 Opening the GSKit Key Manager 124
6-6 Creating a new key repository. 124
6-7 Creating a password for the key repository. 125
6-8 The default CA root certificates in a new key repository 126
7-1 Selecting custom installation. 143
7-2 Ensuring that correct options are selected on installation. 144
7-3 Client invoking Web Service using WebSphere MQ transport for SOAP 147
7-4 Full environment setup 149
7-5 Opening the WebSphere MQ Explorer 152
7-6 Creating a new queue manager 153
7-7 Setting up a default queue manager 154
7-8 Creating a new local queue. 155
7-9 Naming a new local queue 156
8-1 Banking service class figure 160
8-2 SOAP WebSphere MQ infrastructure on the service side 161
8-3 Configuring the classpath 166
8-4 Simple deployment of BankingService.java 169
8-5 Environment setup for a client mode connection 172
8-6 Creating a server connection channel. 173
8-7 Information flow: Configuration using different queue managers for client and service 175
8-8 Starting a service listener 178
8-9 A service listener ending 179
8-10 Listener unable to get messages from the request queue 180

© Copyright IBM Corp. 2006. All rights reserved.

8-11 Output of listener if the request queue does not exist 181
8-12 Listener is unable to put a message to the response queue. 182
8-13 An unexpected message on the request queue 183
9-1 Client infrastructure 188
9-2 Importing proxy files 193
9-3 Adding additional libraries in Rational Application Developer 195
9-4 Configuring RAD to start the client 198
9-5 Adding a folder to the classpath in RAD 199
9-6 The Axis client running 200
9-7 Connecting through a client channel. 205
9-8 Queue manager-to-queue manager connection 206
10-1 SOAP WebSphere MQ infrastructure on the service side 216
10-2 Creating a queue manager 223
10-3 Environment setup for a client mode connection 224
10-4 Environment setup for a binding mode connection. 226
10-5 Output when the amqwsetcp script is run 228
10-6 Executing a simple deployment 229
10-7 Message flow during Web Service invocation in server binding mode . 234
10-8 Output of listener if the request queue does not exist. 237
11-1 Creating a new Windows application project 247
11-2 Proxy generation from Web Service's WSDL using .NET Framework's wsdl.exe 249
11-3 Adding proxy to the project 250
11-4 Windows application form serving as the BankingService Web Service client 251
11-5 Environment setup for a server binding mode connection 259
11-6 Message flow during Web Service invocation in server binding mode . 260
11-7 Debug information when put is inhibited 262
11-8 Debug information when get is inhibited 263
11-9 Debug information when the request queue specified does not exist. . 264
11-10 Debug information when listener is not started, causing timeout 265
12-1 SOAP/JMS components 271
12-2 Web Service and Router EJB project 273
12-3 Changing the soap:binding transport attribute 274
12-4 Web Service skeleton configuration page. 276
12-5 WebSphere MQ V6 as a generic provider 283
13-1 Importing the WSDL 293
14-1 Overview of long-term asynchrony 309
16-1 Illustrating the three levels of transactional control 341
17-1 Generating a key pair for signing the assembly with a strong name. . . 371
18-1 WebSphere MQ clustering with Web Services 377
A-1 Request-and-reply message identification 396
A-2 Result of running all the three demos 403

B-1 Approach to point-to-point application development. 412

Tables

5-1 Summarizing request queue validation at deployment 71
6-1 Environment setup for UNIX platforms 127
6-2 Setting the MQSSLKEYR environment variable 133
7-1 Apache Axis V1.1 runtime location 145
7-2 Software installation 150
8-1 BankingService method description 160
9-1 Client source code structure 188
9-2 Proxy files generated 191
9-3 External libraries required by proxy files 193
10-1 BankingService method description 215
11-1 Proxy files 245
11-2 Description of the GUI buttons 251
12-1 BankingService method description 270
17-1 BankingService method description 356
17-2 TransactionHelper methods 358
C-1 Deployment utility parameters 426
C-2 URI parameters 427
C-3 connectionFactory parameters 428

© Copyright IBM Corp. 2006. All rights reserved.

Notices

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

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

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

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

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

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

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

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

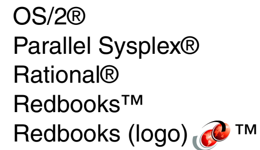
This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used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

COPYRIGHT LICENSE:
This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

© Copyright IBM Corp. 2006. All rights reserved.

Trademarks

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

AIX®	OS/2®	RS/6000®
CICS®	Parallel Sysplex®	SupportPac™
developerWorks®	Rational®	WebSphere®
GDDM®	Redbooks™	z/OS®
IBM®	Redbooks (logo)  ™	

The following terms are trademarks of other companies:

Enterprise JavaBeans, EJB, Java, Java Naming and Directory Interface, JavaBeans, JavaScript, JVM, J2EE, Solaris, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

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

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

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

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

Preface

This IBM® Redbook demonstrates the use of IBM WebSphere® MQ in Microsoft® .NET and Apache Axis environments as a middleware product that is used in the implementation of Web Services.

This book introduces concepts, standards, and technologies pertaining to Web Services and WebSphere MQ first. It then covers in depth the operation, development, and deployment of a Web Service using WebSphere MQ transport. It also discusses issues that may arise, such as security and service availability.

Step-by-step examples are provided for the development of the following:

- Microsoft .NET Web Services
- Apache Axis Web Services
- IBM WebSphere Application Server Web Services
- Microsoft .NET client
- Apache Axis client
- IBM WebSphere Application Server client

The IBM SupportPac™ MA0V introduces features that are only available when using WebSphere MQ as a transport for Web Services. This book also discusses and demonstrates the advanced features, asynchrony, and transactions.

The appendixes provide additional reference information and include examples showing the use of base WebSphere MQ application programming interfaces (API) in .NET and Java™ applications. The appendixes also provide information about how to obtain the source used for the examples in this IBM Redbook.

© Copyright IBM Corp. 2006. All rights reserved.

The team that wrote this IBM Redbook

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

Front row, from left: Paul Slater, Sushil Sharma, Ope Soyannwo, and Saida Davies.
Back row, from left: Gary O'Connor, Jerry L Stevens, and Craig Both.

Saida Davies is a Project Leader for ITSO, and has 15 years of experience in IT. She has published several IBM Redbooks™ on various business integration scenarios on distributed and IBM z/OS® platforms. Saida has experience in the architecture and design of WebSphere MQ solutions, besides extensive knowledge of the IBM z/OS operating system and a detailed working knowledge of both IBM and Independent Software Vendors' operating system software. In a client-facing role as a senior IT specialist with IBM Global Services, her role includes the development of services for z/OS and WebSphere MQ within the z/OS and Windows® platform. This covers the architecture, scope, design, project management, and implementation of software on stand-alone systems or on systems in an IBM Parallel Sysplex® environment. She has received Bravo Awards for her project contributions. She has a degree in Computer Studies and her background includes z/OS systems programming. Saida supports Women in Technology activities, and contributes and participates in their meetings.

Craig Both graduated from the University of Newcastle, United Kingdom (UK), in 2001 with a Bachelor of Science degree in Computing Science and Mathematics. He joined IBM UK Hursley Labs as a graduate working on testing WebSphere MQ Everyplace. Working closely with the test team in Bangalore, India, Craig spent a month with the team in India to finalize the test transfer. On his return to the UK, he moved into a test role on WebSphere MQ V6. Craig is currently the team lead for a functional verification test team for WebSphere MQ. One of his main areas of focus is WebSphere MQ security, including Secure Sockets Layer and cryptographic hardware on distributed platforms. He participates in programs that bring IT to schoolchildren, promoting programming and information technology. Craig currently leads an initiative to better facilitate international communication within a technical arena for IBM UK Hursley Labs.

Gary O'Connor is an IT Specialist working for Application Management Services (AMS), part of IBM Global Services UK. He has worked for IBM for five years. During this period, he worked on a number of large-scale client engagements in financial services, retail, and public sectors. In these engagements, Gary participated in full life cycle development using Microsoft .NET and Java 2 Platform, Enterprise Edition (J2EE™). He has undertaken significant work pertaining to Web Services with Microsoft .NET. He has experience in using WebSphere MQ with both J2EE and Microsoft .NET.

Sushil Sharma has over 25 years experience in the software industry. After a degree in Logic and Physics at the University of Sussex, he built programming skills at mini computer manufacturers Perkin-Elmer and Raytheon. Development projects included operating system and database management software. He has undertaken projects for the European Space Agency and the European Commission. At IBM UK, he has worked as a Software Engineer on both mainframe and distributed systems, including development of IBM Graphical Data Display Manager (GDDM®), IBM Operating System/2 (OS/2®), and WebSphere MQ. He is currently working as a technical specialist as part of the WebSphere MQ Distributed Change Team.

Paul Slater has a Bachelor of Science degree in Computer Science from Durham University, UK. He joined IBM as a graduate in 2002. During his time at IBM, he has worked in various test and level 3 service roles, including WebSphere MQ and WebSphere Application Server messaging. During WebSphere MQ V6 development, Paul was responsible for system testing the SOAP/MQ feature. His current role is in the Scenarios Test Team for WebSphere Business Integrator Message Broker.

Ope Soyannwo is an IT consultant for iMeta Technologies, Southampton. She graduated with a first class Bachelor of Engineering degree in Computer Systems Engineering from the University of Hull. Prior to graduation, she joined IBM on an internship program, where she co-authored her first IBM Redbook

WebSphere MQ Solutions in a Microsoft .NET Environment, SG24-7012 and wrote a White Paper that was published on the IBM developerWorks® Web site. Ope received an Author Recognition Award for this contribution. Her experience lies mainly within the Web Services sector, where she has worked on several projects involving end-to-end solutions design, application development, and testing. Ope has been instrumental in the completion of this IBM Redbook through her additional post-residency contribution and reviews.

Jerry L Stevens graduated from Exeter University with a first class Honors degree in Mathematics and has 27 years of IT experience. He currently works in the WebSphere MQ Technical Strategy and planning group at IBM UK Hursley Labs, where he has been working for three years on the development of Web Services facilities for WebSphere MQ. Prior to joining Hursley Labs, Jerry worked in an IBM RS/6000® and SP consultancy practice for IBM UK Global services in a client-facing role as an IBM AIX® Consultant. He is one of the authors of the IBM Redbooks *Sizing and Tuning GPFS*, SG24-5610 and *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012. Prior to joining IBM in 1997, Jerry worked for Shell as a Senior Systems Engineer, where he undertook a range of technical consultancy and development roles and worked with a variety of Open Systems platforms and architectures.

The IBM Redbook team would like to thank the following people for their assistance in the initial planning of this IBM Redbook:

Peter Broadhurst, Level 3 Service for WebSphere MQ
Software Group, Application and Integration Middleware Software
IBM Hursley, UK

The IBM Redbook team would like to thank the following people for their assistance and contributions to this IBM Redbook:

Flora Batca, Manager, WebSphere Application Server z/OS L2 Support and Software Engineer
IBM Software Group, Application and Integration Middleware Software, IBM USA

Stuart Reece, WebSphere MQ - Java Messaging L3 Service Team Leader
IBM Software Group, Application and Integration Middleware Software, IBM UK

Paul Titheridge, WebSphere MQ - Java Messaging L3 Service
IBM Software Group, Application and Integration Middleware Software, IBM UK

Richard J Scheuerle Jr, WebSphere Web Services Software Engineer
IBM Software Group, Application and Integration Middleware Software, IBM USA

Phil Adams, Team Lead WebSphere Web Services Engine Development and Software Engineer
IBM Software Group, Application and Integration Middleware Software, IBM USA

Stephen Todd, Manager STSM, messaging systems (especially message/database interactions)
IBM Software Group, Application and Integration Middleware Software, IBM UK

Mike Horan, WebSphere MQ Software Developer
IBM Software Group, Application and Integration Middleware Software, IBM UK

Jitu Patel, WebSphere MQ Software Developer
IBM Software Group, Application and Integration Middleware Software, IBM UK

Karl Donald, ITCL TopGun, Test Architect for WebSphere Business Integration Broker & ESB Products, Software Engineer
IBM Software Group, Application and Integration Middleware Software, IBM UK

Richard M Conway, WebSphere Support, IT Specialist, Technical Support
ITSO z/OS, IBM USA

Cheryl Gera, Center Support and Administrative Services
ITSO z/OS, IBM USA

Become a published author

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

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

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

Overview

This IBM Redbook is an introduction and guide to the implementation of Web Services using IBM WebSphere MQ in order to provide a reliable and adaptable transport for SOAP. The IBM Redbook *WebSphere MQ Solutions in a Microsoft .NET Environment*, SG24-7012 covered similar ground for WebSphere MQ V5.3 enhanced with SupportPac MA0R. This book is available on the Web at:

http://publib-b.boulder.ibm.com/abstracts/sg247012.html?Open

Since then, the SOAP support and the tools provided by SupportPac have been integrated into WebSphere MQ V6. In addition, a new SupportPac MA0V that extends the SOAP support to include asynchrony and transactions is also available.

In Part 1 of this book, Web Services and the challenges they present to developers are introduced, challenges, which this book helps you overcome. This part also provides an overview of the standards, tools, and other resources that are available to implement robust and reliable solutions. In particular, it emphasizes the exciting addition that a SOAP transport based on a reliable messaging bus makes to the toolbox.

© Copyright IBM Corp. 2006. All rights reserved.

1

Introduction

Globalization and an explosive increase in global communications since the 1980s contributed to a dramatic increase in communication through computers. An increase in mergers and acquisitions has, along with the Internet, brought about a corresponding increase in business-to-business communications. A huge demand exists for connecting homogenous and heterogeneous networks, systems, and applications. Looking ahead, connectivity requirements may increase from attempts to network even the smallest appliances and personal devices, including everyday items from the supermarket. The software industry has made every effort to keep ahead of these growing requirements by delivering the necessary tools to manage changes and optimize the value gained from the changes.

In the struggle to develop and maintain systems that are both useful and usable, some fundamental ideas arise repeatedly in various guises. Basic patterns are seen behaving as genetic material in a rapid evolution towards complex systems. This chapter examines a few of the patterns that have come into prominence.

© Copyright IBM Corp. 2006. All rights reserved.

1.1 Object-orientation

Since the first attempt at writing maintainable software, developers have realized that program structure must be controlled. Otherwise, complexities set in rapidly and reach a point where the program becomes incomprehensible. From the simplest structured programming techniques, there has been a move through data-based structures and formal methods to the current predominance of object-oriented design and programming.

With the advent of object-oriented methods, problems that were previously considered as complex are being tackled. However, this is due to more than just software development in object-oriented languages. Object-based application programming environments specify the containers within which groups of objects exist concurrently. Components created in a container can discover and interact with each other. Both Microsoft .NET and Java 2 Platform, Enterprise Edition (J2EE) environments provide such containers.

The requirement to communicate with older systems that were developed to different patterns is ever present. Frequent interfaces to them are created using proxies or adaptors in order to make them look as objects within the current view.

1.2 Self-description

From the time of the earliest applications, programmers are faced with a plethora of data formats. Besides the application data that must be processed, applications must find and exchange data with other applications.

Self-describing data carries information about its own content and structure. Nearly all the major software architectures are characterized by the development of such data formats. This includes the simplest word processing formats, graphical data, and documents for business interchange through to the discovery and use of objects or components at run time.

The current trend is to use Extensible Markup Language (XML) as the standard way to tag or mark up all forms of information in order to allow them to be processed easily by any number of alternative, often competing, applications and services. XML is simple and well-documented. Besides, tools to create or parse XML are readily available.

1.3 Messaging

The use of buffered data in both synchronous and asynchronous forms is the basis for all forms of communication between hardware and software systems. Some operating systems are entirely message-based.With asynchronous messaging, applications can send or process messages when it is the easiest. When the nodes involved in a network connection grow and become more widespread, asynchrony facilitates applications to work reliably even when some links are not available.

The World Wide Web relies on long streams of message data assembled by layers of protocol software. The data for each level is, with some provisos, appropriately tagged so that it is recognized and passed for processing to the correct receiver code. When more dynamic behavior came to be expected of the Web, technologies such as Microsoft's Distributed Component Object Model (DCOM), and Java's Remote Method Invocation (RMI) were developed.

These Remote Procedure Call (RPC) technologies allow objects to reside in the most appropriate locations. However, current requirements exceed the capabilities of these technologies. Communication must be implemented between globally distributed applications developed for different ends and using a variety of different languages and formats. The applications may run on any number of different environments, but require coordination to provide specific quality-of-service levels.

It is no surprise that organizations choose to enable the discovery and invocation of applications and services in a way that is similar to that used for the Web itself. It is also not surprising that standards that fully exploit object orientation, self-definition, and messaging have begun to evolve.

Web Services on the Internet dramatically improve the ability to exploit the Web. Web Services that run on an intracompany or intercompany message bus have the potential to radically change the way businesses cooperate or operate internally.

2

Objectives

During the last decade, there has been an emergence in software architectures that enable application integration within and between large businesses and other enterprises. The architectures provide frameworks within which applications written to standard models or patterns can be deployed and made available across networks to many users and systems.

© Copyright IBM Corp. 2006. All rights reserved.

2.1 Programming models

Following are the models adopted by application developers in such environments:

- The three-tier or n-tier model

  This model represents the logical layers of a distributed enterprise application, that is, presentation, business logic, and data or data storage. In n-tier models, these three layers are further divided.

- The publish-and-subscribe model

  This model is used to rapidly distribute information over the network.

- The peer-to-peer or request-and-reply messaging model

  This model is where a server acts on a message from a requesting service or client and sends back a response.

The programming models are supported by a suite of technologies that encompass application servers, distributed object discovery and invocation, message-oriented middleware, and distributed database products.

A Web Service is a collection of related functions, typically modelled as an object that is packaged and published on a network for use by other programs. Web Services can exploit and interconnect systems using those technologies and programming models. A Web Service can act as a client to other Web Service-enabling applications that function as, for example, brokers for other businesses or fulfil a request using multiple independent resources.

If there is to be an infrastructure on which Web Services can work across networks, whether the Internet or company intranets, and wide area networks, the diverse nature of the systems that are connected mandates that the technology employed is based on well-understood and accepted standards. Even with appropriate standards in place, interoperability is difficult to achieve. Without appropriate standards between Web Service clients and servers that have no prior knowledge of each other, interoperability cannot be expected.

2.2 Overview of the chapters

Chapter 3, “Technologies” on page 13 briefly reviews the main technologies adopted by Web Services. These include the standards on which Web Services are based on and the platforms and products that provide the implementations within which the Web Services server and client applications can run. The objective of Web Services is to provide an environment where services are seamlessly available irrespective of the client platform and the server platform.

What are the requirements that applications must meet to provide or use such services? In which environments can they run and how do they establish and use the network connections required? These and other related issues are discussed in Part 2, “Web Services and security considerations” on page 27. Many of these issues can be addressed by using WebSphere MQ as the transport for the SOAP messages between the client and the server. These aspects are discussed in detail in the rest of this book. Concepts, design, and implementation using WebSphere MQ transport are the other topics that are discussed.

Chapter 4, “WebSphere Services with WebSphere MQ” on page 29 provides information about Web Service concepts and overall application communication flows when using WebSphere MQ.

Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 describes in detail how SOAP/WebSphere MQ applications must be implemented and deployed. It also provides information about the features and tools available with WebSphere MQ.

Web Services must be easily located and accessed. At the same time, the services flow and information flow must be secure. Only requests from those who are authorized to access the services must be allowed. All information flow to and from the services must be protected from unauthorized access.

Chapter 6, “Security” on page 107 focuses on security of communication when using WebSphere MQ.

Part 3, “Implementing synchronous Web Services” on page 139 demonstrates how to implement and deploy Web Service clients and servers using WebSphere MQ messaging. Step-by-step examples and code are provided for a variety of platforms, with particular emphasis on interoperability. For each scenario, design, implementation, and deployment are discussed.

Chapter 7, “Environment setup” on page 141 provides information about how to set up the development and test environments used in the examples provided in the following chapters. Some basic WebSphere MQ operations showing the queue manager and queue creation are shown.

Chapter 8, “Axis Web Service” on page 159 describes the implementation of a Web Service running on Apache Axis 1.1.

Chapter 9, “Axis client” on page 187 demonstrates Axis client implementation.

Chapter 10, “.NET Web Service” on page 213 describes the implementation of a Web Service running under Microsoft .NET.

Chapter 11, “.NET client” on page 243 provides information about a .NET client implementation.

Chapter 12, “WebSphere Application Server Web Service” on page 269 discusses the implementation of a Web Service running within the WebSphere Application Server.

Chapter 13, “WebSphere Application Server client” on page 289 demonstrates a WebSphere Application Server client implementation. Messaging with WebSphere MQ is essentially asynchronous. However, up to Chapter 13, messaging WebSphere MQ has only been used in a synchronous manner.

Part 4, “Asynchrony and transactionality” on page 301 discusses some of the advanced facilities that are available.The asynchronous and transactional interfaces provided by SupportPac MA0V are discussed and demonstrated.

Chapter 14, “Long-term asynchronous functionality (MA0V)” on page 303 discusses the exploitation of fully asynchronous messaging.

Chapter 15, “Implementing long-term asynchronous Web Service clients” on page 327 describes how asynchronous messaging is implemented.

Chapter 16, “Transactional functionality (MA0V)” on page 339 introduces the use of transactions with SOAP.

Chapter 17, “Implementing transactionality” on page 355 discusses transaction implementation.

Part 5, “Web Services and WebSphere MQ clustering” on page 373 discusses the benefits of using WebSphere MQ clustering with Web Services.

Chapter 18, “Using WebSphere MQ clustering with Web Services” on page 375 shows how WebSphere MQ Clustering can be used to achieve high availability and work load balancing for Web Services.

Appendix A, “WebSphere MQ using .NET classes” on page 381 and Appendix B, “WebSphere MQ using Java classes” on page 405, include small and simple examples to demonstrate the use of WebSphere MQ application programming interface (API) from the Microsoft .NET and Java perspective. These appendixes

use the classes provided specifically for.NET and Java applications by WebSphere MQ, and include the new, relevant features in WebSphere MQ V6.

Appendix C, “Deployment utility quick reference” on page 425 provides a quick reference to the deployment utility arguments.

Appendix D, “Additional material” on page 431 provides links to the source code used in this book and additional material.

3

Technologies

This chapter reviews some of the significant products and standards that are involved in the implementation of the architectures used in Web Services. Two main, competing infrastructure are listed for the 3-tier or n-tier model, which are the predominant models used by businesses:

- Microsoft .NET
- Java 2 Platform , Enterprise Edition-based application servers

Web Services and the associated technologies can be used to provide a reliable and coherent bridge between these two. This can be further enhanced by using WebSphere MQ as the SOAP transport technology.

© Copyright IBM Corp. 2006. All rights reserved.

3.1 Web Services

The term *Web Services* does not denote any particular product or standard. It generically refers to a kind of application accessible over a network. Implicitly, the network can be the HyperText Transfer Protocol-based Web or the Internet in general, but there are no strict stipulations. The requirement for such applications became inevitable with the growth of the Web, and service providers had to provide applications that client programs could access in the same way that users interacting with applications through their Web browsers did.

The prime requirement for these Web-based applications is that the applications should be interoperable, that is, the client and the server should communicate and work together, independent of the operating system, the implementation language, and so on. Thus, a breed of distributed applications and application components based on open standards have come into existence. These are the applications referred to as Web Services.

A number of standards and technologies are used when implementing Web Services.The primary standard is SOAP, which is discussed in 3.2, “Simple Object Access protocol (SOAP)” on page 18. This chapter discusses the other relevant standards.

3.1.1 Universal Resource Identifier

Each Web Service has a server and one or more clients that issue requests to the server. A request has two parts, the Universal Resource Indicator or Identifier (URI) and the data that the server processes or responds to.

Of late, the term *URI* is often used interchangeably with the familiar *URL* (Universal Resource Locator). URIs or URLs are strings identifying the name or network location of a resource. The resource may be a document, a downloadable file or image, a service, or any other resource. The strings have a syntax specified by Request for Comments (RFC) 3986 for URIs or RFC 1738 for URLs. For a more detailed and formal description, refer to the following Web site:

`http://www.w3.org/Addressing`

Within the URI, in addition to the information that specifies where to find the resource, the access method can also be provided. Thus, for example, in the URL //http:www.ibm.com, the scheme name *http* indicates that HTTP should be used in the request to the server for the Web page.

Web Services are also typically accessed using HTTP, but other access methods or protocols can also be used if they are supported by the server and the client. In effect, the URI, when resolved, allows the client to connect to the

endpoint where the client's request is processed. In WebSphere MQ transport for SOAP, the scheme name *jms* is used to indicate to the SOAP infrastructure, for example, Microsoft .NET or Apache Axis 1.1, whether WebSphere MQ is to be used as the transport.

3.1.2 Extensible Markup Language

A markup language is a way of explicitly stating the logical structure or semantics of text data by using indicators called *tags* within the text. Extensible Markup Language (XML) is a general purpose markup language. It is especially designed for creating Web documents, although it can be used for almost any kind of data. XML allows language designers to create their own sets of custom tags, essentially defining new markup languages specific to an area of interest. Information can then be tagged using an appropriate set of tags and can easily be exchanged with other programs that recognize the same set.

The application that tags information can choose from a number of tools to create a document, which can then be saved as a file in XML format. Similarly, there are tools to parse XML files into runtime representations of the document. These tools are common to all the markup languages based on the rules for XML.The availability of such tools make it easy to define a new markup language and create and read documents in the language.

Information in a particular XML format may subsequently be wrapped as data within an XML language defined for high-level applications. The data can then be exchanged between high-level applications, as though it is being passed within an envelope. The XML envelope, with the original application data wrapped inside, can be handled and processed by applications that do not care about the specific format.

These facilities make it easy to create self-descriptive data that can be exchanged and understood by other applications. A full description of XML and the facilities it provides is beyond the scope of this book. We, however, recommend that you read the tutorials available on the Web at:

http://www.xml.org
http://www.xml.com
http://www.w3.org/XML

There are two main areas in which XML is used in Web Services:

- SOAP

  SOAP specifies an XML format for messages carrying requests for function invocation and the associated responses. This means that data that is to be sent to a service for processing can be wrapped in a SOAP XML envelope. On the server side, a SOAP listener is able to pick up the package and determine which application or component to pass it to. Replies to the client application are wrapped in a similar manner.

- Web Services Description Language

  Web Services Description Language (WSDL) is an XML format used to describe a Web Service. A client can use WSDL information to determine the functions a service can perform and how the functions should be accessed.

More details about SOAP protocol and WSDL are provided later in this chapter.

3.1.3 Universal Description Discovery and Integration

Universal Description Discovery and Integration (UDDI) defines a SOAP-based protocol used for publishing and discovering Web Services. The mechanism it uses involves maintaining a registry containing service descriptions. The registry can be explored manually or accessed as a Web Service. The WSDL for a service can be obtained through the registry. Along with access to registries, UDDI includes protocols and methods to create registries, control access, and replicate or distribute entries.

Microsoft .NET, Apache Axis, and WebSphere Application Server, the three platforms that are considered in this book, provide facilities to access UDDI registries. However, UDDI is not discussed in detail because it does not directly affect the structure of either the client or the server application.

For more information about UDDI and its use, refer to the following Web site:

http://www.uddi.org

3.1.4 Understanding Web Services Description Language

Web Services Description Language is an XML format used to describe and locate Web Services. WSDL files describe the following:

- What the Web Service does
- The interface that is to be used to access the service
- The data types used and returned by the service
- Where the service resides
- How to connect to the service

At the time of writing this book, WSDL was not a standard, although WSDL V2, which is currently in a draft form, is expected to be adopted by the World Wide Web Consortium (W3C). Because WSDL is not a standard, differences in implementation by various Web Service platforms may cause interoperability problems. To access the specification documents, visit the following Web site:

http://www.w3.org/2002/ws/desc

A Web Services client can read the WSDL created for a Web Service to determine which functions are available on a server, and to generate the appropriate SOAP data to invoke the functions found. In most SOAP environments, tools are provided to generate proxy classes for the interfaces provided by the Web Service from the WSDL. Client application code then uses the proxies to access the service as though it was locally implemented.

In a similar manner, Microsoft .NET and Axis infrastructure facilitate the generation of WSDL from the Web Services source code. For Axis, this is from Java source and for Microsoft .NET, it is from any of the Microsoft .NET common language runtime (CLR) languages. Because WSDL provides a formal definition of the service, tools can also be used to derive the skeleton code for a server application that implements the service interface.

WSDL distinguishes two types of message styles used to invoke the service:

- Remote Procedure Call (RPC)

  This style is suitable for services that provide a number of method calls typically taking relatively simple arguments.

- Document

  This style is suitable for services that process data passed to them in large pieces, which they then parse and process themselves.

The message style set by WSDL affects the way the data that is to be sent to the service is formatted within the SOAP envelope.

WSDL also distinguishes two encoding styles, Literal and SOAP-encoded. These determine how the application data values or structures are *serialized* or *deserialized* into the message formats exchanged between the client and the server.

3.2 Simple Object Access protocol (SOAP)

The formal definition of SOAP by the World Wide Web Consortium (W3C) is as follows:

"Simple Object Access Protocol (SOAP) is a lightweight format and protocol for exchange of information in a decentralized, distributed environment. It is an Extensible Markup Language-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses."

For more details, refer to the following Web site:

http://www.w3.org/2000/xp/Group/

The envelope provides control information, the address the message should be delivered to, and the message itself. The representation of the data in the message corresponds to the styles and encodings defined for WSDL.The SOAP message does not have to be parsed and understood by the software that is handling the envelope. SOAP allows headers that are also carried within the envelope and contain information that is useful in processing the service request, but does not form a part of the request itself.

In addition to these different styles, there may also be differences in SOAP formatting by different vendors of Web Service infrastructure, according to their interpretations of the specification. Standards for SOAP specification are still in their infancy. There are currently several versions of the SOAP specification in use, such as V1.1 and V1.2.

SOAP shares some features with the earlier implementations of distributed component technologies, namely, Microsoft Distributed Component Object Model (DCOM), Java Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP). However, SOAP far exceeds these as it is not tied to any particular technology, language, or implementation. It has the potential to be universally employed in the way Hypertext Transfer Protocol (HTTP) is used for Web access.

The self-contained nature of SOAP messages leads to the possibility of a number of different transports over which they can be carried. In effect, Web Services clients and servers can function independent of the actual communications mechanism used to transfer the data between them. SOAP is bound to the transport mechanism chosen. The most common binding is, naturally HTTP, due to its ubiquitous availability and ease-of-use. However, the other transports proposed are:

- SOAP/ Simple Mail Transfer Protocol

  SOAP/Simple Mail Transfer Protocol (SMTP) takes advantage of SMTP's store-and-forward messaging facilities to allow SOAP messages to flow asynchronously. SMTP transport is essentially a one-way transport. Message correlation using the standard SMTP message ID and reply-to headers are used to support the request-reply model.

- SOAP/User Datagram Protocol

  This proposes a SOAP binding to User Datagram Protocol (UDP) for applications that have to use multicast transmission or do not require the packet acknowledgment and retransmission provided by Transmission Control Protocol (TCP).

- SOAP/Java Message Service

  A SOAP binding to Java Message Service (JMS) provides a powerful messaging mechanism because the delivery of the SOAP message is guaranteed and transactions employed to meet the quality of service objectives.

  To a large extent, the full range of facilities available depends on the JMS implementation used. Typically each JMS vendor, known as a *JMS provider*, uses a proprietary message format and offers features and tools that differentiate their product from others.

  IBM WebSphere MQ JMS implementation has the advantage of the underlying, fully featured, messaging bus.

3.3 Microsoft .NET

Microsoft .NET is a brand used by Microsoft for a collection of technologies and products. The main elements and their role within Web Services implementation are discussed in the following sections.

Microsoft Distributed interNet Applications (DNA) is a comprehensive architecture for the development of Web or Internet applications. Microsoft Distributed Internet Architecture's central approach is to logically partition applications into the three layers of the three-tier model.

Microsoft .NET technology, together with the Microsoft Distributed Internet Architecture three-tier model and the Windows operating system provides a comprehensive and powerful platform for Web Services development and deployment.

3.3.1 .NET Framework and the Common Language Runtime

Microsoft .NET Framework's CLR and Global Assembly Cache (GAC) provide a fully managed runtime environment for applications with reference counting, garbage collection, exception management, and namespace support. Versioning and security can be controlled in both CLR and GAC.

Applications can be written in any of the .NET languages, including C#, VisualBasic.NET (VB.NET), and JScript .NET. The Microsoft .NET Framework supplies a rich set of classes that include commonly required application objects and functions.

It is the Microsoft .NET Framework that provides the standard protocols and services required for Web Services, that is, SOAP, WSDL, and UDDI.

3.3.2 Internet Information Services and Active Server Pages

Internet Information Services (IIS) is a Web and application server that supports the hosting of dynamic Web applications.

Active Server Pages (ASP) are server-side application components hosted by IIS that are responsible for presenting information in the Web browser. Earlier versions only supported client-side scripting in VBScript or JavaScript™. ASP.NET provides a framework that supports all the languages that run on the Microsoft .NET CLR, with most of the presentation work performed by ASP.NET components.

These two products effectively provide the environment for the presentation layer of the application.

3.3.3 COM+

COM+, now known as *Enterprise Services*, is an environment formed from the merger of the Component Object Model (COM) and Microsoft Transaction Server. These interfaces and services support the creation of application components.

Typically, such application components implement the business layer of the 3-tier architecture. In particular, they are often used to implement Web Services. They exploit, as appropriate, the COM+ transactional facilities.

Object and thread pooling components are provided. The Windows registry and Active Directory® interfaces provide naming services.

A set of Microsoft-provided COM objects known as Active Data Objects (ADO) is used for accessing relational data. SQL Server provides the Microsoft Relational Database Management System.

All .NET application components do not have to reside on the same computer system. When they are distributed across machines, Microsoft .NET uses .NET Remoting to invoke remote components and Web Services through SOAP or a binary protocol.

3.3.4 Visual Studio .NET

Visual Studio® is the Microsoft development environment. It provides comprehensive and well-integrated facilities for Web page and ASP.NET Web forms design, implementation, and deployment. Solutions or project workspaces and projects for applications in any of the .NET languages can be written and built.

For the examples used later in this book, the use of Microsoft Visual Studio .NET 2003 is recommended. When installed, it provides the Microsoft Development Environment 2003 V7.1 and the Microsoft .NET Framework V1.1. WebSphere MQ SOAP support requires updating the Framework with Service Pack 1 (SP1).

Visual Studio 2005 and the Microsoft .NET Framework V2 are also available.

3.4 IBM WebSphere Application Server

WebSphere Application Server is an IBM proprietary platform that is used to build Java applications in the 3-tier model. It implements J2EE specifications.

WebSphere Application Server provides a host of services to applications along with those that are required by J2EE. This section discusses the main features related to Web Services implementation in WebSphere Application Server.

3.4.1 Java 2 Platform, Enterprise Edition

J2EE specifies a standard application programming model for distributed Java applications. It defines four types of application components, with each type running in a separate container. Each container provides an environment and the services for the applications that run within them, performing services on their behalf.

- Application clients

  These components run in a container, the client container in a client process, that is distinct from a server process, although they may run on the same system.

- Applets

  These are lightweight client components that run in an applet container with limited access to the underlying system. Typically, the container resides in a Web browser.

- Web components

  These are Java Server Pages (JSPs) or servlets, running in the Web container, and typically performing the work of the presentation layer of the 3-tier system.

- Enterprise JavaBeans™

  These components run in an Enterprise JavaBeans (EJB™) container and implement the business layer. EJBs can model business data and participate in transactions coordinated by the EJB container.

Because language support in J2EE is limited to Java, the Java Virtual Machine (JVM™) performs class loading, unloading, reference counting, and garbage collection. The containers provide the thread and object pooling as required.

Communication between remote components or objects is through the use of Java resource manager interface (RMI) and Internet Inter-ORB Protocol (IIOP), which allow use of objects written in other languages. The Java Native Interface (JNI) allows Java objects to access non-Java system resources. The Java Naming and Directory Interface™ (JNDI) provides naming services, that is, searching for resources and their attributes by name.

3.4.2 IBM Rational Application Developer for WebSphere Software

IBM Rational® Application Developer WebSphere Software is a comprehensive integrated development environment provided for WebSphere Application Server and J2EE application development. For Web Services development, it includes the following:

- Wizards to ease Web Services development
- Support for UDDI, SOAP, and WSDL
- WSDL editor
- Deployment and test tools
- A Web Services Explorer

3.4.3 SOAP/Java Message Service

WebSphere Application Server supports Java Message Service (JMS) as a Java messaging standard and SOAP/JMS to allow Web Services to take advantage of messaging. Various JMS providers can be used. If WebSphere MQ is selected, WebSphere Application Server SOAP/JMS applications can interoperate with WebSphere MQ SOAP applications running on .NET or Axis systems.

SOAP/JMS is not yet a standard. Specifically, there is no common accepted structure for the JMS messages that carry the SOAP data. This means that SOAP/JMS is only interoperable when the same JMS provider is used at both ends of the communication. WebSphere MQ SOAP support interoperates with WebSphere Application Server or IBM Customer Information Control System (CICS®) because all three use the same message format that is used by SOAP/JMS where WebSphere MQ is the JMS provider.

If a widely accepted standard for SOAP/JMS does become available, SOAP/JMS can be expected to rapidly become widespread because JMS makes an excellent *message bus*. It provides both synchronous and asynchronous messaging, with the ability to use transactions and enforce security. It caters to both point-to-point and publish and subscribe programming models.

3.5 Apache Axis

Apache Axis is a fully featured SOAP infrastructure, including not only an implementation of SOAP, but also a significant number of extras with extensive support for WSDL. For a full description, refer to the following Web site:

http://ws.apache.org/axis/java/user-guide.html#Introduction

This Web site describes Axis 1.2, which supports applications written in Java. A version for C++ is also available.

> **Restriction:** WebSphere MQ V6 currently supports only the earlier Apache Axis 1.1.

3.6 WebSphere MQ V6

WebSphere MQ provides messaging between applications and Web Services and offers reliable and resilient connectivity. It uses queuing and transactional facilities to help preserve the integrity of messages across the network. Message delivery *exactly once* reduces the risk of data being lost when services or networks fail.

In WebSphere MQ V6, the support for SOAP messaging is integrated into the product. It was previously available as SupportPac MA0R. This support is the main focus of this book and is covered in detail in later chapters. It supports .NET and Axis 1.1 SOAP implementations. For transmission, WebSphere MQ wraps SOAP messages with the same headers that are used for SOAP/JMS, where WebSphere MQ is the JMS provider. This means that it can interoperate with WebSphere Application Server and CICS Transaction Service.

Communications can be secured using Secure Sockets Layer (SSL). Clustering is provided to support high availability and workload balancing.

WebSphere MQ provides a consistent API across a number of platforms and includes classes that provide access to the messaging facilities for .NET and Java applications and a full JMS implementation.

For an introduction to WebSphere MQ and the facilities it provides, refer to *WebSphere MQ V6 Fundamentals*, SG24-7128, which is also available on the following Web site:

http://www.redbooks.ibm.com

Appendix A, “WebSphere MQ using .NET classes” on page 381 and Appendix B, “WebSphere MQ using Java classes” on page 405 provide examples pertaining to the use of the .NET and Java interfaces provided by WebSphere MQ to implement messaging between the two applications.

Some of the other significant enhancements provided in V6 are:

- Support for 64 bit systems
- A trigger monitor (.NET monitor) for .NET applications
- Compression on message channels
- Online monitoring, accounting, and statistics

SupportPac MA0V is an asynchronous support in WebSphere MQ transport for SOAP that facilitates invocation of Web Services and transactional handling of messages over WebSphere MQ. The reason these functions are not installed as part of the main SOAP support in WebSphere MQ is that they are not defined by the SOAP standard. However, they are expected to be powerful tools in extending the capabilities of Web Service applications.

Note: Full support is not available for MA0V because it is a Category II SupportPac.

Part 2

Web Services and security considerations

This part discusses the theory behind using WebSphere MQ as a reliable transport mechanism for Web Services.

Chapter 4, “WebSphere Services with WebSphere MQ” on page 29 and Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 present the concepts of how Web Services can be integrated with a WebSphere MQ environment, including how WebSphere MQ replaces HyperText Transfer Protocol (HTTP) as the transport mechanism, and some considerations to be aware of when implementing Web Services.

© Copyright IBM Corp. 2006. All rights reserved.

Chapter 6, “Security” on page 107 describes the concept of securing message flows using the Secure Sockets Layer (SSL), as supported by WebSphere MQ V6.

4

WebSphere Services with WebSphere MQ

Chapter 1, “Introduction” on page 3, Chapter 2, “Objectives” on page 7, and Chapter 3, “Technologies” on page 13 provided a broad overview of Web Services and SOAP technology. This chapter examines how the various standards and products can work together to achieve reliable services. In particular, this chapter focuses on the use of WebSphere MQ as the transport provider. It also discusses the development, deployment, and use of WebSphere MQ’s advanced features.

© Copyright IBM Corp. 2006. All rights reserved.

4.1 SOAP over Hypertext Transfer Protocol

Figure 4-1 illustrates a Web Services client making a request to a Web Services server over Hypertext Transfer Protocol (HTTP) transport.

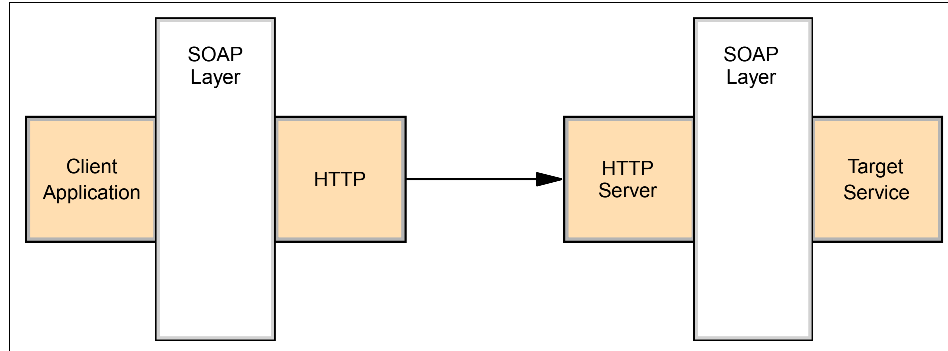


Figure 4-1 Web Services over HTTP

The following steps provide a simple view of the process involved in creating and processing the request:

1. The client invokes a method of a class hosted by the server through a proxy of the class accessible to the client. The creation and deployment of proxies is discussed later in this chapter.
2. The SOAP layer, such as an implementation of SOAP provided by Microsoft .NET, Apache Axis, or WebSphere Application Server, catches the method call and marshalls the function name and parameters, creating a representation of the call in Extensible Markup Language (XML) form.
3. The XML for the method call is itself wrapped in a SOAP envelope, which is also in XML format. See 4.4.1, “SOAP message styles and encodings” on page 35 for a detailed description of the XML used to package the raw request in this manner.
4. The data built up so far is then sent to the target service as a HTTP request. Implicitly, this is over TCP/IP.
5. The HTTP request is received by a HTTP server, typically, a Web server such as the Microsoft Internet Information Services (IIS) or the Apache Web Server.

6. The SOAP data is passed to the SOAP layer by the HTTP server.

> **Note:** The SOAP layer at the receiver end can be a different implementation of SOAP from that used at the client end of the exchange. This is the basis of Web Services interoperability.

7. The server side SOAP layer parses the SOAP envelope, extracting the XML representation of the method invocation. It then extracts the data for the method call itself.
8. The SOAP layer locates the target service and invokes the function desired by the client.
9. The service runs the function and returns the results.
10. The results are returned to the client in a similar manner.

4.2 SOAP over WebSphere MQ

Figure 4-2 illustrates a Web Services client making a request to a Web Services server over WebSphere MQ transport instead of HTTP.

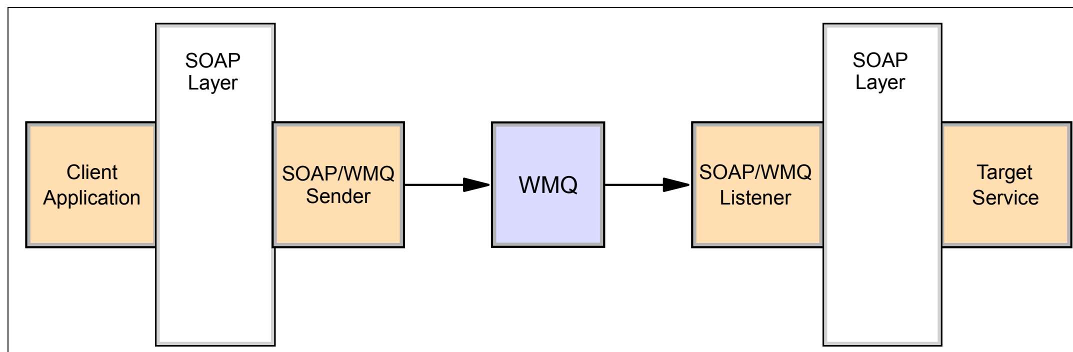


Figure 4-2 Web Services over WebSphere MQ

All the steps are the same as for SOAP/HTTP except for steps 4, 5, and 6. Instead of sending the data using HTTP, the SOAP layer passes the SOAP envelope and contents to a SOAP/WebSphere MQ sender. The SOAP/WebSphere MQ sender puts the SOAP message into a queue as a WebSphere MQ message. A process known as SOAP/WebSphere MQ listener then reads the message from the queue and passes the SOAP data to the server side SOAP layer in the same manner that the Web server does in the case of HTTP[1]. The responses are returned in a similar fashion.

At first it may look as though this is just adding an indirect layer for transporting data. However, there are a number of advantages in having WebSphere MQ as the transport for the data:

- The application can take advantage of the assured delivery feature to be certain that the request reaches its destination, and reaches it exactly once.
- An existing WebSphere MQ infrastructure that is already being used for application integration can be used for transport, if necessary.
- The transport can be secured or compressed by exploiting the features of WebSphere MQ V6. With HTTP transport, SSL encryption of the transmission is available using HTTP over SSL (HTTPS). However, with WebSphere MQ, access to the queuing mechanism can also be controlled.
- WebSphere MQ clustering features can be employed for load balancing and enhanced reliability and availability.
- All transport using WebSphere MQ is asynchronous in nature (HTTP transport is logically synchronous) and the application can take advantage of this, if necessary.
- The Web Service client, service or both can take advantage of WebSphere MQ units of work or XA transactions to coordinate respectively on updates to WebSphere MQ and other resources.
- Where WebSphere MQ is selected as the transport for SOAP/JMS it is possible for either the client or the server to interoperate with SOAP/JMS implementations.

Before considering implementation in depth, review how the various parts of the system work. This is explained in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

4.3 Client applications

Logically, client applications initiate the protocol flow or message flow seen when a Web Service request is processed.

Typically, the client code uses a proxy to access the service functions or methods as illustrated in Figure 4-3.

---

[1] Although the SOAP/WebSphere MQ senders and SOAP/WebSphere MQ listeners are referred to simply as senders and listeners throughout this book, they must not be confused with WebSphere MQ sender channels or WebSphere MQ listeners. Sender channels and WebSphere MQ listeners are components of WebSphere MQ responsible for transmitting queued messages over a network. SOAP/WebSphere MQ senders and listeners merely queue and dequeue SOAP messages. Similarly, they must not be confused with Java Messaging Service (JMS) senders and listeners that remain distinct, although performing a similar function.

A proxy is a class that presents the same interface and method signatures as the service, but instead of directly implementing the desired behavior, invokes the method remotely. The proxy class is generated from Web Services Description Language (WSDL), and compiled and made available locally as part of the deployment process. Deployment is discussed in detail in 4.8, “Service deployment” on page 43.

WebSphere MQ client applications can run on one of the two possible SOAP infrastructure that is provided by .NET and Apache Axis 1.1. Both environments provide tools to generate proxies from WSDL.

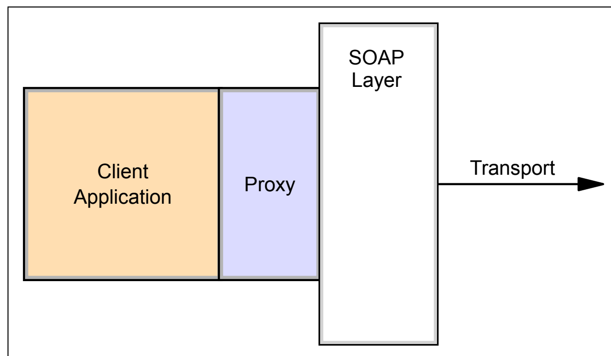


Figure 4-3 Use of a proxy by a Web Service client

4.3.1 Axis clients

Axis clients are written in Java. Typically, the client instantiates a proxy class for the service and invokes one of its methods. However, a proxy is *not* always required.

Whether an Axis client requires a proxy or not is determined by the programming style that is used. Axis supports three programming styles, SOAP, WSDL, and PROXY.

- SOAP style

  This style assumes that the client knows the location and function signatures of the service and does not use a WSDL definition of the service or proxy.

- WSDL style

  This style uses WSDL to locate the service, but assumes that the client knows the correct parameters to pass on each call. No proxy is used with the methods being invoked using Java application programming interface (API) for XML-based RPC (JAX-RPC) classes.

- PROXY style

  This style uses a proxy generated from the WSDL for the service.

In this book, the PROXY style for Axis clients is used because it is the simplest. However, an example using the WSDL style is provided as one of the Java samples installed with WebSphere MQ.

4.3.2 Microsoft .NET clients

Microsoft .NET clients must be written in one of the Microsoft common language runtime (CLR) languages such as C# or Visual Basic®. A proxy is always used. As with the Axis environment, the proxy is generated from WSDL and is compiled and installed locally as part of the deployment process. Example 4-1 shows a call from a .NET client to a Web Service. A step-by-step description of the development of a .NET Web Services client is given in the scenario described in Chapter 10, ".NET Web Service" on page 213.

Example 4-1 .NET client call to a Web Service

```
private void btnGetBalance_Click(object sender, System.EventArgs e)
      {
         //Make a new instance of the Web service
         BankingService.BankingService service = new
BankingService.BankingService();
         //Call the web method to get the current balance then display
it
         lblBalance.Text = service.getBalance().ToString();
      }
```

4.3.3 Registration

The Web Service that is to be accessed by a client is always specified as a Universal Resource Indicator (URI) or HTTP Universal Resource Locator (URL). The URI that is to be used is defined in the proxy generated from the WSDL for the service. If WSDL-style programming is used by an Axis client, it is obtained by the Axis infrastructure from the WSDL directly.

After the SOAP layer has marshalled the call into the SOAP message to be sent, it should invoke a component that is able to process the URI and set up a connection with the service so that the message is transferred. For both .NET and Axis SOAP implementations, the component to be called for a particular URI scheme name should be registered to the environment.

WebSphere MQ registers its URI scheme name (jms) when a special registration call is made by the client[2]. This means that it is not possible to run a client that already accesses a Web Service over HTTP to use WebSphere MQ SOAP transport without altering and rebuilding the client code.

The registration calls that have to be made are provided in Chapter 5, "SOAP/WebSphere MQ implementation" on page 49.

4.4 The SOAP layer

The SOAP infrastructure in use is responsible for marshalling the method or function invocations and formatting the invocation as a message in XML format, ready for transfer to the service. When the message is ready, it should be passed to the transport software that is registered to handle the URI for the Web Service.

On the server (service) side, the SOAP layer unwraps the message, reconstitutes the call, and invokes the service. Return values are built into a response message and the SOAP transport to transfer it back.

This section takes a look at the SOAP envelope and the effect of selecting different message styles or SOAP encoding (interoperability), and the URI used for WebSphere MQ SOAP transport.

4.4.1 SOAP message styles and encodings

As mentioned in Chapter 3, "Technologies" on page 13 there are several different SOAP styles and encoding variants, including the following:

- Remote Procedure Call (RPC) encoding
- Remote Procedure Call (RPC) Literal encoding
- Document-style encoding. This is also known as Messaging-style encoding.
- Direct Internet Message Encapsulation (DIME)
- SOAP with attachments

The first three of these encoding options are the most commonly used. RPC and Document-style encoding differ in many ways. However, note that both of them specify how to translate the definition of a method or function call in WSDL, known as *WSDL binding*, to a SOAP message. Use either of the styles depending on whether the programming model for the service is purely RPC-based or message-based.

[2] SupportPac MA0R uses a different scheme specifier, wmq. See Chapter 5, "SOAP/WebSphere MQ implementation" on page 49.

The structure of the SOAP envelope is not examined in detail in this book. However, for purposes of illustration, Example 4-2 and Example 4-3 may be of interest. Compare the SOAP data for a simple function call with the scenario described in Chapter 15, “Implementing long-term asynchronous Web Service clients” on page 327.

Example 4-2 shows SOAP RPC encoding.

Example 4-2 SOAP RPC encoding

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tns="http://tempuri.org/"
    xmlns:types="http://tempuri.org/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body
        soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <types:creditResponse>
            <creditResult xsi:type="xsd:boolean">true </creditResult>
            </types:creditResponse>
        </soap:Body>
    </soap:Envelope>
```

Example 4-3 shows SOAP Document encoding.

Example 4-3 SOAP Document encoding

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
        <creditResponse xmlns="http://tempuri.org/">
            <creditResult> true </creditResult>
            </creditResponse>
        </soap:Body>
    </soap:Envelope>
```

The RPC message essentially describes the details of a procedure call with its name and parameter values or procedure returns.

Document-style services publish their data to services in a more generic XML form. Services can parse such messages from any client that follows a set XML schema. This is therefore, a less rigid implementation than RPC.

RPC encoding is easiest to implement within a SOAP engine, and Document-style encoding the most difficult.

A more detailed discussion of the strengths and weaknesses of the various SOAP styles and encodings can be found in the following Web site:

http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/index.html

Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 discusses implementation using specific styles.

DIME and SOAP with attachments are two different methods for encoding binary data. Neither of these are currently supported in WebSphere MQ transport for SOAP. RPC Literal is not supported.

4.4.2 Interoperability

Although the SOAP specification provides us with a basis for the interoperability of Web Services across platforms and SOAP infrastructure developers, it cannot *guarantee* such an interoperability. In particular, the different options for formatting the data, as described in 4.4.1, “SOAP message styles and encodings” on page 35, means that interoperability depends on the formats specifically supported by the different implementations involved.

WebSphere MQ as a transport for SOAP, does not process or attempt to convert SOAP data by itself. Therefore, it supports interoperability only to the extent that the SOAP implementations supports it.

The message format used by SOAP/WebSphere MQ does enable interoperability with the WebSphere Application Server and the Customer Information Control System (CICS) Transaction Server, as illustrated in Figure 4-4 and Figure 4-5.

Figure 4-4 shows WebSphere Application Server SOAP/JMS client accessing a SOAP/WebSphere MQ service.

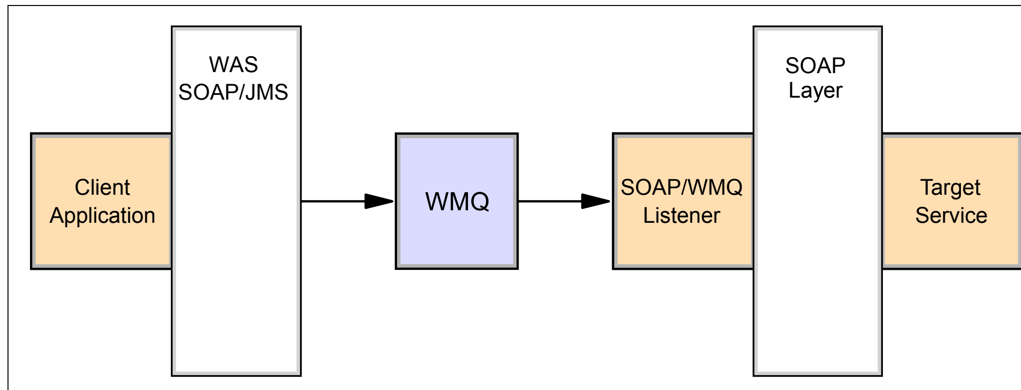


Figure 4-4 WebSphere Application Server SOAP/JMS client accessing SOAP/WMQ service

Figure 4-5 shows SOAP/WebSphere MQ client accessing a CICS service.

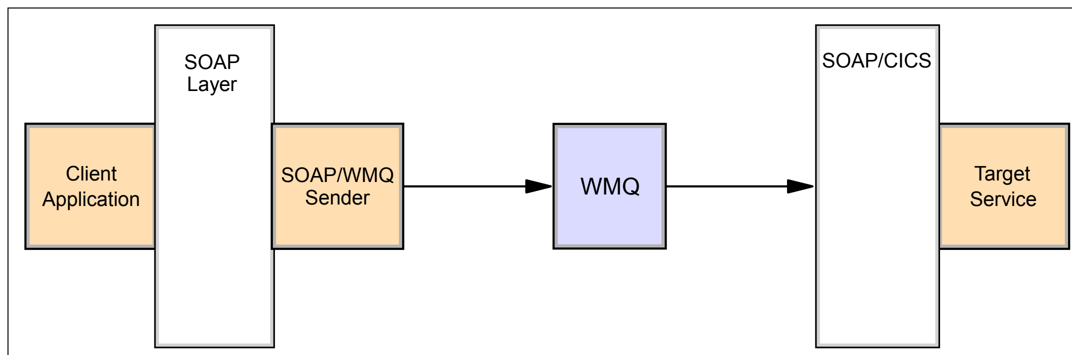


Figure 4-5 SOAP/WebSphere MQ client accessing a CICS service

Interoperability is possible in these cases because all the three IBM products use the same message format, which is described in 4.5, "SOAP/WebSphere MQ sender" on page 41. The use of the same format enables SOAP/WebSphere MQ to interoperate with all SOAP/JMS implementations where WebSphere MQ is used as the JMS provider.

Simple and complex objects

The SOAP specification defines a set of simple data types such as String, Int Long, and so on. In addition, it specifies encoding rules that make it possible to represent complex data types such as arrays, Java beans, and custom objects.

Although complex objects can be represented in SOAP, this does not mean that the exchange of complex objects is always possible. With reference to Microsoft .NET and AXIS SOAP implementations, Java objects and .NET objects cannot be expected to share a representation as instantiated within their own environments. They may not share a representation even when serialized into SOAP-wrapped and encoded message data. Essentially, this means that interoperability can be severely restricted with respect to complex objects, for example, when they are passed as arguments on a service invocation.

Note: Even for Java objects being passed in a homogenous Axis environment, interoperability using complex data types may not be possible. Only Java beans can be passed because these are the only classes for which the serialized format is established.

SOAPAction

Another aspect that currently affects interoperability between Microsoft .NET and other implementation is that of SOAPAction. This is a SOAP header that must be present for Microsoft .NET to pass the request to the Web Service. However, other implementations neither require it nor generate it unless the WSDL makes it clear that it is required.

This means that in most cases, for a client to use a .NET Web Service, the WSDL used to generate the proxies should have been generated from the WSDL derived from the .NET server source.

4.4.3 WebSphere MQ SOAP Uniform Resource Indicator

Section 4.3.3, “Registration” on page 34 discusses the registration of the transport to which the SOAP implementation passes the SOAP data according to the URI obtained from the proxy or WSDL. This section takes a closer look at the WebSphere MQ over SOAP URI.

The URI for a Web Service to be accessed using WebSphere MQ follows the syntax used for SOAP/JMS. It always starts with //jms:/queue?. The remainder of the URI is a series of name-value pairs.

The URI is discussed in detail in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

The name-value pairs provide the following critical information:

- Destination

  This specifies the request queue, the queue on which the request should be placed. For WebSphere MQ, it is a WebSphere MQ queue name or queue and queue manager name of the request queue.

  In a general sense, this specifies a connection factory object that can be invoked to create a connection to the JMS provider. WebSphere MQ does not use connection factories and this parameter provides a series of name-value pairs that specify precisely how the WebSphere MQ sender should connect to a WebSphere MQ queue manager to place the request. It provides details of a response queue to which replies to the SOAP messages are routed. Options to provide security can also be included.

- ResponseQueue

  The queue to which replies from a Web service are sent.

- InitialContextFactory

  In the customary use of the jms URL format, this parameter specifies the Java Naming and Directory Interface (JNDI) provider that is used to look up the connectionFactory parameter. Because WebSphere MQ SOAP specifies the WebSphere MQ connection details directly in the connectionFactory parameter in a JMS environment such as WebSphere Application Server, this must be set to a special class that avoids the JNDI lookup.

- TargetService

  This names the target service. The server side SOAP layer uses this to distinguish between multiple services that may be running.

Further parameters allow the attributes related to message persistence and lifetime to be specified.

The URI is passed to the SOAP/WebSphere MQ sender along with the encoded SOAP message. 4.5, “SOAP/WebSphere MQ sender” on page 41 describes how the sender uses the information in the URI to ensure that the message reaches the service.

4.5 SOAP/WebSphere MQ sender

A WebSphere MQ queue manager is a service that hosts a number of queues on which applications can leave messages that can be read later by the same application or other applications. Queue managers can be configured to route messages to queues in the queue managers' remote, from the sending application. In our example, the sending application is the SOAP/WebSphere MQ sender. The application reading the message is the SOAP/WebSphere MQ listener associated with the Web Service.

The sender performs the following actions when invoked with the URI and SOAP message by the SOAP layer:

1. A header structure, the WebSphere MQRFH2, is created and added before the SOAP message data. This contains metadata about the SOAP message that can be processed by the SOAP/WebSphere MQ listener when passing the SOAP data to the target service.
2. The sender connects to a queue manager as specified in the connectionFactory parameter. The queue manager may be local or remote from the sender and may or may not host the destination queue for the SOAP message.
3. A response queue, which receives the reply from the service, is opened for input. If the queue that is to be used has not been specified in the URI, the sender uses a default.
4. The sender puts the message to a queue in the queue manager. If the destination queue is local, WebSphere MQ checks whether the sender has the write permission. As part of the put operation, another header, the Message Descriptor, is added to the message.

   The Message Descriptor contains information about the sending application, message persistence and lifetime, and the destination queue and queue manager. Information about the response queue is also included. This information is used by WebSphere MQ to route the message and provide information about the message sender to the reading application.

   After the sender puts the request message, it waits for a response to arrive in the response queue (assuming synchronous operation).

4.6 SOAP/WebSphere MQ listener

The WebSphere MQ infrastructure flows the message placed by the sender to the request queue where the SOAP/WebSphere MQ listener expects to receive the requests. The use and configuration of the WebSphere MQ components that make up the infrastructure is discussed later in this chapter. For now, it can be assumed that the message is safely and securely delivered to the request queue.

If the SOAP/WebSphere MQ listener is not already running and the listener triggering is configured during deployment, the listener is started when the message arrives.

The listener reads the message and obtains the Message Dispatcher and WebSphere MQRFH2 headers. The listener then extracts the information required to locate and invoke the Web Service and passes it to the SOAP infrastructure.

4.7 Service applications

Web Services server application code is dependent on the SOAP infrastructure within which it runs. Axis services are written in Java, and Microsoft .NET services are written in C# or other .NET languages.

Restrictions that apply to complex data types are described in “Simple and complex objects” on page 38

Developers must consider whether to use a top-down approach or a bottom-up approach for services development. Top-down development begins with the creation of a WSDL description of the service and using the definition to develop the service source code. Bottom-up development involves deriving the WSDL description from the service source code. Toolsets are available to support both the methods.

This issue may affect the eventual interoperability of the services because client proxies are always derived from the WSDL. Bottom-up development is likely to be the cheapest and easiest approach when existing services are modified to support the Web Services architecture. It is a known fact that clients and servers operate in a homogenous network, with both running on similar operating systems and SOAP implementations.

Top-down development is most likely to result in services that can interoperate and allow clients on other SOAP platforms to access the service without compatibility problems. This is mainly because the service is under the complete control of the service designer, and the tools generating the service skeleton code from the WSDL strictly adheres to it, enabling interoperability from all environments.

Service applications can use transactions in which messages that are received or those that are to be sent using SOAP/WebSphere MQ, are included. These transactions are coordinated by WebSphere MQ in the Axis environment and by COM+ in the Microsoft .NET environment. The form of transactional control that is to be used depends on how the listener is configured, typically, as part of deployment.

4.8 Service deployment

One of the most important tasks in making a Web Service available is to deploy the servers, client proxies, and clients. Typically, this occurs after the service and client applications are written. The appropriate WSDL, proxies, and executable code should be distributed to the locations and directories where they finally run. The SOAP implementation under which the service runs should be configured to locate and run the service's methods when requested. Scripts to start clients and listeners may also be required.

Deployment must ensure that the communications infrastructure required between the client and the server is configured correctly. For Hypertext Transfer Protocol (HTTP) communication, this may include configuring a Web server or security. Similarly, when using WebSphere MQ for SOAP transport, deployment also involves configuring WebSphere MQ queue managers and the client or server connections to them.

WebSphere MQ includes a deployment utility that is installed as part of SOAP support. Because of the diverse nature of the environments in which it may be useful and the many different WebSphere MQ network configurations possible, it is supplied as a sample, rather than a final implementation.

The sample deployment utility concerns itself with Web Service deployment, not service implementation. It assumes that the server application code is already developed. It uses a bottom-up approach, generating the WSDL for the service from the server code.

If the service source code is not derived from a service interface design in WSDL, deployment should begin with the generation of WSDL from the service source code. From the WSDL, client proxies are derived. Scripts should be prepared to start the listeners.

In Chapter 5, “SOAP/WebSphere MQ implementation” on page 49, the use of and customization of the deployment utility is considered in detail. In Part 3, “Implementing synchronous Web Services” on page 139, a series of scenarios are described, in which examples of Web Service and client design on a number of platforms are provided. This includes the use of the deployment utility and its outputs.

4.9 WebSphere MQ infrastructure

The WebSphere MQ infrastructure is responsible for delivering the request to the target request queue (where the SOAP/WebSphere MQ listener finds it) after the sender puts the message in a queue manager. The design and deployment of this infrastructure is either simple or complicated depending on the communication requirements of the system being implemented.

This section examines some of the major issues that should be considered.

4.9.1 The request queue and the response queue

The request queue is the queue from which the SOAP listener reads the message sent to the service. It resides on the queue manager to which the listener connects to. The request queue can be created with one of the WebSphere MQ administration tools. However, assuming that the queue manager is on the same system as the service, it is easier to use the deployment utility provided with WebSphere MQ to create the queue. The listener obtains the name of the response queue and the name of the queue to which it belongs to, from the headers in the request message.

4.9.2 Queue manager connections

WebSphere MQ applications may or may not be located on the same system as the queue manager hosting the queues, from which the applications read or place messages in the following:

- If the application runs on the same system as the queue manager, it can connect in a manner termed as the *server* mode or the *binding* mode.
- If the application runs on a different system from the queue manager, it connects in the *client* mode.

> **Note:** It is possible to connect in the client mode even when the application and the queue manager are collocated.

This method is used quite often for simplicity of application deployment.

In the simplest of configurations, the SOAP/WebSphere MQ listener and the server application run on the same computer. The server runs in the binding mode and the clients connect to the same queue manager in the client mode. Other configurations can involve clients connecting in either mode to queue managers other than those to which the server connects.

The SOAP/WebSphere MQ deployment utility caters well to the simple case. However, for more complicated configurations, it is necessary to consider whether the same URI can be used by both the senders and the listeners. This is considered in detail in Chapter 5, "SOAP/WebSphere MQ implementation" on page 49.

In addition, more sophisticated WebSphere MQ topologies require configuration of channels between the queue managers involved.

Consider the case where a client and the associated sender connect to a local queue manager, but the server application connects to another. The sender places the message in the local queue manager using the destination queue and the queue manager name from the URI. The queue manager recognizes that the destination is a queue on a remote queue manager, and puts the message in a transmission queue, which is a queue used to hold messages temporarily, until it can be delivered to the destination queue manager.

4.9.3 WebSphere MQ channels

WebSphere MQ uses channels to transfer messages from one queue manager to another. A channel or a message channel agent is a process that reads messages from a transmission queue and dispatches them over a communications link to a similar process on a remote system that writes them to the destination queue.

Channels can either be configured individually by an administrator or defined automatically when the queue managers are joined, by using WebSphere MQ clustering support. See Chapter 18, "Using WebSphere MQ clustering with Web Services" on page 375.

In contrast to the creation of request queues, the SOAP/WebSphere MQ deployment utility does *not* generate scripts that configure channels between the queue managers. Therefore, manual or automatic configuration is necessary.

Alternatively, consider writing a customized deployment procedure. See Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 for information about the different methods available to customize deployment.

WebSphere MQ also uses channels, such as *client connection channel* or the *server connection channel*, for connections between client applications and a queue manager. These operate differently, but share many configuration options. If senders are to connect as WebSphere MQ clients, these channels should be configured. The connection options in the URI should be specified so that clients are able to access the channels. WebSphere MQ also provides other ways to make client connection information available to clients. These too can be used.

4.9.4 Security and error handling

The earlier sections of this chapter explored the Web Services flow from the client to the service and back. When you step back and take a look at the flow as a whole, other issues are visible:

- In almost all real life implementations, the security of the flow from end-to-end must be ensured.
- In any distributed system, possible errors that may be encountered in each component, and the way in which to process or report them should be considered.
- In some systems, demanding service capacity and availability requirements have to be met.

These issues are considered in this book’s subsequent chapters.

4.9.5 Advanced features

All WebSphere MQ communications occur asynchronously, except those between a WebSphere MQ client and a queue manager. The SOAP transport, as described until now in this chapter, uses the underlying WebSphere MQ asynchronous messaging in a synchronous manner, implementing what are essentially remote procedure calls from the client in which the client is blocked until the response is returned[3].

---

[3] A degree of asynchronous behavior is available in the .NET environment without MA0V.

Facilities provided by the MA0V SupportPac extend the client programming model in such a way that the asynchronous nature of WebSphere MQ communications can be fully exploited. These extensions also allow SOAP/WebSphere MQ clients to participate in transactions. For a detailed discussion on this, refer to Part 4, “Asynchrony and transactionality” on page 301.

5

SOAP/WebSphere MQ implementation

This chapter builds on the concepts of WebSphere MQ transport for SOAP discussed in Chapter 4, “WebSphere Services with WebSphere MQ” on page 29, and shows you how to use it. This chapter provides an overview of the principal implementation details. If you do not have a clear understanding of the concepts relating to WebSphere MQ transport for SOAP, we recommend that you read Chapter 4, “WebSphere Services with WebSphere MQ” on page 29 thoroughly before reading this chapter.

This chapter discusses the following topics:

- Setting up the environment necessary to use WebSphere MQ transport for SOAP and running the samples provided
- The typical service and client development process
- An overview of the WebSphere MQ transport for SOAP Universal Resource Indicator (URI)
- SOAP formatting
- WebSphere MQ transport for SOAP deployment
- Customization of the deployment process
- WebSphere MQ transport for SOAP listeners

© Copyright IBM Corp. 2006. All rights reserved.

- Error handling facilities
- Use of short-term asynchrony
- WebSphere Application Server and Customer Information Control System (CICS) interoperability

5.1 Setting up the environment and using the samples

WebSphere MQ transport for SOAP includes a series of sample programs that can be used to verify if the install is completed successfully. This suite of programs can be started through the supplied Installation Verification Test (IVT) utility. It is recommended that you start the IVT before you attempt to configure the custom services and clients. The IVT confirms whether the installation is successful and verifies if the prerequisite software is available and accessible in the local environment.

5.1.1 Setting the environment variable WMQSOAP_HOME

Before attempting to start the IVT, set the environment variable WMQSOAP_HOME. Setting this permanently in the environment is recommended. Otherwise, you have to set it each time a new command prompt is opened. To perform this on Windows XP systems, perform the following tasks:

1. Right-click **My Computer** and select **Properties**.
2. Click the **Advanced** tab.
3. Click **Environment Variables**.
4. Click **Edit** under the user variables for <userid>.
5. Add the definition for WMQSOAP_HOME as appropriate. For a default installation of WebSphere MQ on Windows, set this to: `C:\Program Files\IBM\WebSphere MQ`.

Refer to Chapter 7, “Environment setup” on page 141 for more details about the environment setup for WebSphere MQ transport for SOAP.

5.1.2 Running the amqwsetcp.cmd/sh command

When starting a new command prompt to deploy or start WebSphere MQ transport for SOAP services, run the amqwsetcp.cmd/sh script. This script is located in the WebSphere MQ bin directory. Although this script sets up the CLASSPATH for the Java environment, it also makes changes to the PATH environment variable that is used with Microsoft .NET services. The script must therefore be run, whether using the Java environment or the Microsoft .NET environment.

The amqwsetcp.cmd script sets up the basic environment for WebSphere MQ transport for SOAP. Many of you may have constraints about the way things are organized, for example, you may have to refer to external classes referenced from a service. It is therefore, quite likely that further customization is required beyond that provided by amqwsetcp.cmd/sh. For this type of customization, using a wrapper script that invokes amqwsetcp.cmd/sh is the best course of action. It is possible to avoid editing the script directly wherever possible.

Note: If the Microsoft .NET Framework is installed in a nondefault location, edit amqwsetcp.cmd to specify the actual location. The default assumed by the script is %SystemRoot%\Microsoft.NET\Framework\v1.1.4322.

5.1.3 Using the Installation Verification Test to verify installation

The IVT, as installed with the product, includes a set of synchronous Microsoft .NET and Axis sample services and client programs. The test configuration file ivttests.txt defines how these samples are run. This is the default test configuration file used by the IVT mechanism. It defines the various synchronous client samples provided with the product. When installing the optional asynchronous SupportPac MA0V, two more IVT test configuration files are installed. These files can be used to demonstrate asynchronous and transactional clients. Refer to 14.3, “The SOAP/WebSphere MQ Installation Verification Testing and MA0V” on page 306 for further details.

Refer to Chapter 7, “Environment setup” on page 141 and Chapter 3 of the manual titled *WebSphere MQ transport for SOAP,* SC34-6651 for more details on running the IVT.

Note: It is not possible in WebSphere MQ V6 to install the Microsoft .NET or Axis components of WebSphere MQ transport for SOAP separately. Both environments have to be installed even if only one is required.

5.1.4 Executing the setupWMQSOAP.cmd/sh script

The IVT mechanism runs the setupWMQSOAP.cmd/sh script. This script creates the following default WebSphere MQ objects that are required:

- The queue manager used by the samples
- The default response queue (SYSTEM.SOAP.RESPONSE.QUEUE)
- The default model queue (SYSTEM.SOAP.MODEL.RESPONSE.QUEUE). This is used in conjunction with the dynamic response queues.
- The side queue used with asynchronous client requests (SYSTEM.SOAP.SIDE.QUEUE)

When the IVT runs setupWMQSOAP.cmd/sh, the script creates and uses the WMQSOAP.DEMO.QM queue manager for use by the samples. It is a good practice to use queue managers that are different from those used by the IVT. You can then invoke the setupWMQSOAP.cmd/sh script against the target queue manager by supplying the target queue manager name as an argument, for example, setupWMQSOAP myQM.

WebSphere MQ transport for SOAP supports the use of permanent and temporary dynamic response queues for synchronous clients. This is the reason why a model queue is created by setupWMQSOAP.cmd/sh. The model queue is also used for creating dynamic response queues when using the optional asynchronous SupportPac MA0V.

It is not mandatory to use the script setupWMQSOAP.cmd/sh when configuring WebSphere MQ transport for SOAP with different queue managers. However, it does give a useful pointer to the various WebSphere MQ objects that may have to be created.

5.2 The development process

This section provides an overview of the typical process used to prepare Web Services for use with WebSphere MQ transport for SOAP with client applications. This process assumes the use of the supplied deployment tool.

Writing and preparing a service

The initial step in the process is to prepare the source for the service.

For Microsoft .NET Web Services, the service source can be C#, Visual Basic, or any other Microsoft .NET CLR language. Prepare an asmx file for this source, which can either have the source inline or reference the source as an external file.

For Axis services, the service must be written in Java.

A Microsoft .NET or Java Web Service that has already been written and used with Hypertext Transfer Protocol (HTTP) transport can be taken and used without modification. However, when using complex objects, ensure that the objects used for input and output arguments to the service are supported by the Microsoft .NET or Axis infrastructure. See 4.4.2, "Interoperability" on page 37.

Accessing SOAP/WebSphere MQ transport features with Universal Resource Indicator

A WebSphere MQ service is identified by an URI that is specific to WebSphere MQ transport. This URI helps exploit the various features and parameters available within the transport. The URI is set either at the time of deployment or when a client application starts.

You can manage the following transport-specific features through the settings in the URI:

- The name of the target service
- The name of the request queue and the target queue manager
- The name of the response queue
- The name of the connect queue manager. This is the queue manager to which the client application connects. This is different from the target queue manager.
- The type of WebSphere MQ binding, for example, server binding or client binding
- Connect options, if making a WebSphere MQ client connection
- The timeout value. This is the time for which a client waits for a response message.
- The WebSphere MQ time-to-live parameter. This allows the expiry time of the message to be set. The default is an unlimited lifetime.
- The message persistence
- The message priority

The settings for many of these options can be left as default values. However, when preparing client applications for use with WebSphere MQ transport for SOAP, it is worth taking the time to consider these options so that the transport is configured correctly and optimally for the target environment.

Details about the options that may be set in the URI, together with an explanation of the URI syntax is provided in 5.4.2, “The SOAP/WebSphere MQ Universal Resource Indicator” on page 65.

Deploying the service with amqwDeployWMQService

After preparing the service code, deploy the service to the hosting Microsoft .NET or Axis Web Services infrastructure. An example deployment utility is provided with WebSphere MQ. This utility performs the following actions:

- It defines the service to the hosting Microsoft .NET or Apache Axis infrastructure.
- It creates a proxy source that the client applications must call in order to access the service. Java proxies are compiled by the utility but Microsoft .NET proxies are not.
- It performs the WebSphere MQ configuration that is necessary to implement WebSphere MQ as a transport for this service. This configuration includes the creation of the request queue for the service and scripts that can be used to start a WebSphere MQ transport for SOAP listener.

Refer to 5.4, “The deployment process” on page 59 for more details on deployment.

Writing and preparing the client code

You can use the existing client code that was used with the HTTP transport with the WebSphere MQ transport with minor modifications. The use of WebSphere MQ transport must be indicated through a single method call which registers the transport to the Microsoft .NET or Axis infrastructure. This is essential for the infrastructure to recognize the jms: prefixed URI and invoke the SOAP/WebSphere MQ sender code.

For Microsoft .NET clients, the registration call is:

```
IBM.WMQSOAP.Register.Extension();
```

For Axis clients, the registration call is:

```
com.ibm.mq.soap.Register.extension();
```

In the case of Microsoft .NET, link the client to the proxies that were generated at deployment. In the case of Java, the CLASSPATH must be set in such a way that the proxies can be correctly located and loaded.

Calling amqwClientConfig.cmd/sh (Axis services only)

When configuring Axis clients and when deploying a service for the first time, run the amqwClientConfig.cmd/sh script from the directory the service was deployed from. This script is located in the WebSphere MQ bin directory. The purpose of this script is to use a deployment descriptor file to define the com.ibm.mq.soap.transport.jms.WMQSender class as the implementation of the WebSphere MQ transport for SOAP sender. This definition is made in the client-config.wsdd file, which is created in the directory the service is deployed from, and is independent of the actual service itself. It is therefore, not necessary to run the amqwClientConfig.cmd/sh script every time a service is redeployed. However, before attempting to invoke the service from a client, run it once in each of the directories from which a service is deployed. If this is not done, a runtime error stating that the transport jms: cannot be found may occur when executing the client.

By way of example, amqwClientConfig.cmd/sh is used in the script regenDemo.cmd/sh that is provided with WebSphere MQ V6. The regenDemo.cmd/sh script prepares and deploys the samples supplied with the product and is used by the IVT.

Listener activation

After preparing and deploying the service and preparing the client for use with the proxies generated during deployment, there is one final task you must perform before executing the service. This is ensuring that a WebSphere MQ transport for SOAP listener is running or is configured to run when a request for a service is issued from a client application.

The role of the listener is to monitor a request queue for an incoming request message, process that message, invoke the service through Apache Axis or Microsoft .NET, and return the response message to the response queue.

There are three ways in which you can start the listeners. These are:

- By entering the appropriate command in a DOS command window manually.
- With the special scripts generated at deployment, which can perform either of the following actions:
  - Run the listener as a stand-alone process
  - Configure the target queue manager to run the listener as a WebSphere MQ service

- By WebSphere MQ trigger monitoring, so that a listener is only initialized when a request for a service is made.

These three techniques for executing listeners are described in 5.6, “The WebSphere MQ transport for SOAP listener” on page 81.

To test a new service for the first time, use the automatically generated listener startup script, because this is the easiest method.

Refer to Chapter 4, “WebSphere Services with WebSphere MQ” on page 29 for more details about the concepts of the listener, and 5.6.2, “Methods to start listeners” on page 83 for more information about invoking it.

Client execution

The final task involves running the prepared client application. Start this by employing the same method used for a client using HTTP as a transport. Execution can be from a command line or within an integrated development environment (IDE). However, ensure the following:

- The CLASSPATH environment variable is set appropriately for the Java environment. Refer to 5.1.2, “Running the amqwsetcp.cmd/sh command” on page 51 for details on how to do this.
- The definition for the environment variable LD_LIBRARY_PATH must include a reference to the WebSphere MQ lib directory when using server binding on UNIX® or Linux® platforms. The environment variable SHLIB_PATH is used on Hewlett-Packard UNIX (HP-UX) platforms.
- The PATH environment variable is set correctly for the Microsoft .NET environment.

5.3 SOAP formatting

One of the design considerations for WebSphere MQ transport for SOAP was to decouple its implementation from the specifics of the SOAP version or the format options used in the transported messages. The transport level is responsible for posting and receiving messages from a service’s host and is not required to have a knowledge of the details about the actual SOAP formatting.

Although in theory, the transport is independent of the SOAP format, ensure that the client and the server Web Services infrastructure are compatible, so that they are able to understand and respond to each other’s SOAP messages.

WebSphere MQ transport for SOAP can process Remote Procedure Call-style (RPC) and Document-style messages. However, at the time of writing this book, limitations existed, not in the WebSphere MQ transport for SOAP implementation, but in the underlying service implementations. Following is a list of these limitations:

- RPC Literal encoding is not implemented currently.
- An Axis client cannot call a Microsoft .NET service using RPC-style encoding.
- SOAP complex type support is not implemented currently.
- Neither SOAP with attachments nor Direct Internet Message Encapsulation/Multipurpose Internet Mail Extensions (DIME/MIME) encoding are currently implemented. This is primarily a restriction in the current implementation of WebSphere MQ transport for SOAP.

Note: Former SupportPac versions of WebSphere MQ transport for SOAP made syntactic assumptions about the format of a SOAP message in order to extract details of the service that is to be invoked. This information is now flowed in the Rules and Formatting Header (RFH2) of the request message. Therefore, no assumptions are made by WebSphere MQ transport for SOAP on the SOAP message format.

5.3.1 Specifying Remote Procedure Call-style encoding or Document-style encoding

By default, Microsoft .NET creates services using Document-style encoding. Services can, however, use RPC encoding by including the SoapRpcMethod attribute in the method declaration. Alternatively, the attribute [SoapRpcService] can be used in the class definition. Example 5-1 from the WebSphere MQ transport for SOAP sample illustrates this in practice, with the method getQuote being declared to use RPC-style encoding and the method getQuoteDOC being declared to use the default Document-style encoding.

Example 5-1 Setting RPC-style encoding or Document-style encoding in service code

```
//RPC method
[WebMethod] [SoapRpcMethod]
public float getQuote(String symbol) {
if (symbol.ToUpper().Equals("DELAY")) Thread.Sleep(5000);
return 88.88F;
}

//Document style method
```

```
[WebMethod]
public float getQuoteDOC(String symbol) {
return 77.77F;
}
```

SoapAction is a SOAP header that is required by Microsoft .NET in order to be able to invoke a service. In Apache Axis V1.1, the specification of the SoapAction is not required.

WebSphere MQ V6 takes the approach that the sender passes SoapAction in a request message if it is present in the SOAP message generated by the infrastructure. To do this, WebSphere MQ sets the SoapAction field in the RFH2 header of the request message. Because Microsoft .NET mandates the use of SoapAction, the Microsoft .NET SOAP/WebSphere MQ listener rejects, with a report message, incoming requests, without this parameter or incoming requests that are empty in the request message RFH2.

SoapAction is not mandated by the Axis listener. This is because current versions of Axis do not require the SoapAction attribute to be explicitly set in order to identify and use the deployed services. If the Axis listener determines that the attribute is set in the RFH2 header of an incoming request message, its use is honored and passed into the Axis infrastructure. This is to ensure consistency with the approach used in Microsoft .NET and to maintain compatibility if future versions of Axis make use of SoapAction. See 5.10, “WebSphere Application Server and CICS Transaction Server interoperability” on page 100 and 4.4.2, “Interoperability” on page 37 for more details.

5.4 The deployment process

Deployment is essentially the process of configuring the host Web Services infrastructure to recognize the prepared Web Service. After deployment, clients can invoke the service by using a proxy class generated by the deployment process. Java clients can also invoke the Web Service directly with low-level calls or by using a Web Services Description Language (WSDL) configuration file. On the server side, the infrastructure recognizes the calls to the service and is able to call it as specified in the SOAP message.

The WebSphere MQ transport for SOAP deployment procedure is invoked with a script called amqwdeployWMQService.cmd/sh. This is a simple wrapper script that is used to invoke the Java program com.ibm.mq.soap.util.DeployWMQService. It is a simple console-based utility. There are currently no graphical implementations or other means to access it directly from IDE environments. The utility is provided as a sample because it is

recognized that, in practice, it may be necessary to adapt the deployment process to local requirements. Refer to 5.5, “Customizing the deployment process” on page 73 for further details about customized deployment.

Although it is not necessary to use a deployment utility such as the one provided with the product, it is simpler to do this rather than undertake the various deployment steps manually. The provided utility is implemented by IBM in Java language in order to minimize platform-dependency issues across the range of distributed platforms.

The deployment utility operates in a *bottom-up* manner, meaning that it starts with an implemented class for the service. This is illustrated in Figure 5-1.

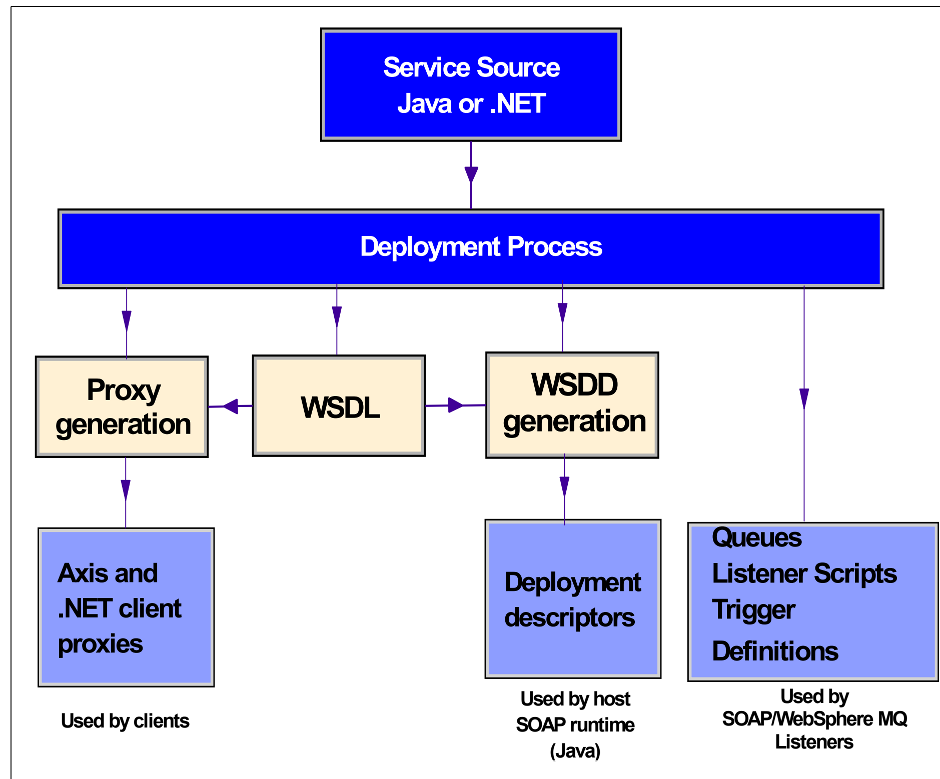


Figure 5-1 WebSphere MQ transport for SOAP deployment

The main activities the deployment utility undertakes are:

- Validating the supplied URI
- Preparing the WSDL
- Deploying the service from the WSDL

- Generating client proxies from the WSDL. The use of proxies simplifies the process of invoking the Web Service.
- Preparing a script to invoke a Microsoft .NET listener on the Web Service platform.
- Configuring WebSphere MQ with the required queues and processes necessary to implement the service.

These steps can be run individually through the utility for alternative deployment scenarios.

In practice, deployment is highly dependent on the target environment. The deployment utility that is provided acts as a useful guide. Developers with a more complex deployment scenario must build their own deployment processes in order to match their requirements more closely. The source code for the Java deployWMQService utility is therefore included with WebSphere MQ transport for SOAP in the Tools\soap\samples subdirectory in order to help with this process. In many instances, however, a customized deployment procedure may be more easily created by building deployment scripts based on a capture of the commands the supplied deployment tool runs. To view the commands that are run, use the -v option. Refer to 5.5, "Customizing the deployment process" on page 73 for more details on customized deployment.

The Queues, Listener Scripts, Trigger Definitions shown in Figure 5-1 illustrate the WebSphere MQ configuration activities performed by the deployment utility. This only undertakes a basic level of WebSphere MQ configuration, such as creating the request queue, generating scripts to start and stop the Microsoft .NET or Java listeners, and setting up process definitions to enable these to be started with trigger monitors. Further configuration may be required in other situations, for example:

- To create channel definitions and enable communication between queue managers located on different machines. This is necessary when the client is operating with server binding to a service on a different machine.
- To create a server connection channel where the client application is to invoke the service with WebSphere MQ client binding and there is no local queue manager in the client. It is necessary to create and configure a channel to enable the client and the server to communicate.
- To configure Secure Sockets Layer (SSL) communication where required. Configuration requirements depend on whether the client is operating with server or client binding. Modify the WebSphere MQ queue manager and channel definitions to use SSL. Before enabling SSL, complete several configuration actions. Configuration involves the creation of digital certificates and stores to hold the certificates. These steps are described in Chapter 6, "Security" on page 107.

The deployment utility is called from the directory in which the source is located. This is also the directory from which the service is started. The deployment procedure creates various directories and files within a subdirectory called *Generated*, which is created in the directory the service was deployed from.

The deployment utility performs the following actions:

- For Java services, it compiles the source into the classes subdirectory, for example, generated\server\soap\server\StockQuoteAxis.class.
- It generates the appropriate WSDL in the file generated\<classname>_Wmq.wsdl, for example, generated\javaDemos.server.StockQuoteAxis_Wmq.wsdl.
- For Java services, it prepares the deployment descriptor files, <*classname*>_deploy.wsdd and <*classname*>_undeploy.wsdd, and deploys into the execution directory to create or update server-config.wsdd, for example:
  - generated\server\soap\server\StockQuoteAxis_deploy.wsdd
  - generated\server\soap\server\StockQuoteAxis_undeploy.wsdd
  - server-config.wsdd
  - client-config.wsdd
- It generates the appropriate proxies for Java, C#, and VisualBasic (VB) from the WSDL. On Windows platforms, proxies are generated in Java, VB, and C# regardless of the language in which the service is written. The package and file directories for Java proxies reflect the original package. The C# and VB proxies are placed directly into the generated directory, for example:
  - Java proxies may be located in generated\client\remote\soap\server\StockQuoteAxisServiceLocator.java.
  - A C# proxy may be located in generated\client\StockQuoteAxisService.cs.
  - A Visual Basic proxy may be located in generated\client\StockQuoteAxisService.vb.
- It compiles the Java proxies into the appropriate directory beneath the generated\remote directory, for example, generated\remote\soap\server.
- It creates the WebSphere MQ queue within which messages requesting invocation of the service are passed. This queue is named as specified in the URI. If the request queue name is not specified in the URI, the queue name is generated. See 5.4.2, “The SOAP/WebSphere MQ Universal Resource Indicator” on page 65 for details about the SOAP/WebSphere MQ URI, and how this is performed.

- It prepares command files to start and stop the listener that monitors this request queue and begins the process of service invocation. These files are placed in the generated\server directory, for example:
  - generated\server\startWMQNListener.cmd
  - generated\server\endWMQNListener.cmd

  Note: Scripts are written either to start the listener directly or for it to be started as a WebSphere MQ service according to the options provided to the deployment utility.

- It prepares WebSphere MQ definitions that permit a listener process to be automatically triggered, for example:
  - WebSphere MQ Process Name: SOAPN.demos
  - WebSphere MQ Trigger Initiation Queue: SOAP.INITQ

  Although these definitions are automatically created, the use of triggered listeners are optional.

  The WSDL and the proxies generated from it have the appropriate WebSphere MQ URI set within it for the service, for example, jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=(connectQueueManager(WMQSOAP.DEMO.QM))&initialContextFactory=com.ibm.mq.jms.Nojndi&targetService=StockQuoteDotNet.asmx&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE.

  In this URI, the service being accessed is a .NET service StockQuoteDotNet.asmx. The request queue is SOAPN.demos, and both the destination queue manager and the connect queue manager are WMQSOAP.DEMO.QM. The response queue is SYSTEM.SOAP.RESPONSE.QUEUE. This is the default name for the response queue. Therefore, the replyDestination attribute can be omitted from the URI.

  When deploying Java services, the deployment utility creates C# proxies for Microsoft .NET clients. For this reason, it is necessary to have the Microsoft .NET Framework software development kit (SDK) installed even if there is no requirement to develop a Microsoft .NET client.

Note: The name of the compiled proxy source file generated by the deployment utility is the same as the name of the service. The proxy is created in the directory generated/client/remote/<package>/MyClass.class. The service is compiled by the deployment utility in the directory generated/server/<package>/MyClass.class. Ensure that these are defined appropriately in the CLASSPATH, particularly where the default CLASSPATH environment provided by amqwsetcp.sh/cmd is modified.

5.4.1 Deployment utility syntax

The syntax for the deployment utility is shown in Example 5-2.

Example 5-2 Syntax for deployment utility

```
amqwdeployWMQService -f className [-a integrityOption] [-b bothresh]
[-c operation] [-i passContext] [-n num] [r] [-s] [-tmp programName]
[-tmq queueName] [-u URI] [-v] [-x transactionality] [-?] [SSL options]
```

The -f parameter is the only mandatory option to the utility. This specifies the name of the service. For Java services, the path name specified to the utility must match the package name, for example, if the source file defines the service as being in the myService package, the source must be located in a directory called myService that is relative to the deployment directory. The path name may be specified to the utility either with directory separators (forward or backward slashes) or with class element separators (periods). For a Microsoft .NET service, the service file can be specified with a directory. However, the Java proxies generated by the utility for a Microsoft .NET service are always placed in the package called dotNetService. Therefore, either reference the proxy with this package name in the client, or import the package.

For more details about the deployment utility options, refer to the *WebSphere MQ transport for SOAP*, SC34-6651.

5.4.2 The SOAP/WebSphere MQ Universal Resource Indicator

A WebSphere MQ service is identified by an URI prefixed with jms. A typical URI for a Microsoft .NET service may look as that shown in Example 5-3.

Example 5-3 A typical URI for a Microsoft .NET service

```
jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=co
nnectQueueManager(WMQSOAP.DEMO.QM)&replyDestination=SYSTEM.SOAP.RESPONS
E.QUEUE&targetService=StockQuoteDotNet.asmx&initialContextFactory=com.i
bm.mq.jms.Nojndi
```

This URI notation is used instead of the http: style of URI that is used when using HTTP as the Web Service transport protocol. Although both forms of the URI essentially define a service that is to be started, it is clear from Example 5-3 that the SOAP/WebSphere MQ URI has a syntax that is totally different from the http: style URI. Along with defining the service to be started, the SOAP/WebSphere MQ URI also defines the various parameters necessary to specify the precise routing of the request and response messages over WebSphere MQ.

In a typical scenario, the URI is first specified at the time of deployment with the -u parameter, as illustrated in the syntax of the deployment utility provided in 5.4.1, “Deployment utility syntax” on page 64. It is possible to override the URI specified in the WSDL or proxy from the client code. This is not a normal practice in a production environment, but is useful in various testing scenarios.

The deployment process causes the URI to be set into:

- The generated WSDL
- The generated client proxy code
- The scripts that are used to start the listeners

The deployment utility that is provided in WebSphere MQ V6 allows only a single URI to be specified as an input parameter. This URI is assumed for use both in the client and the listener. In general, there is no reason why different URIs cannot be used in the client and the listener. This is required if, for example, a WebSphere MQ client connection has to be enforced at the client-end and a server connection has to be enforced at the listener-end. To use different URIs in this manner, perform the following actions:

- The client must override the URI at runtime
- The deployment process must be modified
- The deployment utility must be run twice, with the required components manually extracted from each deployment

This does not mean that it is always necessary to use two different URIs if you use different binding types in the sender and listener, because, with the default binding=auto option on the URI, WebSphere MQ transport for SOAP first attempts a server connection, and if that fails, tries a client connection. It is therefore, possible for a common URI to result in a client connection at one end of the transport and a server connection at the other end. However, if it is necessary to ensure specific connection types at each end, different URIs must be used. See , "Uniform Resource Indicator syntax" on page 66 for more details.

A client program calls the appropriate Microsoft .NET or Apache Axis Web Services framework the same way that it does for HTTP transport. The Axis or Microsoft .NET Framework marshals the call into a SOAP request message exactly the way it is done for SOAP/HTTP. When the framework identifies the jms: URI, it calls the WebSphere MQ transport sender code. This sender code is accessed from the client process either through amqsoap.dll for Microsoft .NET services, or through com.ibm.mq.soap.jar for Axis clients. The sender places the SOAP message in a request queue according to the various options specified in the URI. The provided listeners, SimpleJavaListener (for Java) or amqwSOAPNETListener (for Microsoft .NET), monitor the request queues, start the service, and return the response through the response options that are specified in the URI.

Tip: In earlier SupportPac editions of WebSphere MQ transport for SOAP, wmq: was the URI prefix. However, this has changed in WebSphere MQ V6 in order to facilitate interoperability with WebSphere Application Server. When migrating services from the SupportPac environment to the V6 environment, it is therefore, necessary to redeploy the service and rebuild client applications to switch over to the URI's new form.

Uniform Resource Indicator syntax

The URI syntax takes the following form:

```
jms:/queue?name=value&name=value...
```

Here, *name* is a parameter name and *value* is an appropriate value. The *name=value* element can be repeated any number of times, with the second and subsequent occurrences preceded by an ampersand (&).

Parameter names and values are listed in this section. Parameter names are case-sensitive, as are the names of WebSphere MQ objects. If any parameter is specified more than once, the final occurrence of the parameter takes effect. This allows client applications to override parameter values by appending to the URI. If any additional unrecognized parameters are included, they are ignored.

The *name=value* element can take the following values:

- destination=<*requestQueueName*>

  This parameter, which is required, must be the first parameter in the URI after the initial jms:/queue string.

  The name of the request queue must either be a WebSphere MQ queue name or a queue name and queue manager name connected by the @ symbol, for example, SOAPN.trandemos@WMQSOAP.DEMO.QM.

  > **Note:** WebSphere MQ Publish/Subscribe is not currently supported.

- connectionFactory=*name(value)name(value)...*

  Here, *name* is a subparameter name and *value* is an appropriate value, and the *name(value)* element is repeated as necessary. There are no separators between the *name(value)* occurrences.

  All the subparameters are optional. If none are to be set, code the connectionFactory parameter as `connectionFactory=()`. Following are the subparameter names and values:

  – connectQueueManager=<*QueueManagerName*>

    This specifies the queue manager to which the client connects. The default is blank, which indicates that the default queue manager is to be used.

  – binding=<*bindingType*>

    This specifies the type of binding to be used on the queue manager connection. The options here are:

    • auto

      If no client type attributes are specified and no binding type is specified, the default is auto, which means that the client attempts a server connection first. If this is not successful, a client connection is attempted.

    • client

      If this option is specified, the sender code assumes a client type binding. Use this option in situations where you know that a server binding type connection is not appropriate.

      If the binding option is not specified, but the options appropriate to a client binding are specified, such as clientConnection, a client binding type is assumed by default.

  - server

    If server is specified as the binding type, the client does not attempt a client binding connection if the server connection fails. If this option is used, but the URI contains the clientChannel or clientConnection string, the URI is not accepted by SOAP/WebSphere MQ and an exception is thrown.

  - xaclient

    This option applies to Microsoft .NET only. If you are making a client binding connection to a queue manager, it is not possible for a client using that connection to participate in a two-phase commit transaction using an external or non-WebSphere MQ transaction coordinator unless the WebSphere MQ Extended Transactional Client is installed. The xaclient binding option enables the use of the extended transactional client by setting the value of the NMQ_MQ_LIB to mqic32xa.dll environment variable.

    The xaclient option does not apply to the Axis environment because in that environment, WebSphere MQ is the only supported transaction coordinator.

    The SOAP/WebSphere MQ sender checks the URI for any inconsistencies in the specified options, for example, if the URI specifies binding=server, but also has client type parameters such as clientConnection= or security parameters specified, an error message is shown by the SOAP/WebSphere MQ sender, and the request fails.

– clientChannel=<*channelName*>

  This specifies the channel to be used when a WebSphere MQ transport for SOAP client makes a WebSphere MQ client connection. The default value is null. If the clientConnection keyword is specified, a value must be given for clientChannel.

– clientConnection=<*connectionString*>

  This specifies the connection string to be used when a SOAP client makes a WebSphere MQ client connection. For TCP/IP, this is in the form of either a host name, for example, MACH1.ABC.COM, or the network address in IPV4 format, for example, 19.22.11.162, or IPV6 format, for example, fe80:43e4:0204:acff:fe97:2c34:fde0:3485. It must include the

port number if a port number other than 1414 is to be used. For Java clients, the round brackets encompassing the port number must be escaped by specifying "(" as %2528 and ")" as %2529. The connectionFactory example shown in Example 5-4 illustrates this.

Example 5-4 Example of connectionFactory

```
&connectionFactory=(connectQueueManager(QM_WAS)binding(client)clientCha
nnel(WAS.JMS.SVRCONN)clientConnection(9.1.39.93%25281415%2529))
```

For Microsoft .NET clients, it is not necessary to escape the brackets in this manner. However, even if they are escaped, there are no negative effects.

If Secure Sockets Layer (SSL) is being used, add further SSL-specific subparameters. Refer to "Secure Sockets Layer in the Universal Resource Indicator" on page 135.

- initialContextFactory=<*initialContextFactory*>

  This parameter is required and must be set to `com.ibm.mq.jms.Nojndi`. This is for compatibility with WebSphere Application Server and other products.

- timeout=<*timeoutValue*>

  This is the time (in milliseconds) that the client waits for a response message. It overrides any values set by the infrastructure or the client application. If this is not specified, the application value (if specified) or the infrastructure default is used.

- targetService=<*serviceName*>

  This option is mandatory for accessing Microsoft .NET and WebSphere Application Server services. In the Microsoft .NET environment, this option makes it possible for a single SOAP/WebSphere MQ listener to be able to process requests for multiple services. These services must be deployed from the same directory. It is optional for Java services because the Axis infrastructure permits SOAP/WebSphere MQ listener to access multiple services. If it is specified in the Axis environment, it overrides the default Axis mechanism.

  The value for this parameter is a service name. For a Microsoft .NET service, the service name must be specified with no directory qualification because Microsoft .NET services are always assumed to be located directly within the deployment directory, for example, targetService=myService.asmx. For a Java service, the service name must be fully qualified, for example, targetService=javaDemos.service.StockQuoteAxis.java

- timeToLive =<*timeToLive*>

  This specifies the expiry time of the message in milliseconds. The default is 0, which indicates an unlimited lifetime.

  Note: No relationship is enforced between timeout and expiry.

- persistence=<*messagePersistency*>

  This specifies message persistence. Following standard Java Message Service (JMS) conventions, this is specified as a number with meanings:

  – 0 means no persistence is specified. WebSphere MQ treats this as PersistenceAsQDef. This is the default.

  – 1 means the message is nonpersistent.

  – 2 means the message is persistent.

- priority=<*messagePriority*>

  This specifies the message priority. Valid values are in the range of 0 - 9, with 0 being the lowest and 9 being the highest. The default is environment-specific. For WebSphere MQ, the default is 0.

- replyDestination=<*responseQueueName*>

  This is the queue at the client side that is used for the response message. The default setting is SYSTEM.SOAP.RESPONSE.QUEUE.

5.4.3 Request queues

The deployment process creates the request queues that are to be used with the service. If the request queue is specified in the optional URI given to the deployment procedure, then that queue name is assumed. If no request queue name is specified in the URI, the queue name is generated. The name is generated from the name of the service specified to the deployment process with the -f parameter. It is formed by removing the file name extension and replacing any directory separator characters with spaces. Embedded spaces are replaced with underscore (_) characters. On Windows, colon (:) characters are replaced with periods (.). The prefix SOAPN. is then applied for Microsoft .NET services and SOAPJ. for Axis services.

If the generated queue name exceeds 48 characters and no drive prefix is specified in the service path, the deployment utility truncates the name by retaining the SOAPx. prefix and taking the right-most 42 bytes in the queue name. If a drive prefix is specified, the name is based on the SOAPx. prefix combined with the drive prefix, with a period replacing the colon, and the right-most 40 bytes.

In instances where queue names are generated by the deployment utility, the truncation process may mean that the generated names are not unique within the target queue manager, in which case, the deployment fails. The names may not be readable or meaningful if they have been truncated. To avoid this lack of readability, the queue name must either be specified or the directory structure altered so that the name is not truncated.

Request queue validation

The deployment utility carries out checks to safeguard against the possibility of a request queue name already in use. The request queue name is either specified in the URI given to the deployment utility or it is generated. In both these scenarios, it is possible that the name is not unique. If the request queue name is not unique, the same queue is used for request messages of different services. However, such a configuration is not supported by WebSphere MQ transport for SOAP.

A check is first made as to whether or not the queue exists. This check is made by trying to open the queue. The deployment process then makes the following checks to determine whether or not the deployment is allowed to proceed:

- Looks for the presence of a listener startup script in the subdirectory Generated\server, which is a relative to the directory from which the deployment utility is being run.
- Scans the corresponding file for the URI that is specified to the command that runs the listener. This is done by extracting the first line containing either the word SimpleJavaListener or amqwSOAPNETListener, and scanning that line for the URI provided with the -u option.
- Compares the URI given to the deployment utility and any URI found in the listener startup script.

Depending on the outcome of these checks, the deployment process is either allowed with or without a warning, or is failed with an error message. The possible outcomes are summarized in Table 5-1.

Table 5-1 Summarizing request queue validation at deployment

	Queue exists	**Queue does not exist**
Script not found	OK	ERROR
Script found, but URI does not match	ERROR	ERROR
Script found and URI matches	WARNING	OK

If the script is deleted but the queue still exists, the listener exits with an error message to cover the case where a queue name matches the one that is being used for another service. If the queue is not in use, the simplest thing to do is to delete the queue and restart the deployment process.

The validation assumes that the -u option is on the same line as the command invoking the listener. Therefore, the use of continuation lines on the listener command causes the validation to fail.

If the deployment tool fails to validate the request queue and exits with an error or warning, it exits without any of the deployment steps being taken.

5.4.4 Response queues

The default name for the response queue is SYSTEM.SOAP.RESPONSE.QUEUE. This queue is generated by the setup script setupWMQSOAP.cmd that is provided for use with the samples. It is not necessary to use this name for the response queue. However, if an alternative queue is to be used, create it manually before using a client to request a service.

Users typically want to have one queue manager servicing sender requests and responses and a second queue manager servicing the listener. These two queue managers may be on separate systems. This is the reason why the deployment utility does not create the response queue.

> **Note:** When using separate queue managers this way, create the required transmission queues after running the deployment process.

5.4.5 Queue manager connection types

In many configurations, the types of WebSphere MQ connections made by a client application may be different from those made by the listener, for example, it may be necessary to have a client application use a WebSphere MQ client connection to a remote machine that is hosting the Web Service and running the WebSphere MQ transport for SOAP listener. On the hosting machine, the queue manager may be local. Therefore, a server connection is made to the local queue manager. The use of different connection styles does not necessarily mean that different WebSphere MQ URIs must be used in the client and the listener. In some situations, the use of the binding=auto option on the URI may be sufficient to cover the two environments. As described in , “Uniform Resource Indicator syntax” on page 66, the use of auto on the binding option means that a server connection is attempted first. If this fails, a client connection is attempted. Thus, it is possible for a common URI to be used in the two environments.

However, this practice is not always acceptable because the connection styles in the two environments may have to be explicitly qualified. This is required, for example, to mandate a server connection on a listener and a client connection on a SOAP client without using the binding=auto option. The supplied deployment process does not currently support the specification of different URIs for the client and the server environments. In the event that the client and the server environments have different URIs, several configuration options are available:

- Use the deployment tool twice, once in the server environment and once in the client environment.
- Work from a server-based deployment as follows:
  a. Run the deployment tool once for the server environment.
  b. Copy the generated proxies to the client environment.
  c. Change the URIs in the proxies to the required URIs before compiling them, or override the URI to be used at run time.
- Work from a client-based deployment as follows:
  a. Run the deployment tool once for the client environment.
  b. Copy the generated subdirectory structure to the server environment.
  c. Edit the listener scripts to use the required server URI.
- Build a customized deployment tool to allow the provision of separate URIs. In this case, it is still necessary to copy the generated proxies from the server system where the service is deployed, to the client environment.

The next section, 5.5, “Customizing the deployment process” on page 73, provides details about customizing the deployment process.

5.5 Customizing the deployment process

The deployment utility provided with WebSphere MQ transport for SOAP is intended as a sample and there may be various reasons why, in practice, it may be necessary to implement a custom deployment process. Following are some of the common reasons a custom utility must be built:

- The ability to specify separate URIs for the proxy and the listener
- The facility to deploy the service from WSDL in a top-down manner
- The ability to change parameters to the Axis utilities that are called internally. This is required, for example, if services return or pass arguments that are in a different package from the service.

Customize the deployment process in one of the following ways:

- By executing the various deployment steps manually
- By capturing the commands used by the supplied deployment process and turning these into a script that may be directly edited and customized as required
- By modifying the sample Java deployment utility directly and recompiling it. This is the best choice when you have to make general changes to the process, such as deploying top-down rather than bottom-up. It can also be considered in situations where it is acceptable to make more assumptions about the target environment, for example, where it is not necessary to carry out the checks described in “Request queue validation” on page 71.

5.5.1 Illustrating the Microsoft .NET customized deployment

Consider a situation where you have to provide a customized deployment process for a Microsoft .NET service.

Run the deployment utility with the -v parameter to capture the commands that are issued by it, as shown in Example 5-5.

Example 5-5 Running the deployment utility

```
amqwdeployWMQService -u
"jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=c
onnectQueueManager(WMQSOAP.DEMO.QM)&initialContextFactory=com.ibm.mq.jm
s.Nojndi" -n 10 -f StockQuoteDotNet.asmx -v 2>&1 > deploy.log
```

The output of the deploy utility is captured in the file deploy.log and the contents of this file are as shown Example 5-6.

Example 5-6 Capturing the commands issued by the deployment utility for a Microsoft .NET service

```
RunCommand: Command = amqswsdl
jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&amp;connectionFactory=(connectQueu
eManager(WMQSOAP.DEMO.QM))&amp;initialContextFactory=com.ibm.mq.jms.Nojndi&amp;target
Service=StockQuoteDotNet.asmx&amp;replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE
StockQuoteDotNet.asmx generated\StockQuoteDotNet_Wmq.wsdl5724-H72 (C) Copyright IBM
Corp. 1994, 2004.  ALL RIGHTS RESERVED.

RunCommand: MQ Command = DEFINE QL('SOAPN.demos') BOTHRESH(3) REPLACE
RunCommand: Command = cmd /V:ON /C "amqwcallWSDL.cmd ..\StockQuoteDotNet_Wmq.wsdl &&
exit /b !errorlevel!"
RunCommand: Command = cmd /V:ON /C "amqwcallWSDL.cmd ..\StockQuoteDotNet_Wmq.wsdl
/language:VB && exit /b !errorlevel!"
```

```
RunCommand: Command = java com.ibm.mq.soap.util.RunWSDL2Java --output
generated\client\remote -p dotNetService generated\StockQuoteDotNet_Wmq.wsdl
RunCommand: Command = javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNet.java"
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetLocator.java"
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetSoap.java"
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetSoapStub.java"
```

The output of the commands are turned into a script as shown in Example 5-7.

Example 5-7 A sample Microsoft .NET deployment script

```
@ rem Configure the PATH and CLASSPATH
call amqwsetcp

@ rem Delete the generated directory
del /s /q generated\*.*

@ rem Now re-create the generated directory and required sub-directories
mkdir generated\server
mkdir generated\client\remote

@ rem Generate the WSDL for the service
amqswsdl
"jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&amp;connectionFactory=(connectQue
ueManager(WMQSOAP.DEMO.QM))&amp;initialContextFactory=com.ibm.mq.jms.Nojndi&amp;targe
tService=StockQuoteDotNet.asmx&amp;replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE"
StockQuoteDotNet.asmx generated\StockQuoteDotNet_Wmq.wsdl

@ rem Create the request queue
echo DEFINE QL('SOAPN.demos') BOTHRESH(3) REPLACE | runmqsc WMQSOAP.DEMO.QM

@ rem set up the listener script (in this case from one that was generated earlier)
copy startWMQNListener.cmd generated\server\startWMQNListener.cmd

@ rem Generate the C# proxy
cmd /C amqwcallWSDL.cmd ..\StockQuoteDotNet_Wmq.wsdl

@ rem Generate the Visual Basic proxy
cmd /C amqwcallWSDL.cmd ..\StockQuoteDotNet_Wmq.wsdl /language:VB

@ rem Generate the Java proxy
java com.ibm.mq.soap.util.RunWSDL2Java --output generated\client\remote -p
dotNetService generated\StockQuoteDotNet_Wmq.wsdl
```

```
@ rem Compile the Java proxy files
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNet.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetLocator.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetSoap.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\dotNetService\StockQuoteDotNetSoapStub.java"

@ rem Compile the client
csc "/lib:%WMQSOAP_HOME%\bin" /r:amqsoap.dll
"%WMQSOAP_HOME%\Tools\soap\samples\dotnet\SQCS2DotNet.cs" generated\client\*.cs
```

Edit the log file to turn it into a workable script. The script illustrates the steps involved in deploying a Microsoft .NET service:

1. Generating the WSDL for the service from the WebSphere MQ transport for SOAP amqswsdl utility.
2. Configuring the WebSphere MQ request queue associated with the service.
3. Setting up a script to start the Microsoft .NET listener.
4. Generating the proxies required for a client.
5. Compiling the proxies and then compiling the client using these proxies.

Notes:

- Delete the contents of the generated subdirectory. The key directories are then recreated.
- Quotes around the URI argument to the amqswsdl utility is a must.
- In this example, a script to start the listener is copied into place from one that was generated and saved earlier.
- In this example, only a Microsoft .NET client is generated. Therefore, the Java proxies did not have to be generated using the Axis RunWsdl2Java utility. However, Java proxies are included for demonstration purposes.

5.5.2 Illustrating the Axis customized deployment

Follow the same method that is used to write a customized deployment script for a Microsoft .NET service for an Axis service too.

The output of the deployment utility is captured using the -v option, as shown in Example 5-8.

Example 5-8 Output of deployment utility

```
amqwdeployWMQService -u "jms:/queue?destination=SOAPJ.demos@WMQS
OAP.DEMO.QM&connectionFactory=connectQueueManager(WMQSOAP.DEMO.QM)&init
ialContextFactory=com.ibm.mq.jms.Nojndi" -n 10 -f
soap\server\StockQuoteAxis.java -v 2>&1 > deploy_axis.log
```

The commands that are issued are captured in the log file, as shown in Example 5-9.

Example 5-9 Capturing the commands issued by the deployment utility for an Axis service

```
RunCommand: Command = javac -d generated\server soap\server\StockQuoteAxis.java
RunCommand: Command = java org.apache.axis.wsdl.Java2WSDL  --output
generated\soap.server.StockQuoteAxis_Wmq.wsdl --namespace
soap.server.StockQuoteAxis_Wmq --location
jms:/queue?destination=SOAPJ.demos@WMQSOAP.DEMO.QM&connectionFactory=(connectQueueMan
ager(WMQSOAP.DEMO.QM))&initialContextFactory=com.ibm.mq.jms.Nojndi&targetService=soap
.server.StockQuoteAxis.java&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE --bindingName
soap.server.StockQuoteAxisBindingSoap --servicePortName
soap.server.StockQuoteAxis_Wmq soap.server.StockQuoteAxis
RunCommand: Command = java com.ibm.mq.soap.util.RunWSDL2Java --server-side  --timeout
-1 -p soap.server --output generated\temp.server
generated\soap.server.StockQuoteAxis_Wmq.wsdl
RunCommand: Command = cmd /c java com.ibm.mq.soap.util.PatchWsdd
soap.server.StockQuoteAxis
"C:\temp\redbook\generated\temp.server\soap\server\deploy.wsdd"
generated\server\soap\server\StockQuoteAxis_deploy.wsdd
RunCommand: Command = cmd /c java com.ibm.mq.soap.util.PatchWsdd
soap.server.StockQuoteAxis
"C:\temp\redbook\generated\temp.server\soap\server\undeploy.wsdd"
generated\server\soap\server\StockQuoteAxis_undeploy.wsdd
RunCommand: Command = java org.apache.axis.utils.Admin server
generated\server\soap\server\StockQuoteAxis_deploy.wsdd
RunCommand: MQ Command = DEFINE QL('SOAPJ.demos') BOTHRESH(3) REPLACE
RunCommand: Command = cmd /V:ON /C "amqwcallWSDL.cmd
..\soap.server.StockQuoteAxis_Wmq.wsdl && exit /b !errorlevel!"
```

```
RunCommand: Command = cmd /V:ON /C "amqwcallWSDL.cmd
..\soap.server.StockQuoteAxis_Wmq.wsdl /language:VB && exit /b !errorlevel!"
RunCommand: Command = java com.ibm.mq.soap.util.RunWSDL2Java --timeout -1 --output
generated\client\remote -p soap.server generated\soap.server.StockQuoteAxis_Wmq.wsdl
RunCommand: Command = javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\soap\server\SoapServerStockQuoteAxisBindingS
oapStub.java"
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxis.java"
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxisService.java"
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxisServiceLocator.jav
a"
```

The output of the commands are turned into a script, as shown in Example 5-10.

Example 5-10 Sample Axis deployment script

```
@ rem Configure the PATH and CLASSPATH
call amqwsetcp

@ rem Delete the generated directory
del /s /q generated\*.*

@ rem Now re-create the generated directory and required sub-directories
mkdir generated\server
mkdir generated\client\remote

@ rem Compile the service code
javac -d generated\server soap\server\StockQuoteAxis.java

@ rem generate WSDL for this service
java org.apache.axis.wsdl.Java2WSDL  --output
generated\soap.server.StockQuoteAxis_Wmq.wsdl --namespace
soap.server.StockQuoteAxis_Wmq --location
"jms:/queue?destination=SOAPJ.demos@WMQSOAP.DEMO.QM&connectionFactory=(connectQueueMa
nager(WMQSOAP.DEMO.QM))&initialContextFactory=com.ibm.mq.jms.Nojndi&targetService=soa
p.server.StockQuoteAxis.java&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE"
--bindingName soap.server.StockQuoteAxisBindingSoap --servicePortName
soap.server.StockQuoteAxis_Wmq soap.server.StockQuoteAxis

@ rem generate the wsdd deploy and undeploy files for configuring the service to Axis
java com.ibm.mq.soap.util.RunWSDL2Java --server-side  --timeout -1 -p soap.server
--output generated\temp.server generated\soap.server.StockQuoteAxis_Wmq.wsdl

@ rem patch the deploy and undeploy wsdd for this service so as to use MQ as a
transport
```

```
cmd /c java com.ibm.mq.soap.util.PatchWsdd soap.server.StockQuoteAxis
"C:\temp\redbook\generated\temp.server\soap\server\deploy.wsdd"
generated\server\soap\server\StockQuoteAxis_deploy.wsdd
cmd /c java com.ibm.mq.soap.util.PatchWsdd soap.server.StockQuoteAxis
"C:\temp\redbook\generated\temp.server\soap\server\undeploy.wsdd"
generated\server\soap\server\StockQuoteAxis_undeploy.wsdd

@ rem deploy the service to axis
java org.apache.axis.utils.Admin server
generated\server\soap\server\StockQuoteAxis_deploy.wsdd

@ rem Create the request queue
echo DEFINE QL('SOAPJ.demos') BOTHRESH(3) REPLACE | runmqsc WMQSOAP.DEMO.QM

@ rem set up the listener script (in this case from one that was generated earlier)
copy startWMQJListener.cmd generated\server\startWMQJListener.cmd

@ rem generate a C# proxy
cmd /C amqwcallWSDL.cmd ..\soap.server.StockQuoteAxis_Wmq.wsdl

@ rem generate a Visual Basic proxy
cmd /C "amqwcallWSDL.cmd ..\soap.server.StockQuoteAxis_Wmq.wsdl /language:VB

@ rem generate a Java proxy
java com.ibm.mq.soap.util.RunWSDL2Java --timeout -1 --output generated\client\remote
-p soap.server generated\soap.server.StockQuoteAxis_Wmq.wsdl

@ rem compile the Java proxies
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\soap\server\SoapServerStockQuoteAxisBindingS
oapStub.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxis.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxisService.java"
javac -d generated\client\remote
"C:\temp\redbook\generated\client\remote\soap\server\StockQuoteAxisServiceLocator.jav
a"
```

Edit the log file to turn it into a workable script. The script illustrates the following steps involved in deploying an Axis service:

1. Compiling the source for the service
2. Generating the WSDL for the service from the Axis Java2WSDL utility

3. Generating the Java proxies for the service from the WebSphere MQ transport for SOAP utility RunWSDL2Java
4. Patching the deploy and undeploy wsdd files so that they are correctly configured with the WebSphere MQ transport for SOAP URI
5. Deploying the service to Axis with the Axis admin utility
6. Configuring the WebSphere MQ request queue associated with the service
7. Setting up a script to start the Java listener
8. Generating C# proxy and Visual Basic proxy for a Microsoft .NET client
9. Generating a Java proxy for a Java client
10. Compiling the Java proxies

Notes:

- Delete the contents of the generated subdirectory first. The key directories are then recreated.
- In this example, a script to start the listener is copied into place from the one that was generated and saved earlier.
- In this example, only a Java client is generated. Therefore, there is no necessity to generate Microsoft .NET proxies using the WebSphere MQ amqwcallWSDL.cmd script. However, the Microsoft .NET proxies are included for demonstration purposes.

The examples described here do *not* demonstrate actual modification of the deployment process. However, having written the deployment scripts of Axis or .NET or both by capturing the output of the supplied deployment utility, it becomes easy to perform a high-level customization of the deployment process by editing the scripts as appropriate. Use this technique to, for example, specify the use of mixed package names, as discussed in 5.5.4, “The use of mixed package names” on page 81.

5.5.3 Using complex objects in Java and Microsoft .NET

When using Axis clients with Axis services, if it is necessary to use complex objects in arguments to service calls or return from service calls, *one* of the following conditions must be met:

- The classes from which those objects are instantiated must be compatible with the JavaBeans style specification.

> **Note:** Refer to the following Web site for information about the JavaBeans specification:
>
> http://java.sun.com/products/javabeans/docs/spec.html

- A custom serializer/deserializer is written for the class.

No specifications are provided in the SOAP standards regarding the use of complex objects. Interoperability using complex objects between the Microsoft .NET and Apache Axis environment cannot be guaranteed. The user is asked to refer to the documentation provided with these products, for further details.

5.5.4 The use of mixed package names

As mentioned earlier, the deployment utility does not work where services return or pass arguments that are in a different package from the service. This is because the deployment utility forces a single package name by using the -p option on one of the Axis deployment utilities. This is used to circumvent other naming issues. Where mixed package names must be used in this manner, the deployment procedure is customized, as described in 5.5, "Customizing the deployment process" on page 73, to first capture the deployment process into a script and then change the arguments used on the Axis utilities. Accomplish the actual customization by using the -p flag on the Axis utility org.apache.axis.wsdl.Java2WSDL and replacing the -p flag on the com.ibm.mq.soap.util.RunWSDL2Java utility, and using the -n flag to map individual package names to namespace names. Refer to the Axis documentation for more information about the use of these flags.

5.6 The WebSphere MQ transport for SOAP listener

The SOAP/ WebSphere MQ listener is a stand-alone process. The listener is the principal WebSphere MQ transport for SOAP functionality, which is provided on the server side for the execution of Web Service requests that are transported over WebSphere MQ. Refer to 4.6, "SOAP/WebSphere MQ listener" on page 42 for a conceptual view of the way the listener functions.

The listener is a multithreaded program. Specifying the number of threads on startup and tailoring them according to the expected usage profile of a particular service is possible.

There are three ways in which to start a listener:

- By entering the appropriate command in a disk operating system (DOS) command prompt manually.
- With the special scripts generated at the time of deployment, which can either:
  - Run the listener as a standalone process or
  - Configure the target queue manager to run the listener on a WebSphere MQ service.
- By WebSphere MQ trigger monitoring, so that a listener is only initialized when a request for a service is made. The use of trigger monitoring is applicable to environments where there are a large number of different services. Trigger monitoring avoids the necessity for a listener to run continuously.

Details about the different listener startup methods are discussed later in this chapter.

> **Note:** In Appendix A, “WebSphere MQ using .NET classes” on page 381 of this book, the WebSphere MQ Microsoft .NET monitor is introduced as a new triggering facility provided for Microsoft .NET applications. This monitor is unsuitable for triggering the SOAP/WebSphere MQ listeners provided in WebSphere MQ V6.

A WebSphere MQ transport for SOAP listener monitors a specific request queue for service requests. A single listener is able to service requests for different services, provided they are all deployed from the same directory. This means that it is only necessary to start a single listener instance from any deployment directory structure for any service that is deployed from the directory to be invoked.

> **Note:** In the earlier MA0R SupportPac versions of WebSphere MQ transport for SOAP, it was *not* possible for a single Microsoft .NET listener to process requests for the different services deployed from the same directory.

5.6.1 Microsoft .NET listener runtime syntax

The Microsoft .NET SOAP/WebSphere MQ listener is implemented as a stand-alone program, amqwSOAPNETlistener.exe. This is located in the WebSphere MQ bin directory. The calling syntax is shown in Example 5-11.

Example 5-11 Microsoft .NET listener runtime syntax

```
amqwSOAPNETlistener -u wmqUri [-w directory] [-n numListenerThreads]
[-d msecs] [-i passContext|owncontext ] [-x none | onePhase | twoPhase]
```

The Java SOAP/WebSphere MQ listener is implemented in the com.ibm.mq.soap.axis.transport.jms.SimpleJavaListener class. It is packaged in the java\lib\com.ibm.mq.soap.jar jar file. The Java SOAP/WebSphere MQ listener calling syntax is shown in Example 5-12.

Example 5-12 Java SOAP/WebSphere MQ calling listener syntax

```
java com.ibm.mq.soap.axis.transport.jms.SimpleJavaListener -u wmqUri -a
[LowMsgIntegrity|HighMsgIntegrity|DefaultMsgIntegrity] [-d msecs] [-i
passContext|ownContext] [-n numListenerThreads] [-v] [-x
none|onePhase|twoPhase] [-?]
```

Refer to *WebSphere MQ transport for SOAP,* SC34-6651 for a full description of these arguments and options.

5.6.2 Methods to start listeners

This section describes the different ways of activating a listener.

By generating the start and stop scripts

The simplest way to start a listener is by using the startup script that is generated at the time of deployment. This script is located in the Generated\server subdirectory underneath the directory where the deployment utility is run. The script is called startWMQNListener.cmd for Microsoft .NET services and startWMQJListener.cmd/sh for Axis services.

The script starts the listener according to the options that are supplied to the deployment utility. Because the script contains the WebSphere MQ URI together with the other associated parameters, it is not necessary to provide any further parameters when invoking the script. The script as generated does not accept any input parameters. If further customization is required, edit the script. Using this script is the easiest way to start a listener when first building and testing the invocation of a service.

By manual invocation

Instead of using the generated script, start a listener manually. To do this, run the script amqwsetcp.cmd/sh, change the directory to the directory from which the service is deployed, and start the listener. In practice, there is little reason for starting a listener in this manner unless particular tests with different options or parameters have to be carried out.

By configuring a listener as a WebSphere MQ service

Configure a listener to start as a WebSphere MQ service. To do this, use the -s option at the time of deployment. When using this option, the deployment tool creates a script called defineWMQNlistener.cmd/sh. When this script is run, it defines the listener as a WebSphere MQ service and starts it. The service starts the listener using the same startWMQNListener.cmd/sh script that is used if the listener is started directly.

By using WebSphere MQ trigger monitoring

Use WebSphere MQ triggering to start listeners automatically when necessary. This is likely to be more appropriate for a production environment, where there is a potential to start multiple listeners. Triggering avoids the necessity to keep all the listeners running continuously during periods when particular services are not being started.

The deployment process creates the definitions that are necessary to implement triggering if the -tmq option is given to the deployment tool. This option specifies the name of the trigger monitor queue that is used. The deployment tool creates the queue if the queue does not already exist, along with the necessary process definitions.

Triggering configuration performed at the time of deployment causes a message to be written to the specified initiation queue when the message depth in the request queue changes from 0 to 1. This message is sent to the initiation queue as a request to start the process named by the PROCESS parameter on the server system. To activate the triggering function, start a trigger monitor on the service machine. A trigger monitor, runmqtrm, is supplied with WebSphere MQ. For more details, refer to *WebSphere MQ System Administration Guide,* SC34-6584.

Note: In WebSphere MQ V6, the name of the trigger monitor queue to be used can be defined at the time of deployment. In the earlier SupportPac versions of WebSphere MQ transport for SOAP, this was not possible.

5.6.3 Stopping a listener

The deployment process creates a script called endWMQNListener.cmd/sh for Microsoft .NET services and endWMQJListener.cmd for Axis services. Use this script to stop a listener, irrespective of the method used to start it, as described earlier.

A more automated way of stopping a listener is by supplying the -d parameter on the listener command line. This instructs the listener to exit after a specified number of milliseconds have elapsed, with no messages being received by any of the listener threads.

It is recommended that you do not interrupt a listener from the command prompt by using Ctrl+C. If a listener is interrupted in this manner, the environment within the queue manager may become unstable for several seconds afterwards.

> **Note:** In the current version, the deployment utility does not check for the -d option. This option is also not passed into the listener startup scripts. Modify the startup script manually if the -d option is to be used.

5.6.4 The role of identity context

Earlier SupportPac versions of WebSphere MQ transport for SOAP did not support the passing of identity context, which is a mechanism used by WebSphere MQ to establish an identity from which a message is sent, and to assume that personality when posting a response.

In WebSphere MQ V6, by default, both the listeners attempt to pass the identity context. This means that the identity context of the original request message is set into the response message.

It is possible that the listener does not have sufficient authority to save and pass the context. This cannot be determined when starting the listener. It can be done only when the response queue is opened to post a reply for a particular service execution. The listener does not therefore, treat an authorization failure as a serious situation. If it cannot open the response queue for saving the context from a particular request, the request message is sent to the dead letter queue. In this example, the return code in the message is set to that being returned from the failed open call on the response queue. After this, the listener continues to process other request messages, if any.

To configure a listener to run without passing the identity context, run it with the option -i owncontext. In this case, the returned context reflects the userID against which the listener is running and not the userID that created the original request message.

Refer to *WebSphere MQ Security*, SC34-6588 for an overview of the identity context mechanism.

5.6.5 Listener transactionality

Three separate areas of transactional control can be exploited when using WebSphere MQ transport for SOAP. These are:

- Transactional control of a client request

  If a request message cannot be successfully dispatched to the request queue, the entire request is backed out. Participating resources that take part in this request are also backed out.

- Transactional control of the execution of the service

  If a service request cannot be successfully started, it is backed out and the original request left on the request queue. Participating resources involved in service execution are also backed out.

- Transactional control of a client response

  If a response message cannot be successfully processed, the entire response phase is backed out. This includes any work performed by external resources participating in the transaction.

The three areas of transactional control are illustrated in Figure 5-2.

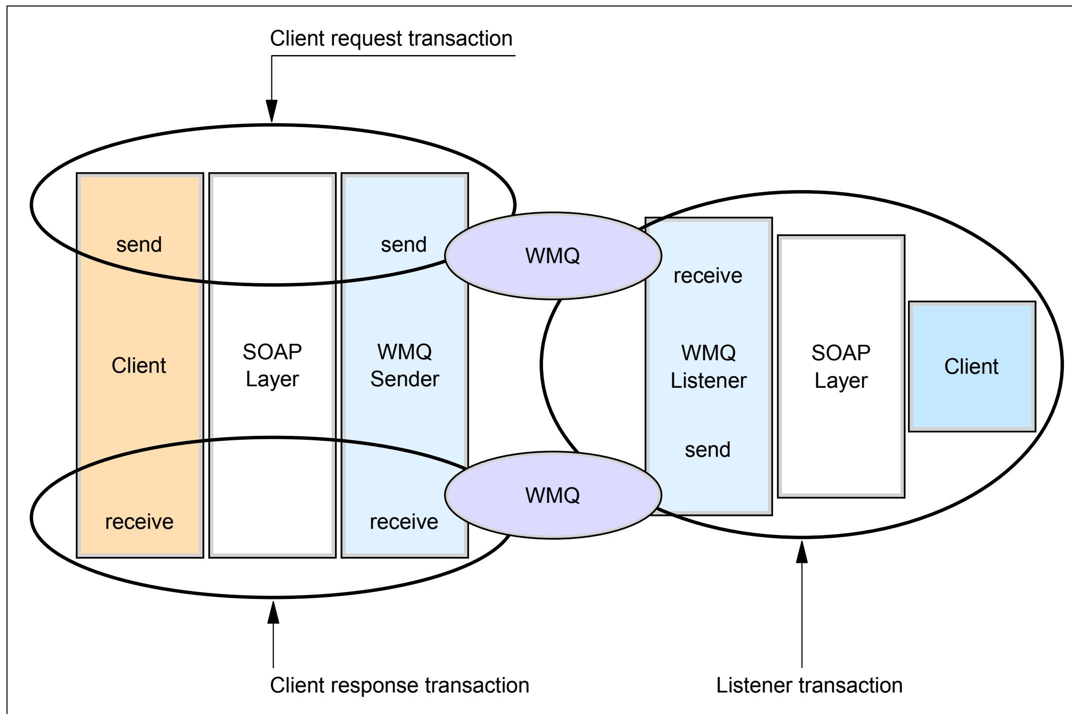


Figure 5-2 The three levels of WebSphere MQ transport for SOAP transactionality

The second option, the ability to make the service start the transactionality within the listener, is provided in WebSphere MQ V6. Listener transactionality is supported in this option.

The first and third options, that is, client transactionality, require the use of the asynchronous SupportPac MA0V. This is not currently supported. This means that although client request and response transactionality are possible by using SupportPac, neither of these actions are currently supported.

For details about client request and response transactionality, refer to Chapter 16, “Transactional functionality (MA0V)” on page 339.

A listener provides the option to control the transactionality of the service through the -x option. This can be set to one of the following values:

- onePhase

  This is the default if -x is not specified when starting the listener. onePhase transactionality means that WebSphere MQ implements transactionality within the context of WebSphere MQ only. If the listener fails to process a

service request, the request is left on the request queue. This is not the same as guaranteeing a single execution of the service. It is therefore possible that although a message itself may be backed out in the event of problems when executing a service, the processing within the service that is only partially completed, is not backed out. For this reason, if other resources, such as a database, are involved in completing the execution of the service, twoPhase transactionality is more appropriate.

- twoPhase

  With twoPhase transactionality, if the other resources pertinent to the execution of the service are coordinated and the resource managers and the service written appropriately, the message is delivered exactly once with a single committed execution of the service. When using the Microsoft .NET environment, the Microsoft Transaction Server (MTS) acts as the transaction coordinator. Using WebSphere MQ as a transaction coordinator in the Microsoft .NET environment is not supported. However, when using the Axis environment, WebSphere MQ acts as the transaction coordinator. In fact, this is the only coordinator that is supported.

- none

  If no transactionality is selected, the request message may be lost during processing, even if it is persistent. Execution of the service may or may not be completed and the status of the report and the dead letter messages cannot be guaranteed.

Note: If an invoked Java service issues WebSphere MQ calls within the same queue manager that the listener is using, the service may well be required to participate in the same transaction as the listener. In this case, make a special call in the service code to obtain the queue manager object as a reference to the queue manager connection that is used in the listener. Refer to the Appendix B, “WebSphere MQ using Java classes” on page 405 for further details.

5.7 Permanent and temporary dynamic response queues

The use of permanent and temporary dynamic response queues is supported in a situation where static queues do not have to be used. The setupWMQSOAP.cmd script creates a model queue called SYSTEM.SOAP.MODEL.RESPONSE.QUEUE, which is set to function as a permanent dynamic model queue. This queue can be used as a response queue name in a URI. However, this is not recommended because the attributes of this model queue may have to be altered. This is because this model queue is also

used by the asynchronous sender and must therefore be left with the attributes with which it was created. If using permanent or temporary dynamic response queues for synchronous clients is necessary, the creation of a specific model queue for this purpose is recommended. To create this, use SYSTEM.SOAP.MODEL.RESPONSE.QUEUE as a template, for example, by using a **runmqsc** command, as shown in Example 5-13.

Example 5-13 Creating a model queue from the SOAP default

```
define qmodel('MY.MODEL.QUEUE')
LIKE('SYSTEM.SOAP.MODEL.RESPONSE.QUEUE')
    11 : define qmodel('MY.MODEL.QUEUE')
LIKE('SYSTEM.SOAP.MODEL.RESPONSE.QUEUE'
)
AMQ8006: WebSphere MQ queue created.
```

There are no particular advantages or disadvantages to using dynamic response queues. Both have their own advantages and disadvantages. One of the benefits of using temporary dynamic queues is that they offer a better performance than the standard queues. However, remember that it is not possible to use a temporary dynamic queue in conjunction with persistent messages. In practice, the use of dynamic response queues is not likely to result in an overall performance advantage when considered against the overheads of the SOAP infrastructure. Some users may, however, enforce dynamic response queues as part of the local administrative conventions. Another reason why multiple dynamic queues are favoured is because they provide tighter security control than the single, shared nondynamic queue.

Where dynamic response queues are used, the WebSphere MQ transport for SOAP sender software automatically deletes the queue when a response is returned successfully and the queue is closed.

Note: There are separate issues relating to the use of dynamic response queues when using asynchronous WebSphere MQ transport for SOAP clients. Refer to Chapter 11, ".NET client" on page 243 for details about short-term asynchronous client, and Chapter 14, "Long-term asynchronous functionality (MA0V)" on page 303, and Chapter 15, "Implementing long-term asynchronous Web Service clients" on page 327 for details about long-term asynchronous clients.

5.8 WebSphere MQ transport for SOAP error handling

You can use two WebSphere MQ mechanisms to notify clients of failed request messages:

- The report messages, which are put into a nominated response queue
- The dead letter queue

The sender code that is provided with WebSphere MQ transport for SOAP sets the default options that specify the use of these two mechanisms. These mechanisms generate report messages in the event of a failed request message, and then discard the original message. This means that if a report message is returned successfully to the response queue, the message is not written to the dead letter queue.

If it is determined that a report message must be returned to the response queue, but the listener is unable to do so because the response queue is full, for example, the listener attempts to write the failed report message to the dead letter queue. It is, however, essential for a dead letter queue to be defined in the queue manager. If this task is not performed, no default dead letter queue name is assumed, and the message is not written to any dead letter queue.

In a typical WebSphere MQ application, customize the use of report messages and the dead letter queue handling options by using the appropriate options when placing a target message. However, in WebSphere MQ transport for SOAP, there are currently no convenient options to change these options at run time or the facility on the URI to set them. The only possibility is to write customized transport sender software. However, this subject is beyond the scope of this book.

WebSphere MQ transport for SOAP, SC34-6651 states that the listener honors cases where different options have been set. One scenario in which this may have to be done is if the listener should always place messages in the dead letter queue when report messages are returned. To do this in the current release, a customized sender must be built, an activity many people may not want to perform. This exercise too is beyond the scope of this book. In effect, writing a customized sender means that a user's implementation is not supported by IBM.

5.8.1 Report messages

Following are the scenarios in which report messages may be returned:

- An exception condition or message expiry. In these situations, WebSphere MQ automatically generates a report message.
- If the request message format is unrecognized, a report message is specifically returned by a WebSphere MQ transport for SOAP listener.
- The Microsoft .NET listener returns a report message if the targetUri or SoapAction fields are missing from the RFH2 section of a request message.

  The Axis listener does not enforce SoapAction to be set, but does require the targetUri. SoapAction is set automatically by the WebSphere MQ transport for SOAP sender. Therefore, this is not an issue unless customized senders are being used.

  The targetUri is flowed internally into the RFH2 header of a request message. Therefore, this too is unlikely to be missing unless a customized sender has been used.
- If a basic integrity check of the expected RFH2 header on a request message fails, the WebSphere MQ transport for SOAP listener generates and returns a report message. Following are the checks conducted by the basic integrity check:
  - There is an RFH2 structure identifier (“RFH”).
  - The RFH2 structure is V2.
  - The length of the RFH2 header is greater than 0 and less than the length of the complete message.
  - The message format is set to MQFMT_NONE.
  - The length of each folder is not greater than the amount of unread data in the message.
  - The USR folder in the RFH2 message contains a contentType of text/xml. charset=utf-8, and the transportVersion is set to 1.

  One of the most common reasons an integrity check fails is if a *foreign* message is placed on the request queue. (A foreign message is a message that is not placed by a WebSphere MQ transport for SOAP sender).
- The backout or retry threshold is exceeded when a WebSphere MQ transport for SOAP listener attempts to process a request.

In all the instances where report messages are returned, it is the responsibility of the client application to detect whether a response message is a report type message, and take the required action. It is also the responsibility of the client application or some dedicated utility to monitor the dead letter queue for report messages that are dead lettered, and to once again take the necessary action, if necessary.

> **Note:** WebSphere MQ transport for SOAP no longer uses a special dead letter queue. In the earlier SupportPac versions, a SOAP-specific dead letter queue was used. Now, the WebSphere MQ transport for SOAP dead letter mechanism checks the default dead letter queue that is defined in the queue manager. If this is not defined, no default dead letter queue is assumed, and the message is not written to any dead letter queue.

5.8.2 Message integrity options

The -a option of the listener provides support for various message integrity options. This allows the listener's behavior to be customized when it is not possible to write a failed request message to the dead letter queue. For details about these concepts, refer to *WebSphere MQ V6 Fundamentals*, SG24-7128.

Following are the message integrity options:

- To always show a warning message and continue executing. This is the low message integrity option as specified with -a LowMsgIntegrity.
- To always show an error message, back out the request message so that it remains in the request queue, and exit the listener. This is the high message integrity option as specified with -a HighMsgIntegrity.
- To act according to the persistency option of the request message. Otherwise, the request and response cycle can continue indefinitely for the individual message. If it is nonpersistent, the listener shows a warning message and continues to run. If it is persistent, it shows an error message. In such a situation, back out the request message and exit the listener. This is the default message integrity option as specified with -a DefaultMsgIntegrity.

In the case of low message integrity, the message is discarded even as the listener continues to run because, otherwise, the request-and-response cycle continues indefinitely. Thus, in this context, the continued operation of the service is deemed more important than being able to preserve an original failed request.

In the case of high message integrity, the message is always backed out so that it remains on the request queue. Here, the contents of the message are deemed valuable and must therefore be preserved, and the listener stopped from making further attempts to invoke the service until the reason for the underlying failure is identified and corrected.

If this option is omitted when invoking the listener, the default message integrity is assumed.

> **Restriction:** In WebSphere MQ V6, the -a option is not propagated into the generated listener scripts through the deployment utility. It is therefore, necessary to manually edit these scripts after deployment if a nondefault message integrity option is to be used. This limitation is expected to be corrected in a future update.

5.9 Microsoft .NET asynchronous interface

WebSphere MQ V6 is supported with the use of Microsoft .NET asynchronous interface. This interface permits a service request to be issued asynchronously in order to enable a client application to be able to perform other useful tasks while it waits for a response.

This interface is designed to work within the context of a single process. Therefore, the same process that made the service request must obtain the response. It is not designed for a separate process to be invoked later to gather responses.

The Microsoft .NET asynchronous interface is ideal for graphical user interface (GUI) clients, where it is necessary to be able to initiate Web Service requests and allow the application to remain responsive while the request is being transported, the service started, and the response returned.

This form of asynchrony is hereafter referred to as Microsoft .NET short-term asynchrony to underline the fact that the asynchrony is considered only within the lifetime of a single process.

Following are the limitations pertaining to Microsoft .NET short-term asynchrony:

- It is only relevant to a single process environment.
- It is not based on standard interfaces.
- It is only relevant to the Microsoft .NET client environment and there is no equivalent for the Axis environment.

WebSphere MQ transport for SOAP provides additional features for asynchrony that are designed to work beyond the lifetime of a single process, so that one process initiates the Web Service requests and another process runs separately and later polls for and obtains responses. This functionality is provided in MA0V SupportPac. It is discussed in detail in Chapter 14, "Long-term asynchronous functionality (MA0V)" on page 303.

This form of asynchrony is referred to as WebSphere MQ transport for SOAP long-term asynchrony.

Note: The MA0V SupportPac is classified as a Category II SupportPac and is therefore *not* formally supported by IBM.

Because there are currently no standards for the asynchronous invocation of Web Services, the long-term asynchrony facilities cannot claim to adhere to any standards. However, these facilities have been designed in such a way that when such standards do emerge, the MA0V functionality can be migrated to conform to the standards.

An example pertaining to the use of Microsoft .NET short-term asynchrony is illustrated in 5.9.1, "Using Microsoft .NET short-term asynchrony" on page 94.

5.9.1 Using Microsoft .NET short-term asynchrony

There are no examples provided with WebSphere MQ V6 that demonstrate the use of Microsoft .NET short-term asynchrony.

The following example client provides a simple demonstration:

- The client initiates a short-term asynchronous request to the StockQuoteDotNet service sample that is provided in the samples.
- The service is given an argument of DELAY, which causes the service to sleep for five seconds before returning a response.
- Immediately after the asynchronous request is made, a synchronous request is also made.
- This time, because a delay is not requested, the synchronous service returns directly.
- The asynchronous response is then returned a few seconds later. This demonstrates the asynchronous nature of the original request, in that, the application remains responsive while waiting for the asynchronous response to be received.

Example 5-14 shows a sample Microsoft .NET short-term asynchronous client using a callback.

Example 5-14 Sample Microsoft .NET short-term asynchronous client using a callback

```
/// This sample program makes an asynchronous request to the getQuote service.
getQuote
/// is called with an argument of "DELAY" to cause a delay before the response is
/// returned. After making the async request, the demo makes a synchronous request to
the
/// same service with an argument of "XXX" so that the service will return directly.
The
/// synchronous response will be returned before the asynchronous response,
/// thereby demonstrating the asynchronous nature of the original request.
///
/// The proxies required by the client for the Dotnet service are generated from the
ASMX
/// file provided as part of the SOAP samples.

using System;
using System.Net;
using System.Threading;

class SQCS2DotNetShortTerm
{
   static void Main(string[] args)
   {
      // Register the WMQSOAP URL extension with DotNet
      IBM.WMQSOAP.Register.Extension();

      StockQuoteDotNet stockobj = new StockQuoteDotNet();

      // Set timeout to 30secs
      stockobj.Timeout=30000;

      // Any first argument is used as the target Url
      if (args.GetLength(0) >= 1) stockobj.Url = args[0];

      Console.WriteLine("\n(Press Enter to close application)");

      // Make asynchronous request with "DELAY" to cause 10 secs delay before
response
      System.Console.WriteLine("\nRequesting async service with delayed response");
      stockobj.BegingetQuote("DELAY", new AsyncCallback(MyCallbackDotNet), stockobj);
```

```
      // Now call the service synchronously - this should complete before the async
response is returned
      System.Console.WriteLine("\nCalling service synchronously");
      System.Single res = stockobj.getQuote("XXX");
      System.Console.WriteLine("Synchronous answer: " + res);

      // Block here, otherwise demo will exit before response has been returned
      Console.WriteLine("\nBlocking to allow async response to be returned.");
      Console.ReadLine();

   }

   // Short term async Callback method
   static void MyCallbackDotNet(IAsyncResult ar)
   {
      // Recover the .NET proxy object from the AsyncState paramater
      StockQuoteDotNet proxyDotNet = (StockQuoteDotNet) ar.AsyncState;

      try
      {
         System.Single ret = proxyDotNet.EndgetQuote(ar);
         Console.WriteLine("Short term async response: " + ret);
      }
      catch (Exception e)
      {
         Console.WriteLine(">>> Exception caught in callback: " + e);
      }
   }
}
```

To demonstrate this client, you must build samples in a specific directory first, as detailed in the product documentation. The client can then be built with the following command:

```
csc "/lib:%WMQSOAP_HOME%\bin" /r:amqsoap.dll SQCS2DotNetShortTerm.cs
generated\client\*.cs
```

To start the client, invoke the SQCS2DotNetShortTerm binary built earlier. When started, the client makes the asynchronous request, calls the service synchronously, and then blocks until the asynchronous response is returned. This is shown in Example 5-15.

Example 5-15 Client's behavior on execution

```
C:\temp\redbook>SQCS2DotNetShortTerm
"jms:/queue?destination=SOAPN.demos@WMQSOAP
.DEMO.QM&connectionFactory=connectQueueManager(WMQSOAP.DEMO.QM)&replyDe
stination=SYSTEM.SOAP.RESPONSE.QUEUE&targetService=StockQuoteDotNet.asm
x&initialContextFactory=com.ibm.mq.jms.Nojndi"

(Press Enter to close application)

Requesting async service with delayed response

Calling service synchronously
Synchronous answer: 88.88

Blocking to allow async response to be returned.
Short term async response: 88.88


C:\temp\redbook>
```

There are different techniques by which to exploit the short-term interface. In this example, the client makes an asynchronous request by calling the service's BegingetQuote() method and supplying in the argument list, a reference to an asynchronous callback method called MyCallbackDotNet(). This method is called by the infrastructure when the response is ready to be returned.

Note: The timeout parameter is set to 30 seconds (30,000 milliseconds). This is to ensure that the infrastructure does not time out the response before the service is completed.

It is also possible to use the WaitHandle technique instead of the callback technique to process a request asynchronously using the short-term interface. This technique may be appropriate where a known piece of post-processing must be performed after making the asynchronous request before waiting for the response. Example 5-16 shows this.

Example 5-16 Sample Microsoft .NET short-term asynchronous client using WaitHandle technique

```
/// This sample program makes an asynchronous request to the getQuote service.
getQuote
/// is called with an argument of "DELAY" to cause a delay before the response is
/// returned. After making the async request, the demo makes a synchronous request to
the
/// same service with an argument of "XXX" so that the service will return directly.
The
/// synchronous response will be returned before the asynchronous response, thereby
demonstrating
/// the asynchronous nature of the original request.
///
/// This demo waits for the async response with the WaitHandle technique
///
/// The proxies required by the
/// client for the Dotnet service are generated from the ASMX file provided as
/// part of the SOAP samples.

using System;
using System.Net;
using System.Threading;

class SQCS2DotNetShortTermWaitHandle
{
   static void Main(string[] args)
   {
      // Register the WMQSOAP URL extension with DotNet
      IBM.WMQSOAP.Register.Extension();

      StockQuoteDotNet stockobj = new StockQuoteDotNet();

      // Set timeout to 30secs
      stockobj.Timeout=30000;

      // Any first argument is used as the target Url
      if (args.GetLength(0) >= 1) stockobj.Url = args[0];

      Console.WriteLine("(Press Enter to close application)");
```

```
      // Make asynchronous request with "DELAY" to cause 10 secs delay before
response
      System.Console.WriteLine("\nRequesting async service with delayed response");

      IAsyncResult ar = stockobj.BegingetQuote("DELAY", null, null);

      // Now call the service synchronously. This should complete before the
      // asynchronous response is returned
      System.Console.WriteLine("\nCalling service synchronously");
      System.Single res = stockobj.getQuote("XXX");
      System.Console.WriteLine("Synchronous Ans: " + res + "\n");

      // Wait for the WaitHandle to become signaled.
      ar.AsyncWaitHandle.WaitOne();

      // Now get async result
      System.Single ret = stockobj.EndgetQuote(ar);

      Console.WriteLine("Short term async response: " + ret);
   }
}
```

Example 5-14, Example 5-15, and Example 5-16 use the short-term interface with Microsoft .NET clients to drive Microsoft .NET Web Services. This interface can also be used in the same manner to drive Axis Web Services from Microsoft .NET clients. However, it is not possible to use this short-term interface from Axis clients to either Microsoft .NET Web Service or Axis Web Service.

The Microsoft .NET documentation Web site provides information about the Microsoft .NET asynchronous interface. Refer to Chapter 10, “.NET Web Service” on page 213 for more details about implementing short-term asynchronous Web Service clients.

5.10 WebSphere Application Server and CICS Transaction Server interoperability

Transporting SOAP messages using a messaging bus has its advantages, as outlined in 4.4.2, “Interoperability” on page 37. However, this must not restrict the choice of SOAP infrastructure that may be used if a services-oriented approach to software design is to live up to its promise of standards-based interoperability. SOAP/Java Message Service (JMS) is a technology that must mitigate this problem, and allow the use of a messaging bus for SOAP to become ubiquitous in the enterprise.

There is currently no specification for SOAP/JMS, but its use asks the question of interoperability. Both WebSphere Application Server and CICS Transaction Server can use messaging as a transport for Web Services or SOAP.

In WebSphere Application Server, this is achieved with SOAP/JMS, and in CICS, it is achieved with SOAP/WebSphere MQ.

The Web Services of all these are interoperable when they use WebSphere MQ as the transport. This allows the interoperation of truly heterogeneous Web Services, using a messaging bus provided by WebSphere MQ. WebSphere Application Server must use WebSphere MQ as the JMS Provider for the SOAP/JMS transport.

Note: Using default messaging for SOAP/JMS in WebSphere Application Server V6 is interoperable only when the WebSphere MQ link is used.

Interoperability is achieved by conforming to the structure of WebSphere MQ JMS messages when WebSphere MQ transport is used in CICS and WebSphere MQ. This allows interoperability between four separate SOAP infrastructure:

- .NET
- Axis
- WebSphere Application Server
- CICS Transaction Server

This list is expected to grow with the introduction of SOAP/JMS as a standard.

> **Note:** WebSphere MQ transport for SOAP does not guarantee interoperability between different Web Service host environments such as Apache Axis, Microsoft .NET, CICS, or WebSphere Application Server. This is because there are different standards for SOAP and many implementations of SOAP environments, and it is these implementations that determine the specifics of each SOAP message. In addition, there are various options for formatting the details of a service within a particular implementation, for example, Remote Procedure Call (RPC), Doc, or Literal. WebSphere MQ transport for SOAP delivers the message content, but cannot ensure that the content is meaningful to the service that receives it.

5.10.1 WebSphere Application Server interoperation

Adherence to the JMS message structure is obviously implicit in WebSphere Application Server using WebSphere MQ as the JMS provider. Interoperation is supported, subject to the following authorized program analysis reports (APARs) being applied to the WebSphere Application Server:

- APAR PK05013 for WebSphere Application Server 5.x
- APAR PK05012 for WebSphere Application Server 6.x

These APARs ensure that the SOAPAction header is included in the transport in the client, if specified.

SOAPAction

Microsoft .NET currently makes use of a SOAP header called SOAPAction. This header must be present in a SOAP message to invoke a Microsoft .NET Web Service.

The SOAP 1.1 specification says the following about SOAPAction: “The SOAPAction HTTP request header field can be used to indicate the intent of the SOAP HTTP request. The value is an URI, identifying the intent. SOAP places no restrictions on the format or specificity of the URI or that it is resolvable. An HTTP client MUST use this header field when issuing a SOAP HTTP Request.

The presence and content of the SOAPAction header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP. The header field value of empty string ("") means that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication of the intent of the message.”

There is a lot of debate about how exactly the SOAPAction header must be used by SOAP infrastructure and what its benefits are. WebSphere Application Server and Axis do not use the SOAPAction header in the infrastructure itself, although it may be added to the SOAP/JMS transport manually or is present if generated from the WSDL that specifies a SOAPAction header.

WebSphere Application Server and WebSphere MQ transport for SOAP have resolved this issue by including the SOAPAction header only if the client proxies have specified so. This is as the SOAP specification requires. Thus, interoperability can be ensured by using the WSDL generated by Microsoft .NET, which includes SOAPAction.

The WebSphere Application Server APARs that are the prerequisites for interoperability in the *WebSphere MQ Transport for SOAP*, SC34-6651 should ensure that the SOAP/JMS transport includes SOAPAction, if present in the proxies.

> **Note:** The client proxies specify SOAPAction if they are generated from WSDL, which also specifies SOAPAction. The xxxxxxStub.java file sets SOAPAction in the transport.

com.ibm.mq.jms.Nojndi

One of the strengths of JMS is that it uses contexts and namespaces to decouple administration from code. A JMS application can look up JMS-administered objects from the namespace at run time and use them to perform messaging. This is done by using an InitialContextFactory. Therefore, finding the queue manager and the queue that hosts a Web Service is achieved by defining the appropriate JMS QueueConnectionFactories and Queues in the namespace. The client proxies look these up at run time. For SOAP/JMS, the QueueConnectionFactories and Queues are specified in the URI in a format similar to WebSphere MQ, as shown in Example 5-17.

Example 5-17 URI format

```
jms:/queue?destination=jms/BankingServiceQueue&connectionFactory=jms/Ba
nkingServiceQCF&targetService=BankingService.asmx
```

Using the URI specified in Example 5-17 causes the SOAP/JMS classes to look up jms/BankingServiceQCF and jms/BankingServiceQueue from the WebSphere Application Server namespace. The InitialContextFactory used is the WebSphere Application Server InitialContextFactory, com.ibm.ws.naming.util.WsnInitCtxFactory (WsnInitCtxFactory). These JMS-administered objects can be configured. Therefore, moving the location of the Web Service is trivial and requires no code change.

WebSphere MQ provides a utility that allows reusage of the WebSphere MQ URI, which is the URI that is specified when deploying the service. This utility is simply another namespace called com.ibm.mq.jms.Nojndi (Nojndi).

NoJndi is an InitialContextFactory that parses the WebSphere MQ SOAP URI and creates equivalent JMS-administered objects, JMS QueueConnectionFactories, and Queues. Consequently, WebSphere Application Server Web Service clients can use the same URI as their equivalent WebSphere MQ Web Service clients, as shown in Example 5-18.

Example 5-18 WebSphere MQ SOAP URI

```
jms:/queue?destination=BANKING.SERVICE.REQUEST.QUEUE@QM_LocalToSvc&conn
ectionFactory=(connectQueueManager(QM_LocalToSvc)binding(client)clientC
hannel(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))&i
nitialContextFactory=com.ibm.mq.jms.Nojndi&targetService=BankingService
.asmx&replyDestination=BANKING.SERVICE.RESPONSE
```

Using the URI specified in Example 5-18 causes the SOAP/JMS classes to use the InitialContextFactory specified in the com.ibm.mq.jms.Nojndi URI to look up the destination and connectionFactory. These options are parsed by Nojndi to create the JMS-administered objects that are the WebSphere MQ JMS equivalent of the values specified in the URI. A QueueConnectionFactory and Queue are passed back to the SOAP/JMS classes, which uses them to make the client invocation. Figure 5-3 illustrates this.

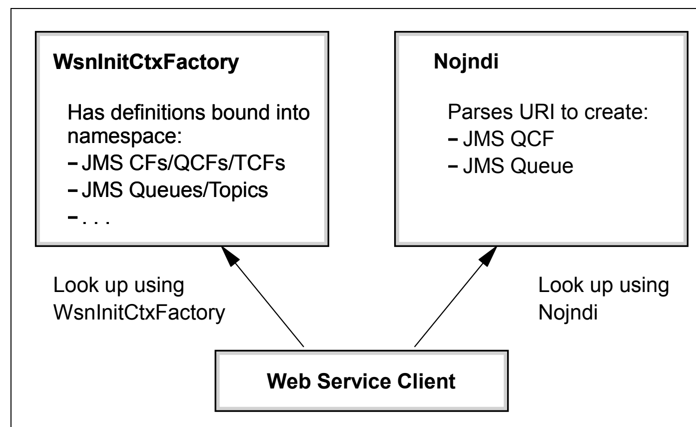


Figure 5-3 Comparing WsnInitCtxFactory and Nojndi

Note: Nojndi supports all the options available on the URI, including Secure Sockets Layer (SSL).

Nojndi is packaged in a jar file called com.ibm.mq.jms.Nojndi.jar. It is installed by WebSphere MQ when the Java Messaging and the SOAP transport feature are selected during install time. It is therefore available to the WebSphere Application Server run time if the Java MQ Client is installed on the same machine and the WebSphere Application Server is configured appropriately. Refer to the WebSphere Application Server documentation for more details. Alternatively, Nojndi may simply be copied to the WAS_HOME/classes directory, where it is picked up according to the class loading process in the WebSphere Application Server.

5.10.2 CICS interoperation

CICS is able to utilize WebSphere MQ as a transport for SOAP. The structure of the messages used adheres to the JMS structure to achieve interoperability, and is subject to APAR PK04615 being applied to CICS.

WebSphere MQ also requires APAR IC46192 to correctly handle the encoding of CICS client requests and CICS Web Service responses. Without this APAR, the WebSphere MQ sender and WebSphere MQ listener get MQRHF2 format errors.

The WebSphere MQ SOAP URI is used in the same manner with CICS as with WebSphere MQ or WebSphere Application Server.

5.11 Summary

This chapter discussed the key topics regarding the implementation of WebSphere MQ transport for SOAP with target Web Services and client applications. The basics for setting up the implementation environment correctly and the IVT mechanism were reviewed.

This chapter demonstrated an overview of a typical development process when implementing WebSphere MQ transport for SOAP services and clients. The SOAP/WebSphere MQ URI syntax is explained because a clear understanding of the URI is essential to understand how various WebSphere MQ options can be set when using the transport.

Various SOAP formatting issues and the deployment utility provided with SOAP/WebSphere MQ were also discussed in detail. Customization of this process in certain situations and the ways in which to do this were discussed. The SOAP/WebSphere MQ listeners and the different ways in which the listeners can be activated were also discussed.

The transactionality facilities that are provided in WebSphere MQ V6 were reviewed, and the boundaries of transactional support between the product and the optional and unsupported MA0V SupportPac illustrated. The error handling mechanisms provided with WebSphere MQ transport for SOAP were reviewed and examples of how Microsoft .NET short-term asynchronous interface can be used in client applications, were provided. Finally, the manner in which WebSphere MQ transport for SOAP interoperates with WebSphere Application Server and CICS were discussed.

6

Security

Securing data flows in any network is of paramount importance to all the systems. This chapter discusses how to achieve secure communication using the Secure Sockets Layer (SSL) protocol and WebSphere MQ transport for SOAP.

Support for SSL on Windows in WebSphere MQ V6 has changed from using Microsoft Certificate Stores to the IBM Global Security Kit (GSKit). This brings uniformity across the Windows and the UNIX platforms in terms of certificate management.

This chapter describes the use of GSKit and the IBM Key Management tool (iKeyman) to configure SSL on WebSphere MQ channels. It also discusses the enabling of a configured SSL environment when using WebSphere MQ transport for SOAP. The concepts behind secure communication are discussed and are made specific in their application to WebSphere MQ V6.

© Copyright IBM Corp. 2006. All rights reserved.

6.1 Concepts of security

Protecting a system is of paramount importance. If a system's security requirements are not addressed, it is vulnerable to abuse in the following forms:

- Unauthorized access

  Accessing a resource as a user unknown to the system

- Eavesdropping

  Reading and understanding the data being transferred when it is being transferred

- Tampering

  Reading, intercepting, and altering the data sent across a network connection

- Impersonation

  Sending data across a network under the guise of another user

Figure 6-1 shows an illustration of eavesdropping and tampering.

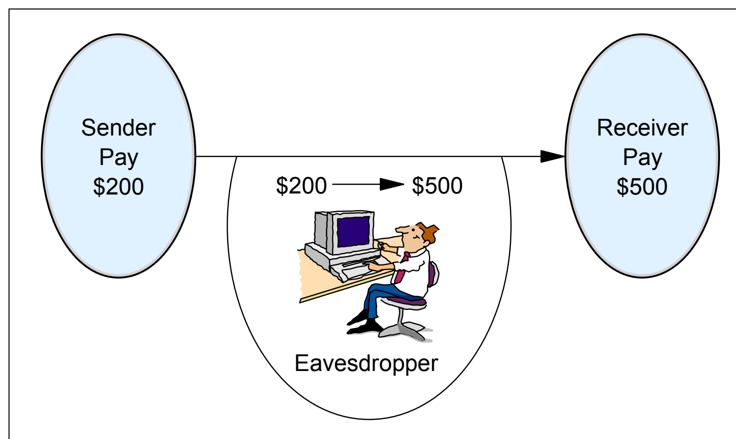


Figure 6-1 Example of eavesdropping and tampering

Before discussing the process involved in securing the data flow between the applications, WebSphere MQ and Web Services, it is important to discuss the concepts of security used to protect data. Protection of data within a system relies on the security services being provided and implemented. The security services and implementation mechanisms are described in this chapter.

6.1.1 Security services

The resources within a system must be protected. This protection is provided by the security services within a system. A system's security can be built based on the following security service types:

- Identification and authentication

  Identification is not just being able to decide who the user or the application accessing the system is. Rather, it is about identifying *uniquely* that user or application. To be able to uniquely identify the user or application, the system must implement authentication, whereby, a basis of trust must be established before access is granted to the resource. It is important for the system to be able to trust the user or application before granting access to its resources.

- Authority

  The implementation of an authority service within a system provides different users and applications with various access permissions with regard to each resource. This prevents unauthorized access of the system's resources. Further, authority can be implemented to prevent unauthorized use of a resource after access is granted, for example, allowing read access, but not write access.

- Confidentiality

  In many systems, it is likely that the data being passed around is confidential. Implementing a confidentiality service provides the system with a mechanism to pass data around without exposing the contents of that data.

- Data integrity

  It is important to know that the data a user or application receives is correct, and that it has not been tampered with during transmission. A data integrity service provides users and applications within the system with confirmation that the information they receive is not different from the original that was sent.

- Nonrepudiation

  A security service implementing nonrepudiation provides the receiver of the data with a mechanism to determine, with confidence, the origin of the information. Nonrepudiation allows the receiver to confirm whether the incoming data is from the sender it purports to be from, and whether the integrity of the data is maintained. This security service ensures that the sender of the data cannot deny sending the information, and is hence tied to that particular transmission. If a problem within the system is traced back to a particular data that is received, the source from where the data originated cannot deny sending it.

It is not necessary to have each of these security services implemented within a system. Each one provides protection for a different vulnerability, and they can exist independently of one another or all can coexist within the same system. Implementing all these services gives the maximum possible protection against all security violations.

6.1.2 Security mechanisms

A security mechanism is an implementation of a security service, for example, confidentiality may be implemented by encryption algorithms, which in turn provide only the sender and the receiver with the authorization to read the data being sent. Security mechanisms take many forms, some of which are listed here:

- Access control list (ACL)
- Firewall
- Cryptography
- Public key infrastructure (PKI)
- Digital certificates
- Digital signatures

This book describes cryptography concepts, including PKI, digital certificates, and digital signatures as mechanisms to secure communication between clients and Web Services using WebSphere MQ.

6.2 Security considerations

Whether you add security to an existing system or design a new system with security features, several considerations should be taken into account. It is important to be aware that as part of the system architecture, security measures must be implemented at the application layer and the data transmission layer. These are two distinct layers of the architecture, and the security features of each of them can operate independently of the other.

This section discusses the security mechanisms in terms of where they fit in a system's architecture.

6.2.1 Application layer security

Security at the application level must be concerned only with the person to whom the data is being sent to or received from, and subsequently, what can be done with that data, once it is received. This implies the use of identification and authentication, and authority.

Identification and authentication can be implemented using a system-specific user ID database. This provides confirmation that a particular user is a valid user on a given system.

An authority service can be implemented in such a way that one user has only read access to a resource, while another user has complete administrative authority over the resource. After the system confirms whether a user really is who they claim to be, it ensures that they can perform only those actions that have been authorized to them by the system administrator.

The implementation of security at the application layer level is system-specific, and is based primarily on the security features provided by the operating system.

Access authority in WebSphere MQ

WebSphere MQ provides an authority service called the object authority manager (OAM). The OAM provides the facility to define the access authority to WebSphere MQ objects on an individual basis. The OAM maintains an access control list (ACL) for each of the resources that it is managing. Access can be granted or denied on an individual user basis or a user group basis. On UNIX platforms, access authority can be granted only on the basis of group membership, for example, all the members of a certain user group within a UNIX system can be given access to place messages in a particular queue, but it is not possible to single out a user from that group and deny access to that person. However, within a Microsoft Windows environment, the OAM can grant or deny access to a particular resource based on both group membership and individual user IDs.

The OAM functionality is provided by three utilities:

- dspmqaut

  This shows the access authority information for a user or a group with regard to a given resource.

- dmpmqaut

  This shows all the access authority information for each resource for a given user ID or group.

- setmqaut

  This sets the access authority information for a given user ID or group with regard to a particular WebSphere MQ resource.

For more details about the syntax of these utilities, refer to *WebSphere MQ Security*, SC34-6588 and *WebSphere MQ System Administration Guide* SC34-6584. Use the OAM to provide application layer security in situations where access to all the resources within the WebSphere MQ environment must be strictly controlled.

6.2.2 Transmission layer security

The implementation of security services at the transmission layer is more complex than at the application layer. At the transmission layer, there are many more potential vulnerabilities to exploit than at the application layer. The system architecture must implement identification and authentication, confidentiality, data integrity, and nonrepudiation at the transmission layer.

It is not only users of the local system who connect at the transmission layer, but also users of systems that exist as external, separate entities. It is therefore, not practical to maintain a database of trusted users. It is still, however, important that a basis of trust is formed between the system being connected to and the system trying to connect. To implement this effectively, use digital certification. For more information about this, refer to 6.3.4, “Digital certificates” on page 117.

When a system sends data across a network to a destination, there is a period when the data is neither in the sending system nor in the destination, that is, it is in transit. During the time the information is in transit, it must be kept secure and confidential. One way of ensuring confidentiality is to encrypt all the data that is sent over a connection. Data can be encrypted using a wide variety of algorithms, and changed from *plain text* to *cipher text*. Anybody who reads the cipher text sees only unclear content and thus, the confidentiality of the data is maintained. For more information about this, refer to 6.3, “Concepts of cryptography” on page 113.

When data is transferred across a network, it is almost always expected to arrive at its destination in the same state it was in when it was sent. To ensure data integrity, use a *message digest*. The use of a message digest ensures that the receiver is confident that the contents of the data is exactly the same as it was when it was sent, that is, it has not been modified. For more information about this, refer to 6.3.2, “Message digest” on page 115.

Digital signature handling must be implemented within a system architecture to ensure that any data that is received is from the same sender it states it is from. This prevents impersonation, where, sources claiming to be somebody else (an

authorized person), sends data. Digital signatures also provide the system with the nonrepudiation facility, to ensure that those who send data cannot deny the origin of the data they have sent. For more information about this, refer to 6.3.3, "Digital signature" on page 116.

6.3 Concepts of cryptography

This section discusses some of the important concepts of cryptography and cryptographic services. Encryption and decryption, which are sometimes referred to as encipherment and decipherment, are essential to keep data confidential. They can also be added to some of the security services discussed earlier for providing greater enhancement to the system's security.

6.3.1 Cryptography

Cryptography is defined as the practice of encryption and decryption. The terms encryption and decryption, on their own, describe a broad area of securing any kind of data. Encryption is the process of converting readable data, (plain text) to unreadable data (cipher text), and decryption is the reverse process. There are a large number of computational algorithms to perform this conversion. In general, the encryption/decryption process follows these steps:

1. The sender converts the plain text to cipher text (encryption).
2. The sender transmits the cipher text to the receiver.
3. The receiver converts the cipher text back to plain text (decryption).

In practice, these three tasks are broken down into more specific steps to ensure that data is *not* readable during transmission, the senders and receivers trust each other, the mechanisms to ensure that data is not tampered with is implemented, and the receivers are able to confirm, on receipt, that the data came from the senders they were expecting it from.

Communication that takes place during the encryption process involves the execution of a computational algorithm, This algorithm converts plain text to cipher text and back again to plain text. This algorithm requires the use of a *key* in order to make the encryption and decryption specific to the communication at that time. It is standard to use a different key each time a new communication occurs. There are two types of encryption/decryption algorithms:

- Symmetric key algorithm
- Asymmetric key algorithm

Symmetric key algorithm

A symmetric key algorithm requires the same key for both encryption and decryption as shown in Figure 6-2.

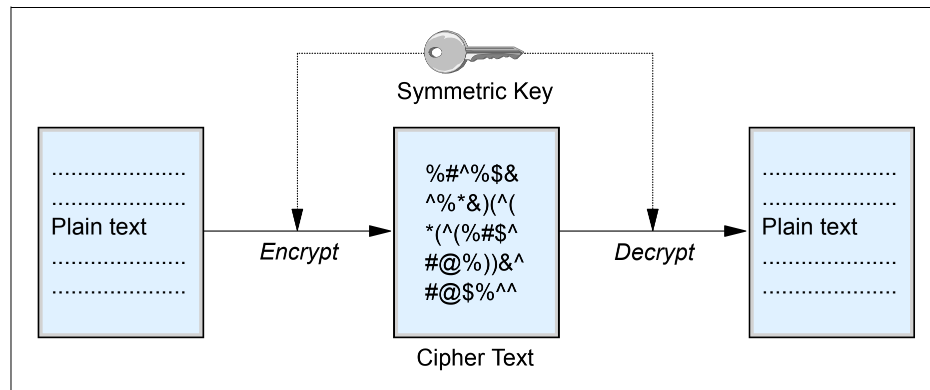


Figure 6-2 Symmetric key encryption

If the key that is used to generate the cipher text is not used to convert it back to plain text, the resulting decryption is equally unreadable. The main limitation of this algorithm is the secure distribution of the key that is used for encryption and decryption. There is always the possibility that the symmetric key is intercepted before *secured* communication begins. If this happens, the person who intercepts the key can read all the data being sent across the network despite it being encrypted.

Asymmetric key algorithm

An asymmetric key algorithm works on the basis that one key is used to encrypt the data and another key used to decrypt the data. This is shown in Figure 6-3.

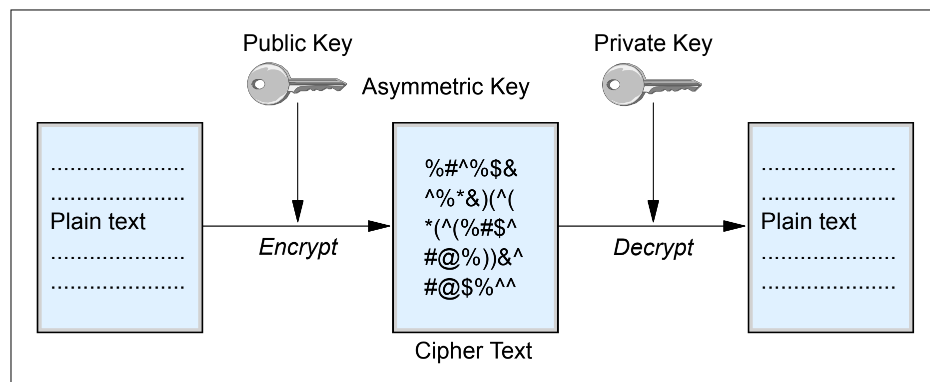


Figure 6-3 Asymmetric key encryption

The two keys used in this algorithm must be different. However, they must be associated with each other. One key must be made publicly available, while the other key must be kept secret (private) and should be known only to the owner of the corresponding public key. Using any other key results in unclear content being returned as the decrypted plain text. It is not possible to decrypt data using the same public key that was used to encrypt it. Asymmetric key cryptography is also known as public key cryptography and forms the basis for a public key infrastructure (PKI).

Public key infrastructure

A PKI is defined based on the system's requirements, and typically provides the following services:

- Public key distribution
- Digital certificate issuance
- Digital certificate validation
- Digital certificate management

Optionally, a PKI can provide many more services. These are provided based on the requirements of the system infrastructure. For more information about PKIs, refer to the Internet X.509 Public Key Infrastructure Charter in the following Web site:

http://www.ietf.org/html.charters/pkix-charter.html

6.3.2 Message digest

A message digest can be applied to the data being sent across a network to address the issues of tampering and data integrity. It is a fixed size numerical representation of the contents of the data. Message digests are also known as message authentication codes (MACs).

Computation of a message digest is performed by a *hash function*, which is used to turn a long string of data into a fixed size, smaller representation of the original data. The sender and the receiver must agree to the hash function. The hash function must adhere to the following rules:

- It must be a one-way hash function and it must not be possible to generate the contents of the message from a message digest.
- It must be computationally nonfeasible to find two different pieces of data that hash to the same message digest.

When the sender computes the message digest, it passes the data to the receiver. On receipt of the data, the receiver must decrypt the message if it is encrypted and run the hash function on the data contents. Because the hash function almost never produces the same message digest from two different

pieces of data when the same message digest is returned, the receiver can be sure that the data contents have not been altered since the time it was sent. If the resulting message digest is not the same, the receiver knows that the data contents have been altered, and that a security violation has occurred.

6.3.3 Digital signature

The use of a digital signature enables a data receiver to be confident that the message was sent by the sender claiming to have sent it. The digital signature is generated by encrypting a representation of the data to be sent, typically the message digest.

Unlike a handwritten signature, a digital signature is entirely dependant on the contents of the data. It is possible and indeed likely, that if one sender adds a digital signature to two separate pieces of data, the resulting signature on each piece is different. When encrypting the message digest, senders use their private key, which can only be decrypted using the senders' public key. Clearly, this process alone leaves the data contents open to view by anyone, since everyone has access to the public key meant for the senders. The following steps are necessary to send and receive secured messages:

1. The sender creates the data contents.
2. The data contents are hashed to generate the message digest.
3. The sender encrypts the message digest using a private key to generate the digital signature.
4. The data contents and digital signature form the message that is sent to the receiver.
5. The entire message is encrypted using the receiver's public key. Thus, the receiver is the only entity who can decrypt the message, since only the receiver knows the corresponding private key.
6. The fully encrypted message is sent across the network to the receiver.
7. The receiver decrypts the message using a private key and decrypts the message, resulting in decrypted data contents and a digital signature.
8. The receiver decrypts the digital signature using the sender's public key to generate a message digest.
9. The receiver also hashes the data contents to generate another instance of the message digest.

10. A comparison of the message digests from step 8 and step 9 reveals whether the data contents have been altered. If the two match, the receiver can be sure about having the correct data. However, if the message digests do not match, one of the following has probably occurred:

   - The data contents have been altered since the time they were sent.
   - The message was sent from someone other than the proclaimed sender.

6.3.4 Digital certificates

Although there are many benefits to digital signatures, as specified in the previous section, there is one problem with regard to these signatures. How does the receiver know that the sender can be trusted? For both sides to communicate, mutual trust should exist. Digital certification provides a method on which to base who to trust and who not to trust.

A digital certificate is constructed from the following entries, with optional extras based on the X.509 standard:

- The owner's public key
- The owner's distinguished name
- The distinguished name of the certificate authority that issued the certificate
- The date from which the certificate is valid
- The expiry date of the certificate
- The version number
- The serial number

Digital certification aims to bind a particular public key with a particular entity in order to avoid impersonation. Digital certificates can been viewed as a proof that is endorsed by a globally trusted authority, that the owner of a given public key is who they claim to be.

Third-party globally trusted authorities, known as certificate authority (CA) or certification authorities, charge a fee for issuing digital certificates.

Certificate authorities

A certificate authority is an independent, trusted authority who provides digital certificates to entities who want to implement security within an information technology system. A system user must construct a certificate request and send it to the CA. On receiving the request, the CA carries out background checks on the entity who is requesting a certificate, and follows up by either issuing a certificate it rejecting the request.

The certificate that a CA issues to a user is called *user certificate*. This certificate contains only the user's public key. The public-private key pair is generated by the tool that is used to create the certificate request. The certificate is also signed by

the CA as a sign of authenticity. A separate certificate, which is the CA certificate, is also supplied. This is used as proof that the CA who signed the user certificate is a valid and trusted authority. This certificate is called *signer certificate*.

When digital certificates are used as an authentication method, an extra verification task must be performed. Before secure communication begins, both sides must exchange digital certificates. Each side verifies whether the other side's certificate is one they can trust, for example, if both the certificates are signed by the same CA and neither of them have expired, the communication is allowed to continue. Alternatively, if they are signed by different CAs, but each side considers the other side's CA as a trusted source, communication is allowed to continue. However, if the certificates are signed by different CAs and one side does not trust the other's CA, communication is stopped and no data is transferred.

One other facility a CA provides is publishing a certificate revocation list (CRL). This is a list of certificates that are known to the CA as being revoked and no longer valid for authentication purposes. Certificates are most likely to be revoked on the basis of misuse. CRLs can be stored and queried on a system in many ways. For more information about CRLs, refer to the Internet X.509 Public Key Infrastructure Charter on the Web at:

http://www.ietf.org/html.charters/pkix-charter.html

Distinguished names

A certificate always has a distinguished name (DN) which is used to uniquely identify the entity it belongs to. The following attributes typically form part of the DN:

- CN: Common name
- T: Title
- O: Organizational name
- OU: Organizational unit name
- L: Locality
- ST: State
- C: Country or region

There are other attributes that are defined as part of the X.509 standard for digital certificates. The DN is represented as a string, for example, CN=Joe Bloggs, O=IBM, OU=ITSO, L=San Jose, ST=CA, C=US.

Note: The CN attribute is used to describe an individual user or any other entity, for example, a WebSphere MQ queue manager.

Certificate chains

A CA certificate can be signed by another CA in the same way that user certificates are signed by CAs. Extending this further can create a certificate chain consisting of many certificates. It is the responsibility of the application to verify each certificate in the chain and ensure that it is from a trusted authority. When one certificate is shown to be signed by an untrusted source, the entire chain, from the user certificate to the failed certificate, becomes invalid and communication cannot be allowed to continue. The two certificates at each end of the chain are the *user certificate* and the *root CA certificate*. The root CA certificate is signed by itself, and is implicitly trusted by the users who have certificates signed by it.

Obtaining a digital certificate

To obtain a certificate that can be used in a secure system, a user must create and send a certificate request. Figure 6-4 shows the way in which this process works.

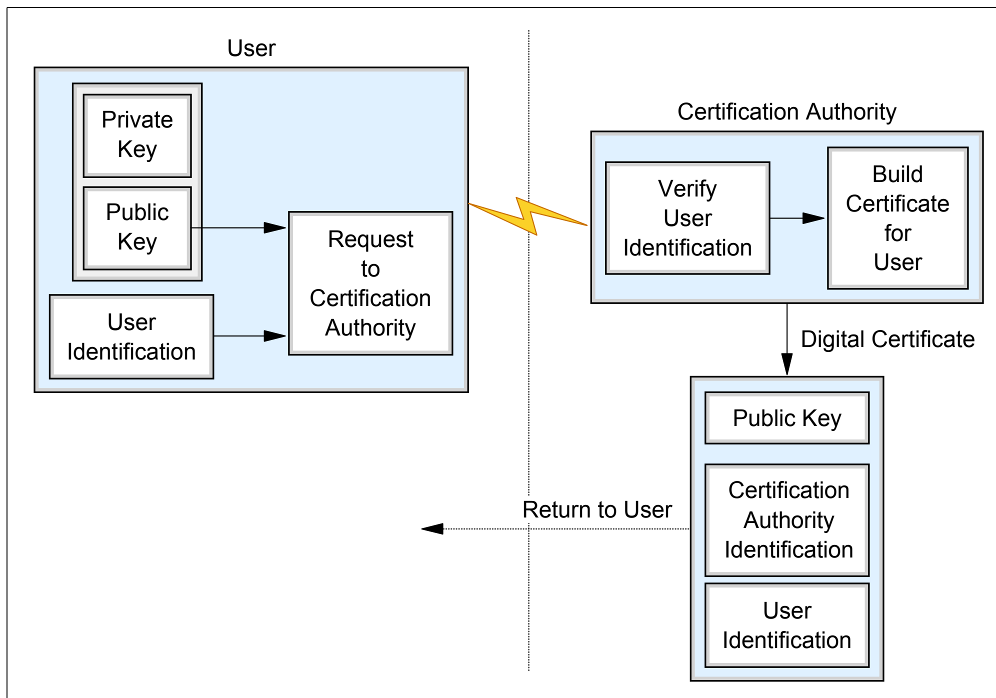


Figure 6-4 Obtaining a digital certificate

In Figure 6-4, the user identification and certification authority identification represent the DN of each of the users and the CA. The public and private keys are generated before the request is sent. The public key is sent as part of the request, while the private key is kept safely by the user. Digital certificates can be obtained from a variety of sources for a fee. There are, however, tools available that allow a user to create CA certificates and subsequently sign user certificates with it. OpenSSL, for example, provides all the mechanisms to generate certificates, both CA and user, which can be used as queue manager and WebSphere MQ client certificates. For more information about the use of OpenSSL, refer to the following Web site:

http://www.openssl.org

6.4 Introduction to Secure Sockets Layer

This section discusses some of the concepts underlying the Secure Sockets Layer (SSL), which is a common secure protocol that is used in communication over the Internet.

6.4.1 Concepts of Secure Sockets Layer

SSL is an industry standard protocol, widely used in the Internet and intranets as a method of securing data that flows over an insecure network. Secure Sockets Layer provides methods for implementing many of the security services mentioned earlier for a reliable transport protocol such as TCP/IP, for example, authentication, data integrity, and data encryption.

The current version of specification for SSL is SSL 3.0. To access the complete specifications, visit the following Web site:

http://wp.netscape.com/eng/ssl3/

SSL connections employ both symmetric and asymmetric cryptographic techniques. An SSL connection is initiated by a calling application and is then known as the SSL client. The responding application becomes the SSL server until the communication ends. SSL connections are sometimes referred to as SSL sessions. Every SSL session is started with a handshake, as defined by the protocol.

The SSL handshake is the process by which the SSL client and the SSL server begin to build a trust. During the handshake, each end of the session agrees to a version of the protocol, a session ID, and a cipherSuite. The authentication process then takes place, with the SSL server sending its certificate to the client, and the client sending its certificate to the server, if necessary. When each side has authenticated the other side, application data can begin to flow securely between the two sides.

6.4.2 CipherSuites and cipherSpecs

A cipherSuite represents a suite of cryptographic algorithms that can be used by an SSL connection. A cipherSpec represents a specific set of algorithms that is used by a given SSL connection. The cipherSpec for any particular connection is built from one of each of the following types of algorithm:

- The key exchange and authentication algorithm that is used during the SSL handshake
- The encryption algorithm that is used to encrypt the application data
- The hash or message authentication code algorithm that is used to generate the message digest from the data content

For more information about cipherSuites and cipherSpecs, refer to *WebSphere MQ Security,* SC34-6588, *WebSphere MQ Using Java,* SC34-6591. Alternatively, refer to the SSL 3.0 specification, which is available in the following Web site:

`http://wp.netscape.com/eng/ssl3/`

6.5 Secure Sockets Layer support in WebSphere MQ

WebSphere MQ provides support for SSL 3.0 on both message channels and Message Queue Interface (MQI) channels, enabling link-level security. It is important to map the idea of an SSL client and SSL server to the different types of channels provided in WebSphere MQ. For any channel pair, the initiator of communication is considered the SSL client, while the other end of the channel becomes the SSL server. All channel types within WebSphere MQ have the attribute SSLCIPH. It is this attribute which, when set, enables SSL on the channels. This is also the attribute where the cipherSpec is set. To access the entire set of cipherSpecs that are available, refer to *WebSphere MQ Security,* SC34-6588.

The message channel agent at the server end of an MQI channel and at each end of a message channel, acts as a security service provider for the queue manager to which it is connected. During the SSL handshake, it is the MCA that sends the server's digital certificate to the opposite end of the channel for authentication.

At the client end of an MQI channel, the WebSphere MQ client code handles the task of sending digital certificates, besides acting as the security service provider. The WebSphere MQ client code acts on behalf of the user of the client application in this case.

Fundamental to WebSphere MQ's support of SSL is the SSL key repository. This is known as a key database. The repository file has the file extension .kdb. All the certificates, both user and CA, are stored in this database. The database is used by the queue manager to decide on who to allow secured SSL connections to. The queue manager attribute SSLKeyRepository holds the location of the database on the file system without the kdb file extension specified.

WebSphere MQ clients also use their own SSLKeyRepository as a certificate store to decide which server queue managers they trust for secure communication, and to store its own certificate, should the server require authentication information from the client. By default, server connections have the requirement to authenticate the client set. However, this can be changed from REQUIRED to OPTIONAL, in the SSLCAUTH server connection channel attribute. To specify the location of the SSLKeyRepository for WebSphere MQ client, use one of the following methods:

- Specify the location of the database on the client system using the MQSSLKEYR environment variable without the kdb file extension.
- Set the KeyRepository field in the SSL configuration options structure MQSCO within an MQCONNX call.

For details about the function of the SSL key repository, refer to "The Secure Sockets Layer key repository" on page 124.

When using a Java client, SSL is configured in a slightly different manner. Details of this are discussed in 6.6, "Working with WebSphere MQ and Secure Sockets Layer" on page 123.

6.6 Working with WebSphere MQ and Secure Sockets Layer

This section discusses the use of SSL within a WebSphere MQ environment, including details about how to set up secure message channels and secure MQI channels. This section discusses the following steps in detail:

1. Creating the key repository.
2. Generating certificates for the WebSphere MQ queue manager and a client.
3. Adding digital certificates to the key repository, ready for use.
4. Specifying the SSL attributes in the Universal Resource Indicator (URI) when using WebSphere MQ transport for SOAP.

6.6.1 Configuring WebSphere MQ for secured communication

When using a message channel, it is important for the WebSphere MQ queue managers at each end to have a user certificate. The channel is an MQI channel, and only if the SSLCAUTH attribute on the server connection channel is set to REQUIRED, does the client on that channel require a user certificate.

> **Note:** If the client has a digital certificate when setting SSLCAUTH to OPTIONAL, the certificate is sent to the server and authenticated. If this authentication fails, the channel does not start. However, if the client does not have a certificate and SSLCAUTH is set to OPTIONAL, no attempt is made to authenticate, and the channel starts.

The tool that is used to demonstrate the key repository and certificate management is the IBM Global Security Kit (GSKit) because it is provided with the installation of WebSphere MQ. For some part of this demonstration, it is assumed that the WebSphere MQ Explorer is installed.

The Secure Sockets Layer key repository

The SSL key repository is where all the digital certificates that are to be used within WebSphere MQ, are stored. To create an SSL key repository, perform the following tasks:

1. Open the WebSphere MQ Explorer by selecting **Start** → **Programs** → **IBM WebSphere MQ** → **WebSphere MQ Explorer**.
2. Start the GSKit key manager by right-clicking **IBM WebSphere MQ** and selecting **Manage SSL Certificates...** as shown in Figure 6-5.

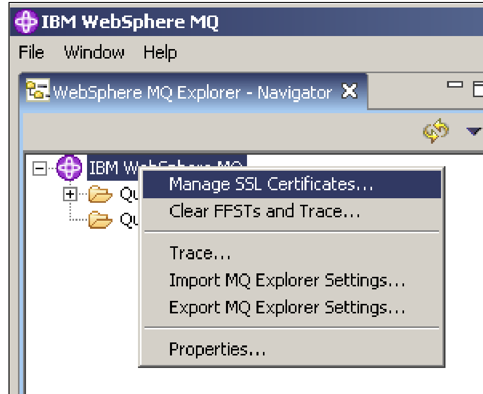


Figure 6-5 Opening the GSKit Key Manager

3. In the IBM Key Management tool, right-click **Key Database File** → **New...**.
4. Enter the File Name and Location to create the key repository and click **OK**, as shown in Figure 6-6.

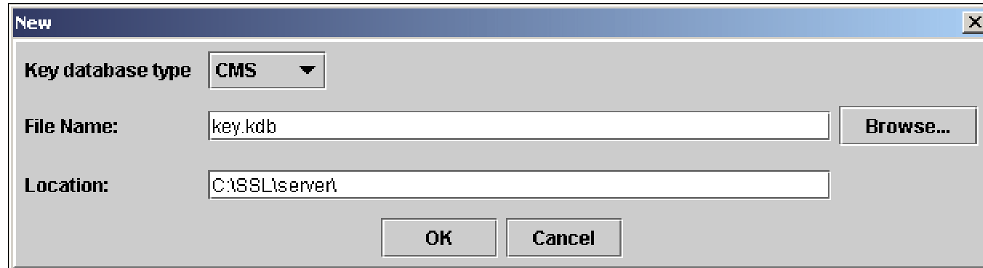


Figure 6-6 Creating a new key repository

Note: If you are using a Java client, select **JKS** rather than the default CMS for the SSL key repository that is used by the client**.** This is because SSL is being handled by the Java Secure Socket Extension (JSSE). The file extension automatically changes to .jks.

5. Create a password to access the key repository, as shown in Figure 6-7. Select **Stash the password to a file?** and click **OK**.

Figure 6-7 Creating a password for the key repository

Important: It is vital that you select the **Stash the password to a file?** box. WebSphere MQ channels of any type do not start if the password is not stashed to a file. However, this is not required if Java client is being used.

The password is stashed to a file in the same directory as the key repository, with the same filename, but with an .sth extension.

The key repository is created. By default, when it is created for the first time, it contains a selection of trusted root CA certificates as shown in Figure 6-8.

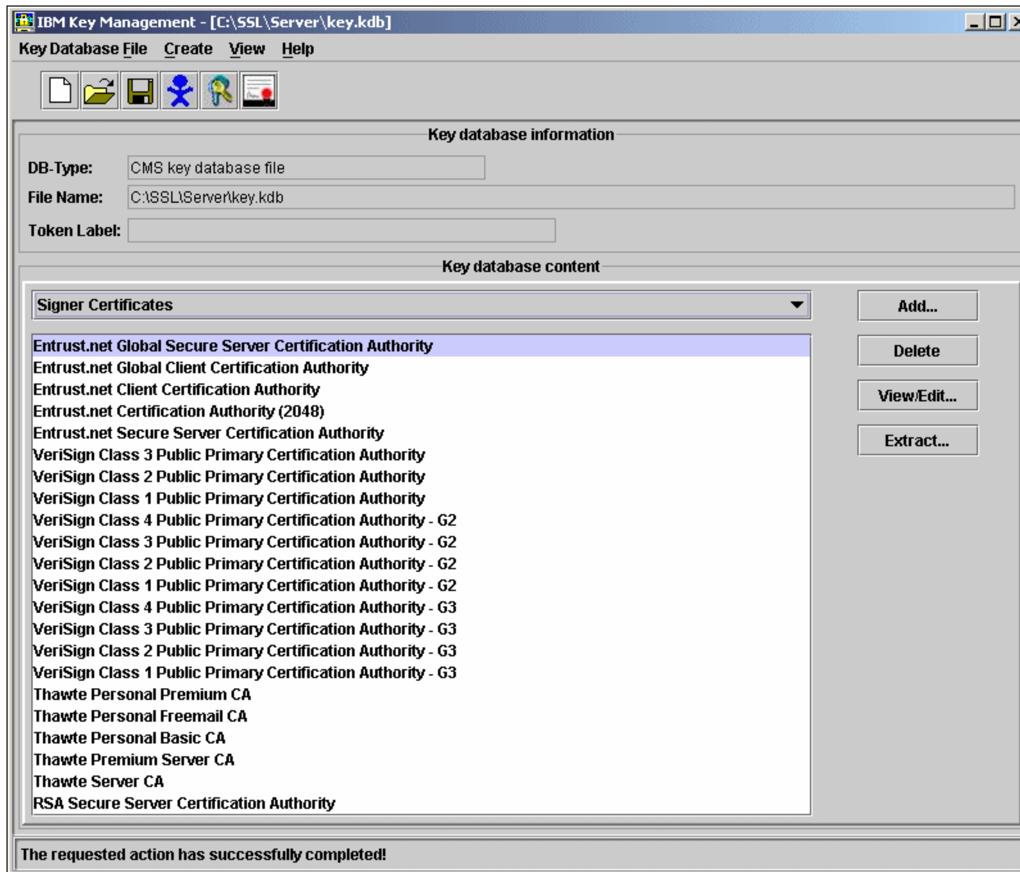


Figure 6-8 The default CA root certificates in a new key repository

The graphical view of the key repository provides the facility to look at the CA or signer certificates, the user or personal certificates, and any requests for personal certificates that have been generated.

6. To access any of these, click the **Signer Certificates** drop-down box. Each of the different views provides different action buttons along the right side of the window.

   For more information about this, refer to *IBM Tivoli Access Manager Secure Sockets Layer Introduction and iKeyman Users Guide, V5.1,* SC32-1363.

7. In this example, the remaining tasks under certificate management are performed using the command line version of the IBM Key Management (iKeyman) Graphical User Interface (GUI). Before using the command-line tool, issue the commands shown in Example 6-1 to ensure that the environment is correctly configured. These are the default install directories for the command line version of iKeyman and the Java installation used by iKeyman.

Example 6-1 Ensuring that the environment is correctly configured

```
set PATH=%PATH%;C:\Program Files\IBM\gsk7\bin\
set JAVA_HOME=C:\Program Files\IBM\WebSphere MQ\gskit\jre\
```

On UNIX platforms, set the JAVA_HOME variable as specified in Table 6-1.

Table 6-1 Environment setup for UNIX platforms

Platform	Command
AIX	export JAVA_HOME=/usr/mqm/ssl/jre
Hewlett-Packard UNIX (HP-UX)	export JAVA_HOME=/opt/mqm/ssl
Linux	export JAVA_HOME=/opt/mqm/ssl/jre
Solaris™	export JAVA_HOME=/opt/mqm/ssl

The Java Secure Sockets Layer key repository

When using the Java client, certain extra options must be specified for authentication to succeed during communication. The JSSE handles the SSL functionality and recommends that there must be a *trust store* and a *key store*. Each of these are represented as a standard SSL key store, as described in the earlier section. However, it is their contents that define whether they are a trust store or a key store.

- A trust store contains only CA certificates and can be used by every user on a given system as a reference to the trusted authorities.
- A key store contains at least a personal or user certificate. It may also optionally contain a list of CA certificates. It must be specific to one user on the system.

It is possible for a key repository to behave as a trust store and a key store at the same time. The contents of a key store can be accessed only by using the correct password. The CA certificate contents of the key repository cannot be viewed by anyone without a password. Thus, even if a key store is also used as a trust store, the confidentiality of the personal certificate is preserved. When the key repository is used in this manner, it can be treated the same way as the standard key repository, as described earlier.

To use a separate key repository for each of the trust stores and key stores, the process of creating a key store with a .jks extension must be followed twice, once for each type. Care must be taken when generating and importing a CA certificate, and subsequently signing and adding user certificates to the correct repositories. In the commands provided in the subsequent sections, note the differences in the option and parameters when using the Java client.

Creating the root Certificate Authority certificate using GSKit

In the examples that follow, each of the user certificates have been signed by a CA certificate. This section provides details about how to generate the CA or signer certificate. iKeyman provides the facility to create a user certificate that is self-signed. This means that the certificate is a root CA certificate, which can also be used as a user certificate. In the tasks described here, the self-signed certificate is created as a normal user certificate, and then moved around the key repository so that it becomes a true CA or signer certificate, which is then used to sign more user certificates. The following instructions assume that a temporary key repository, which holds the generated CA certificate, is created, and is used to sign the certificate requests.

To create the root CA certificate using GSKit, perform the following tasks:

1. Issue the command shown in Example 6-2 to create a self-signed certificate and add it to the temporary key repository.

   Example 6-2 Creating a self-signed certificate

   ```
   gsk7cmd -cert -create -db "C:\SSL\temp\key.kdb" -pw password -label
   "Root CA Certificate" -dn "CN=Root CA, O=IBM, OU=ITSO, C=US" -expire
   1000
   ```

   > **Note:** When using the Java client, the db value has the .jks extension instead of the .kdb extension.

   The values for db and pw must be the location of the key repository and the password to access it, respectively. Label, dn, and expire can be set to anything suitable according to the system's requirements. The value of expire is the number of days until the certificate becomes invalid.

2. Extract the certificate from its current key repository so that it can be used as a root CA certificate or signer certificate in the examples provided later in this section. Issue the command shown in Example 6-3 to extract the certificate. This command creates a file called CAExtracted.arm in the same directory as

the key repository. If the American Standard Code for Information Interchange (ASCII) format is being used, the file extension must be .arm. If the binary format is being used, the file extension must be .der. These are file extensions prescribed by GSKit and iKeyman.

Example 6-3 Extracting the certificate

```
gsk7cmd -cert -extract -db "C:\SSL\temp\key.kdb" -pw password -label
"Root CA Certificate" -target "C:\SSL\temp\CAExtracted.arm" -format
ascii
```

> **Note:** When using the Java client, the db value has the .jks extension instead of the .kdb extension.

The key repository used to create the CA certificate must be retained to perform the task of signing certificate requests.

Securing channels between queue managers

To secure the message channels that exist between two servers, WebSphere MQ queue managers using SSL, perform the following tasks:

1. Create an SSL key repository for each queue manager, ensuring that the password for each repository is stashed. For more information about the steps, refer to "The Secure Sockets Layer key repository" on page 124.
2. Each queue manager's key repository requires the root CA certificate in its signer certificate store. Issue the command shown in Example 6-4 for each queue manager's key repository.

Example 6-4 Adding the root certificate

```
gsk7cmd -cert -add -db "C:\SSL\server\key.kdb" -pw password -label
"Root CA Certificate" -file "C:\SSL\temp\CAExtracted.arm" -format ascii
-trust enable
```

   In this command, the value of the file must be the file location that is extracted from the temporary key repository, CAExtracted.arm.

3. Each queue manager requires a user certificate. Therefore, a certificate request must be constructed. Issue the command shown in Example 6-5 for each queue manager.

Example 6-5 Creating an user certificate

```
gsk7cmd -certreq -create -db "C:\SSL\server\key.kdb" -pw password
-label "ibmwebspheremqqm1" -dn "CN=QM1, O=IBM, OU=ITSO, C=US" -file
"C:\SSL\server\ibmwebspheremqqm1_request.arm"
```

> **Important:** Prefix the label option in this command with `ibmwebspheremq`, followed immediately by the name of the queue manager in lowercase. In Example 6-5, the queue manager's name is QM1. Therefore, the label for the certificate is ibmwebspheremqqm1.

4. At this point, the certificate request generated in the previous step (step 3) is sent to an external CA. However, because this example has its own CA, it can accept these certificate requests and process them locally. Issue the command shown in Example 6-6 for each certificate request.

Example 6-6 Sending a certificate request

```
gsk7cmd -cert -sign -file "C:\SSL\server\ibmwebspheremqqm1_request.arm"
-db "C:\SSL\temp\key.kdb" -pw password -label "Root CA Certificate"
-target "C:\SSL\server\ibmwebspheremqqm1_sigend.arm" -expire 364
```

The signed certificate is stored in the file ibmwebspheremqqm1_sigend.arm.

> **Note:** In the command shown in Example 6-6, the value of db is the key repository in which the original CA certificate was created. The original self-signed CA certificate holds both the public key and the private key that are used to sign the certificate requests.

5. Add the user certificates to each queue manager's key repository. For a queue manager called QM1, for example, issue the command shown in Example 6-7.

Example 6-7 Adding user certificate to queue manager's key repository

```
gsk7cmd -cert -receive -db "C:\SSL\server\key.kdb" -pw password -file
"C:\SSL\server\ibmwebspheremqqm1_signed.arm"
```

Issue a similar command for the other server queue manager in order to get its user certificate into its own key repository.

6. At this point, ensure that the queue manager attribute SSLKEYR is set correctly for each queue manager. Example 6-8 shows how to set the SSKLEYR value to the right location. This step is necessary, unless the default location for the key repositories is used and the default name of the repository file has not been changed from key.kdb. When altering the value of SSLKEYR, the .kdb extension must not be used and the secured SSL channels not started.

Example 6-8 Setting the SSLKEYR value

```
>runmqsc QM1
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.

alter qmgr SSLKEYR(‘C:\SSL\server\key’)
   1 : alter qmgr SSLKEYR('C:\SSL\server\key')
AMQ8005: WebSphere MQ queue manager changed.
```

To enable SSL on the channels, it is necessary to set the SSLCIPH value on the channels to a valid cipherSpec. Specify the same cipherSpec at both ends of the channel. Example 6-9 shows one such scenario.

Example 6-9 Setting the SSLCIPH value

```
>runmqsc QM1
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.

alter channel(QM1_2_QM2_SDRC) chltype(SDR) SSLCIPH(RC2_MD5_EXPORT)
    4 : alter channel(QM1_2_QM2_SDRC) chltype(SDR)
SSLCIPH(RC2_MD5_EXPORT)
AMQ8016: WebSphere MQ channel changed.
```

For further information about the cipherSpec available, refer to *WebSphere MQ Security*, SC34-6588.

Changes to the SSL settings on channels are picked up only when the channels are restarted. At this point, restart the channels. The SSL becomes enabled and active.

The commands provided in this section are suitable for both Windows and UNIX platforms. These examples are for Windows systems. The only alteration required for UNIX platforms is changing the way the directory structure is passed to the **gsk7cmd** command.

Securing client-server connections

Enabling SSL on an MQI channel is slightly different from enabling SSL on a message channel. However, many of the concepts apply to both. Most of the commands provided in "Securing channels between queue managers" on page 129, are valid for this section too. This section highlights the differences between the two types of channels and describes how to enable SSL between a WebSphere MQ client and a server queue manager.

WebSphere MQ clients make use of both the SSL key repositories (in the case of a Java client, this is a Java key store) and a user certificate that is signed by a CA. Therefore, the setup stages for an MQI channel are almost the same as those described in "Securing channels between queue managers" on page 129. The first difference occurs in step 3 described earlier. The label for a client certificate must conform to the syntax ibmwebspheremqlogonid.

In the previous section, the suffix is the queue manager name in lowercase for a message channel, for a client certificate, the suffix is the login ID of the client system user. Thus, the command to create a certificate request for a client is as shown in Example 6-10.

Example 6-10 Creating a certificate for a client

```
gsk7cmd -certreq -create -db “C:\SSL\client\key.kdb” -pw password
-label “ibmwebspheremqmmyuserid” -dn “CN=Client 1, O=IBM, OU=ITSO,
C=US” -file “C:\SSL\client\ibmwebspheremqmyuserid_request.arm”
```

Note: If a Java client is being used, the file extension in the db value must be .jks. This relates directly to the type of key store created for the client. Moreover, the value of the db must be the Java key store that holds the user certificate.

The certificate request is then signed in the typical manner, with the commands shown in Example 6-11 for all client types.

Example 6-11 Signing the certificate request

```
gsk7cmd -cert -sign -file
“C:\SSL\client\ibmwebspheremqmyuserid_request.arm” -db
“C:\SSL\temp\key.kdb” -pw password -label “Root CA Certificate” -target
“C:\SSL\client\ibmwebspheremqmyuserid_sigend.arm” -expire 364
```

Note: If a Java client is being used, the file extension in the db value must be .jks.

To add the signed certificate as a user certificate in the client's key repository, issue the command shown in Example 6-12.

Example 6-12 Adding the signed certificate as a user certificate

```
gsk7cmd -cert -receive -db "C:\SSL\client\key.kdb" -pw password -file
"C:\SSL\client\ibmwebspheremqmyuserid_signed.arm"
```

Note: If a Java client is being used, the file extension in the db value must be .jks.

If the system requires a separate Java trust store, the CA that has signed the user certificate must be added to that trust store. To do this, issue the command shown in Example 6-13. This step is, however, not required if the key store is also being used as the trust store.

Example 6-13 Adding the CA to trust store

```
gsk7cmd -cert -add -db "C:\SSL\Java\client\trust.jks" -pw password
-label "Root CA Certificate" -file "C:\SSL\temp\CAExtracted.arm"
-format ascii -trust enable
```

Two methods exist to inform the client about the location of its SSL key repository. The simplest way is to define an environment variable, MQSSLKEYR. Table 6-2 shows how to set this up on Windows and UNIX platforms.

Table 6-2 Setting the MQSSLKEYR environment variable

Platform	Command (example)
Windows	set MQSSLKEYR=C:\client\key
UNIX	export MQSSLKEYR=/client/key

The .kdb extension is not required here too. Adding it to the end of the file name means that the client is unable to find the key repository.

Alternatively, the location of the SSL key repository can be specified as part of the SSL connect options structure MQSCO that is used when issuing an MQCONNX call from with an application. It then becomes the application's responsibility to update the KeyRepository field within the MQSCO to the location of the SSL key repository, again without the .kdb extension. For more information about MQSCO, refer to *WebSphere MQ Application Programming Reference,* SC34-6596.

Specifying the location of the Java trust store and the Java key store to the Java client is a little more complex. Perform this task using the -d option in the **java** command. Set the following parameters:

- javax.net.ssl.keyStore
- javax.net.ssl.keyStorePassword
- javax.net.ssl.trustStore

The command shown in Example 6-14 starts the client application and sets the specified parameters.

Example 6-14 Executing the client application

```
java -Djavax.net.ssl.keyStore=C:\SSL\Java\client\key.jks
-Djavax.net.ssl.keyStorePassword=password
-Djavax.net.ssl.trustStore=C:\SSL\Java\client\trust.jks clientApp
```

> **Note:** clientApp is the compiled Java code representing the client application.

You can specify cipherSpec in three ways. The client connection channel has a SSLCIPH attribute that is used to provide the type of algorithm used to authenticate and encrypt the data flowing across the channel. It is possible to enable each of the three methods. Therefore, each of them are described here in the order of precedence.

- Within an MQCONNX call

  Within the channel definition structure MQCD, there is a field called SSLCipherSpec that can be used to specify which cipherSpec should be used for communication.

- Using a client channel definition table

  If the client system is unable to access the client channel definition table as a shared file on the server machine, the definition table must be copied to the client machine. As an entry in the definition table, the SSLCIPH is defined and altered on the server machine before being copied to the client machine. For more information about using client channel definition tables and the setup required, refer to *WebSphere MQ Clients,* GC34-6590.

- Using Active Directory on Windows

  On Windows systems, it is possible to publish the client channel definition tale using the **setmqscp** command to Active Directory. For more information about using the Active Directory, refer to *WebSphere MQ System Administration Guide,* SC34-6584.

When using a Java client, the equivalent value of SSLCIPH is SSLCIPHERSUITE and can be configured using the JMSAdmin tool. For more information about specifying SSL on a Java client, refer to *WebSphere MQ Using Java,* SC34-6591.

The location of the key repositories should be specified only when a WebSphere MQ client or a Java client is communicating with a server queue manager in the typical manner. Within the area of Web Services, the SSL configuration options are specified using the URI.

Secure Sockets Layer in the Universal Resource Indicator

Secure Sockets Layer configuration for WebSphere MQ transport for SOAP is provided in the form of several options in the WebSphere MQ URI. The options that are available depend on the environment being used.

In a Microsoft .NET environment, the following options are specific to the environment:

- sslKeyRepository

  This is the location of the SSL key repository, specified as a full path name to the .kdb file. Specify this without the .kdb extension.

- sslCipherSpec

  This can be any value. It is specified in the SSLCIPH value on any channel. For a full list of the possible values, refer to *WebSphere MQ Security,* SC34-6588. This is a mandatory field if the sslKeyRepository is set.

In a Java environment, the following options are specific to the environment:

- sslKeyStore

  This is the location of the JSSE key store that is specified as a full path name to the .jks file. Specify this without the .jks extension.

- sslKeyStorePassword

  This is the password that grants access to the JSSE key store that is specified in the sslKeyStore option.

- sslTrustStore

  This is the location of the JSSE trust store that is specified as a full path name to the .jks file. Specify this without the .jks extension.

- sslTrustStorePassword

  This is the password that grants access to the JSSE trust store that is specified in the sslTrustStore option.

- sslCipherSuite

  This is the cipherSuite as specified in *WebSphere MQ Using Java,* SC34-6591*.* In a Java environment, there is a direct mapping from the value of a cipherSuite to cipherSpec, as used in message channels.

If any of these are specified in an environment that they are not specific to, they are ignored. For more information about the SSL options available on the WebSphere MQ URI, refer to *WebSphere MQ Transport for SOAP*, SC34-6651*.*

If necessary, another important, but optional option can be set on the URI, SSLPeerName. This represents a portion of the distinguished name that must be present in any certificate received from the remote participant in the communication. This is the certificate the WebSphere MQ queue manager receives from the client. The value of SSLPeerName can contain an asterisk (*) representing a wild card, for example, an SSLPeerName of CN=IBM* matches CN=IBM Corporation. For more information, refer to "Distinguished names" on page 118.

Chapters 8 - 13 discuss the implementation of Web Services and the clients to invoke them within a .NET, Axis, and WebSphere Application Server environments. Each client uses WebSphere MQ transport for SOAP to invoke the Web Service and therefore, SSL support within WebSphere MQ is integral to securing the communication from the client through WebSphere MQ on to the invocation of the Web Service and the response that flows back.

There are three distinct areas where securing communication using SSL is important within the scope of this book:

- The connections between the server queue managers within the WebSphere MQ network topology
- How the invoking client connects to a queue manager
- How the invoked Web Service connects to a queue manager

Details about how to secure communication between the server queue managers is well-defined in *WebSphere MQ Security*, SC34-6588.

Securing the client/server connections between the invoking client and a queue manager, and between the invoked Web Service and a queue manager are the focus of the subsequent chapters. SSL enablement comes entirely from the option settings supplied in the URI. The underlying implementation of the security services that SSL supplies is achieved by putting in place the key

repositories and populating each of them with the required certificates. The steps for implementing these security services were detailed in this chapter. Each of the subsequent chapters detail the enablement of these services provided by WebSphere MQ and SOAP.

Part 3

Implementing synchronous Web Services

Part 1, “Overview” on page 1 outlined the challenges faced by software engineers when attempting to exploit Web Services. Part 2, “Web Services and security considerations” on page 27 described how WebSphere MQ transport for SOAP added another string to the bow, and detailed the components and architecture of this technology. Part 3 demonstrates the practical implementation of Web Services with WebSphere MQ, using a basic scenario. This part also describes interoperation with WebSphere Application Server.

© Copyright IBM Corp. 2006. All rights reserved.

7

Environment setup

This chapter provides information about the software that is required for the implementation demonstrations provided in the subsequent chapters. It also discusses the installation of the software that is required. In situations where deviation from the default installation process is necessary, details are provided.

This chapter also discusses the setup used to demonstrate the capabilities of WebSphere MQ transport for SOAP, with details about the physical and logical layout of the software and the systems used.

© Copyright IBM Corp. 2006. All rights reserved.

7.1 Software prerequisites

The following software are required to implement the scenarios discussed in the subsequent chapters:

- Microsoft Windows 2000 Professional
- IBM AIX 5.2 ML4
- IBM WebSphere MQ V6
- Microsoft .NET Framework Redistributable V1.1
- Microsoft .NET Software Development Kit (SDK) V1.1
- IBM Rational Application Developer V6
- Visual Studio .NET (optional)
- IBM WebSphere Application Server V6 for AIX

This is an exhaustive list of the software used to demostrate the implementation of the clients and Web Services in the subsequent chapters. It may not be necessary to install each piece of software, depending on the system requirements discussed in each chapter.

7.2 Software installation

This section discusses the important points to be aware of while installing the prerequisite software.

7.2.1 Installing IBM WebSphere MQ V6

To install IBM WebSphere MQ V6, follow the instructions provided in *WebSphere MQ for Windows V6.0: Quick Beginnings,* GC34-6476, and *WebSphere MQ for AIX V6.0: Quick Beginnings,* GC34-6478.

Ensure that the instructions provided in the quick beginning guides are followed when the tasks described here are integrated into the installation process.

Windows installation

To install WebSphere MQ V6 on Windows, perform the following tasks:

1. After accepting the licence, in the WebSphere MQ V6.0 Setup window (Figure 7-1), select the setup type as **Custom**, and click **Next**.

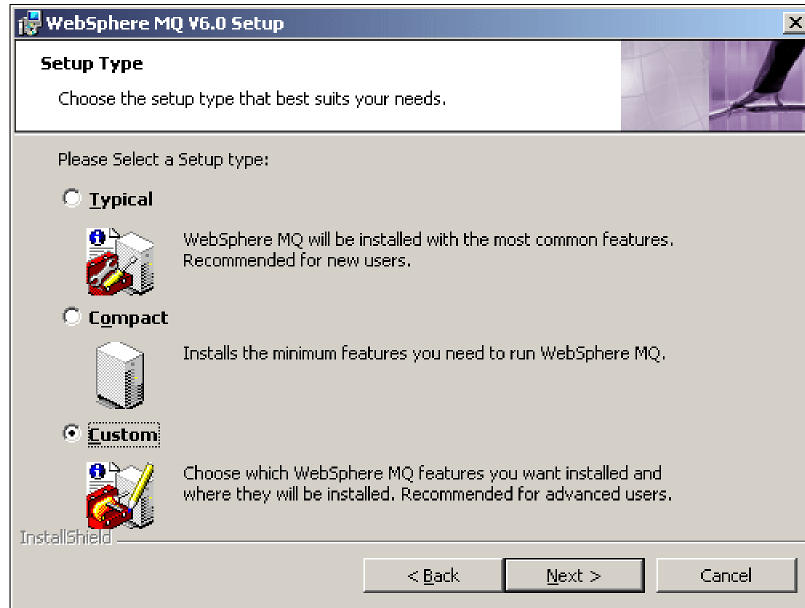


Figure 7-1 Selecting custom installation

The subsequent windows are for configuring the default locations within the file system for WebSphere MQ.

2. In the window that opens (Figure 7-2), ensure that **Windows Client** and **Java Messaging and SOAP Transport** are selected for installation, by clicking the drop-down boxes to the left of each installable option. After selecting all the relevant options, click **Next**.

> **Tip:** Altering the default install location to C:\WMQ\, for example, can make navigating directories within a command-line environment and setting environment variables much simpler and less prone to errors. The installation used for this book is the default location.

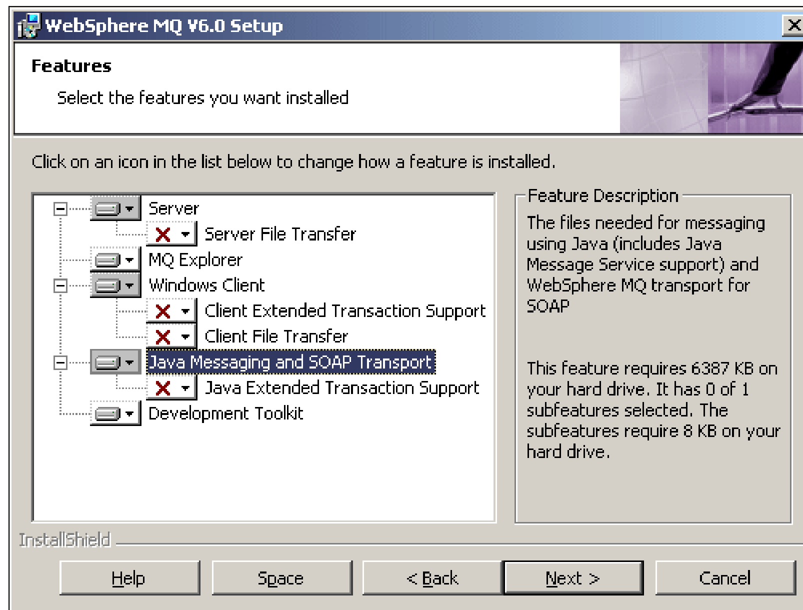


Figure 7-2 Ensuring that correct options are selected on installation

3. In the window that opens, click **Install** to begin the installation.

AIX installation

When installing WebSphere MQ V6 on AIX, ensure that the following packages are included:

- mqm.client for WebSphere MQ client for AIX
- mqm.java for WebSphere MQ Java client, Java Message Service (JMS), and SOAP support
- mqm.keyman for WebSphere MQ support for IBM Global Secure Toolkit (GSKit)

If the system requires the use of an Axis client or Web Service, copy the Apache Axis V1.1 runtime Java archive (jar) file to the system. The location of the axis.jar file on both Windows and AIX is detailed in Table 7-1. Because the Axis runtime is not copied as part of the install process, copy it manually from the installation media. All the locations specified in Table 7-1 are from the top directory of the install media.

Table 7-1 Apache Axis V1.1 runtime location

Platform	Copy from location	Copy to location
Windows	...\PreReqs\Axis\axis.jar	C:\Program Files\IBM\WebSphere MQ\Java\lib\soap\axis.jar
AIX	.../PreReqs/axis/axis.jar	/usr/mqm/java/lib/soap/axis.jar

7.2.2 Installing Microsoft .NET Framework Redistributable V1.1

Download and install the Microsoft .NET Framework Redistributable V1.1 with the help of the instructions provided in the following Web site:

http://www.microsoft.com/downloads/details.aspx?familyid=262D25E3-F589-4842-8157-034D1E7CF3A3&displaylang=en

7.2.3 Installing Microsoft .NET Software Development Kit V1.1

Download and install the Microsoft .NET SDK V1.1 with the help of the instructions provided in the following Web site:

http://www.microsoft.com/downloads/details.aspx?FamilyID=9B3A2CA6-3647-4070-9F41-A333C6B9181D&displaylang=en

Note: Microsoft .NET Framework is a prerequisite for Microsoft .NET SDK, and must be installed as instructed in the download instructions for the SDK package.

For development purposes, it may be necessary to install Microsoft Visual Studio .NET 2003. Installing Visual Studio also installs the .NET Framework and the .NET SDK. In such a situation, the actions discussed earlier are not necessary.

On Windows 2000, in order to deploy and run .NET services, Microsoft Internet Information Services (IIS) must already be installed. If the .NET Framework installation occurs before the installation of IIS, the latter must be registered to the .NET Framework. To do this, use the aspnet_regiis utility provided with the .NET Framework. Refer to *WebSphere MQ Transport for SOAP,* SC34-6651 for more information about configuring the .NET Framework and IIS.

7.2.4 Verifying the installation of WebSphere MQ transport for SOAP

Along with the installation of SOAP transport, a basic test program is provided to ensure that all the necessary components are installed correctly. For more information about this program's function, refer to 5.1.3, "Using the Installation Verification Test to verify installation" on page 52. This program is supplied as an installation verification test (IVT), and is called runIVT. Following are a few simple steps to ensure successful installation:

1. Create and set an environment variable `WMQSOAP_HOME` to be the WebSphere MQ root installation directory, for example, set WMQSOAP_HOME=C:\Program Files\IBM\WebSphere MQ\
2. Run the amqwsetcp program located in the %WMQ_HOME%\bin directory. This program sets up the CLASSPATH and PATH environment variable as required by WebSphere MQ transport for SOAP.
3. Run the regenDemo script. This sets up the test samples that are ready to be run.
4. Issue the runIVT command. This runs each test, and on completion, verifies if the installation is successful.

For more details about testing the installation of the WebSphere MQ transport for SOAP components, follow the instructions provided in *WebSphere MQ Transport for SOAP*, SC34-6651.

7.2.5 Installing WebSphere Application Server V6 for AIX

The installation of WebSphere Application Server V6 for AIX used in this book, is the standard WebSphere Application Server installation. After installing V6.0.0, PTF2 was applied to bring the installation level up to V6.0.2. For more details about the installation procedure, refer to the following Web site:

`http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html`

Note: We recommend that you apply the latest fix pack.

7.2.6 Installing Rational Application Developer V6

The installation of Rational Application Developer V6 used in this book is the standard Rational Application Developer install. Refer to the on-screen instructions during installation, along with the documentation provided in the install media.

The installation of Rational Application Developer used by the team that wrote this book is level 6.0.0.1.3, and is used only for Web Service development.

7.3 Environment setup

This section discusses the system setup used to demonstrate the contents of this book. It details the software installed on each system and the interoperability that is possible between each of the clients and the Web Services.

This section acts as a precursor to the subsequent chapters by introducing the setup of a client invoking a Web Service by using WebSphere MQ as a transport mechanism for SOAP. Figure 7-3 shows a simple setup, whereby a client invokes a Web Service using WebSphere MQ.

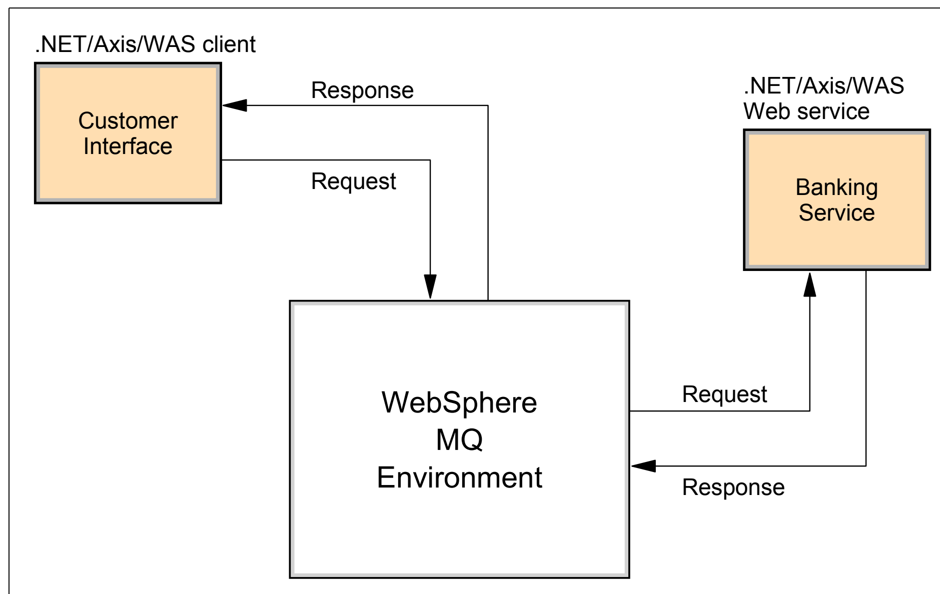


Figure 7-3 Client invoking Web Service using WebSphere MQ transport for SOAP

The client can be written within a Microsoft .NET, Axis, or WebSphere Application Server environment, and can invoke a Web Service that has been written in any of these development environments. The client and the Web Service do not have to be written within the same type of environment for them to interoperate.

The WebSphere MQ environment shown in Figure 7-3 can include a multitude of queue managers, queues, and connections. There is no reason for the WebSphere MQ environment to just be, for example, a single queue manager, with a request queue and a response queue. It is beyond the scope of this book to define all the potential WebSphere MQ environments that can be configured as a transport mechanism for SOAP messages. The following chapters discuss specific WebSphere MQ environments and provide details about how the clients and Web Services interact with them.

Figure 7-4 shows the entire setup used for this book.

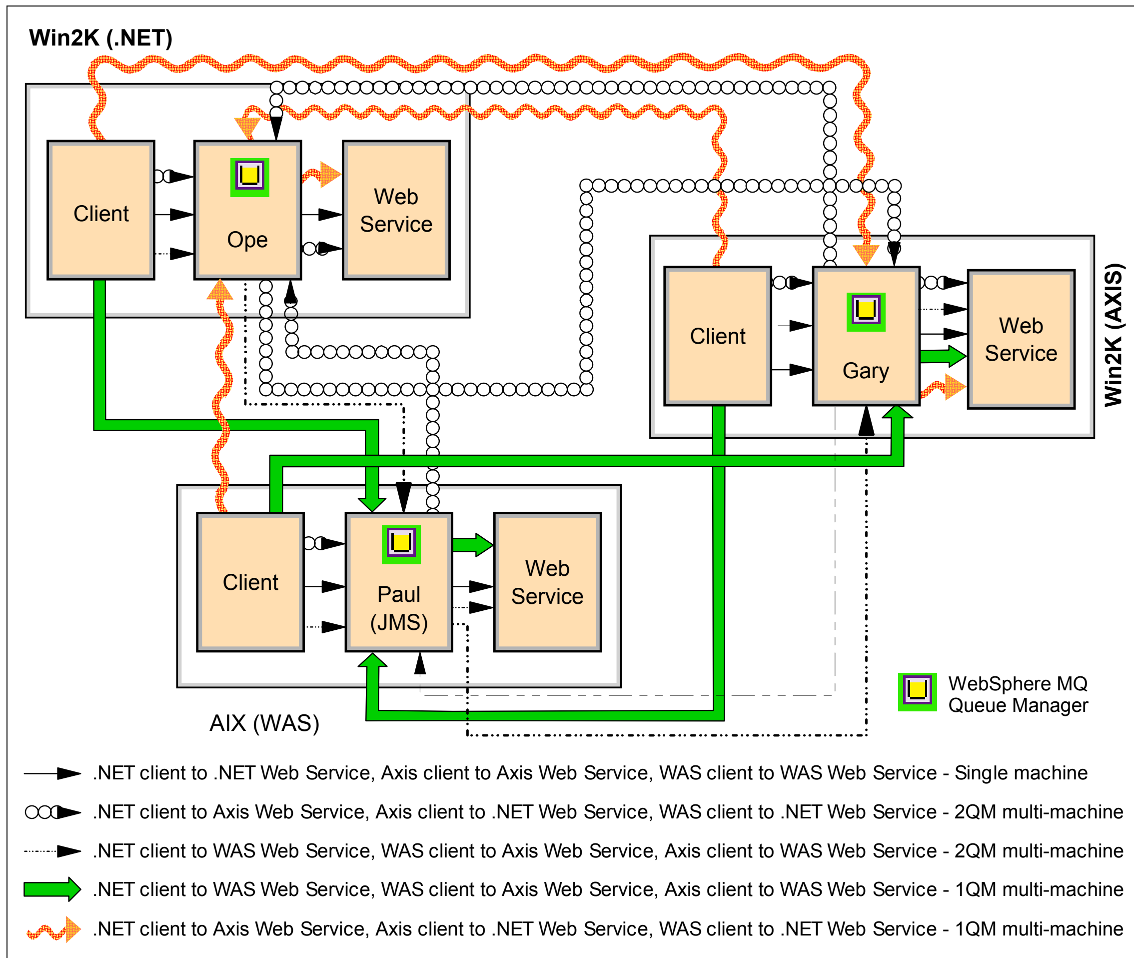


Figure 7-4 Full environment setup

The environment setup used in this example, and shown in Figure 7-4, allows for the following invocation methods:

- A .NET client invoking a .NET Web Service within a single machine, using a single queue manager
- An Axis client invoking an Axis Web Service within a single machine, using a single queue manager
- A WebSphere Application Server client invoking a WebSphere Application Server Web Service within a single machine, using a single queue manager

- A .NET client invoking an Axis Web Service across two machines, using two queue managers
- An Axis client invoking a .NET Web Service across two machines, using two queue managers
- A WebSphere Application Server client invoking a .NET Web Service across two machines, using two queue managers
- A .NET client invoking a WebSphere Application Server Web Service across two machines, using two queue managers
- A WebSphere Application Server client invoking an Axis Web Service across two machines, using two queue managers
- An Axis client invoking a WebSphere Application Server Web Service across two machines, using two queue managers
- A .NET client invoking a WebSphere Application Server Web Service across two machines, using a single queue manager
- A WebSphere Application Server client invoking an Axis Web Service across two machines, using a single queue manager.
- An Axis client invoking a WebSphere Application Server Web Service across two machines, using a single queue manager
- A .NET client invoking an Axis Web Service across two machines, using a single queue manager
- An Axis client invoking a .NET Web Service across two machines, using a single queue manager
- A WebSphere Application Server client invoking a .NET Web Service across two machines, using a single queue manager

However, not all the invocation combinations are discussed in the subsequent chapters. This is to avoid repetition of content.

The exact WebSphere MQ configuration details for each of the scenarios is discussed in the subsequent chapters.

Table 7-2 provides information about the software installed on each machine in the environment.

Table 7-2 Software installation

Software	OPE	GARY	PAUL
Microsoft Windows 2000 Professional	x	x	
IBM AIX 5.2 ML4			x

Software	OPE	GARY	PAUL
IBM WebSphere MQ V6	x	x	x
Microsoft .NET Framework Redistributable V1.1	x		
Internet Information Services (IIS)	x		
Apache Axis Runtime V1.1	x	x	
IBM WebSphere Application Server for AIX V6.0			x

However, Table 7-2 does not provide details about the software installed on each machine for development only purposes. The software used for development includes the following:

- Microsoft Visual Studio .NET 2003
- Microsoft .NET SDK V1.1

Install either of these on any Microsoft Windows 2000 Professional machine that is used to develop the .NET client and the Web Service, and use the resultants on any other Windows 2000 Professional machine configured according to the requirements of the environment.

For purposes of development with regard to this book, IBM Rational Application Developer V6 software is installed for the Axis and WebSphere Application Server clients and Web Services.

Code development for the WebSphere Application Server client and Web Service is performed on Windows 2000 Professional. However, each is deployed on an AIX 5.2 ML4 machine configured according to the requirements of the environment.

7.3.1 Basic WebSphere MQ administration

This section provides details about how to create the basic WebSphere MQ objects that are required in the scenarios that follow. It shows how to create a queue manager and a local queue on that queue manager.

Creating a WebSphere MQ queue manager

To create a WebSphere MQ queue manager, perform the following tasks:

1. Open the WebSphere MQ Explorer by selecting **Start** → **Programs** → **IBM WebSphere MQ** as shown in Figure 7-5.

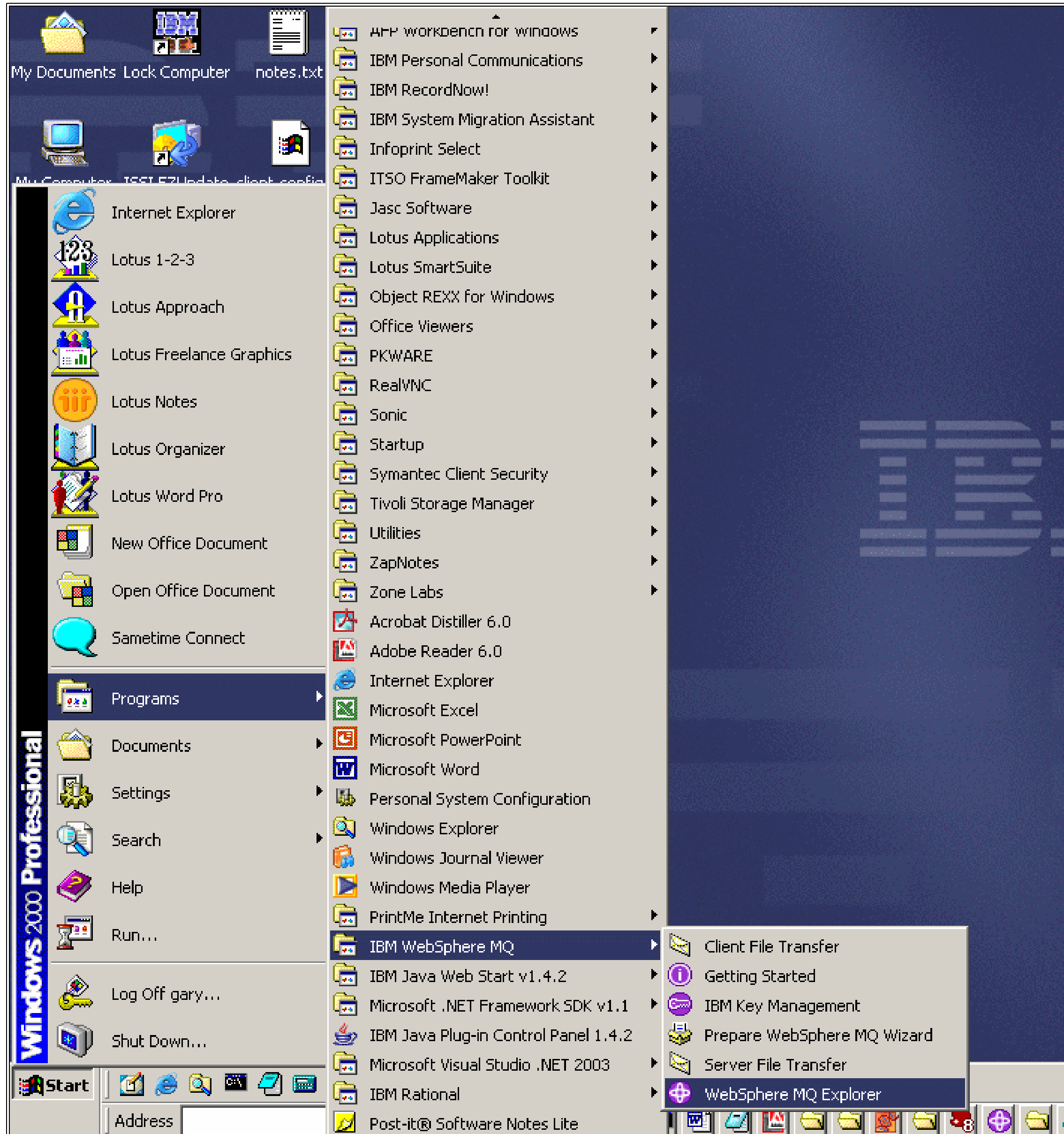


Figure 7-5 Opening the WebSphere MQ Explorer

2. In WebSphere MQ Explorer, right-click the **Queue Managers** folder and select **New** → **Queue Manager** as shown in Figure 7-6.

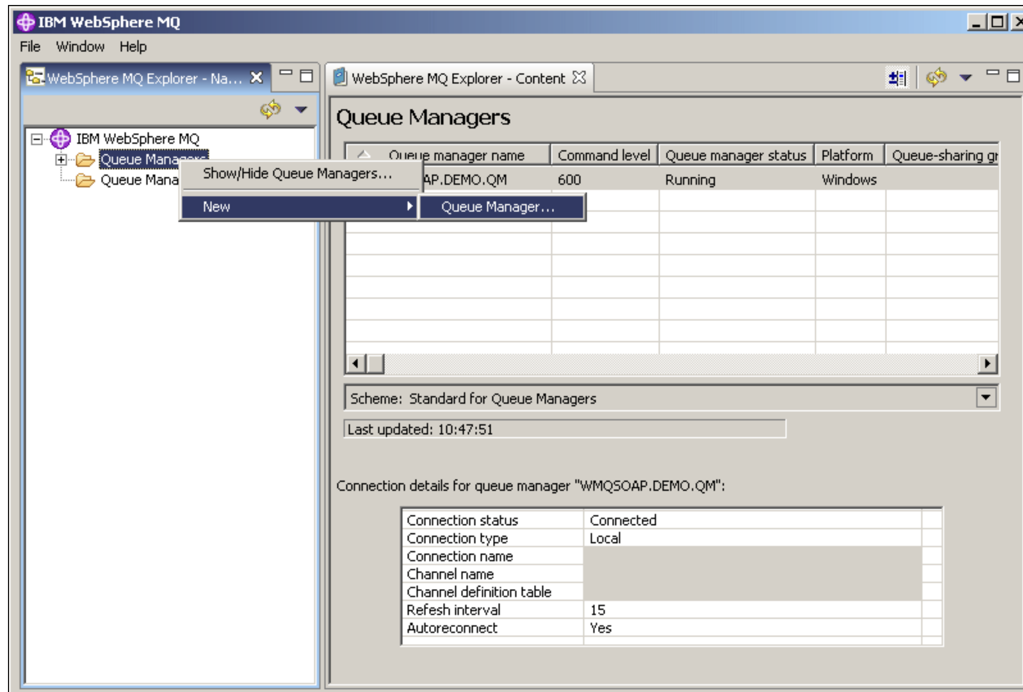


Figure 7-6 Creating a new queue manager

3. In the Create Queue Manager window that opens (Figure 7-7), specify the queue manager name. Click **Next**.

> **Note:** To create a default queue manager, select **Make this the default queue manager**, as shown in Figure 7-7.

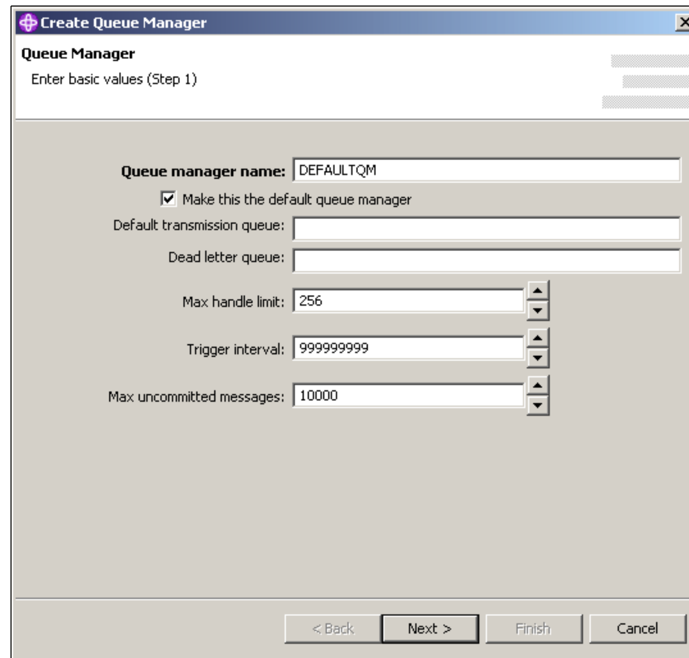


Figure 7-7 Setting up a default queue manager

4. Click **Next** in the subsequent windows, accepting the defaults, if any.
5. Click **Finish** in the final window. This creates and starts the new WebSphere MQ queue manager.

Creating a queue manager using command-line tools

To create a queue manager using command-line tools, perform the following tasks:

1. To create a queue manager in a Windows command prompt, enter the following command:

   ```
   crtmqm <QUEUE MANAGER NAME>
   ```

2. Start the queue manager by entering the following command:

   ```
   srtmqm <QUEUE MANAGER NAME>
   ```

Creating a default queue manager using command-line tools

To create a default queue manager using command-line tools, perform the following tasks:

1. To create a default queue manager in a Windows command prompt, enter the following command:

   ```
   crtmqm -q <QUEUE MANAGER NAME>
   ```

2. Start the queue manager by entering the following command:

   ```
   srtmqm <QUEUE MANAGER NAME>
   ```

Creating a local queue

To create a local queue, perform the following tasks:

1. Open the WebSphere MQ Explorer by selecting **Start** → **Programs** → **IBM WebSphere MQ**.
2. Within the WebSphere MQ Explorer, expand the **Queue Managers** folder. Select the Queue Manager in which you want to create a queue and expand its subfolders. Right-click the **Queues** folder and select **New** → **Local Queue** as shown in Figure 7-8.

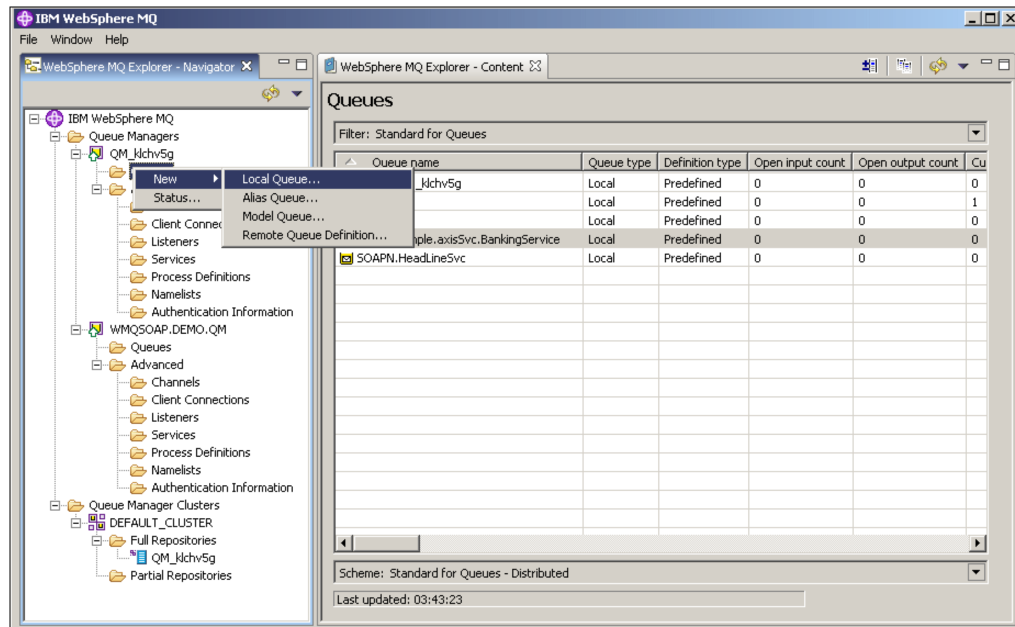


Figure 7-8 Creating a new local queue

3. In the New Local Queue dialog box, enter the name of the queue you want to create, as shown in Figure 7-9. Click **Finish**.

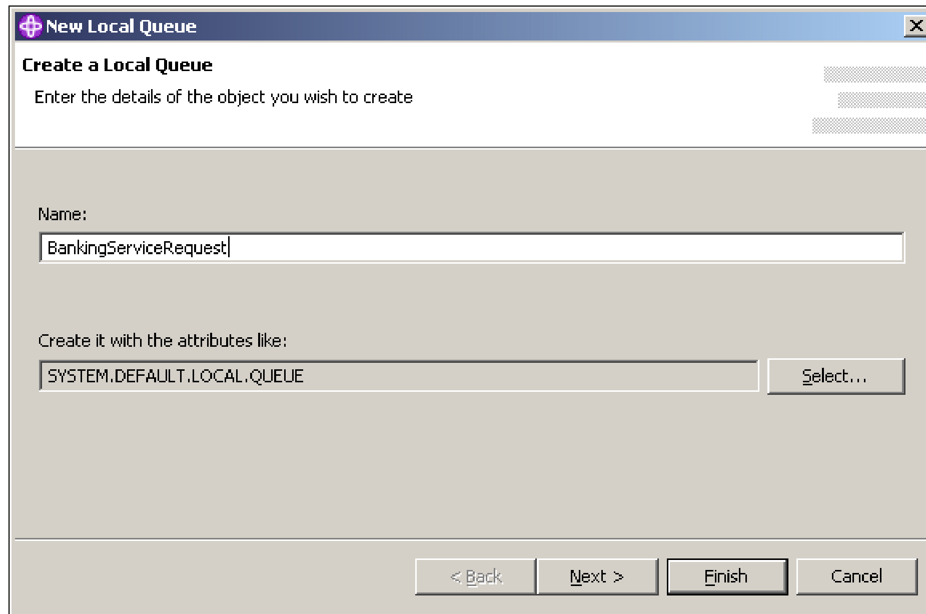


Figure 7-9 Naming a new local queue

Example channel configuration

In the subsequent chapters, a description of a WebSphere MQ environment involving two queue managers is provided. This section provides an example configuration for that environment. Example 7-1 shows the **runmqsc** command that is necessary to configure the following:

- A receiver channel on the queue manager
- A sender channel on the queue manager to send messages to a remote queue manager
- A transmission queue to send messages to a remote queue manager
- A local queue, where response messages are expected to arrive

Example 7-1 Sender-receiver channel configuration

```
>runmqsc QM_SVC_HOST
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager REDQM.

DEF CHL(CLNT_HOST.SVC_HOST) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
   1 : DEF CHL(CLNT_HOST.SVC_HOST) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
```

```
AMQ8014: WebSphere MQ channel created.

DEF CHL(SVC_HOST.CLNT_HOST) CHLTYPE(SDR) TRPTYPE(TCP)
CONNAME('9.1.39.128(1420)') XMITQ(QM_SVC_HOST) REPLACE
   2 : DEF CHL(SVC_HOST.CLNT_HOST) CHLTYPE(SDR) TRPTYPE(TCP)
CONNAME('9.1.39.128(1420)') XMITQ(QM_SVC_HOST) REPLACE
AMQ8014: WebSphere MQ channel created.

DEF QL(QM_CLNT_HOST) DEFPSIST(YES) MAXDEPTH(5000000) USAGE(XMITQ)
TRIGGER TRIGTYPE(FIRST) TRIGDATA(SVC_HOST.CLNT_HOST)
INITQ('SYSTEM.CHANNEL.INITQ') REPLACE DESCR('transmission queue to
SVC_HOST')
   4 : DEF QL(QM_CLNT_HOST) DEFPSIST(YES) MAXDEPTH(5000000)
USAGE(XMITQ) TRIGGER TRIGTYPE(FIRST) TRIGDATA(SVC_HOST.CLNT_HOST)
INITQ('SYSTEM.CHANNEL.INITQ') REPLACE DESCR('transmission queue to
SVC_HOST')
AMQ8006: WebSphere MQ queue created.

DEF QL(BANKING.SERVICE.REQUEST) DEFPSIST(YES) REPLACE DESCR ('Queue on
which service requests arrive')
   5 : DEF QL(BANKING.SERVICE.REQUEST) DEFPSIST(YES) REPLACE DESCR
('Queue on which service requests arrive')
AMQ8006: WebSphere MQ queue created.
```

Run a similar set of commands on the remote queue manager, configuring similar WebSphere MQ resources, for the request-and-response model used in Web Services to become effective. The commands for this are shown in Example 7-2.

Example 7-2 Completing two queue manager setup for a request-and-response model

```
runmqsc QM_CLNT_HOST
5724-H72 (C) Copyright IBM Corp. 1994, 2004.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager REDQM.

DEF CHL(SVC_HOST.CLNT_HOST) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
   1 : DEF CHL(SVC_HOST.CLNT_HOST) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
AMQ8014: WebSphere MQ channel created.

DEF CHL(CLNT_HOST.SVC_HOST) CHLTYPE(SDR) TRPTYPE(TCP)
CONNAME('9.1.39.127(1420)') XMITQ(QM_CLNT_HOST) REPLACE
   2 : DEF CHL(CLNT_HOST.SVC_HOST) CHLTYPE(SDR) TRPTYPE(TCP)
CONNAME('9.1.39.127(1420)') XMITQ(QM_CLNT_HOST) REPLACE
AMQ8014: WebSphere MQ channel created.
```

```
DEF QL(QM_SVC_HOST) DEFPSIST(YES) MAXDEPTH(5000000) USAGE(XMITQ)
TRIGGER TRIGTYPE(FIRST) TRIGDATA(CLNT_HOST.SVC_HOST)
INITQ('SYSTEM.CHANNEL.INITQ') REPLACE DESCR('transmission queue to
CLNT_HOST')
   4 : DEF QL(QM_SVC_HOST) DEFPSIST(YES) MAXDEPTH(5000000) USAGE(XMITQ)
TRIGGER TRIGTYPE(FIRST) TRIGDATA(CLNT_HOST.SVC_HOST)
INITQ('SYSTEM.CHANNEL.INITQ') REPLACE DESCR('transmission queue to
CLNT_HOST')
AMQ8006: WebSphere MQ queue created.

DEF QL(BANKING.SERVICE.RESPONSE) DEFPSIST(YES) REPLACE DESCR ('Queue on
which service responses arrive')
   5 : DEF QL(BANKING.SERVICE.REQUEST) DEFPSIST(YES) REPLACE DESCR
('Queue on which service responses arrive')
AMQ8006: WebSphere MQ queue created.
```

An illustration of what this script produces and how messages flow between the Web Service and the client is provided in 10.5.4, “Executing a deployment to a remote queue manager” on page 232 and 8.4.4, “Executing a deployment to a remote queue manager” on page 171.

8

Axis Web Service

This chapter demonstrates the creation and deployment of an Axis Web Service, which sends its SOAP messages over WebSphere MQ instead of Hypertext Transfer Protocol (HTTP). Web Services are based on a request-and-response model that uses messages encoded in SOAP, which is a messaging protocol designed to be network-neutral, transport-neutral, and programming language-neutral. These messages are formatted with the popular Extensible Markup Language (XML). Typically, SOAP messages are sent through HTTP, which is the underlying protocol used by the World Wide Web. The SOAP protocol is transport-independent. Therefore, WebSphere MQ is used as an alternative transport mechanism. This chapter discusses how to write an Axis Web Service that uses WebSphere MQ as the transport mechanism.

This chapter covers the following topics:

- Creating an Axis Web Service
- Setting up an environment for deploying the Axis Web Service
- Deploying the Axis Web Service
- Security considerations and enablement
- Error handling

© Copyright IBM Corp. 2006. All rights reserved.

8.1 Design

In order to illustrate a Web Service using WebSphere MQ as the transport mechanism, it is necessary to develop some sample business functionality in order to create the service to be exposed. The main focus of this chapter is the Web Service infrastructure. Therefore, the service functionality is deliberately kept simple. In this illustration, two simple classes are created to mimic some basic bank account functionality. Figure 8-1 illustrates the classes.

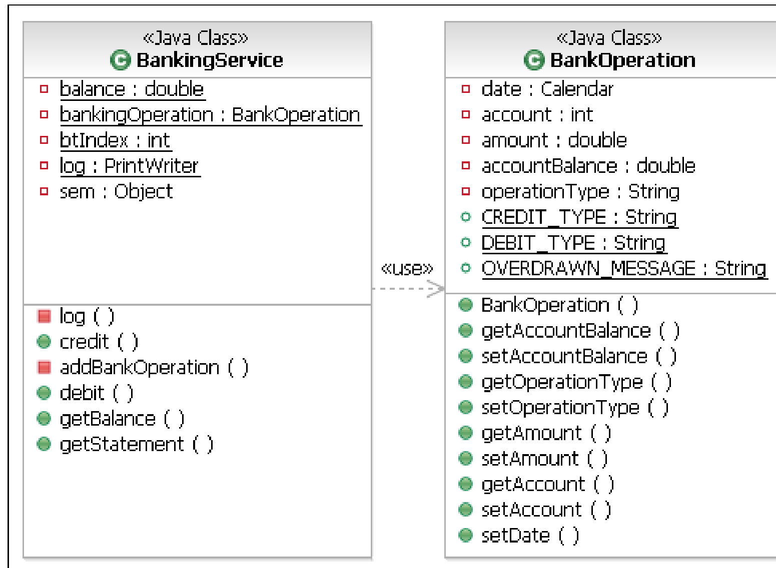


Figure 8-1 Banking service class figure

Note the following points pertaining to these classes:

- getStatement returns a complex user-defined type
- debit throws a BankOperationException
- BankOperation contains accessor methods in order to enable serialization

Table 8-1 lists the methods that are exposed by the Web Service, and a brief description of each method.

Table 8-1 BankingService method description

Method	Description
debit	Removes specified amount for transfer to the account ID provided. This implementation simply subtracts the amount specified from the balance. If the amount is greater than the balance, an exception is thrown.

Method	Description
credit	Adds specified amount to the current balance
getBalance	Returns the current balance
getStatement	Returns an array of BankOperation objects

In this chapter, the BankingService class is implemented as a Web Service using WebSphere MQ. The infrastructure for implementing this is slightly more complicated than a standard Web Service because the transport mechanism must be altered. WebSphere MQ is used instead of the typical HTTP method. This is illustrated in Figure 8-2.

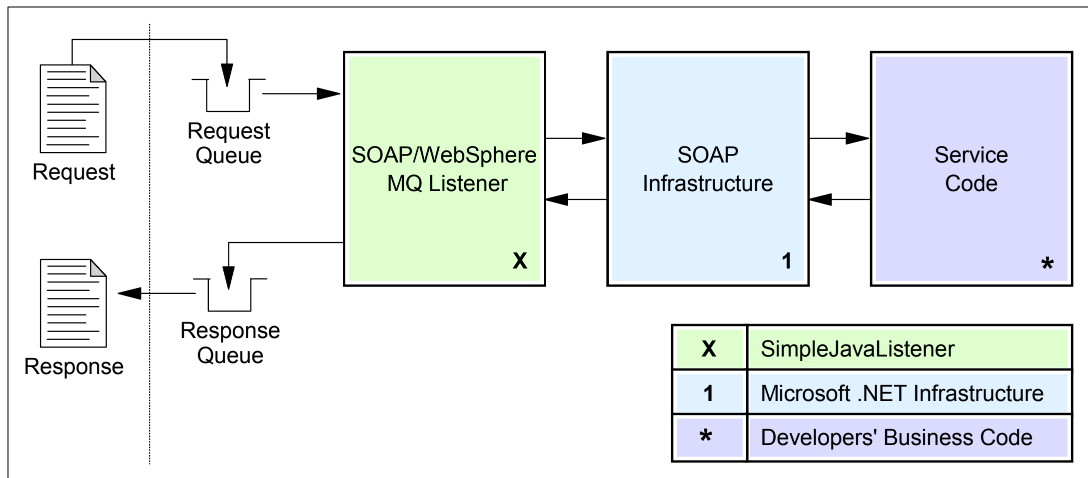


Figure 8-2 SOAP WebSphere MQ infrastructure on the service side

Figure 8-2 illustrates the main components on the service side of a WebSphere MQ Web Service. The Service Code, indicated by an asterisk (*) in Figure 8-2, is the code containing the business functionality. In our example, this is the BankingService class. The component in the middle, indicated by 1, is part of the infrastructure that handles the interaction with SOAP. In this example, it is Axis. Both these components exist in the standard HTTP Web Service infrastructure. The additional component, the SOAP/WebSphere MQ listener, indicated by an

X, is the new component of the infrastructure. This listener is responsible for interfacing with WebSphere MQ in order to perform the following actions:

- Read request messages from the request queue
- Write response messages to the response queue

For a complete overview of the infrastructure, including the client and the service, refer to Chapter 4, “WebSphere Services with WebSphere MQ” on page 29.

8.2 Requirements

In order to implement an Axis service using WebSphere MQ, install the following software on the development machine:

- Java Software Development Kit (SDK) V1.4.2
- WebSphere MQ V6
- Microsoft .NET software development kit (SDK) V1.1
- Rational Application Developer V6 (optional)

  Rational Application Developer is used to develop the Axis Web Service in this chapter.

Attention: Microsoft .NET Framework is required for the deployment tool, which generates proxies in Java, C#, and VisualBasic.NET (VB.NET).

Note: Ensure that in the Features window of the WebSphere MQ, the **Install the Java Messaging and SOAP Transport** option is selected. For more details, see 7.2.1, “Installing IBM WebSphere MQ V6” on page 142.

8.3 Implementation

This section discusses the implementation of the BankingService Web Service used in this chapter. The BankingService Web Service code is available for download in Appendix D, “Additional material” on page 431.

8.3.1 Implementation of Web Service

In order to help those who do not have the source code download to carry out the actions described in this chapter, the method stubs provided by the service are shown here. Example 8-1 shows the service method stubs.

Example 8-1 Service method stubs

```
public boolean credit (double amount)

public void debit(int account, double amount) throws
BankOperationException

public double getBalance()

public BankOperation[] getStatement()
```

The BankOperationException

When a debit operation is requested on an account, if the debit amount is greater than the account balance, a BankOperationException derived from java.lang.exception is thrown. While this may not be the ideal way to handle a client becoming overdrawn, it does allow an exception to be easily generated. This makes exception handling easy to demonstrate.

After implementing the code, compile it. However, because the deployment tool compiles the service code, this step is not necessary.

8.3.2 Preparing the WebSphere MQ environment

The first deployment demonstrated in this chapter is a simple deployment. This may require additional WebSphere MQ configuration.

For this simple deployment to work, perform the following WebSphere MQ configuration tasks:

- Ensure that a default queue manager is created to use the supplied Universal Resource Indicator (URI).
- Run the setupWMQSoap script. This script is called by the IVT tests provided with WebSphere MQ for SOAP.

> **Note:** The deployment tool can be used on any queue manager, not just the default queue manager.

To create a default queue manager, refer to the details provided in 7.3.1, “Basic WebSphere MQ administration” on page 151.

Three different deployment scenarios are illustrated in this chapter. Each scenario requires a different WebSphere MQ configuration. These are discussed against the corresponding deployment scenario. In summary, the three scenarios require the following configurations:

- Executing a simple deployment
  - Queue manager configured through setupWMQsoap
- Specifying local deployment, response, and request queues
  - Queue manager with a response and request queue created
- Deployment creating client connection proxies
  - Queue manager for service with server connection channel created
- Deployment creating queue manager-to-queue manager proxies
  - Queue manager for service
  - Queue manager for client

8.4 Deployment

The deployment phase is the key to implementing WebSphere MQ as a transport mechanism for the Web Service. The steps described up to this point apply to a standard Axis Web Service too.

The deployment process described here illustrates the use of a deployment utility supplied with WebSphere MQ for SOAP. This utility consists of:

- amqwdeployWMQService.java

  This is the Java source code for the deployment utility.
- amqwDeployWMQService.cmd

  This is a Windows script to launch the Java deployment code.
- amqwDeployWMQService.sh

  This is a shell script to launch the deployment utility.

In this chapter, the supplied deployment utility, henceforth referred to as amqwdeployWMQService, is sufficient. However, in more complex scenarios, amqwdeployWMQService must be customized. Hence, the inclusion of the source code in the WebSphere MQ transport for SOAP installation. The scenarios that may require the customization of amqwdeployWMQService include the following:

- Deploying from the existing Web Services Description Language (WSDL), rather than from the service source code
- A Web Service returning a complex object is defined in a different package from that of the Web Service

The deployment process may vary in complexity, depending on the environment the developer is working on. A number of deployment scenarios are illustrated in this chapter. The first scenario is simple. Thereafter, the scenarios increase in complexity. 8.4.1, “Common deployment steps” on page 165 describes moving the code and setting up the classpath, the two steps that are common for all deployments.

8.4.1 Common deployment steps

The tasks that are common to any deployment are split into the two steps discussed here.

- Moving the source code

  Although this is not a requirement, copying the source code to an empty directory is recommended. This is mainly due to the amqwdeployWMQService generating a number of files. The use of an empty directory is recommended because this makes it easier to see the output of amqwdeployWMQService and avoids interference with any other deployment directives.

  In this chapter, the folder c:\temp\AXISSvc\ is created. The sample service uses the package bankingService; package definition. Therefore, in this case, the two Java files are copied to c:\temp\AXISSvc\bankingService\.

> **Tip:** The code must be placed in this directory structure for the **`javac`** command used by amqwdeployWMQService to work. For further information, refer to the following Web site:
>
> `http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html`

- Setting the classpath

  Because amqwdeployWMQService is a Java program, it requires the classpath to include specific Java libraries. A template script, amqwsetcp, is provided with WebSphere MQ to set the classpath in order to include these directories.

  In practice, amqwsetcp may have to be customized to include additional classpath configurations for the service source code.

  > **Tip:** The amqwsetcp script is contained in the bin directory of the WebSphere MQ install. If there is a *not recognized* message, check if this directory is set in the path.

  If this runs successfully, there is no output from the script, as seen in Figure 8-3.

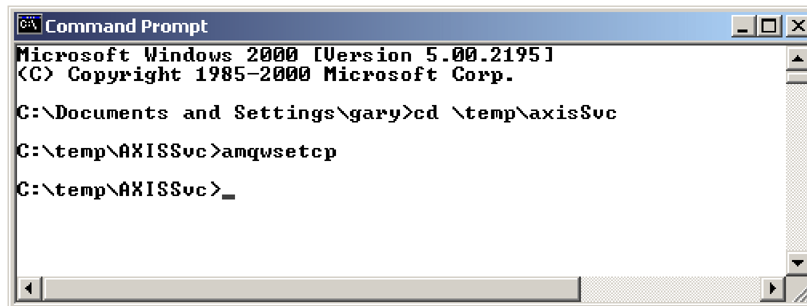


Figure 8-3 Configuring the classpath

It may be necessary to add the directory created for the source code to the classpath. In this case, it involves adding c:\temp\AXISSvc\ to the classpath.

Tip: An easy way of doing this in Windows is to output the classpath after executing amqwsetcp to a batch file and editing it by performing the following tasks:

1. Issue the following command:
   `SET CLASSPATH > setMyCP.bat`
2. Open **setMyCP.bat** in a text editor.
3. At the start of the file, add the word `SET`.
4. At the end, add `;<source directory>`.

The file looks as follows:

```
SETCLASSPATH=classes;generated\client\remote;generated\server;c:\
temp\AXISSVC;
```

8.4.2 Executing a simple deployment to a local default queue manager

This section provides more details about WebSphere MQ configuration and the deployment of an Axis Web Service that uses queue manager local to the service.

Additional WebSphere MQ configuration

As part of the deployment process, a request queue is created. For this simple deployment to work, a default queue manager and a response queue must be set up.

Important: A listener configuration is created during the deployment process for a particular request queue. This listener listens only on the specified request queue.

Because the only action performed on the response queue by a listener is a write, multiple listeners can share a response queue.

In a simple deployment, where no queue manager or response queue is specified, amqwdeployWMQService uses defaults. The default queue manager is the queue manager listening on port 1414. If a default queue manager is created as part of the WebSphere MQ install, this is the queue manager used. See 7.3.1, “Basic WebSphere MQ administration” on page 151 for further details.

The default response queue is a local queue called SYSTEM.SOAP.RESPONSE.QUEUE.

At the very least, the response queue must be created, and possibly, a default queue manager. To help with this, a script is supplied as part of the SOAP/WebSphere MQ install. This script is called setupWMQSOAP, and is found in the location <WebSphere MQ install directory>\Tools\soap\samples.

Executing setupWMQSOAP with no parameters leads to a queue manager called WMQSOAP.DEMO.QM being created. A default response queue is created in this queue manager. Alternatively, you can specify a queue manager name as a parameter:

```
setupWMQSOAP myQueueManager
```

If a queue manager is specified and it does not exist already, the queue manager is created. If the queue manager is created or if it already exists, a default response queue is created.

> **Tip:** SYSTEM.SOAP.RESPONSE.QUEUE is a system queue. In order to view this queue in WebSphere MQ Explorer, select the **Show System Objects** option.

Deployment

After completing the WebSphere MQ configuration, run amqwdeployWMQService. The classpath configuration mentioned earlier is required by amqwdeployWMQService. It therefore follows that you must run amqwdeployWMQService from within the same command prompt as the classpath script. amqwdeployWMQService can take a number of parameters. This initial deployment involves a simple configuration, where you must specify only one parameter, that is, the source file containing the service BankingService.java. In situations with multiple source files, specify the name of the source file containing the methods exposed by the Web Service. Qualify this file name by the package name. Thus, in this example, bankingService/BankingService.java is the full file name.

The decision about which of the two script files (amqwdeployWMQService.cmd and amqwdeployWMQService.sh) supplied with the deployment utility must be used, is based on the deployment platform. Figure 8-4 illustrates the execution of the deployment utility in the Windows environment, that is, amqwdeployWMQService.cmd.

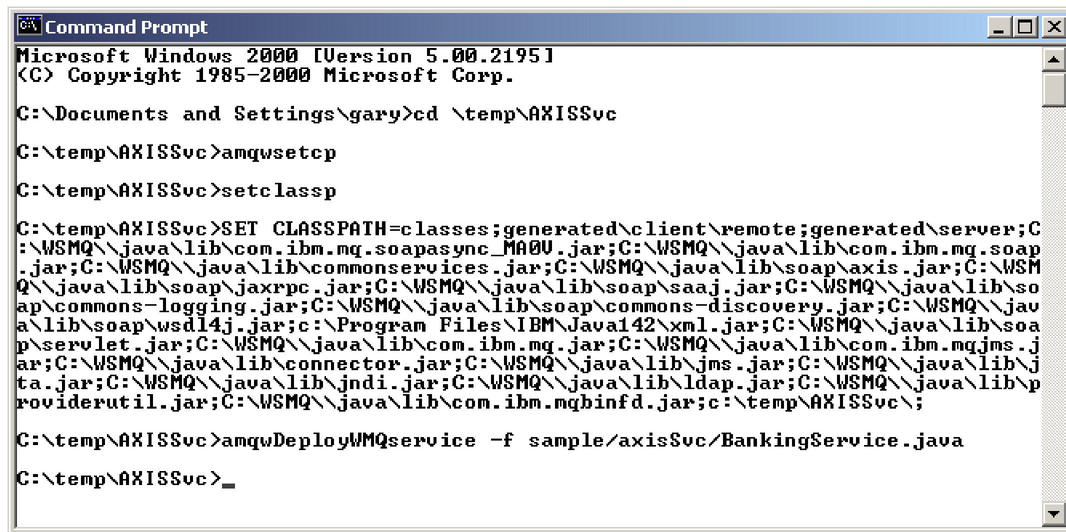


Figure 8-4 Simple deployment of BankingService.java

As the window shown in Figure 8-4 demonstrates, by default, a successful deployment generates no output on the screen. The amqwdeployWMQService output in a non-Windows environment looks similar. If the -v switch is used, the deployment tool shows details about all the actions performed.

Within the directory that amqwdeployWMQService is run, there must be an additional file and folder. The new file is server-config.wsdd, a generated service deployment descriptor for Axis. The new folder is called Generated, and contains the rest of the output from amqwdeployWMQService, including the following:

- WDSL file

  This is named after the service, including the package name, with the suffix _Wmq. In this case, it is bankingService.BankingService_Wmq.wsdl.

- A folder named Client

  This contains the generated proxy code for VisualBasic and C# proxy code with the suffix Service. In this case, it is BankingServiceService.cs and BankingServiceService.vb.

- A folder named Remote

  This contains the Java client code in the appropriate folder structure.

- A folder named Server

  This contains the script files to start and stop the listener. For a Windows system, they are called startWMQJListener.cmd and endWMQJListener.cmd.

- A folder structure based on the package name

  In this case, this is a folder called BankingService. At the bottom of this structure is the class file for the service and Axis deploy/undeploy files for the services.

In order to use WebSphere MQ as a transport mechanism, the service has to access a queue manager. There is also a request queue set up by the SOAPJ.bankingService.BankingService deployment utility.

This is the request queue used by the service. Client request messages are placed on this queue. Because a request queue was not specified during deployment, amqwdeployWMQService creates a request queue. The name of the new queue is a generated name based on the service package and source file names.

8.4.3 Executing a deployment to a local queue manager with specific request and response queues

This section describes a deployment that is similar to that described in 8.4.2, "Executing a simple deployment to a local default queue manager" on page 167. The difference is that in this deployment, the request and response queues are specified.

> **Tip:** If the deployment described in 8.4.2, "Executing a simple deployment to a local default queue manager" on page 167 has been performed, delete the Generated folder. If an identical redeployment is to be performed, delete the request queue before the deployment. If this does not happen, the deployment utility produces the following error:
>
> ```
> Queue SOAPJ.sample.axisSvc.BankingService already in use.
> ```

To aid consistency across scenarios, a specific queue manager, QM_localToSvc, is created for this deployment.

To begin with, create the request and response queues. Both these are local queues with no special attributes:

- BANKING.SERVICE.REQUEST
- BANKING.SERVICE.RESPONSE

> **Tip:** For a simple guide to creating local queues, see 7.3.1, "Basic WebSphere MQ administration" on page 151.

When amqwdeployWMQService is run, it should be told to use the newly created queues. This configuration information forms a part of the URI used to locate the service. The deployment described in 8.4.2, "Executing a simple deployment to a local default queue manager" on page 167 masked the URI, generating it automatically. In order to specify the request and response queues, the call to amqwdeployWMQService must provide the URI. Use a command-line switch, -u, for this. The complete command line used for this deployment is shown in Example 8-2.

Example 8-2 Using the -u command-line switch

```
amqwdeployWMQService -f sample/axisSvc/BankingService.java -u
"jms:/queue?destination=BANKING.SERVICE.REQUEST
@QM_localToSvc&connectionFactory=()&replyDestination=BANKING.SERVICE.RE
SPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi
```

> **Tip:** This URI is valid only for deployment to a queue manager listening on the default port used by WebSphere MQ, 1414. For queue managers using nondefault ports, additional parameters must be specified. This is discussed further in 8.4.4, "Executing a deployment to a remote queue manager" on page 171.

8.4.4 Executing a deployment to a remote queue manager

The connectionFactory part of the URI can be set up to establish different types of connections to a remote queue manager.

Client connection

A scenario that is more realistic than the two previous scenarios involves the client and the service running on separate machines. This section illustrates deploying a service for such a configuration. The service runs on a machine that has WebSphere MQ installed. This machine has a local queue manager used by

the service. The service is configured to support a client connecting from a machine with no local queue manager. Instead, it only has the WebSphere MQ client installed. To facilitate this, create a server connection. This configuration is illustrated in Figure 8-5.

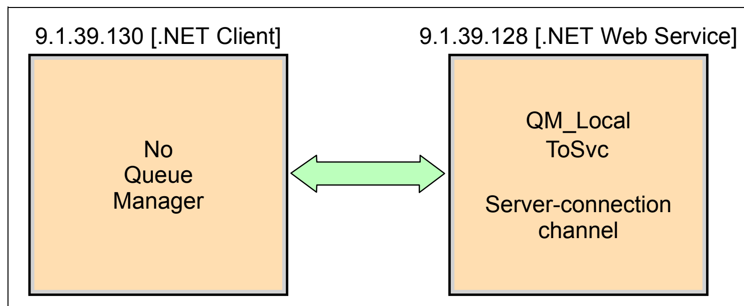


Figure 8-5 Environment setup for a client mode connection

Create the server connection channel on the queue manager used by the service, in this case, QM_localToSvc.

To create this channel, use the runmqsc utility. This is a command-line utility supplied with WebSphere MQ, which provides powerful scripting support. The simplest form of launching it takes one parameter, that is, the queue manager name:

```
runmqsc QMgrName
```

Here, *QMgrName* is the name of the queue manager to script against.

To run a script, use the following syntax:

```
runmqsc QMgrName <myScript.txt>
```

Here, *myScript.txt* is the name of the script file.

The command to create the server connection channel is shown in Example 8-3.

Example 8-3 Creating a server connection channel

```
DEFINE CHANNEL (BANKING.SVR.CHL) CHLTYPE(SVRCONN) +
TRPTYPE(TCP) MCAUSER('demoUser') +
DESCR('Server connection channel for BankingService') replace
```

Note: For demonstration purposes, the server connection channel MCAUSER is set to a userID that is in the mqm user group, for the client connection to be successful. This is purely for purposes of simplicity and is *not* recommended in a production environment. The security implications of this must be considered.

Because only a single command must be issued, using a script is not necessary. Instead, launch the utility as discussed earlier and enter the command in an interactive format. To summarize, perform the following tasks:

1. Start the runmqsc utility.
2. Issue the command provided in Example 8-3 to create the channel.
3. Quit the runmqsc utility using the END command. This is illustrated in Figure 8-6.

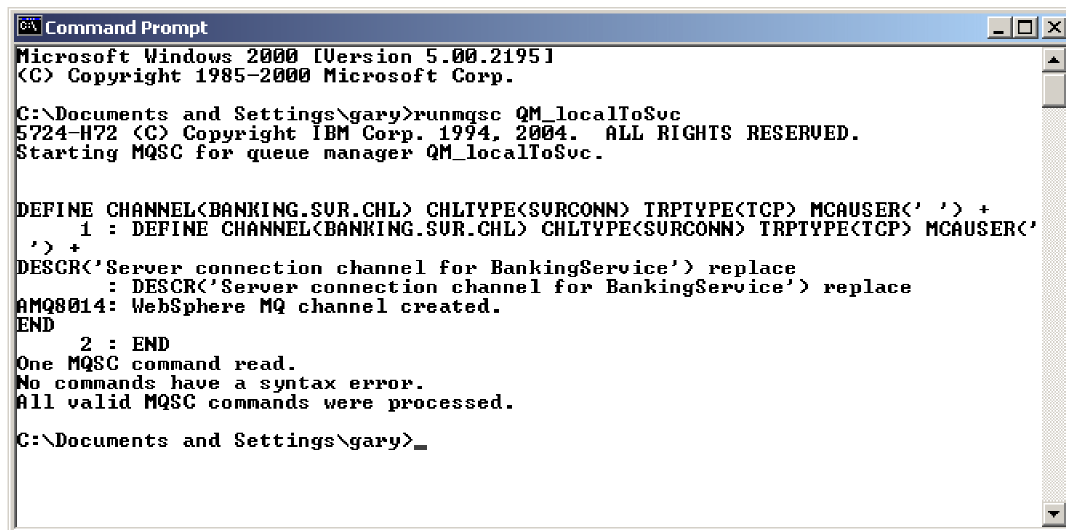


Figure 8-6 Creating a server connection channel

4. After the channel is created, run the deployment utility. The command for this is shown in Example 8-4.

Example 8-4 Command to run deployment utility

```
amqwdeployWMQService -f sample/axis/BankingService.java -u
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_localToSvc&connectionFactory=conne
ctQueueManager(QM_localToSvc)binding(client)clientChannel(BANKING.SVC.CHL)clientConne
ction(9.1.39.127%25281420%2529))&replyDestination=BANKING.SERVICE.RESPONSE&initialCon
textFactory=com.ibm.mq.jms.Nojndi"
```

The difference between this URI and the URI in Example 8-3 on page 172 is that there are a number of parameters specified for the connectionFactory attribute. When amqwdeployWMQService generates the proxy files, it includes this URI in the proxy code. A detailed explanation of the URI is provided in “Uniform Resource Indicator syntax” on page 66. ConnectionFactory allows you to specify in more detail how the WebSphere MQ connection is made. In this context, note the following points:

- The queue manager name is specified as a parameter to the connectQueueManager attribute.
- The binding type is specified as a parameter to the binding attribute.
- The name of the client channel is specified as a parameter to the clientChannel attribute.
- The IP address or host name is specified as a parameter to the clientConnection attribute.

It is important to note that this example uses a nonstandard port to connect to WebSphere MQ. A nonstandard port is used to illustrate an URI when connecting to a queue manager listening on a port, other than the default. This clientConnection attribute includes a suffix to the IP address 9.1.39.127. This suffix specifies the %2528<u>1420</u>%2529 port. This example uses the 1420 port.

Tip: The SOAP WebSphere MQ infrastructure does *not* recognize bracket characters. Insert these into an URI by using a combination of escape and American Standard Code for Information Interchange (ASCII) characters. The “(” character is represented by %2528, and the “)” character is represented by %2529.

Server binding mode connection

The final deployment scenario involves queue manager-to-queue manager communication. This is where the client application is connected to one queue manager and the service to another queue manager. Typically, this scenario sees the service deployed on a different machine from the client application. Figure 8-7 illustrates this.

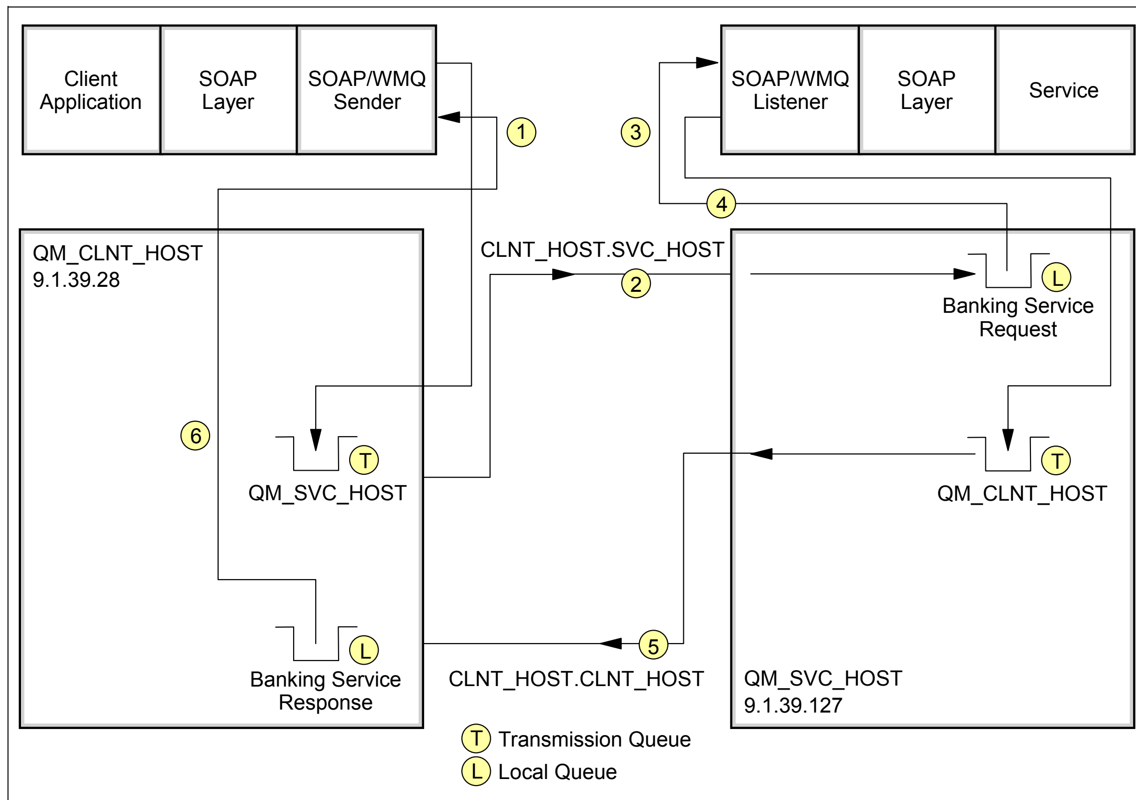


Figure 8-7 Information flow: Configuration using different queue managers for client and service

Following is the procedure followed for this deployment scenario:

1. The client begins by sending a request.
2. The SOAP/WebSphere MQ sender knows that this request is destined for the service queue manager from the URI. This is because the destination parameter includes the service queue manager name. The SOAP/WebSphere MQ sender places the request on QM_SVC_HOST. This request then travels across a channel between the two queue managers.
3. The service listener reads the message from the request queue.

4. The request is passed to the service and processed.
5. The resulting response is placed on the transmission queue by the service listener.
6. The service listener determines the destination queue manager through the message header.
7. The message returns to the client queue manager through a channel.
8. The sender reads the response from the response queue and passes it to the client applications.

In order to deploy a service for this scenario, create a service queue manager, QM_SVC_HOST, which is shown in Figure 8-7. Scripts to configure WebSphere MQ are included with the source code download. To configure a service queue manager called QM_SVC_HOST with the downloaded script file, use the following command:

```
runmqsc QM_SVC_HOST < WMQ_SVC_HOST_01.txt
```

For more information about the WebSphere MQ configuration for the machines hosting the service and the client, see 7.3.1, "Basic WebSphere MQ administration" on page 151.

After the service machine is correctly configured, start the deployment utility with the command shown in Example 8-5. After the channel is created, start the deployment utility with the command shown in Example 8-5.

Example 8-5 Deploying BankingService

```
amqwdeployWMQService -f sample/axis/BankingService.java -u
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_SVC_HOST&connectionF
actory=connectQueueManager(QM_SVC_HOST)&replyDestination=BANKING.SERVIC
E.RESPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi"
```

This URI is similar to that used in the earlier scenarios. The only difference is that the queue manager has a new name, QM_SVC_HOST. The difference comes from the connecting client. It must override the URI in order to specify its local queue manager, QM_CLNT_HOST, for the destination. This is covered in detail in Chapter 9., "Axis client" on page 187. Refer to Figure 9-8 on page 206.

Axis client

An important point to note is that along with the proxies generated, an Axis client requires an additional file to start. This file is called client-config.wsdd and is used to define the prefix jms: as a valid prefix for an URI. Without this prefix, the Axis infrastructure generates an error every time an attempt is made to invoke one of the services.

At the time of deployment, this file is generated by a utility supplied with WebSphere MQ, amqwclientConfig. This must be run from the same directory as the deployment process. In our case, it is c:\temp\axisSvc. It takes no parameters.

The output file client-config.wsdd is placed in the same directory and must be placed in the classpath of an Axis client.

The SOAP WebSphere MQ listener

The earlier sections illustrated deployments of gradually increasing complexity. Each of these saw the service deployed on the local machine and the appropriate proxies generated. When the service is ready for use, one more step is necessary before the service starts processing the requests, that is, it must be started. Rather, the service listener must be started. The service listener is responsible for reading messages from the request queue and forwarding them to the custom service code. The service listener is generated as part of the deployment, and can be started using a single command. To do this, perform the following tasks:

1. Switch to the server directory found in <code root directory>/generated/server.
2. Enter the following command:

   ```
   startWMQJListener
   ```

A series of commands are shown in a window (Figure 8-8), with the last command starting with the word **java** and ending with the URI and the number of service listener threads executing.

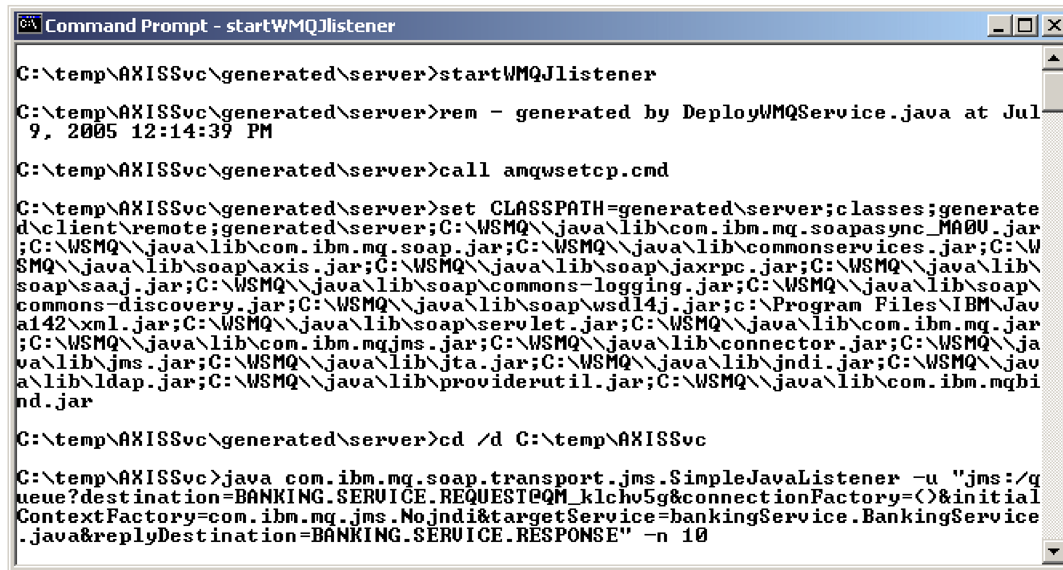


Figure 8-8 Starting a service listener

By default, the service listener runs indefinitely, unless a serious error occurs, or the listener is stopped. The command to stop the listener is similar to the command used to start it:

1. Open a new command prompt.
2. Switch to the server directory found in <code root directory>/generated/server.
3. Enter the following command:

```
endWMQJListener
```

The service listener ends with the command prompt containing the listener, as shown in Figure 8-9.

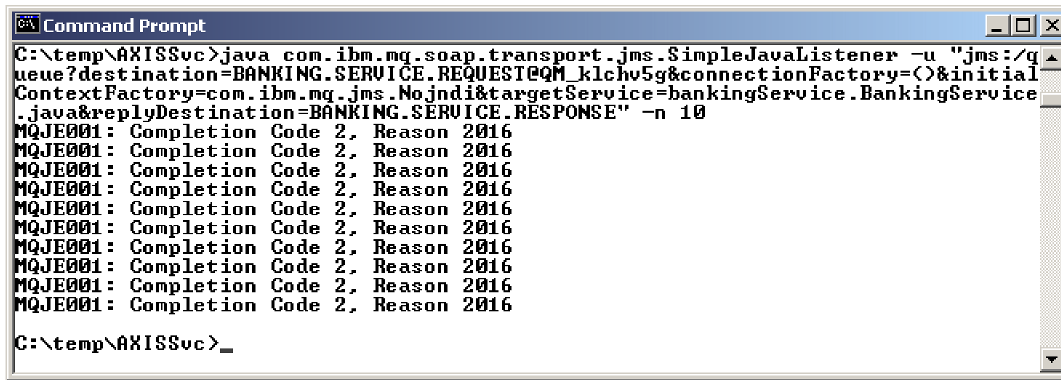


Figure 8-9 A service listener ending

A WebSphere MQ error code is shown numerous times. This is shown once per thread and must be expected. It is a byproduct of the way the listener ends.

Attention: We recommend that you do *not* close the listener using the Ctrl+C keys or by closing the command prompt. The safest way to close the service listener is to follow the procedure just described.

8.5 Error handling

This section explains some of the common error messages you may see when executing a SOAP WebSphere MQ Web Service.

Unable to get response from queue

Problems occur if the service is unable to get messages from the request queue. This is simulated in the test environment by using the WebSphere MQ Explorer to configure the request queue, so that applications cannot get messages from the queue. The output on the service listener (Figure 8-10), is the result of this action.

Figure 8-10 Listener unable to get messages from the request queue

This output shows an error code indicating that a queue is inhibited. The WebSphere MQ error code is repeated ten times in Figure 8-10 because ten listener threads are executing.

Unable to find specified request queue

This error condition is most likely to occur if the request queue does not exist. In our example, the service listener fails to start. The error message that is shown looks similar to that shown in Figure 8-11.

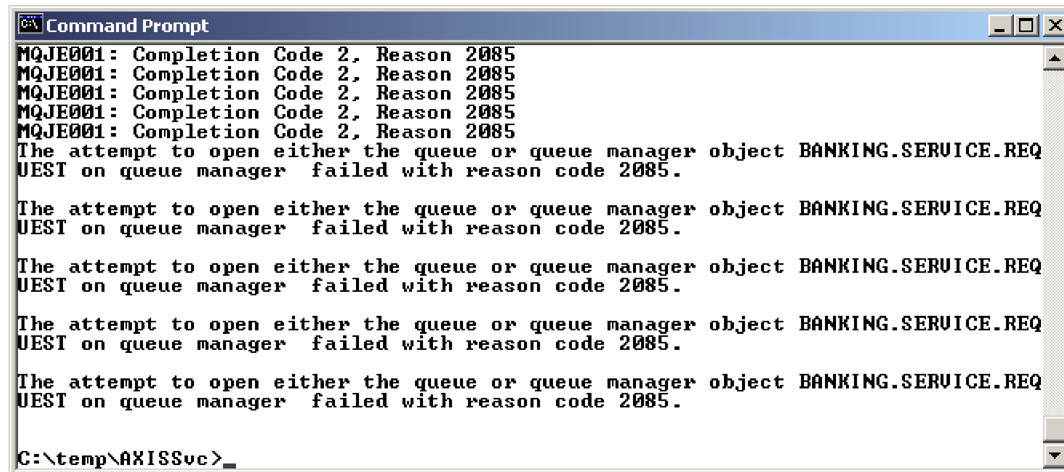


Figure 8-11 Output of listener if the request queue does not exist

The error code output is the error code for an unknown object and the object in this case is a queue that WebSphere MQ cannot find as displayed.

> **Tip:** To obtain a brief description of a WebSphere MQ error, use the utility mqrc. The syntax is:
>
> ```
> mqrc <error code>
> ```

Unable to put to a response queue

The response queue is another source of problem. One possible error is the listener not being able to put a message to the response queue. In the test environment, this is simulated by inhibiting the put functionality on the response queue. The output from the listener is as shown in Figure 8-12.

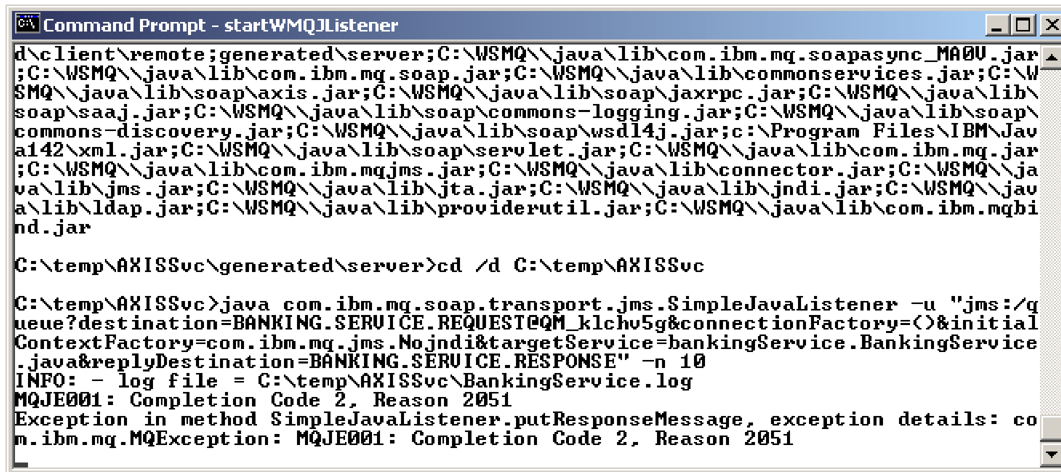


Figure 8-12 Listener is unable to put a message to the response queue

The result is an exception message with the code 2051. The mqrc utility informs that this is an error code, indicating that a put operation is inhibited.

Unable to find specified response queue

What happens to a message when a put operation fails depends on the integrity and persistence settings of the message. The first check is against the message integrity, the default rule being that for low-integrity messages, a warning is shown and the message discarded. Persistent messages are backed out and the put retried, with an error message shown. This sequence is repeated until the backout threshold is exceeded.

> **Note:** The default backout threshold is three. To change this, use the -b switch during deployment. For further details, see 5.4.2, "The SOAP/WebSphere MQ Universal Resource Indicator" on page 65.

Message persistence affects the error handling process in a similar way for a failed put operation. In general, nonpersistent messages result in the service listener showing the error message and discarding the message. Persistent messages are backed out retried, as mentioned earlier.

> **Note:** The behavior based on message integrity and persistence can be altered during the deployment process. For further information, refer to 5.4.2, “The SOAP/WebSphere MQ Universal Resource Indicator” on page 65.

Another potential problem for the SOAP WebSphere MQ listener is if the response queue specified in the URI does not exist. In this case, the SOAP WebSphere MQ listener starts as usual and shows no error messages. This is the correct behavior because the client can potentially override the URI, thereby specifying a different response queue. Error messages, if any, are shown on the client. This involves an exception containing the WebSphere MQ error code 2085. For further details, see 9.4, “Error handling” on page 208.

Unexpected message on a queue

It is not just missing queues or limited queues that may lead to errors. An unexpected message on a queue may also cause problems. In this example, it is an unexpected message on a request queue. In the test environment, this is simulated by placing a simple text message on the request queue. The resulting output from the listener is as shown in Figure 8-13.

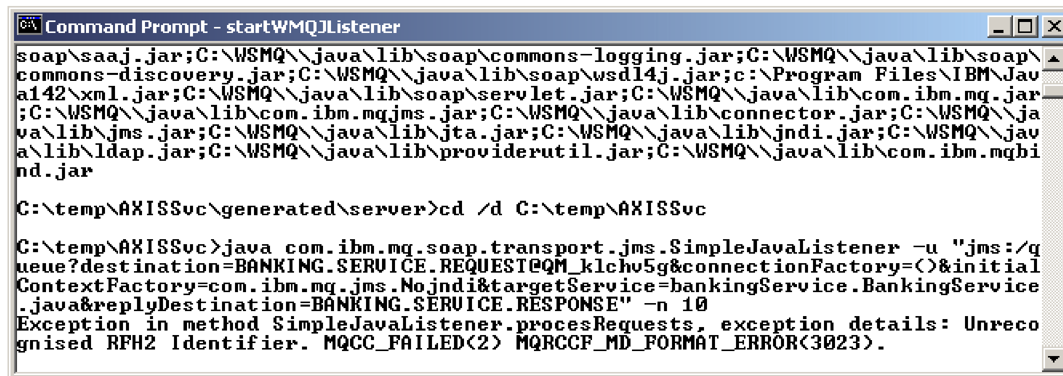


Figure 8-13 An unexpected message on the request queue

As illustrated, the listener shows an error message, and a report message is returned. The listener continues executing, and the subsequent messages are processed as usual, provided they are in the correct format.

8.6 Security

Securing communication between a Web Services client and a Web Service is most effectively achieved using the Secure Sockets Layer (SSL). This section describes the enablement of the security services provided by SSL within the scenarios described in the earlier sections.

The URI provides several key words that can be configured to enable SSL. Depending on the Web Services client environment, different key words exist. In a Java environment, these are:

- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword
- sslCipherSuite

In a Microsoft .NET environment these are:

- sslKeyRepository

> **Note:** No password is explicitly required for this because this is stored on creation of the .kdb file, in the stash file.

- sslCipherSpec

Before setting these values on the URI, create and configure the key repositories and certificate chains. In the examples provided in the list that follows, it is assumed that this initial configuration is completed. Refer to Chapter 6, “Security” on page 107 for further details. The key store locations and passwords discussed in Chapter 6, “Security” on page 107 are used in Example 8-6.

To enable security using a Java client, set the following values on the URI at deployment:

- sslKeyStore=C:\SSL\client\key without the .jks extension
- sslKeyStore=password
- sslTrustStore=C:\SSL\client\key or C:\SSL\client\trust if the trust store is different from the key store, without the .jks extension
- sslTrustStorePassword=password
- sslCipherSuite=SSL_RSA_WITH_3DES_EDE_CBC_SHA

The values of the sslKeyStore and sslTrustStore must be the locations on the machine that the client is running on, whether that machine is remote or local to the Web Service. The sslCipherSuite value that is provided in this list is one of the many that can be selected. For more information about the possible choices, refer to *WebSphere MQ Using Java,* SC34-6591.

A full URI may look as shown in Example 8-6.

Example 8-6 A full URI: Java environment

```
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_SVC_HOST&connectionF
actory=connectQueueManager(QM_SVC_HOST)&replyDestination=BANKING.SERVIC
E.RESPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi&sslKeyStore=C:\S
SL\client\key&sslKeyStorePassword=password&sslTrustStore=C:\SSL\client\
trust&sslTrustStorePassword=password&sslCipherSuite=SSL_RSA_WITH_3DES_E
DE_CBC_SHA"
```

If the Web Services client is running a Microsoft .NET environment, set the values on the URI as follows:

- sslKeyRepository=C:\SSL\client\key without the .kdb extension
- sslCipherSpec=TRIPLE_DES_SHA_US

The value of sslCipherSpec given here is the equivalent of the sslCipherSuite value given for the Java client. There are many choices for sslCipherSpec that you can select. For more information about the choices, refer to *WebSphere MQ Security,* SC34-6588.

A full URI may look as shown in Example 8-7.

Example 8-7 A full URI: Microsoft .NET environment

```
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_SVC_HOST&connectionF
actory=connectQueueManager(QM_SVC_HOST)&replyDestination=BANKING.SERVIC
E.RESPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi&sslKeyRepository
=C:\SSL\client\key&sslCipherSpec=TRIPLE_DES_SHA_US"
```

It is important that the value selected for sslCipherSuite or sslCipherSpec is equivalent to the one set on the SVRCONN’s SSLCIPH parameter on the WebSphere MQ queue manager being connected to.

Optionally, it may also be necessary to use the sslPeerName value on the URI. Using this option forces the client to send the WebSphere MQ queue manager its certificate, so that the queue manager can check if the distinguished name on the certificate matches the distinguished names it has been configured to trust.

8.7 Using the Web Service

After the Web Service is created and deployed, the Web Service can be used with a client that resides on the same machine or with a client that resides on a different machine from the Web Service. One final action is required for the Web Service to be usable, that is, the listener must be started. To do this, use the startWMQJListener script created by the deployment process.

In a situation where the client resides on a different machine from the Web Service, the Web Service can be deployed on the server machine and the proxies copied across to the client machine. The Web Service can also be deployed on both the machines. The redundant elements from each platform can be removed. See 5.4, “The deployment process” on page 59 for further details.

8.8 Summary

This chapter discussed the creation of a Web Service using WebSphere MQ as the transport mechanism. A simple class providing four methods to be exposed as services, was created. This class was then deployed as a Web Service using WebSphere MQ in a number of different configurations. These configurations started small and gradually increased in complexity. Although real world scenarios involve even greater complexity, the aim of this chapter is to introduce the concepts involved.

This chapter also discussed simple error handling and security concepts. These concepts are discussed in greater detail in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 and Chapter 6, “Security” on page 107.

9

Axis client

This chapter introduces an Axis client. A client must be capable of calling any of the services created in the previous three chapters, regardless of its implementation. This chapter discusses and demonstrates the process involved in writing a Java client for a Web Service that uses WebSphere MQ as the transport mechanism.

© Copyright IBM Corp. 2006. All rights reserved.

9.1 Design

To demonstrate how to call a Web Service that uses WebSphere MQ as the transport mechanism, this chapter explains the development and deployment of a simple client. The client is written in Java and uses the Axis infrastructure. It provides a mechanism to utilize any of the Web Services discussed in Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269.

The client consists of two Java files, as shown in Table 9-1.

Table 9-1 Client source code structure

File name	Description
BankClient.java	Implements a main method. Locates the service and starts the graphical user interface (GUI).
BankingGUI.java	Implements the GUI. Calls the appropriate service methods based on user input.

The client is a simple implementation. The focus of this chapter is to connect to the SOAP WebSphere MQ framework rather than the Java graphical user interfaces.

Figure 9-1 provides a high-level view of the significant components of the client side of the SOAP WebSphere MQ configuration.

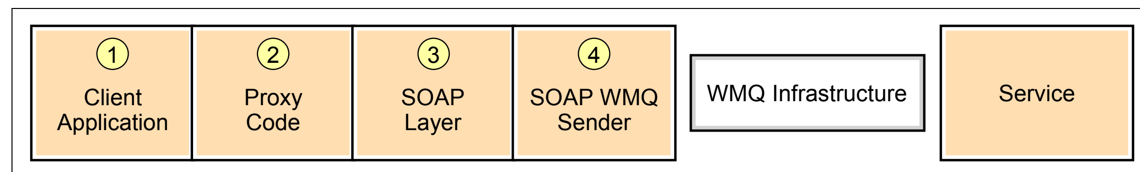


Figure 9-1 Client infrastructure

The procedure for a client implementation is as follows:

1. The client application code is implemented by the two Java files in Table 9-1. This code uses the proxy code to interface with the service.
2. The proxy code is generated. Obtain this in one of the following ways:
   - By using a utility to create the proxies from the Web Services Description Language (WSDL) of the service being used. In the case of Java, the WSDL2Java tool is used.

– As a result of the deployment of the service, the deployment tool supplied with WebSphere MQ transport for SOAP produces proxy code for Java, C#, and VisualBasic.NET (VB.NET). It uses the WSDL2Java tool to do this.

The client code directly calls the proxy code.

3. The SOAP layer. This is the Web Service host platform, in this chapter, Axis.
4. The SOAP /WebSphere MQ sender. This is a part of WebSphere MQ transport for SOAP.

This chapter begins with an illustration of a simple Axis client-service interaction, with the client and the service executing on the same machine. While this may not be a particularly realistic scenario, it focuses on the concepts involved. The subsequent sections demonstrate the following:

- Interpretability
- Distributed client and service
- More complicated WebSphere MQ implementations

For a detailed discussion of the service the client connects to, refer to Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269.

In the context of this chapter, it is sufficient to know that the service provides four simple banking functions. The method stubs are shown in Example 9-1.

Example 9-1 Service method stubs

```
public boolean credit (double amount)

public void debit(int account, double amount) throws
BankOperationException

public double getBalance()

public BankingOperation[] getStatement()
```

9.2 Requirements

In order to implement an Axis client, install the following software on the development machine:

- Java software development kit (SDK) V1.4.2, which is available in the WebSphere MQ CD
- WebSphere MQ V6
- Microsoft .NET Framework SDK V1.1

In addition, in this example, Rational Application Developer V6 is used. This is not essential, although it may be useful.

In order to be able to demonstrate the client, the BankingService requirements must also be implemented. To implement BankingService in .NET, Axis, or WebSphere Application Server, refer to Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269. To create the client, the developer requires a proxy code for the service. This is discussed in detail in 9.3.1, "Proxy code" on page 190.

> **Note:** Ensure that in the Features window of the WebSphere MQ, the **Install the Java Messaging and SOAP Transport** option is selected. For more details, see Chapter 7, "Environment setup" on page 141.

9.3 Implementation

This section explains the tasks involved in implementing the client. Discussions pertaining to the implementation of a GUI in Java is outside the scope of this book.

9.3.1 Proxy code

In order to create a client for any Web Service, the developer creating the client requires proxy code. This proxy code allows a client developer to call a remote method as though it is a local method. The proxy hides the complexity of the underlying Web Service infrastructure from the client. Proxy code is required for Web Services that use WebSphere MQ for SOAP as the transport mechanism.

To generate a proxy code from a WSDL file, use WSDL2Java, a tool supplied with Axis. To run this tool, issue the following command:

```
java com.ibm.mq.soap.util.RunWSDL2Java -o <output directory> <WSDL
filename>
```

In order to create proxy files for Web Service using WebSphere MQ, issue the following commands from within the same directory as the WSDL file:

```
amqwsetcp
java com.ibm.mq.soap.util.RunWSDL2Java
bankingService.BankingService_Wmq.wsdl
```

The result is two subdirectories containing the proxy files:

- bankingService, which contains the interfaces for the complex objects.
- bankingService_wmq, which contains the interface for the service and locator objects.

A discussion of the WSDL2Java tool is outside the scope of this book. However, note the following points:

- WSDL2Java is not invoked directly. Instead it is invoked by a wrapper class provided with WebSphere MQ, RunWSDL2Java. This is because the jms: prefix used in the URI must be registered.
- If WSDL2Java is called programatically, a call to Register.Extension must be made first.

A similar tool exists for Microsoft .NET. During the execution of the SOAP/WebSphere MQ deployment tool, both the WSDL tools are called to generate Java, C#, and VB.NET proxy code. This code is found in the client subdirectory of the generated folder created during the service deployment process.

Returning to the banking scenario, the six proxy files shown in Table 9-2 are generated.

Table 9-2 Proxy files generated

File name	Description
BankingService.java	Interface for the methods exposed by the banking service. These must match the stubs in Example 9-1
BankingServiceBankingServiceBinding.java	Implementation of the proxy code.
BankingServiceServiceLocator.java	Implementation of the service locator.

File name	Description
BankingServiceService.java	Interface to the service locator BankingServiceServiceLocator
BankOperation.java	Class describing the complex object returned by the getStatement method
BankOperationException.java	Class describing the custom exception that can be thrown by the debit method

9.3.2 A client for a local Axis service

To create a client for a local Axis service, perform the following tasks:

1. Create a project within the development environment of choice. In this example, an empty Java project called BankGUI is created in Rational Application Developer.
2. The proxy code is imported directly from the client folder created within the generated folder. The location of the proxy code is <deployment directory>\generated\client\remote\<namespace>\

   In this example, the full path is c:\temp\axisSvc\generated\client\remote\bankingService\

   > **Note:** For a more detailed discussion of the files generated during the deployment process, see 8.4, “Deployment” on page 164 for details.

3. Before importing the files, create a package with an appropriate name. In this example, a package called bankingService is created. Import all the Java files listed in Figure 9-2.

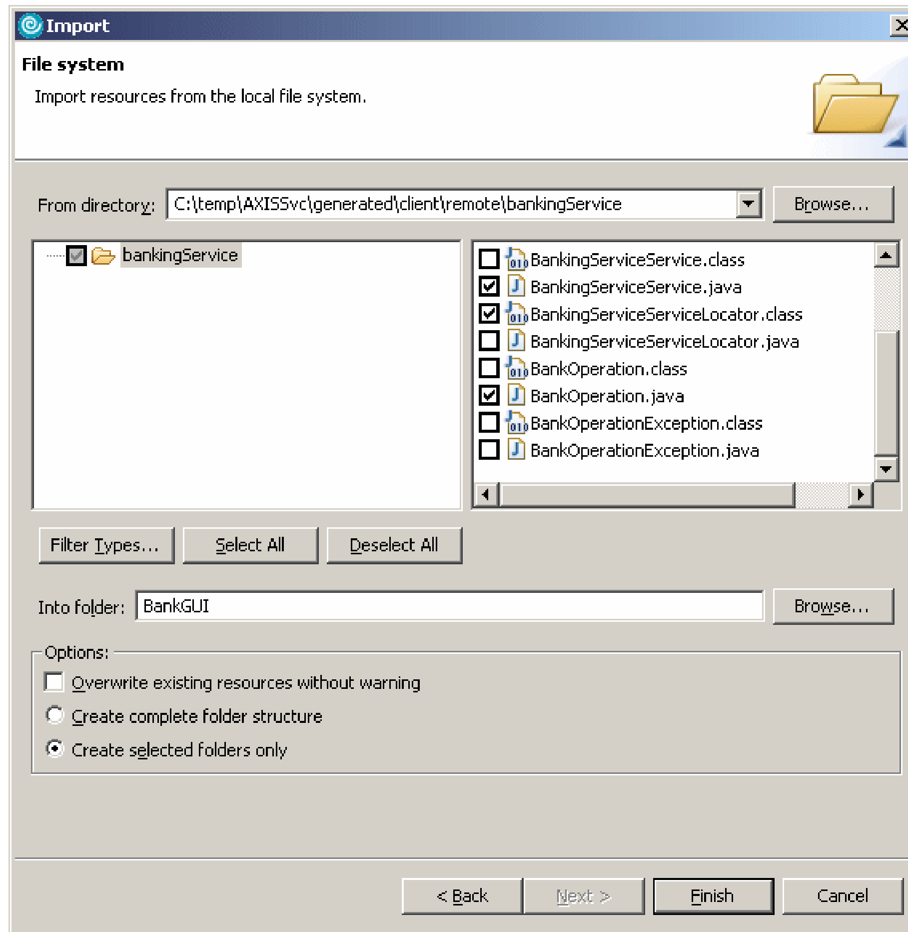


Figure 9-2 Importing proxy files

4. The development environment shows a number of errors due to missing external libraries. Add these libraries to the project. The libraries shown in Example 9-2 are required.

Table 9-3 External libraries required by proxy files

Library	Location
axis.jar	<WebSphere MQ install directory>\Java\lib\soap

Library	Location
commons-discovery.jar	<WebSphere MQ install directory>\Java\lib\soap
commons-logging.jar	<WebSphere MQ install directory>\Java\lib\soap
jaxrpc.jar	<WebSphere MQ install directory>\Java\lib\soap
saaj.jar	<WebSphere MQ install directory>\Java\lib\soap
servlet.jar	<WebSphere MQ install directory>\Java\lib\soap
wsdl4j.jar	<WebSphere MQ install directory>\Java\lib\soap
com.ibm.mq.jar	<WebSphere MQ install directory>\Java\lib
com.ibm.mq.soap.jar	<WebSphere MQ install directory>\Java\lib
commonservices.jar	<WebSphere MQ install directory>\Java\lib

Within Rational Application Developer, perform the following tasks:

a. Right-click the project and select **Properties**.

b. In the pop-up window, select **Java Build Path** as shown in Figure 9-3.

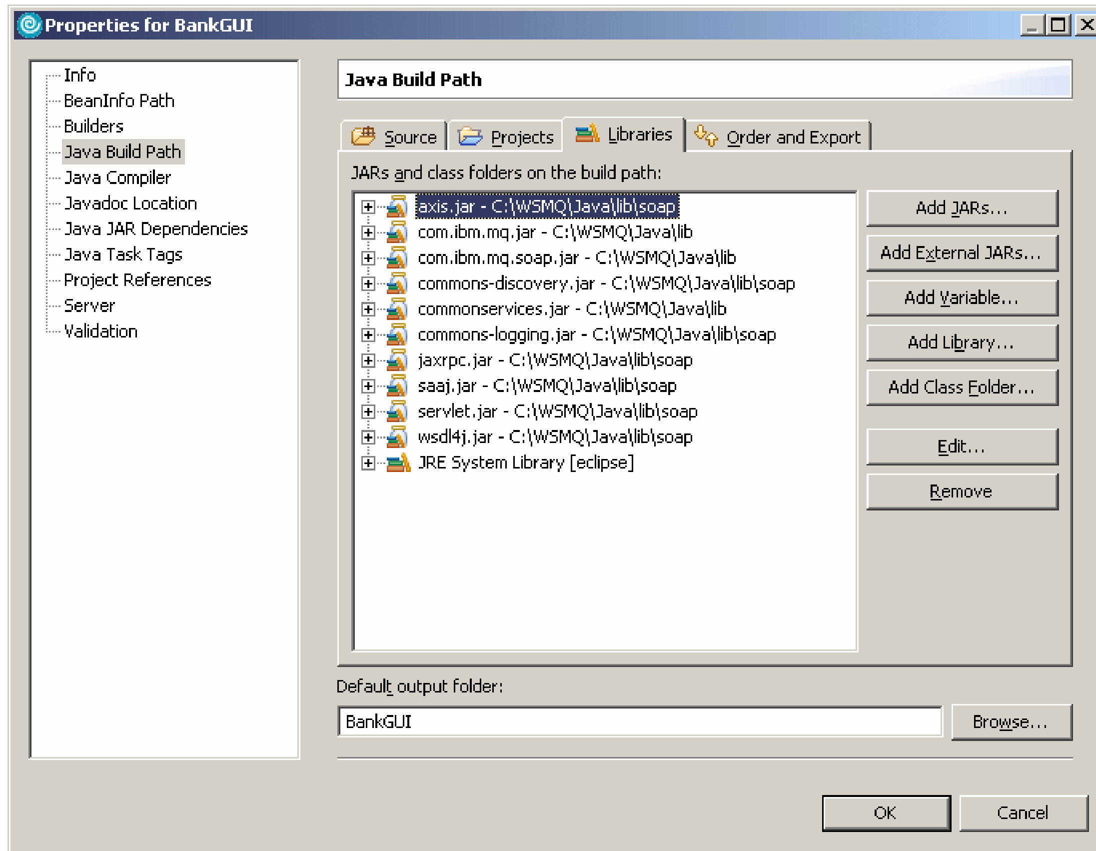


Figure 9-3 Adding additional libraries in Rational Application Developer

On completing this task, all the errors must be resolved.

c. Import the client code. The client code is implemented in a package called bankingClient. Create this package and import the code the same way the proxies are imported. This must result in both BankClient.java and BankingGUI.java being present in the bankingClient package.

In order to illustrate the implementation of a SOAP/WebSphere MQ client, the next two sections (“BankClient.java” on page 196 and “BankingGUI.java” on page 197) demonstrate client calls to the proxy code and client calls to the Web Service using the SOAP/WebSphere MQ infrastructure.

BankClient.java

The *Main* method begins with the following line:

```
com.ibm.mq.soap.Register.extension();
```

This line is responsible for registering the SOAP/WebSphere MQ sender with the underlying Web Services framework. In the example, it is the Axis infrastructure. This effectively informs Axis that an Universal Resource Indicator (URI) with the prefix jms: is valid and must be passed to the WebSphere MQ sender.

After registering the URI, instantiate a BankingService object using the code shown in Example 9-2. This code begins by creating a locator object. This object is part of the proxy. It talks to the service and ensures that the proxy finds the service in the network. Following are the two ways in which to call the get method:

- With no parameters

  This informs the proxy to use the URI generated at deploy time.

- With an Universal Resource Locator (URL) object

  This overrides the deploy time URI. In this example, the sample application allows an URI to be passed through the command line, although this is not required.

Example 9-2 Instantiating a BankingService object

```
static BankingService service = null;
...
BankingServiceServiceLocator locator = new
BankingServiceServiceLocator();


if( args.length == 0 )
{
   service = locator.getBankingServiceBankingService_Wmq();
}
else
{
   service = locator.getBankingServiceBankingService_Wmq(new
java.net.URL(args[0]));
}

...

BankingGUI gui = new BankingGUI(service);
gui.launchGUI();
```

After the BankingService object is created, it is passed into the BankingGUI class.

BankingGUI.java

The BankingGUI class receives an instance of a BankingService class through the constructor. With the communication being established, it simply calls methods on the BankingService class in the same way that it invokes methods on a local object, as shown in Example 9-3.

Example 9-3 Calling a BankingService method

```
if( e.getActionCommand().equals("deposit") )
{
   service.credit(Double.parseDouble(textField.getText()));
   balance.setText("£" + service.getBalance());
}
```

The final step is to provide a client configuration file for Axis. Download this source code (client-config.wsdd) from Appendix D, “Additional material” on page 431. The method that is recommended to create this file is to run the following script:

```
<WebSphere MQ install directory>\bin\amqwclientconfig.cmd
```

Run this script within the directory used by the deployment process for the Web Service, and then copy this into the classpath of the machine running the client.

> **Tip:** In this example, the client-config.wsdd file is copied into the directory containing the client source code. This is then added to the classpath because the client is launched from Rational Application Developer, as demonstrated here.

The client is ready for testing. Rational Application Developer environment users must perform the following tasks:

1. Select **Run...** from the Run menu.
2. Click the **New** button located in the bottom left of the Run window.
3. Change the name of the configuration to `BankGUI - Axis`.
4. Enter `bankingClient.BankClient`.

5. Click the **Arguments** tab and enter the following in the Program arguments box, as shown in Example 9-4:

Example 9-4 Commands to enter in the Program arguments box

```
"jms:/queue?destination=BankingServiceRequest@QM_localToSvc&connectionF
actory=()&initialContextFactory=com.ibm.mq.jms.Nojndi&targetService=sam
ple.axisSvc.BankingService.java&replyDestination=BankingServiceResponse
"
```

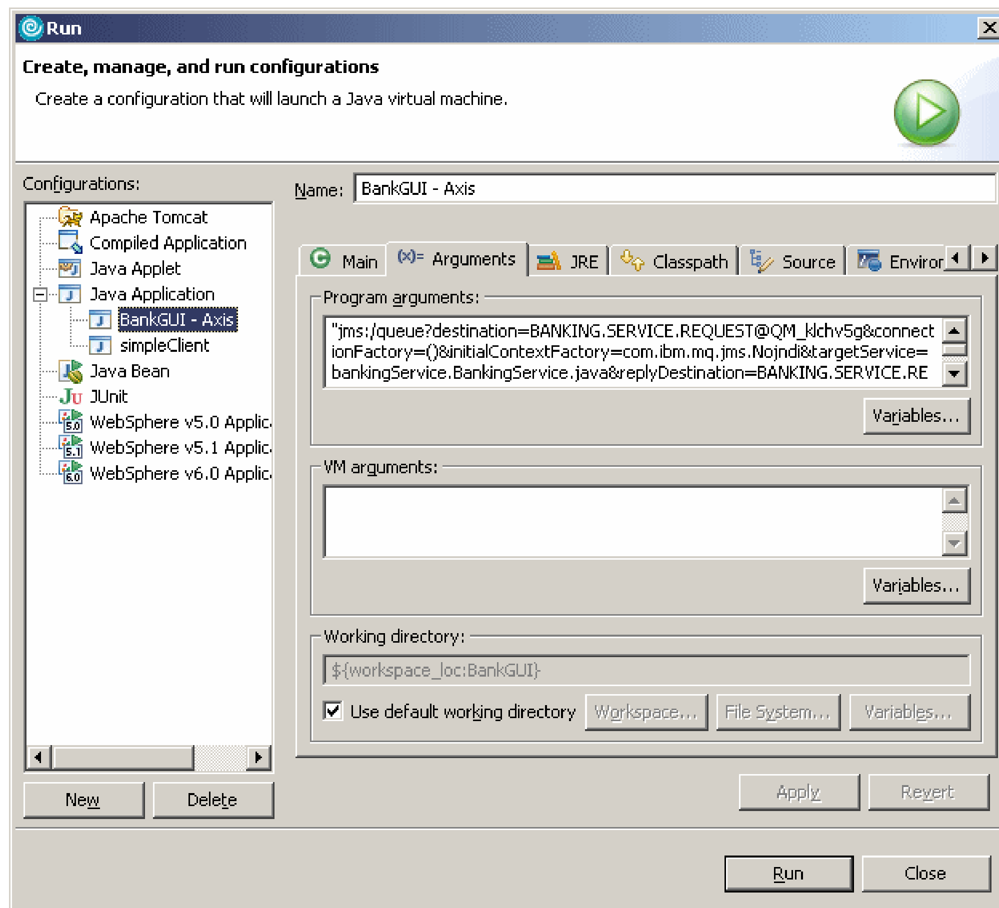


Figure 9-4 Configuring RAD to start the client

6. To use the client-config.wsdd, add the directory containing the client source code by performing the following tasks:

   a. Click the **Classpath** tab.

b. Click the **Advanced...** button.

c. In the window that opens, select the default option.

d. To add folders, click **OK**.

e. In the window that opens (Figure 9-5), expand **BankGUI** and select **bankingClient** and click **OK**.

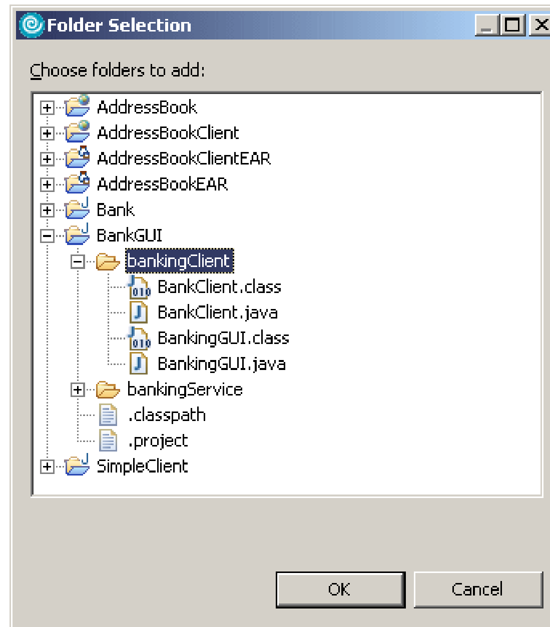


Figure 9-5 Adding a folder to the classpath in RAD

7. The client is ready to be executed. To do this, click the **Run** button. A window opens, as shown in Figure 9-6. Test the client using the following actions:

   – Enter an amount in the top left text box and click **deposit**. The balance in the top right label increases correspondingly.

– Enter an amount in the top left text box and click **transfer**. The balance in the top right label decreases accordingly. If an amount greater than the balance is entered, an exception is thrown.

– Click the **get statement** button to have the three most recent transactions shown in the bottom right.

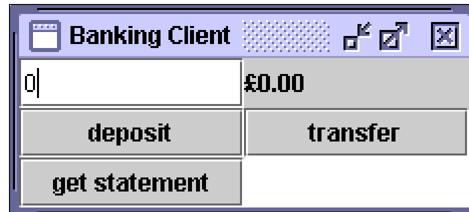


Figure 9-6 The Axis client running

> **Important:** For the client to work in the scenario described in this section, start the Web Service that it is invoking, on the client machine. The client must also have the following setup in WebSphere MQ:
>
> - Queue manager: QM_localSvc
> - Request queue: BANKING.SERVICE.REQUEST
> - Response queue: BANKING.SERVICE.RESPONSE

9.3.3 A client for a remote .NET service

This client is the same as in the previous section, except that it connects to a Web Service written in C# for the Microsoft .NET platform. The main difference is that a different set of proxies is used. These proxies are created during the deployment of the .NET service. Despite both the services exposing similar method stubs, the proxy code differs slightly. This is due to differences in Microsoft .NET and Axis implementation of Web Services.

> **Note:** For a description of these differences, refer to Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

Because this client uses a set of proxies that are different from the previous client, the client is created in a separate project. In this case, an empty Java project called BankGUI_msSvc is created with Rational Application Developer.

When the .NET Web Service is deployed, Java proxy code is created in a way that is similar to the way it is created during an Axis Web Service deployment. As discussed earlier, a folder called Generated is created within the deployment

folder. The proxy code is created in the subfolders of generated-client\remote. Within this folder, the Java proxy code can be found in a folder called dotNetService. This matches the package name for the proxy code, which is created within a package called dotNetService. When the proxies are generated using the default deployment utility:

- The package name is always dotNetService
- The folder structure is always generated\client\remote\dotNetService\

> **Note:** If you created a .NET Web Service in Chapter 10, ".NET Web Service" on page 213, these files are also created. If not, these are available as part of the source code for download from Appendix D, "Additional material" on page 431.

Import the proxy code files into the project. In Rational Application Developer, this is performed in the following manner:

1. Create a package called `dotNetService`.
2. Import the following files:
   - BankingService.java
   - BankingServiceLocator.java
   - BankingServiceSoap.java
   - BankingServiceSoapStub.java
   - BankOperation.java

> **Tip:** For more information about creating a package and importing files in Rational Application Developer, see section 9.3.2, "A client for a local Axis service" on page 192.

The first difference you may notice when comparing the proxy files against those in 9.3.2, "A client for a local Axis service" on page 192 is that there are only five proxy files compared to six in Axis service. The file that is missing is the BankOperationException definition.

As in step 4 on page 193 of 9.3.2, "A client for a local Axis service" on page 192, a number of errors are caused by the absence of the required class libraries from the build path. The libraries that are required are:

- axis.jar
- commons-discovery.jar
- commons-logging.jar
- axrpc.jar

– saaj.jar
– servlet.jar
– wsdl4j.jar
– com.ibm.mq.jar
– com.ibm.mq.soap.jar
– commonservices.jar

For details about the location of these libraries and a discussion about how to add them to a project in Rational Application Developer, see 9.3.2, “A client for a local Axis service” on page 192. Resolve all the errors.

3. Import the client code. The client code is implemented in a package called bankingClient. Create this package and import the code the same way the proxies were imported. The result must be that BankClient.java and BankingGUI.java are both present in the bankingClient package.

The next two sections (“BankClient.java” on page 202 and “BankingGUI.java” on page 204) illustrate the implementation of a SOAP/WebSphere MQ client for a .NET service. The implementation illustrates calls to the proxy code and the SOAP/WebSphere MQ infrastructure. These calls are similar to those described in “BankClient.java” on page 196 and “BankingGUI.java” on page 197. The only differences are in the names of some of the function calls because new proxy files are used.

BankClient.java

One of the differences is the import statement. The proxy code resides in a package with a different name:

```
import dotNetService.*;
```

The other difference is the reference to the interface to the Web Service class. This interface is called BankingServiceSoap instead of BankingService:

```
static BankingServiceSoap=null;
```

To instantiate the BankingServiceSoap object in order to make calls to the service methods, use the code shown in Example 9-5.

Example 9-5 Instantiating a proxy object

```
BankingServiceLocator locator = new BankingServiceLocator();

if( args.length == 0 )
{
   service = locator.getBankingServiceSoap();
}
```

```
else
{
   service = locator.getBankingServiceSoap(new java.net.URL(args[0]));
}
```

This code is similar to the code provided in Example 9-2 in “BankClient.java” on page 196, except for two differences:

```
import dotNetService.*;
```

The two differences are:

- The locator object is called BankingServiceLocator instead of BankingServiceServiceLocator, and the method to create an instance of the proxy object is called getBankingServiceSoap instead of getBankingServiceBankingService_wmq.
- The next difference is the reference to the interface to the Web Service class. This interface is called BankingServiceSoap rather than BankingService:

  ```
  static BankingServiceSoap=null;
  ```

To instantiate the BankingServiceSoap object in order to make calls to the service methods, use the code shown in Example 9-6.

Example 9-6 Instantiating a proxy object

```
BankingServiceLocator locator = new BankingServiceLocator();

if( args.length == 0 )
{
   service = locator.getBankingServiceSoap();
}
else
{
   service = locator.getBankingServiceSoap(new java.net.URL(args[0]));
}
```

Note: At this point, there is still an error in the BankClient class. This is due to a parameter mismatch with the BankingGUI class. The mismatch is fixed in the next section “BankingGUI.java” on page 204.

BankingGUI.java

Modify the import statement. As with BankingClient.java, the following import statement is required:

```
import dotNetService.*;
```

Again, as with BankClient.java, BankingGUI.java too has a reference to the interface class that defines the service:

- The parameter for the constructor
- The private reference used to store this object

Modify these from BankingService to BankingServiceSoap as shown in Example 9-7.

Example 9-7 Modifying the interface reference

```
private BankingServiceSoap service=null;
...
public BankingGUI(BankingServiceSoap service)
{
   service=service;
}
```

Start the client. As in the previous section, select **Run** from the Run menu. Ensure that the appropriate class is selected and set the URI as shown in Example 9-8.

Example 9-8 Starting the URI

```
"jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=(
connectQueueManager(WMQSOAP.DEMO.QM)binding(client)clientChannel(SYSTEM
.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))&initialContext
Factory=com.ibm.mq.jms.Nojndi&targetService=BankingService.asmx&replyDe
stination=SYSTEM.SOAP.RESPONSE.QUEUE"
```

> **Tip:** It is useful to wrap the URI within quotes when invoking the deployment tool from the command line in order to avoid syntax errors.

The graphical user interface is shown in Figure 9-6 on page 200 and is started in exactly the same manner. The only difference between connecting to the Axis service and to the .NET service is that a different proxy code is used. This means that class names differ slightly.

9.3.4 The WebSphere MQ environment

The following section describes two possible distributed WebSphere MQ configurations.

Connecting by using client binding

The earlier examples illustrated how to connect to services based on different host environments. Ultimately, the only difference is that a different proxy code is used. The client connects to the WebSphere MQ environment whether a client and service are on the same machine or are on different machines.

The client connects to the WebSphere MQ environment when a client and service are on the same machine or are on different machines.

Figure 9-7 shows the client connected by using a client channel.

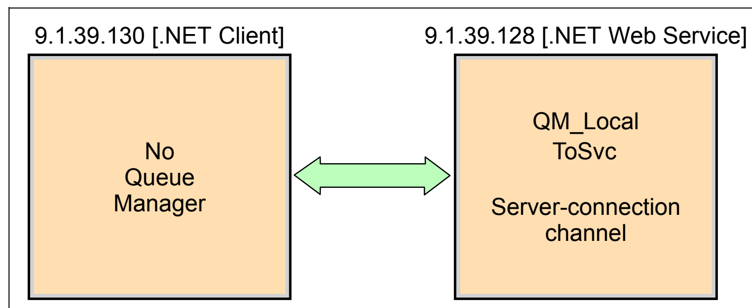


Figure 9-7 Connecting through a client channel

The client machine requires a channel to be created on the queue manager hosting the service. The client connects to this channel through the TCP/IP, using the specified settings. The following URI illustrates this:

```
binding(client)clientChannel(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39
.128%25281414%2529)
```

Note: %2528 is the code for the “(” character and %2529 is the code for the “)” character.

This URI can be broken down as follows:

- binding(client)

  Mandates a client connection to the queue manager. In other words, only WebSphere MQ client is required on the client machine.

- clientChannel(SYSTEM.DEF.SVRCONN)

  The name of the client channel to use.
- clientConnection(9.1.39.128%25281414%2529)

  The client connection location, either an IP address or a host name.

Queue manager-to-queue manager connections

A more complicated setup involves having the client application connected to a queue manager on one machine and the service connected to a queue manager on another machine. This is illustrated in Figure 9-8.

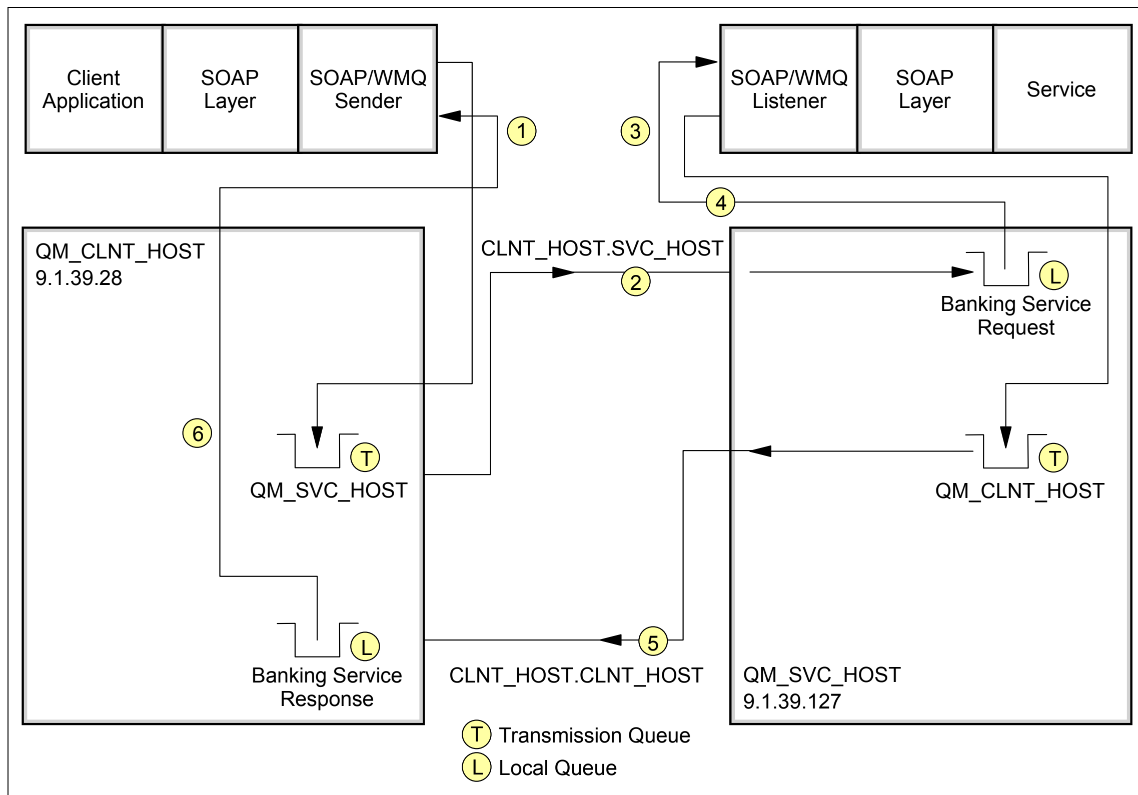


Figure 9-8 Queue manager-to-queue manager connection

This figure illustrates the queue manager-to-queue manager setup. The message flow within this setup is as follows:

1. The client sends a request that gets put on a transmission queue, as illustrated by 1.
2. The message is then sent across a channel, as illustrated by 2, to the service queue manager.
3. The message is picked up from the request queue by the SOAP/WebSphere MQ listener as illustrated by 3.
4. The request is processed and the response placed on the other transmission queue as illustrated by 4.
5. From here, the request is sent across a channel, as illustrated by 5, back to the client queue manager. It is then picked up by the SOAP/WebSphere MQ sender, as illustrated by 6.

The service chapters (Chapter 8, “Axis Web Service” on page 159, Chapter 10, “.NET Web Service” on page 213, and Chapter 12, “WebSphere Application Server Web Service” on page 269) illustrate how to implement the service part of this topology. Refer to Chapter 7, “Environment setup” on page 141 for details about the capabilities of WebSphere MQ transport for SOAP.

After the client queue manager is configured appropriately, the service can be called using the appropriate URI, as shown in Example 9-9.

Example 9-9 Calling the Web Service

```
jms:/queue?destination=SOAPN.demos@QM_SVC_HOST&connectionFactory=(conne
ctQueueManager(QM_CLNT_HOST))&initialContextFactory=com.ibm.mq.jms.Nojn
di&targetService=BankingService.BankingService.java&replyDestination=SY
STEM.SOAP.RESPONSE.QUEUE
```

Following are the important points to note about this URI:

- The destination parameter is queue@destinationQueueManager. In this case, the destination queue manager is QM_SVC_HOST. It is this parameter that tells the underlying infrastructure where to send the request.
- The connectQueueManager parameter specifies the queue manager that the client connects to, in this case, QM_CLNT_HOST.

The rest of the URI is the same as that seen earlier in the chapter.

9.4 Error handling

This section details some of the common errors you may encounter when running a client that calls a Web Service using WebSphere MQ as a transport mechanism.

9.4.1 Unable to put a request to queue

In this error, the client is unable to put a request to the queue. In the test environment, WebSphere MQ Explorer is used to inhibit the put operation. This generates an exception on the client, the first few lines of which look as shown in Example 9-10.

Example 9-10 Put inhibited for request queue

```
MQJE001: Completion Code 2, Reason 2051
AxisFault
 faultCode:
{http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultSubcode:
 faultString: (2) (2051); nested exception is:
   com.ibm.mq.MQException: MQJE001: Completion Code 2, Reason 2051
```

A utility called mqrc is shipped with WebSphere MQ to provide brief descriptions for these error codes. The syntax for mqrc is:

```
mqrc <error code>
```

The output for a 2051 code of is:

```
2051 0x00000803 MQRC_PUT_INHIBITED
```

9.4.2 Specified request queue does not exist

In this error, the client is started without the specified request queue existing. In the test environment, the client is started with an URI, including a request queue name that does not exist. When attempting to invoke the service, the first few lines of the output is similar to that shown in Example 9-11.

Example 9-11 Request queue does not exist

```
MQJE001: Completion Code 2, Reason 2085
AxisFault
 faultCode:
{http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
```

```
 faultSubcode:
 faultString: (2) (2085); nested exception is:
   com.ibm.mq.MQException: MQJE001: Completion Code 2, Reason 2085
```

Using mqrc again, the output is:

```
2051 0x00000825 MQRC_UNKNOWN_OBJECT_NAME
```

9.4.3 Response not received

Another possible error is where the client does not receive a response. In the test environment, the client is started, but not the listener. The result of this is that the client gives the impression of hanging while waiting for a response. Eventually, the client times out and an exception message is shown. The first few lines of the output is similar to that shown in Example 9-12.

Example 9-12 Timeout exception

```
MQJE001: Completion Code 2, Reason 2033
AxisFault
 faultCode:
{http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultSubcode:
 faultString: (2) (2033); nested exception is:
   com.ibm.mq.MQException: MQJE001: Completion Code 2, Reason 2033
```

The output of mqrc this time is:

```
2033 0x00000833 MQRC_NO_MSG_AVAILABLE
```

9.4.4 Cannot find the client-config.wsdd file

When invoking a service from the client, you may encounter a message similar to that shown in Example 9-13. This error occurs when the client cannot find the client-config.wsdd file. This file must be within the classpath of the client.

Example 9-13 No Java Message Service transport

```
AxisFault
 faultCode:
{http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultSubcode:
 faultString: No client transport named &apos;jms&apos; found!
 faultActor:
```

Note: This file is generated when the service is deployed and is required by the Axis infrastructure. It informs Axis that the prefix jms: is valid.

9.4.5 Incorrect message format

Another exception message that you may encounter begins as shown in Example 9-14. This error occurs when the client encounters a message in an incorrect format. This is often caused by a report message being returned by the service. To test this, a simple text file message is placed in the response queue while the client is executing.

Example 9-14 Invalid message exception

```
AxisFault
 faultCode:
{http://schemas.xmlsoap.org/soap/envelope/}Server.generalException
 faultSubcode:
 faultString: Unexpected message type received. MQCC_FAILED(2)
MQRC_MSG_TYPE_ERROR(2029).
 faultActor:
 faultNode:
 faultDetail:
```

9.5 Security

The security configuration used by the Web Services client is prescribed by the URI in the proxies in the form of Secure Sockets Layer (SSL) key words. Following are the key words that are of relevance to an Axis Web Services client:

- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword
- sslCipherSuite

When an Axis client connects to a WebSphere MQ queue manager using a client connection through a SVRCONN, the specific values for these key words are used to determine the following:

- The location of the user certificate for the client
- The password to access that certificate

- The location of the trust store containing the Certification Authority (CA) certificates
- The password to access the certificates in the trust store
- The cipherSuite to be used during communication in order to secure the data

The values of each of these key words are picked up from the URI. For the SSL to become enabled, the key store and trust store must exist and contain the correct certificates. For more details about the configuration required to set up SSL, refer to Chapter 6, “Security” on page 107.

9.6 Summary

This chapter discussed the creation of clients for multiple Web Services. All the Web Services have a common transport mechanism, WebSphere MQ. However, the Web Services are implemented on multiple platforms using multiple development environments. This chapter discussed how calls can be made to Web Services over SOAP/WebSphere MQ. The code may differ across platforms. Using this chapter in conjunction with one of the Web Service chapters (Chapter 8, “Axis Web Service” on page 159, Chapter 10, “.NET Web Service” on page 213, and Chapter 12, “WebSphere Application Server Web Service” on page 269) enables the creation of a client to which

allows a reader to create a client and a service on different platforms with configurations of differing complexity.

After demonstrating how to create the client, this chapter discussed some of the common errors that clients may encounter. Error handling concepts are discussed in detail in 4.9.4, “Security and error handling” on page 46.

The knowledge gained while developing clients and services is built on in the subsequent chapters, as more advanced concepts, including invoking methods asynchronously and using transactions on the client side, are discussed.

For more information about writing Web Services using WebSphere MQ as a transport mechanism, refer to Chapter 10, “.NET Web Service” on page 213, Chapter 8, “Axis Web Service” on page 159, and Chapter 12, “WebSphere Application Server Web Service” on page 269.

10

.NET Web Service

This chapter demonstrates the creation and deployment of a .NET Web Service that sends its SOAP messages over WebSphere MQ instead of Hypertext Transfer Protocol (HTTP). Web Services are based on a request-and-response model using messages encoded in SOAP, which is a messaging protocol designed to be network-neutral, transport-neutral, and programming language-neutral. These messages are formatted with Extensible Markup Language (XML). Typically, SOAP messages are sent through a HTTP, the underlying protocol used by the World Wide Web. The SOAP protocol is transport-independent. Therefore, WebSphere MQ is used as an alternative transport mechanism. A .NET service that is already prepared as a HTTP Web Service does not have to be modified further to use WebSphere MQ transport for SOAP. It requires redeployment through a SOAP/WebSphere MQ deployment process.

This chapter covers the following topics:

- Creation of a .NET Web Service using Visual Studio .NET (2003)
- Environment setup for deployment of .NET Web Service
- Deployment of .NET Web Service

© Copyright IBM Corp. 2006. All rights reserved.

- Security considerations and enablement
- Error handling
- Creation of a .NET Web Service using any text editor and command-line tools shipped with the Microsoft .NET Framework software development kit (SDK) (for developers without Visual Studio .NET)

10.1 Design

This section discusses the design of a simple .NET Web Service that is used in demonstrating WebSphere MQ transport for SOAP.

In this example, a BankingService Web Service is designed to model a bank account and the common operations that take place on it. The Web Service design is kept simple. However, the design takes returning simple data types and complex data types, and exceptions into consideration. The operations are listed in Table 10-1.

Table 10-1 BankingService method description

Method	Description
debit	Removes specified amount for transfer to the account ID provided. This implementation simply subtracts the amount specified from the balance. If the amount is greater than the balance, an exception is thrown.
credit	Adds specified amount to current balance
getBalance	Returns the current balance
getStatement	Returns an array of BankOperation objects

Following are important information pertaining to the methods:

- getStatement returns a complex, user-defined type.
- debit throws a BankOperationException.
- BankOperation is a serializable class that contains details of the operations.

Note: Static variables are used in order to preserve values from different operations.

The infrastructure for implementing this Web Service is different from the standard Web Service because the transport mechanism must be altered. WebSphere MQ is used instead of the typical HTTP method. This is illustrated in Figure 10-1.

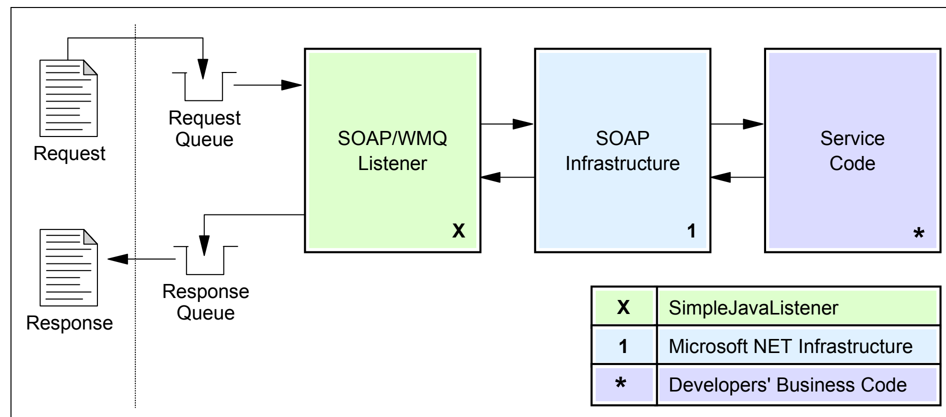


Figure 10-1 SOAP WebSphere MQ infrastructure on the service side

Figure 10-1 illustrates the main components on the service side of a WebSphere MQ Web Service. The service code that is indicated by an asterisk (*) in Figure 10-1 is the code containing the business functionality. In this example, this is the BankingService class. The component in the middle that is indicated with 1 in Figure 10-1, is that part of the infrastructure that handles the interaction with SOAP, in this example, Microsoft .NET.

These two components exist in the standard HTTP Web Service infrastructure. The additional component, the SOAP/WebSphere MQ listener that is indicated by an X in Figure 10-1, is the new piece of the infrastructure. This listener is responsible for interfacing with WebSphere MQ in order to perform the following functions:

- Read request messages from the request queue
- Write response messages to the response queue

For a complete overview of the infrastructure, including client and service, see 4.2, “SOAP over WebSphere MQ” on page 31.

10.2 Requirements

To implement the .NET Web Service, the following are required:

- Windows 2000 Service Pack 2 (SP2) or earlier
- WebSphere MQ V6
- Microsoft .NET Framework 1.0 SP1 or earlier
- Microsoft .NET Framework SDK and any text editor
- Visual Studio .NET 2003 (optional)

> **Note:** It is essential to select the **Java Messaging and SOAP Transport** option from the Features to install menu when installing WebSphere MQ. See 7.2.1, “Installing IBM WebSphere MQ V6” on page 142.

10.3 Implementation

This section discusses the implementation and compilation of the BankingService Web Service used in this chapter. Download the BankingService Web Service code from Appendix D, “Additional material” on page 431.

This section also discusses the WebSphere MQ environment setup that is required before deploying the Web Service.

10.3.1 Implementation of the Web Service

The Web Service code is implemented in C#. The details of the credit and debit operations on the account are stored in a BankOperation object. The BankOperation object is made serializable so that instances of the same can be saved to Extensible Markup Language (XML), which allows persistence and cross-platform distribution. Example 10-1 shows the serializable BankOperation object stub.

Example 10-1 The BankOperation object

```
[Serializable]
public class BankOperation
{
   //private members
   //include getters and setters or make members public
}
```

This section discusses the methods contained within the Web Service.

The credit method

The credit method returns a boolean, indicating whether or not the credit operation is successful. It also creates a BankOperation object from the details of the credit operation. The BankOperation object is stored in an array, which is later returned as a bank statement. The code snippet in Example 10-2 shows the stub of the credit method.

Example 10-2 The credit method code stub

```
[WebMethod] [SoapRpcMethod]
public bool credit(double amount)
{
   //credit the account
   //Create the BankOperation object
   //Store the BankOperation object in an array
   return true;
}
```

The debit method

This void method subtracts the specified amount from the account balance. As in the case of the credit method, it also creates a BankOperation object from the details of the debit operation. The BankOperation object is then stored in an array, which is later returned as a bank statement.

To demonstrate exception handling, the debit method throws a BankOperationException when the amount specified for debit is greater than the account balance. The code snippet in Example 10-3 shows the stub of the debit method.

Example 10-3 The debit method code stub

```
[WebMethod] [SoapRpcMethod]
public void debit(int account, double amount)
{
      //Check if amount does not exceed bank balance
      //debit the account
      //Create the BankOperation object
      //Store the BankOperation object in an array
      //Otherwise, throw a BankOperation exception
}
```

The getBalance method

This method returns a double value, indicating the account balance. The code snippet in Example 10-4 shows the stub of the getBalance method.

Example 10-4 The getBalance method code stub

```
[WebMethod] [SoapRpcMethod]
public double getBalance()
{}
```

The getStatement method

This method returns an array of the last three BankOperation objects created. This array serves as the bank statement, detailing the last three transactions. The code snippet in Example 10-5 shows the stub of the getStatement method.

Example 10-5 The getStatement method code stub

```
[WebMethod] [SoapRpcMethod]
public BankOperation[] getStatement()
{
}
```

The Web methods are explicitly declared with the SoapRpcMethodAttribute to specify that the SOAP messages to and from the Web Service use Remote Procedure Call (RPC) formatting. Although any other type of encoding can be used, RPC is selected in this example. This choice maintains the consistency of message formatting and ensures interoperability between the Web Services and the Web Service clients. This is discussed in other chapters within this book. For more information about SOAP encoding and interoperability, see 4.4.2, “Interoperability” on page 37.

The BankOperationException

When a debit operation is requested on an account, if the debit amount is greater than the account balance, a BankOperationException derived from System.Exception is thrown.

The BankingService Web Service discussed earlier is prepared the same way in which a HTTP Web Service is prepared in .NET. At this stage, the Web Service is ready for compilation and deployment through the SOAP/WebSphere MQ deployment process.

In situations where the service code has not been previously prepared as a Web Service, it must be modified for it to be declared a Web Service and to identify how each of its method's parameters are formatted. The code snippet in Example 10-6 shows a .NET class that has been prepared as a Web Service. The bold type text indicates the additions to the code that make it a Web Service that uses RPC encoding.

Example 10-6 Implementing a .NET Web Service

```
<%@ WebService Language="C#" Class="BankingService' %>

using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Web.Services.Description;
using System.Threading;

[WebService (Namespace="http://dotnet.BankService")]
[SoapRpcService]
public class BankingService {

private static double account_balance = 0 ;

   [WebMethod] [SoapRpcMethod]
   public bool credit(double amount)
   {
      account_balance += amount;

      return true;
   }

   [WebMethod] [SoapRpcMethod]
   public double getBalance()
   {
      return account_balance;
   }
}
```

Tip: To convert application code to a Web Service, add `[WebMethod]` above every method you want to expose in the Web Service and include the Web Service directive in the first line of the code as shown in the code snippet in Example 10-6.

To convert the Web Service to an RPC Web Service, add `[SoapRpcService]` above the class definition and `[SoapRpcMethod]` above all the Web method definitions.

The code is written and it must now be saved in the appropriate Web Service code files.

Note: When preparing Web Service code for use as a SOAP/WebSphere MQ service, if the service uses classes that are external to the .NET infrastructure and the SOAP/WebSphere MQ runtime environment, the Web Service source code must be written and built as noninline. This means that the source for the service is built using codebehind.

10.3.2 Compiling the Web Service

Compile .NET Web Services in Visual Studio .NET or by using the command-line tools shipped with the Microsoft .NET Framework SDK.

To compile Web Service code in Visual Studio .NET, right-click the solution file and select the **build** option.

If Visual Studio .NET is *not* used to write and compile the Web Service code, use a text editor to write the code. Use the command-line tools shipped with the Microsoft .NET Framework SDK to compile it.

Note: For commands within the Microsoft .NET Framework compiler to work from any folder, include the location of the Microsoft .NET Framework, typically, root\WINNT\Microsoft.NET\Framework\version, in your computer's PATH environment variable. To do this:

- Right-click **My Computer** and select **Properties**.
- Click **Environment Variables** in the Advanced Tab.
- Add the path to the Microsoft .NET Framework to the path variable.

Use the **csc** command to compile the code. To compile the BankingService Web Service, open the Windows command prompt and change the current directory to the location of the directory where the BankingService.asmx is. Create a folder called `bin` and issue the **csc** command as shown in Example 10-7.

Example 10-7 Issuing the csc command

```
csc /lib:c:\progra~1\ibm\websph~1\bin /r:System.dll /r:System.Data.dll
/r:System.Web.Services.dll /target:library /out:bin\BankingService.dll
BankingService.asmx.cs BankOperation.cs
```

For more information about using the Microsoft .NET Framework SDK to compile .NET Web Services, consult the Microsoft .NET documentation.

From the files that are manually created and from those files the compilation auto-generated, only the following files are required for the deployment of the Web Service code for use as a SOAP/WebSphere MQ Web Service:

- The bin directory containing the compiled output of the project, the dynamic link library files (.dll files).
- The .asmx file and the .asmx.cs file that constitute the Web Service. The .asmx file contains the Web Service processing directive, and serves as the entry point for the Web Service, while the asmx.cs class file contains the code behind the class for the Web Service that is used to separate the Web Service directive from the source code. These are created while writing the Web Service code.
- Any .cs files that are also created when writing the Web Service code. In this example, the BankOperation object code is stored in a separate BankOperation.cs file.

10.4 Preparing the WebSphere MQ environment

The WebSphere MQ environment for deploying SOAP/WebSphere MQ Web Services must have a queue manager setup and a response queue setup. The request queue is optional. If not specified otherwise, a request queue is created during deployment. This section discusses the setup required for the deployment of Web Service for access from local and remote Web Services clients.

Create a queue manager

You may either choose to specify a queue manager during the deployment process or decide against it.

If you choose against specifying a queue manager during the deployment process, the default queue manager within your WebSphere MQ environment must be set up. To create a default queue manager, run the setupWMQSOAP.cmd script file that is provided in WebSphere MQ home directory\Tools\soap\samples.

To specify a name for the default queue manager, run the script by typing setupWMQSOAP <*name of queue manager*> as shown in Figure 10-2.

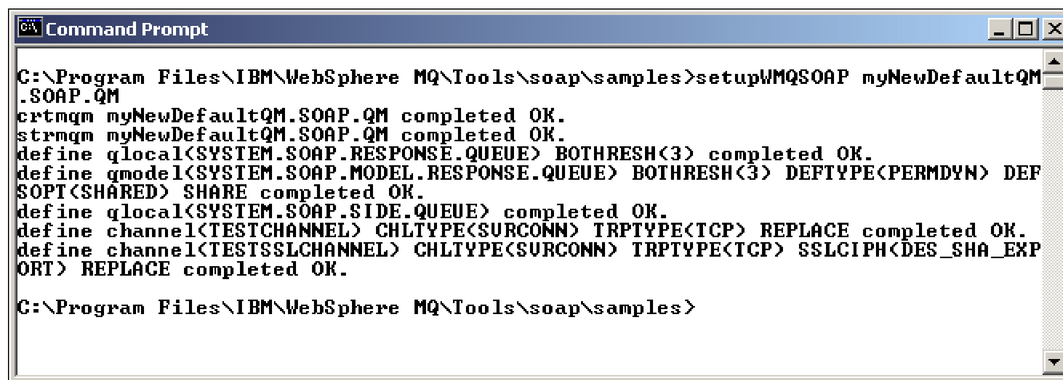


Figure 10-2 Creating a queue manager

If you *do* choose to specify a queue manager during the deployment process, which is a more common practice, set up a queue manager within the WebSphere MQ environment. See 7.3.1, "Basic WebSphere MQ administration" on page 151 for instructions about how to create a queue manager.

Create a response queue

You may either choose to specify a response queue during the deployment process or decide against it.

If you choose to *not* specify a response queue during deployment, the deployment process does not automatically create a response queue for the Web Service. It assumes SYSTEM.SOAP.RESPONSE.QUEUE as the default response queue. If you run the setupWMQSOAP script provided in WebSphere MQ home directory\Tools\soap\samples, this default response queue is created. Otherwise, run the setupWMQSOAP.cmd script as instructed in "Create a queue manager" on page 223.

> **Tip:** SYSTEM.SOAP.RESPONSE.QUEUE is a system queue. In order to view this queue in WebSphere MQ Explorer, select the **Show System Objects** option.

If you *do* choose to specify a response queue, set up a local queue that serves as the response queue. See 7.3.1, “Basic WebSphere MQ administration” on page 151 for instructions about how to create a queue.

Create a request queue

You may either choose to specify a request queue during the deployment process or decide against it.

If you choose against specifying a request queue during deployment, the deployment process automatically creates a request queue for the Web Service called SOAPN.Web Service.

If you *do* choose to specify a response queue, set up a local queue that serves as the request queue. See 7.3.1, “Basic WebSphere MQ administration” on page 151 for instructions on how to create a queue.

Setup for client mode and server binding mode connections

In order to be invoked by a remote Web Service client, a Web Service must specify how the client connects to it. Depending on the client’s WebSphere MQ environment, one of the two connections is required:

- Scenario 1

  A client resides on a machine that serves as a WebSphere MQ client without a queue manager being specified. The Web Service deployment must specify a client connection to the SOAP/WebSphere MQ Web Service queue manager. Figure 10-3 represents this.

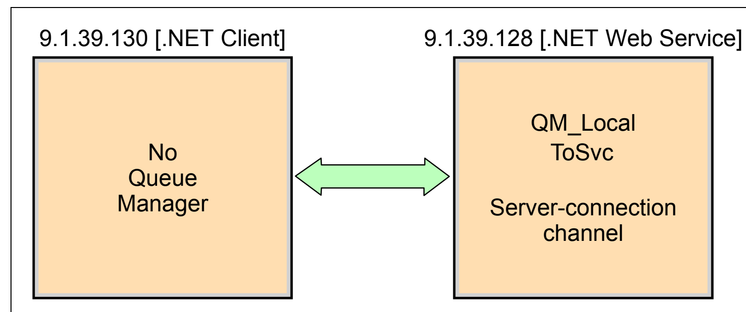


Figure 10-3 Environment setup for a client mode connection

For this scenario, the following are required:

– A WebSphere MQ queue manager in the Web Service's WebSphere MQ environment, in this case, QM_LocalToSvc
– clientChannel, which is a server connection channel set up on the WebSphere MQ queue manager in the Web Service's WebSphere MQ environment. The command to create the channel is shown in Example 10-8.

Example 10-8 Command to create channel

```
DEFINE CHANNEL (BANKING.SVR.CHL) CHLTYPE(SVRCONN) TRPTYPE(TCP)
MCAUSER('demoUser') DESCR('Server connection channel for
BankingService') replace
```

– The host machine name or IP address, which the WebSphere MQ queue manager used by the Web Service resides on.
– The TCP/IP port on which the queue manager is listening. The default configuration of the queue manager typically sets this value to 1414.
– The WebSphere MQ listener setup in the Web Services WebSphere MQ environment.

> **Note:** For demonstration purposes, for the client connection to be successful, the server connection channel MCAUSER is set to a userID that is in the mqm user group. This is purely for purposes of simplicity, and is *not* recommended in a production environment. The security implications of this must be considered.

For details about a Web Service deployment that is similar to that described here, refer to 10.5.4, "Executing a deployment to a remote queue manager" on page 232.

- Scenario 2

  A client resides on a machine that serves as a WebSphere MQ server with its own WebSphere MQ environment, with a queue manager defined. The Web Service deployment must specify a server binding to the SOAP/WebSphere MQ Web Service queue manager. The setup requires the following:

  – A WebSphere MQ queue manager in the Web Service's WebSphere MQ environment, in this case, QM_SCV_HOST
  – A WebSphere MQ queue manager in the Web Service client's WebSphere MQ environment, in this case, QM_CLNT_HOST
  – A sender channel in the Web Service's WebSphere MQ environment to send response to the client, in this case, SVC_HOST.CLNT_HOST

– A receiver channel in the Web Service's WebSphere MQ environment to receive requests from the client, in this case, CLNT_HOST.SVC_HOST
– A sender channel in the Web Service client's WebSphere MQ environment to send requests to the Web Service, in this case, CLNT_HOST.SVC_HOST
– A reciever channel in the Web Service client's WebSphere MQ environment to receive response from the Web Service, in this case, SVC_HOST.CLNT_HOST
– A transmission queue in the Web Service's WebSphere MQ environment with the same name as the remote queue manager name, in this case, QM_CLNT_HOST
– A transmission queue in the Web Service client's WebSphere MQ environment with the same name as the remote queue manager name in order to help send requests to Web Service, in this case, QM_SVC_HOST
– A local queue in the Web Service's WebSphere MQ environment for servicing requests to the Web Service, in this case, BANKING.SERVICE.REQUEST
– A local queue in the Web Service client's WebSphere MQ environment for receiving response from the Web Service, in this case, BANKING.SERVICE.RESPONSE

Figure 10-4 shows the environment setup.

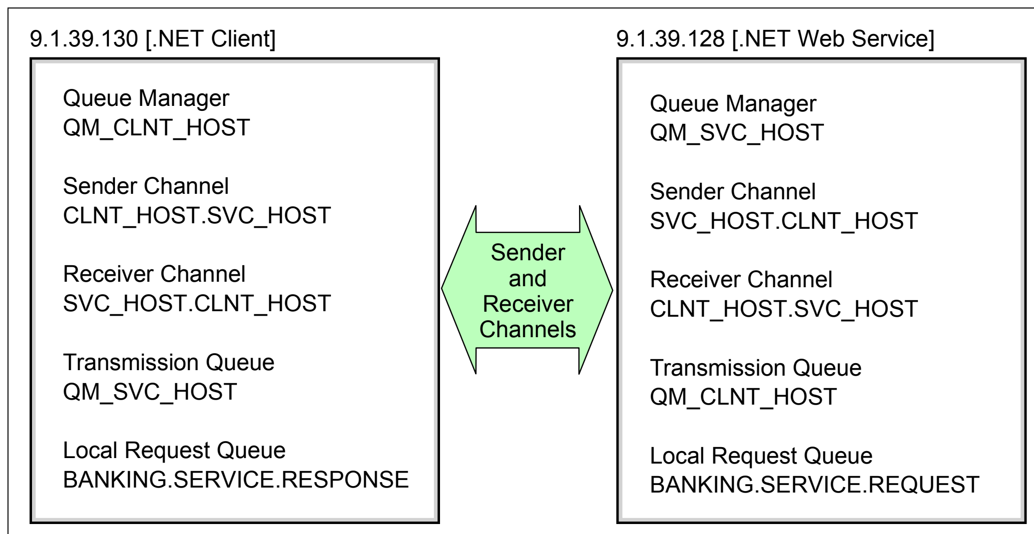


Figure 10-4 Environment setup for a binding mode connection

The scripts to configure WebSphere MQ are included with the source code download. To configure a service queue manager called QM_SVC_HOST with the downloaded script file, refer to 7.3.1, “Basic WebSphere MQ administration” on page 151.

For details about the deployment of a Web Service similar to that described here, refer to 10.5.4, “Executing a deployment to a remote queue manager” on page 232.

10.5 Deployment

Web Services produce interface contracts that are described in a standard XML document called Web Services Description Language (WSDL), which is used to generate a proxy. This proxy acts as an intermediary between the client and the Web Service. It hides the complexity of the Web Service and forwards calls from the client to it.

Web Services must be processed through a series of deployment steps for them to send their SOAP messages over WebSphere MQ. These steps define the Web Service to the host infrastructure, in this case, .NET.

A deployment utility is provided as part of WebSphere MQ transport for SOAP. The deployment utility consists of a Java program, com.ibm.mq.soap.util.amqwdeployWMQService. It also consists of a command file, amqwdeployWMQService.cmd, which invokes the amqwdeployWMQService Java program. For more information about the deployment utility and its customization, see Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

The deployment process generates WSDL and proxy methods for invoking the service from the client, prepares a script file to start a service listener, and performs queue and process configuration within WebSphere MQ.

This chapter starts off by describing the initial steps to take during the deployment process. It then demonstrates a simple deployment process making use of a local queue manager, and goes on to demonstrate more complex deployments on remote queue managers.

10.5.1 Common deployment steps

The common tasks in any deployment process can be split into the following steps:

1. Moving the relevant files to a separate directory

   Although this task is not always essential, it is advisable to move the relevant files for the deployment process, that is, .asmx, .asmx.cs, .cs files, and bin directory into a separate folder. The deployment process generates a few files in this location. Therefore, the auto-generated files are stored in the same location in order to avoid confusion. In our case, the folder created is \REDBOOK\dotNETService\BankService.

2. Setting the Java class path

   Setting the class path is essential for running any Java application. Because the deployment utility consists of a Java program, the location of the Java classes necessary for it to run must be specified. A script, amqwsetcp, is provided with WebSphere MQ in order to set the classpath. The script is located in WebSphereMQ Installation directory\bin. Figure 10-5 shows the results of running the script.

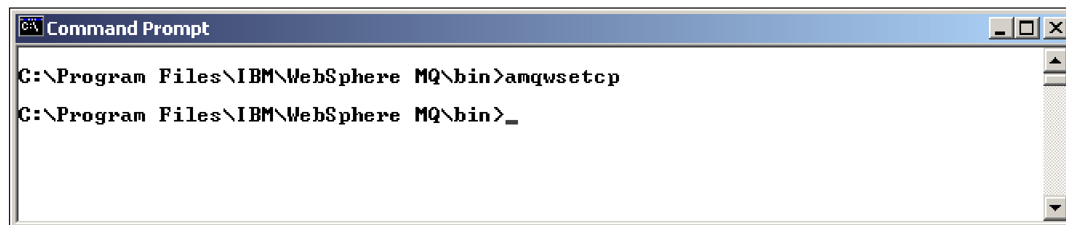


Figure 10-5 Output when the amqwsetcp script is run

> **Note:** If errors are generated when you run the amqwsetcp script, ensure that the axis.jar file is copied into the WebSphere MQ home directory\Java\lib\soap directory, the Java runtime location is included in the path, and the WMQSOAP_HOME path is defined. See 7.2.1, "Installing IBM WebSphere MQ V6" on page 142 for more information.

3. Deploying the Web Service

   Now that the classpath is set, the environment is ready to run the deployment utility. Ensure that the deployment is carried out within the same command prompt that was used earlier to set the classpath.

10.5.2 Executing a simple deployment to a local default queue manager

To deploy the Web Service, perform the following tasks:

1. Use the same command that was used in step 2 on page 228, as shown in Figure 10-5 in order to set the classpath or run the amqwsetcp script again in a new command prompt.
2. Change the current directory to the location of the directory where the BankingService.asmx is, and run the deployment command. The deployment command that is used to deploy the BankingService is as follows:

   ```
   call "%WMQSOAP_HOME%\bin\amqwdeployWMQService" -f
   BankingService.asmx
   ```

   This is shown in Figure 10-6.

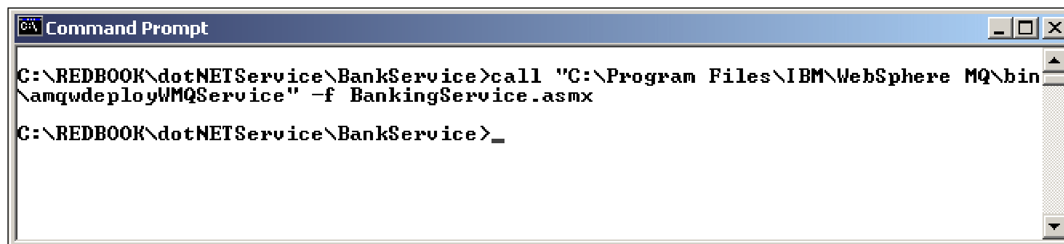


Figure 10-6 Executing a simple deployment

In this command, amqwdeployWMQService.cmd is called with the -f parameter in order to specify the BankingService.asmx Web Service as the Web Service to deploy. The deployment process creates a folder called Generated in the directory where the Web Service is stored. The folder contains the following output of the deployment process:

- A WSDL file named Web Service name_Wmq.wsdl, in this case, BankingService_Wmq.wsdl
- A folder named client that consists of the Web Service proxy generated in two .NET languages, Visual Basic (VB) and C#. In our case, BankingService.cs and BankingService .vb are created. The client folder also consists of the compiled Java proxies stored in the remote/dotNetService folder.
- A folder named Server, which consists of two command files, startWMQNListener.cmd and endWMQNListener.cmd. These command files are responsible for starting a WebSphere MQ listener and stopping a WebSphere MQ listener respectively.

– In the WebSphere MQ environment, a new request queue named SOAPN.name of service is created. In this example, it is SOAPN.BankingService. This queue is where the client puts the request messages to be picked up by the listener. The response messages from the Web Service are placed on a default queue called SYSTEM.SOAP.RESPONSE.QUEUE. The request queue and the response queue are created on the default queue manager.

Note: To redeploy the Web Service without explicitly specifying queue names, it is essential to delete the request queue and the Generated folder created during an earlier deployment. This ensures a clean start and prevents the deployment utility from failing when it discovers that the request queue that is to be created already exists.

10.5.3 Executing a deployment to a local queue manager with specific request and response queues

The deployment process permits a comprehensive degree of control over SOAP/ WebSphere MQ-specific parameters and options when accessing target services.

Web Services define their location in an URI. The URI for the Web Service can be included during the deployment. Although it is optional during deployment, an URI is supplied in order to avoid specifying an URI at run time. The URI's format can include any number of the SOAP/WebSphere MQ-specific parameters and options.

See 5.4.2, "The SOAP/WebSphere MQ Universal Resource Indicator" on page 65 for more information about URI specification. For a quick reference, refer to Appendix C, "Deployment utility quick reference" on page 425.

This section demonstrates a typical deployment specifying an URI. The URI specifies the request queue and the response queue. Define these within the WebSphere MQ environment. See 7.3, "Environment setup" on page 147 for details about creating a queue. In this deployment, the request queue that is used is called BANKING.SERVICE.REQUEST, and the response queue is called BANKING.SERVICE.RESPONSE.

To perform a typical deployment specifying an URI, execute the following tasks:

1. Repeat step 1 on page 229, that is, run the **amqwsetcp** command.
2. Redeploy the BankingService Web Service using the command shown in Example 10-9.

Example 10-9 Redeploying the BankingService Web Service

```
call "%WMQSOAP_HOME%\bin\amqwdeployWMQService" -f BankingService.asmx
-u
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_LocalToSvc&connectio
nFactory=()&replyDestination=BANKING.SERVICE.RESPONSE&initialContextFac
tory=com.ibm.mq.jms.Nojndi"
```

The deployment command that is used contains the class name of the Web Service declared by the -f parameter, in this case, BankingService.asmx, and the URI declared by the -u parameter. The URI specifies the following:

– The transport

This part of the URI declares WebSphere MQ as the transport for the Web Service SOAP messages. This is denoted by jms:/queue.

– The destination

This parameter comes just after the initial jms:/queue string discussed earlier. It specifies the name of the queue that is used for the request message as either a WebSphere MQ queue name or a queue name and queue manager name connected by an @ symbol. In this case, it is denoted by destination=BANKING.SERVICE.REQUEST@QM_LocalToSvc.

– The replyDestination

This specifies the name of the queue in the client side that is used for the response message. In this case, it is denoted by replyDestination=BANKING.SERVICE.RESPONSE.

– The ConnectionFactory

This parameter is required for specifying the client connections and the queue manager and channels used for the client connections. Set this to `connectionFactory=()` if none of the parameters mentioned earlier are to be set.

– The initialContextFactory

This parameter is required and must be set to `com.ibm.mq.jms.Nojndi`. This is required for compatibility with WebSphere Application Server and other products.

The simple deployment to a local default queue manager in 10.5.2, "Executing a simple deployment to a local default queue manager" on page 229 details the output of the deployment process.

10.5.4 Executing a deployment to a remote queue manager

Set up the connectionFactory part of the URI to establish a client or server connection to a remote queue manager.

Client connection

To set up a client connection, a server connection channel is required on the remote queue manager. The deployment URI requires the following:

- connectQueueManager

  This is the name of the remote WebSphere MQ queue manager.

- binding

  This specifies the WebSphere MQ client connection used to connect to the WebSphere MQ queue manager. This can be set to *server*, *auto*, or *client*. In our case, the transport type is set to client.

- clientChannel

  This is the server connection channel setup on the remote queue manager.

- clientConnection

  This is a combination of the name or IP address of the machine the remote queue manager resides on and the TCP/IP port on which it is listening. The default configuration of the queue manager typically sets the port value to 1414.

The URI definition in Example 10-10 sets up the Web Service to allow clients to make a client connection to the Web Service through the local queue manager called QM_LocalToSvc, using a server connection channel called SYSTEM.DEF.SVRCONN. The local queue manager to the Web Service listens on port 1414 on a machine with the IP address 9.1.39.128, as shown in Example 10-10.

Example 10-10 URI definition

```
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_LocalToSvc&connectio
nFactory=(connectQueueManager(QM_LocalToSvc)binding(client)clientChanne
l(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))&replyD
estination=BANKING.SERVICE.RESPONSE&initialContextFactory=com.ibm.mq.jm
s.Nojndi"
```

To deploy this Web Service in order to allow clients to connect remotely using the client connection, use the command shown in Example 10-11.

Example 10-11 Command to deploy the Web Service

```
call "%WMQSOAP_HOME%\bin\amqwdeployWMQService" -f BankingService.asmx
-u
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_LocalToSvc&connectio
nFactory=(connectQueueManager(QM_LocalToSvc)binding(client)clientChanne
l(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))&replyD
estination=BANKING.SERVICE.RESPONSE&initialContextFactory=com.ibm.mq.jm
s.Nojndi"
```

Note: The SOAP WebSphere MQ infrastructure does *not* recognize bracket characters. Therefore, %2528 and %2529, a combination of escape characters and American Standard Code for Information Interchange (ASCII) characters are used to define open brackets “(” and close brackets “)” respectively.

Server binding mode connection

To set up a server binding mode connection, set up the WebSphere MQ environment with a sender channel and a receiver channel, along with the transmission queue, the request queue, and the response queue. See 10.4, “Preparing the WebSphere MQ environment” on page 222 for server binding connection configuration. The deployment URI requires the following:

- The Web Service machine’s local queue manager
- The request queue on the local machine
- The response queue on the Web Service client machine

When the client makes a request, the SOAP/WebSphere MQ sender from the specified URI knows to place the message on the service queue manager’s request queue. In this case, it places the message with the help of the transmission queue on the BANKING.SERVICE.REQUEST queue on the QM_SVC_HOST queue manager. The message is sent across the client’s sender channel. On reaching the service queue manager’s request queue, the request is picked up and processed by the Web Service. The response message from the Web Service message is then placed on a transmission queue by the service listener, which determines the destination queue using the message header and returns it to the destination queue manager using the client’s receiver channel. The listener then reads the message off the local response queue and passes it to the client.

For details about this configuration and how to set it up, see 10.4, “Preparing the WebSphere MQ environment” on page 222.

Figure 10-7 illustrates the message flow.

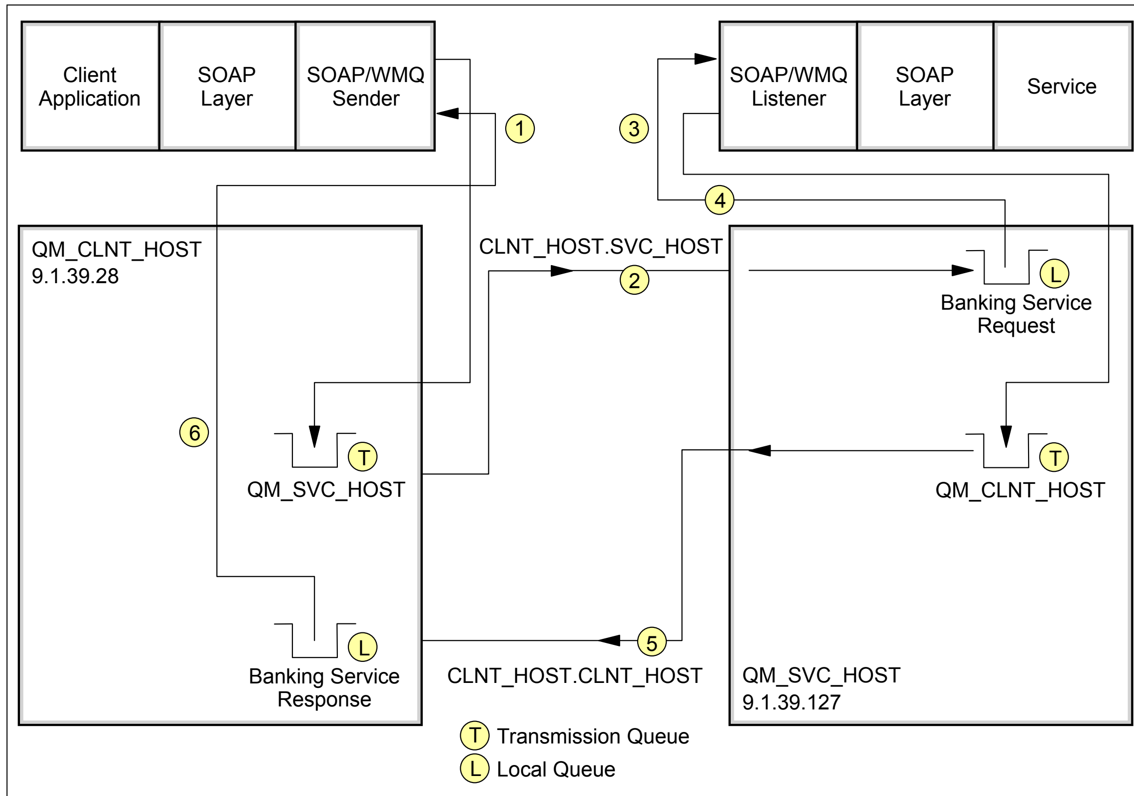


Figure 10-7 Message flow during Web Service invocation in server binding mode

To deploy the BankingService Web Service in order to allow clients to connect remotely using the server binding connection, the command URI is shown in Example 10-12.

Example 10-12 Command URI to deploy the BankingService Web Service

```
"jms:/queue?initialContextFactory=com.ibm.mq.jms.Nojndi&connectionFacto
ry=(connectQueueManager(QM_SVC_HOST)binding(server))&destination=BANKIN
G.SERVICE.REQUEST&replyDestination=BANKING.SERVICE.RESPONSE&targetServi
ce=BankingService.asmx"
```

To deploy the BankingService Web Service in order to allow clients to connect remotely using the server binding connection, the command is shown in Example 10-13.

Example 10-13 Command to deploy the BankingService Web Service

```
call "%WMQSOAP_HOME%\bin\amqwdeployWMQService" -f BankingService.asmx
-u
"jms:/queue?initialContextFactory=com.ibm.mq.jms.Nojndi&connectionFacto
ry=(connectQueueManager(QM_SVC_HOST)binding(server))&destination=BANKIN
G.SERVICE.REQUEST&replyDestination=BANKING.SERVICE.RESPONSE&targetServi
ce=BankingService.asmx"
```

10.6 The SOAP/WebSphere MQ listener

The earlier sections illustrated deployments of gradually increasing complexity. In each of these deployments, the service was deployed on the local machine and the appropriate proxies were generated. After the service is ready for use, one more step is required before the service starts processing requests, that is, the service listener must be started. The service listener is responsible for reading messages from the request queue and forwarding them to the custom service code. The service listener is generated as part of the deployment and can be started by using a single command, as follows:

- Start the service listener by double-clicking **startWMQNListener**, which is located in the Generated/Server folder that was created during deployment.
- Stop the listener by double-clicking **endWMQNListener**.

Attention: We recommend that you do not close the listener either by using the Ctrl+C keys or by closing the command prompt. The safest way to close the service listener is to follow the procedure discussed earlier.

The big picture

The development and deployment is complete. Following is the process involved in a .NET client invoking the BankingService Web Service's credit method:

1. The Web Service is created and deployed. The proxy that is generated is imported into the .NET client environment.
2. The Web Service owner starts a listener so that clients can invoke the service. The Web Service is invoked by the client through a proxy.
3. An appropriate connection to the Web Service queue manager, which is determined from the URI within the proxy, is established.

4. The client sends a request message for the credit operation to be performed on the account with a given amount.
5. The SOAP/WebSphere MQ sender, called by the .NET infrastructure, writes a SOAP request to invoke the Web Service's credit method.
6. The sender causes the request to be put on a WebSphere MQ request queue waiting for pickup.
7. The SOAP/WebSphere MQ listener monitors the request queue, senses that a message has arrived, picks up the message, and invokes the Web Service.
8. The BankingService Web Service credits the account with the specified amount and returns the feedback (response) about whether the credit is successful or not.
9. The response from the Web Service in then placed on a response queue, which the SOAP/WebSphere MQ listener returns to the SOAP/WebSphere MQ sender.
10. The SOAP/WebSphere MQ sender finally passes the response to the Web Service client.

10.7 Error handling

This section explains some of the common error messages you may see when executing a SOAP/WebSphere MQ Web Service.

Unable to get response from queue

Problems occur if the service is unable to get messages from the request queue. In the test environment, this is simulated by using the WebSphere MQ Explorer to configure the request queue so that applications cannot get messages from the queue. The result is that the listener stops running. If the listener is restarted, the get on the request queue is allowed automatically again.

Unable to find specified WebSphere MQ object-request queue

Another possible, and more likely, error condition is if the request queue does not exist. In this case, the service listener fails to start. The error message shown looks as shown Figure 10-8.

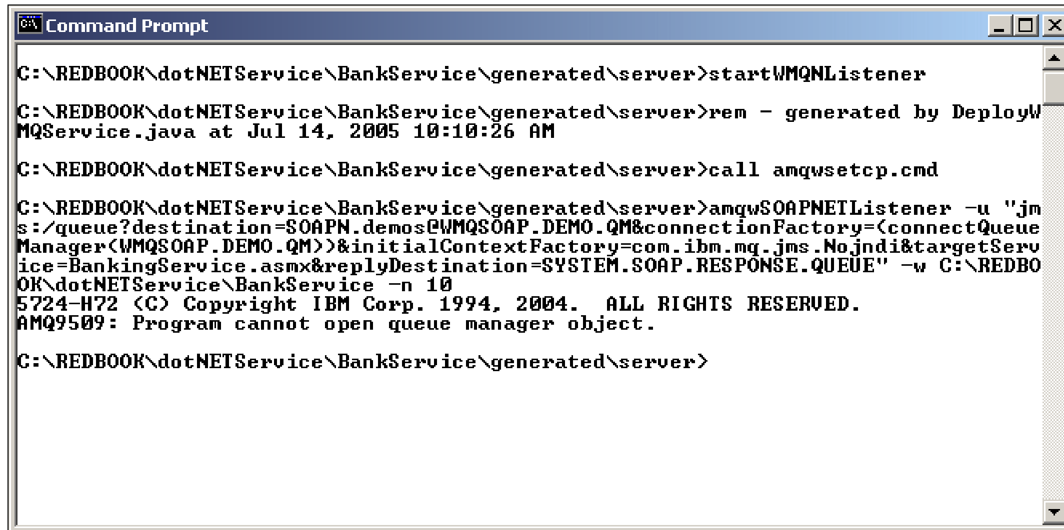


Figure 10-8 Output of listener if the request queue does not exist

The error code output is the error code for an unknown object, and the object, in this case, the request queue that WebSphere MQ cannot find, is shown.

> **Tip:** To obtain a brief description of a WebSphere MQ error, the mqrc utility can be used. The syntax is:
>
> ```
> mqrc <error code>
> ```

Unable to put to a response queue

If the put on the response queue is inhibited, the client is timed out with a timeout error. See 11.4, “Error handling” on page 262. What happens to the message when a put operation fails depends on the integrity and persistence settings of the message. The first check is against the message integrity, with the general rule being that for low-integrity messages, an error message is shown and then

discarded. Persistent messages are then backed out and the put retried. This sequence is repeated until the backout threshold is exceeded.

Note: The default backout threshold is 3. This can be changed by using the -b switch during deployment. For further details, see Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

Message persistence affects the error handling process in a similar manner for a failed put operation.

Note: The behavior based on message integrity and persistence can be altered during the deployment process. For further information, see Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

Unable to find specified WebSphere MQ object

Another potential problem for the service listener is if the response queue specified in the URI does not exist. In this case, the service listener starts as usual and shows no error messages. This is the correct behavior because the client can potentially override the URI, thereby specifying a different response queue. Error messages, if any, are shown on the client. This involves an exception containing the WebSphere MQ error code 2085. For further details, see Chapter 11, “.NET client” on page 243.

Unexpected message on queue

It is not just missing or limited queues that may lead to errors. An unexpected message on a queue may also cause problems. In our case, it is an unexpected message on a request queue. In the test environment, this is simulated by placing a simple text file on the request queue. This leads to the generation of the following error:

```
Unrecognized RFH2 identifier. MQCC_FAILED(2)
MQRCCF_MD_FORMAT_ERROR(3023)
```

Subsequent messages continue to be processed as usual, provided they are in the correct format.

10.8 Security

Securing communication between a Web Services client and a Web Service is achieved most effectively by using the Secure Sockets Layer (SSL). This section discusses enabling the security services provided by SSL within the scenarios described in the earlier sections.

The URI provides several key words that you can configure to enable SSL. There are different key words depending on the Web Services client environment. In a Microsoft .NET environment these are:

- sslKeyRepository

  No password is explicitly required for this because this is stored on creation of the .kdb file in the stash file.

- sslCipherSpec

In a Java environment, the key words are:

- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword
- sslCipherSuite

Before setting these values on the URI, create and configure the key repositories and certificate chains. The examples provided in the list that follows assumes that this initial configuration is completed. Refer to Chapter 6, "Security" on page 107 for details about this. The key store locations and passwords are those used in the examples.

If the Web Services client is running a Microsoft .NET environment, set the values on the URI as follows:

- sslKeyRepository=C:\SSL\client\key (without the .kdb extension)
- sslCipherSpec=RC4_SHA_US

The sslCipherSpec example provided in this list is one of many that you can select. For more information about the possible choices, refer to *WebSphere MQ Security,* SC34-6588.

A full URI may look as shown in Example 10-14.

Example 10-14 A full URI

```
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_SVC_HOST&connectionF
actory=connectQueueManager(QM_SVC_HOST)&replyDestination=BANKING.SERVIC
E.RESPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi&sslKeyRepository
=C:\SSL\client\key&sslCipherSpec=RC4_SHA_US"
```

To enable security using a Java client, set the following values on the URI at deployment:

- sslKeyStore=C:\SSL\client\key (without the .jks extension)
- sslKeyStore=password
- sslTrustStore=C:\SSL\client\key or C:\SSL\client\trust if the trust store is different to the key store, without the .jks extension)
- sslTrustStorePassword=password
- sslCipherSuite=SSL_RSA_WITH_RC4_128_SHA

The values of the sslKeyStore and sslTrustStore must be the locations on the machine the client is running on, whether the machine is remote or local to the Web Service. The sslCipherSuite example provided in this list is the value equivalent of the sslCipherSpec of the .NET client. For more information about the possible choices, refer to *WebSphere MQ using Java,* SC34-6591.

A full URI may look as shown in Example 10-15.

Example 10-15 A full URI

```
"jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_SVC_HOST&connectionF
actory=connectQueueManager(QM_SVC_HOST)&replyDestination=BANKING.SERVIC
E.RESPONSE&initialContextFactory=com.ibm.mq.jms.Nojndi&sslKeyStore=C:\S
SL\client\key&sslKeyStorePassword=password&sslTrustStore=C:\SSL\client\
trust&sslTrustStorePassword=password&sslCipherSuite=SSL_RSA_WITH_RC4_12
8_SHA"
```

It is important that the value selected for sslCipherSpec or sslCipherSuite is equivalent to the one set on the SVRCONN's SSLCIPH parameter on the WebSphere MQ queue manager being connected to.

Optionally, it may also be necessary to use the sslPeerName value on the URI. Using this option forces the client to send the WebSphere MQ queue manager its certificate, so that the queue manager can check whether the distinguished names on the certificate matches the distinguished names it is configured to trust.

10.9 Using the Web Service

Now that the .NET Web Service is created and deployed, use the Web Service with a client that resides on the same machine or with a client that resides on a different machine from the Web Service.

In a situation where the client resides on a different machine from the Web Service, the Web Service is either deployed on the server machine and the proxies copied across to the client machine, or the Web Service is deployed on both machines and the redundant elements from each platform removed. See 5.4, “The deployment process” on page 59 for details about how to deploy Web Services. The listener must also be started before invocation from the client.

10.10 Summary

This chapter discussed the creation of a Web Service using WebSphere MQ as the transport mechanism. A simple class providing four methods to be exposed as services, was created. This class was then deployed as a Web Service using WebSphere MQ in a number of different configurations. These configurations were simple to begin with, but gradually increased in complexity. A production environment involves even greater complexity. However, the aim of this chapter is to introduce the concepts involved.

This chapter also discussed some simple error handling and security concepts. These concepts are discussed in greater detail in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 and Chapter 6, “Security” on page 107.

11

.NET client

This chapter demonstrates the development of .NET Web Service clients that invoke Web Services by sending their messages over WebSphere MQ. A .NET client that has already been prepared as a HyperText Transfer Protocol (HTTP) Web Service client must register WebSphere MQ as its transport before invoking a SOAP/WebSphere MQ Web Service. This client is capable of invoking any of the three Web Services created in the earlier chapters, that is, Chapter 10, “.NET Web Service” on page 213, Chapter 8, “Axis Web Service” on page 159, and Chapter 12, “WebSphere Application Server Web Service” on page 269.

This chapter covers the following topics:

- Creating a .NET Web Service client using Visual Studio .NET
- Registering the .NET Web Service client for transport over WebSphere MQ
- Invoking the Web Service
- Running the .NET Web Service client
- Security considerations and implementation
- Error handling

© Copyright IBM Corp. 2006. All rights reserved.

11.1 Design

This section discusses the design of the .NET Web Service client.

The .NET client design

A BankingService Web Service is developed in the preceding Web Services chapters, Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269. This .NET client is designed to invoke any of these Web Services.

The Web Service models a bank account and the common operations that take place on it. Following are the operations:

- getBalance

  This is a method to return the account balance.

- credit

  This is a method to add a provided amount to the existing account balance and store the operation details in a user-defined object called a BankOperation object.

- debit

  This is a method to deduct a provided amount from the existing account balance and store the operation details in a user-defined object called a BankOperation object.

- getStatement

  This is a method to return the last three operations on an account. This is returned in an array of BankOperation objects.

The .NET Web Service client calls each of these methods and shows the results in a graphical user interface (GUI).

11.2 Requirements

For the implementation of this .NET Web Service client, the following are required:

- Windows 2000 Service Pack 2 (SP2) or earlier, Windows XP
- WebSphere MQ V6 client or server
- Microsoft .NET Framework 1.1 (SP1 or earlier)

- Microsoft .NET Framework software development kit 1.1 (SDK 1.1) and any text editor
- Visual Studio .NET (2003) (optional)

11.3 Implementation

This section discusses the implementation of the BankingService Web Service client used in this chapter. Download the BankingService Web Service client code from Appendix D, “Additional material” on page 431.

This section also discusses the WebSphere MQ environment setup that is required before the Web Service client invokes the Web Service.

11.3.1 Proxy code

In order to create a client for any Web Service, the developer creating the client requires a Web Services Description Language (WSDL) or proxy code. The deployment process of a SOAP/WebSphere MQ Web Service produces a WSDL for the Web Service, which can be used by the client to generate a proxy for invoking the service. The proxy acts as an intermediary between the client and the Web Service. It hides the complexity of invoking the Web Service and forwards calls from the client to it. The deployment process creates the proxy and the WSDL.

Therefore, to invoke the .NET Web Service, the client can be provided with a WSDL, in which case, the proxy is generated for the Web Service or is generated during deployment. For details about the location of these files, see 10.5.2, “Executing a simple deployment to a local default queue manager” on page 229.

Returning to the banking scenario, the proxy files shown in Table 11-1 are generated.

Table 11-1 Proxy files

File name	Description
BankingService.cs	Interface for the methods exposed by the banking service, that is, the proxy code (C#)
BankingService.vb	Interface for the methods exposed by the banking service, that is, the proxy code (VB)

11.3.2 Implementing .NET client to make synchronous calls

The Web Service client code is implemented as a Windows application in C#. This section discusses the GUI used to invoke the Web Service. It also discusses the following:

- The registration of the Web Service client for transporting SOAP messages over WebSphere MQ
- The import of the Web Service proxy
- The method calls to the Web Service on each of the buttons on the GUI
- The WebSphere MQ client environment setup required for invoking the Web Service, both in the client mode and the server binding mode

To create a Windows application project in Visual Studio .NET, open Visual Studio .NET, select **File** → **New** → **Project**, as shown in Figure 11-1.

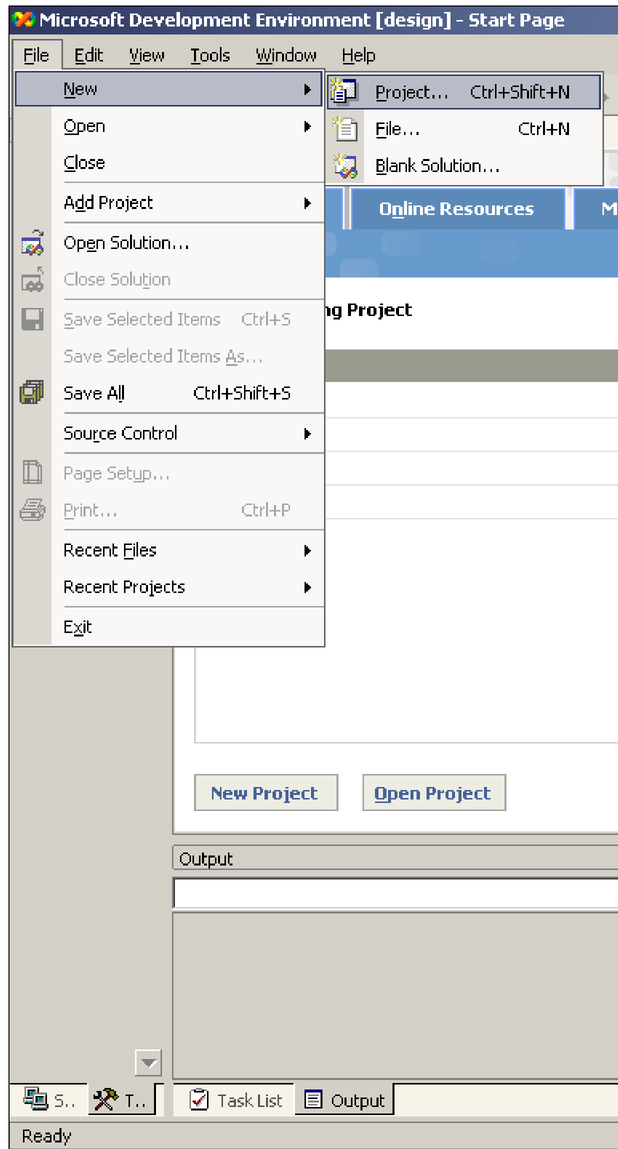


Figure 11-1 Creating a new Windows application project

Registering WebSphere MQ as transport

In order to make the .NET client aware that its SOAP messages are sent over WebSphere MQ, the client code must include the code shown in Example 11-1. In the .NET Windows application form that serves as the Web Service client, this code is included in the forms constructor.

Example 11-1 Registering WebSphere MQ transport for SOAP

```
public Form1()
{
   //Register the WMQ SOAP extension
   IBM.WMQSOAP.Register.Extension();
}
```

After this is done, the Microsoft .NET Web Services framework is able to accept an Universal Resource Indicator (URI) prefixed with jms:.

The .NET HTTP Web Service proxy has an URI in the form of http://localhost/BankService/BankingService.asmx.

The SOAP/WebSphere MQ Web Services have an URI in the form of jms:/queue?initialContextFactory=com.ibm.mq.jms.Nojndi&connectionFactory=()&destination=SOAPN.BankingService&targetService=BankingService.asmx.

To permit the use of WebSphere MQ transport for SOAP dynamic link library (DLL), amqwsoap.dll located in the WebSphere MQ home directory\bin must be added as a reference. The Windows application project must also reference the System.Web.Services library for the Web Service proxy to compile.

Invoking the Web Service

At this stage, the .NET Web Service client is ready for transporting its SOAP messages over WebSphere MQ, but requires a SOAP/WebSphere MQ Web Service to invoke.

Invoking a Web Service using a provided Web Services Description Language

If the WSDL is provided, use the Microsoft .NET Framework's wsdl.exe command to generate a proxy from the Web Service's WSDL by performing the following tasks:

1. Save the WSDL file to a convenient location. In this case, it is saved in C:\REDBOOK\dotNETService\BankService\generated.
2. Open a Windows command prompt.

3. Change the current directory to the location in which to store the created proxy. In this example, the location is C:\REDBOOK\dotNETService\BankService\service proxy.
4. Type `wsdl`, and then the location of the WSDL file as follows:

```
wsdl
“C:\REDBOOK\dotNETService\BankService\generated\BankingService_Wmq.w
sdl”
```

This is shown in Figure 11-2.

:

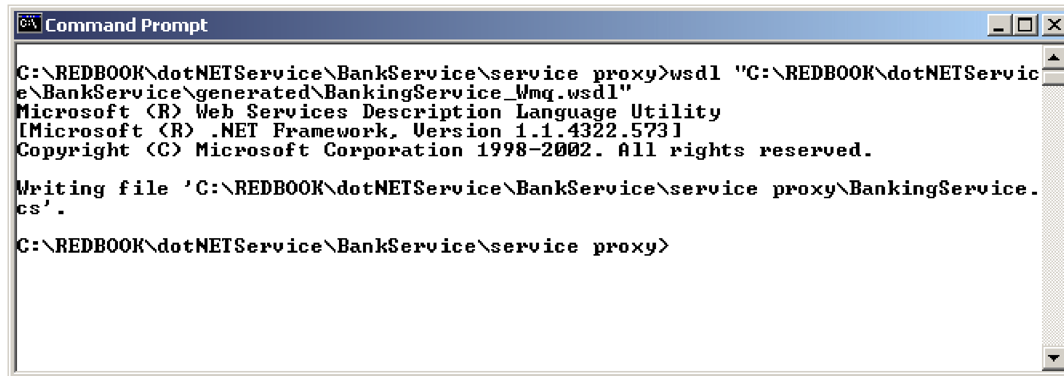


Figure 11-2 Proxy generation from Web Service’s WSDL using .NET Framework’s wsdl.exe

Note: For the **`wsdl.exe`** command to work from any folder, include its location, which is typically root\Program Files\Microsoft.NET\SDK\version\Bin, in your computer’s PATH environment variable. To do this:

1. Right-click **My Computer** and select **Properties**.
2. Click the **Environment Variables** button in the Advanced Tab.
3. Add the path to the wsdl.exe to the path variable.

Invoking a Web Service using a provided proxy

If the Web Service proxy is provided, you can import it into the application immediately.

If you are *not* using Visual Studio .NET, copy the proxy to the directory where the client code is saved.

To import the proxy into the application in Visual Studio .NET, right-click the project and select **Add** → **Add Existing item**. Browse to the location of the Web Service proxy, and double-click it. Figure 11-3 shows the addition of proxy to a project.

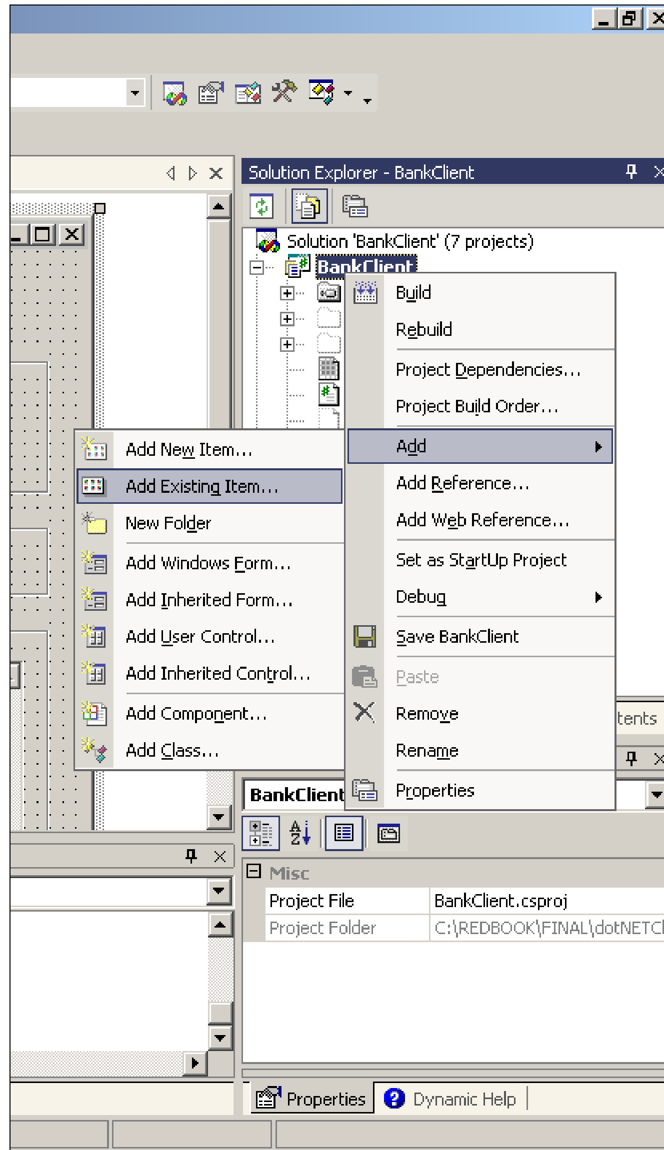


Figure 11-3 Adding proxy to the project

The Graphical User Interface

The Windows application form that is developed serves as a graphical user interface (GUI) consisting of four buttons, whose event handlers invoke the Web Service methods. The Windows application form is shown in Figure 11-4.

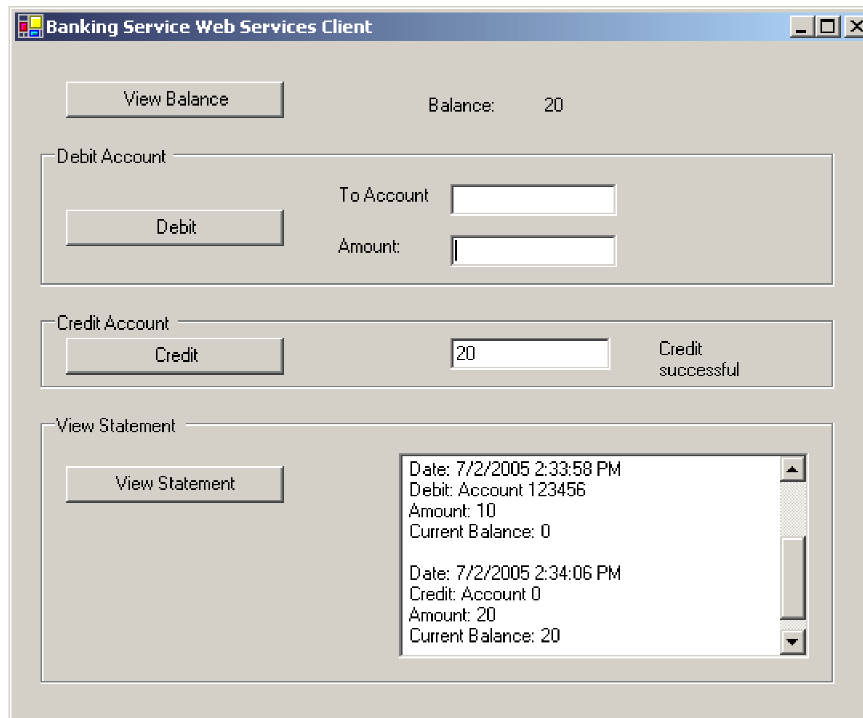


Figure 11-4 Windows application form serving as the BankingService Web Service client

Table 11-2 provides a brief description of the GUI buttons.

Table 11-2 Description of the GUI buttons

Button name	Description
View Balance	Calls the Web Service's getBalance method. The balance is shown in the label next to it.
Debit	Calls the Web Service's debit method. If the amount specified for debit in the Amount text box is greater than the balance in the account, a BankOperation Exception is thrown and shown in a message box. Otherwise, the balance after debit is shown in the label next to the View Balance button. [a]

Button name	Description
Credit	Calls the credit method within the BankingService Web Service. The balance after the credit is shown in the label next to the View Balance button.
View Statement	Calls the getStatementmethod within the BankingService Web Service. The getStatement methods returns an array of three BankOperation objects that are broken down and shown in the text area next to the View Statement button.

a. If an exception is thrown from this debit method within the Web Service, the exception is wrapped up and returned in a SoapException.

Each button's event handler invokes the Web Service by creating an instance of the imported proxy. This exposes the Web Service methods. The code in Example 11-2 shows the general invocation code, where BUTTON indicates the buttons described in Table 11-2 and METHOD indicates the method within the Web Service that is invoked.

Example 11-2 Invoking the Web Service by creating an instance of the proxy

```
private void BUTTON_Click(object sender, System.EventArgs e)
{
      //Make a new instance of the Web Service
      BankingService service = new BankingService();
      //Call the web method
      service.METHOD();
}
```

11.3.3 Implementing the .NET client to make asynchronous calls

The .NET client developed earlier can be modified to demonstrate short-term asynchrony support provided by WebSphere MQ transport for SOAP. The short-term asynchronous call is implemented the same way it is implemented in .NET. It is demonstrated on credit method only. The Web Service is modified to include a setDelay function, which causes the credit method to sleep for a specified time in seconds before returning a response.

The demonstration of the asynchronous call is performed by setting a delay in the credit method on the Web Service, as shown in Example 11-3.

Example 11-3 Code to implement delay on credit method

```
public static int delay = 0;

[WebMethod] [SoapRpcMethod]
```

```
public void setDelay(int delaySpecified)
{
   //convert number of seconds for delay to milliseconds
   delay = delaySpecified * 1000;
}

[WebMethod] [SoapRpcMethod]
public bool credit(double amount)
{
   //credit the account

   //if delay has been specified
   if(delay > 0)
   {
      //make service sleep for specified number of seconds
      Thread.Sleep(delay);
   }
   return true;
}
```

The credit button event handler contains the code to make a synchronous call with a delay in the credit method's response, and the code to make an asynchronous call with a delay in the credit method's response. The synchronous call is commented out. It is used for demonstration purposes only. If the call is made synchronously, the GUI remains inactive during the call. If the call is made asynchronously, other operations can be conducted on the GUI. Example 11-4 shows the new credit button event handler.

Example 11-4 Credit button event handler

```
private void btnCredit_Click(object sender, System.EventArgs e)
{

   //Make a new instance of the Web Service
   BankingService service = new BankingService();
   //UNCOMMENT THIS CODE TO: Call the Web Service synchronously
   /*
   //set the service to time out after a long time
   service.Timeout= 300000;
   //delay the response on the credit method
   service.setDelay(Convert.ToInt32(txtAsyncDelay.Text));
   bool creditSuccessful =
         service.credit(Convert.ToDouble(txtCreditAmount.Text));
   lblBalance.Text =  service.getBalance().ToString();
   */
```

```
   //Call the web method to credit the account using async call back
   IAsyncResult ar =
         service.Begincredit(Convert.ToDouble(txtCreditAmount.Text),
new
         AsyncCallback(creditCallback), null);

   while (!ar.IsCompleted)
   {
      //Keep updating the balance label
      //NOTE: While we are waiting for the response, we are making
      //asynchronous calls to get the balance
      lblBalance.Text =  service.getBalance().ToString();
   }
   bool creditSuccessful = service.Endcredit(ar);

}
```

.NET clients can implement asynchronous calls to .NET Web Services with two techniques, the *callback technique* and the *wait technique*. WebSphere MQ transport for SOAP supports asynchronous calls to the Web Service regardless of how the asynchronous call is implemented. This example uses the wait technique and the callback technique.

For the credit button event handler to invoke the BankingService Web Service asynchronously using the callback technique, modify the code as follows:

1. Define a callback function that implements the AsyncCallback delegate as shown in Example 11-5.

Example 11-5 Code to implement asynchronous callback for credit method

```
// Short term async Callback method
public static void creditCallback(IAsyncResult ar)
{
   // Recover the .NET proxy object from the AsyncState parameter
   BankingService service = (BankingService) ar.AsyncState;
   bool creditSuccessful = service.Endcredit(ar);
}
```

2. In the credit button event handler, call the beginCredit method, passing an instantiated AsyncCallback delegate as the second argument and the object providing the state as null, as shown in Example 11-6.

Example 11-6 Code to implement asynchronous request

```
private void btnCredit_Click(object sender, System.EventArgs e)
{

   //Make a new instance of the Web service
   BankingService service = new BankingService();

   IAsyncResult ar =
      service.Begincredit(Convert.ToDouble(txtCreditAmount.Text),
      new AsyncCallback(creditCallback), null);

```

3. Write a while loop to check if the asynchronous request is complete. In this case, this is used for demonstration purposes only because this is not the most practical way of polling for response from the asynchronous request. Example 11-7 shows the code to poll for response.

Example 11-7 Code to poll for response

```
int i = 0;
while (!ar.IsCompleted)
{
   lblBalance.Text =  service.getBalance().ToString();
}
```

For the credit button event handler to invoke the BankingService Web Service asynchronously using the wait technique, modify the code as follows:

1. In the credit button event handler, call the beginCredit method with two null arguments, as shown in Example 11-8.

Example 11-8 Code to implement the asynchronous request

```
private void btnCredit_Click(object sender, System.EventArgs e)
{

   //Make a new instance of the Web service
```

```
BankingService service = new BankingService();

IAsyncResult ar =
   service.Begincredit(Convert.ToDouble(txtCreditAmount.Text),
   null, null);
```

2. Call the AsyncWaitHandle.WaitOne to wait for the processing of the call to complete as shown in Example 11-9.

Example 11-9 Waithandler on asynchronous call

```
ar.AsyncWaitHandle.WaitOne();
```

3. When the wait method returns, the client calls the end method, as shown in Example 11-10.

Example 11-10 Calling the end method

```
bool creditSuccessful = service.Endcredit(ar);
lblBalance.Text =  service.getBalance().ToString();
```

The code to implement both the techniques is included in the downloadable code, with the callback technique commented out.

11.3.4 Preparing the WebSphere MQ environment

The client can connect to the Web Service in two ways:

- Client mode

  If the client resides on a different machine from the Web Service and does not have its own queue manager, a client connection is used to enable communication between the client machine and the queue manager located on the Web Service machine.

- Server binding mode

  If the client has its own queue manager and resides on the same machine or a different machine from the Web Service's queue manager, this type of connection is used. It requires the setting up of queue managers, transmission queues, and local queues, and the sender and receiver channels to enable communication between the client's queue manager and the queue manager located on the Web Service machine.

The WSDL generated during the deployment process specifies the type of connection that is used to connect to the Web Service. This is specified in the soap:address tag, for example, the BankingService Web Service deployed for a client connection has a soap:address tag as shown in Example 11-11.

Example 11-11 WSDL URI definition

```
<soap:address
location="jms:/queue?initialContextFactory=com.ibm.mq.jms.Nojndi&connec
tionFactory=(connectQueueManager(QM_LocalToSvc)binding(client)clientCha
nnel(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))” />
```

The URI with the proxy generated during the Web Service deployment also specifies what type of connection is used to connect to it. This is specified within the Web Service proxy class constructor as shown in Example 11-12 for the BankingService Web Service deployed for a client connection.

Example 11-12 Proxy URI definition

```
public BankingService() {
        this.Url =

"jms:/queue?destination=SOAPN.demos@WMQSOAP.DEMO.QM&connectionFactory=”
+
   “(connectQueueManager(WMQSOAP.DEMO.QM))&initialContextFactory=” +
   “com.ibm.mq.jms.Nojndi&targetService=BankingService.asmx” +
   “&replyDestination=SYSTEM.SOAP.RESPONSE.QUEUE";
}
```

Where the Web Service is not deployed for the type of connection the client wants to use, the URI specifying connection details must be supplied by the client. However, this is not recommended in the production environment. Example 11-13 shows how to override the URI at runtime.

Example 11-13 Overriding the URI at runtime

```
private void btnGetBalance_Click(object sender, System.EventArgs e)
{
   //Make a new instance of the Web service
   BankingService service = new BankingService();
   //override the Web Services URI
   service.Url =
"jms:/queue?initialContextFactory=com.ibm.mq.jms.Nojndi&”+

“connectionFactory=(connectQueueManager(QM_CLNT_HOST)binding(server))&”
+
```

```
    "destination=BANKING.SERVICE.REQUEST&replyDestination="+
    "BANKING.SERVICE.RESPONSE&"+
    "targetService=sample.axisSvc.BankingService.java";

    //Call the web method to get the current balance then display it
    lblBalance.Text = service.getBalance().ToString();

}
```

11.3.5 Setup for client mode and server binding mode connection

A client that has a WebSphere MQ environment, but does not have a queue manager defined, or one that is running only a WebSphere MQ client environment does not require additional setup in its WebSphere MQ environment. The URI in the WSDL and Web Service proxy specifies the location of the Web Service and the way the client connects to it.

If the client connects to the queue manager in the Web Service location using its own queue manager, the following is required:

- A WebSphere MQ queue manager, in this case, QM_CLNT_HOST
- A sender channel to transport messages to the service queue manager, in this case, CLNT_HOST.SVC_HOST
- A receiver channel to receive messages from the service queue manager, in this case, SVC_HOST.CLNT_HOST
- A transmission queue with the same name as the remote queue manager (service queue manager) name, in this case, QM_SVC_HOST
- A local queue for receiving response, in this case, BANKING.SERVICE.RESPONSE

Figure 11-5 shows the environment setup.

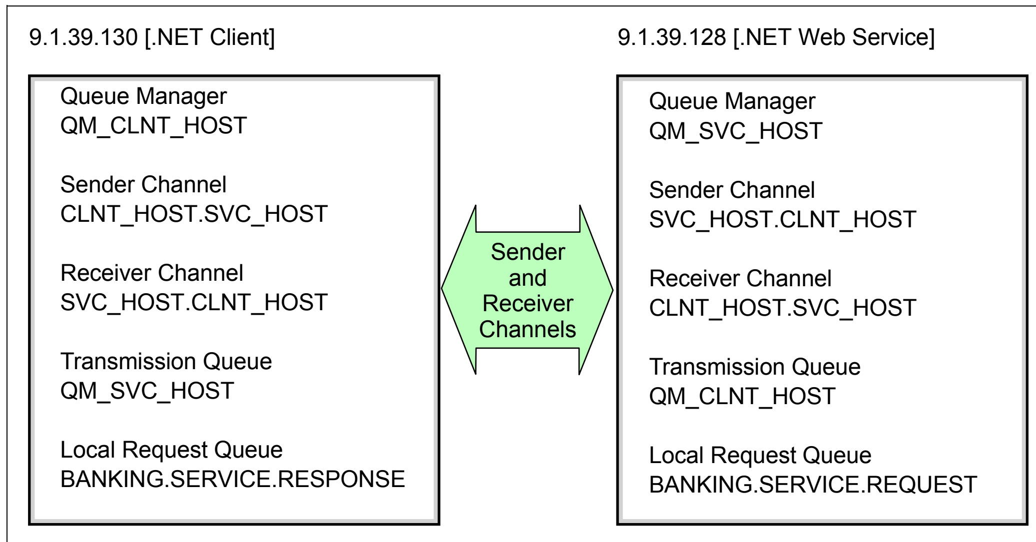


Figure 11-5 Environment setup for a server binding mode connection

Scripts to configure WebSphere MQ are included with the source code download. To configure a service queue manager called QM_SVC_HOST with the downloaded script file, refer to 7.3.1, “Basic WebSphere MQ administration” on page 151.

Provided the Web Service has configured its environment appropriately, the WebSphere MQ configuration described earlier allows the client to send messages and receive replies. The URI provided within the WSDL and proxy specifies the connections.

1. When the client makes a request, the SOAP/WebSphere MQ sender from the URI knows to place the message on the service queue manager’s request queue. In this case, it places a message with the help of the transmission queue on the BANKING.SERVICE.REQUEST queue on the queue manager QM_SVC_HOST, as shown in Figure 11-6, with arrows labelled 1 and 2.
2. The message is sent across the client’s sender channel, CLNT_HOST.SVC_HOST.
3. On reaching the service queue manager’s request queue, the request is picked up and processed by the Web Service as indicated by the arrow labelled 3 in Figure 11-6.
4. The response message from the Web Service message is then placed on a transmission queue by the service listener as indicated by the arrow labelled 4 in Figure 11-6.

5. The service listener determines the destination queue through the message header and returns it to the destination queue manager through the client's receiver channel, SVC_HOST.CLNT_HOST, as indicated by the arrow labelled 5 in Figure 11-6.
6. The listener then reads the message off the local response queue and passes it to the client as indicated by the arrow labelled 6 in Figure 11-6.

Figure 11-6 illustrates the message flow.

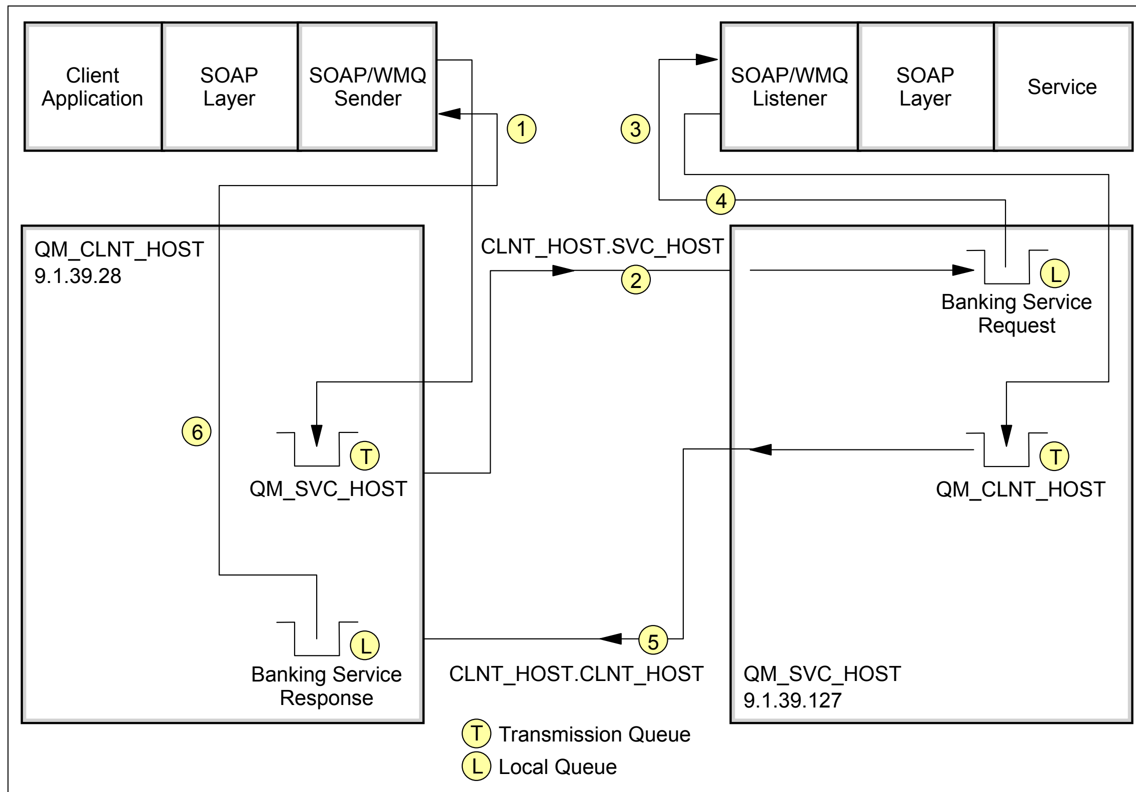


Figure 11-6 Message flow during Web Service invocation in server binding mode

At this stage, the client can be run to invoke the BankingService Web Service. It is worth emphasizing that the client can invoke any Web Service, provided the URI specified within the Web Service WSDL and the proxy specify the location of the Web Service and how the client can connect to it. If in server binding mode, extra configuration is required in the client's WebSphere MQ environment.

The downloadable code consists of separate implementations of the BankingService .NET client, which does the following:

- Makes synchronous calls to the Web Service on the same machine that sends its calls over WebSphere MQ. The project is called BankClientSOAPWMQ.
- Makes synchronous calls to the Axis Web Service, which is located on a machine that is different from the one sending its calls over WebSphere MQ. The project is called BankClientAxisWebService.
- Makes synchronous calls to the WebSphere Application Server Web Service, which is located on a machine that is different from the one sending its calls over WebSphere MQ. The project is called BankClientWASWebService.
- Makes short-term asynchronous calls to the Web Service on the same machine that sends its calls over WebSphere MQ. The project is called BankClientSTAsync.

11.4 Error handling

This section details some of the common errors you may see when running a client that calls a Web Service using WebSphere MQ as a transport mechanism.

Unable to put request to queue

Sometimes, a client is unable to put a request to the queue. In the test environment, this is simulated by using WebSphere MQ Explorer to inhibit the put operation. This generates an exception on the client, with a completion code of 2 and reason code of 2051, the first few lines of which are shown in Figure 11-7.

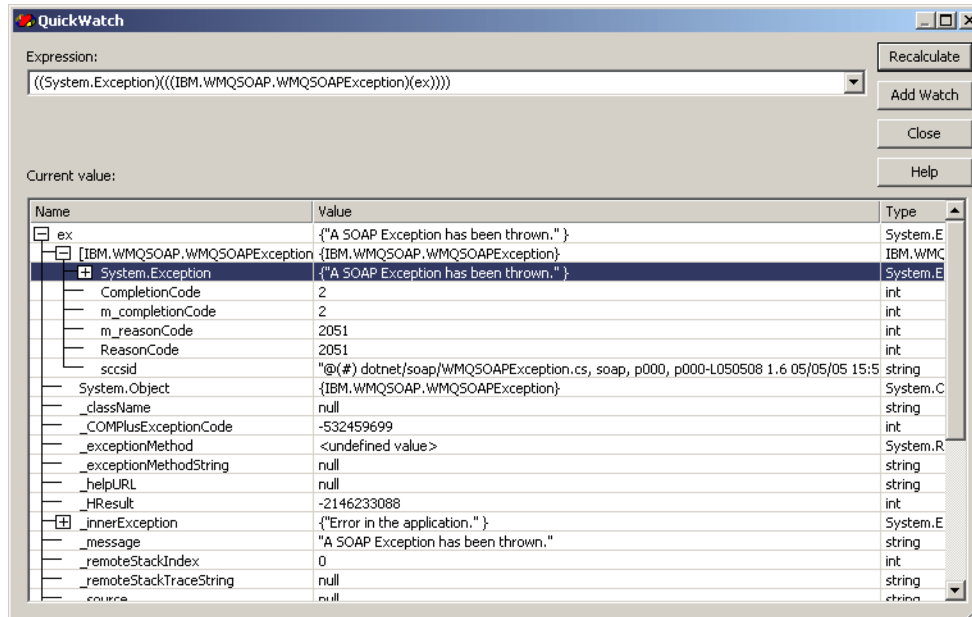


Figure 11-7 Debug information when put is inhibited

A utility called mqrc is shipped with WebSphere MQ to provide brief descriptions for these error codes. The syntax for mqrc is:

```
mqrc <error code>
```

If this command is typed in a Windows command prompt, the output for code 2051 is:

```
2051 0x00000803 MQRC_PUT_INHIBITED
```

Unable to get response from queue

If the response queue is inhibited, an exception is generated with a completion code of 2 and reason code of 2210, the first few lines of which are shown in the debug window shown in Figure 11-8. In the test environment, this is simulated by using WebSphere MQ Explorer to inhibit the get operation on the response queue.

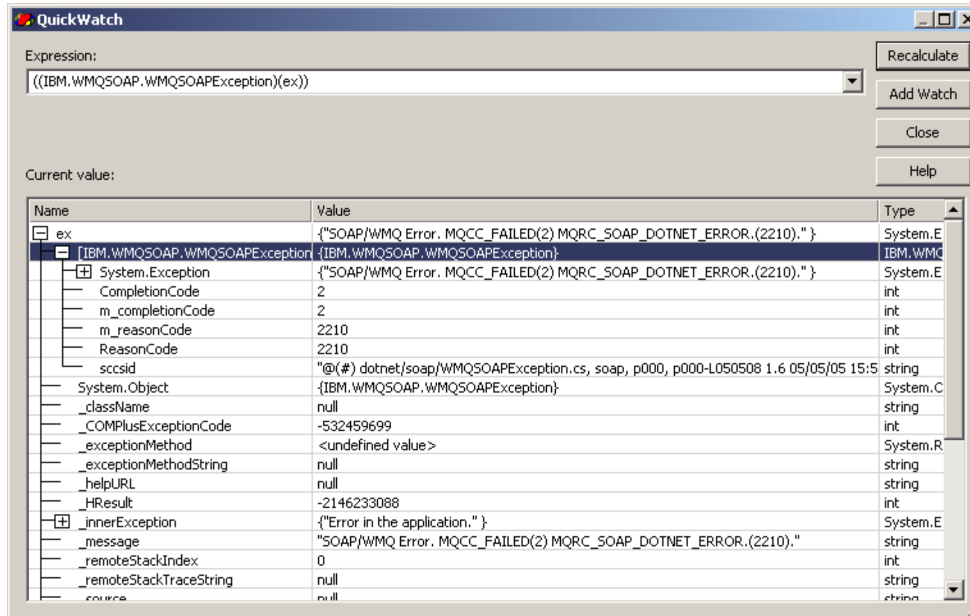


Figure 11-8 Debug information when get is inhibited

Unable to find specified WebSphere MQ object

A more realistic scenario is if the client is started when the specified request queue does not exist. In the test environment, this is simulated by executing the client with an URI, including a request queue name that does not exist. This generates an exception on the client, with a completion code of 2 and reason code of 2085, the first few lines of which are shown in Figure 11-9.

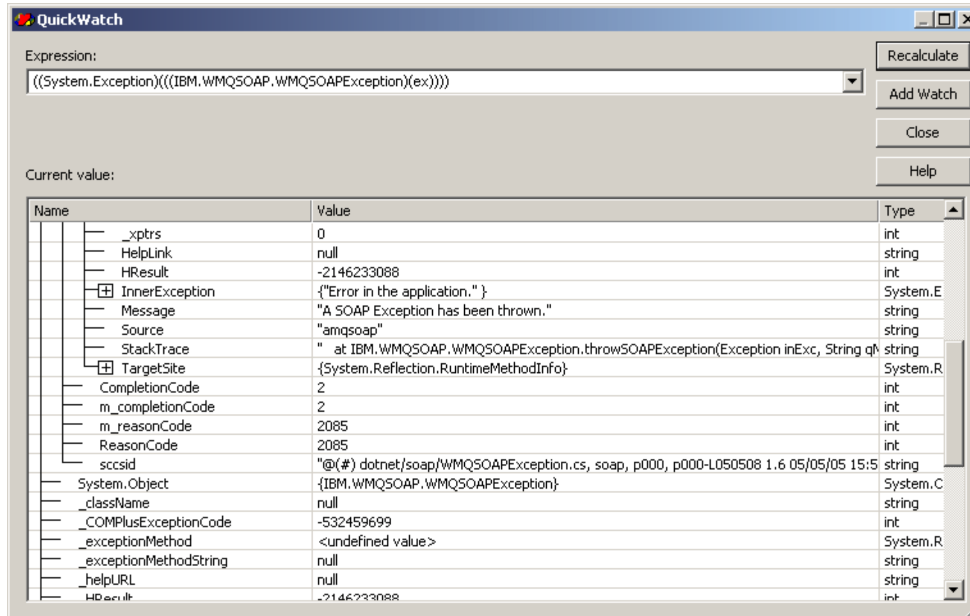


Figure 11-9 Debug information when the request queue specified does not exist

Use the mqrc again. Following is the output:

```
2051 0x00000825 MQRC_UNKNOWN_OBJECT_NAME
```

Listener not started

Another possible situation is the client not receiving a response. In the test environment, this is simulated by starting the client, but not the listener. The result of this is that the client gives the impression of hanging while waiting for a response. Eventually, the client times out and an exception with a completion code of 2 and reason code of 2033 is returned, the first few lines of which are shown in the debug window in Figure 11-10.

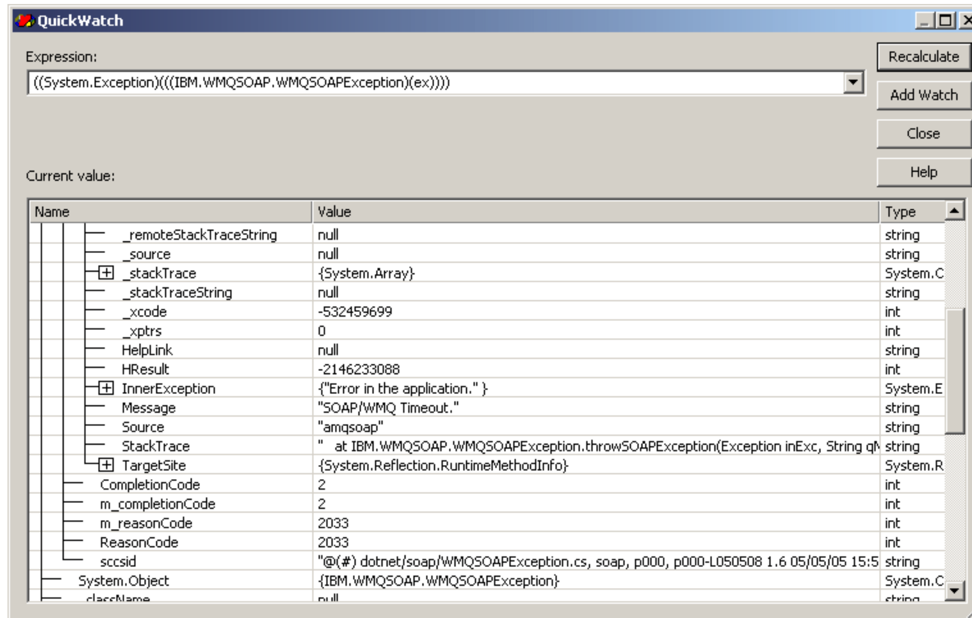


Figure 11-10 Debug information when listener is not started, causing timeout

Following is the output of mqrc in this instance:

```
2033 0x00000833 MQRC_NO_MSG_AVAILABLE
```

Incorrect message format

Another exception message that you may encounter is when the client encounters a message in an incorrect format. This is often caused by a report message being returned by the service. In the test environment, this is simulated by placing a simple text message on the response queue while the client is executing. The Web Service client can deal with the report messages appropriately.

11.5 Security

The security configuration used by the Web Services client is prescribed by the URI in the proxies in the form of SSL key words. The key words in the .NET Web Services client are:

- sslKeyRepository
- sslCipherSpec

When a .NET client connects to a WebSphere MQ queue manager using a client connection through a SVRCONN, the specific values for these key words, are used to determine the following.

- The location of the user certificate for the client and any certificate authority (CA) certificates.
- The password to access that certificate stored in the stash file with the same name as the key repository, but with a .sth extension.
- The cipherSpec to be used during communication to secure the data.

The values of each of these key words are picked up from the URI. For the SSL to become enabled, the key repository must already exist and must contain the correct certificates. For more details about the configuration required to set up SSL, refer to Chapter 6, “Security” on page 107.

11.6 Summary

This chapter discussed the creation of clients that can be used with multiple Web Services. These Web Services have a common transport mechanism, WebSphere MQ. However, the Web Services are implemented on multiple platforms, using multiple development environments.

Using this chapter in conjunction with one of the service chapters (Chapter 8, “Axis Web Service” on page 159, Chapter 10, “.NET Web Service” on page 213, and Chapter 12, “WebSphere Application Server Web Service” on page 269) allows you to create a client and a Web Service on different platforms, with configurations of differing complexity.

After demonstrating how to create the client, this chapter discussed some of the common errors that clients may encounter. Error handling concepts are discussed in 4.9.4, “Security and error handling” on page 46.

The knowledge gained while developing clients and services is built on in the later chapters, as more advanced concepts, including invoking methods asynchronously and using transactions on the client side, are discussed.

For more information about writing Web Services using WebSphere MQ as a transport mechanism, refer to Chapter 8, “Axis Web Service” on page 159, Chapter 10, “.NET Web Service” on page 213, and Chapter 12, “WebSphere Application Server Web Service” on page 269.

12

WebSphere Application Server Web Service

This chapter demonstrates the implementation of a WebSphere Application Server Web Service using Rational Application Developer. The implementation is top-down, using Web Services Description Language (WSDL) generated from the .NET Web Service in Chapter 10, ".NET Web Service" on page 213. This is available for download.

The transport used by the WebSphere Application Server Web Service is SOAP/Java Messaging Service (JMS). This is the existing functionality in WebSphere Application Server. As discussed in Chapter 5, "SOAP/WebSphere MQ implementation" on page 49, WebSphere MQ provides a messaging bus that is capable of connecting SOAP/JMS in WebSphere Application Server with WebSphere MQ and Customer Information Control System (CICS) using SOAP/WebSphere MQ.

The end result of this chapter is a Web Service that can be invoked by the .NET Web Service client created in Chapter 11, ".NET client" on page 243 and the WebSphere Application Server Web Service client created in Chapter 13, "WebSphere Application Server client" on page 289. Interoperation with Axis is achieved using the WSDL generated from Chapter 8, "Axis Web Service" on page 159.

© Copyright IBM Corp. 2006. All rights reserved.

12.1 Design

This section discusses the design of a simple WebSphere Application Server Web Service that is used to demonstrate the WebSphere MQ transport for SOAP.

Web Service design

The Web Services design is the same as that detailed in Chapter 10, ".NET Web Service" on page 213 because it is derived from the WSDL produced there. WSDL is the key to the interoperability of Web Services, and by using this WSDL, any Web Service client that also uses this WSDL can invoke this Web Service.

The BankingService Web Service is designed to model a bank account and the common operations that take place on it. The design of the Web Service is kept simple. However, it demonstrates returning simple data types and complex data types, and throwing exceptions that ultimately become SOAP faults. The methods are detailed in Table 12-1.

Table 12-1 BankingService method description

Method	Description
debit	Removes specified amount for transfer to the account ID provided. This implementation simply subtracts the amount specified from the balance. If the amount is greater than the balance, an exception is thrown.
credit	Adds specified amount to current balance
getBalance	Returns the current balance
getStatement	Returns an array of BankOperation objects

Note: Static variables are used to maintain state.

As with most WebSphere Application Server applications, the Web Service resides inside an enterprise archive (EAR) file. There are various Web Service types available in WebSphere Application Server. This chapter uses the Enterprise JavaBeans (EJB) Web Service. See the WebSphere Application Server documentation for information about the other types.

SOAP/JMS works in a similar manner to SOAP/WebSphere MQ. In the WebSphere Application Server, SOAP/JMS supports the invocation of multiple services through a single client invocation. Although this chapter does not demonstrate this, it does focus on the equivalent that is supported by the

WebSphere MQ SOAP transport. Figure 12-1 shows the components of the SOAP/JMS transport.

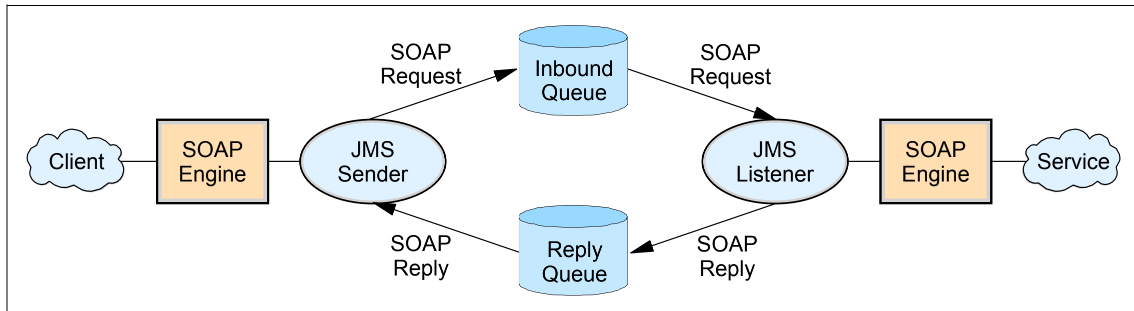


Figure 12-1 SOAP/JMS components

With SOAP/JMS, the JMS listener is a message-driven bean (MDB). The arrival of an invocation message from a Web Service client kicks off the MDB, which in turn invokes the service. The MDB then takes the response and sends it to the replyDestination. You do not have to be concerned about the implementation of this MDB because it is already implemented by the SOAP/JMS transport classes. However, it must be given a router project inside the EAR file. Create this router project as described in 12.3, “Implementation” on page 272.

WebSphere MQ design

Unlike the other chapters that demonstrate the use of the deploy tool to facilitate different WebSphere MQ configurations, this chapter simply shows the use of WebSphere MQ when configured for client binding.

WebSphere Application Server provides a namespace where the JMS-administered objects required by the SOAP transport may be looked up. In this chapter, WebSphere MQ V6 JMS is used. Currently, WebSphere Application Server V6.0.2 does not support the configuration of WebSphere MQ V6 JMS-administered objects through the WebSphere MQ messaging resources. In order to use WebSphere MQ V6, it must be configured as a generic provider. The Web Service uses two QueueConnectionFactory objects, one for receiving a request and one for sending the response. A single queue is required for the request.

The use of a namespace means changing the WebSphere MQ configuration, for example, to use bindings is trivial. Simply update the definitions of the JMS-administered objects.

12.2 Requirements

To generate the Web Service skeleton from the WSDL and deploy this to WebSphere Application Server in an EAR file, the following products are required:

- Rational Application Developer V6.0.0.4.3
- WebSphere Application Server Version 6.0.2

> **Note:** WebSphere Application Developer may also be used when writing Web Service applications for WebSphere Application Server V5.

12.3 Implementation

This section discusses the implementation of a Web Service from WSDL. This involves creating appropriate projects in Rational Application Developer, importing the WSDL, and using it to generate a skeleton Web Service. This section also provides information about exporting the code to an EAR file. The result is available for download in Appendix D, “Additional material” on page 431. This section also discusses the WebSphere MQ and WebSphere Application Server environment setup that is required before the Web Service is deployed.

12.3.1 Creating and implementing the Web Service skeleton

To implement the service, import the WSDL from Chapter 10, “.NET Web Service” on page 213 into an Enterprise Application Project. Using this WSDL, Rational Application Developer generates a skeleton implementation. This WSDL is available for download in Appendix D, “Additional material” on page 431.

To create and implement the Web Service skeleton, perform the following tasks:

1. In Rational Application Developer, create the project by selecting from the menu, **File** → **New** → **Enterprise Application Project**. Call the project `BankingService`. Two new Enterprise JavaBeans (EJB) module projects are required, one for the Web Service, and one for the router project. Click **New Module**.
2. Leave Create default module projects selected, but deselect all but the EJB project check boxes, leaving the name of this project as BankingServiceEJB. Click **Finish**.

3. Click **New Module** again. Leave the Create default module projects selected and deselect all but the EJB project check boxes. This time, rename the default to `BankingServiceEJBRouter`. Click **Finish**. Figure 12-2 shows the result.

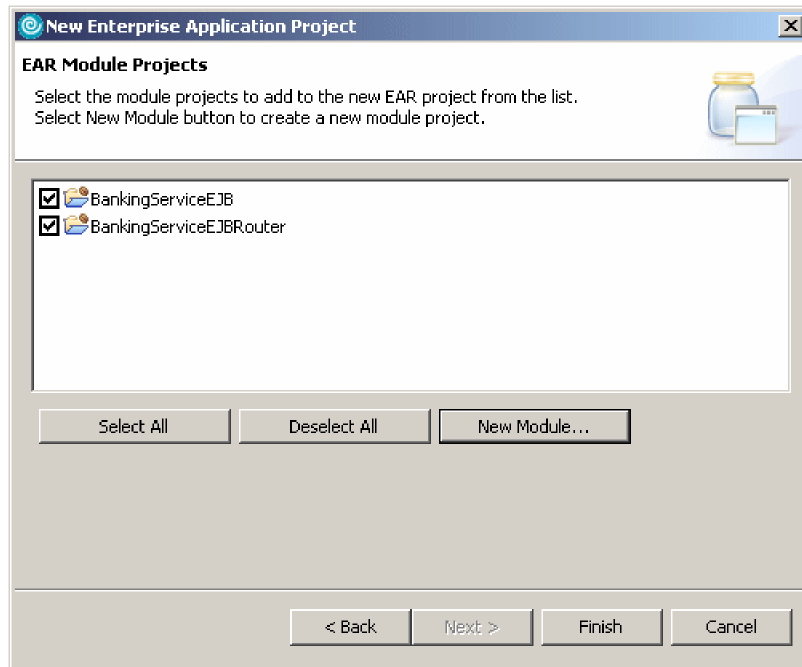


Figure 12-2 Web Service and Router EJB project

4. Import the WSDL into the BankingServiceEJB project. Right-click this project after selecting it from **Project Explorer** → **Import...** → **Import**.
5. Select **File system** and browse to and select the directory containing the WSDL. Check the box next to the WSDL file and click **Finish**.

> **Attention:** If the Rational Application Developer version that is used generates an error in the WSDL definition of the BankOperation array, refer to the latest Problem Management Record (PMR).

6. The WSDL generated by WebSphere MQ leaves the soap:binding transport attribute set to the Hypertext Transfer Protocol (HTTP) transport. It does not change it to JMS transport. This facilitates ease-of-use and promotes

interoperability. Since the SOAP/JMS transport is being used, it must be changed to allow a JMS endpoint address in WebSphere Application Server.

a. Double-click the WSDL file.

b. In the editor, find the soap:binding element and the associated transport attribute. Change this transport attribute from http://schemas.xmlsoap.org/soap/http to `http://schemas.xmlsoap.org/soap/jms`. Figure 12-3 shows this change.

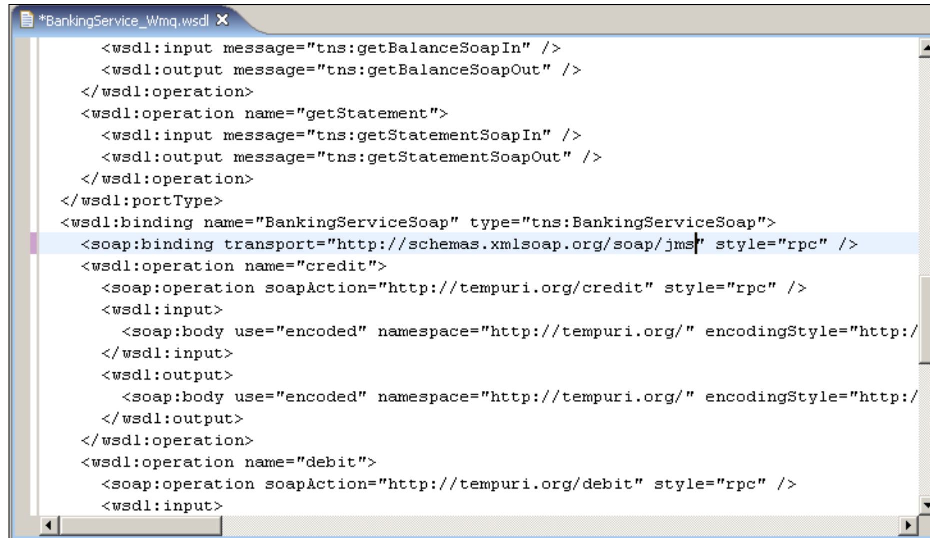


Figure 12-3 Changing the soap:binding transport attribute

7. Start generating the Web Service skeleton. By default, Web Services Development capability is off. Because this capability is required, enable it as follows:

   a. Select **Window** → **Preferences**.

   b. Expand **Workbench** and select **Capabilities**.

   c. In the right-hand pane, expand **Web Service Developer** and check **Web Services Development**.

Attention: If the Rational Application Developer version that is used generates an error from the Web Service wizard about an EJB instance being absent from the BankingServiceEJB project, refer to the latest Problem Management Record (PMR) that fixes this issue. To work around this issue, create an EJB:

1. Right-click **BankingServiceEJB** project and select **New** → **Other**.
2. Expand **EJB**.
3. Select **Enterprise Bean**.
4. Enter a Bean name, `Dummy`, and click **Finish**.
5. Close the **dnx file editor**.

6. In the Project Explorer, right-click the WSDL file, select **Web Services** → **Generate Java bean skeleton**. Select **Skeleton EJB Web Service** from the drop-down box. Deselect **Start Web Service in Web project** and select **Create folders when necessary**. Click **Next**.
7. In the Object Selection page, click **Next**.
8. The options specified in the Service Deployment Configuration must be correct, given the location of the WSDL in the workspace. Therefore, click **Next**.
9. In the Web Service skeleton EJB Configuration page, ensure that the router project created earlier is selected as the Router Project. SOAP/JMS must already be selected as the transport. The Connection factory, Destination,

and MDB deployment mechanism must be corrected. Figure 12-4 shows how it is configured. Click **Next**.

WSDL folder /BankingServiceEJB/ejbModule/wsdl
WSDL file name BankingService_Wmq.wsdl

Select Router Project BankingServiceEJBRouter
Security Configuration No Security

Define custom mapping for namespace to package.

Select transports:
SOAP over HTTP
SOAP over JMS

JMS URI Properties
Queue kind queue
WSDL service name BankingService.asmx
Connection factory jms/BankingServiceQCF
Destination jms/BankingServiceQueue
MDB deployment mechanism Listener Port
ActivationSpec JNDI name
Listener port name BankingServiceListenerPort

Figure 12-4 Web Service skeleton configuration page

10.Click **Finish**.

The skeleton is created in the BankingServiceEJB project, and the Java editor opened with the skeleton to be implemented, that is, BankingServiceSoapImpl.java.

Attention: The EJB 2.0 specification no longer permits the throwing of java.rmi.RemoteException from the user methods defined in the EJB's remote or home interface. Consequently, a warning is given for each of the methods in BankingServiceSoapImpl.java. However, no such exceptions are illustrated here.

The actual implementation of the skeleton that is available for download has the same characteristics as the Web Services in Chapter 10, ".NET Web Service" on page 213 and Chapter 8, "Axis Web Service" on page 159.

> **Tip:** Exception throwing is slightly different in Java and .NET. In .NET, there is no concept of checked or unchecked exceptions. Therefore, Java style does not have to throw a clause on the method signature. Consequently, the .NET BankOperationException class used in Chapter 10, ".NET Web Service" on page 213 is not described in the WSDL, and the skeletons generated have no such equivalent. To achieve a similar behavior, this implementation uses a java.lang.RuntimeException. If the WSDL from the Axis Web Service in Chapter 8, "Axis Web Service" on page 159 is used, a skeleton implementation of the BankOperationException.java must have been created. Using this, it is possible to throw user exceptions on service methods that are serialized in a SOAP fault, and reserialized in the client and thrown there, correctly typed.

Assuming the skeleton is implemented, the EAR file can be generated from the Project Explorer by performing the following tasks:

1. Right-click **BankingService Enterprise Application Project** and select **Export..**. → **EAR File**.
2. Choose a destination for the EAR file and call it `BankingService.ear`.

12.3.2 WebSphere MQ and WebSphere Application Server setup

For the Web Service to work, both WebSphere MQ and WebSphere Application Server must be configured appropriately. This section details the setup required for each.

WebSphere MQ setup

For a WebSphere MQ setup, perform the following tasks:

1. Set up a queue manager called `QM_WAS`.
2. Because the JMS resources use the CLIENT binding to connect to WebSphere MQ, a SVRCONN is required. Run the Message Channel Agent (MCA) under a user name with appropriate authority. For purposes of simplicity, in this example, mqm is used.

> **Important:** Using mqm for the MCA is not necessarily a good idea in a production environment. Details about how security must be approached are too involved to be addressed in this simple scenario. Consult WebSphere MQ security documentation for information about the correct approach to channel security and how to use the MCA user.

To create the SVRCONN, start **runmqsc** with a user name that has the appropriate authority and create a SVRCONN channel and a request queue. Example 12-1 shows you how to do this.

Example 12-1 runmqsc command to create SVRCONN channel and request queue

```
$ runmqsc QM_WAS
5724-H72 (C) Copyright IBM Corp. 1994, 2005.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM_WAS.


DEFINE CHL(WAS.JMS.SVRCONN) CHLTYPE(SVRCONN) MCAUSER('mqm')
     1 : DEFINE CHL(WAS.JMS.SVRCONN) CHLTYPE(SVRCONN) MCAUSER('mqm')
AMQ8014: WebSphere MQ channel created.
DEFINE QL('BANKING.SERVICE.REQUEST')
     2 : DEFINE QL('BANKING.SERVICE.REQUEST')
AMQ8006: WebSphere MQ queue created.
END
     3 : END
2 MQSC commands read.
No commands have a syntax error.
All valid MQSC commands were processed.
```

The queue manager is configured.

3. Start a WebSphere MQ TCP listener in a terminal. Use port number 1414. From a shell, use the following command:

```
runmqlsr -m QM_WAS -t TCP -p 1414 &
```

> **Tip:** Running the WebSphere MQ TCP listener in the background may be more convenient, but in production environments, it is best to run it in its own shell. This is because the listener can output information to stdout and the WebSphere MQ error logs in the event of a TCP error.

In “WebSphere Application Server setup” on page 281, WebSphere MQ is configured as an external JMS provider. This allows names of JMS objects in the WebSphere Application Server namespace to be mapped to names, and the real definitions of the JMS objects in another namespace, typically managed by the specific JMS providers’ namespace implementation. In WebSphere MQ, JMSAdmin is used to define the required connection factories and queues.

Although this example uses the Java file system context, any other context implementation may be used. The file system context provides a persistent store that is always available on the WebSphere Application Server machine. However, this precludes the ability to use these resources at the cell scope.

> **Tip:** Placing the file system context in a Storage Area Network facilitates the use of cell scope.

To bind JMS objects into this namespace, perform the following tasks:

1. Create a JMSAdmin configuration file. A basic configuration file is shown in Example 12-2.

Example 12-2 Basic configuration file

```
#Set the service provider
   INITIAL_CONTEXT_FACTORY=com.sun.jndi.fscontext.RefFSContextFactory
#Set the initial context
   PROVIDER_URL=file:///home/wasuser/jms/
#Set the authentication type
   SECURITY_AUTHENTICATION=none
```

2. Create a directory, /home/wasuser/jms. A .bindings file is created by the file system context to store objects bound into the namespace. This demonstration assumes that the configuration file is saved as /home/wasuser/jms.cfg, where wasuser is simply a pseudo name for the user name that WebSphere Application Server runs under. The user name can be any user and the location can be anywhere.
3. Ensure that the following jars are in the CLASSPATH:
   - com.ibm.mq.jar
   - com.ibm.mqjms.jar
   - fscontext.jar (shipped with WebSphere MQ under java/lib)
4. Use JMSAdmin to bind two connection factories.

   Web Service requires two connection factories and one queue. The next section explains what each is used for. For now, assume that they are required. The connection factories must be XA-enabled because the message-driven bean (MDB) consumes the request messages in a

transaction. Example 12-3 shows the steps involved in doing this. Because these objects are in the file system namespace, they can be referenced from the WebSphere Application Server namespace.

Example 12-3 Using JMSAdmin to bind two connection factories

```
# cd /usr/mqm/java/bin
$ export
CLASSPATH=/usr/mqm/java/lib/com.ibm.mq.jar:/usr/mqm/java/lib/com.ibm.mqjms.jar:/usr/m
qm/java/lib/fscontext.jar
$ ./JMSAdmin -cfg /home/wasuser/jms.cfg

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

InitCtx> DEF XAQCF(BankingServiceQCF) QMGR(QM_WAS) TRANSPORT(CLIENT)
HOSTNAME(9.1.39.93) PORT(1414) CHANNEL(WAS.JMS.SVRCONN)

InitCtx> DEF XAQCF(WebServicesReplyQCF) QMGR(QM_WAS) TRANSPORT(CLIENT)
HOSTNAME(9.1.39.93) PORT(1414) CHANNEL(WAS.JMS.SVRCONN)

InitCtx> DEF Q(BankingServiceQueue) QMANAGER(QM_WAS) QUEUE(BANKING.SERVICE.REQUEST)

InitCtx> dis ctx

  Contents of InitCtx

      .bindings                java.io.File
   a  WebServicesReplyQCF      com.ibm.mq.jms.MQXAQueueConnectionFactory
   a  BankingServiceQueue      com.ibm.mq.jms.MQQueue
   a  BankingServiceQCF        com.ibm.mq.jms.MQXAQueueConnectionFactory

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

InitCtx> END

Stopping Websphere MQ classes for Java(tm) Message Service Administration
```

WebSphere Application Server setup

A basic Web Service using the SOAP/JMS transport in WebSphere Application Server requires two connection factories, one for the MDB to listen to the invocation destination, and one for the MDB to send the response. A single destination is required for the invocation. The MDB also requires a listener port.

> **Note:** Listener port is an overloaded phrase. It can be the WebSphere MQ TCP listener port number that is used to listen for TCP connections over channels to the queue manager, or it can be the component of the MDB that listens on a queue for incoming messages.

The MDB listener uses a resource environment reference of java:comp/env/jms/WebServicesReplyQCF to send the response. Link this to an actual QueueConnectionFactory Java Naming Directory Interface (JNDI) name during the EAR file install time.

In order to create these JMS-administered objects and bind them into the WebSphere Application Server namespace, configure WebSphere MQ V6 as a generic provider.

> **Note:** WebSphere MQ V6 can also be configured as the WebSphere MQ JMS Provider in the WebSphere Application Server AdminConsole. This allows WebSphere Application Server to cache and manage the JMS-managed objects. For instructions about how to do this, refer to the information center on the Web at:
>
> http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.express.doc/info/exp/ae/tmj_instm.html

Binding generic JMS-administered objects into Java Naming and Directory Interface

The admin console supports configuring and binding WebSphere MQ JMS objects into the WebSphere Application Server namespace. However, in WebSphere Application Server V6.0.2, this is restricted to WebSphere MQ V5.3 JMS objects only. To bind WebSphere MQ V6 JMS objects, a generic provider must be configured.

> **Note:** Using WebSphere MQ V5.3, JMS classes with SOAP/JMS is supported. There is, however, a limitation, in that, the V5.3 JMS classes cannot connect to a 64-bit queue manager.

To configure WebSphere MQ V6 as a generic provider, perform the following tasks:

1. From the admin console, expand **Resources**, **JMS Providers**.
2. Select **Generic**, and choose the server scope. server1 is the default profile.
3. Click **New**.
4. In the window that opens, perform the following actions:
   - Specify WebSphere MQ as the provider by entering `WebSphere MQ 6` as the name of the provider.
   - Add the following jar files, including the full path, to the class path text box:
     - com.ibm.mq.jar
     - com.ibm.mqjms.jar
     - dhbcore.jar
   - If bindings mode is to be used, the native library path must be MQ_HOME/java/lib.
   - In the initial context factory field, specify `com.sun.jndi.fscontext.RefFSContextFactory`.
   - Finally, the external provider URL must be the URL used in “WebSphere MQ setup” on page 277.

   Click **OK** and save the configuration.
5. After it is saved, go back to view the newly configured provider and expand **Resources, JMS Providers**. Select **Generic** and then the provider

WebSphere MQ 6. Figure 12-5 shows what this window looks like on an AIX machine.

General Properties

* Scope

cells:kodiakNode01Cell:nodes:kodiakNode01:servers:server1

* Name

WebSphere MQ 6

Description

Class path

/usr/mqm/java/lib/com.ibm.mq.jar
/usr/mqm/java/lib/com.ibm.mqjms.jar
/usr/mqm/java/lib/dhbcore.jar

Native library path

/usr/mqm/java/lib

* External initial context factory

com.sun.jndi.fscontext.RefFS

* External provider URL

file:///home/wasuser/jms/

Apply | OK | Reset | Cancel

Additional Properties

- Custom properties
- JMS connection factories
- JMS destinations

Figure 12-5 WebSphere MQ V6 as a generic provider

Now that the provider is configured, define the JMS objects. Because a generic JMS provider is being used, the definitions of these objects are already specified in “WebSphere MQ setup” on page 277. For WebSphere Application Server applications to look up these objects, a name is bound into the WebSphere Application Server namespace that maps to the actual definition in the file system context setup in “WebSphere MQ setup” on page 277.

To define JMS objects, perform the following tasks:

1. The QCF required for the reply. In the window shown in Figure 12-5, select **JMS connection factories** → **New**.
   – Enter `WebServicesReplyQCF` against the Name field.
   – The JNDI name must be `jms/WebServicesReplyQCF`.

   – The External JNDI name must be the name of QCF in JMSAdmin, `WebServicesReplyQCF`.

   Click **OK**.

2. Repeat these steps for the QCF used by the MDB to listen to the queue, specifying the following names for the three fields:
   – Name: `BankingServiceQCF`
   – JNDI name: `jms/BankingServiceQCF`
   – External JNDI name: `BankingServiceQCF`
3. Specify the queue used for the invocation by navigating back to the page shown in Figure 12-5. Select **JMS destinations** → **New**. Use the following names for the three fields:
   – Name: `BankingServiceQueue`
   – JNDI name: `jms/BankingServiceQueue`
   – External JNDI name: `BankingServiceQueue`
4. Finally, configure the listener port for the MDB. This uses the jms/BankingServiceQCF and jms/BankingServiceQueue JMS-administered objects.
   a. Expand **Servers** → **Application servers**.
   b. Select **server1** and under Communications, expand **Messaging** → **Message Listener Service**.
   c. Select **Listener Ports** → **New**.
      - Call the listener port `BankingServiceListenerPort`.
      - Provide the JNDI names of the QCF and the queue, `jms/BankingServiceQCF` and `jms/BankingServiceQueue` respectively.
      - Accept the default for the remaining fields.

   Select **OK**. Save the configuration.

12.3.3 Deployment

Now that the Web Service is implemented, the EAR file created, the WebSphere Application Server and WebSphere MQ configured, the EAR file is ready to be deployed on a WebSphere Application Server.

To deploy the EAR file on a WebSphere Application Server, perform the following tasks:

1. Open the admin console and select **Applications** → **Install New Applications**. Select either the local file system or the remote file system depending on where the EAR file resides, and specify its location. Click **Next**.

2. Select **Generate Default Bindings**. Click **Next**.

> **Note:** Accept the defaults for step 1 and step 2.

3. In the Provide listener bindings for message-driven bean page, select the **listener port** option. This must already have the listener port name. Click **Next**.
4. In the window that opens, accept the defaults and click **Next**.

> **Attention:** The Provide JNDI Names for Beans page shows that the EJB is used as a workaround as described in 12.3.1, “Creating and implementing the Web Service skeleton” on page 272. Ignore this.

5. In the window that opens, accept the defaults by clicking **Next**. Ignore any warnings about deployment scope and click **Continue**.
6. In the window that opens, accept the defaults and click **Next**.
7. In the window that opens, accept the defaults and select **Next**.
8. Select **Finish**.

The Web Service is deployed to the WebSphere Application Server and is ready to be invoked. Restart the WebSphere Application Server server to bind the JMS-administered objects into the namespace before the BankingService application starts.

This Web Service can be invoked by the .NET client. For more details, refer to Chapter 11, “.NET client” on page 243.

12.4 Security

Using SSL for security with WebSphere Application Server using WebSphere MQ JMS is similar to using SSL for security in plain WebSphere MQ JMS. It is up to the definition of the QCF in JMSAdmin to specify the appropriate options for SSL. The queue manager must also be configured to work with SSL. Refer to Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 for details about this.

WebSphere MQ clients, like WebSphere MQ SOAP and WebSphere MQ JMS (SOAP/JMS as well) clients, require connect options that enable SSL. For Java, these are:

- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword
- sslCipherSuite
- sslPeerName

More options are available. However, this subset is used to demonstrate the processes involved in using them.

Before SSL is used, the key repositories and certificate chains must be configured. As with WebSphere MQ, the WebSphere Application Server uses the iKeyman utility to do this. Consult either the WebSphere Application Server SSL documentation or Chapter 6, “Security” on page 107 for more information.

Setting up Secure Sockets Layer in the WebSphere Application Server

The definition of the QCFs in JMSAdmin or in the admin console if you are using WebSphere MQ V5.3, can have the sslCipherSuite and sslPeerName options set on them, that is, SSLCIPHERSUITE and SSLPEERNAME, when defining the QCF in JMSAdmin. There are other SSL attributes available on the QCF, but this section does not provide details about them.

Define the four key store and trust store options to the Java Virtual Machine (JVM) using generic JVM arguments. To do this, perform the following tasks:

1. From the admin console, select **Servers** → **Application servers**.
2. Under Server Infrastructure, expand **Java and Process Management** and select **Process Definition**.
3. Select **Java Virtual Machine** and add the following properties to the Generic JVM arguments text field, with multiple arguments separated by a space:
   - `-Djavax.net.ssl.keyStore=xxx`
   - `-Djavax.net.ssl.keyStorePassword=xxx`
   - `-Djavax.net.ssl.trustStore=xxx`
   - `-Djavax.net.ssl.trustStorePassword=xxx`

> **Note:** These properties are set for the entire JVM. Consider the implications for other SSL-enabled components.

4. Restart the server for these properties to take effect.

After completing the configuration and restarting the server, SSL secures messaging with WebSphere MQ.

12.5 Summary

This chapter provided information about implementing a Web Service from a WSDL that uses SOAP/JMS transport. It also described the configuration of WebSphere Application Server V6 to use WebSphere MQ V6 as a generic JMS provider. Apart from this, there is no WebSphere MQ-specific information here. However, this chapter demonstrated the interoperability provided by WebSphere MQ transport for SOAP.

13

WebSphere Application Server client

This chapter describes the implementation of a WebSphere Application Server SOAP/Java Message Service (JMS) Web Service client so that it interoperates with WebSphere MQ transport for SOAP. This chapter assumes that you have a certain level of understanding about WebSphere Application Server and JMS.

As with Chapter 12, “WebSphere Application Server Web Service” on page 269, this chapter uses the Web Services Description Language (WSDL) generated by the deploy tool in Chapter 10, “.NET Web Service” on page 213. This is available for download in Appendix D, “Additional material” on page 431.

The transport used by the WebSphere Application Server Web Service client is SOAP/JMS. This is an existing functionality in WebSphere Application Server. As detailed in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49, WebSphere MQ provides a messaging bus that is capable of connecting SOAP/JMS in WebSphere Application Server with WebSphere MQ and Customer Information Control System (CICS) using SOAP/WebSphere MQ.

The end result of this chapter is a Web Service client invoking the WebSphere Application Server Web Service implemented in Chapter 12, “WebSphere Application Server Web Service” on page 269 and the .NET Web Service implemented in Chapter 10, “.NET Web Service” on page 213.

© Copyright IBM Corp. 2006. All rights reserved.

The platform used to run WebSphere Application Server is AIX 5.2, but the instructions are general enough to be used on any other supported platform. The platform used to run Rational Application Developer is Windows.

13.1 Design

Unlike the other client chapters, this chapter does not use a graphical user interface (GUI) to demonstrate the function. Instead, it gives a basic demonstration that the service can be invoked. This consists of three simple operations:

- Crediting the account with $10
- Debiting the account with $3
- Getting and displaying a statement

An enterprise archive (EAR) is used to encapsulate the client. This is known as the client container. A Web Service can be invoked by a client in any of the following WebSphere Application Server containers:

- Client container
- Web container
- Enterprise JavaBeans (EJB) container

This demonstration uses the client container for simplicity, although proxies may be generated for use in either of the other containers or a stand-alone Java application.

The client proxies are generated from the WSDL and added to the EAR as an application client. These proxies use the SOAP/JMS classes. This transport uses dynamic queues for the response if no replyDestination is specified in the Universal Resource Indicator (URI).

Using the same WSDL to generate the proxies is the key to the interoperability of Web Services. The decision to use the WSDL from the .NET service is not only a good example of the heterogeneous capabilities of Web Services, but also demonstrates the interoperation with a SOAP infrastructure that requires the SOAPAction header. This header must be present in the transport of the SOAP request message in order to invoke a .NET Web Service. For more information about SOAPAction, see Chapter 4, “WebSphere Services with WebSphere MQ” on page 29 and Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

In order to interoperate with the Axis service fully, follow a similar process by using the WSDL from the Axis service. Alternatively, use the proxies generated by the .NET Web Service deploy.

13.2 Requirements

To generate the Web Service client proxies from the WSDL and implement and start the client, the following products are required:

- Rational Application Developer V6.0.0.4.3
- WebSphere Application Server V6.0.2

> **Note:** WebSphere Application Developer may also be used when writing Web Service applications for WebSphere Application Server V5.

13.3 Implementation

This section demonstrates how to implement a WebSphere Application Server Web Service client from WSDL. This involves creating appropriate projects in Rational Application Developer, importing the WSDL into the workspace, and using it to generate proxy code. The result is available for download in Appendix D, “Additional material” on page 431.

An Enterprise Application Project with a Client Application Project is required to house the proxies and the main class that constitutes the Web Service client code.

> **Note:** If the Web Service is already implemented as described in Chapter 12, “WebSphere Application Server Web Service” on page 269, add the Client Application Project to the BankingService project. The WSDL does not have to be imported again.

To implement a WebSphere Application Server Web Service client from WSDL, perform the following tasks:

1. From the Project Explorer, select **Enterprise Applications** → **New** → **Enterprise Application Project**. Call the project `BankingClient`. Click **Next**.
2. Click **New Module**. Leave the Create default module projects checked and deselect all but the Application Client project. Call it `BankingServiceClient`. Click **Finish** to create the project.
3. From the Project Explorer, expand **Application Client Projects**. Right-click the newly created **BankingServiceClient** project and select **Import** → **Import...**

4. Select **File system** and click **Next**.
5. Browse and select the directory that contains the WSDL generated in Chapter 10, “.NET Web Service” on page 213. (This is available for download in Appendix D, “Additional material” on page 431.) Select the box next to the WSDL file name, as shown in Figure 13-1. Click **Import**.

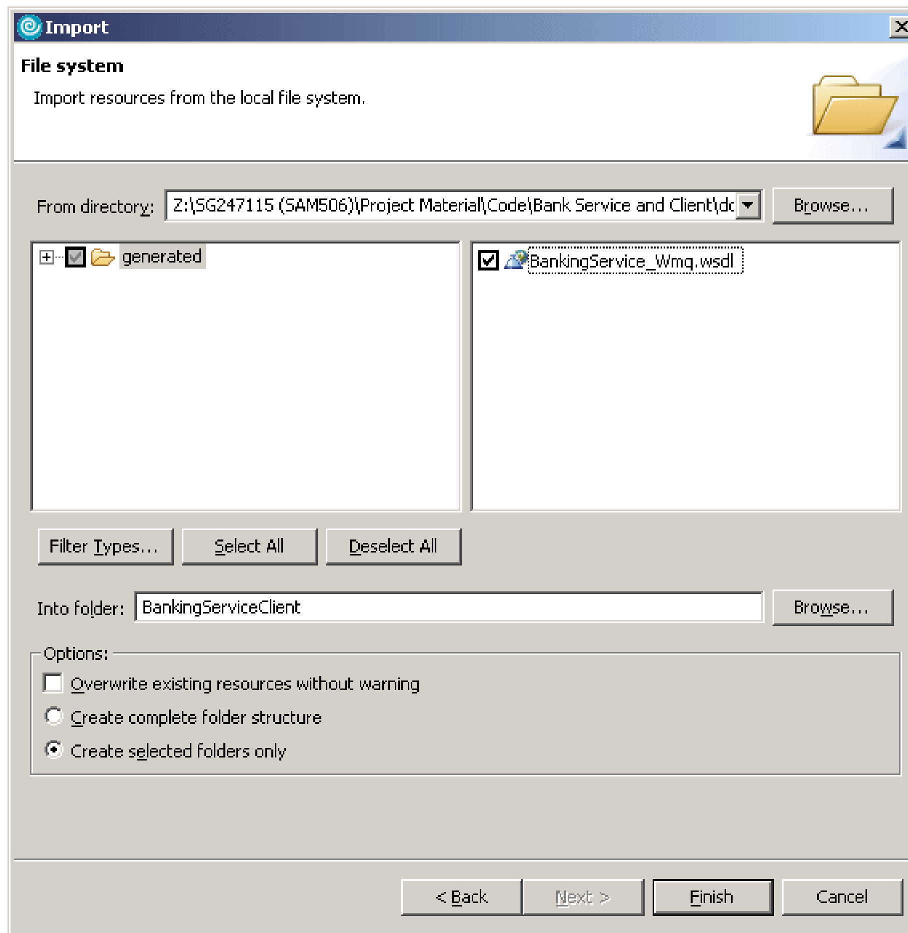


Figure 13-1 Importing the WSDL

Attention: If the Rational Application Developer version that is used generates an error in the WSDL definition of the BankOperation array, refer to the latest Problem Management Record (PMR).

6. Right-click the WSDL file, select **Web Services** → **Generate Client**. Leave Test the Web Service unchecked and click **Next**.
7. In the Web Service Selection Page, click **Next**.
8. In the Client Environment Configuration page, choose **Application Client** from the Client type drop-down box. In the Client project drop-down box, select the Client Application Project created earlier, **BankingServiceClient**. The EAR project drop-down box must automatically select the BankClient project. Click **Finish**.

   The BankingServiceClient project has a package called org.tempuri. In this package, the client proxy classes exist. By using these proxies to invoke the Web Service, the Main.java class can be implemented. The downloadable implementation of this class is a basic demonstration of the functionality of the Web Service.
9. Assuming Main.java is implemented, export the EAR file by performing the following tasks:
   a. From the Project Explorer, right-click the **BankingClient** project under Enterprise Applications.
   b. Select **Export**... → **EAR file**. Enter a suitable destination.

> **Note:** When invoking the client, the EAR file must be on the machine where the WebSphere Application Server is installed.

13.3.1 WebSphere MQ setup

A Web Service must be implemented for the client to work. Whether this Web Service is the one described in Chapter 10, ".NET Web Service" on page 213 or Chapter 12, "WebSphere Application Server Web Service" on page 269 does not matter. The WebSphere MQ configuration from either chapter is sufficient.

13.4 Deployment

After the client is created and compiled into an EAR file, it can be run in the client container provided by the WebSphere Application Server.

The implementation of the client that is available for download uses an URI that is passed in as argument in double quotes. If no argument is passed, a hard-coded URI is used as shown in Example 13-1.

Example 13-1 Hard-coded URI

```
jms:/queue?destination=jms/BankingServiceQueue&connectionFactory=jms/Ba
nkingServiceQCF&targetService=BankingServiceSoap
```

This URI causes the SOAP/JMS transport to use the same QueueConnectionFactory (QCF) and queue definitions as the message-driven bean (MDB) does. This separation means that coping with the reconfiguration of the messaging bus and moving the location of the Web Service from the Web Service client perspective is trivial.

Tip: The targetService option in the URI is determined by the port component in the Web Service deployment descriptor. In this example, this is derived from the wsdl:portType name attribute in the imported WSDL used to generate the skeleton implementation of the Web Service in Chapter 12, "WebSphere Application Server Web Service" on page 269.

If the Web Service is already implemented as described in Chapter 12., "WebSphere Application Server Web Service" on page 269, perform the following tasks:

- Expand **BankingServiceEJB project**, **ejbModule**, **META-INF**.
- Double-click **webservice.xml**.
- Select the **Port Components** tab and note the name of the Port Component defined, BankingServiceSoap. This is the targetService.

  The use of any other value causes WebSphere Application Server to complain that it is unable to find the requested target service.

To invoke the client application, use the WebSphere Application Server Java 2 Platform, Enterprise Edition (J2EE) Application Client tool, launchClient. On UNIX platforms, issue the following command from the WAS_HOME/profiles/default/bin directory:

```
./launchClient.sh /full_path_to/BankingClient.ear
```

The result of invoking the client is displayed, demonstrating the basic function of the Web Service. Example 13-2 shows the output.

Example 13-2 BankingClient.ear output

```
$ pwd
/usr/IBM/WebSphere/AppServer/profiles/default/bin
$ ./launchClient.sh /home/wasuser/BankingClient.ear
IBM WebSphere Application Server, Release 6.0
J2EE Application Client Tool
Copyright IBM Corp., 1997-2004
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has completed.
WSCL0014I: Invoking the Application Client class Main

Crediting the account with 10...done.
Debiting the account with 3...done.
Balance = 35.0

Date / Time            Operation Type         Account         Amount

13:6:2005 19:39:53            Credit          0               10.0
13:6:2005 19:39:53            Debit           1234            3.0
14:6:2005 20:15:9             Credit          0               10.0
14:6:2005 20:15:9             Debit           1234            3.0
14:6:2005 20:16:22            Credit          0               10.0
14:6:2005 20:16:23            Debit           1234            3.0
14:6:2005 20:18:12            Credit          0               10.0
14:6:2005 20:18:12            Debit           1234            3.0
14:6:2005 20:19:56            Credit          0               10.0
14:6:2005 20:19:57            Debit           1234            3.0
```

Using Nojndi to invoke the .NET Web Service

As described in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49, Nojndi is an InitialContextFactory that allows the reuse of the WebSphere MQ URI, for example, as specified when using the deploy tool. To use Nojndi in this case, it is simply a matter of setting the options on the URI in such a way that they are equivalent to the jms/BankingServiceQCF and jms/BankingServiceQueue. However, this is not the intended use of Nojndi. Instead, Nojndi comes into its own when connecting to Web Services that are hosted by WebSphere MQ or the CICS Transaction Server.

To invoke the .NET Web Service implemented in Chapter 10, ".NET Web Service" on page 213, for example, pass an URI to the BankingClient application. Example 13-3 shows how to do this.

Example 13-3 Passing an URI to the BankingClient application

```
./launchClient.sh /full_path_to/BankingClient.ear
"jms:/queue?destination=BANKING.SERVICE.REQUEST.QUEUE@QM_LocalToSvc&con
nectionFactory=(connectQueueManager(QM_LocalToSvc)binding(client)client
Channel(SYSTEM.DEF.SVRCONN)clientConnection(9.1.39.128%25281414%2529))&
initialContextFactory=com.ibm.mq.jms.Nojndi&targetService=BankingServic
e.asmx&replyDestination=BANKING.SERVICE.RESPONSE"
```

Example 13-4 shows the output.

Example 13-4 BankingClient.ear output using Nojndi

```
$ pwd
/usr/IBM/WebSphere/AppServer/profiles/default/bin
$ ./launchClient.sh /home/wasuser/BankingClient.ear
"jms:/queue?destination=BANKING.SERVICE.REQUEST.QUEUE@QM_LocalToSvc&connectionFactory
=(connectQueueManager(QM_LocalToSvc)binding(client)clientChannel(SYSTEM.DEF.SVRCONN)c
lientConnection(9.1.39.128%25281414%2529))&initialContextFactory=com.ibm.mq.jms.Nojnd
i&targetService=BankingService.asmx&replyDestination=BANKING.SERVICE.RESPONSE"
IBM WebSphere Application Server, Release 6.0
J2EE Application Client Tool
Copyright IBM Corp., 1997-2004
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has completed.
WSCL0014I: Invoking the Application Client class Main
Using
jms:/queue?destination=BANKING.SERVICE.REQUEST@QM_LocalToSvc&connectionFactory=(conne
ctQueueManager(QM_LocalToSvc)binding(client)clientChannel(SYSTEM.DEF.SVRCONN)clientCo
nnection(9.1.39.128%25281414%2529))&initialContextFactory=com.ibm.mq.jms.Nojndi&targe
tService=BankingService.asmx&replyDestination=BANKING.SERVICE.RESPONSE

Crediting the account with 10...done.
Debiting the account with 3...done.
Balance = 14.0
```

```
Date / Time              Operation Type         Account         Amount
------------------------------------------------------------------------
15:6:2005 15:47:21              Debit           1234            3.0
15:6:2005 15:46:39              Debit           1234            3.0
15:6:2005 15:47:21              Credit          0               10.0
```

Tip: Reconfiguring the JMS-administered objects using JMSAdmin allows the Web Service client to invoke the Web Service in another location.

13.5 Security

Using Secure Sockets Layer (SSL) for security with WebSphere Application Server using WebSphere MQ JMS is similar to using SSL for security in plain WebSphere MQ. It is up to the QCF definition in JMSAdmin to specify the appropriate options for SSL. The queue manager must also be configured to work with SSL. Refer to Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 for more information about this.

WebSphere MQ clients, like WebSphere MQ SOAP and WebSphere MQ JMS (including SOAP/JMS) clients, require connect options that enable SSL. For Java, these are:

- sslKeyStore
- sslKeyStorePassword
- sslTrustStore
- sslTrustStorePassword
- sslCipherSuite
- sslPeerName

Although more options are available, this subset is used to demonstrate how to use them.

Before SSL is used, the key repositories and certificate chains must be configured. As with WebSphere MQ, the WebSphere Application Server uses the iKeyman utility to do this. Consult either the WebSphere Application Server SSL documentation or Chapter 6, “Security” on page 107 for more information.

Using SSL is different when using QCFs or Nojndi:

- QCFs

  The definition of QCFs in JMSAdmin or in the admin console if WebSphere MQ V5.3 is being used, can have the sslCipherSuite and sslPeerName options set on them (SSLCIPHERSUITE and SSLPEERNAME when defining the QCF in JMSAdmin). Other SSL attributes are available on the QCF. However, this section does not discuss them.

  Define the four key store and trust store options to the JVM, using generic JVM arguments. To define them, perform the following tasks:

  a. From the admin console, expand **Servers** → **Application servers**.

  b. Under Server Infrastructure, expand **Java and Process Management** and select **Process Definition**.

  c. Select **Java Virtual Machine**, and add the following properties to the Generic JVM arguments text field (multiple arguments are separated with a space):

     - `-Djavax.net.ssl.keyStore=xxx`
     - `-Djavax.net.ssl.keyStorePassword=xxx`
     - `-Djavax.net.ssl.trustStore=xxx`
     - `-Djavax.net.ssl.trustStorePassword=xxx`

     Note: These set the properties for the entire JVM. Consider the implications for other SSL-enabled components.

  d. Restart the server for these properties to take effect.

- Nojndi

  When using Nojndi, specify all the SSL options mentioned earlier in the URI.

  Note: If the Web Service client is running on one of the WebSphere Application Server containers, the four key store and trust store options are *overridden* if they are already set for the WebSphere Application Server JVM.

13.6 Summary

This chapter discussed the implementation of a WebSphere Application Server Web Service client that uses the SOAP/JMS transport from WSDL. While this is not WebSphere MQ-specific, it demonstrates the interoperability that WebSphere MQ provides. This chapter also showed how Nojndi can be used when invoking a Web Service hosted by WebSphere MQ.

Part 4

Asynchrony and transactionality

WebSphere MQ V6 provides support for the transactional execution of Web Services. This is discussed in Part 2, “Web Services and security considerations” on page 27, of this book. Part 2 also covers the asynchronous invocation of Web Services using the Microsoft .NET asynchronous interfaces. Additional and separate facilities are available for the asynchronous and transactional invocation of WebSphere MQ transport for SOAP clients. These features are provided in the MA0V WebSphere MQ SupportPac, which is not included in the WebSphere MQ installation CDs, but is available for download from the WebSphere MQ SupportPac Web site.

© Copyright IBM Corp. 2006. All rights reserved.

Part 4 reviews the facilities provided by MA0V and the reasons why they are made available as a SupportPac, rather than as part of the actual product. This part provides a demonstration about how these facilities can be used to deliver asynchronous Java 2 Platform , Enterprise Edition (J2EE) and Microsoft .NET client applications with transactional control over Web Service requests and responses. This asynchrony is designed to provide asynchrony beyond the lifetime of a single client process, so that a response to a service request can be received on a separate process from the one that requested it.

14

Long-term asynchronous functionality (MA0V)

This chapter provides an overview of the asynchronous facilities provided in MA0V SupportPac. This SupportPac was available for download in the IBM WebSphere MQ SupportPac Web site shortly after the release of WebSphere MQV6. Following are the main topics covered in this chapter:

- Overview of long-term asynchronous facilities
- Installing MA0V
- Developing a client to use long-term asynchrony
- Illustrating an asynchronous implementation
- Long-term asynchrony error handling

© Copyright IBM Corp. 2006. All rights reserved.

14.1 Overview of asynchronous facilities

This chapter discusses the additional asynchronous functionality provided in the MA0V SupportPac. This functionality is in addition to the asynchronous functionality provided in the V6 GA product. This is described in Chapter 5, “SOAP/WebSphere MQ implementation” on page 49.

The asynchronous functionality provided in the product is limited because of the following reasons:

- It is only relevant to a single-process environment.
- It is not based on standard interfaces.
- It is only relevant to the Microsoft .NET client environment and there is no equivalent for the J2EE environment.

The asynchronous functionality in the SupportPac provides an implementation of long-term asynchrony. This enables a response from Web Services to be received asynchronously, which means that the client application is free to perform other work in the interval that the request is being serviced. It is possible for an asynchronous response to be returned in an entirely different process from the one that requested it. When working across processes in this manner, the response process may be run either at the same time as the request process or at any suitable time after the request process is completed.

> **Note:** MA0V is a Category II (Unsupported) SupportPac. Therefore, this additional asynchronous functionality is *not* supported currently.

This asynchronous functionality is appropriate for various situations such as those listed here:

- Where it is required to collect a group of responses in one operation to a series of requests that were made over a period of time
- Where communication links are available only at certain times of the day
- Where communication links are unreliable

The asynchronous facilities provided by MA0V are referred to in the rest of this chapter as *long-term asynchrony*, while that provided by the Microsoft .NET short-term interfaces are referred to as *short-term asynchrony*.

The requirement to be able separate a Microsoft .NET service request and response into different processes is the main reason why the development of an asynchronous client is based on long-term interface.

Although the interfaces provided in the long-term asynchronous support are not standard, its decoupling into a SupportPac is with the intent that if and when standards for asynchronous Web Services do emerge, MA0V can be developed to adopt the standard interfaces. It can then be integrated into the product so that it becomes supported.

The implementation of long-term asynchrony works with both Microsoft .NET clients and Axis clients. It is the only asynchrony that works with Axis clients.

Long-term asynchronous clients are required to provide transactional client requests or responses. This is discussed separately in Chapter 16, "Transactional functionality (MA0V)" on page 339.

Long-term asynchronous functionality is directed only at the client side of the Web Service request and response. The SOAP/WebSphere MQ listeners always process incoming service requests synchronously. The listeners are not affected by the implementation of long-term asynchrony.

14.2 Installation of MA0V

The MA0V SupportPac is *not* provided in the WebSphere MQ installation CDs. Download it from the following Web site:

http://www-306.ibm.com/software/integration/support/supportpacs/category.html#cat2

The installation of MA0V is a straightforward process. The SupportPac is packaged as a compressed file on the Windows platform. Extract this into the main WebSphere MQ installation directory. For UNIX and Linux systems, the file is a compressed tar file and operating system-specific installation utilities are necessary to install this after it is decompressed. The MA0V link in the Web site address provided earlier gives detailed installation instructions.

The ma0v.pdf document provided with the SupportPac is a super set of the standard WebSphere MQ transport for SOAP documentation. It includes all the information in the product version of the manual, with additional information and sections to describe the functionality provided by MA0V.

The key files installed by MA0V are:

- amqsoapasync_MA0V.dll
- com.ibm.mq.soapasync_MA0V.jar

Other files that are installed include additional asynchronous and transactional samples, a maintenance utility, amqwAsyncConfig.cmd, which is described in this chapter, and a script, amqwuninstall_MA0V.cmd, to uninstall MA0V.

Note: There is no .sh equivalent for the amqwuninstall_MA0V.cmd script.

14.3 The SOAP/WebSphere MQ Installation Verification Testing and MA0V

An additional Installation Verification Testing (IVT) configuration file, ivttestsasync.txt, is provided with MA0V on the Windows platform. It is called ivttests_unixasync.txt on UNIX and Linux platforms.

In order to configure the asynchronous samples, run the regenDemoasync.cmd/sh script from a working directory where it is necessary to build them.

Attention: We recommend that you do *not* configure IVT samples under the WebSphere MQ installation directory.

The test configuration files can be run through the IVT by using the runivt.cmd/sh script with the -c option, for example:

```
runivt -c ivttestsasync.txt
```

This runs the following asynchronous samples:

- Axis long-term asynchronous request and response from within the same process
- Axis long-term asynchronous request and response from within different processes
- Multiple Axis long-term asynchronous requests and responses from within the same process
- Microsoft .NET long-term asynchronous requests and responses from within the same process
- Microsoft .NET long-term asynchronous requests and responses from within different processes
- Multiple Microsoft .NET long-term asynchronous requests and responses from within the same process

14.4 Developing a client to use long-term asynchrony

Client software must be modified to use the long-term asynchronous interface. The methodology for this is specified here:

1. Asynchronous request notification

   Before making a service request call, a new call must be provided to WebSphere MQ transport for SOAP to inform whether an asynchronous request is about to be made.

2. Trapping an AsyncResponseExpectedException exception

   The client must trap a special exception that is thrown when the service request is to be made. WebSphere MQ transport for SOAP throws this exception when it has successfully initiated an asynchronous request. It does this as a means to tell the client that the request is made. The exception is not an error. It is the normal behavior. This is due to the fact that at that stage, the sender software cannot return a valid response because the service itself is being invoked asynchronously.

3. Instantiating an asynchronous response listener

   After the asynchronous request is made, a ResponseListener object must be instantiated. The response listener drives the response collection mechanism. If the request and response sides are to be split into separate processes, starting the response listener forms a part of the response process. Note that these ResponseListener objects are entirely separate from the amqwSOAPNETListener and SimpleJavaListener. A ResponseListener is provided purely for processing responses for asynchronous requests.

4. Implementing an asynchronous callback

   The client code must define a callback class that is derived from a specified WebSphere MQ transport for SOAP base class. An object of this class must be instantiated and passed as an argument when the method in step 1 is called.

   The callback() method of this object is invoked by the response listener when it establishes that a response to the request is available.

   The callback() method then drives the process of obtaining the response. To obtain the response, the method must first make a call to WebSphere MQ transport for SOAP to inform WebSphere MQ transport for SOAP infrastructure that it is about to request a return of the asynchronous response. A second dummy invocation of the service is then made, which causes the response to be returned by WebSphere MQ transport for SOAP.

These actions must be performed by a client to use the long-term asynchronous interface.

A service returns a response to an asynchronous request in the same way as for a synchronous request. The listener has no understanding that the client is performing the request mechanism and the response mechanism asynchronously. It is not involved in the asynchronous operation. The methodology is as follows:

1. The sender code in the asynchronous case is the same sender code used in the synchronous case. It works by being given a hint by the asynchronous request notification call, as mentioned in step 1 on page 307, that a particular request is to be processed asynchronously. WebSphere MQ transport for SOAP does this using thread local storage because this is the easiest way to communicate between the request notification call in the client layer and the sender code, which is invoked by Microsoft .NET or Axis.
2. After an asynchronous request is dispatched to the appropriate request queue, the sender code skips the logic to wait for a WebSphere MQ response message that it follows in the synchronous case. It also skips the SOAP layer response processing, that is, the action of returning a SOAP message to the client. Instead, the sender code throws an AsyncResponseExpectedException, which is what the client application must catch to indicate that the service request is successful.
3. The response listener works by browsing the response queue for response messages. To collect the message, the response listener is started and the Universal Resource Indicator (URI) for the service is passed as an argument so that it can determine which queue to monitor for responses.

   The response listener browses the response queue and when it determines that a response is available for a particular request, it enables the waiting client to collect it.
4. To collect the message, the client uses the callback object that is supplied to the asynchronous mechanism earlier. The callback object does not form part of the request message or the response message. Therefore, WebSphere MQ transport for SOAP must preserve a serialized form of it when the request is made and reconstruct it when the response message is ready.
   a. To do this, it uses a side queue, SYSTEM.SOAP.SIDE.QUEUE. The sender code serializes the callback object, which means that the callback object representation is converted to a sequence of bytes.
   b. The sender code then writes the callback object in a message to the side queue with the message ID of the original request message set as the callback object correlation ID.

c. After the response listener obtains a response, it retrieves the side queue entry by using the same correlation ID as that set in the response, which is again set to the correlation ID of the original request message.

The callback is then deserialized, that is, converted back into a callback object, and that object's CallbackFunction is then invoked to allow the client to obtain the response.

5. The side queue is created by the script setup, WMQSOAP.cmd/sh.

> **Note:** The name of the side queue is hardcoded into the WebSphere MQ transport for SOAP sender code and is not easily customizable.

The long-term asynchronous message flow is depicted in Figure 14-1.

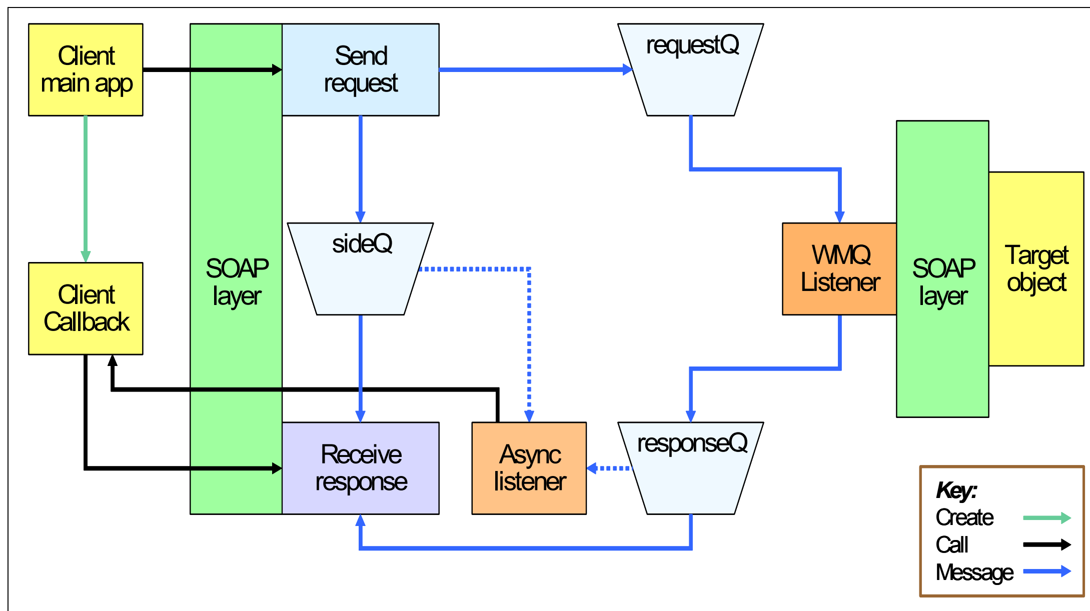


Figure 14-1 Overview of long-term asynchrony

Figure 14-1 depicts 12 different flows of information. Each of these comes under one of the following types of information flow and process flow:

- Create operation

  This is the creation of the callback object. This flow is depicted by enclosing the number that labels it within a diamond. Only one create operation is depicted in Figure 14-1 as a callback object. This is created only once for an individual service request.

- Call operation

  This is a call to a method. This is depicted by enclosing the number that labels it in a square.

- Message operation

  This is a WebSphere MQ Put, Get, or Browse operation. This is depicted by enclosing the number that labels it within a circle.

Following are the12 elements of the asynchronous flow:

1. Client provides a callback class derived from Async.Callback (Create).
2. Client makes an asynchronous request (Call).
3. Sender puts request message on the request queue (Message).
4. Serialized callback object is put to the side queue (Message).
5. Listener reads the request (Message).
6. Listener puts the response (Message).
7. Asynchronous response listener browses a response message (Message).
8. Asynchronous response listener browses the corresponding side queue message (Message).
9. Asynchronous listener recreates the callback object and calls it (Call).
10. Callback makes asynchronous request to receive the response (Call).
11. Sender gets the response message from the response queue (Message).
12. Sender gets the callback object from the side queue to delete it (Message).

The three additional key components in the long-term asynchronous model are:

- Client callback

  This is a class that the customer must implement from an interface provided in WebSphere MQ transport for SOAP. The customer must provide a callback method, which retrieves the response that it is waiting for.

- Side queue

  The callback objects provided by the client are serialized and stored on this queue. These objects are provided by the customer so that the response can be processed under the control of the response listener.

- Response listener

  This browses the response queue for messages and invokes the customer's callback when a response is received.

The basic pattern for using asynchronous clients involves the following steps, which do not have to follow a prescribed order:

- The client notifies the sender that it wishes to make an asynchronous request and specifies a callback object associated with the request.
- The client invokes the service.

- The client ensures that an asynchronous response listener is running on the response queue.
- When a response arrives, the asynchronous response listener invokes the callback method in the callback object.
- The callback method processes the actual response.
- The client stops the asynchronous response listener

It does not matter in which order the first two steps are started. The processes that invoke the request and start the response listener to enable the response to be returned can either be in the same process or in different processes. It is the responsibility of the client process to start the response listener in whichever process to collect responses from asynchronous service requests.

Unlike the service listeners amqwSOAPNETListener and SimpleJavaListener that are used on the server side for both synchronous and asynchronous requests, the asynchronous response listener that runs on the client side is single-threaded. There are currently no options for running a response listener in a multithreaded mode. This limitation is essential because of the way the asynchronous response mechanism browses a response queue before invoking specific request callbacks. A multithreaded response listener must protect individual threads from interfering with each other. In the same way, trying to instantiate multiple response listeners for the same clientID within a single client application causes coordination problems, and must be avoided.

14.5 Response queues and asynchronous clientID

There is another important concept in long-term asynchrony that is not yet introduced. This is the role of the clientID. This is a mechanism used to specify a logical client name to which an asynchronous request belongs. The clientID is the string specified by the client application in the previous section, when it informs WebSphere MQ transport for SOAP that it is about to make an asynchronous request. An individual application may use different clientIDs for different groups of asynchronous requests or it may specify a common clientID for all the requests that it processes. A second client application can use the same clientID. The mechanism by which groups of requests are grouped under clientIDs is completely under the control of client applications.

A permanent dynamic response queue is created for each clientID submitting asynchronous requests. This means that when the response listener is started for a particular clientID, it is only monitoring the specific queue associated with that clientID and not all the responses for all the requests. Given the fact that the response listener works by browsing the response queue, this avoids

performance issues, which may otherwise arise from large numbers of responses being presented to a single response queue for multiple clientIDs. When the sender software determines that a service is to be made asynchronously, a permanent dynamic queue is created if it does not already exist for that clientID.

The use of the clientID and the automatically generated permanent dynamic queue mechanism also provides a means to decouple synchronous and asynchronous responses and prevent them from interfering with each other.

Note: clientIDs and the use of dynamic response queues for asynchronous clients are provided for resilience and performance. They do not provide any form of additional security control over and above that provided by WebSphere MQ.

The clientID is an unbounded string and may be of any reasonable length. It may be far longer than the maximum permitted length of a queue name, which is currently 48 characters. The side queue is used to store messages that associate a physical response queue name with a clientID and a base response queue name. The physical queue name of the dynamic response queue is determined by both the clientID and the name of the response queue as specified in the URI. The name of the response queue given in the URI is referred to as the base response queue name.

Note: When a base response queue name is prefixed with SYSTEM., that prefix is changed to DYN. This is to prevent dynamic response queues being generated as WebSphere MQ system objects.

The permanent dynamic queues created for asynchronous service requests are left in place when a response is returned to the client. They are not deleted when a response is returned. Instead, if the response queue is already created for that clientID and base response queue name, its physical name is retrieved from the side queue entry that defines the mapping. The sender software uses a hash code based on the clientID and base queue name as a correlationID for locating this mapping entry.

Deleting the permanent dynamic response queues that are no longer required is recommended. A special utility script, amqwAsyncConfig.cmd/sh, is provided for this purpose. This is described in 14.10.1, “Removing queue mapping entries from the side queue” on page 321.

> **Note:** It is not possible to create dynamic response queues or any other form of queues transactionally. This means that there is a risk of failure between the creation of a dynamic queue and writing its queue mapping entry to the side queue. Failure in this window creates an orphaned dynamic queue that is never used.

14.6 Illustration of client software modification

Having described how a client application must be modified to use the long-term asynchronous interface, this section uses example code fragments to illustrate the process for a specific case.

14.6.1 Asynchronous request notification

Example 14-1 illustrates how notification of an asynchronous request is made from a client. The call to IBM.WMQSOAP.Async.Request provides this notification based on a prepared callback object, requestContext, and the clientID of myClient. The call must be made after registering the WebSphere MQ transport for SOAP URI with the Register.Extension() method and before the service is invoked.

Example 14-1 Illustrating the notification of an asynchronous request

```
// Register the WMQSOAP URL extension with DotNet
IBM.WMQSOAP.Register.Extension();

// Create a context object to pass to the WMQSOAP Async class
testStateClass requestContext = new testStateClass();

String clientId = “myClient”;
IBM.WMQSOAP.Async.Request(requestContext, clientId);
```

The preparation of the callback class is discussed in 14.6.4, “Implementing an asynchronous callback” on page 315.

14.6.2 Trapping an AsyncResponseExpectedException

After informing WebSphere MQ that an asynchronous request must be performed, the code in Example 14-1 shows the request is made. The actual request invocation is the same as the synchronous request, except that an AsyncResponseExpectedException is expected when the asynchronous request is successfully made. Example 14-2 illustrates the invocation of an asynchronous request.

Example 14-2 Illustrating the invocation of an asynchronous request

```
StockQuoteDotNet stockobj = new StockQuoteDotNet();
try
   {
   res = stockobj.getQuote("XXX");

   // If we hit this line, something went wrong with the asynchronous
request
   callSomeErrorMethod();
   }
catch (IBM.WMQSOAP.AsyncResponseExpectedException e)
   {
   if (e.CompletionCode != MQC.MQCC_OK) callSomeErrorMethod();
   }
```

In this example, the AsyncResponseExpectedException must be thrown. If it is not, that is, the line immediately after the stockobj.getQuote() call is thrown, then the asynchronous request does not complete successfully. If the exception is caught, the completion code is checked to ensure that it is set to OK.

14.6.3 Instantiating an asynchronous response listener

Having successfully made a request, the client must start an asynchronous response listener to monitor responses and action a callback to the provided callback object when a response is returned. This is illustrated in the code fragment shown in Example 14-3.

Example 14-3 Illustrating the starting of an asynchronous response listener

```
// Create a proxy object so we know what URL to point
// the response listener to.
StockQuoteDotNet stockobj = new StockQuoteDotNet();
```

```
// Now start the response listener
String clientId="myClient";
m_responseListener = new IBM.WMQSOAP.ResponseListener(stockobj.Url,
clientId);
```

In this example, a proxy object is created. Therefore, the URI for the service is passed to the response listener. This is required where the asynchronous response is being collected in a process that is different from the request. If the request and the response are collected from the same process, there is no reason why the original proxy object is not used. The clientID is also passed as an argument to the response listener.

14.6.4 Implementing an asynchronous callback

The other modification that must be made in order to make a client asynchronous is implementing a callback class. An object of this class must be instantiated and specified as an argument when making a notification that an asynchronous request is about to be made.

The key points to be noted here are:

- The class must be derived from the abstract class AsyncCallback.
- The class must be marked as Serializable so that objects for the class can be serialized and preserved on the side queue.
- WebSphere MQ transport for SOAP must be notified that an asynchronous response is about to be collected. This is the call to IBM.WMQSOAP.Async.Response().
- A proxy object for the service must be instantiated and a further request made on the service. When the WebSphere MQ transport for SOAP sender actions this request, it realizes that it is being asked to return a waiting response because of the previous call to Async.Response(). It therefore, goes ahead with the process of obtaining the response from the response queue. This second call on the service does not result in a further request message being dispatched to amqwSOAPNETListener. Example 14-4 illustrates asynchronous callback implementation.

Example 14-4 Illustration of asynchronous callback implementation

```
[Serializable] class testStateClass : IBM.WMQSOAP.AsyncCallback
{
   public override void CallbackFunction()
   {
   try
      {
```

```
      // Create a proxy object so we know what URL to point
      // the response listener to.
      StockQuoteDotNet stockobj = new StockQuoteDotNet();

      // Register the WMQSOAP URL extension with DotNet
      IBM.WMQSOAP.Async.Response(this);

      System.Single res = stockobj.getQuote("unused dummy parameter");
      Console.WriteLine("ASYNC response is: " + res);
      }
   catch (System.Exception e)
      {
      Console.WriteLine("\n>>> EXCEPTION WHILE RUNNING
SQCS2DotNetAsyncReqeustResponse DEMO <<<\n" + e.ToString());
      }
   }
}
```

14.6.5 Stopping the response listener

At some point, the client determines that it no longer wishes to retrieve responses, for example, if all the expected responses are received or if no responses are received in the interval in which they were expected. When the client finishes processing the responses, it closes down the response listener with the stop method:

```
m_responseListener.Stop();
```

Not closing down the response listener in this manner does not cause any loss of integrity. However, we recommend it as part of good programming practice.

14.7 Building client applications

Client applications must reference the additional long-term asynchronous support code that is provided in MA0V. For Microsoft .NET, the client must reference the additional dynamic link library (DLL). For Java, the CLASSPATH is already set to include the additional jar file if the amqwsetcp.sh/cmd is being used.

14.7.1 Microsoft .NET client applications

In order to build client applications using the asynchronous interface, the client applications must be linked with the additional asynchronous SOAP DLL provided with MA0V. This DLL is amqsoapasync_MA0V.dll. Example 14-5 shows this with one of the supplied asynchronous clients.

Example 14-5 amqsoapasync_MA0V.dll with one of the supplied asynchronous clients

```
csc "/lib:%WMQSOAP_HOME%\bin"
/r:amqsoap.dll,amqsoapasync_MA0V.dll,amqmdnet.dll
"%WMQSOAP_HOME%\Tools\soap\samples\dotnet\SQCS2DotNetAsyncRequestRespon
se.cs" generated\client\*.cs
```

14.7.2 Java client applications

For Java client applications, it is not necessary to make any explicit reference to the additional asynchronous jar file, com.ibm.mq.soapasync_MA0V.jar. This is because the reference to this jar file is already made in the CLASSPATH set in amqwsetcp.sh/cmd. If the amqwsetcp.sh/cmd is not being used, the CLASSPATH must be amended to include %WMQSOAP_HOME%\java\lib\com.ibm.mq.soapasync_MA0V.jar.

This must be present in the CLASSPATH before the product jar file for WebSphere MQ transport for SOAP, %WMQSOAP_HOME%\java\lib\com.ibm.mq.soap.jar

Note: For both Microsoft .NET and Axis client applications, the client must be run on a system where the MA0V SupportPac is installed. Keep this in mind when clients are built on one machine and distributed to target client systems.

14.8 Long-term asynchrony and error handling

Error handling with asynchronous clients in WebSphere MQ transport for SOAP listeners is similar to that of synchronous clients. The listeners are not aware that the client is working asynchronously. However, in the context of asynchronous response listeners, some issues must be considered.

If a report message is returned from an asynchronous request, there is no special requirement for the response listener to remove the context message, because this is already consumed when the response listener retrieves the report message. However, other instances, where context messages are left on the side queue, may exist. This is why the amqwCleanSideQueue.cmd/sh utility is provided.

In the event that the response listener cannot retrieve a side queue entry for a particular response, the behavior first depends on the report options set in the response message. The WebSphere MQ transport for SOAP listener does not set any specific report options in a response message. In particular, this means that MQC.MQRO_DISCARD_MSG is not set. The response listener checks this report option. If it has been left as default, which is the case unless a customized listener is implemented, the response message is written to the dead letter queue. If the response message cannot be dead lettered, the same logic holds good as with the main WebSphere MQ transport for SOAP listeners, that is, the action depends on the message integrity setting.

The message integrity setting is an optional argument in the response listener constructor. It can be set as follows:

- For Microsoft .NET clients, it can be one of the following:
  - WMQSOAP.ErrorHandler.DEFAULT_MSG_INTEGRITY
  - WMQSOAP.ErrorHandler.LOW_MSG_INTEGRITY
  - WMQSOAP.ErrorHandler.HIGH_MSG_INTEGRITY
- For Java clients, it can be one of the following:
  - com.ibm.mq.soap.transport.wmq.ErrorHandler.DEFAULT_MSG_INTEGRITY
  - com.ibm.mq.soap.transport.wmq.ErrorHandler.LOW_MSG_INTEGRITY
  - com.ibm.mq.soap.transport.wmq.ErrorHandler.HIGH_MSG_INTEGRITY

The default value for both the environments is default message integrity. This is the value assumed if a signature that does not include the option is used.

The effect of the message integrity option when a response message cannot be dead lettered by the response listener is as follows:

- DefaultMsgIntegrity

  The behavior is defined by the persistency of the response message. If it is nonpersistent, the response listener shows a warning and continues to run, with the response message being discarded. For persistent messages, it shows an error message, leaves the response message on the response queue, and exits. The WebSphere MQ transport for SOAP listeners pass the same persistency into the response message as is set in the request message.

- Low message integrity

  The response listener shows a warning, discards the response message, and continues to run, regardless of the response message persistency.
- High message integrity

  The response listener shows an error message, leaves the response message on the response queue, and exits, regardless of the response message persistency.

14.9 ResponseListener start/finish notification

An interface is provided in WebSphere MQ transport for SOAP, which can be used to enable a notification to be made to a client when an asynchronous response listener initializes or exits. This interface is IBM.WMQSOAP.IResponseListenerNotification for Microsoft .NET and com.ibm.mq.soap.transport.jms.IResponseListenerNotification for Java.

In order to use this mechanism, the asynchronous client must implement the interface in a class and then create an object from this class, which must be passed to the response listener when it is started.

Example clients that use this interface are included with the samples that are provided with WebSphere MQ V6. These are called SQCS2DotNetAsyncReqRespNtfy.cs and SQAxis2AxisAsyncReqRespNtfy.java. These samples are not integrated with the asynchronous Installation Verification Test (IVT) described earlier in this chapter.

Example 14-6 shows the Microsoft .NET interface that is provided.

Example 14-6 The Microsoft .NET interface

```
public interface IResponseListenerNotification
{
   void onInitialize();
   void onTerminate(Exception anException);
}
```

The Java interface is the same. A client can implement this interface and take action when the listener initializes or terminates. The onTerminate() method is called whether or not the response listener terminates normally or under an exception condition. Example 14-7 illustrates how a notifier structure is built into a client.

Example 14-7 Creating a ResponseListener notifier

```
class SQCS2DotNetAsyncReqRespNtfy
{
   // Define Callback function
   ...

   // Define the response listener notification class
   class DemoNotifier : IBM.WMQSOAP.IResponseListenerNotification
   {
   public void onInitialize()
      {
      // Take some action when the response listener starts
      // This might be for example to check databases are on line and available
      System.Console.WriteLine("The DemoAsyncNotifier onInitialize method has been
called.");
      }


   public void onTerminate(Exception anException)
      {
      // Take some action when the response listener stops
      // This might be for example to check databases are closed down properly
      // or to purge redundant side queue entries
      System.Console.WriteLine("The DemoAsyncNotifier onTerminate method has been
called.");
      }
   }



   static void Main(string[] args)
   {

      // Perform set up etc
      ...

      // Initiate request
      ...
```

```
        // Create an error handler object to pass to the
        // WMQSOAP Async class
        DemoNotifier notifier = new DemoNotifier();

        // Register the WMQSOAP URL extension with DotNet
        IBM.WMQSOAP.Register.Extension();

        // Create a proxy object so we know what URL to point
        // the response listener to.
        StockQuoteDotNet stockobj = new StockQuoteDotNet();

        // Now start the response listener, passing it the
        // error handler object
        m_responseListener = new IBM.WMQSOAP.ResponseListener(stockobj.Url, clientId,
notifier);
    }
}
```

The onTerminate() method is the most useful and is typically used to perform clean-up operations in the event of an error condition.

14.10 Maintaining the side queue

As discussed earlier, permanent dynamic queues are created for every clientID and base response queue combination. There is no mechanism in WebSphere MQ transport for SOAP to automatically delete these queues because it has no way of knowing when they are no longer required. Deleting these permanent dynamic queues when appropriate is recommended. The side queue contains a queue mapping entry for each permanent dynamic queue that is created. These too must be deleted when the queues are no longer required.

14.10.1 Removing queue mapping entries from the side queue

A utility is provided with WebSphere MQ transport for SOAP for deleting asynchronous permanent dynamic response queues and their queue mapping entries. This utility is provided both as a callable method and as a script. The script is called amqwAsyncConfig.cmd/sh. The callable method is given with

Microsoft .NET and Java interfaces. It is easier to use the script version, unless applications that generate high volumes of response queues are being written. This may be the case, for example, when an application design involves the use of many different clientIDs or base response queue names.

Invoke the script as follows:

```
amqwAsyncConfig [options]
```

The options are:

- -qm queue manager

  This is the name of the queue manager. The default is default queue manager.
- -clientID filter

  This is the filter to be used for clientID. The default is no filtering.
- -baseQ filter

  This is the filter to be used for the base queue name. The default is no filtering.
- -dynQ filter

  This is the filter to be used for the dynamic queue name. The default is no filtering.
- -java

  This shows the dynamic queues for Java clients.
- -dotnet

  This shows the dynamic queues for Microsoft .NET clients.
- -list

  This lists the output to the terminal.
- -delete

  If set, matching entries are deleted, including the side queue entry and the dynamic queue.

To understand the use of the script, perform the following tasks:

1. Run the IVT demo SQCS2DotNetAsyncRequestResponse.exe with a client ID of accountUpdate, as shown in Example 14-8.

Example 14-8 Running the SQCS2DotNetAsyncRequestResponse.exe

```
SQCS2DotNetAsyncRequestResponse request accountUpdate
Async service(s) successfully requested.

SQCS2DotNetAsyncRequestResponse response accountUpdate
ASYNC response is: 88.88
All Async response(s) successfully received.
```

2. Interrogate the list of permanent dynamic queues mapped for long-term asynchrony with the command shown in Example 14-9.

Example 14-9 Interrogating the list of permanent dynamic queues

```
amqwAsyncConfig -qm WMQSOAP.DEMO.QM -list
  clientId(accountUpdate) baseQ(DYN.SOAP.RESPON)
dynQ(DYN.SOAP.RESPON_NaccountUpdate42CC134E022B0020) env(dotnet)
```

   This shows that the dynamic response queue DYN.SOAP.RESPON_NaccountUpdate42CC134E022B0020 is created for the clientID accountUpdate because the base response queue name defaults to SYSTEM.SOAP.RESPONSE.QUEUE. The base queue name that is created is changed from a SYSTEM. prefix to a DYN. prefix and then truncated. A SYSTEM. prefix is replaced with DYN. as described earlier, in order to prevent dynamic response queues from being created as WebSphere MQ system objects.

   The truncation of the base response queue name is performed as part of the process of generating the permanent dynamic response queue name. This truncated form of the base queue name is stored on the side queue. If you try to filter the base response queues with the utility, the original full form of the response queue name does not work. The base queue name must be specified because it is stored on the side queue.

The example dynamic response queue that is created can be deleted with the following command:

```
amqwAsyncConfig -qm WMQSOAP.DEMO.QM -clientId accountUpdate -baseQ .*
-delete
```

If the permanent dynamic response queues are listed, an empty list is returned:

```
amqwAsyncConfig -qm WMQSOAP.DEMO.QM -list
```

For details about how to invoke the callable form of this utility, refer to the WebSphere MQ transport for SOAP documentation.

14.10.2 Removing redundant context objects from the side queue

In addition to the amqwAsyncConfig utility, a separate utility is used for deleting redundant context entries from the side queue. These context messages are the serialized forms of callback objects and are entirely different messages from the queue mapping entries that associate a clientID and base response queue name with the generated permanent dynamic queue name. Context messages are typically deleted when a client successfully obtains a response through the long-term asynchronous mechanism.

Use this utility when you know that long-term asynchronous calls have failed and that the context messages are no longer required. It must not be run automatically without checking why the asynchronous service requests failed to complete. The utility is called amqwCleanSideQueue.cmd/sh. It is a script wrapper around a Java program. Invoke this utility with:

```
amqwCleanSideQueue [options]
```

The options here are:

- -d days

  This specifies an age in days. Messages older than this age are removed. The default is 1.

- -h hours

  This specifies an age in hours. Messages older than this age are removed.

- -mn mins

  This specifies an age in minutes. Messages older than this age are removed.

- -m queue-manager

  This specifies the queue manager that contains the side queue.

- -q side-queue

  This specifies the name of the side queue. The default is SYSTEM.SOAP.SIDE.QUEUE.

- -?

  This shows help information, describing how the command must be used.

Only one of the -d, -h, and -mn options are permitted. If none are specified, the default is -d 1, one day.

The amqwCleanSideQueue.cmd/sh utility prompts for confirmation before deleting any messages, the target queue manager, queue name, and age. It shows error messages for any messages that it cannot delete. It also shows the total number of messages that are successfully deleted, before it exits. It is required to delete all the context messages on the side queue, no matter how old they are. Use the **-d 0** command as shown in Example 14-10.

Example 14-10   Deleting all context messages on the side queue

```
C:\temp\rb_sync>amqwCleanSideQueue -d 0
Cleaning messages more than 0 day old, for side queue
SYSTEM.SOAP.SIDE.QUEUE, for queue manager WMQSOAP.DEMO.QM
Are you sure? (y/n)
y
Successfully deleted 1 messages

C:\temp\rb_sync>
```

Because there is nothing specific in this utility for WebSphere MQ transport for SOAP, there is no reason why customers cannot write their own utility if it is essential for customizing the way old context messages are deleted.

Note: As described earlier, the -q option allows a side queue name to be specified. However, in WebSphere MQ V6, the side queue name is hard-coded in the sender software. Therefore, it is not easy to use an alternative side queues.

14.11 Uninstalling MA0V SupportPac

To uninstall MA0V on the Windows platform, use the script amqwuninstall_MA0V.cmd, which is located in the WebSphere MQ bin subdirectory. This script individually deletes all the files that are installed with the SupportPac. On the Windows platform, it also deletes the asynchronous support DLL, amqsoapasync_MA0V.DLL, from the Global Assembly Cache. For uninstallation on other platforms, refer to the documentation provided with SupportPac.

14.12 Summary

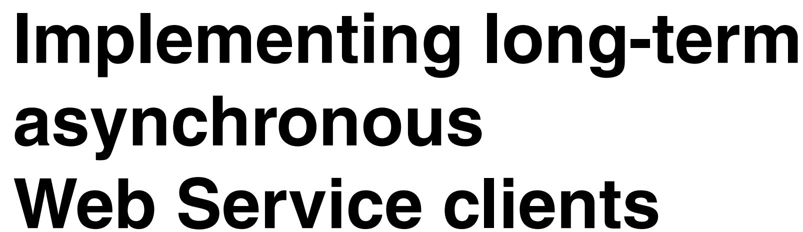
This chapter looked closely at the long-term asynchronous facility provided in the MA0R SupportPac and compared this to the short-term facility provided in WebSphere MQ V6. It illustrated situations where the use of long-term asynchrony is beneficial, and briefly reviewed how the additional asynchronous samples provided with MA0V can be run through the SOAP/WebSphere MQ IVT mechanism.

This chapter discussed the long-term asynchronous mechanism in detail, including the role of the side queue, the callback object, the different ways in which the response queues are used, and the asynchronous response listener.

The steps involved in changing a client from a synchronous style of operation to a long-term asynchronous style were illustrated for Microsoft .NET.

Error handling techniques and the use of a customized notification mechanism with the response listener were also addressed. Utilities that are provided to maintain the side queue were discussed. The chapter concluded with some general guidance about the installation and uninstallation of MA0V.

15



Implementing long-term asynchronous Web Service clients

Web Services can be invoked by their clients either synchronously or asynchronously. An asynchronous invocation is one where clients send requests and do not expect replies immediately. The client application can send a request, continue with other processes, and request a response later.

It is possible for a response to be returned to a process that is separate from the one that sent the request. When this happens, it is typically referred to as long-term asynchrony. Having the responses to several requests that are made over a period of time come back on separate processes means that the responses can be collected and grouped. This is useful in situations where it is neither practical nor efficient to prolong the lifetime of a client until all the responses to separate calls are received.

This chapter aims at demonstrating the implementation of Web Service clients that make use of long-term asynchrony.

© Copyright IBM Corp. 2006. All rights reserved.

15.1 The Web Service

Before long-term asynchronous calls are made from a client, the Web Service it invokes must be specified.

The development of a BankingService Web Service was described in the earlier chapters. Refer to Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269.

The development of a .NET client to invoke any of these Web Services was also described in the earlier chapters. Refer to Chapter 11, ".NET client" on page 243, Chapter 9, "Axis client" on page 187, and Chapter 13, "WebSphere Application Server client" on page 289.

The Web Service models a bank account and the common operations that take place on it. Following is a list of these operations:

- getBalance

  This is a method to return the account balance.

- credit

  This is a method to add a provided amount to the existing account balance and store the operation details in a user-defined object called a BankOperation object.

- debit

  This is a method to deduct a provided amount from the existing account balance and store the operation details in a user-defined object called a BankOperation object.

- getStatement

  This is a method to return the last three operations on an account. This is returned in an array of BankOperation objects.

The .NET and Axis Web Service client calls each of these methods and shows the results on a graphical user interface (GUI).

This chapter demonstrates the credit method call. This method, which was implemented synchronously earlier, is used to make a long-term asynchronous call.

15.2 Implementation of long-term asynchrony

This section discusses the implementation of a Web Service client that calls a Web Service asynchronously by sending SOAP messages over WebSphere MQ in one process and receiving it in another process.

The .NET client and Axis client developed previously (Chapter 9, “Axis client” on page 187 and Chapter 11, “.NET client” on page 243) are modified in this chapter to demonstrate long-term asynchrony support provided by WebSphere MQ transport for SOAP. Long-term asynchrony is demonstrated on the method call to the credit method of the Web Service.

The following sections explain the additions required for the credit method call to make it a long-term asynchronous call. Download the code for this from Appendix D, “Additional material” on page 431.

> **Note:** The code includes comments such as `STEP 4 of CHAPTER 14.5`. It indicates which part of the code corresponds to the steps to modify the client code in order to implement long-term asynchrony discussed in 14.3, “The SOAP/WebSphere MQ Installation Verification Testing and MA0V” on page 306.

Implement an asynchronous callback object

This section demonstrates the creation of an object containing a method, callback method, that is invoked when a response is returned. This class extends the IBM.WMQSOAP.AsyncCallback class, and must override a method called CallbackFunction. This object must be created before invoking the request. This enables the response listener to retrieve this object from a predefined location.

> **Note:** The predefined location is a side queue. This requires the object containing the callback method to be serialized. For further information, see Chapter 14, “Long-term asynchronous functionality (MA0V)” on page 303.

Use the code snippet in Example 15-1 to implement the asynchronous callback function in the .NET client.

Example 15-1 Code to implement the asynchronous callback in the .NET client

```
//STEP 4 of CHAPTER 14.5: IMPLEMENT AN ASYNCHRONOUS CALLBACK
// This is the overridden method that is invoked by WMQSOAP
// when a response is received for this client. It prints the result of 
the
// request.
[Serializable] class creditCallback : IBM.WMQSOAP.AsyncCallback
```

```
{
   // This is the overridden method that is invoked by WMQSOAP
   // when a response is received for this client. It prints the
   // result of the request.
   public override void CallbackFunction()
   {
      try
      {
      // Create a proxy object so we know what URL to point the
response
      //listener to.
      BankingService service = new BankingService();
      //then point the response listener to it
      IBM.WMQSOAP.Async.Response(this);

      //make the initiation Web service call
      bool creditSuccessful = service.credit(0.0);

      }
      catch (System.Exception e)
      {
         Console.WriteLine("\n>>> EXCEPTION WHILE RUNNING
BankingService " +
                       " Credit method call <<<\n" + e.ToString());
      }
   }
}
```

Note: In order to build, this method requires the library file amqsoapasync_MA0V.dll to be referenced.

Use the code snippet in Example 15-2 to implement the asynchronous callback in the Axis client.

Example 15-2 Code to implement the asynchronous callback in the Axis client

```
import com.ibm.mq.soap.transport.jms.*;

public class BankClient extends AsyncCallback
{
   static BankingService service;

   //STEP 4 of CHAPTER 14.5: IMPLEMENT AN ASYNCHRONOUS CALLBACK
   // This is the overridden method that is invoked by WMQSOAP
```

```
   // when a response is received for this client. It prints the result
of the
   // request.
   public void CallbackFunction()
   {
      // flag used to indicate whether transactionality is being used
      boolean isTransacted=false;

      try
      {
         // prepare the WMQ-SOAP sender to return an asynchronous
response
         Async.Response(this, isTransacted);

         // call to retrieve response. Doesn’t call the service again,
         // just indicates that the callback method is asking for the
response
         boolean result = service.credit(0.0);
         ...
         ...
      }
      catch (Exception e)
      {
         System.out.println("Exception in CallBackFunction: " +
e.getStackTrace());
      }
}
```

Note the following points about the Axis implementation of the callback function:

- The call to Async.Response takes an additional parameter that is not found in the C# code. This is due to a difference in the way the client side transactions work in the two languages.
- This callback method is implemented in BankClient.java. Since this has already instantiated a proxy object, it is reused. The alternative is to use the locator object to create a new proxy object, as in the C# code.

> **Note:** In order to build, this method requires the library file com.ibm.mq.soapasync_MA0V.jar in the classpath. The script amqwsetcp does this.

Making the Web Service call

After preparing an object containing a callback method, the client can create an instance of this object and make a request. Perform the following tasks to make a Web Service call:

1. Before a request is made for sending SOAP messages over WebSphere MQ, generate the latter as the transport mechanism by using the Register.Extension() call. This call has been made earlier in the case of both the clients:
   - In the .NET client constructor method, Form1()
   - In the main method of the Axis client
2. Create a new instance of the object containing the callback function. This enables you to call the method Async.Request in order to advise the SOAP for WebSphere MQ transport that an asynchronous request is about to be made. This method takes an instance of the object and a specified ID as parameters.
3. Finally, the Web Service call is made. The call, if successful, throws an AsyncResponseExpectedException. This exception must have a completion code of MQC.MQCC_OK to reflect the success of the asynchronous call. Use the code shown in Example 15-3 to implement asynchronous Web Service call in .NET client.

Example 15-3 Code to implement the asynchronous Web Service call in .NET client

```
private void btnCredit_Click(object sender, System.EventArgs e)
{
   bool x = false;

   //STEP 1 of CHAPTER 14.5: Asynchronous request notification
   //Register the WMQSOAP URL Extension with .NET
   //Already done in the constructor Form1()

   //instantiate the call back object to pass to the WMQSOAP Async
class
   creditCallback callBack = new creditCallback();
   //create a client request ID
   string requestClientID = "creditrequest";
   //Finally call IBM.WMQSOAP.Async.Request passing it the callBack
object
   //and the requesting client ID
   Async.Request(callBack, requestClientID);

   //STEP 2 of CHAPTER 14.5: Trapping an AsyncResponseExpectedException
   // WMQ has been informed that an asynchronous request is going to be
made,
```

```
   // We can now make the request.
   BankingService service = new BankingService();

   try
   {
      x = service.credit(10.10);
   }
   catch (AsyncResponseExpectedException ex)
   {
      //if a WMQ error is thrown with a completion code 'OK', then the
      //request has been successful
      if(ex.CompletionCode != MQC.MQCC_OK)
         throw ex;
   }
}
```

The request code for the Axis client is shown in Example 15-4.

Example 15-4 Code to implement the asynchronous Web Service call in Axis client

```
class BankingGUI extends Async implements ActionListener, Runnable
{

public BankingGUI(BankingService service, BankClient bankClient)
{
   this.bankClient=bankClient;
   ...
}

private void startAsyncCreditCall(double amountToCredit) throws
Exception
{
   //Registration of the WMQSOAP URL prefix has taken place in the main
method

   boolean isTransactional=false;

   try {

         //STEP 1 of CHAPTER 14.5: Asynchronous request notification
         Async.Request(bankClient,clientID,isTransactional);
         // get handle on service
         BankingServiceServiceLocator locator=
                 new BankingServiceServiceLocator();
         BankingService service=
```

```
                locator.getBankingServiceBankingService_Wmq(
                    new java.net.URL(BankClient.bankingServiceURL));

            //STEP 2 of CHAPTER 14.5: Trapping an
AsyncResponseExpectedException
            // WMQ has been informed that an asynchronous request is going
to be
            // made, We can now make the request, catching the exception
            // generated as a result

            try {
              boolean result=service.credit(amountToCredit);
            }
            catch (AsyncResponseExpectedException ex)
            {
              if (ex.completionCode!=MQTrace.MQCC_OK)
              {
                throw ex;
              }
            }
        }
        catch (Exception ex)
        {
            System.out.println("Exception occurred: "+ex.getStackTrace());
        }
}
```

Note the following points about the invocation of the service in Axis client:

- The bankClient object, which contains the callback method, is passed into the BankingGUI class through the constructor.
- The call to register the asynchronous request (Async.Request) takes an additional parameter that is not found in the C# implementation. This parameter is a boolean flag specifying transactionality.
- myBankClient is the ID used to register the asynchronous request.

After the service is successfully invoked, listen for a response. The asynchronous response listener IBM.WMQSOAP.ResponseListener provides this functionality. Create and provide an instance of this response listener with the proxy URI. The request ID is also specified when making the request.

The response listener then browses the response queue for responses with the specified ID. When it finds a matching response from the Web Service, the object

containing the callback function is retrieved from the side queue. The callback function is then invoked to handle the response. See 14.1, "Overview of asynchronous facilities" on page 304 for a description of the long-term asynchronous process.

Use the code snippet in Example 15-5 to implement the asynchronous response listener in .NET client.

Example 15-5 Code to implement the asynchronous response listener in .NET client

```
private void btnCredit_Click(object sender, System.EventArgs e)
{

   //STEP 3 of CHAPTER 14.5: Instantiating the asynchronous response
listener
   //the request has been made asynchronously, the response needs to
come back
   //Start up the response listener
   //create an instance of our error handler
   AsyncErrorHandler listenerErrorHandler = new AsyncErrorHandler();
   //now listen for a response

   ResponseListener listener = new ResponseListener(service.Url,
      requestClientID);
}
```

Note: When using the Microsoft .NET Framework, reference the WebSphere MQ transport for SOAP libraries, amqsoap.dll and amqsoapasyn_ma0v.dll. In order to use the WebSphere MQ constants such as MQC.MQCC.OK, the library amqmdnet.dll is required. These libraries are located in <WebSphere MQ home directory>\bin.

A reference the System.Web.Services library is required for the Web Service proxy to compile.

Use the code snippet in Example 15-6 to implement the asynchronous listener in Axis client.

Example 15-6 Code to implement the asynchronous response listener in Axis client

```
class BankingGUI extends Async implements ActionListener, Runnable
{
   ResponseListener listener;
   ...
   private void startAsyncCreditListener()
   {
```

```
      //STEP 3 of CHAPTER 14.5: Instantiating the asynchronous response
      // listener. The listener begins automatically
      listener=new
ResponseListener(BankClient.bankingServiceURL,clientID);
   }
}
```

In the Axis client, the listener is started after the asynchronous request is sent. When the response returns, the method responsible for updating the balance stops the listener, as shown in Example 15-7.

Example 15-7 Stopping the listener in Axis client

```
public void setNewBalance(double newBalance)
{
    balance.setText(Double.toString(newBalance));
    listener.Stop();
}
```

This method of stopping the listener is implemented to keep the example simple. A response listener is able to process multiple responses although it is single-threaded. A better way of implementing the response listener is to start it when the graphical client starts, and leave it running for the duration of the client's lifetime, stopping it when the user closes the graphical client.

Note: The Axis client must have the library files com.ibm.mq.soap.jar and com.ibm.mq.soapasync_MA0v.jar in the classpath. Additional WebSphere MQ jar files are also required. See Table 9-3 for the complete list.

At this stage, the asynchronous implementation of the credit method call is completed. If the Web Service listener created during the deployment of the Web Service is started, the client can invoke the Web Service.

15.3 Executing the .NET client

The client is ready to demonstrate the use of the asynchronous credit method. A delay is inserted into the credit method on the service in order to help illustrate the asynchronous call. This is done using the set delay method that makes the call sleep for the specified number of seconds before returning a response as shown in the code shown in Example 15-8.

Example 15-8 Using the set delay method

```
public static int delay = 0;

[WebMethod] [SoapRpcMethod]
public void setDelay(int delaySpecified)
{
   //convert number of seconds for delay to milliseconds
   delay = delaySpecified * 1000;
}

[WebMethod] [SoapRpcMethod]
public bool credit(double amount)
{
   //credit the account

   //if delay has been specified
   if(delay > 0)
   {
      //make service sleep for specified number of seconds
      Thread.Sleep(delay);
   }
   return true;
}
```

Start the service listener using the **startWMQNListener** command file generated when the Web Service is deployed. See 10.5, “Deployment” on page 227 for more details.

Enter an amount in the text box and click the **Credit** button. This invokes the credit method. The balance on the graphical user interface does not change. The interface can still respond to further click actions. This allows one of the other functions to be invoked while waiting for the response. After about five seconds, the balance is updated to reflect the call-to-the-credit method.

15.4 Executing the Axis client

The client can be started to demonstrate the use of the asynchronous credit method. In this example, a delay is inserted into the credit method on the service in order to help illustrate the asynchronous call. This is done using the Thread.Sleep method to pause for five seconds:

```
Thread.Sleep(5000);
```

Start the service listener using the **startWMQJListener** command file. See 8.4, "Deployment" on page 164 for further details.

When you enter an amount in the text box and click the **Credit** button, the credit method is invoked. The balance on the graphical interface does not change, but the interface can still respond to further click actions. This allows one of the other functions to be invoked while waiting for the response. After about five seconds, update the balance to reflect the call-to-the-credit method.

16

Transactional functionality (MA0V)

This chapter discusses the transactional functionality provided in MA0V SupportPac. This functionality provides transactional control on client request operations and response operations. This chapter covers the following topics:

- Overview of MA0V transactional functionality
- Microsoft .NET client transactionality with MA0V
- Developing a Microsoft .NET client to use transactional functionality
- Axis client transactionality with MA0V
- Developing a Java client to use transactional functionality

© Copyright IBM Corp. 2006. All rights reserved.

16.1 Overview of MA0V transactional functionality

WebSphere MQ V6 provides transactional options on the WebSphere MQ transport for SOAP listeners. The listeners allow Web Services to be started within the context of a transaction. This means that after the listener receives a request for a Web Service to be started, the process of executing the service can be performed under transactional control.

This provides the facility for the entire execution phase to be backed out if any aspect of the service fails. If a Web Service is, for example, updating the database, and the database access fails, this allows the entire execution phase to be backed out, so that the original request message is left on the request queue.

It is up to the customer's service code to decide the circumstances under which a transaction must be committed or backed out. WebSphere MQ transport for SOAP does not make any input into this process.

When considering the ability to impose transactional control over Web Services, there are three totally independent levels at which this control can be applied:

- The client issuing a request for the service to be started
- The WebSphere MQ transport for SOAP listener executing the service
- The client receiving a response from the service that was started

The three levels of transactional control is illustrated in Figure 16-1.

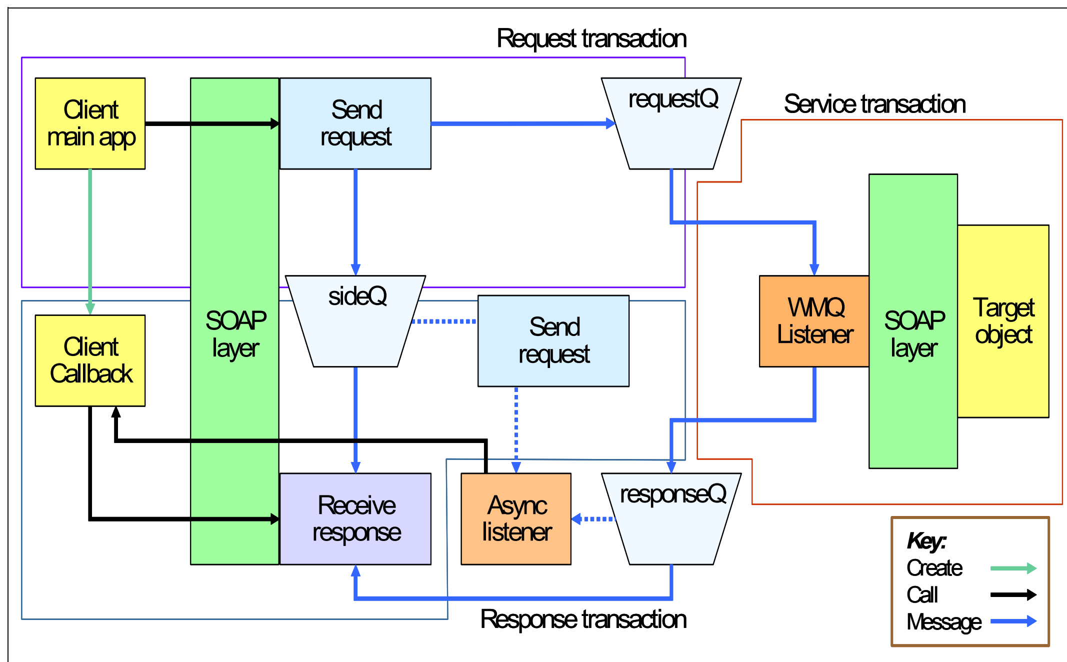


Figure 16-1 Illustrating the three levels of transactional control

Figure 16-1 represents the same asynchronous design as shown in Figure 14-1 on page 309 of Chapter 14, “Long-term asynchronous functionality (MA0V)” on page 303, with the three large boxes illustrating the three different transactional controls. Refer to 14.4, “Developing a client to use long-term asynchrony” on page 307 for a detailed explanation of the flow.

Figure 16-1 illustrates the tasks performed as part of a transactional request by enclosing these activities within the Request Transaction box. Those activities that form a part of a transactional service execution are within the Service Transaction box, and those that form a part of a transactional response are within the Response Transaction box.

The ability to start Web Services transactionally, which is provided as part of WebSphere MQ V6, is entirely independent of the ability to post requests or receive responses from the clients transactionally. This provision for clients to start transactional control over Web Service requests and responses is not provided in the product, but is made available in MA0V SupportPac. Client transactionality assumes the use of the long-term asynchronous interface.

Axis clients can start by using one-phase transactionality or two-phase transactionality. Microsoft .NET clients start by using the two-phase style of control only.

One-phase transactionality means that local WebSphere MQ transactions are used to ensure that a request message is either delivered to the request queue or, if the request cycle does not complete successfully, to ensure that all WebSphere MQ operations in the request cycle are backed out.

Two-phase transactionality means that resources other than WebSphere MQ can participate in a request transaction or response transaction. In case of Java, WebSphere MQ is used as the transaction coordinator. In case of Microsoft .NET, Microsoft .NET Microsoft Transaction Server (MTS) is used as the transaction coordinator, and the use of WebSphere MQ as a transaction coordinator is not supported. In two-phase transactionality, resources such as databases can participate in the same transaction. Thus, if a client transaction is to be backed out, all the WebSphere MQ operations must be backed out along with all the database updates.

The use of one phase if persistent and two phase if persistent styles of transactional control is not provided by MA0V. Although this may be valuable in many real scenarios, in that, it contributes to improved performance instead of always having to process messages transactionally regardless of persistence, it has not been included for two reasons:

- To avoid complicating further the interfaces between the client code and WebSphere MQ transport for SOAP.
- To avoid the risk of unpredictable errors in the WebSphere MQ transport for SOAP software in the event of errors in the client code.

It is the responsibility of clients to own the request and response transactions and to determine under what circumstances they must be committed or aborted.

To make an existing asynchronous client transactional, it is necessary to make some modifications to the client. These are illustrated in the rest of this chapter.

Note: The client transactional functionality provided in MA0V is currently not supported by IBM.

16.2 Transactional demonstration samples

Sample clients illustrating the basic use of the transactional client functionality are included with MA0V. A script, regenTranDemoAsync.cmd/sh, is provided with MA0V. Use this to build the clients. The samples are not integrated into the Installation Verification Testing (IVT) mechanism. Therefore, run the SOAP/WebSphere MQ listeners and clients manually.

Detailed instructions about how to use the supplied transactional samples are provided in the WebSphere MQ transport for SOAP asynchronous manual. These illustrate the use of two-phase commit style of transactions only. In Microsoft .NET, only this form can be initiated directly.

16.2.1 Microsoft .NET client transactionality

To implement transactionality, modify an existing asynchronous Microsoft .NET client as follows:

- The client must be derived from the System.EnterpriseServices.ServicedComponent class.
- Deriving from the System.EnterpriseServices.ServicedComponent class in turn requires the assembly to be signed by using a strong name key.
- The client must use the [Transaction(TransactionOption.Required)] directive.
- When making requests, the client must issue ContextUtil.SetComplete() calls to commit a transaction or ContextUtil.SetAbort() calls to abort a request transaction.
- When processing responses, the client's callback function must be modified to perform the following actions:
  - Make the callback transactional. The client's callback function must issue a special WebSphere MQ transport for SOAP call in order to give the response code a transactional context. This is a call to IBM.WMQSOAP.Async.MakeTransaction(). It is normal in Microsoft .NET to derive classes that are to be used transactionally from the ServicedCompent class and specify transactional use through class attributes. However, this is not possible in the case of asynchronous callback class because callback objects must be serialized and there is no simple way in Microsoft .NET V1.1 to serialize an object that inherits from the System.EnterpriseServices.ServicedComponent class. That is the reason why this special call must be made.
  - Commit or backout the response transaction. This is done by setting the [AutoComplete] directive on the callback method so that the transaction is automatically committed when the method exits. Writing a callback

method is recommended so that it can handle any exception that is thrown while processing the response in a real scenario, and take the appropriate corrective action, so that a transaction can still be committed. This is preferable to using the ContextUtil.SetAbort() call to abort the response transaction.

Note: When building a client, the assembly must be signed with a strong name key.

As stated earlier, Microsoft .NET MTS is used as a transaction coordinator with Microsoft .NET transactional clients. The use of WebSphere MQ as a transaction coordinator is not supported. When using Microsoft .NET transactions, the transaction context is passed implicitly between the customer code and the WebSphere MQ transport for SOAP sender code by the Microsoft .NET Framework. Unlike in Axis, there is no requirement for the customer code to give the sender code the ability to participate in the transaction. This is handled directly by the infrastructure.

It is not possible for a customer's Microsoft .NET client to specify a one-phase style of transactional control. WebSphere MQ transport for SOAP checks the Microsoft .NET MTS transaction status and operates only within a unit of work if there is an MTS transactional context. This is different from Java, where the asynchronous registration calls permit the client's code to tell WebSphere MQ transport for SOAP to start within a unit of work, irrespective of whether one-phase or two-phase transactionality is being used. Microsoft .NET has internal one-phase optimization that automatically optimizes the process if it detects that only a single resource is participating in the transaction. In this eventuality, the overhead of a two-phase commit style is reduced. If the Microsoft Distributed Transaction Coordinator (DTC) detects that WebSphere MQ is the only participating resource in the transaction, it instructs WebSphere MQ to commit directly, rather than issuing a separate **prepare** command.

To take advantage of one-phase optimization, the client application must *not* have a WebSphere MQ connection open at the time of invoking a service request. Any WebSphere MQ connection in the client software must instead be handled serially with service invocation requests. When connections are used serially in this way, WebSphere MQ pooling means that the same connection handle is used internally by WebSphere MQ, thereby enabling MTS to use the one-phase optimization. If connections are not made serially, that is, a connection is already open when a service request is being made, the connection handles are separate and MTS considers that multiple resources are involved in the transaction. This results in a two-phase commit transaction with no optimization.

16.2.2 Developing a transactional Microsoft .NET client

This section demonstrates, in practical terms, the modification of an asynchronous Microsoft .NET client in order to make it transactional.

Transactional request

In the code fragment shown in Example 16-1, the changes that are necessary to enable a client to make a transactional request are highlighted in bold. These illustrate how the client request typically looks like in the source code.

Example 16-1 shows Microsoft .NET client transactional request code. This code snippet illustrates the use of the directives on the class definition and shows that the client request class must be derived from System.EnterpriseServices.ServicedComponent. It also illustrates the use of ContextUtil.SetComplete() and ContextUtil.SetAbort() calls to commit or back out the request transaction.

Example 16-1 Microsoft .NET client transactional request code

```
using System.EnterpriseServices;

[Transaction(TransactionOption.Required)]
public class TranRequest : ServicedComponent
{
   public TranRequest()
   {
   }

   public void TestRequests(int count, String clientId, String tranArg)
   {
      try
      {
         // Register the WMQSOAP URL extension with DotNet
         IBM.WMQSOAP.Register.Extension();

         // Create a context object to pass to the WMQSOAP Async class
         testStateClass requestContext = new testStateClass();

         IBM.WMQSOAP.Async.Request(requestContext, clientId);

         System.Single res = -1;
         StockQuoteDotNetTran stockobj = new StockQuoteDotNetTran();
         try
         {
            res = stockobj.getQuoteTran(tranArg);
```

```
            System.Console.WriteLine("ERROR - a \"no response expected exception\" was 
expected");
            System.Console.WriteLine("Aborting transaction");
            ContextUtil.SetAbort();
         }
         catch (IBM.WMQSOAP.AsyncResponseExpectedException e)
         {
            if (e.CompletionCode != MQC.MQCC_OK)
               throw(e);

            System.Console.WriteLine("Request transaction can be committed");
            ContextUtil.SetComplete();
         }
      }
      catch (System.Exception e)
      {
         Console.WriteLine("\n>>> EXCEPTION WHILE RUNNING SQCS2DotNetAsyncReqRespTran
DEMO <<<\n" + e.ToString());
         System.Console.WriteLine("Calling ContextUtil.SetAbort");
         ContextUtil.SetAbort();
      }
   }

// ... Main program not shown
}
```

Transactional response

The code fragment in Example 16-2 illustrates the changes that are necessary for an asynchronous client's callback class to make the response transactional.

Example 16-2 Microsoft .NET client transactional response code

```
using System.EnterpriseServices;
using System.Reflection;

using IBM.WMQ;

[assembly: AssemblyKeyFileAttribute("tranDemo.snk")]

[Serializable]
public class testStateClass : IBM.WMQSOAP.AsyncCallback
{
   public testStateClass()
   {
```

```
   }

   // This is the overridden method that is invoked by WMQSOAP
   // when a response is received for this client. It prints the
   // result of the request.

   [AutoComplete]
   public override void CallbackFunction()
   {
      try
      {
         if (IBM.WMQSOAP.Async.MakeTransaction(this)) return;

         // Create a proxy object so we know what URL to point
         // the response listener to.
         StockQuoteDotNetTran stockobj = new StockQuoteDotNetTran();

         IBM.WMQSOAP.Async.Response(this);

         System.Single res = stockobj.getQuoteTran("unused dummy parameter");
         Console.WriteLine("ASYNC response is: " + res);
      }
      catch (System.Exception e)
      {
         Console.WriteLine("\n>>> EXCEPTION WHILE RUNNING SQCS2DotNetAsyncReqRespTran
DEMO <<<\n" + e.ToString());

         //In a real case the exception would be handled
         //so the transaction can be completed.
         //But if the transaction was to be aborted it
         //would be done as follows:

         //ContextUtil.SetAbort();
      }

   }

   // Rest of client code not shown
}
```

This code snippet illustrates the use of the [AutoComplete] directive on the CallbackFunction() and the call to Async.MakeTransaction() to give the invocation of the response callback a transactional context. It also demonstrates signing the assembly with the tranDemo.snk key file.

Signing the client assembly with a strong name key

The client must be derived from the System.EnterpriseServices.ServicedComponent class. Therefore, the client assembly must be signed. In order to do this, a key file must be generated with the sn utility. This utility is provided as part of the Microsoft .NET Framework software development kit (SDK). To generate a key file, perform the following tasks:

1. Invoke the sn utility as shown in Example 16-3.

Example 16-3 Invoking the sn utility

```
c:\temp>sn -k tranDemo.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 1.1.4322.573
Copyright (C) Microsoft Corporation 1998-2002. All rights reserved.

Key pair written to tranDemo.snk

c:\temp>
```

> **Note:** The bin directory of the Microsoft .NET Framework SDK must be in PATH for you to use the sn utility.

2. Place a reference to this key file in the AssemblyInfo.cs file for the assembly or directly into the client with a directive of the [assembly: AssemblyKeyFileAttribute("tranDemo.snk")] form.

> **Note:** A runtime error occurs when running the client if the assembly is not signed in this manner.

Refer to the Microsoft .NET Framework documentation for more information about signing assemblies.

16.3 Axis client transactionality

Client transactionality in the Axis environment is somewhat different from that of the Microsoft .NET environment. The key difference is that in two-phase transactionality, WebSphere MQ is used as the transaction coordinator. A begin() call is used on a MQQueueManager object to initiate two-phase distributed transactions. The backout() and commit() calls are then used to indicate whether transactional units of work must be aborted or committed, respectively. The client software takes on the responsibility of deciding whether a

request or response transaction must be committed or aborted. WebSphere MQ transport for SOAP does not influence this decision. All that SOAP/WebSphere MQ does in this regard is to perform syncpointed operations, having been told that the client is operating transactionally.

The transaction status is passed explicitly between the client's client software and WebSphere MQ. This is accomplished with a special queue manager open call that allows WebSphere MQ transport for SOAP to use the same queue manager connection handle as that used at the client level, and therefore, to participate in the same transaction. Ensure that these calls are actioned correctly. Otherwise, WebSphere MQ transport for SOAP is unable to participate in a client's transaction context.

Note: If a Web Service implementation is required to participate in an execution transaction under the control of the WebSphere MQ transport for SOAP listener, the service must make a special style connect call to ensure that it inherits the same queue manager connection as that being used by the listener. This is illustrated later in this chapter.

In order to make a long-term asynchronous Axis client transactional, the following tasks must take place:

1. When processing requests, the client must connect to the queue manager using the MQC.ASSOCIATE_THREAD queue manager association property. This allows the WebSphere MQ transport for SOAP to participate as part of the same transaction that is being owned and managed by the client.

   Note: To make this call, the client must know the queue manager to which the sender is attempting to connect. In a nontransactional client, it does not have to know this information at the user level.

2. The client must inform WebSphere MQ transport for SOAP that it is to start transactionally. This is through an additional boolean "isTransacted" argument on the Async.Request() call that advises the sender code that the client is about to make an asynchronous request. This step is not necessary in Microsoft .NET because, in this environment, the sender software can determine for itself whether or not it is executing in a transactional context.
3. When making requests, the client must issue qm.commit() calls to commit a request transaction or qm.backout() calls to abort a request transaction.
4. When processing responses, the client's callback function must be modified to perform the following actions:
   - Connect to the queue manager using the MQC.ASSOCIATE_THREAD queue manager association property, as in the client request code.

– Inform WebSphere MQ transport for SOAP that it is about to start transactionally. This is through an additional argument in the Async.Response() call that advises the sender code that the client is about to issue an asynchronous response call. This too is not necessary in Microsoft .NET because the infrastructure is able to determine for itself whether or not it is operating in a transactional context.
– Commit or back out the response transaction. Specific commit() call does not have to be added to commit a transaction because this is the default when the queue manager is disconnected. In a real scenario, in the event of an error in the response, it is recommended that a service take the necessary corrective action so that a transaction is still committed. This is preferable to using the qm.backout() call to abort the response transaction.

16.3.1 Developing a transactional Axis client

This section demonstrates, in practical terms, how an asynchronous Axis client is modified to make it transactional.

Transactional request

In the code shown in Example 16-4, the changes necessary to enable a client to make a transactional request are highlighted in bold. This illustrates how the client request changes described earlier typically look in the source code.

Example 16-4 Axis client transactional request code

```
   private static final int MQRC_NO_EXTERNAL_PARTICIPANTS=2121;
   private String qMgrString = "WMQSOAP.DEMO.QM";

   // This method connects to the SOAP demo queue manager and starts a
   // transaction.
   private MQQueueManager connToQMgr() throws MQException
   {
      MQQueueManager _qm = null;
      try
      {
         Hashtable props = new Hashtable();
         props.put(MQC.MQ_QMGR_ASSOCIATION_PROPERTY, new
Integer(MQC.ASSOCIATE_THREAD));
         _qm = new MQQueueManager(qMgrString, props);

         MQException.logExclude(new Integer(MQRC_NO_EXTERNAL_PARTICIPANTS));
         _qm.begin();
         MQException.logInclude(new Integer(MQRC_NO_EXTERNAL_PARTICIPANTS));
```

```
   }
   catch (MQException e)
   {
      if (MQRC_NO_EXTERNAL_PARTICIPANTS != e.reasonCode) throw e;
   }

   return _qm;

}


// This is the method that makes the request for a Stock Quote
// to the Axis stock quote service.
private void TestRequests( int count, String clientId, String tranArg )
{
   MQQueueManager _qm = null;

   // Must register WMQ transport extensions before doing SOAP/MQ
   Register.extension();

   try
   {
      _qm = connToQMgr();

      for(int i=0; i < count; i++)
      {
         // Create a context object to pass to the
         // WMQSOAP Async class
         testStateClass requestContext = new testStateClass();

         boolean isTransacted = true; // In a transaction
         Async.Request(requestContext, clientId, isTransacted);

         // Use the locator to get a handle to the service on a specific WSDL Port
         StockQuoteAxisService locator = new StockQuoteAxisServiceLocator();

         StockQuoteAxis service=null;
         service = locator.getSoapServerStockQuoteAxis_Wmq();

         // Invoke the target service
         try
         {
            float result = service.getQuoteTran( tranArg );
         }
         catch (Exception e)
```

```
          {
             if (!(e instanceof AsyncResponseExpectedException))
                throw(e);

             AsyncResponseExpectedException wmqe =
(AsyncResponseExpectedException)e;
             if (wmqe.completionCode != MQTrace.MQCC_OK)
                throw e;

             _qm.commit();
             _qm.disconnect();
          }
       }

       System.out.println("Async service(s) successfully requested.");
    }
    catch ( Exception e )
    {
       System.out.println("\n>>> EXCEPTION WHILE RUNNING
SQAxis2AxisAsyncReqRespTran DEMO <<<\n");
       e.printStackTrace();
       if (_qm != null)
       {
          try
          {
             _qm.backout();
          }
          catch (MQException e2)
          {
             System.out.println("Exception raised trying to backout");
          }
       }
       System.exit( 2 );
    }
 }
```

This sample code shows the following:

- The client must know the target queue manager. In this instance, it is WMQSOAP.DEMO.QM.
- The use of the ASSOCIATE_THREAD property when connecting to the queue manager is necessary to allow WebSphere MQ transport for SOAP to participate in the client's transaction.

- The additional argument to Async.Request() to tell the sender code to start within a unit of work.
- The use of backout() or commit() calls to abort or complete the request transaction as appropriate.

Transactional response

The code in Example 16-5 illustrates the changes that must be made to an asynchronous client's callback class to make the response transactional.

Example 16-5 Axis client transactional response code

```
public class testStateClass extends AsyncCallback
{
   private transient MQQueueManager _qm1=null;

   public void CallbackFunction()
   {
      // Register the WMQSOAP URL extension with Axis
      Register.extension();

      try
      {
         _qm1 = connToQMgr();

         StockQuoteAxisService locator = new StockQuoteAxisServiceLocator();

         StockQuoteAxis service=null;
         service = locator.getSoapServerStockQuoteAxis_Wmq();

         boolean isTransacted=true;
         Async.Response(this, isTransacted);

         float result = service.getQuoteTran( "DUMMY" );

         System.out.println("ASYNC RPC reply is: " + result);
         _qm1.disconnect();
      }
      catch (Exception e)
      {
         System.out.println("Exception in CallBackFunction: " + e.toString());

         //In a real case the exception would be handled so the transaction can be 
completed.
         //But if the transaction was to be aborted it would be done as follows:
         /* try
```

```
         {
         _qm1.backout();
         _qm1.disconnect();
         }
         catch (MQException e1)
         {
         System.out.println("Exception raised backing out response: " +
e1.toString());
         } */

      }
   }
}
```

This sample code illustrates that the queue manager uses the ASSOCIATE_THREAD property on the connection. The actual implementation of connToQmgr can be the same, as in the case of request. This sample code also illustrates the additional flag on the Async.Response() call. It also illustrates the use of the [AutoComplete] directive on the CallbackFunction() and the call to Async.MakeTransaction() to give the invocation of the response callback a transactional context.

Service participating in an execution transaction

As mentioned earlier, if an Axis service is required to perform WebSphere MQ operations within the same transactional context as the WebSphere MQ transport for SOAP listener, a special style of connection call must be made to ensure that the service inherits the same connection handle as that being used by the listener. The connection is made in the following manner:

```
MQQueueManager qmgr =
MQEnvironment.getQueueManagerReference(MQC.ASSOCIATE_THREAD, “MYQMGR”);
```

16.4 Summary

This chapter discussed the functionality of the SupportPac MA0V. After an introduction to the transactional functionality provided in MA0V, this chapter discussed the steps that are necessary to make Microsoft .NET and Axis asynchronous clients transactional. Illustrations, with example code for each type of client, were also provided.

17

Implementing transactionality

Chapter 16, “Transactional functionality (MA0V)” on page 339 introduced the additional transactionality concepts provided by the MA0V SupportPac. The functionality introduced by the SupportPac allows the transactionality concept to be implemented on the client for the Web Services called asynchronously.

This chapter provides information about how to implement client side transactionality. This is demonstrated by modifying the BankClient classes introduced in Chapter 11, “.NET client” on page 243, Chapter 9, “Axis client” on page 187, and Chapter 13, “WebSphere Application Server client” on page 289.

© Copyright IBM Corp. 2006. All rights reserved.

17.1 Overview

The standard WebSphere MQ V6 install allows transactions to be implemented on the service side. In other words, they are created with a service that rolls back all the operations if a particular operation fails.

The enhanced functionality provided by SupportPac allows two particular sections of work in the client to be placed within a transaction:

- The following operations form a part of the call to the Web Service:
  - Successful placement of the request on to the request queue, and by implication, the transmission queue
  - Successful placement of the callback object within the side queue
- The following operations form a part of processing the response:
  - Browsing of the response by the asynchronous response listener
  - Retrieval of the callback object from the side queue
  - Call to the callback method

> **Note:** The asynchronous response listener does not operate the transactionally because it browses only messages. However, it does fall within the scope of the transaction.

The Web Service implemented in Chapter 10, ".NET Web Service" on page 213, Chapter 8, "Axis Web Service" on page 159, and Chapter 12, "WebSphere Application Server Web Service" on page 269 provides four simple banking operations. These are listed in Table 17-1.

Table 17-1 BankingService method description

Method	Description
credit	Adds specified amount to current balance
debit	Removes specified amount for transfer to account ID provided. This implementation simply subtracts the amount specified from the balance. If the amount is greater than the balance, an exception is thrown.
getBalance	Returns the current balance
getStatement	Returns an array of BankingOperation objects

In this example, the transactionality discussions are concentrated on one particular method. The method chosen for demonstration is the *credit* method. In order to aid the development of the transactional functionality, a delay is inserted into the method. The method is adjusted in the Axis Web Service as shown in Example 17-1.

Example 17-1 Adjustment of the credit method in the Axis Web Service

```
public boolean credit(double amount)
{
   log("Trace::in::credit(amount = " + amount + ")");
   try {
         Thread.sleep(5000);
   }
   catch (InterruptedException ex)
   {
      log("Trace::in::credit - interrupted exception thrown -
"+ex.getStackTrace());
   }
...
}
```

Similarly, the method is adjusted in the .NET Web Service as shown in Example 17-2.

Example 17-2 Adjustment of the credit method in the .NET Web Service

```
[WebMethod] [SoapRpcMethod]
public bool credit(double amount)
{
   //Credit the account
   account_balance += amount;
   //if delay has been specified
   if(delay > 0)
   {
      //make service sleep for specified number of seconds
      Thread.Sleep(delay);
   }
   return true;
}
```

This change was made to only illustrate the asynchronous nature of the client. Client side transactions can be implemented with no changes to the Web Service.

The earlier scenario in Chapter 9., “Axis client” on page 187 and Chapter 11., “.NET client” on page 243 illustrated the creation of a graphical client to invoke the Web Service. It is this graphical client that is being modified in this chapter.

17.2 Java

This section demonstrates invoking a Web Service within a transaction from a Java Web Service client.

17.2.1 Invoking the service within a transaction

As discussed in Chapter 16, “Transactional functionality (MA0V)” on page 339, WebSphere MQ is used to coordinate the transaction. This means that the client must be connected to an appropriate WebSphere MQ queue manager in order to issue *begin*, *commit*, and *backout* calls.

In order to isolate this functionality, a helper class called TransactionHelper is created.

TransactionHelper.java

TransactionHelper implements the methods shown in Table 17-2.

Table 17-2 TransactionHelper methods

Method stub	Description
TransactionHelper()	Basic constructor. Does nothing.
TransactionHelper(String QMName)	Constructor that assigns qmName to an instance variable
MQQueueManager connToQMgr()	Method to connect to the queue manager and begin the transaction
commitTransaction()	Commits a transaction
backoutTransaction()	Backs out a transaction

Apart from the constructors, these methods merit some more discussion. The first one to be discussed is connToQMgr, as it is this method that starts the transaction. The source code for the method is shown in full in Example 17-3.

Example 17-3 connToQMgr method

```
public MQQueueManager connToQMgr() throws MQException
{
   qm=null;
   try {
      Hashtable props=new Hashtable();
      props.put(MQC.MQ_QMGR_ASSOCIATION_PROPERTY,
      new Integer(MQC.ASSOCIATE_THREAD));
      qm = new MQQueueManager(qMgrName, props);

      MQException.logExclude(
               new Integer(MQRC_NO_EXTERNAL_PARTICIPANTS));
      qm.begin();
      MQException.logInclude(
               new Integer(MQRC_NO_EXTERNAL_PARTICIPANTS));
   }
   catch (MQException ex)
   {
      if (MQRC_NO_EXTERNAL_PARTICIPANTS != ex.reasonCode)
      {
         throw ex;
      }
   }
   return qm;
}
```

Note: qMgrName is an instance variable set by one of the constructors. It is assigned a default value of QM_localToSvc if the other constructor is used. This can be found at <WebSphere MQ install directory>\Java\lib\.

1. The method begins by creating a hashtable to hold the properties that are to be set when opening a connection. This is used to set the MQ_QMGR_ASSOCIATION_PROPERTY to MQC.ASSOCIATE_THREAD. Setting this property allows WebSphere MQ for SOAP Transport Sender to participate in the same transactions as the client.

The method calls before and after beginning the transaction, MQException.logExclude and MQException.logInclude, are important. If an external resource, such as a database, is participating in the transaction, WebSphere MQ does not generate warning messages. If no external resource is participating, a warning message is generated by WebSphere MQ. These two function calls prevent this warning message from occurring.

> **Important:** In order to facilitate this direct connection to a queue manager, an additional library file, connector.jar, must be referenced in the classpath.

2. The next step is to create a queue manager object with these properties. The MQRC_NO_EXTERNAL_PARTICIPANTS exception is then excluded from the log before the connection is established.
3. The begin call is then issued, signalling the beginning of the transaction. The use of the begin call indicates a two-phase transaction, rather than a one-phase transaction. See Chapter 16, “Transactional functionality (MA0V)” on page 339.

   On completion of this step, the error message logging is returned to normal. The queue manager object that is created is then returned to the calling method.

> **Note:** The class retains a reference to the queue manager method, which it uses in the commitTransaction and backoutTransaction methods.

The commitTransaction method and backoutTransaction method are both fairly straightforward. The commitTransaction method is shown in Example 17-4. It calls two methods on the queue manager, one to commit the transaction and the other to end the connection. Exceptions, if any, are passed on to the calling method.

Example 17-4 commitTransaction method

```
public void commitTransaction() throws MQException
     {
      if (qm!=null)
      {
         // call Queue Manager commit method, then disconnect
```

```
        qm.commit();
        qm.disconnect();
      }
    }
```

The backoutTransaction is shown in Example 17-5. This method calls the backout method on the queue manager. Errors, if any, are logged, and the method exits the program.

Example 17-5 backoutTransaction method

```
public void backoutTransaction()
{
   if (qm!=null)
   {
      // try to backout, outputting any exceptions
      try {
         qm.backout();
      }
      catch (MQException ex)
      {
         System.out.println("Exception raised during backout:");
         System.out.println(ex.getStackTrace());
      }
   }
   System.exit(2);
}
```

After the creation of this helper class, use it along with the libraries supplied with the SupportPac to implement a transaction on a call to a Web Service. This functionality is implemented within the BankingGUI class.

BankingGUI.java

All the changes here take place within the startAsyncCreditCall method. This call initiates a transaction flag and the TransactionHelper class. Example 17-6 shows variable initialization.

Example 17-6 Variable initialization

```
private void startAsyncCreditCall(double amountToCredit) throws
Exception
{
```

```
   boolean isTransactional=true;
   TransactionHelper th=new TransactionHelper();
...
}
```

The flag is passed to WebSphere MQ transport for SOAP to indicate that the call is wrapped in a transaction. The helper class constructor takes no parameters because the default queue manager name, QM_localToSvc, is used in the code demonstrated in this example.

The next step is to indicate to WebSphere MQ that an asynchronous transaction is starting. This is done by using the TransactionHelper class for this.

WebSphere MQ transport for SOAP is then notified that the calling method is being executed asynchronously within a transaction, as shown in Example 17-7.

Example 17-7 Beginning the transaction

```
qm=th.connToQMgr();
Async.Request(bankClient,clientID,isTransactional);
```

Notes:

- The returned reference to the queue manager object is not used here.
- The transaction flag created in the previous code snippet is used.

The service can be invoked and committed if the transaction is successful. If the call fails, use backout method, as shown in Example 17-8.

Example 17-8 Invoking the method

```
try {
...
   try {
               boolean result=service.credit(amountToCredit);
   }
   catch (AsyncResponseExpectedException ex)
   {
      if (ex.completionCode!=MQTrace.MQCC_OK)
      {
         throw ex;
      }
      th.commitTransaction();
   }
```

```
}
catch (Exception ex)
{
   System.out.println("Exception occurred: "+ex.getStackTrace());
   th.backoutTransaction();
}
```

The service is called immediately after the second *try* statement. The AsyncResponseExpectedException is thrown, indicating that the request is sent. The transaction is committed using a call to the commitTransaction method in the helper class. If an exception occurs, the backoutTransaction method is used.

In the event that the commit of the transaction fails, the request message is left on the request queue. To illustrate this, the client side method is altered to deliberately generate an exception between the beginning and the end of the transaction, as shown in Example 17-9.

Example 17-9 Modified startAsyncCreditCall method designed for commit to fail

```
try {
...
   qm=th.connToQMgr();
   // prepare Async request
   Async.Request(bankClient,clientID,isTransactional);
   // get handle on service
   BankingServiceServiceLocator locator=new 
BankingServiceServiceLocator();
   BankingService 
service=locator.getBankingServiceBankingService_Wmq(new 
java.net.URL(BankClient.bankingServiceURL));
   try {
         boolean result=service.credit(amountToCredit);
   }
   catch (AsyncResponseExpectedException ex)
   {
      if (ex.completionCode!=MQTrace.MQCC_OK)
      {
         throw ex;
      }
      int i=0;
      int j=0;
      int k=i/j;
      th.commitTransaction();
      ...
   }
```

```
}
catch (Exception ex)
{
   System.out.println("Exception occurred: "+ex.getStackTrace());
   th.backoutTransaction();
   ...
```

The divide by zero causes an exception before the transaction is committed. The code therefore, catches this exception and backs out the transaction. As a result, the request is not sent.

17.2.2 Processing the response within a transaction

To implement the request processing within a transaction, the TransactionHelper class from the previous method must be used. This time, it is the CallbackFunction that requires modification.

BankClient.java

As with invoking the service, the initial step here is to set the transaction flag and instantiate the helper class, as shown in Example 17-10.

Example 17-10 Instantiating variables

```
public void CallbackFunction()
{
   boolean isTransacted=true;
   TransactionHelper th=new TransactionHelper();
   ...
```

The next step is to connect to the queue manager and begin the transaction.

This is followed by notification WebSphere MQ transport for SOAP that the response is about to be retrieved, as shown in Example 17-11. This code snippet starts the transaction and retrieves the result of the method. Note the dummy parameter that is passed in. This is a part of the framework for asynchronous calls, requiring a call to retrieve the response.

Example 17-11 Retrieving the response

```
th.connToQMgr();
Async.Response(this, isTransacted);
boolean result = service.credit(0.0);
```

As part of this transaction, the new balance is retrieved before closing the transaction, as shown in Example 17-12.

Example 17-12 Retrieving the balance and closing the transaction

```
try
{
   ...
   boolean result = service.credit(0.0);
   if (result)
   {
      newBalance=service.getBalance();
      gui.setNewBalance(newBalance);
      th.commitTransaction();
   }
   else{
      System.out.println("result is false");
   }
}
catch (Exception e)
{
   System.out.println("Exception in CallBackFunction: " +
e.toString());
   th.backoutTransaction();
}
```

After the balance is retrieved and the graphical user interface updated, the transaction is committed. In the case of an exception, the transaction is backed out.

In order to demonstrate an exception occurring in this scenario, the callback method is modified to generate an exception. In this case, a divide by zero is again used to generate the exception, as shown in Example 17-13.

Example 17-13 Generating an exception in the Callback method

```
try
{
   ...
   th.connToQMgr();
   Async.Response(this, isTransacted);
   boolean result = service.credit(0.0);

   if (result)
   {
      newBalance=service.getBalance();
```

```
      int i=0;
      int j=0;
      int k=i/j;
      gui.setNewBalance(newBalance);
      th.commitTransaction();
}
catch (Exception e)
{
   th.backoutTransaction();
   ...
}
```

The result of this is that the response from the service is not correctly processed. When the transaction is backed out, the response message is returned to the response queue. This is a dynamic queue, created by the asynchronous framework, with a name based on the service name and client ID. In this chapter, the response queue is called BANKING.SERVICE_JmyBankClient42C6762D03957520. When the transaction is backed out, the response message is left on the queue.

Note: To delete the dynamic queue, use the amqwAsyncConfig utility. In this example, the following command is used:

```
amqwAsyncConfig -qm QM_localToService -clientId myBankClient -baseQ
BANKING.SERVICE
```

For more information about the dynamic response queues and the amqwAsyncConfig utility, see 14.10.1, “Removing queue mapping entries from the side queue” on page 321.

17.3 Microsoft .NET

The .NET Web Service client developed in Chapter 11, “.NET client” on page 243, for implementing long-term asynchronous Web Services clients, is further modified to demonstrate transactionality. The credit button event handler code calls the credit method in the Web Service in a transaction along with a series of other calls. If even one call fails, the transaction is backed out.

17.3.1 Invoking the service within a transaction

To modify the .NET client code to implement transactions, the actions described in the following sections are required.

Note: The code includes comments such as Step1, Chapter 16.4.1, indicating which part of the code corresponds to the steps to modify the client code to implement transactions.

Implementing a callback function

This class is a derivative of the IBM.WMQSOAP.AsyncCallback. Although it is not required immediately, it is defined and used later in the implementation of a request to the Web Service. The callback is specified in the asynchronous request to the Web Service. It constitutes a response listener, which is required to process the response received when a request is sent to the Web Service. See 14.6.4, "Implementing an asynchronous callback" on page 315 for details on the callback function. This callback function is similar to the one implemented for a nontransactional long-term asynchronous client. The only difference is that the callback function is made transactional with the following code:

```
if (IBM.WMQSOAP.Async.MakeTransaction(this)) return;
```

The code snippet shown in Example 17-14 shows the callback function implemented for the .NET client. The line of code that makes it transactional is in bold type text.

Example 17-14 The implementation of the callback function

```
[Serializable] class creditCallback : IBM.WMQSOAP.AsyncCallback
{
   [AutoComplete]
   public override void CallbackFunction()
   {
      try
      {
         //Console.WriteLine("  CallBackFunction: intran=" +
         ContextUtil.IsInTransaction);
         if (IBM.WMQSOAP.Async.MakeTransaction(this)) return;

         // Create a proxy object so we know what URL to point the
response
         //listener to.
         BankingService service = new BankingService();
         //then point the response listener to it
         IBM.WMQSOAP.Async.Response(this);
```

```
         //make the initiation Web service call
         bool creditSuccessful = service.credit(0.0);
      }
      catch (System.Exception e)
      {
         System.Diagnostics.Debug.WriteLine("\n>>> EXCEPTION WHILE
RUNNING
         BankingService Credit method call<<<\n" + e.ToString());
      }
   }
}
```

An addition to the callback function is the [AutoComplete] directive above the method. This directive allows the transaction to automatically commit when the method exits, unless explicitly stated otherwise. The code snippet in Example 17-15 shows the [AutoComplete] directive.

Example 17-15 Autocomplete implementation on the callback function

```
[AutoComplete]
public override void CallbackFunction()
{
}
```

Making a transactional request

The callback described earlier facilitates the client knowing what to do when a response comes back from an asynchronous request. Now a request can be made. The requests are grouped into a method call called performCreditTransaction(). This method call resides within a class called creditTransaction. The method contains the call to the credit method and other operations within the transaction. The other operations are a simple add operation and a divide by zero operation. The divide by zero operation causes the transaction to fail. If commented out, the transaction completes and gets committed.

To make a transactional request, perform the following steps:

1. Before a request is made to send SOAP messages over WebSphere MQ, register WebSphere MQ as the transport mechanism using the Register.Extension() call. This call has already been performed in the .NET application forms constructor.

2. Create a new instance of the object containing the callback function. The Async.Request method is then called with this object and a specified identification as parameters. This ties up the request to the response.
3. Finally, make the Web Service call and the other operations. The call, if successful, throws an AsyncResponseExpectedException. The exception thrown must have a completion code of MQC.MQCC_OK to reflect the success of the asynchronous call.
4. In order to specify that the requests should be made a part of a transaction, the class must specify the [Transaction] directive above the class definition. The transactional request class must also be derived from the System.EnterpriseServices.ServicedComponent class.
5. The client project must import the System.EnterpriseServices dll. Along with this dll, the mqsoap dll, mqsoapasync_ma0v dll, and the amqmdnet dll must be included in the project in order to make use of the WebSphere MQ Transport for SOAP. The project must also be made aware of this by adding the code shown in Example 17-16 to the list of namespaces to use.

Example 17-16 Declaring the enterprise services and mqsoap namespaces

```
using IBM.WMQSOAP;
using IBM.WMQ;
using System.EnterpriseServices;
```

6. ContextUtil.SetComplete() and ContextUtil.SetAbort are two classes from the System.EnterpriseServices namespace. These two classes are used to commit transactions if no errors are thrown. The ContextUtil.SetComplete is called after the credit method, and the addition and the division operations in order to complete the transaction.

   The code snippet in Example 17-17 shows the transactional request, with the most important parts highlighted in bold text.

Example 17-17 Implementation of the transactional request

```
//Make a client make a transactional request
//STEP 1 of CHAPTER 16.3: The client must use the [Transactional]
directive
[Transaction(TransactionOption.Required)]
public class creditTransaction : ServicedComponent //STEP 2 of CHAPTER
16.3:
{
   public void performCreditTransaction(double amount)
   {
      try
      {
         bool creditSuccessful = false;
```

```
         BankingService service = new BankingService();
         //synchronous call to delay the response from the Web service
         service.setDelay(10);

         //instantiate the call back object to pass to the WMQSOAP
Async class
         creditCallback callBack = new creditCallback();
         //create a client request ID
         string requestClientID = "bankRequest";
         //Finally call IBM.WMQSOAP.Async.Request passing it the
callBack
         //object and the requesting client ID
         Async.Request(callBack, requestClientID );

         try
         {
            creditSuccessful = service.credit(amount);
         }
         catch (AsyncResponseExpectedException ex)
         {
            //if a WMQ error is thrown with a completion code 'OK',
then the
            //request has been successful
            if(ex.CompletionCode != MQC.MQCC_OK)
               throw ex;
            System.Console.WriteLine("Completing transaction");
            //also do something that will fail
            int i = 1;
            int j = 0;
            int k = i + j;
            //This divide by zero will cause transaction to fail
            //comment it out for transaction to complete
            int l = i/j;
            //STEP 3 of CHAPTER 16.3: Context.Util.SetComplete commits
the
            //transaction
            ContextUtil.SetComplete();
         }

         //STEP 3 of CHAPTER 14.5: Instantiating the asynchronous
response
         //listener. The request has been made asynchronously, but the
         //response need to come back
         //Start up the response listener
         //create an instance of our error handler
```

```
            AsyncErrorHandler listenerErrorHandler = new
AsyncErrorHandler();

            //now listen for a response
            ResponseListener listener = new ResponseListener(service.Url,
                                            requestClientID);
        }
        catch(System.Exception ex)
        {
            ContextUtil.SetAbort();
        }
    }
}
```

7. Finally, the client code assembly must be signed with a strong name. A cryptographic key pair is used to achieve this. To create a key pair, perform the following tasks:
   a. In the Windows command prompt, change the current directory to the location in which to store the created key pair. In this case, the chosen location is C:\REDBOOK\dotNETService\KeyPair.
   b. Type `sn -k` *<the desired name for the key pair file>*. In this case, it is `bankingService.snk` as shown in Figure 17-1.

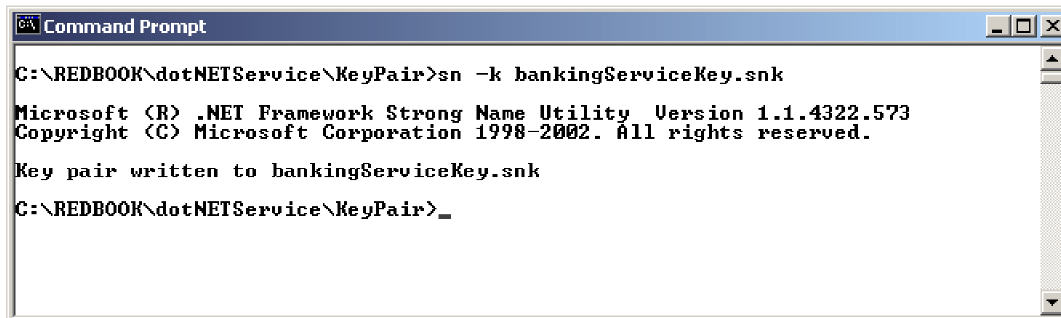


Figure 17-1 Generating a key pair for signing the assembly with a strong name

c. After the key pair is created, use the AssemblyKeyFileAttribute with the key that is generated in order to sign the assembly with a strong name as shown in Example 17-18.

Example 17-18 Code for signing the assembly with a strong name

```
[assembly:
AssemblyKeyFileAttribute(@"C:\REDBOOK\dotNETService\KeyPair\bankingServ
iceKey.snk")]
```

d. For the AssemblyKeyFileAttribute to be recognized, import the System.Reflection namespace into the project, and make the project made aware of this by declaring it:

```
System.Reflection;
```

At this stage, the implementation is complete. If the .NET client application is run, the credit button invokes a transaction, which makes a call to the Web Service, adds two numbers, and divides by zero. The transaction fails if the line of code that divides an integer by zero is not commented out. If the line of code that divides an integer by zero is commented out, the transaction is committed and the account balance reflects the credit.

17.4 Summary

The client side transactionality is illustrated in this chapter. The basic concepts are demonstrated using the Web Services clients developed from this book's earlier chapters. Java and C# code are modified to illustrate how to use client-side transactions, because the method calls are different for the two environments. The basic building blocks provided in this chapter allow you to implement client transactionality that is more complex and is of more practical use.

Part 5

Web Services and WebSphere MQ clustering

This part discusses the benefits of using WebSphere MQ clustering with Web Services and presents an example scenario pertaining to the use of this technology with WebSphere MQ transport for SOAP.

© Copyright IBM Corp. 2006. All rights reserved.

18

Using WebSphere MQ clustering with Web Services

This chapter discusses how WebSphere MQ clustering can be used with Web Services. In addition to this, it discusses high availability with respect to the clustering technology in WebSphere MQ, and the benefits of employing this technology over the regular WebSphere MQ distributed queuing model.

For more information about high availability with WebSphere MQ, refer to *WebSphere MQ Queue Manager Clusters*, SC34-6061 or the White Paper *Understanding high availability with WebSphere MQ*, which is available on the Web at:

http://www-128.ibm.com/developerworks/websphere/library/techarticles/0505_hiscock/0505_hiscock.html

This chapter uses a simple WebSphere MQ clustering scenario to demonstrate the use of this technology with WebSphere MQ transport for SOAP.

© Copyright IBM Corp. 2006. All rights reserved.

18.1 Benefits of WebSphere MQ clustering with Web Services

WebSphere MQ clustering's benefits over distributed queuing are two-fold:

- Reduced system administration

  Even small WebSphere MQ clusters ease the burden of system administration. Establishing a network of queue managers in a cluster involves far fewer definitions than an equivalent network using distributed queuing. Consequently, changing and scaling a clustered network of queue managers is quicker and easier. Day-to-day administration is also simplified, with fewer definitions reducing the scope for error.

- Increased availability and workload balancing

  WebSphere MQ clustering allows high availability of Web Services through clustered queues. The ability to define instances of the same queue on more than one queue manager (clustered queue) ensures high availability of this queue, and hence, high availability of the Web Services using this queue. Additionally, workload balancing, which is configurable, is achieved.

When using WebSphere MQ clustering, the integrity of persistent messages is maintained, and messages are never duplicated or lost.

Even if it is not required to make queues highly available, taking the time to understand and set up clustering from the outset is often worth the effort. It is far easier to incorporate clustering from the beginning than to migrate to it from an extensive distributed queuing configuration later.

18.2 An example scenario

For purposes of demonstration, a typical scenario is set up, where two data centers that are geographically remote from one another are used to distribute workload and provide high availability. WebSphere MQ clients, including Web Service clients, use clustered queues to perform tasks and obtain services from the remote data centers, utilizing what is commonly called a messaging bus.

Figure 18-1 depicts this scenario with Web Services.

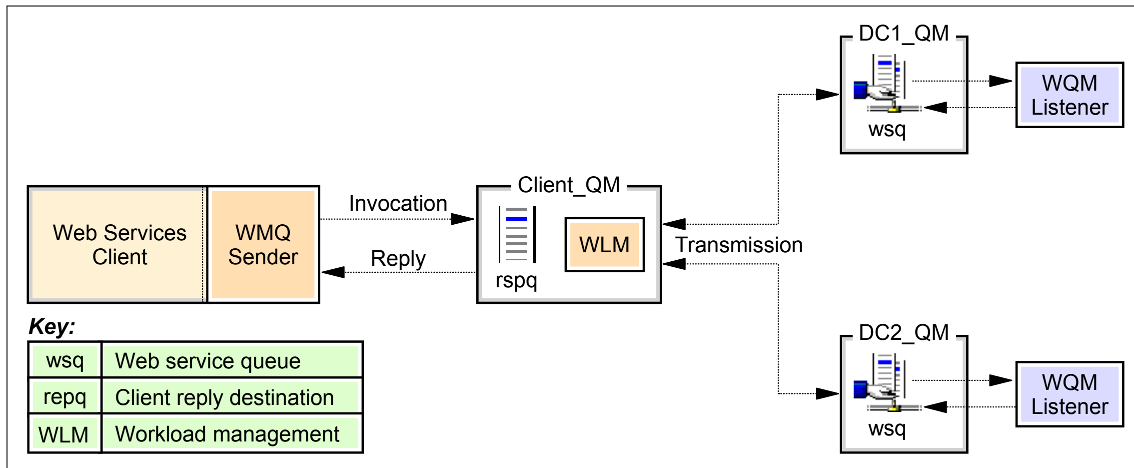


Figure 18-1 WebSphere MQ clustering with Web Services

Figure 18-1 shows two data centers, each with a clustered queue manager, DC1_QM and DC2_QM. In each of these clustered queue managers, instances of the same queue (clustered queue), wsq, is defined. A WebSphere MQ listener monitors each instance of wsq. The Web Service provided by these listeners is identical, and any backend resources that are used, are conjoined in some way, for example, by using a common database.

In the client, the queue manager Client_QM is also a part of the same WebSphere MQ cluster as DC1_QM and DC2_QM. Client_QM defines a single local queue for the response or a dynamic queue. See Chapter 5, “SOAP/WebSphere MQ implementation” on page 49 for more information about using dynamic queues with WebSphere MQ Web Services.

18.2.1 The client invocation and the WebSphere MQ sender

The Web Services client invokes the Web Service, either synchronously or asynchronously, specifying the wsq clustered queue as the destination in the Universal Resource Locator (URL), but *not* the queue manager. This means that the queue manager is not fixed and can be selected by the Client_QM queue manager. In most cases, for example, the destination and the queue manager are specified together at the beginning of the URL:

```
jms:/queue?destination=wsq@DC1&connectionFactory=...
```

In doing so, there is no opportunity for Client_QM to decide on which of the two cluster queue managers hosting wsq to send the request to.

Note: Specifying the queue manager has the effect of populating the ObjectQMgrName field of the MQOD used on the MQOPEN call by the WebSphere MQ sender. This dictates the queue manager to which the message is sent, preventing the Client_QM from performing cluster name resolution on the queue name and selecting a suitable queue manager from the cluster.

To allow cluster name resolution, drop the “@” separator and the queue manager name from the URL, for example:

```
jms:/queue?destination=wsq&connectionFactory=...
```

Client_QM is able to choose the queue manager to which the request is sent through the workload choose algorithm, represented by WLM in Figure 18-1. First, a list of all the suitable and available cluster queues matching the queue name specified is created. Workload balancing is then performed to select a queue manager to which the request is to be sent. In the default configuration, this is essentially round robin. Therefore, in the example scenario, the Web Service is invoked in each queue manager alternately. If one of the data centers becomes available, the workload-selected algorithm routes all the new requests to the available data center.

Tip: The behavior of the default workload balancing algorithm can be altered by tuning the various queue manager, queue, and channel attributes. One of the new cluster workload balancing feature introduced in WebSphere MQ V6 is cluster workload priority on channels and queues. Cluster workload priority can be used to nominate a primary data center and a backup data center, in which, instead of routing in a round robin manner, all the requests are routed to the primary data center, unless it becomes unavailable, in which case, all the requests are routed to the backup data center. Refer to *WebSphere MQ Queue Manager Clusters*, SC34-6061 for more details. The following Web site provides details pertaining to WebSphere MQ:

http://publib.boulder.ibm.com/infocenter/wmqv6/v6r0/index.jsp

A Web Service client has the ability to choose the destination by specifying the queue manager name in the URL as normal. If this destination queue manager is not available, the messages are not delivered, but held on the cluster transmission queue until the destination queue manager is available again. Messages are not routed to another queue manager because of the risk of duplication.

18.2.2 The Web Service and the WebSphere MQ listener

When the Client_QM picks a queue manager to send the request to, the message is transported through the transmission queue in Client_QM to either DC1_QM or DC2_QM, and then to the local wsq queue instance. Here, the WebSphere MQ listener picks up the message and invokes the service. After completion of the service, the WebSphere MQ listener sends the reply to the replyDestination queue specified in the URL. The replyDestination queue does not have to be a cluster queue. It can be a dynamic queue too.

Note: The replyDestination and connectQueueManager options in the URL are used together by the WebSphere MQ listener to locate the queue specified by replyDestination. These options correlate to the ReplyToQ and ReplyToQMgr fields respectively of the WebSphere MQ message descriptor in the request message.

18.3 Summary

Using WebSphere MQ clustering for easy administration also builds flexibility into the WebSphere MQ infrastructure by providing a level of direction in the client. If the client's connectQueueManager is in the same cluster as the Web Service queue manager, the invocation finds its way to the Web Service and the response back to the client.

WebSphere MQ clustering also provides the ability to scale and alter the WebSphere MQ configuration. If clustering is not already started, moving to it at a later stage is difficult and time consuming. It is well worth the investment to use clustering from the outset.

A

WebSphere MQ using .NET classes

This appendix discusses WebSphere MQ programming using the WebSphere MQ .NET classes provided for writing .NET applications that interface with WebSphere MQ. The WebSphere MQ .NET classes, formerly the MA7P SupportPac, is included within WebSphere MQ V6, and allows programs written in the .NET programming framework to access WebSphere MQ objects by issuing calls to and queries from them.

This appendix provides an overview of the WebSphere MQ .NET classes. It also provides information about the following tasks:

- Putting messages on a queue
- Getting messages off a queue
- Putting request messages and getting back replies

© Copyright IBM Corp. 2006. All rights reserved.

WebSphere MQ .NET classes

Applications interact with WebSphere MQ objects such as queues by using the WebSphere MQ .NET classes. Before applications are written to interact with these objects, a connection to the WebSphere MQ queue manager must be established in one of the following ways:

- Client connection

  This type of connection uses a TCP/IP connection to the WebSphere MQ server in order to enable communication directly with the queue manager. This type of connection uses client channels on the queue manager to enable communication. The host name, which is the name of the WebSphere MQ server machine that hosts the queue manager, the channel name, which is the name of the channel for client connection, and the port number on which the WebSphere MQ server listens, must be known.

- Server binding mode or server connection

  In binding mode, which is also known as server connection, the communication to the queue manager utilizes interprocess communications. Ensure that the binding mode is available only for programs running on the same machine as the queue manager. A program using binding mode does *not* run from a remote machine. In other words, the application is tied to the same machine that the queue manager is on.

  Binding mode is a fast and efficient way to interact with WebSphere MQ.

Overview

This section provides a summary of the classes contained within the WebSphere MQ .NET classes, and their methods and properties.

The WebSphere MQ .NET classes consist of the following classes:

- MQChannelDefinition
- MQEnvironment
- MQException
- MQGetMessageOptions
- MQManagedObject
- MQMessage
- MQPutMessageOptions
- MQQueue
- MQQueueManager

MQChannelDefinition

This class specifies information pertaining to the connection to the queue manager. This class does not apply when connecting directly to WebSphere MQ in server binding mode.

MQEnvironment

This class contains static member variables that are set when a queue manager is constructed in order to control the way the queue manager object is constructed. Because the values set in this class take effect when the MQQueueManager constructor is called, the values in the MQEnvironment class must be set before an MQQueueManager instance is constructed. Following are the properties that are defined:

- The *name of the channel* to connect to on the target queue manager. This applies only when connecting in client mode.
- The *host name of the machine* on which the WebSphere MQ server resides. This applies only when connecting in client mode.
- The *port* which the WebSphere MQ server is listening to for incoming connection requests. The default value is 1414. This applies only when connecting in client mode.
- The *SSLCipherSpec*, which, if set, enables Secure Socket Layer (SSL) for the connection. This applies only when connecting in client mode.
- If the SSLCipherSpec is specified, the *SSLKeyRepository* provides the fully qualified file name of the key repository. If the SSLCipherSpec is not specified, this property is ignored.
- If SSLCipherSpec is set, the *SSLPeerName* ensures that the correct queue manager is used. This variable is ignored if SSLCipherSpec is null and does not apply when connecting in server binding mode.

MQException

An MQException is thrown whenever a WebSphere MQ error occurs. This class contains the definitions of the WebSphere MQ completion code (errors beginning with MQCC_) and the error reason code (errors beginning with MQRC_).

MQGetMessageOptions

Use the MQQueue.Get() method to retrieve messages from queues. This class contains the options that control the behavior of the MQQueue.get() method. Following are the properties that are defined:

- *GroupStatus* is an output field that indicates whether the retrieved message is in a group, and if it is, whether it is the last in the group.

- *MatchOptions* is a selection criteria that determines which messages are retrieved, for example, it may be set to MQC.MQMO_MATCH_CORREL_ID so that messages with a certain Correlation ID are retrieved.
- *Options* control the action of MQQueue.Get() so that message data can be truncated, browsed, locked and unlocked, or converted to a certain style. It also specifies the actions to be taken if the queue manager is quiescing or if there are no suitable messages on the queue. Specifying the segmenting and grouping options of messages is also possible. You can, for example, specify that messages must be retrieved in groups only when all the messages in the group are available. Alternatively, you can specify that messages must be retrieved in segments only when all the segments in the group are available.
- *ResolvedQueueName* is an output field that the queue manager sets to the local name of the queue from which the message is retrieved. This is different from the name that is used to open the queue if an alias queue or a model queue is opened.
- *Segmentation* is an output field that indicates whether or not segmentation is allowed for the retrieved message.
- *SegmentStatus* is an output field that indicates whether the retrieved message is a segment of a logical message. If the message is a segment, the flag indicates whether or not it is the last segment.

MQManagedObject

The MQManagedObject provides the ability to inquire and set attributes of queue managers and queues. The following methods are included in the MQManagedObject class:

- *Close* closes the WebSphere MQobject (queue, process, or queue manager). No further operations against this object are permitted after this method is called.
- *Inquire* returns an array of integers and a set of character strings containing the attributes of a WebSphere MQ object (queue, process, or queue manager).
- *Set* sets the attributes defined in a selector's vector. Refer to *WebSphere MQ Application Programming Reference*, SC34-6062 for details about the permissible selectors and their corresponding integer values.

Following are the properties that are defined:

- *AlternateUserId*, which is the alternate userID, if any, specified when this resource is opened.
- *CloseOptions* controls the way the WebSphere MQ object is closed.

- *ConnectionReference* returns the queue manager to which the WebSphere MQ object belongs.
- *IsOpen* returns a boolean value to indicate whether the WebSphere MQ object is currently open.
- *Name* is the name of the WebSphere MQ object.
- *OpenOptions* specifies when the WebSphere MQ object was opened.

MQMessage

This class represents the message descriptor and the data for a WebSphere MQ message. The following methods are included in the MQMessage class:

- *ClearMessage* discards any data in the message buffer, and sets the data offset back to 0 (zero).
- Several *Read and Write methods* are included to read and write bytes, booleans, characters, strings, decimals, integers, double values, and so on.
- *ResizeBuffer* gives a hint to the MQMessage object about the size of the buffer that may be required for subsequent get operations. If the message currently contains message data, and the new size is less than the current size, the message data is truncated.
- *Seek* moves the cursor to the absolute position in the message buffer given by position, pos. Subsequent reads and writes act at this position in the buffer.
- *SkipBytes* moves forward n bytes (number of bytes) in the message buffer. This method blocks until all the bytes are skipped or the end of the message buffer is detected or an exception is thrown.

MQPutMessageOptions

Place messages on queues using the MQQueue.Put() method. This class contains the options that control the behavior of the MQQueue.Put() method. The following properties are defined:

- *ContextReference* indicates the source of the information.
- *InvalidDestCount* is an output field set by the queue manager to the number of messages that cannot be sent to queues in a distribution list.
- *KnownDestCount* is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to local queues.
- *Options* control the action of MQQueue.put. Any or none of these values can be specified. Options can be set in such a way that a put operation fails if the queue manager is quiese, the queue manager puts logical messages and segments in message groups into their logical order, generates a new correlation ID or message ID for each sent message, and so on.

- *RecordFields* set flags indicating which fields must be customized in each queue when putting a message to a distribution list.
- *ResolvedQueueManagerName* is an output field that is set by the queue manager to the name of the queue manager that owns the queue specified by the remote queue name. This may be different from the name of the queue manager from which the queue is accessed if the queue is a remote queue.
- *ResolvedQueueName i*s an output field that is set by the queue manager to the name of the queue on which the message is placed. This may be different from the name used to open the queue if the opened queue is an alias or model queue.
- *UnknownDestCount* is an output field set by the queue manager to the number of messages that the current call has sent successfully to queues that resolve to remote queues.

MQQueue

This class provides inquiry, set, put, and get operations for WebSphere MQ queues. The inquire and set capabilities are inherited from MQManagedObject.

MQQueueManager

This class represents the queue manager for WebSphere MQ.

Environment setup

To create .NET applications that interface with WebSphere MQ V6, the following software is required:

- WebSphere MQ V6
- Microsoft .NET Framework Redistributable V1.1
- Microsoft .NET Software Development Kit V1.1
- Text editor (Notepad)
- Visual Studio .NET2003 (optional)
- amqmdnet.dll dynamic link library (DLL) provided with WebSphere MQ V6 located in WebSphere MQ Installation directory\bin

Interacting with queues

In order to perform any operation on the queue, create a queue handle or queue object by opening the queue. There are two ways to open the queue. Open it by using the accessQueue method of the MQQueueManager object or through the constructor call of the MQQueue class.

The two calls are of the following forms:

- MQQueue queue=qmgr.accessQueue(“qName’, openOption, “qMgrName”, “dynamicQname”, “alternateUserId”);
- MQQueue queue=new MQQueue(qmgr, “qName”, openOption, “qMgrName”, “dynamicQname”, “alternateUserId”);

  The second approach of using the MQQueue class constructor is much the same, with an added queue manager parameter.

WebSphere MQ validates the openOption against the user authorization during the process of opening the queue.

The object of the MQQueue class represents a queue. It has methods to facilitate messaging, namely, put, get, set, inquire, and properties that correspond to the attributes of a queue.

Working with messages

The MQMessage object represents a message that is put or got from a queue. It encapsulates the application data and the message descriptor (MQMD). It has properties corresponding to the MQMD fields and methods to write or read different application data of different data types to and from the message. Within the application, the MQMessage represents a buffer. An application does not have to declare the buffer size because it resizes itself to accommodate the data being written to it. However, if the message size is more than the MaximumMessageLength property of the queue, further put messages are disabled.

To create a message, create a new instance of the class MQMessage. Application data is written to the message using the writeXXX methods for the specific application data type. The format of data types, such as numbers and strings, can be controlled by the MQMD properties, characterless and encoding. The MQMD fields can be set before the message is put on the queue and can be read upon getting the message from the queue. When a message is instantiated,

the MQMD fields are set to their default values. Applications control the way messages are put on the queue or are got from the queue, by setting appropriate options with the put or get operations. Similarly, the way messages are retrieved from the queue is controlled by setting the appropriate get message options.

Putting a message on a WebSphere MQ queue

The way messages are put on the queue is determined by the value of the Options field of the instance of the MQPutMessageOptions class. The instance of the MQPutMessageOptions class has the value for the Options property set to the default value. This may be sufficient in most of the simple messaging scenarios.

MQPutMessageOptions

Set the value of the options by using the MQPutMessageOptions (MQPMO) of WebSphere MQ constants MQC, as shown in Example A-1. The example sets the value of the Options field to instruct the queue manager to generate a new message ID for the message and set the MsgId field of the MQMD.

Example: A-1 WebSphere MQ put message option

```
MQPutMessageOptions Pam = new MQPutMessageOptions();
pmo.options = pmo.options + MQC.MQPMO_NEW_MSG_ID
```

Getting a message off a WebSphere MQ queue

The way messages are retrieved from the queue is determined by the value of the Options field of the instance of the MQGetMessageOptions class. The new instance of the MQGetMessageOptions class has the value of the options property set to default.

MQGetMessageOptions

The way messages are retrieved from the queue is determined by the value of the Options field of the instance of the MQGetMessageOptions class. Set the value of Options by using the MQGetMessageOptions (MQGMO) of WebSphere MQ Constants' Modular Quality of Service Command-line Interface (MQC), as shown in Example A-2. This option specifies that the get message call must return immediately if there are no messages in the queue.

Example: A-2 WebSphere MQ get message option

```
MQGetMessageOptions gmo = new MQGetMessageOption();
gmo.options = gmo.options + MQC.MQGMO_NO_WAIT;
```

Sending messages

Messages are sent using the put(MQMessage message) or put(MQMessage message, the MQPutMessageOptions pmo) methods of the MQQueue class. The put method call places the message on the WebSphere MQ queue, while the put message options control the way the messages are placed on the queue.

Receiving messages

Messages are retrieved from the WebSphere MQ queue using the get(MQMessage message) or get(MQMessage, MQGetMessageOptions gmo), get(MQMessage, MQGeMessageOptions gmo, int maxMessageSize) methods of the MQQueue class.

Application development

With the help of simple examples, the following sections demonstrate how messages can be put and got from a queue or queues.

Simple WebSphere MQ put operation

To write applications in .NET to put messages on a queue, perform the following tasks:

1. In Visual Studio .NET, create a new blank solution. Within the blank solution:
   a. Create a new empty project.
   b. Add a new code file to the empty project.

   If Visual Studio .NET is not being used for the development, the text editor file is sufficient.
2. In Visual Studio .NET, import the amqmdnet.dll DLL file. This file contains the libraries required for .NET applications to program WebSphere MQ objects.
   a. To do this, right-click the project and select **Add Reference**.
   b. Browse to the location of amqmdnet.dll, which is typically the WebSphere MQ Home\bin directory, and double-click **amqmdnet.dll**.

   If Visual Studio .NET is not being used, link amqmdnet.dll to the solution when it is being compiled.
3. After the library is included, write the code. Applications that write to a WebSphere MQ queue must first connect to the WebSphere MQ queue manager. Before connecting to the queue manager, it must be known whether a server binding connection or a client connection is required. For demonstration purposes, a client connection is used in this appendix.

To define a client connection, set the MQEnvironment properties, such as the host name and channel with the code shown in Example A-3.

Example: A-3 Code to define WebSphere MQ environment

```
string hostName = "78FDCTC." ;
string channel = "DOTNET.SVRCONN" ;

//Set up the MQEnvironment properties for Client Connections
MQEnvironment.Hostname = hostName ;
MQEnvironment.Channel = channel ;
```

> **Note:** Before establishing the client connection specified earlier, set up a server connection channel. In WebSphere MQ explorer, right-click the channel under the queue manager's Advanced folder and select **New, server-connection channel**.

4. Establish a connection to the queue manager with the code snippet in Example A-4.

Example: A-4 Code to connect to queue manager

```
string qManager = "WMQDOTNET.DEMO.QM" ;

//Connection To the Queue Manager
MQQueueManager qMgr = new MQQueueManager(qManager) ;
```

5. Open a queue and put messages in it. This is where the open options are useful. See "MQPutMessageOptions" on page 385 for open options. In this example, the queue is opened for putting messages. The put fails if the queue manager is quiesce. Use the code snippet in Example A-5 to open the queue. The null parameters to the AccessQueue constructor are the queue manager name, the dynamic queue name, and the alternate userID. Because the connection to the queue manager is already established, it is not necessary to include this.

Example: A-5 Code to open the queue

```
string qName = "DOTNET.QUEUE" ;
/* Set up the open options to open the queue for out put and
additionally we have set the option to fail if the queue manager is
quiescing.
*/

int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
//Open the queue
```

```
MQQueue queue = qMgr.AccessQueue(qName, openOptions, null, null, null);

// Set the put message options , we will use the default setting.
MQPutMessageOptions pmo = new MQPutMessageOptions();
```

6. The queue is ready to accept messages. Construct the message and send it, using the code snippet in Example A-6.

Example: A-6 Code to construct a message to put on the queue

```
/* Next build a message The MQMessage class encapsulates the data
buffer
that contains the actual message data, together with all the MQMD
parameters that describe the message.
To Build a new message, create a new instance of MQMessage class and
use
writxxx (we will be using writeString method). The put() method of
MQQueue also takes an instance of the MQPutMessageOptions class as a \
parameter.
*/

MQMessage outMsg = new MQMessage(); //Create The message buffer
outMsg.Format = MQC.MQFMT_STRING ; // Set the MQMD format field.

//Prepare message with user data
string msgString = "Test Message from MQPUT operation program.";
outMsg.WriteString(msgString);
```

7. Call the MQPut method using the code shown in Example A-7.

Example: A-7 Code to put the message on the queue

```
// Now we put The message on the Queue
queue.Put(outMsg, pmo);
//Commit the transaction.
qMgr.Commit();
Console.WriteLine("  The message '" + msgString + "' has been
                 successfully put");
```

8. At this stage, write an application to put a message in the queue. The rest of the code pertains to housekeeping. Close the queue and the queue manager objects using the code snippet in Example A-8.

Example: A-8 Code to close and disconnect from queue and queue manager

```
// Close the Queue and Queue manager objects.
queue.Close();
qMgr.Disconnect();
```

9. Compile the program in Visual Studio .NET by right-clicking the solution, and selecting **build**. If Visual Studio .NET in not being used, save the Notepad document in a desired location with a .cs file extension. Run the following command in the Windows command prompt:

   ```
   csc /lib:c:\progra~1\ibm\websph~1\bin /reference:amqmdnet.dll
   mqPUT.cs
   ```

Notes:

- This class must have a main method defined before it can compile.
- For commands within the Microsoft .NET Framework compiler to work from any folder, include the location of the Microsoft .NET Framework, typically root\WINNT\Microsoft.NET\Framework\version, in your computer's PATH environment variable. To do this:
  a. Right-click **My Computer** and select **Properties**.
  b. Click **Environment Variables** in the Advanced tab.
  c. Add the path to the Microsoft .NET Framework variable.

Simple WebSphere MQ get operation

To write applications in .NET to get messages from a queue, perform the following tasks:

1. In Visual Studio .NET, create a new blank solution. Within the blank solution:
   a. Create a new empty project.
   b. To the empty project, add a new code file.

   If Visual Studio .NET is not being used for the development, the text editor file is sufficient.

2. In Visual Studio .NET, import the amqmdnet.dll DLL file. This file contains the libraries required for .NET applications to program WebSphere MQ objects. To do this:
   a. Right-click the project and select **Add Reference**.

   b. Browse to the location of amqmdnet.dll, typically in the WebSphere MQ Home\bin directory, and double-click **amqmdnet.dll**.

   If Visual Studio .NET is not being used, link amqmdnet.dll to the solution when it is being compiled.

3. After the library is included, write the code. Applications that read from a WebSphere MQ queue must connect to the WebSphere MQ queue manager. Before connecting to the queue manager, it must be known whether a server binding connection or a client connection is required. For demonstration purposes, a client connection is used in this example.

   To define a client connection, set the MQEnvironment properties such as the host name and the channel, using the code shown in Example A-9.

Example: A-9 Code to define WebSphere MQ environment

```
string hostName = "78FDCTC." ;
string channel = "DOTNET.SVRCONN" ;

//Set up the MQEnvironment properties for Client Connections
MQEnvironment.Hostname = hostName ;
MQEnvironment.Channel = channel ;
```

> **Note:** Before the client connection specified earlier is established, a server-connection channel must be set up. In WebSphere MQ explorer, right-click the channel under the queue manager's Advanced folder and select **New, server-connection channel**.

4. Establish a connection to the queue manager by using the code snippet in Example A-10.

Example: A-10 Code to connect to queue manager

```
string qManager = "WMQDOTNET.DEMO.QM" ;

//Connection To the Queue Manager
MQQueueManager qMgr = new MQQueueManager(qManager) ;
```

5. Open a queue and get messages from it. This is where the open options are useful. See "MQGetMessageOptions" on page 388 for open options. In this example, the queue is opened to get messages. The get operation waits until there are messages in the queue. The get fails if the queue manager is quiesce. Use the code snippet in Example A-11 to open a queue and get

messages from it. The null parameters to the AccessQueue constructor are the queue manager name, the dynamic queue name, and the alternate userID. The connection to the queue manager is already established. Therefore, it is not necessary to include this.

Example: A-11 Code to open the queue and get messages from it

```
string qName = "DOTNET.QUEUE" ;
/* Set up the open options to open the queue for out put and
additionally we have set the option to fail if the queue manager is
quiescing.
*/
int openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;

//Open the queue
MQQueue queue = qMgr.AccessQueue(qName, openOptions, null, null, null);

MQGetMessageOptions gmo = new MQGetMessageOptions();
// Wait if no messages on the queue
gmo.Options = gmo.Options + MQC.MQGMO_WAIT ;
// Fail if QueueManager Quiescing
gmo.Options = gmo.Options + MQC.MQGMO_FAIL_IF_QUIESCING ;
gmo.WaitInterval = 3000 ; // Sets the time limit for the wait.
```

6. Create the message buffer that stores the message when the get method is called. Call the get method. Use the code snippet in Example A-12 to construct a message out of the data read in from the queue.

Example: A-12 Code to construct a message out of the data read in from the queue

```
MQMessage inMsg = new MQMessage(); //Create the message buffer

//Get the message from the queue on to the message buffer.
queue.Get(inMsg, gmo) ;

// Read the User data from the message.
string msgString = inMsg.ReadString(inMsg.MessageLength);
Console.WriteLine("  The Message from the Queue is : " + msgString );
//Commit the transaction.
qMgr.Commit();
```

7. Write an application to get a message off the queue. The rest of the code pertains to housekeeping. Close the queue and the queue manager objects using the code snippet shown in Example A-13.

Example: A-13 Code to disconnect and close the queue and the queue manager

```
// Close the Queue and Queue manager objects.
queue.Close();
qMgr.Disconnect();
```

8. Compile the program in Visual Studio .NET by right-clicking the solution and selecting **build**. If Visual Studio .NET in not being used, save the Notepad document with a .cs file extension in a desired location and run the following command in the Windows command prompt:

   ```
   csc /lib:c:\progra~1\ibm\websph~1\bin /reference:amqmdnet.dll
   mqGET.cs
   ```

Notes:

- This class must have a main method defined before it can compile.
- For commands within the Microsoft .NET Framework compiler to work from any folder, include the location of the Microsoft .NET Framework, typically root\WINNT\Microsoft.NET\Framework\version, in your computer's PATH environment variable. To do this:
  a. Right-click **My Computer** and select **Properties**.
  b. Click **Environment Variables** in the Advanced tab.
  c. Add the path to the Microsoft .NET Framework path variable.

Request and reply

Following is the process involved in a request-and-reply messaging pattern:

1. One application sends a message (request message) to another application (reply producer).
2. The reply producer then responds to the request message.
3. The reply-producing application gets the request message, processes the request, and sends a response back to the requesting application.
4. The request message header property of replyToQueue specifies the queue the reply message goes to. The replyToQueueManager message header property of the request message specifies the queue manager to which it belongs.
5. The requesting application sets these message header properties on the request message before putting the message in the queue.

6. The requesting application lets the queue manager generate a unique messageID.
7. The replying application copies the messageID of the request message to the correlationID of the reply message.
8. The requesting application uses the correlationID value of the reply message to map a response back to the original request.

   Figure A-1 shows the request-and-reply message identification.

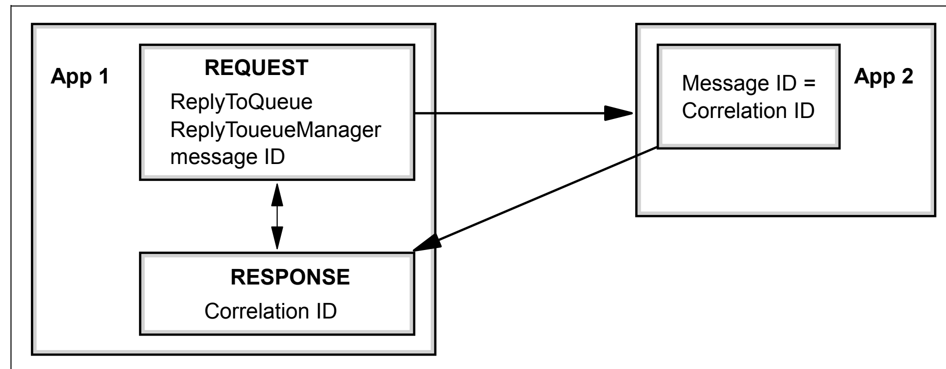


Figure A-1 Request-and-reply message identification

The request-reply pattern is illustrated with a pair of simple applications.

1. The first application, the requester, puts a simple message in the queue (request queue).
2. The requester sets the replyToQueue and replyToQueueManager message header properties on the request message before putting the request message on the request queue.
3. The requester then opens the reply queue and waits for messages with the correlationID matching the messageID value of the outgoing request message.
4. The responding application servicing the request message gets the request message, prepares the reply message, and sends it to the reply queue under the queue manager specified on the request message. It also copies the messageID from the request message on to the correlationID message header field of the response message.

The application, RequestReply.cs, is the application that sends the request message and expects a reply from the responding application.

To write applications in .NET to put request and get response messages from a queue, perform the following tasks:

1. In Visual Studio .NET, create a new blank solution. Within the blank solution:

   a. Create a new empty project.
   b. To the empty project, add a new code file.

   If Visual Studio .NET is not being used for the development, the text editor file is sufficient.

2. In Visual Studio .NET, import the amqmdnet.dll DLL file. This file contains the libraries required for .NET applications to program WebSphere MQ objects. To do this:

   a. Right-click the project and select **Add Reference**.
   b. Browse to the location of amqmdnet.dll, typically in the WebSphere MQ Home\bin directory, and double-click **amqmdnet.dll**.

   If Visual Studio is not being used, link amqmdnet.dll to the solution when it is being compiled.

3. After the library is included, write the code. Create a method called `Request`. A reply application is created later, and is invoked immediately after a request is placed on the queue.

   Applications that write to a WebSphere MQ queue must first connect to the WebSphere MQ queue manager. Before connecting to the queue manager, it must be known whether a server binding connection or a client connection is required. For demonstration purposes, a client connection is used in this example.

   To define a client connection, set the MQEnvironment properties such as the host name and the channel using the code shown in Example A-14.

Example: A-14 Code to set MQEnvironment properties

```
string hostName = "78FDCTC." ;
string channel = "DOTNET.SVRCONN" ;

//Set up the MQEnvironment properties for Client Connections
MQEnvironment.Hostname = hostName ;
MQEnvironment.Channel = channel ;
```

> **Note:** Before the client connection specified earlier is established, a server-connection channel must be set up. In WebSphere MQ Explorer, right-click the channel under the queue manager's Advanced folder and select **New, server-connection channel**.

4. Establish a connection to the queue manager using the code snippet in Example A-15.

Example: A-15 Code to connect to the queue manager

```
string qManager = "WMQDOTNET.DEMO.QM" ;

//Connection To the Queue Manager
MQQueueManager qMgr = new MQQueueManager(qManager) ;
```

5. Open a queue and set the open options. The put options specify that the put operation fails if the queue manager is quiesce. Use the code snippet in Example A-16 to set the open options.

Example: A-16 Code to set the open options

```
/* Set up the open options to open the queue for out put and
additionally we have set the option to fail if the queue manager is
quiescing.
*/
int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
//Open the queue
MQQueue queue = qMgr.AccessQueue(requestQueue,
openOptions,
null,
null,
null);
```

6. The queue is open. Prepare a message and put it in the queue. The message is set as a request message. The message also specifies the replyToQueue and replyToQueueManager to which the response is returned. Example A-17 shows the code to construct the request message.

Example: A-17 Code to construct the request message

```
MQMessage outMsg = new MQMessage(); //Create the message buffer
outMsg.Format = MQC.MQFMT_STRING ; // Set the MQMD format field.
outMsg.MessageType = MQC.MQMT_REQUEST ;
outMsg.ReplyToQueueName = replyToQueue;
outMsg.ReplyToQueueManagerName = replyToQueueManager ;
//Prepare message with user data
string msgString = "Test Request Message from Requester program ";
outMsg.WriteString(msgString);
```

7. Put the prepared message in the opened queue, commit the put, and close the queue, as shown in the code snippet in Example A-18.

Example: A-18 Code to put the prepared message in a queue

```
// Now we put The message on the Queue
queue.Put(outMsg, pmo);
//Commit the transaction.
qMgr.Commit();
Console.WriteLine(" REQUESTING APPLICATION: \n The request message has
“ +
                 “been successfully put\n");
// Close the Request Queue
queue.Close();
```

8. The request is sent and it sits on the specified request queue, waiting to be picked up, processed, and a response sent back to the response queue. A responding application is required.
9. Create a new method, `Reply`, which serves as the responding application. This responding application defines its environment, makes a connection to the queue manager, and opens the request queue the same way in which the requesting application does.
10. The getMessageOptions are set to wait for messages in the request queue and pull out messages from the request queue with the code shown in Example A-19.

Example: A-19 Code to get the message off the request queue

```
// Set the get message options.
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.Options = gmo.Options + MQC.MQGMO_WAIT ; //Wait if no messages on
queue
gmo.Options = gmo.Options + MQC.MQGMO_FAIL_IF_QUIESCING ;
gmo.WaitInterval = 3000 ; // Sets the time limit for the wait.
/* Next we Build a message The MQMessage class encapsulates the data
buffer
that contains the actual message data, together with all the MQMD
parameters that describe the message.
To Build a new message, create a new instance of MQMessage class and
use
writxxx (we will be using writeString method). The put() method of
MQQueue also takes an instance of the MQPutMessageOptions class as a
parameter.
*/
```

```
MQMessage inMsg = new MQMessage(); //Create the message buffer
// Get the message from the queue on to the message buffer.
queue.Get(inMsg, gmo) ;
// Read the User data from the message.
string msgString = inMsg.ReadString(inMsg.MessageLength);
```

11. Analyze the message to check if it is a request type message, and then to find out where its replies goes to, that is, the replyToQueueName, using the code snippet in Example A-20.

Example: A-20 Code to check if the message is a request message

```
Console.WriteLine(" RESPONDING APPLICATION: \n The message received from" +
                    "the Queue is : \n" + " " + msgString + "\n");
//Check if message if of type request message and reply to the request.
if (inMsg.MessageType == MQC.MQMT_REQUEST )
{
Console.WriteLine(" Preparing To Reply To the Request " );
string replyQueueName = inMsg.ReplyToQueueName ;
}
```

12. The queue to reply to is known. Open it, prepare a response message, and put the message into the response queue. The response message's correlationID is assigned the value of the request message's message ID. This is what allows the request message to know which response is coming back for which request. Example A-21 shows the code to prepare a response message and put it in the response queue.

Example: A-21 Code to prepare a response message and put it in the response queue

```
openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
MQQueue respQueue = qMgr.AccessQueue(replyQueueName,
openOptions,
inMsg.ReplyToQueueManagerName,
null,
null);
MQMessage respMessage = new MQMessage() ;
respMessage.CorrelationId = inMsg.MessageId;
MQPutMessageOptions pmo = new MQPutMessageOptions();
respMessage.Format = MQC.MQFMT_STRING ;
respMessage.MessageFlags = MQC.MQMT_REPLY ;
```

```
string response = "Reply from the Responder Program " ;
respMessage.WriteString(response);
respQueue.Put(respMessage, pmo);
Console.WriteLine(" The response '" + response +
              "' has been successfully sent \n\n");
```

13. Close the queue and disconnect the queue manager using the code snippet shown in Example A-22.

Example: A-22 Code to close the queue and disconnect from the queue manager

```
qMgr.Commit();
respQueue.Close();

queue.Close();
qMgr.Disconnect();
```

14. Go back to the requesting application and call this newly created Reply() method, which sends a response to the request message, using the code snippet shown in Example A-23.

Example: A-23 Code to call the responder application

```
//Now Reply
Console.WriteLine(" CALL RESPINDING APPLICATION\n\n");
Reply();
```

15. Open the response queue the same way you opened the other queues. The message with the correlationID that matches the messageID of the request message is retrieved from the response queue. Set the getMessageOption's matchOption property to match the correlationID using the code shown in Example A-24.

Example: A-24 Code to get response message for a request from the response queue

```
//Get Back Response
// Set openOption for response queue
openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;
MQQueue respQueue = qMgr.AccessQueue(replyToQueue, openOptions, null,
null,
                    null);
MQMessage respMessage = new MQMessage();
MQGetMessageOptions gmo = new MQGetMessageOptions();
gmo.Options = gmo.Options + MQC.MQGMO_SYNCPOINT;
gmo.Options = gmo.Options + MQC.MQGMO_WAIT ;
gmo.MatchOptions = MQC.MQMO_MATCH_CORREL_ID;
```

```
gmo.WaitInterval = 10000 ;
respMessage.CorrelationId = outMsg.MessageId ;

// Get the response message.
respQueue.Get(respMessage, gmo);
string response = respMessage.ReadString(respMessage.MessageLength);
Console.WriteLine(" REQUESTING APPLICATION: \n The response message is
: "
                        + response);
qMgr.Commit();
```

16. Close the queue and the queue manager. Example A-25 shows the code to close the response queue and to disconnect from the queue manager.

Example: A-25 Code to close the response queue and disconnect from queue manager

```
respQueue.Close();
qMgr.Disconnect();
```

17. Compile the program in Visual Studio .NET by right-clicking the solution and selecting **build**. If Visual Studio .NET in not being used, save the Notepad document with a .cs file extension in a desired location and run the following command in the Windows command prompt:

```
csc /lib:c:\progra~1\ibm\websph~1\bin /reference:amqmdnet.dll
mqRequestReply.cs
```

Notes:

- This class must have a main method defined before it can compile.
- For commands within the Microsoft .NET Framework compiler to work from any folder, include the location of the Microsoft .NET Framework, typically root\WINNT\Microsoft.NET\Framework\version, in your computer's PATH environment variable. To do this:
  a. Right-click **My Computer** and select **Properties**.
  b. Click **Environment Variables** in the Advanced tab.
  c. Add the path to the Microsoft .NET Framework path variable.

Running the applications

Download the application code from Appendix D, “Additional material” on page 431. It contains the mqPUT.cs, mqGET.cs, mqRequestReply.cs, and runDemos.cs. The runDemos.cs invokes all the three applications. Figure A-2 shows the result of running all the demos.

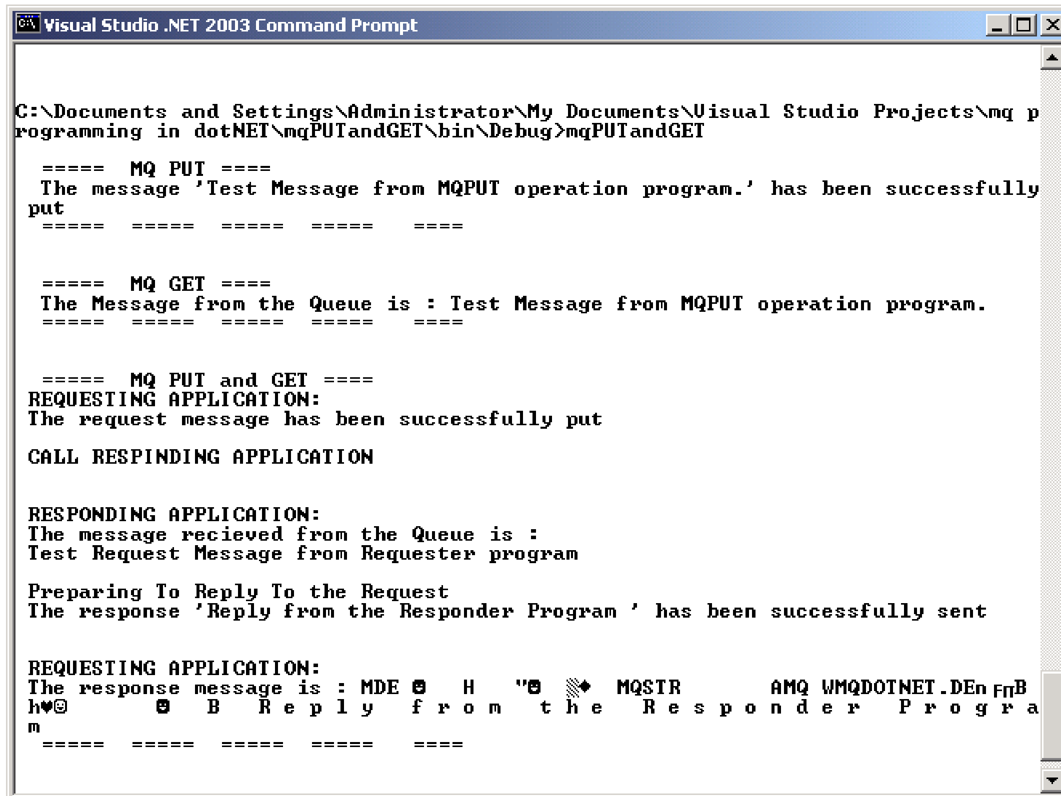


Figure A-2   Result of running all the three demos

The .NET monitor

The .NET Monitor, which is new in WebSphere MQ V6, is a utility that provides a trigger monitor for WebSphere MQ .NET applications. The WebSphere MQ .NET application must implement a new interface, IMQObjectTrigger, which is introduced in this version.

The monitor, runmqdnm, performs the following functions:

- Runs as a standalone or can be triggered
- Supports either WebSphere MQ or .Net transactions
- Supports backout threshold processing

A command, **`endmqdnm`**, to end the monitor is also available.

For more information about the IMQObjectTrigger interface and the use of the commands, refer to *WebSphere MQ V6.0 Using .NET*, GC34-6605-00.

B

WebSphere MQ using Java classes

This appendix covers programming WebSphere MQ using the WebSphere MQ classes for Java. It provides the basic information required by programmers who want to write Java applications for WebSphere MQ. It also serves as a reference when using WebSphere MQ transport for SOAP. This is by no means a complete guide. For more information about WebSphere MQ Java classes, refer to *WebSphere MQ Using Java*, SC34-6591.

© Copyright IBM Corp. 2006. All rights reserved.

Overview

WebSphere MQ Java classes expose the Message Queue Interface (MQI) that WebSphere MQ provides to programmers. The MQI provides WebSphere MQ client access to all the facilities of the WebSphere MQ messaging platform, and gives full and detailed control of the messages and the way they flow.

When a WebSphere MQ client connects to the queue manager, two methods are available:

- Client mode

  Client connection uses a TCP/IP connection to the WebSphere MQ queue manager. Programs using client connections can run on a WebSphere MQ client machine and on a WebSphere MQ server machine. Client connections use client channels to communicate with the queue manager.

  When using client connection, a few additional environment properties to establish connection must be specified with the queue manager. These include the host name, which is the name of the WebSphere MQ server machine that hosts the queue manager, and the channel name, which is the name of the channel for client connection. Additionally, the port number on which the WebSphere MQ server listens can be specified. If the port number is not specified, the default port number 1414 is used.

- Binding mode

  In binding mode, also known as server connection, the communication to the queue manager utilizes interprocess communications. Ensure that the binding mode is available only to programs running on the same machine as the queue manager. A program using binding mode does not run from a remote machine. In other words, the application is tied to the same machine the queue manager is on.

  Binding mode is a fast and efficient way to interact with WebSphere MQ.

The rest of this appendix describes commonly used WebSphere MQ Java classes. This appendix only considers the point-to-point programming approach and not the publish-subscribe approach.

Using the WebSphere MQ Java classes

This section discusses how to use the WebSphere MQ Java classes. It gives a broad overview of the classes themselves, and delves into the issues associated with using them.

What are WebSphere MQ Java classes?

The list of WebSphere MQ Java classes provided here is not exhaustive. However, it gives an idea about what is required in order to write simple Java applications that interact with the queue manager.

- MQChannelDefinition

  This class is used to pass on information regarding connection to the queue manager to the send, receive, and security exits. This class does not apply when connecting directly to the queue manager in binding mode.

- MQEnvironment

  This class contains static member variables that control the WebSphere MQ client environment. This environment controls various aspects of the WebSphere MQ client, but most significantly, it directs the means of connection to the queue manager. When an MQQueueManager object and its corresponding connection to the queue manager is constructed, it uses the values in the MQEnvironment. Therefore, the MQEnvironment variables must be set before constructing the MQQueueManager.

  Following is a list of some of the commonly used static member variables:

  – channel

    The name of the channel used to connect to the target queue manager. This is used only in client mode.

  – connOptions

    The queue manager connection options. This applies to binding mode connection only. There are various aspects that trade off performance for robustness in terms of the safety with which the connection to the queue manager is made.

  – hostname

    The TCP/IP host name of the machine where the queue manager resides. This is used only in client mode.

  – port

    The port used when connecting to a remote queue manager in client mode. This port must match the port number of the WebSphere MQ TCP listener running on the queue manager machine.

  – sslCipherSuite

    This is valid only in client mode. It specifies the ciphersuite to be used. Various other Secure Sockets Layer (SSL) options also exist.

- MQException

  This class contains the definitions of the WebSphere MQ completion code and error code constants. Constants beginning with MQCC_ are WebSphere MQ completion codes and constants beginning with MQRC_ are WebSphere MQ reason codes. An MQException is thrown whenever a WebSphere MQ error occurs.

- MQGetMessageOptions

  This class contains the options that control the behavior of the MQQueue.get() method. These options are fields in this class. The following are the commonly used options:

  – matchOptions

    This specifies the selection criteria that control the messages that are retrieved, for example, messages that have a certain correlation ID.

  – options

    This specifies the behavior of MQQueue.get(), for example, not waiting for a message if none are on the queue at the time of MQGET.

- MQManagedObject

  This class is a superclass for the MQQueueManager, MQQueue, and MQProcess classes. It provides the ability to inquire and set the attributes of these resources.

- MQMessage

  This class represents the message descriptor and the data for a WebSphere MQ message.

- MQPutMessageOptions

  This class contains the options that control the behavior of the MQQueue.put() method. As with MQGetMessageOptions, these options are fields in the class. The following option is commonly used:

  – options

    This specifies the behavior of MQQueue.put(), for example, performing the MQPUT under sync point.

- MQQueue

  This class provides inquiry, set, put, and get operations for WebSphere MQ queues. The inquire and set capabilities are inherited from MQManagedObject, which is not discussed here.

- MQQueueManager

  This class represents the WebSphere MQ queue manager.

Environment setup

To write WebSphere MQ V6 Java applications, the following software is required:

- WebSphere MQ Java client
- IBM Java SDK 1.4.2
- Preferred editor for Java or Integrated Development Environment (IDE)

Interacting with queues

In order to perform operations on the queue, create a queue handle or queue object by opening the queue. There are two ways to open the queue, by using the accessQueue method of the MQQueueManager object or through the constructor call of the MQQueue class.

The two calls are of the following form:

- MQQueue queue=qmgr.accessQueue("qName', openOption, "qMgrName" ,"dynamicQname", "alternateUserId");
- MQQueue queue=new MQQueue(qmgr, "qName', openOption, "qMgrName" , "dynamicQname", "alternateUserId");

  The second approach of using the constructor of the MQQueue class is much the same, with an added queue manager parameter.

WebSphere MQ validates the openOption against user authorization during the process of opening the queue.

The object of the MQQueue class represents a queue. It has methods to facilitate messaging, namely, put, get, set, inquire, and properties that correspond to the attributes of a queue.

Working with messages

The MQMessage object represents a message that is put or got from a queue. It encapsulates the application data and the message descriptor (MQMD). It has properties corresponding to the MQMD fields and methods to write or read different application data of different data types to and from the message. Within the application, the MQMessage represents a buffer. An application does not have to declare the buffer size because it resizes itself to accommodate the data being written to it. However, if the message size is more than the MaximumMessageLength property of the queue, further put message operations are disabled.

To create a message, create a new instance of the class MQMessage. Write application data to the message using the writeXXX methods for the specific application data type. To control the format of data types such as numbers and strings, use the MQMD properties, characterSet, and encoding. The MQMD fields can be set before the message is put on the queue, and can be read on getting the message from the queue. When a message is instantiated, the MQMD fields are set to their default values. Applications control the way messages are put on the queue or are got from the queue by setting appropriate options with the put or get operation. The way the message is put on the queue is controlled by setting the appropriate put message option values. Similarly, the way messages are retrieved from the queue is controlled by setting appropriate get message options.

Putting a message on a WebSphere MQ queue

Messages are put or sent using the put(MQMessage message) or put(MQMessage message, MQPutMessageOptions pmo) methods of the MQQueue class. The put method call places the message on the WebSphere MQ queue.

MQPutMessageOptions

The way messages are put on the queue is determined by the value of the options field of the instance of the MQPutMessageOptions class. Set the value of the options by using the MQPutMessageOptions (MQPMO) constants in the MQC interface.

The instance of the MQPutMessageOptions class has the value for the options property set to the default value. This may be sufficient in most of the simple messaging scenarios. You can set any specific options using the MQPMO constants in the MQC interface. Example B-1 shows the WebSphere MQ put message option. This example sets the value of the options field to instruct the queue manager to generate a new message ID for the message, and set the MsgId field of the MQMD.

Example: B-1 WebSphere MQ put message option

```
MQPutMessageOptions pmo = new MQPutMessageOption();
pmo.options = pmo.options + MQC.MQPMO_NEW_MSG_ID
```

Getting a message off a WebSphere MQ queue

To retrieve messages from the WebSphere MQ queue, use the get(MQMessage message) or get(MQMessage, MQGetMessageOptions gmo), get(MQMessage, MQGeMessageOptions gmo, int maxMessageSize) methods of the MQQueue class.

MQGetMessageOptions

The way messages are retrieved from the queue is determined by the value of the Options field of the instance of the MQGetMessageOptions class. Set the value of Options using the MQGetMessageOptions (MQGMO) constants in the MQC interface.

The new instance of the MQGetMessageOptions class has the value of the Options property set to default. Set the appropriate get message options using the MQGMO constants in the MQC interface. Example B-2 shows the WebSphere MQ get message option. This option specifies that the get message call must return immediately if there are no messages on the queue.

Example: B-2 WebSphere MQ get message option

```
MQGetMessageOptions gmo = new MQGetMessageOption();
gmo.options = gmo.options + MQC.MQGMO_NO_WAIT;
```

Application development

The following sections demonstrate how messages can be put and got from a queue or queues with the help of simple examples. The examples use a point-to-point programming approach, where applications act in pairs. A sending application called *sender* puts messages on a WebSphere MQ application queue on the sending side. On the destination system or receiving side, an application called a *receiver* retrieves messages from the WebSphere MQ application queue.

The approach to point-to-point application development is summarized in Figure B-1.

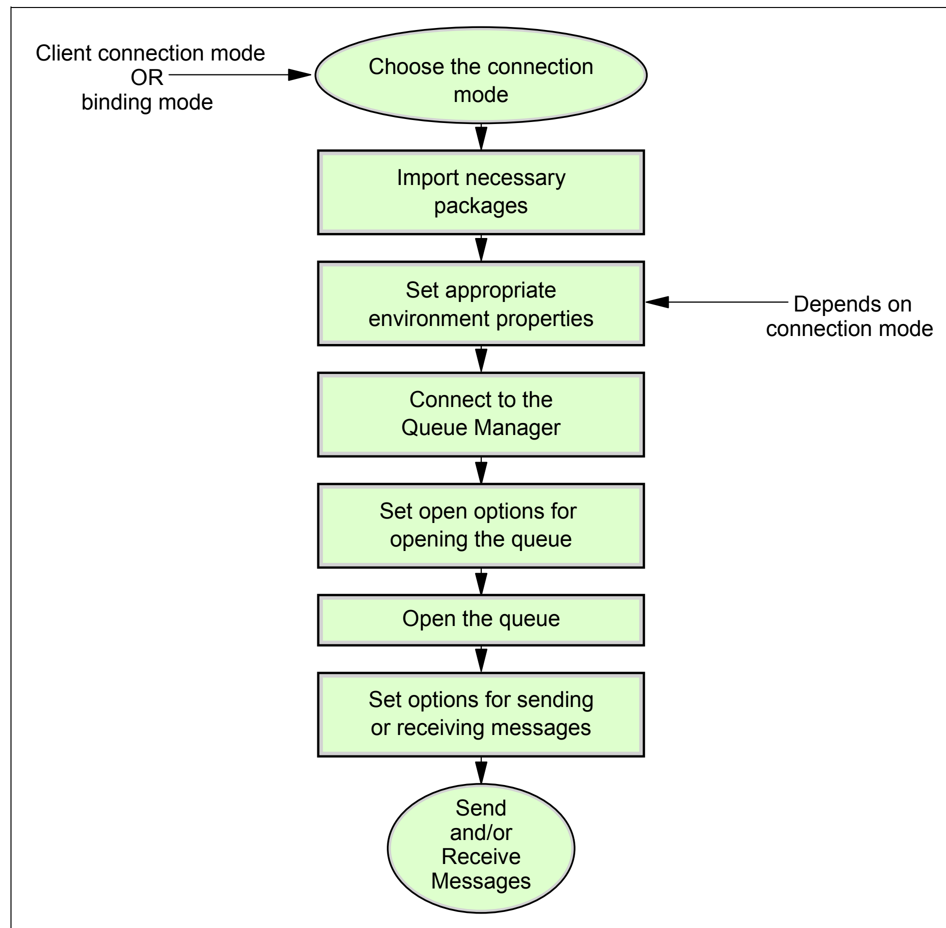


Figure B-1 Approach to point-to-point application development

Simple WebSphere MQ put operation

This section demonstrates a client program that creates a simple message and sends it to a WebSphere MQ queue.

The steps involved are:

1. Import the WebSphere MQ Java application programming interface (API) package.
2. Set up the environment properties for the client connection.
3. Connect to the queue manager.

4. Set the options for opening the WebSphere MQ queue.
5. Open the application queue for sending messages.
6. Set the options to put messages on the application queue.
7. Create a message buffer.
8. Prepare the message with user data and message descriptor fields, if any.
9. Put the message on the queue.

The program example, SimpleSender.java, which is shown in Example B-3, is a sender application that sends messages to a queue.

Example: B-3 SimpleSender.java

```
import com.ibm.mq.*;
public class SimpleSender {
   public static void main(String args[]) {
      try
      {
         String hostName = "kodiac" ;
         String channel = "JAVA.CLIENT.SVRCONN" ;
         String qManager = "ITSO" ;
         String qName = "SAMPLE.QUEUE" ;
         //Set up the MQEnvironment properties for Client Connections
         MQEnvironment.hostname = hostName ;
         MQEnvironment.channel = channel ;
         MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                       MQC.TRANSPORT_MQSERIES);
         //Connection To the Queue Manager
         MQQueueManager qMgr = new MQQueueManager(qManager) ;
         /* Set up the open options to open the queue for out put and
            additionally we have set the option to fail if the queue manager is
            quiescing.
         */
         int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
         //Open the queue
         MQQueue queue = qMgr.accessQueue(qName,
                                          openOptions,
                                          null,
                                          null,
                                          null);
         // Set the put message options , we will use the default setting.
         MQPutMessageOptions pmo = new MQPutMessageOptions();
         /* Next we Build a message. The MQMessage class encapsulates the data
            buffer that contains the actual message data, together with all the MQMD
            parameters that describe the message. To Build a new message, create a
            new instance of MQMessage class and use writxxx (we will be using
            writeString method). The put() method of MQQueue also takes an instance
```

```
            of the MQPutMessageOptions class as a parameter.
         */
         MQMessage outMsg = new MQMessage(); //Create The message buffer
         outMsg.format = MQC.MQFMT_STRING ; // Set the MQMD format field.
         //Prepare message with user data
         String msgString = "Test Message from SimpleSender program ";
         outMsg.writeString(msgString);
         // Now we put The message on the Queue
         queue.put(outMsg, pmo);
         //Commit the transaction.
         qMgr.commit();
         System.out.println(" The message has been put\n\n#########");
         // Close the Queue and Queue manager objects.
         queue.close();
         qMgr.disconnect();
      }
      catch (MQException ex){
         System.out.println("An MQ Error Occurred: Completion Code is :\t" +
                           ex.completionCode + "\n\n The Reason Code is :\t" +
ex.reasonCode );
         ex.printStackTrace();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

SimpleSender.java uses client mode to connect to the queue manager. MQEnvironment.port is not set because it defaults to 1414. To use binding mode, comment out the calls at the beginning to set options in the MQEnvironment. By default, the binding mode is used.

Simple WebSphere MQ get operation

This section demonstrates a simple client, SimpleReceiver.java, which gets the message from a queue and prints out the message data to the console.

The steps involved are:

1. Import the WebSphere MQ Java API package.
2. Set up the environment properties for client connection.
3. Connect to the queue manager.
4. Set the options for opening the WebSphere MQ queue.
5. Open the application queue for getting messages.

6. Set the options to get messages from the application queue.
7. Create a message buffer.
8. Get the message from the queue to the message buffer.
9. Read the user data from the message buffer and display on the console.

Example B-4 shows the code for SimpleReceiver.java.

Example: B-4 SimpleReceiver.java

```
import com.ibm.mq.* ;
public class SimpleReceiver {
   public static void main(String args[]) {
      try
      {
         String hostName = "kodiac" ;
         String channel = "JAVA.CLIENT.SVRCONN" ;
         String qManager = "ITSO" ;
         String qName = "SAMPLE.QUEUE" ;
         //Set up the MQEnvironment properties for Client Connections
         MQEnvironment.hostname = hostName ;
         MQEnvironment.channel = channel ;
         MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                      MQC.TRANSPORT_MQSERIES);
         //Connection To the Queue Manager
         MQQueueManager qMgr = new MQQueueManager(qManager) ;
         /* Set up the open options to open the queue for out put and
            additionally we have set the option to fail if the queue manager is
            quiescing.
         */
         int openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING;
         //Open the queue
         MQQueue queue = qMgr.accessQueue(qName,
                                          openOptions,
                                          null,
                                          null,
                                          null);
         // Set the put message options.
         MQGetMessageOptions gmo = new MQGetMessageOptions();
         gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get under sync
         gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait if no messages
         gmo.options = gmo.options + MQC.MQGMO_FAIL_IF_QUIESCING ;
         gmo.waitInterval = 3000 ; // Sets the time limit for the wait.
         /* Next we Build a message The MQMessage class encapsulates the data
            buffer that contains the actual message data, together with all the
            MQMD parameters that describe the message.
         */
```

```
         MQMessage inMsg = new MQMessage(); //Create the message buffer
         // Get the message from the queue on to the message buffer.
         queue.get(inMsg, gmo) ;
         // Read the User data from the message.
         String msgString = inMsg.readStringOfCharLength(inMsg.getMessageLength());
         System.out.println(" The Message from the Queue is : " + msgString);
         //Commit the transaction.
         qMgr.commit();
         // Close the Queue and Queue manager objects.
         queue.close();
         qMgr.disconnect();
      }
      catch (MQException ex){
         System.out.println("An MQ Error Occurred: Completion Code is :\t" +
                            ex.completionCode + "\n\n The Reason Code is :\t" +
ex.reasonCode );
         ex.printStackTrace();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Request-and-reply messaging pattern

Following is the process involved in a request-and-reply messaging pattern:

1. An application sends a message (request message) to another application (reply producer).
2. The reply producer then responds to the request message.
3. The reply-producing application gets the request message, processes the request, and sends a response back to the requesting application.
4. The request message header property of replyToQueue specifies the queue the reply message goes to. The replyToQueueManager message header property of the request message specifies the queue manager to which it belongs.
5. The requesting application sets these message header properties on the request message before putting the message on the queue.
6. The requesting application lets the queue manager generate an unique messageID.

7. The replying application copies the messageID of the request message to the correlationID of the reply message. The requesting application uses the correlationID value of the reply message to map a response back to the original request.

The request-and-reply pattern is illustrated with a pair of simple applications.

1. The first application, requester, puts a simple message on the queue (request queue).
2. The requester sets the replyToQueue and replyToQueueManager message header properties on the request message before putting the request message on the request queue.
3. The requester then opens the reply queue and waits for messages with the correlationID matching the messageID value of the outgoing request message.
4. The responding application servicing the request message gets the request message, prepares the reply message, and sends it to the reply queue under the queue manager specified in the request message. It also copies the messageID from the request message to the correlationID message header field of the response message.

Requester application

The application, Requester.java, is the application that sends the request message and expects a reply from the responding application. Following are the steps involved in this process:

1. Import the necessary package.
2. Set the MQEnvironment properties for client connection.
3. Connect to the queue manager.
4. Open the request queue for output.
5. Set the put message options.
   a. Prepare the request message.
   b. Set the reply to the queue name.
6. Set the reply to queue manager name.
7. Put the request message on the request queue.
8. Close the request queue.
9. Open the reply queue for input.
10. Set the get message options.
    a. Set the option to match the correlationID in the response message.
    b. Issue get on the reply queue with wait for response message with matching correlationID.

Important: Using a definite wait time on the get call for response messages is recommended. The wait interval can be derived from the maximum time the system is allowed to wait for a response.

Requester.java is an application that sends a request message and expects a reply, as shown in Example B-5.

Example: B-5 Requester.java

```
import com.ibm.mq.*;
public class Requester {
   public static void main(String args[]) {
      try
      {
         String hostName = "kodiac" ;
         String channel = "JAVA.CLIENT.SVRCONN" ;
         String qManager = "ITSO" ;
         String requestQueue = "SAMPLE.REQUEST" ;
         String replyToQueue = "SAMPLE.REPLY" ;
         String replyToQueueManager = "ITSO" ;
         //Set up the MQEnvironment properties for Client Connections
         MQEnvironment.hostname = hostName ;
         MQEnvironment.channel = channel ;
         MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                      MQC.TRANSPORT_MQSERIES);
         //Connection To the Queue Manager
         MQQueueManager qMgr = new MQQueueManager(qManager) ;
         /* Set up the open options to open the queue for out put and
            additionally we have set the option to fail if the queue manager is
            quiescing.
         */
         int openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
         //Open the queue
         MQQueue queue = qMgr.accessQueue(requestQueue,
                                          openOptions,
                                          null,
                                          null,
                                          null);
         // Set the put message options , we will use the default setting.
         MQPutMessageOptions pmo = new MQPutMessageOptions();
         pmo.options = pmo.options + MQC.MQPMO_NEW_MSG_ID ;
         pmo.options = pmo.options + MQC.MQPMO_SYNCPOINT ;
         MQMessage outMsg = new MQMessage(); //Create the message buffer
         outMsg.format = MQC.MQFMT_STRING ; // Set the MQMD format field.
```

```
            outMsg.messageType = MQC.MQMT_REQUEST ;
            outMsg.replyToQueueName = replyToQueue;
            outMsg.replyToQueueManagerName = replyToQueueManager ;
            //Prepare message with user data
            String msgString = "Test Request Message from Requester program ";
            outMsg.writeString(msgString);
            // Now we put The message on the Queue
            queue.put(outMsg, pmo);
            //Commit the transaction.
            qMgr.commit();
            System.out.println(" The message has been Sussesfully put\n\n#########");
            // Close the Request Queue
            queue.close();
            // Set openOption for response queue
            openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;
            MQQueue respQueue = qMgr.accessQueue(replyToQueue,
                                                 openOptions,
                                                 null,
                                                 null,
                                                 null);
            MQMessage respMessage = new MQMessage();
            MQGetMessageOptions gmo = new MQGetMessageOptions();
            gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get messages under
syncpoint
            gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait for Response Message
            gmo.matchOptions = MQC.MQMO_MATCH_CORREL_ID;
            gmo.waitInterval = 10000 ;
            respMessage.correlationId = outMsg.messageId ;
            // Get the response message.
            respQueue.get(respMessage, gmo);
            String response =
respMessage.readStringOfCharLength(respMessage.getMessageLength());
            System.out.println("The response message is : " + response);
            qMgr.commit();
            respQueue.close();
            qMgr.disconnect();
         }
         catch (MQException ex)
         {
            System.out.println("An MQ Error Occurred: Completion Code is :\t" +
                               ex.completionCode + "\n\n The Reason Code is :\t" +
ex.reasonCode );
```

```
         ex.printStackTrace();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Responder application

The responder application, Responder.java, processes the request message from the request queue and sends a reply back to the requesting application, as shown in Example B-6.

Example: B-6 Responder.java

```
import com.ibm.mq.* ;
public class Responder {
   public static void main(String args[]) {
      try
      {
         String hostName = "kodiac" ;
         String channel = "JAVA.CLIENT.SVRCONN" ;
         String qManager = "ITSO" ;
         String qName = "SAMPLE.REQUEST" ;
         // set up the MQEnvironment properties for Client Connections
         MQEnvironment.hostname = hostName ;
         MQEnvironment.channel = channel ;
         MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
                                 MQC.TRANSPORT_MQSERIES);
         //Connection To the Queue Manager
         MQQueueManager qMgr = new MQQueueManager(qManager) ;
         /* Set up the open options to open the queue for out put and
          additionally we have set the option to fail if the queue manager is
          quiescing.
         */
         int openOptions = MQC.MQOO_INPUT_SHARED | MQC.MQOO_FAIL_IF_QUIESCING ;
         //Open the queue
         MQQueue queue = qMgr.accessQueue(qName,
                                 openOptions,
                                 null,
                                 null,
                                 null);
         // Set the put message options.
         MQGetMessageOptions gmo = new MQGetMessageOptions();
```

```
         gmo.options = gmo.options + MQC.MQGMO_SYNCPOINT ; //Get messages under
syncpoint control
         gmo.options = gmo.options + MQC.MQGMO_WAIT ; // Wait if no messages on the
queue
         gmo.options = gmo.options + MQC.MQGMO_FAIL_IF_QUIESCING ; // Fail if QMGR
Quiescing
         gmo.waitInterval = 3000 ; // Sets the time limit for the wait.
         /* Next we Build a message The MQMessage class encapsulates the data buffer
          that contains the actual message data, together with all the MQMD
          parameters
          that describe the message.
          To Build a new message, create a new instance of MQMessage class and use
          writexxx (we will be using writeString method). The put() method of
          MQQueue also takes an instance of the MQPutMessageOptions class as a
parameter.
         */
         MQMessage inMsg = new MQMessage(); //Create the message buffer
         // Get the message from the queue on to the message buffer.
         queue.get(inMsg, gmo) ;
         // Read the User data from the message.
         String msgString = inMsg.readStringOfCharLength(inMsg.getMessageLength());
         System.out.println(" The Message from the Queue is : " + msgString);
         //Check if message if of type request message and reply to the request.
         if (inMsg.messageType == MQC.MQMT_REQUEST ) {
            System.out.println("Preparing To Reply To the Request " );
            String replyQueueName = inMsg.replyToQueueName ;
            openOptions = MQC.MQOO_OUTPUT | MQC.MQOO_FAIL_IF_QUIESCING ;
            MQQueue respQueue = qMgr.accessQueue(replyQueueName,
                                                 openOptions,
                                                 inMsg.replyToQueueManagerName,
                                                 null,
                                                 null);
            MQMessage respMessage = new MQMessage() ;
            respMessage.correlationId = inMsg.messageId;
            MQPutMessageOptions pmo = new MQPutMessageOptions();
            respMessage.format = MQC.MQFMT_STRING ;
            respMessage.messageFlags = MQC.MQMT_REPLY ;
            String response = "Reply from the Responder Program " ;
            respMessage.writeString(response);
            respQueue.put(respMessage, pmo);
            System.out.println("The response Successfully send ");
            qMgr.commit();
            respQueue.close();
         }
         queue.close();
```

```
         qMgr.disconnect();
      }
      catch (MQException ex)
      {
         System.out.println("An MQ Error Occurred: Completion Code is :\t" +
                          ex.completionCode + "\n\n The Reason Code is :\t" +
ex.reasonCode );
         ex.printStackTrace();
      }
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Transaction participation with SOAP/WebSphere MQ

The WebSphere MQ SOAP transport has transactional support in the service and in the client when using the MA0V SupportPac. This has ramifications for the way these two components deal with transactions involving WebSphere MQ, considering that they also use WebSphere MQ. Applications that perform transactional messaging share the same MQQueueManager object, which represents a connection to the queue manager. A single transaction cannot span multiple MQQueueManager objects.

To deal with this, the WebSphere MQ Java classes provide a mechanism to allow transaction association or participation within the context of a single thread. This mechanism hinges around two new method calls on MQEnvironment and three new properties that are housed in the MQC class.

The new method calls in the MQEnvironment are:

- MQEnvironment.getQueueManagerReference(int)
- MQEnvironment.getQueueManagerReference(int, Object)

The int is one of the three new properties in the MQC class, and the Object is a Java string containing the name of the WebSphere MQ queue manager.

The new properties in MQC are:

- ASSOCIATE_ALL

  public final static int: This value indicates that the MQQueueManager object that is being created can be shared within the context of the Java Virtual Machine (JVM). As a result, its use must not involve transactions because other threads, or even the same thread, can obtain a reference to this object using the new MQEnvironment methods.

- ASSOCIATE_NONE

  public final static int: This value indicates that the MQQueueManager object that is created cannot be shared in any context. This is the default and behaves the same way as the WebSphere MQ 5.3 Java classes. Because it is not shared, it is safe to use this option for transactions. This is because a reference to this object can only be passed by the user code. It cannot be obtained by using the new MQEnvironment methods.

- ASSOCIATE_THREAD

  This value indicates that the MQQueueManager object being created can be shared within the context of the currently executing thread. Consequently, the code in the same thread is able to obtain a reference to this object using the new MQEnvironment methods. However, it is safe from any other thread. This option is the key to allowing the WebSphere MQ sender code to participate in transactions in the client, and the Web Service code to participate in transactions in the WebSphere MQ listener.

Web Service client transaction participation

Using the MA0V SupportPac, a Web Service client is able to invoke a Web Service asynchronously in a transaction. Refer to Chapter 16, “Transactional functionality (MA0V)” on page 339 for more information about this. If, in the same transaction, the client also wishes to implement messaging, the code must make use of the ASSOCIATE_THREAD option. This allows the underlying WebSphere MQ sender to obtain a reference to the same MQQueueManager object using the new MQEnvironment.getQueueManagerReference(). With this reference, the WebSphere MQ sender is able to participate in the same transaction.

Example B-7 shows how to create such a MQQueueManager object using the MQQueueManager constructor. This code acts as the transaction owner. Subsequent calls to qm.begin() and qm.commit() may encompass invocations of a Web Service asynchronously. Only when qm.commit() is called is the invocation message sent.

Example: B-7 Creating MQQueueManager with ASSOCIATE_THREAD

```
Hashtable props = new Hashtable();
props.put(MQC.MQ_QMGR_ASSOCIATION_PROPERTY, new
Integer(MQC.ASSOCIATE_THREAD);
MQQueueManager qm = new MQQueueManager("ITSO", props);
```

Web Service transaction participation

The WebSphere MQ listener is able to run transactionally as detailed in Chapter 5, "SOAP/WebSphere MQ implementation" on page 49. For the Web Service code to implement messaging and participate in the same transaction, it must obtain a reference to the underlying MQQueueManager object used by the WebSphere MQ listener. Example B-8 shows how to obtain such a MQQueueManager object.

Example: B-8 Obtaining an MQQueueManager reference with ASSOCIATE_THREAD

```
MQQueueManager qm =
   MQEnvironment.getQueueManagerReference(MQC.ASSOCIATE_THREAD,
"ITSO");
```

The Web Service code acts as a transaction participant. Any messaging done with this MQQueueManager object is committed only when the Web Service code completes and the underlying WebSphere MQ listener sends the response, and then calls the qm.commit().

C

Deployment utility quick reference

This appendix provides an overview of the parameters that are used with the deployment utility. The deployment utility is written in Java. Access the source code from the location <WebSphere MQ Install Directory>\Tools\Soap\Samples\DeployWMQService.java.

Two script files are provided to start the deployment utility, one for UNIX and one for Windows. Example C-1 shows the syntax for UNIX.

Example: C-1 Syntax for UNIX

```
.amqwdeployWMQService.sh -f className [-a integrityOption] [-b
bothresh] [-c operation] [-i passContext] [-n num] [r] [-s] [-tmp
programName] [=tmq queueName] [-u URI] [-v] [-x transactionality] [-?]
[SSL options]
```

Example C-2 shows the syntax for Windows.

Example: C-2 Syntax for Windows

```
amqwdeployWMQService.cmd -f className [-a integrityOption] [-b
bothresh] [-c operation] [-i passContext] [-n num] [r] [=s] [-tmp
programName] [-u URI] [-v] [-x transactionality] [=?] [SSL options]
```

© Copyright IBM Corp. 2006. All rights reserved.

The parameters are described in Table C-1.

Table C-1 Deployment utility parameters

Parameter	Description
-f	Name of the class being deployed ► For Java classes, should be fully qualified, for example, com/ibm/test/testSvc.java ► For .NET, should be the name of the Web Service, for example, testSvc.asmx.cs
-a	Changes the behavior of the listener if it cannot place a failed message on the dead letter queue. Following are the values that are allowed: ► DefaultMsgIntegrity: Nonpersistent messages are discarded with a warning and persistent messages are backed out to the request queue with an error message shown and the listener stopped. ► LowMsgIntegrity: For persistent and nonpersistent messages, the message is discarded with a warning. Listener continues. ► HighMsgIntegrity: For persistent and nonpersistent messages, an error message is shown, and the message is backed out to the request queue. Listener terminates.
-b	Numeric value specifying the backout threshold for the request queue. DEFAULT is 3.
-c	Specifies which part of the deployment process to start: ► allAxis: All compile/start steps for Axis/Java service ► compileJava: Compile Java service only ► genAxisWsdl: Generate a Web Services Description Language (WSDL) file for Axis service ► axisDeploy: Deploy the class file (convert .wsdl to .wsdd and apply wsdd) ► genProxiesToAxis: Generate the Axis proxies (.java and .class from .wsdl) ► genAxisWMQBits: Set up queues, listeners, and triggers for Axis service ► allAsmx: Perform all steps for a .NET service ► genAsmxWsdl: Generate WSDL for a .NET service ► genProxiesToDotNet: Generate proxies for .NET service (.wsdl to .java, .class, .cs, and .vb) ► genAsmxWMQBits: Set up queues, listeners, and triggers for .NET service ► startWMQMonitor: Start the trigger monitor for SOAP/WebSphere MQ
-i	Specifies whether the listener should pass identity context. Can be: ► passContext (DEFAULT) ► ownContext
-n	Number of threads to be specified for listener. DEFAULT is 10.

Parameter	Description
-r	Specifies that any existing request queues or trigger monitor queues are replaced with default attributes and no messages
-s	Configures the listener to be started as a WebSphere MQ service. Not to be used with the -tmq option because this leads to an error.
-tmp	Specfies a trigger monitor program
-tmq	Specifies a trigger monitor queue name
-u	Universal Resource Indicator (URI) option.
-v	Sets verbose option for external commands
-x	Sets form of transactional control. Can be one of the following: ► onePhase: WebSphere MQ one-phase support used. In the case of failure, request message returned to the application. Response message delivered exactly once. ► twoPhase: WebSphere MQ two-phase support used. If other resources are used, the message is delivered once with a single committed execution of all. Applies only to server binding. ► none: No transactionality. Even persistent messages can be lost if the system fails.
-?	Shows help text
SSL options	Provides Secure Socket Layer (SSL) security specifications

The URI can take a number of name=value parameters. The syntax for an URI is:

```
jms:/queue?name=value&name=value
```

The parameters are illustrated in Table C-2.

Table C-2 URI parameters

Parameter	Required	Description
destination	YES	Name of the request queue. Can include just queue name or queue and queue manager names seperated by an @ character, for example, myRequestQ@mqQM.
connectionFactory	YES	See Table C-3.
initialContextFactory	YES	Must be set to com.ibm.mq.jms.Nojndi for compatibility reasons

Parameter	Required	Description
timeout	NO	Time in milliseconds the client waits for a response. If not specified, it uses the application or infrastructure default.
targetService	YES (for .NET services)	Allows a single SOAP/WebSphere MQ listener to process requests for multiple .NET services. Axis infrastructure permits this by default. Value parameters is service name. No qualifiers for .NET. Fully qualified for Java.
timeToLive	NO	Expiry time of message in milliseconds. DEFAULT is 0, an unlimited lifetime.
persistence	NO	Message persistence. Can be one of: ► 0: None specified, uses the default for queue. ► 1: Nonpersistent ► 2: Persistent
priority	NO	Message priority. Can be a numeric value between 0 (low) and 9 (high). Default is 0.
replyDestination	NO	Response queue name. DEFAULT is SYSTEM.SOAP.RESPONSE.QUEUE.

The ConnectionFactory parameter can itself take a number of parameters. These are in the following format:

```
connectionFactory=name(value)name(value)
```

If there are no ConnectionFactory parameters, the ConnectionFactory string looks as follows:

```
connectionFactory=()
```

The options are summarized in Table C-3.

Table C-3 connectionFactory parameters

Parameter	Description
connectQueueManager	Queue Manager to connect to. DEFAULT is blank.

Parameter	Description
binding	The type of binding to use when connecting to the queue manager. If none are specified and client options are specified, the code assumes a client binding. If there are no client binding options, the DEFAULT is auto. This means a server connection is attempted, followed by a client connection. Other options are: ► server ► client ► xaclient (.NET only)
clientChannel	Specifies the channel name. DEFAULT is null. Must be specified if clientConnection parameter is used.
clientConnection	Specifies the client connection name. Can be a host name or an IP address.

Sample deployment command lines

Example C-3 shows the sample deployment command lines.

Example: C-3 Sample deployment command lines

```
amqwdeployWMQService -f sample.axisSvc.BankingService.java

amqwdeployWMQService -f service.asmx -u
"jms:/queue?destination=myRequestQ&connectionFactory=()&targetService=s
ervice.asmx&initialContextFactory=com.ibm.mq.jms.Nojndi"

amqwdeployWMQService -f service.asmx -u
"jms:/queue?destination=myRequestQ@myQM&connectionFactory=connectQueueM
anager(myQM)&targetService=service.asmx&initialContextFactory=com.ibm.m
q.jms.Nojndi"

amqwdeployWMQService -f sample.test.testSvc.java -u
"jms:/queue?destination=myRequestQ@myQM&connectionFactory=connectQueueM
anager(myQM)&replyDestination=myResponseQ&initialContextFactory=com.ibm
.mq.jms.Nojndi"
```

```
amqwdeployWMQService -f sample.test.testSvc.java -u
"jms:/queue?destination=myRequestQ@myQM&connectionFactory=connectQueueM
anager(myQM)binding(client)clientChannel(clientChl)clientConnection(192
.160.1.1)&replyDestination=myResponseQ&initialContextFactory=com.ibm.mq
.jms.Nojndi"
```

Additional material

This appendix provides information about the additional material you can download from the Internet as described here.

Locating the Web material

The Web material associated with this IBM Redbook is available in softcopy on the Internet at the IBM Redbooks Web site:

ftp://www.redbooks.ibm.com/redbooks/SG247115

Alternatively, go to the IBM Redbooks Web site, select **Additional materials**, and open the directory that corresponds to the IBM Redbook form number SG24-7115.

http://www.ibm.com/redbooks

© Copyright IBM Corp. 2006. All rights reserved.

Using the Web material

The additional Web material that accompanies this IBM Redbook includes the following files:

File name	Description
Allfiles.zip	All the compressed code files
AppendixB.zip	Appendix B compressed file
Chapter10.zip	Chapter 10 compressed file
Chapter13.zip	Chapter 13 compressed file

How to use the Web material

Create a subdirectory (folder) in your workstation and extract the contents of the Web material compressed file into this folder.

Abbreviations and acronyms

ACL	access control list
ADO	Active Data Object
ASP	Active Server Page
CA	certificate authority
CICS	Customer Information Control System
CLR	common language runtime
COM	Component Object Model
CRL	certificate revocation list
DCOM	distributed component object model
DIME	Direct Internet Message Encapsulation
DLL	dynamic link library
DN	distinguished name
DTC	Distributed Transaction Coordinator
EAR	enterprise archive
EJB	Enterprise JavaBeans
GAC	Global Assembly Cache
GSK	Global Security Kit
GUI	graphical user interface
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines Corporation
IDE	integrated development environment
IIOP	Internet Inter-ORB Protocol
IIS	Internet Information Services
iKeyman	IBM key management utility
ITSO	International Technical Support Organization
IVT	installation verification testing
jar	Java archive
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JSSE	Java Secure Socket Extension
JVM	Java virtual machine
MAC	message authentication code
MCA	message channel agent
MCS	Microsoft Certificate Stores
MDB	message-driven bean
MQI	Message Queue Interface
MQMD	Message descriptor
MTS	Microsoft Transaction Server
OAM	object authority manager
PKI	public key infrastructure
RFC	Request for Comments
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDK	software development kit
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Sockets Layer
UDDI	Universal Description, Discovery, and Integration
UDP	User Datagram Protocol
URI	Universal Resource Indicator
URL	Universal Resource Locator
W3C	World Wide Web Consortium

© Copyright IBM Corp. 2006. All rights reserved.

WSDL Web Services Description Language

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redbook.

IBM Redbooks

For information about ordering these publications, see "How to get IBM Redbooks" on page 436. Note that some of the documents referenced here may be available in softcopy only.

- *WebSphere and .NET Coexistence*, SG24-7027
- *WebSphere and .Net Interoperability Using Web Services*, SG24-6395

Other publications

The following publication is also relevant as further information source:

Englander Robert, *Java and SOAP*, O'Reilly, 2002, ISBN 0596001754

Online resources

The following Web sites and URLs are also relevant as further information sources:

- Apache Axis WSDL information

  `http://ws.apache.org/axis/java/user-guide.html#Introduction`
- Internet X.509 Public Key Infrastructure

  `http://www.ietf.org/html.charters/pkix-charter.html`
- Microsoft .NET Framework V1.1

  `http://www.microsoft.com/downloads/details.aspx?familyid=262D25E3-F589-4842-8157-034D1E7CF3A3&displaylang=en`
- Microsoft .NET SDK V1.1

  `http://www.microsoft.com/downloads/details.aspx?FamilyID=9B3A2CA6-3647-4070-9F41-A333C6B9181D&displaylang=en`

© Copyright IBM Corp. 2006. All rights reserved.

- OpenSSL

  http://www.openssl.org
- SSL 3.0 specification

  http://wp.netscape.com/eng/ssl3/
- UDDI organization run by Oasis

  http://www.uddi.org
- WebSphere Application Server

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.base.doc/info/welcome_base.html

  http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html

  http://www-306.ibm.com/software/integration/support/supportpacs/category.html#cat2

  http://www-128.ibm.com/developerworks/websphere/library/techarticles/0505_hiscock/0505_hiscock.html
- World Wide Web consortium

  http://www.w3.org
- WSDL-related information

  http://www-128.ibm.com/developerworks/webservices/library/ws-whichwsdl/index.html
  http://www.w3.org/2002/ws/desc
- xml.org organization run by Oasis

  http://www.xml.org
  http://www.xml.com
  http://www.w3.org/XML

How to get IBM Redbooks

You can search for, view, or download IBM Redbooks, IBM Redpapers, Hints and Tips, draft publications, and Additional materials, and order a hardcopy of IBM Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

Symbols
.NET client 243
.NET monitor 404
.NET Remoting 21
.NET Web Service
    compiling 221
.NET Web service
    security
        Java environment 239
        Microsoft .NET environment 239
.NET Web Service client
    design 244
    development 243

A
access relational data 21
Active Data Objects, *See* ADO
Active Server Pages, *See* ASP
additional WebSphere MQ configuration 167
ADO 21
algorithm 112
Apache Axis 24
    Java archive file 145
APAR
    PK05012, WAS 6.x 101
    PK05013, WAS 5.x 101
application
    component 22
    development 389
ASP 20
asymmetric key algorithm 113
asynchronous
    callback 315
    calls 252
    facility 303
    messaging 5
asynchrony and transactionality 301
authority service 109
Axis client 33, 187
    calling get method 196
    client code 195
    connect to SOAP WebSphere MQ framework 188
    implementation 190
    security
        key store 211
        trust store 211
Axis client-service interaction 189
Axis Web Service 159
    create 159
    create, deploy 159
    deploy 163
        to local default queue manager 167
        to local queue manager 170
        to remote queue manager 171
    design 160
Axis Web service
    implementation 162

B
binding
    mode 406
    type 67
bracket characters 174
buffered data 5

C
C# proxies 63
CA 117
call from .NET client to Web Service 34
callback
    function 254
    method 97
    technique 254
calls
    begin, commit, backout 358
cell scope 279
certificate authority, *See* CA
certificate request 119
certificate revocation list, *See* CRL
channel configuration example 156
channel or message channel agent 45
CICS 104
    interoperation 104
CICS Transaction Server 100
cipher text 112–113

© Copyright IBM Corp. 2006. All rights reserved.

client
application 32
configuration file 197
connection 382
container 291
invocation 377
mode 256, 406
transactionality
Axis 348
CLR 20, 34
code 2051 208
COM+ or Enterprise Services 21
command
amqwDeployWMQService 55
amqwsetcp 167, 229, 231
amqwsetcp.cmd/sh 51
csc 222
END 173
endWMQJListener 178
runmqsc 89, 156, 278
startWMQJListener 177
wsdl.exe 248
common deployment steps 165
Common Language Runtime, *See* CLR
communication between queue managers 61
compiling proxies 76
completion code of 2 262
complex data types 38
computational algorithm 113
configure
SSL 107
WebSphere MQ 282
connection factories 279
container 22
create
.NET Web service 213
local queue 171
local queue on queue manager 151
queue manager 151
Windows application project 247
create default queue manager 164
credit button event handler 253
credit method 218, 252, 357
CRL 118
cryptography 113
asymmetric key algorithm 114
concepts 110
decryption 113
encryption 113
encryption and decryption algorithms 113
custom utility 73
Customer Information Control System, *See* CICS
customized deployment
Axis 77
Microsoft .NET 74

D

data integrity 112
DCOM 5
debit method 218
decipherment 113
decryption 113
default WebSphere MQ objects 53
default model queue (SYSTEM.SOAP.MODEL.RESPONSE.QUEUE) 53
default response queue (SYSTEM.SOAP.RESPONSE.QUEUE) 53
queue manager 53
side queue (SYSTEM.SOAP.SIDE.QUEUE) 53
deployment
.NET Web service 227
client 295
customize process 73
request queue
validation 71
response queue
SYSTEM.SOAP.RESPONSE.QUEUE 72
sample command lines 429
script 61
steps 228
to local default queue manager 229
to local queue manager 230
to remote queue manager 171, 232
utility 43, 61, 425
syntax 64
WebSphere MQ Transport for SOAP 59
descriptor file 62
design
.NET Web Service 215
WebSphere Application Server client 291
WebSphere MQ 271
design concepts 188
develop
transactional Axis client 350
transactional Microsoft .NET client 345
development process
WebSphere MQ transport for SOAP 53

access features using URI 54
activate listener 56
call amqwClientConfig.cmd/sh 56
deploy service 55
run client application 57
write and prepare client code 55
write and prepare service 53
digital certificate 117
certificate revocation list 118
verification 118
digital signature 116
handling 112
distinguished name, *See* DA
distinguished name, *See* DN
Distributed Component Object Model, *See* DCOM
distributed component technology 18
Distributed Transaction Coordinator, *See* DTC
DN 118, 136
DTC 344
dynamic queue 379
dynamic response queue 88

E

EAR file 284
EJB 22
container 291
enable
application integration 7
SSL 61
encipherment 113
encoding options 35
encrypting data 112
encryption and decryption 113
process 113
Enterprise JavaBeans, *See* EJB
Enterprise Services or COM+ 21
environment setup 147, 386
server binding mode 259
environment variable
LD_LIBRARY_PATH 57
PATH 57
SHLIB_PATH 57
WMQSOAP_HOME, setting 51
error handling
.NET Web Service client 262
incorrect message format 210, 265
listener not started 265
unable to find specified WebSphere MQ object 264
unable to get response from queue 263
unable to put request to queue 262
Axis client 208
Axis Web Service 180
unable to find specified request queue 181
unable to find specified response queue 182
unable to get response from queue 180
unable to put to a response queue 182
unexpected message on queue 183
SOAP/WebSphere MQ Web Service 236
unable to find specified WebSphere MQ object 238
unable to find specified WebSphere MQ object-request queue 237
unable to get from response queue 236
unable to put to a response queue 237
unexpected message on queue 238
error message logging 360
Extensible Markup Language, *See* XML

G

GAC 20
general invocation code 252
generated proxy 73
getBalance method 219
getStatement method 219
Global Assembly Cache, *See* GAC
global communication 3
graphical user interface, *See* GUI
GSKit 107
GUI 244
GUI buttons
Credit 252
Debit 251
View Balance 251
View Statement 252

H

HTTP 14, 27, 213
Hypertext Transfer Protocol, *See* HTTP

I

IBM Global Security Kit, *See* GSKit
IBM Redbooks Web site 436
Contact us xxiv

identity context 85
IIOP 23
IIS 20
implement
  client side transactionality 355
implementation
  BankingService Web Service 217
  BankingService Web Service client 245
  Web Service from WSDL 272
  WebSphere Application Server client 292
installation
  AIX 144
  IBM WebSphere Application Server V6 for AIX 146
  IBM WebSphere MQ V6 142
  Microsoft .NET Framework Redistributable V1.1 145
  Microsoft .NET Software Development Kit V1.1 145
  Rational Application Developer V6 147
  WebSphere MQ Transport for SOAP
    verification 146
  Windows 143
Installation Verification Test, *See* IVT
integrity and persistence settings 237
Internet Information Services, *See* IIS
Internet Inter-ORB Protocol, *See* IIOP
interoperability
  WebSphere Application Server and CICS Transaction Server 100
invoking Web service 248
IVT 51
  tests 163

J

J2EE 4, 13
Java 2 Platform, Enterprise Edition, *See* J2EE
Java Message Service, *See* JMS
Java Naming and Directory Interface, *See* JNDI
Java program
  com.ibm.mq.soap.util.DeployWMQService 59
Java resource manager interface 23
Java Virtual Machine, *See* JVM
Java Web service client 358
JMS 19
  provider 19
JNDI 23, 40, 281
JVM 22

L

listener
  configuration 167
  port 281
  transactionality 86

M

MA0V SupportPac 303
MA7P SupportPac 381
MAOV transactional functionality 340
MCA 277
MDB 271
message
  digest 112, 115
    hash function 115
  flow
    server binding mode 260
  integrity 92
  persistence 70
message authentication code 115
message channel agent, *See* MCA
Message Descriptor 41, 409
message integrity
  high 93
  low 92
Message Queue Interface, *See* MQI
message-driven bean, *See* MDB
messaging
  between applications 24
  bus 376
Microsoft .NET 13, 366
  asynchronous interface 93
  client transactionality 343
  listener runtime syntax 83
  short-term asynchrony 94
Microsoft .NET languages 20
Microsoft Distributed interNet Applications (DNA) 19
Microsoft Distributed Internet Architecture 19
Microsoft DNA (Distributed interNet Applications) 19
mixed package name 81
MQI 406

N

name key 348
name=value pairs
  destination 40

InitialContextFactory 40
TargetService 40
namespace 271
network
heterogeneous 3
homogeneous 3
NoJndi 103
not recognized message 166

O

OAM 111
functionality 111
provided by utilities
dmpmqaut 111
dspmqaut 111
setmqaut 112
object authority manager, *See* OAM
object-oriented design 4
object-oriented language 4
object-oriented method 4

P

PKI 115
plain text 112–113
platform
Microsoft .NET, Apache Axis, WebSphere Application Server 16
port
1414 168, 232, 278
1420 174
preparing
WebSphere MQ environment 222, 256
programming
styles 33
programming model
peer-to-peer messaging 8
publish/subscribe 8
three-tier or n-tier 8
proxy 32–33
code 190, 245
public key infrastructure, *See* PKI

Q

queue
request, response 44
queue handle 409
queue manager connection
client mode 44
server binding mode 44
types 72
queue manager-to-queue manager connection 206
queue object 409
QueueConnectionFactory object 271

R

Rational Application Developer 290
Rational Application Developer WebSphere Software 23
Reason code
2033 265
2051 262
2085 264
2210 263
register WebSphere MQ as transport 248
registration call
Axis client 55
Microsoft .NET client 55
reliable messaging bus 1
remote component 23
Remote Method Invocation, *See* RMI
Remote Procedure Call, *See* RPC
report messages 91
Request for Comments, *See* RFC
request over HTTP 30
request queue 70
request transaction box 341
request-and-reply model
messaging pattern 395, 416
request-and-response model 213
requirements
WebSphere Application Server client 292
resource manager interface, *See* RMI
response queue 72
RFC 14
RMI 5, 23
RPC 5
runtime environment for applications 20

S

scheme name jms 15
script
amqwClientConfig.cmd/sh 56
amqwdeployWMQService.cmd/sh 59
amqwsetcp 228
amqwsetcp.cmd/sh 52, 84

defineWMQNlistener.cmd/sh 84
endWMQJListener.cmd 85
endWMQNListener.cmd/sh 85
regenDemo.cmd/sh 56
regenTranDemoAsync.cmd/sh 343
setupWMQSOAP 168
setupWMQSOAP.cmd 72, 88
setupWMQSOAP.cmd/sh 53
startWMQJListener.cmd/sh 83
startWMQNListener.cmd 83
secure communication 114, 184, 239
Java environment 184
Microsoft .NET environment 184
secure data flow 107
secure message 116
Secure Sockets Layer, *See* SSL
security 46
application layer 111
concepts 108
security services
authority 109
WebSphere Application Server clients 298
securitysslTrustStore 298
sslCipherSuite 298
sslKeyStore 298
sslKeyStorePassword 298
sslTrustStorePassword 298
WebSphere MQ clients
sslCipherSuite 286
sslKeyStore 286
sslKeyStorePassword 286
sslPeerName 286
sslTrustStore 286
sslTrustStorePassword 286
security concerns
eavesdropping 108
impersonation 108
tampering 108
unauthorized access 108
security configuration
.NET Web Service client 266
sslCipherSpec 266
sslKeyRepository 266
Axis client 210
sslCipherSuite 210
sslKeyStore 210
sslKeyStorePassword 210
sslTrustStore 210
sslTrustStorePassword 210
security considerations 110
WebSphere Application Server 285
security mechanism 110
ACL 110
cryptography 110
digital certificate 110
digital signature 110
firewall 110
PKI 110
security services 109
confidentiality 109
data integrity 109
identification and authentication 109
nonrepudiation 109
server binding mode 256
connection 233
server binding mode connection 175, 382
service code 216
service deployment 43
service method stubs 163, 189
service transaction box 341
services development 42
bottom-up development 42
top-down development 42
setup
client connection 232
client mode 258
environment 51
environment variable WMQSOAP_HOME 51
server binding mode 258
WebSphere Application Server 281
WebSphere MQ 277
signer certificate 118
simple data types 38
Simple Mail Transfer Protocol, *See* SMTP
Simple Object Access Protocol, *See* SOAP
SMTP 19
SOAP 1
definition 18
header 101
infrastructure
Microsoft .NET, Apache Axis 15
layer 35
support and tools 1
transport mechanism 19
JMS 19
UDP 19
SOAP WebSphere MQ infrastructure 216
SOAP WebSphere MQ listener 177

SOAP/HTTP 31
SOAP/JMS components 271
SOAP/WebSphere MQ listener 41, 235
SOAP/WebSphere MQ sender 41
SOAP/WebSphere MQ URI 65
SOAPAction 39, 59, 101
software installation
  WebSphere MQ transport for SOAP 142
software prerequisites
  WebSphere MQ transport for SOAP 142
    AIX 5.2 ML4 142
    IBM WebSphere Application Server V6 for AIX 142
    IBM WebSphere MQ V6 142
    Microsoft .NET Framework Redistributable V1.1 142
    Microsoft .NET Software Development Kit V1.1 142
    Microsoft Windows 2000 Professional 142
    Rational Application Developer V6 142
    Visual Studio .NET 142
software requirements
  .NET Web Service 217
  .NET Web Service client 244
  Axis client 190
  Axis Web Service 162
SSL 28, 107
  client 120
  configuration
    Java environment
      sslCipherSuite 136
      sslKeyStore 135
      sslKeyStorePassword 135
      sslTrustStore 135
      sslTrustStorePassword 136
    Microsoft .NET environment
      sslCipherSpec 135
      sslKeyRepository 135
    SSLPeerName 136
  handshake 121
  in URI 135
  protocol 107
  session 120
  WebSphere Application Server 286
start and stop scripts 83
start listener 56, 83
  configure as WebSphere MQ service 84
  manual invocation 84
  use generated start and stop scripts 83
  use WebSphere MQ trigger monitoring 84
stop listener 85
Storage Area Network 279
structured programming technique 4
symmetric key algorithm 113–114
synchronous and asynchronous forms 5
synchronous calls 246
system architecture 110
systems and applications 3

T

target request queue 44
transaction participation with SOAP over WebSphere MQ 422
transactional control 86–87, 340
  of a client request 86
  of a client response 86
  of the execution of service 86
transactional demo samples 343
transactionality
  one-phase 342
  overview 356
  two-phase 342
transmission layer security 112
trigger monitor 61
try statement 363

U

UDDI 16
UDP 19
Universal Description Discovery and Integration, *See* UDDI
Universal Resource Indicator (URI) 34, 49
Universal Resource Indicator, *See* URI
Universal Resource Indicator, *See* URI
Universal Resource Locator, *See* URL
URI 14, 34, 65
  parameter names 66
  syntax 66
URL 14, 34
user certificate 117
User Datagram Protocol, *See* UDP
using .NET Web Service with client 241
using complex objects
  Java, Microsoft .NET 80
using proxy 249
using Web Service with client 186
using Web Services Description Language 248

using WebSphere MQ Java classes 406
using XML
  in Web Service 16
    SOAP 16
    WSDL 16

V

variable initialization 361
verify installation 52
Visual Studio .NET 21

W

wait technique 98, 254
Web container 291
Web Services 8
  concepts 14
  implementation 1
  interoperability 8, 31, 37
  skeleton 272
  transaction participation 424
Web Services and WebSphere MQ clustering 373
Web Services client
  transaction participation 423
Web Services Description Language, *See* WSDL
WebSphere Application Server 22
  client 289
    security 298
  interoperation 101
  Web Services
    requirements 272
WebSphere MQ
  .NET classes 381
    MQChannelDefinition 382
    MQEnvironment 382
    MQException 382
    MQGetMessageOptions 382
    MQManagedObject 382
    MQMessage 382
    MQPutMessageOptions 382
    MQQueue 382
    MQQueueManager 382
  administration 151
  advanced features 46
  clustering 373, 375
    benefits 376
    clustered queues 376
    example 376
    features 32
  configuration 205
    steps 163
    using client binding 205
  deployment 164
  get operation 392, 414
  infrastructure 32, 44
  Java applications
    environment setup 409
  Java classes
    MQChannelDefinition 407
    MQEnvironment 407
    MQException 408
    MQGetMessageOptions 408
    MQManagedObject 408
    MQMessage 408
    MQPutMessageOptions 408
    MQQueue 408
  put operation 389, 412
  queue manager 41
  sender 377
  SOAP URI 39
  transport for SOAP 15
    error handling 90
    format 57
    long-term asynchrony 94
  transport for SOAP listener 81
  trigger monitoring 57, 82
Windows application form 251
workload choose algorithm 378
WSDL 16, 33, 227
  binding 35
  message styles 17

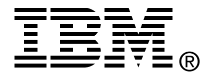
X

XML 4, 15, 30, 213
  tags 15

Redbooks

WebSphere MQ Version 6 and Web Services

(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages



WebSphere MQ Version 6 and Web Services

Interoperable Web Services using .NET, Axis, and IBM WebSphere Application Server

WebSphere MQ transport for SOAP, .NET, and Java classes

Asynchrony and transactionality

This IBM Redbook illustrates how to integrate WebSphere MQ technology in a Web Services environment. This book provides information about and examples pertaining to this topic, including the fundamental concepts, the technology, and the advanced programming features to help you succeed in your projects.

Web Services are fast becoming the platform for application integration. In fact, they are being referred to as the fundamental building blocks of Service-Oriented Architectures. Web Services expedite the move to distributed computing on the Internet or between businesses.

WebSphere MQ, the key component of IBM's Enterprise Service Bus, ensures single and reliable message delivery. Multiplatform support enables application integration on heterogeneous networks. WebSphere MQ's application programming interface, features, and adaptors include support for transactional request/reply, tiered, and publish/subscribe application models.

The use of WebSphere MQ as a transport mechanism for Web Services is enabled by the support for Apache Axis and Microsoft .NET SOAP infrastructure in WebSphere MQ V6.

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

SG24-7115-00 ISBN 0738495786